Complessità e Crittografia

Un'introduzione all'informatica teorica e alla crittografia

F. Fabris

Bozza del 27 marzo 2020

Corso di Laurea Magistrale in *Ingegneria Elettronica e Informatica*

A.A. 2018/19

Il presente materiale didattico è per uso strettamente personale

e non può essere pubblicato in rete

Indice

Capitolo 1

Introduzione storica

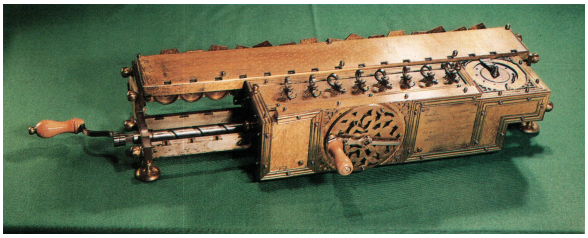
Quando si pensa alla parola *Informatica*, che deriva dalla contrazione francese di *Informa*tion Automa*tique*, l'immagine corre necessariamente al calcolatore e ai suoi accessori periferici (la tastiera, lo schermo, il *mouse* ecc.), al punto che la traduzione inglese viene resa con la locuzione *Computer Science*. Anche se è inevitabile riconoscere l'importanza del calcolatore, inteso come macchina fisica che attua gli schemi concettuali evocati dall'Informatica e che ha decretato il successo e la permeabilità delle tecnologie informatiche, è necessario prendere coscienza del fatto che l'Informatica non è *riducibile* alla macchina. Essa non solo è indipendente dalla tecnologia specifica impiegata per costruire i calcolatori (nella fattispecie la tecnologia elettronica dei semiconduttori), ma è indipendente persino dall'esistenza di una macchina fisica che la renda operativa, tant'è che i fondamenti dell'Informatica, dati dalla *Teoria della Computabilità*, furono sviluppati *prima* della costruzione materiale del primo calcolatore digitale, lo *Z1*, attuata dall'ingegnere tedesco *Konrad Zuse* tra il 1936 e il 1938.

La tecnologia informatica si sviluppa a partire dalla metà degli anni '30, in un momento felice di congiunzione tra due correnti operative e di pensiero ben distinte: da una parte c'era chi inseguiva il sogno millenario di una macchina per fare i calcoli in modo automatico (meccanica nelle prime versioni, elettromeccanica ed elettronica nelle ultime); dall'altra c'era chi si occupava dei *fondamenti logici e assiomatici della Matematica*, sognando una sorta di "meccanizzazione" della stessa, che consentisse di ricavare tutti i *teoremi* di una certa teoria matematica a partire dai suoi *assiomi* e dalle *regole di inferenza*. L'interazione tra queste due correnti di pensiero costituì il contesto fecondo attraverso il quale si passò dai sogni alla realtà.

1.1 Dalle calcolatrici meccaniche al primo computer

La storia della computazione numerica parte dagli abaci cinesi del 1200 D.C., mentre la prima realizzazione di una macchina automatica per il calcolo aritmetico viene fatta risalire a *Blaise Pascal*, filosofo, matematico e fisico francese, che nel 1643 realizzò un dispositivo meccanico per eseguire automaticamente addizioni e sottrazioni, la cosiddetta *Pascalina* (fig.1.1a). È però acclarato che già 150 anni prima *Leonardo da Vinci* aveva progettato una macchina analoga, anche se non arrivò mai a una sua costruzione. Qualche anno dopo, a partire dal 1674, il famoso filosofo e matematico tedesco *Gottfried Wilhelm Leibniz* presentò il progetto di una macchina calcolatrice a ruote e ingranaggi (le *Ruote di Leibniz*) , che era in grado di effettuare moltiplicazioni e divisioni (si veda figura 1.1b). Leibniz è però famoso soprattutto per il suo contributo fondamentale all'individuazione delle basi della Logica Simbolica ("*L'Arte Combinatoria*"), su cui si regge il funzionamento di moderni calcolatori. I successivi sviluppi in tale settore, ad opera di *George Boole*, *Alfred Whitehead*, *Bertrand Russell* e *Giuseppe Peano*, diedero consistenza al sogno di Leibniz di un ragionamento simbolico universale, con la nascita di una nuova disciplina matematica, la *Logica Simbolica*. L'idea di fondo dell'*Arte Combinatoria* è quella di trovare una logica capace non

(a) La macchina *Pascalina* per fare somme e sottrazioni

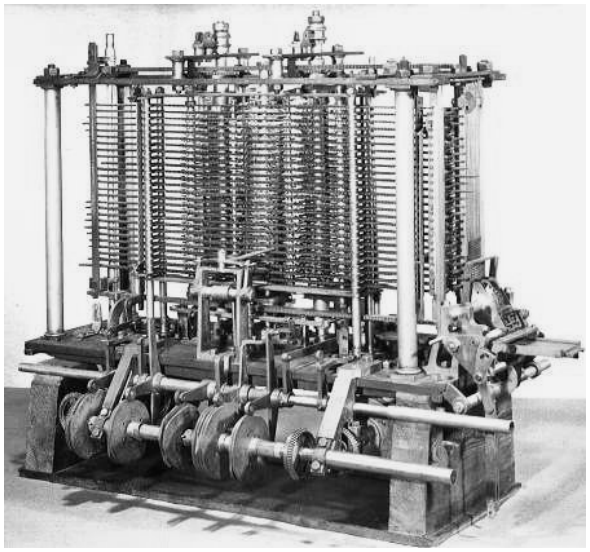


(b) Le *Ruote di Leibniz* per effettuare moltiplicazioni e divisioni

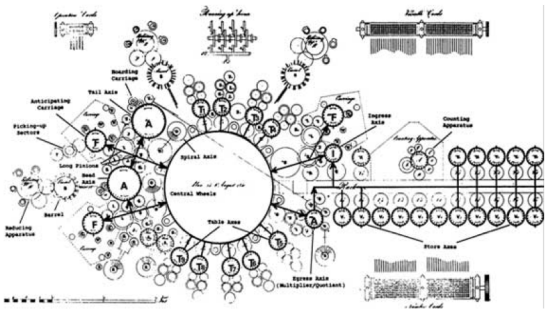
Figura 1.1: Le prime macchine calcolatrici di tipo meccanico

soltanto di dimostrare la verità di ogni proposizione vera, ma anche di costruire nuove proposizioni con la certezza dei procedimenti matematici.

Il primo modello di calcolatore così, come noi lo intendiamo oggi, che fosse cioè in grado di manipolare non solo numeri, ma anche simboli, lo si deve a *Charles Babbage*, matematico, filosofo e ingegnere britannico, il quale descrisse nel 1834 il progetto della *Macchina Analitica*, modello per tutti i successivi calcolatori digitali universali (fig.1.2). La macchina non fu mai realizzata per le difficoltà legate alla complessità meccanica delle



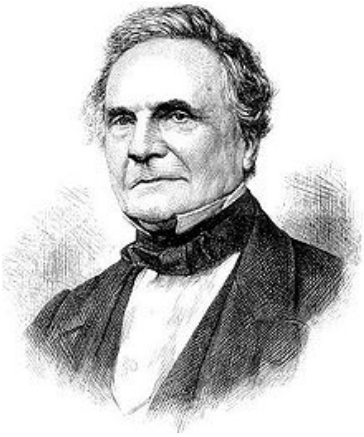
(a) Ricostruzione della *Macchina Analitica*



(b) Schema del 1840 della *Macchina Analitica*

Figura 1.2: La *Macchina Analitica* dell'ingegnere inglese *Charles Babbage*

sue 25 mila parti, anche perché i concetti sui quali avrebbe basato il suo funzionamento anticipavano di almeno cent'anni il livello tecnologico necessario alla loro attuazione pratica. Per questa macchina egli aveva infatti immaginato la possibilità di introdurre da un lato le "regole" della computazione (che noi oggi chiameremmo *algoritmi*) e dall'altro i valori da associare alle variabili e alle costanti, e tutto ciò impiegando schede o nastri perforati del tutto simili a quelli usati nei telai tessili di *Jacquard* fin dai primissimi anni dell'ottocento. I concetti che stanno alla base della *Macchina Analitica* sono gli stessi usati oggi per i moderni calcolatori elettronici. La macchina era costituita da due parti: la memoria (*Store*) che immagazzinava variabili e costanti e nella quale erano conservati anche tutti i risultati intermedi dei calcoli; l'unità di calcolo (*Mill*) che conteneva il programma vero e proprio. Lo schema generale del suo calcolatore è talmente simile a quello dei computer moderni che la tardiva riscoperta dei



(a) Charles Babbage

(b) Ada Byron contessa Lovelace

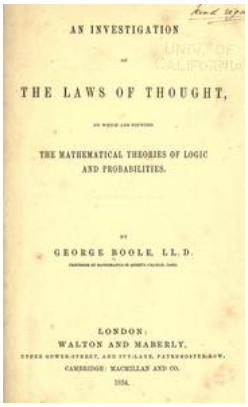
(c) Medaglia commemorativa della *Association for Computing Machinery* (ACM)

Figura 1.3: Charles Babbage e Ada Byron

suoi scritti invalidò alcuni brevetti della IBM. L'opera di Babbage venne poi esaltata da una singolare nobildonna inglese, *Ada Byron*, contessa di *Lovelace* (figlia del poeta Lord Byron), che per prima intuì l'universalità delle idee espresse da Babbage. Tra i due iniziò un fitto scambio di lettere, piene di numeri, idee, fatti e fantasie e nel 1843, in uno scritto ormai famoso, Ada Byron descrisse le possibili applicazioni della macchina nel calcolo matematico, ipotizzando persino il concetto di *Intelligenza Artificiale* e affermando che la macchina, quando realizzata, sarebbe stata cruciale per il futuro della scienza. A titolo di esempio spiegò il modo in cui la macchina avrebbe potuto effettuare il calcolo dei *numeri di Bernoulli*, e così facendo scrisse quello che viene unanimemente riconosciuto come il primo *programma per computer* della storia. In onore di Ada Byron, nel 1979 il Dipartimento della Difesa degli Stati Uniti battezzò *Ada* un linguaggio di programmazione che era stato appena realizzato.

Nel frattempo *George Boole* (fig.1.4a), logico e matematico britannico, cominciò a lavorare sullo strumento concettuale che sta alla base del funzionamento dei moderni calcolatori, cioè la logica binaria, o *Logica Booleana*, scrivendo l'opera "*An investigation of the Laws of Thought*" (fig (1.4b)). Si tratta di un calcolo logico a due valori

(a) George Boole


AN INVESTIGATION
OF
THE LAWS OF THOUGHT,
ON WHICH ARE FOUNDED
THE MATHEMATICAL THEORIES OF LOGIC
AND PROBABILITIES.
BY
GEORGE BOOLE, LL.D.
LONDON:
WALTON AND MABERLY,
CAMBRIDGE: MACMILLAN AND CO.

(b) Il frontespizio dell'opera *An Investigation on the Law of Thought*, di *George Boole*

AND			OR			NOT	
X	Y	$Z = X \cdot Y$	X	Y	$Z = X + Y$	X	$Z = \overline{X}$
0	0	0	0	0	0	0	1
0	1	0	0	1	1	1	0
1	0	0	1	0	1		
1	1	1	1	1	1		

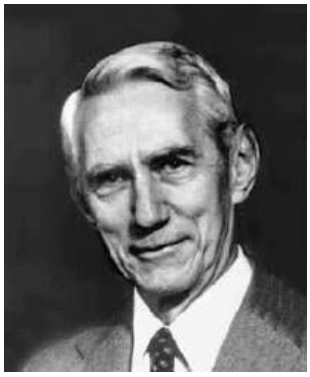
(c) Le tabelle di verità per le funzioni Booleane AND, OR e NOT

Figura 1.4: George Boole, il padre della Logica Booleana

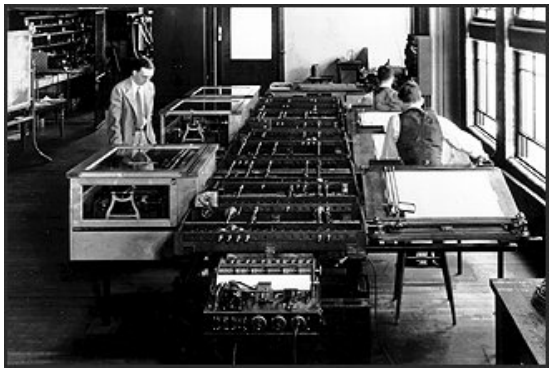
di verità, *Vero* e *Falso*, che consente di operare su proposizioni allo stesso modo in cui si opera su entità matemati-

che. Nel suo lavoro Boole mostrò che la logica Aristotelica può essere rappresentata tramite equazioni algebriche. Boole sviluppò i concetti precedentemente espressi da Leibniz sul sistema binario e descrisse gli operatori logici che da lui presero poi il nome di *Operatori Booleani* (AND, OR, NOT), oggi attuati circuitalmente mediante le cosiddette *porte logiche*.

Il lavoro di Boole fu considerato però d'interesse solo matematico-speculativo, almeno fino al 1937, anno in cui *Claude Elwood Shannon* (fig.1.5a), matematico e ingegnere americano, pubblicò la sua tesi di master intitolata *A Symbolic Analysis of Relay and Switching Circuits*. Shannon stava lavorando al MIT sotto la direzione di *Vannevar Bush*, inventore dell'*Analizzatore Differenziale* (fig.1.5b), il primo calcolatore *analogico* per risolvere equazioni differenziali (1930); in particolare egli era interessato alla teoria e alla progettazione dei complessi circuiti di relay che controllavano le operazioni della macchina di Bush.



(a) Claude Elwood Shannon



(b) L'*Analizzatore Differenziale* di *Vannaver Bush* del MIT

Figura 1.5: Claude Elwood Shannon nel laboratorio del MIT

Fu in questo contesto che si rese conto che la Logica Booleana, così come si applicava alla rappresentazione di *Vero* e *Falso*, poteva essere usata per rappresentare le funzioni degli interruttori nei circuiti elettrici, il cui funzionamento è caratterizzato da due stati, acceso e spento. Ciò divenne la base della progettazione dell'elettronica digitale, con applicazioni pratiche nella commutazione telefonica e nell'ingegneria dei computer. I meriti di Shannon vanno però ben oltre, poiché il suo nome è indissolubilmente legato ai due celeberrimi articoli "*A Mathematical Theory of Communications*" del 1948, e "*Communication Theory of Secrecy Sysytems*" del 1949, che gettarono le fondamenta della *Teoria dell'Informazione* e della *Crittografia* moderna.

(a) Archimedes

(b) Brunsviga

(c) Curta

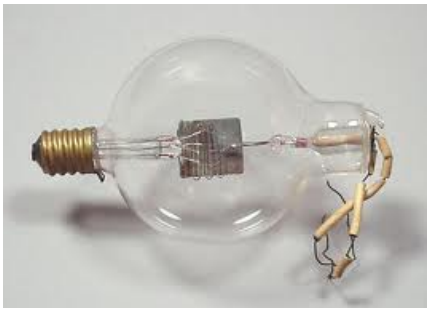
Figura 1.6: Alcuni modelli di macchina calcolatrice meccanica di fine 800, inizi 900

Alla fine dell'ottocento le calcolatrici erano ancora meccaniche, e non essendo stata ancora realizzata la Macchina Analitica prefigurata da Babbage, non esisteva alcuna macchina "programmabile". Nella figura 1.6 ve-

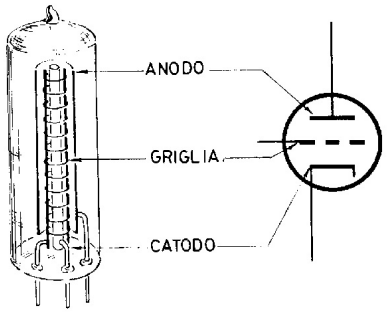
diamo alcuni esempi di calcolatrici in uso all'epoca; tra queste la *Brunsviga*, che ebbe una diffusione notevole e la *Curta*, vero e proprio gioiello nell'arte della meccanica, prodotta fino al 1943.

I primi anni del '900 furono determinanti per il trapasso tra la tecnologia elettromeccanica e quella elettronica dei *tubi termoionici*, che nasce con l'invenzione nel 1904 del *diodo a vuoto* (fig.1.7a), ad opera dell'ingegnere inglese *Sir John A. Fleming*; due anni più tardi l'americano *Lee de Forest*, aggiungendo un terzo elettrodo al diodo di Fleming, la *griglia*, crea il primo *triodo* a vuoto (fig.1.7b), che consente di amplificare un segnale analogico, ma anche di fungere da interruttore comandato in tensione (senza dispendio di potenza), sostituendo così i lenti e pesanti *relay* elettromeccanici, che necessitano per altro di una rilevante potenza per il controllo. La strada è segnata, anche se l'impatto della nuova tecnologia nell'ambito delle macchine da calcolo non sarà immediato, a causa dei problemi di affidabilità ancora presenti.

(a) Diodo a vuoto di *Fleming* (1904)



(b) Triodo a vuoto di *De Forest* (1906)

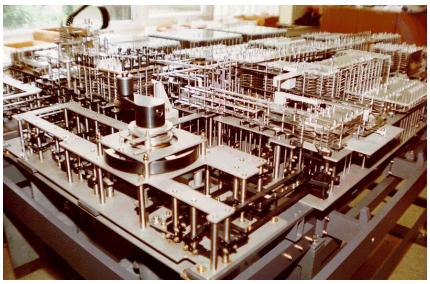


(c) Struttura schematica del triodo

Figura 1.7: I tubi termoionici inventati nei primi anni del 900

Il 1936 è l'anno in cui l'ingegnere tedesco *Konrad Zuse* (fig1.8a) inizia la costruzione del primo calcolatore moderno, la macchina logica "*V1*", successivamente ribattezzata "*Z1*" per evitare qualsiasi riferimento ai tristemente noti razzi V1 tedeschi (fig1.8b). Si tratta di un calcolatore meccanico realizzato artigianalmente e con mezzi rudimentali dallo stesso Zuse, nella propria abitazione (fig1.8c). Il prototipo rappresenta la prima macchina al mondo, basata su codice binario, completamente programmabile. Zuse, convinto che i programmi composti da combinazioni di bit potessero essere memorizzati, chiese anche un brevetto in Germania per l'esecuzione automatica di calcoli.

(a) L'ingegnere tedesco *Konrad Zuse*, che costruì il primo calcolatore nel 1936



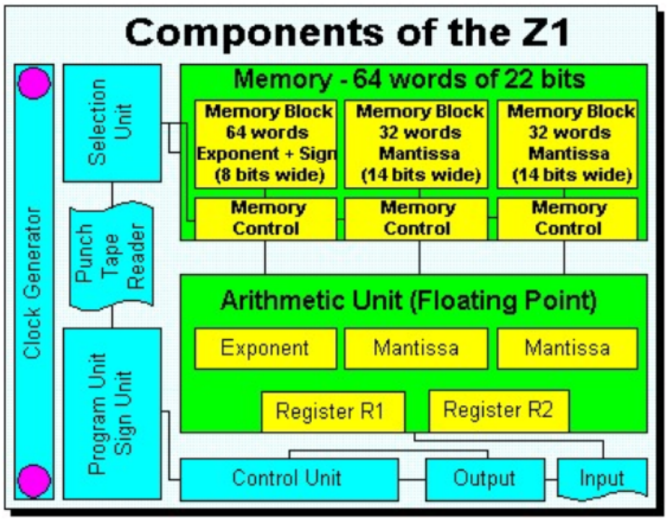
(b) Il primo calcolatore del mondo, lo Z1, del 1937

(c) Soggiorno della casa di Zuse dove venne costruito lo Z1

Figura 1.8: Lo Z1 venne distrutto subito dopo la costruzione, a seguito di un bombardamento di Berlino

Lo *Z1* era un apparecchio programmabile, in grado di processare numeri in formato binario e le cui caratteristiche più apprezzabili, viste con il senno di poi, furono la netta distinzione fra memoria e processore. Questa architettura (fig1.9a), che non venne adottata dall'*ENIAC* o dal *Mark I*, (i primi computer realizzati negli Stati Uniti

quasi dieci anni più tardi), rispecchia l'architettura del calcolatore ipotizzata solo nel 1945 da *John von Neumann*. Lo *Z1* conteneva tutti i componenti di un moderno computer, anche se era completamente meccanico, come ad esempio le unità di controllo, la memoria, la rappresentazione a virgola mobile, ecc. Aveva una frequenza di lavoro di 1 *Hertz*, era in grado di effettuare una moltiplicazione in 5 secondi, disponeva di 64 celle di memoria a 22 bit e usava al posto dei *relay* circa 20.000 piastre in metallo (fig.1.9b). Il calcolatore venne poi distrutto assieme ai progetti dai bombardamenti di Berlino, durante la seconda guerra mondiale, ma nel 1941 venne costruita la sua terza versione, denominata *Z3* (figura 1.10a), che diventerà operativo per qualche tempo, prima di essere a suo volta distrutto da un bombardamento.



(a) Architettura dello Z1

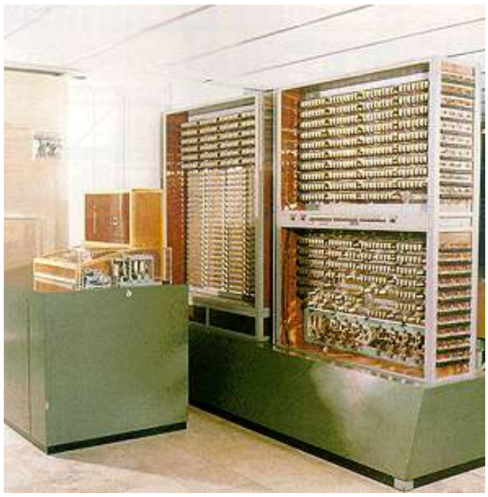
Data sheet

Name of the machine	Z1
Implementation	Thin metal plates, worked with fret saw
Frequency	1 Hertz
Numeric unit	Floating point unit, 22 Bit word length
Average calculation speed	Multiplication approx. 5 seconds
Input	Decimal keyboard, automatic binary coding
Output	Decimal digits
Word length	24 Bit mantissa, 8 bit exponent, 1 sign
Number of relays	No relays, thousands of metal plates, approx. 20,000 parts
Memory	64 cells à 22 Bit
Power consumption	Approx. 1000 watts for the electric cycle motor
Weight	Approx. 500 kg
Area of application	Experimental model, no application, developed for scientific calculations
Number of machines sold	0
Cost in DEM	No price
Comments	The Z1 was not reliable. A

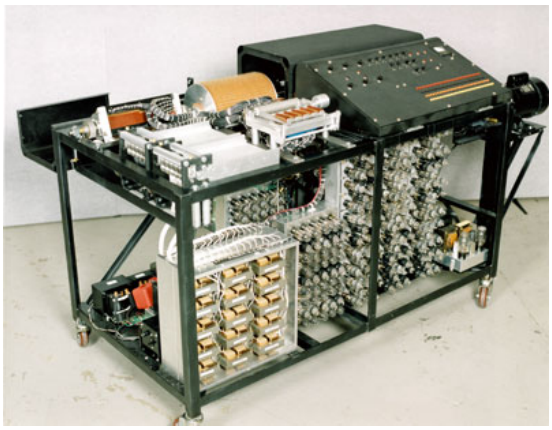
(b) Le principali caratteristiche tecniche dello Z1

Figura 1.9: Architettura e principali caratteristiche dello Z1, basato su una tecnologia puramente meccanica

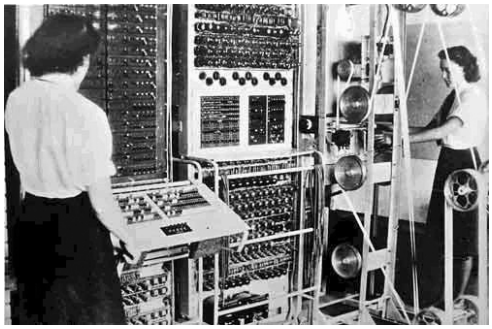
Negli Stati Uniti inizia nel 1939 il progetto dell'*Automatic Sequence Controlled Calculator* (*ASCC*) della IBM, che in seguito verrà ceduto all'università di Harvard e prenderà il nome di *Mark I* (fig.1.10d). Quasi contemporaneamente parte anche il progetto del calcolatore "*ABC*" di *J.V. Atanasov* e *C. Berry* (fig.1.10b), sul quale si sarebbe basato successivamente *J.W. Mauchly* per la costruzione dell'*ENIAC* (fig.1.10e). L'ABC è il primo computer che utilizza la nuova tecnologia dei *tubi termoionici* (o a vuoto). Il prototipo, che realizza somme a 16-bit, non arriverà mai in produzione, ma i concetti contenuti nell'*ABC*, come l'*Unità Aritmetico Logica* (ALU) e la memoria riscrivibile, compariranno nei moderni computer. Negli ultimi anni ci sono state molte controversie su chi avesse veramente inventato il primo computer elettronico digitale, e una corte di giustizia americana decise in favore di Atanasov. Nel Regno Unito si costruisce invece, nel 1943, il *Colossus* (fig.1.10c), progettato per poter forzare i cifrari tedeschi basati sull'impiego della macchina cifrante di Lorenz SZ40/42, in uso al comando generale del III Reich. L'ultimo calcolatore d'interesse storico che menzioniamo è l'EDVAC (fig.1.10f), che fu progettato all'inizio da John von Neumann, il matematico e ingegnere ungherese cui si deve l'odierna architettura dei computer, basata su un *Unità Logico-Aritmetica* (ALU), su dei *registri* di memoria e su una *memoria RAM* per memorizzare i dati e il programma. Diversamente dal suo predecessore decimale ENIAC, l'EDVAC era binario.



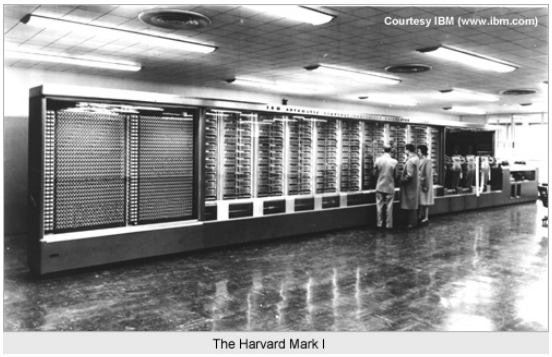
(a) Il calcolatore Z3 di Zuse (Maggio 1941), con un'architettura simile a quella di von Neumann, era basato sulla tecnologia elettromeccanica dei relè



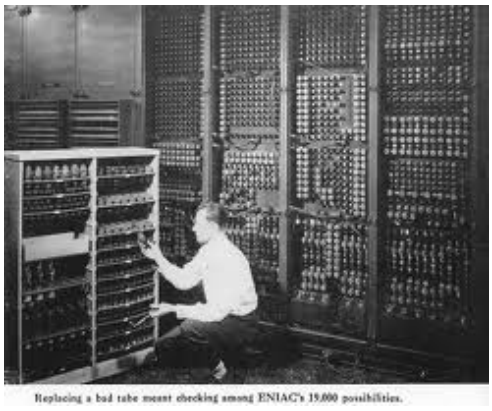
(b) Il primo calcolatore a tubi termoionici, l'ABC di *Atanasov* e *Berry* (Estate 1941)



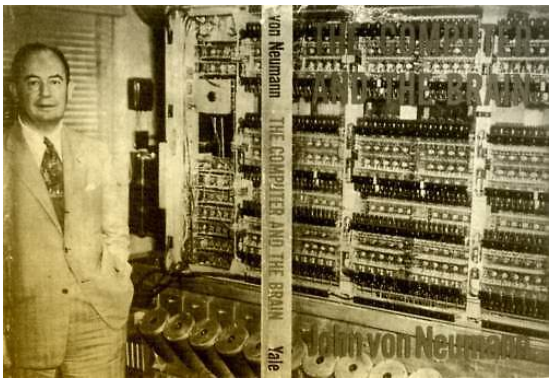
(c) Il calcolatore *Colossus*, prodotto in Inghilterra nel 1943 e usato per la decifrazione della macchina cifrante it Lorenz, in uso al quartier generale di Hitler



(d) Il *Mark I* fu il completamento del progetto *ASCC* realizzato dall'università di Harvard (1944), ma era ancora basato sulla tecnologia elettromeccanica dei relè



(e) Il calcolatore *ENIAC*, basato sulla tecnologia a tubi termoionici dell'ABC (1946-48)



(f) L'EDVAC fu uno dei primi calcolatori con l'architettura ideata da *von Neumann* (1951)

Figura 1.10: I calcolatori del periodo 1941-1951, realizzati da Germania, Regno Unito e USA

1.2 Dal programma di Hilbert ai teoremi di incompletezza di Gödel

L'ideazione e la realizzazione delle prime macchine calcolatrici, secondo il processo storico delineato brevemente nel paragrafo precedente, ebbe come spinta propulsiva la necessità di effettuare in modo automatico le quattro operazioni elementari con i numeri (somme, moltiplicazioni, differenze e divisioni). Tuttavia la complessità strutturale via via crescente di tali macchine, che ebbero come capostipite la *Macchina Analitica* di Babbage, trasformò completamente la loro natura: infatti il "programma" di calcolo, inizialmente incarnato negli ingranaggi meccanici o nei circuiti elettromeccanici delle macchine più avanzate, e deputato alla soluzione di un problema specifico, lascia a un certo punto il posto a un "programma" non cablato, che può essere modificato dall'esterno con lo scopo di poter risolvere un problema nuovo, e ciò senza dover riassemblare la macchina. La macchina acquista dunque una flessibilità che le consente di essere usata più volte per risolvere problemi di natura diversa, e ciò senza dover cambiare la sua topologia. Diventa cioè una *macchina programmabile*. All'inizio questi problemi erano legati soprattutto al calcolo di fattori numerici, ma il potere della *codifica simbolica*, ossia la libertà di attribuire un qualunque significato a un simbolo, coniugato con la possibilità di manipolare i simboli in modo logicamente strutturato, portò a disvelare potenzialità inizialmente insospettabili per le macchine sia pur rudimentali dell'epoca.

È a questo punto che il destino di coloro che inseguivano il sogno di una macchina automatica per fare i calcoli s'intreccia con quello di coloro che miravano a una ricostruzione logica e unitaria di tutta quanta la Matematica, che avrebbe dovuto consentire di ricavarne i teoremi in qualsiasi ambito (analisi, geometria, algebra, ecc.) a partire dagli *assiomi* e dalle *regole di inferenza*, secondo un approccio che si inquadra perfettamente col pensiero razionalista e riduzionista di inizio secolo.

Ricordiamo a tal proposito che ogni *Teoria Matematica*, quale ad esempio l'*Aritmetica*, la *Teoria degli Insiemi* ecc., scaturisce da alcune affermazioni iniziali considerate vere e denominate *assiomi* (o *postulati*), e dall'applicazione ad essi di specifiche *regole di inferenza*, che esprimono le modalità lecite per costruire altre affermazioni vere, denominate *teoremi*. In quest'ottica la *Teoria* è l'insieme di tutti gli assiomi, che giocano il ruolo di verità primitive, e di tutti i teoremi che si possono provare usando le regole di inferenza. Ecco ad esempio i celebri *postulati di Peano*, sui quali si fonda l'Aritmetica dei numeri naturali:

Postulati di Peano

1. "0" è un numero;
2. se n è un numero, il suo successore è un numero;
3. "0" non è successore di alcun numero;
4. numeri diversi non possono avere lo stesso successore;
5. se un insieme S di numeri comprende lo 0, come anche il successore di qualunque numero in S, allora S comprende tutti i numeri.

Per quanto riguarda le regole di inferenza, possiamo dire che esse costituiscono i cardini logici del ragionamento matematico; alcuni esempi sono la *Modus Ponens*, la *Modus Tollens* e la *Reductio ad Absurdum*, illustrate sinteticamente dalla tabella sotto riportata. La barra indica che a partire dalle premesse che stanno sopra, si trae la conseguenza che sta sotto:

Modus Ponens	$\frac{A \to B,\ A}{B}$
Modus Tollens	$\frac{A \to B,\ nonB}{nonA}$
Reductio ad Absurdum	$\frac{A \to B,\ A \to nonB}{nonA}$

Il *Modus Ponens* si legge così: se da A segue B, e A è vero, allora è vero anche B; analogamente le altre.

Il rappresentante sommo dell'impostazione riduzionista prima citata fu il grande matematico tedesco *David Hilbert* (1862–1943, fig1.11a). Al *Secondo Congresso Internazionale di Matematica* di Parigi del 1900, Hilbert tenne un intervento di portata storica - probabilmente la più influente conferenza matematica di ogni tempo - proponendo un elenco di 23 problemi aperti che, a suo giudizio, costituivano una sfida per i matematici del secolo a venire. La tabella di figura 1.12 riporta l'elenco completo. La natura di questi problemi era varia e disomogenea: se

(a) David Hilbert

(b) Kurt Gödel

(c) Alan Turing

Figura 1.11: La formulazione dei 23 problemi di Hilbert consentì a Gödel e a Turing di sviluppare le riflessioni che portarono alla fine alla formulazione del primo modello di computazione nel 1936

alcuni erano molto specifici e tecnicamente ben delineati (p.es. il Problema 3, che venne risolto immediatamente), altri (p.es. il Problema 6, sull'assiomatizzazione della Fisica, o il Problema 4) erano troppo generali o troppo vaghi per ammettere una risposta incontrovertibile. Altri ancora, i Problemi 1, 2 e 10, portarono a una soluzione inaspettata per Hilbert: essi ci riguardano da vicino, per la loro stretta connessione con i fondamenti della *Teoria della Computabilità*, e quindi con un inquadramento formale dei fondamenti dell'Informatica. Data la loro importanza li esaminiamo a parte.

1^o Problema - Ipotesi del continuo (IC)

Non esiste una cardinalità intermedia tra quella dei naturali e quella dei reali.

Si tratta di accertare se esista un insieme S (infinito) dotato di cardinalità inferiore a quella dei reali e superiore a quella dei naturali. Nel 1938 *Kurt Gödel* stabilì che IC non è refutabile nell'ambito assiomatico della teoria (di Zermelo-Fraenkel) degli insiemi; nel 1963, *Paul Cohen* stabilì che in tale ambito non è neppure dimostrabile.

2^o Problema - Assiomi dell'aritmetica

Accertare che gli assiomi dell'aritmetica sono consistenti.

Gödel dimostrò (1931, *1^o Teorema di Incompletezza*) che in qualunque sistema assiomatico sufficientemente espressivo da contenere almeno l'aritmetica, si può individuare un enunciato circa i numeri naturali che:

non può essere nè dimostrato nè refutato all'interno del sistema (sistema *incompleto*);
oppure
può venire sia provato che refutato all'interno del sistema (sistema *inconsistente*).

Problema	1	risolto (1963)	*L'Ipotesi del Continuo*
Problema	2	risolto (1930)	*Consistenza degli assiomi dell'aritmetica*
Problema	3	risolto	Uguaglianza di volumi tra tetraedri
Problema	4	troppo vago	Costruzione di tutte le metriche con linee geodetiche
Problema	5	risolto	Differenziabilità dei gruppi continui di trasformazioni
Problema	6	aperto	Assiomatizzazione della Fisica
Problema	7	risolto	Trascendenza di a^b, con $a \neq 0, 1$ e b irrazionale
Problema	8	aperto	Ipotesi di Riemann e congettura di Goldbach
Problema	9	parzialm. risolto	Trovare la legge più generale di reciprocità in un qualunque campo algebrico numerico
Problema	10	risolto (1970)	*Risolubilità delle equazioni Diofantee*
Problema	11	parzialm. risolto	Forme quadratiche con coefficienti numerici algebrici
Problema	12	aperto	Estensioni di campi numerici algebrici
Problema	13	risolto	Risoluzione di equazioni di 7-imo grado usando funzioni di due argomenti
Problema	14	risolto	Dimostrazione di finitezza di certi sistemi completi di funzioni
Problema	15	parzialm. risolto	Fondamenti del calcolo enumerativo di Schubert
Problema	16	aperto	Topologia di curve e superfici algebriche
Problema	17	risolto	Espressione di funzioni razionali definite come quozienti di somme di quadrati
Problema	18	risolto	Riempimento spaziale tramite poliedri non regolari
Problema	19	risolto	Analiticità delle soluzioni di Lagrangiani
Problema	20	risolto	Risolvibilità di ogni problema variazionale fissate certe condizioni al contorno
Problema	21	risolto	Esistenza di equazioni differenziali lineari aventi un gruppo monodromico assegnato
Problema	22	risolto	Uniformizzazione di relazioni analitiche per mezzo di funzioni automorfiche
Problema	23	risolto	Sviluppi ulteriori del calcolo variazionale

Figura 1.12: I 23 Problemi di Hilbert

In altre parole *ogni sistema assiomatico sufficientemente espressivo è o inconsistente o incompleto*. Se escludiamo la prima eventualità, possiamo esprimere il teorema più semplicemente, dicendo che non è detto che un enunciato vero sia un teorema, e cioè che discenda dagli assiomi tramite le regole di inferenza del sistema.
Da qui Gödel dedusse (*2^o Teorema di Incompletezza*) che quando un sistema assiomatico è consistente e sufficientemente espressivo da contenere almeno l'aritmetica, esso *non può* provare la propria consistenza. Ciò fornisce una risposta, per quanto negativa e imprevista, al 2^o problema di Hilbert.

10^o Problema - Risolubilità delle equazioni Diofantee.

Trovare una procedura in grado di stabilire se una qualunque equazione Diofantea assegnata ammette soluzioni intere.

Un'equazione Diofantea è un'equazione polinomiale $p(x_1, x_2, ..., x_n) = 0$ a coefficienti interi, che s'intende risolvere assegnando valori interi alle incognite x_i. *Yuri Matiyasevich* dimostrò, nel 1970, che una procedura risolutiva generale non può esistere: a meno che non si pongano fortissime limitazioni al numero di incognite o al grado del polinomio; siamo dunque di fronte a un ulteriore importante problema della matematica che viene risolto in senso tanto negativo quanto inaspettato.

È evidente che l'impostazione riduzionista di Hilbert, implicita per altro già negli enunciati dei problemi (nei quali si chiede di trovare *la* soluzione, e non *se* una certa soluzione esista o meno), subì un duro e inaspettato contraccolpo dall'enunciazione dei *Teoremi di Incompletezza* di Gödel, che rimangono sicuramente una delle più importanti scoperte matematiche del '900. Essi gettarono lo scompiglio tra le fila dei matematici dell'epoca, poiché l'idea che qualcosa di matematicamente vero potesse non esser dimostrabile implicava un ridimensionamento essenziale, anche se circoscritto a singoli problemi, nella capacità argomentativa del metodo matematico basato sull'approccio *ipotetico-deduttivo*. D'altra parte il programma riduzionista, chiaramente perseguito anche dai matematici *Gottlob Frege* e *Giuseppe Peano*, aveva già subito qualche incrinatura, soprattutto ad opera di *Bertrand Russell* e in singolare contemporaneità con l'elencazione dei 23 problemi di Hilbert. Il suo famoso *paradosso* aveva destabilizzato l'opera di *Frege*, aprendo un periodo di crisi dei fondamenti logici della matematica. Tale paradosso riguarda *gli insiemi che non sono membri di sé stessi*. A prima vista la loro stessa definizione potrebbe sembrare mal posta; e in effetti, se prendiamo come riferimento un insieme di numeri, esso non è un numero, per cui sembra privo di senso chiedersi se appartenga o meno a sé stesso. Tuttavia, tanto per fare un esempio, l'insieme degli argomenti trattati in questo paragrafo è esso stesso un argomento (ne stiamo parlando ora!) e dunque è un insieme che appartiene a sé stesso.

L'antinomia di Russell

Se chiamiamo T l'insieme di tutti gli insiemi che non appartengono a sé stessi si ha:

se $T \in T$ allora $T \notin T$, poiché T contiene per definizione solo insiemi che *non appartengono a sé stessi*

se $T \notin T$ allora $T \in T$, poiché T contiene per definizione tutti gli insiemi che *non appartengono a sé stessi*

Il paradosso aveva famosi precedenti storici, quali il *Paradosso del mentitore*, attribuito ad *Eubulide di Mileto* (filosofo greco del IV secolo A.C.): *Un uomo dice che sta mentendo. Sta dicendo il vero o il falso?* Di questo paradosso è nota anche la variante *Questa frase è falsa*, e una sua versione precedente, attribuita ad *Epimenide*, cretese: *I cretesi son tutti bugiardi*, che non sembra però essere stata scritta con l'intento di illustrare un paradosso. Tuttavia il *Paradosso del mentitore* è una contraddizione logica che gioca sull'autoreferenzialità in un contesto, come quello linguistico, che non è formalizzato matematicamente; infatti la spiegazione più semplice consiste nell'assumere che ogni frase pronunciata (o scritta) esprima implicitamente un'affermazione di verità sull'oggetto della frase stessa, per cui la frase *Questa frase è falsa* andrebbe letta in realtà come *Questa frase è vera e questa*

frase è falsa, il che corrisponde all'enunciazione di una semplice contraddizione del tipo A *e non* A, che è falsa. Nel caso del paradosso di Russell le cose erano invece molto più compromesse: il suo argomento evidenziava che una teoria matematica proposta come fondamentale, la *Teoria Elementare degli Insiemi* di *Cantor* nell'assetto formale raggiunto a fine '800, era minata da irriducibili contraddizioni interne.

Nel 1908 *Ernst Zermelo* riuscirà a sanare l'antinomia di Russell, impostando un nuovo sistema noto oggi come *Teoria assiomatica degli Insiemi di Zermelo-Fraenkel*; ma con l'effetto destabilizzante causato dai teoremi di Gödel, il programma Hilbertiano, teso alla riorganizzazione di tutta la matematica in un edificio formale che avrebbe dovuto autocertificare la propria consistenza, dovrà venire definitivamente archiviato.

1.3 Nascita dell'Informatica e principali modelli di computazione

Sul solco delle riflessioni inerenti gli aspetti logico-fondazionali della Matematica sopra evocati, si sviluppò quella corrente di pensiero che riuscì in seguito a delineare il nucleo fondante dell'Informatica, cioè la *Teoria della Computabilità*, intesa come studio, modellizzazione e individuazione dei limiti relativi all'approccio computazionale basato sulle *procedure effettive* o *algoritmi*. Di nuovo lo spunto iniziale partì da Hilbert, che nel 1928 scrisse, con W. Ackermann, il libro "*Grundzüge der theoretischen Logik*"; in quest'opera compare per la prima volta l'enunciazione del famoso *Entscheidungsproblem* (Problema della decisione) per la *Logica dei predicati (del Primo Ordine)*, cioè per il sistema formale che incorpora la logica classica basata sugli operatori *and* ($\wedge$), *or* ($\vee$), *not* ($\neg$), *implica* ($\Longrightarrow$), *per ogni* ($\forall$), *esiste* ($\exists$). Per capire il significato del *Entscheidungsproblem*, ricordiamo che in tale sistema formale si possono formare delle formule, le cosiddette *formule ben formate*, come per esempio

$$(\exists F)\{(F(a) = b) \wedge (\forall x)[p(x) \implies (F(x) = g(x, F(f(x))))]\}$$

che si legge come "esiste una funzione F tale che $F(a) = b$ e tale che $\forall x$, se è vero il predicato $p(x)$, allora $F(x) = g(x, F(f(x)))$". Ciascuna formula è suscettibile di una *interpretazione*, che consiste nell'assegnazione delle funzioni, delle variabili e delle costanti. Per esempio, assegnando $f(x) = x - 1$ e $g(x) = xy$ sui numeri naturali, l'interpretazione diventa:

Interpretazione 1: $f(x) = x - 1$ e $g(x, y) = xy$

$$(\exists F)\{(F(0) = 1) \wedge (\forall x)[x > 0 \implies (F(x) = xF(x - 1)]\}$$

che si legge come "esiste una funzione F tale che $F(0) = 1$ e tale che $\forall x$, se $x > 0$ allora $F(x) = xF(x - 1)$; tale interpretazione è vera, poiché corrisponde alla funzione fattoriale. Viceversa, l'interpretazione seguente

Interpretazione 2: $f(x) = x$ e $g(x, y) = y + 1$

$$(\exists F)\{(F(0) = 1) \wedge (\forall x)[x > 0 \implies (F(x) = F(x) + 1]\}$$

risulta evidentemente falsa. Una formula si dice allora *valida* se è vera in tutte le interpretazioni. L'oggetto del *Entscheidungsproblem* riguarda proprio la validità delle formule nella logica dei predicati.

Entscheidungsproblem

Trovare una procedura effettiva (algoritmica) per decidere se una qualunque formula nella logica dei predicati è valida (p.es. se una qualunque formula dell'aritmetica è un teorema, cioè derivabile dagli assiomi mediante le regole di inferenza).

Il problema fu risolto indipendentemente da *Alonzo Church*, che pubblicò nel 1936 un articolo intitolato "*An Unsolvable Problem in Elementary Number Theory*", e da *Alan Turing* (fig 1.11c), che nello stesso anno pubblicò

l'articolo "*On Computable Numbers, with an Application to the Entscheidungsproblem*". Essi dimostrarono, con argomentazioni molto diverse, *la non esistenza di un siffatto algoritmo*. Pertanto, in particolare, è impossibile decidere algoritmicamente se un qualunque enunciato sui numeri naturali è vero o meno. Il lavoro di Church fu l'atto di nascita di un formalismo matematico, denominato λ-*calcolo*, che costituisce un vero e proprio *modello di computazione*. Tuttavia l'approccio di Turing, basato su un semplice dispositivo chiamato *macchina di Turing* (MdT), e che oggi riconosciamo come la descrizione del primo modello formale di calcolatore, risultò subito molto più convincente e credibile, al punto che Gödel stesso rimase inizialmente dubbioso sulla correttezza del λ-*calcolo*, ma immediatamente convinto dal modello di Turing. La *Macchina di Turing* incarna implicitamente la prima definizione del concetto di *algoritmo*, al punto che oggi la stretta corrispondenza tra ciò che si considera intuitivamente calcolabile mediante una procedura effettiva di tipo algoritmico e la *Macchina di Turing* costituisce il nucleo forte della cosidetta *Tesi di Church-Turing*. Turing risolse (negativamente, come accennato sopra) l'*Entscheidungsproblem* facendo riferimento al problema della *fermata della macchina di Turing*, e dimostrando che, assegnata una qualunque MdT, non è possibile decidere algoritmicamente se essa si fermerà o meno a partire da certe condizioni iniziali. Il successivo concetto di *macchina di Turing Universale*, cioè di una macchina che sia in grado di simulare la computazione di qualunque altra macchina, getta poi le basi teoriche del calcolatore programmabile.
I due modelli di computazione sopra citati non sono gli unici che furono sviluppati; Gödel stesso contribuì, fin dall'inizio, a definire due approcci alternativi alla computazione e qualche anno più tardi furono presentati altri modelli da parte di Post e Markov; la tabella di figura 1.13 ci mostra una rassegna dei principali modelli di computazione finora introdotti. Vale al pena osservare che tutti i modelli citati, ideati per caratterizzare il concetto

Gödel-Herbrand-Kleene	(1936)	calcolo equazionale	$\mathcal{H}$
Church	(1936)	λ calcolo	$\mathcal{L}$
Gödel-Kleene	(1936)	funzioni μ ricorsive	$\mathcal{G}$
Turing	(1936)	macchina di Turing	$\mathcal{T}$
Post	(1943)	sistemi di deduzione	$\mathcal{P}$
Markov	(1951)	riscritture di stringhe	$\mathcal{S}$
Shepherdson-Sturgis	(1963)	modello URM (RAM)	$\mathcal{C}$

Figura 1.13: Principali modelli di computazione

di computazione effettiva, partono da idee e presupposti matematici completamente diversi l'uno dall'altro, pur pervenendo alla fine tutti allo stesso insieme di *funzioni computabili*, che è la vera misura dell'espressività di un modello (si veda la figura 1.14). Ci sono in particolare dei teoremi di equivalenza che stabiliscono la doppia inclusione tra gli insiemi di funzioni computabili in un modello e nell'altro; p.es. se chiamiamo, rispettivamente, $\mathcal{L}$ l'insieme delle funzioni calcolabili secondo il modello del λ-calcolo di Church e $\mathcal{T}$ l'insieme delle funzioni calcolabili secondo il modello di Turing, si può dimostrare che $\mathcal{T} \subseteq \mathcal{L}$ e che $\mathcal{T} \supseteq \mathcal{L}$, e dunque $\mathcal{T} \equiv \mathcal{L}$. Questi teoremi di equivalenza sono stati sviluppati avendo Turing come modello comune di riferimento, per cui essi sono tutti nella forma $\mathcal{T} \equiv \ldots$.

Si ricava dunque

$$\mathcal{H} \equiv \mathcal{L} \equiv \mathcal{G} \equiv \mathcal{T} \equiv \mathcal{P} \equiv \mathcal{S} \equiv \mathcal{C} \equiv \mathfrak{C} \tag{1.1}$$

dove con $\mathfrak{C}$ abbiamo indicato l'insieme delle funzioni calcolabili, senza ulteriore specificazione. Se osserviamo la cronologia dei modelli nella tabella 1.13 ci rendiamo conto che il modello di Shepherdson-Sturgis fu introdotto molti anni dopo gli studi pionieristici di Turing, Church, Gödel e altri. Ciò è dovuto alla circostanza che negli anni '60 già si disponeva di un'architettura ben consolidata per i computer (quella di von Neumann, che ci accompagna ancor'oggi), ma paradossalmente nessuno dei modelli di computazione introdotti fino a quel punto aveva un qualche collegamento con tale architettura, rendendo il modello stesso una idealizzazione della computazione troppo distante dalla realtà architetturale che intendeva esprimere. Venne così introdotto il modello URM di Shepherdson-Sturgis (*Unlimited Register Machine*), che rappresenta l'essenza architetturale di un computer moderno, modello

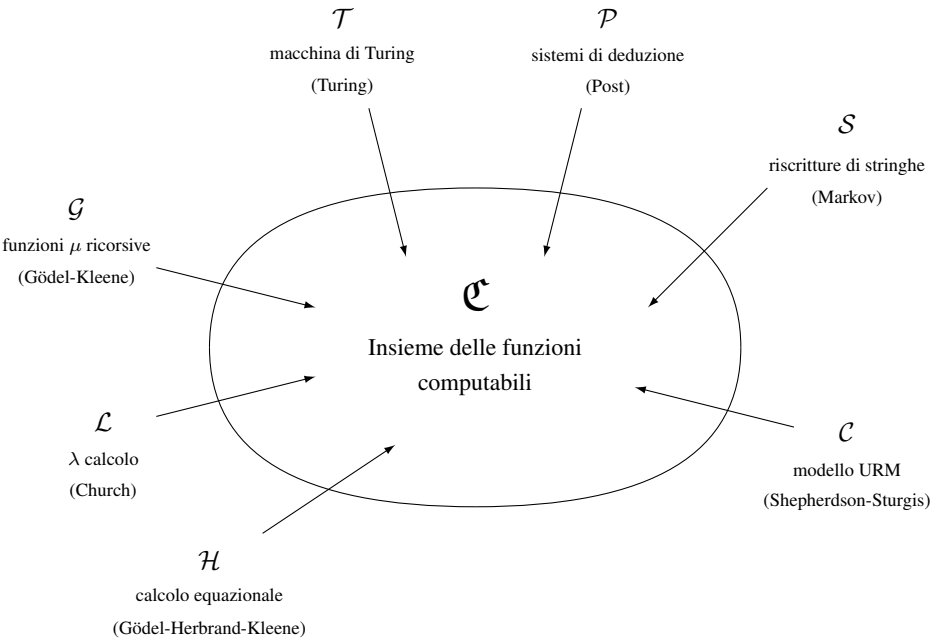


Figura 1.14: L'insieme delle funzioni computabili secondo i vari modelli è unico

dotato di una memoria di lunghezza infinita sulla quale si può scrivere usando delle istruzioni elementari di tipo assembly, organizzate in un programma. Studieremo nel dettaglio questo modello, che sarà per noi il modello di riferimento, e che chiameremo confidenzialmente anche *modello RAM*, visto che è espressione di tale architettura.

Il fatto così tanti modelli di computazione tra loro indipendenti, e che fanno riferimento ad approcci completamente diversi all'idea di computazione, puntino tutti esattamente allo stesso insieme di funzioni è un risultato molto forte che ci consente di introdurre la cosiddetta

Tesi di Church-Turing

L'insieme $\mathfrak{C}$ delle funzioni intuitivamente ed effettivamente computabili coincide con la classe $\mathcal{C}$ delle funzioni URM-calcolabili

In altre parole l'idea intuitiva ed empirica che abbiamo della computazione viene esaurita dalla capacità computazionale del modello URM (e dai modelli a esso equivalenti): tutto cò che si può fare con un computer si può fare anche con la macchina URM; nessuno ha mai mostrato un esempio di una funzione effettivamente calcolabile, ma che non lo sia nel modello URM (o nel modello di Turing o in ogni altro modello).
Si osservi che la Tesi di Church-Turing non è e non può essere un teorema; essa è giustificata dal fatto che partendo da diversi approcci alla computazione si giunge sempre allo stesso insieme di funzioni calcolabili, dati i teoremi di equivalenza tra le classi. Come conseguenza si ha che l'accettazione della tesi di Church-Turing offre una nuova tecnica di dimostrazione della computabilità di una funzione, denominata *dimostrazione mediante Tesi di Church-Turing*: se siamo in grado di fornire un algoritmo informale (che deve essere però rigoroso e non ambiguo) per illustrare il computo di una certa funzione, invocando la Tesi di Church-Turing si conclude immediatamente che la funzione è URM-computabile.

Concludiamo osservando che l'impetuoso sviluppo dell'informatica e il suo rapido affermarsi come disciplina a sé non si può dunque ricondurre solamente alla riflessione millenaria sui limiti del procedimento ipotetico-

deduttivo o all'incontro fra questa componente del pensiero matematico e la tecnologia elettronica: la nuova disciplina si sviluppò anche a partire da nuove idee fondanti, la più importante delle quali è proprio quella di *macchina programmabile*, per l'espletamento dei piú diversi compiti senza modifiche della sua architettura fisica. Il sogno di Leibniz di una logica universale, il fervore progettuale di Babbage e i lavori fondamentali di Turing e Church, hanno instradato il pensiero scientifico verso la conquista di un concetto esplicito di *computabilità*, che ha affiancato la realizzazione dei primi calcolatori. Altra idea-chiave, ben manifesta nei progetti di Turing e Zuse, è che l'autoreferenzialità - tradizionale fonte di intriganti paradossi - può essere sfruttata anche in senso positivo. Non c'è una ragione per cui i programmi debbano risiedere all'esterno del calcolatore (come avveniva per i nastri perforati rispetto ai telai Jacquard); un programma caricato in memoria, viceversa, potendo non solo indirizzare le azioni del computer, ma anche subire modifiche per effetto delle sue elaborazioni, avrebbe la duplicità di ruolo necessaria all'apprendimento automatico e alla gestione delle altre tematiche proprie dell'*Intelligenza Artificiale*. Mentre questa seconda idea tarda a dispiegare tutte le potenzialità presenti nella visione di Turing, l'obiettivo di *universalità* del calcolatore può dirsi largamente raggiunto: in effetti, un modesto *laptop* del giorno d'oggi surclassa di molto i colossali calcolatori realizzati da pionieri dell'informatica quali Zuse e von Neumann, che peraltro suscitavano grandi entusiasmi in chi aveva la consapevolezza delle ambizioni che tali prototipi incarnavano. Sarebbe un vero peccato se proprio oggi, mentre si fa un gran parlare di "informatica pervasiva" (o "*ubiquitous computing*") in quanto aspetti tecnologici particolari dell'informatica sono migrati all'interno di palmari, di dispositivi legati alla casa, all'auto, ecc., l'idea originaria venisse persa di vista a favore di logiche proprietarie e di mercato tendenti a riportare in auge soluzioni *ad hoc* o linguaggi programmativi di nicchia, riducendo di fatto il calcolatore alle funzionalità di una mera calcolatrice cablata, sia pure di tipo sofisticato.

Capitolo 2

Un modello di computazione per il calcolatore

Quando si studia la realtà fisica che ci circonda, seguendo l'approccio scientifico Galileiano, si tenta di condensare all'interno di un *modello* quelle che sono le evidenze sperimentali del particolare sottosistema che si sta analizzando. Un modello è una rappresentazione astratta del sistema, che ne riproduce le caratteristiche, le peculiarità e i comportamenti fondamentali. La definizione del modello va fatta solitamente ammettendo alcune ipotesi generali sul sistema e deducendo le leggi matematiche che sono alla base della descrizione del suo comportamento. Quando si dispone di un modello è possibile fare delle *previsioni* sul comportamento del sistema, sfruttando le equazioni che lo rappresentano.
Per esempio il *modello del corpo rigido*, impiegato nella *meccanica classica*, consente di descrivere il comportamento di un corpo caratterizzato dall'ipotesi che non sia deformabile. Ciò significa che, scelti due punti qualunque x_1, y_1, z_1 e x_2, y_2, z_2 appartenenti al corpo e detta d_{12} la loro distanza iniziale, il vincolo di rigidità è analiticamente espresso dalla relazione:

$$(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2 - d_{12}^2 = 0$$

che deve valere per ogni istante successivo a quello iniziale. Del corpo rigido possiamo studiare la sua *cinematica* e la sua *dinamica*, deducendo delle equazioni che ci consentono di fare delle previsioni sul comportamento del sistema. Per esempio è possibile stabilire il punto esatto in cui una sfera indeformabile di ferro, di diametro di 1 cm, cadrà quando lanciata verso l'alto con un'inclinazione di 32° rispetto al piano orizzontale e con una velocità iniziale di $3,5$ m/s.
Qualunque modello offre tuttavia una rappresentazione solo parziale della realtà, poiché tende a evidenziare le sole grandezze fisiche che ci interessano al livello di analisi in cui stiamo operando; esso è legato, in modo indissolubile, alle ipotesi che facciamo sul sistema. Se le ipotesi diventano meno vincolanti, il modello deve essere "esteso" per poter tener conto del nuovo stato di cose. Se per esempio prendiamo la nostra sferetta di ferro e ipotizziamo che si sposti a velocità molto elevate, prossime alla velocità della luce, allora il modello delle meccanica classica non è più sufficiente a descrivere il sistema poiché, per esempio, il diametro della sfera subirà una contrazione nella direzione del moto descritta dalla famosa legge di *Lorenz-FitzGerald*

$$D_v = D\sqrt{1 - (v/c)^2}$$

dove D_v è il diametro nella direzione dello spostamento a velocità v, c è la velocità della luce e D è il diametro della sfera in quiete. Il modello deve dunque essere esteso per tener conto delle equazioni della *relatività ristretta*, che offrono una descrizione più accurata del sistema. Assumendo il punto di vista della *Relatività ristretta* possiamo allora affermare che il modello della meccanica classica offre una buona rappresentazione della realtà quando le

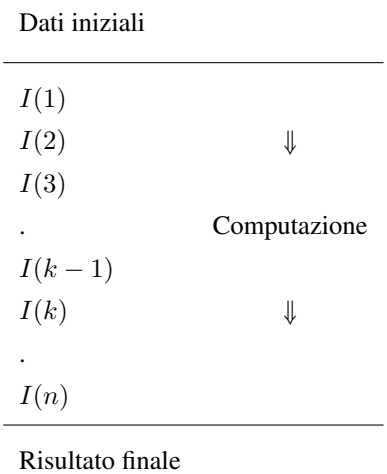
velocità in gioco sono trascurabili rispetto alla velocità della luce.
Sulla base di quanto detto finora sembra ragionevole poter costruire un *modello* anche per il calcolatore descritto nel capitolo 1, che specifichi con precisione le ipotesi che riguardano il funzionamento del sistema, in modo che si possano fare delle *previsioni* sul suo comportamento; per esempio sarebbe interessante individuare quali problemi si possono effettivamente risolvere col metodo algoritmico-procedurale anticipato nel paragrafo 2.1 e quali no, oppure conoscere i tempi necessari per ottenere una soluzione di un problema risolubile. In questo capitolo ci occuperemo proprio della costruzione di un modello di computazione per il calcolatore. Come vedremo di modelli ce ne sono tanti, ma si potrebbe dimostrare che essi sono tra loro tutti *equivalenti*, nel senso che portano tutti allo stesso insieme di problemi risolubili o, per esprimersi in modo un po' più tecnico, allo stesso insieme di *funzioni computabili*.
Il modello che sceglieremo è il *modello RAM*, che al contrario degli altri ha il pregio di essere basato sulla struttura architetturale dei computer. Grazie al modello RAM saremo in grado di capire la vera natura della computazione, quali sono i problemi effettivamente risolubili da un calcolatore e quali non lo sono e quali sono i tempi necessari alla risoluzione, in funzione della dimensione del problema.

2.1 Il metodo procedurale algoritmico

Come già ricordato il termine informatica deriva da *informatique*, contrazione delle parole francesi *information automatique*. Esse sottintendono l'idea di una elaborazione *automatica* delle informazioni, dove l'automatismo è in qualche misura caratterizzato da un aspetto procedurale della successione di operazioni elementari che vengono applicate all'informazione da elaborare con lo scopo di risolvere un problema del mondo reale. Un tale procedimento parte dunque da un insieme di informazioni fornite dall'esterno al sistema di elaborazione, i cosiddetti *dati iniziali*, che vengono successivamente elaborati secondo una procedura ordinata e a passi, descrivibile in modo preciso ed esauriente come successione finita $I(1), I(2), \ldots, I(n)$ di n *istruzioni elementari*. Il risultato dell'esecuzione dell'istruzione $I(k)$ dipende, oltre che dal tipo di istruzione, anche dai dati d'ingresso allo stesso passo, che sono costituiti tanto da eventuali dati esterni resi disponibili al passo k, quanto dai dati che derivano dall'esecuzione dell'istruzione $I(k-1)$ al passo precedente. E' evidente che lo svolgimento di una tale elaborazione procedurale, detta anche *computazione*, ha uno scopo preciso, che dal punto di vista oggettivo corrisponde molto spesso alla *risoluzione* di un qualche problema numerico. Il legame col problema del mondo reale viene stabilito grazie al potere simbolico della matematica; in tale contesto ogni simbolo (o numero) può rappresentare un qualunque ente esterno al contesto della computazione, e ciò grazie a un'operazione astratta di *codifica*, che si esplica mediante la definizione di una corrispondenze tra oggetti del mondo interno (simboli o numeri) e oggetti del mondo esterno (enti che a essi corrispondono). La codifica ci consente dunque di trasformare l'enunciazione di un problema del mondo reale in una equivalente di natura simbolica, che va poi risolta mediante il sistema di elaborazione. In ambito informatico, alla procedura in questione viene attribuito il nome di *algoritmo*, la sua esplicitazione rigorosa in un linguaggio comprensibile alla macchina viene chiamata *programma*, mentre si riserva il termine di *computazione* al processo che consiste nell'esecuzione dell'algoritmo (del programma) a partire dai dati iniziali. Il risultato della computazione, cioè i dati generati in uscita al termine della stessa, consente di esprimere in modo codificato la soluzione al problema del mondo reale.

Definizione 2.1. *Dicesi algoritmo una procedura effettiva a passi, descritta in modo preciso ed esauriente da un numero finito di istruzioni elementari $I(1), I(2), \ldots, I(n)$, la quale a partire dai dati iniziali fornisce in un tempo finito i dati finali, che vengono interpretati come la soluzione di un determinato problema. Per computazione s'intende l'esecuzione dell'algoritmo.*

Anche se la formalizzazione del concetto di algoritmo è recente ed è legata ai modelli di computazione precedentemente citati, l'idea intuitiva risale a molti secoli prima, poiché da una parte appare assodato che il termine derivi dalla trascrizione latina del nome del matematico persiano *al-Khwarizmi*, vissuto nel IX secolo D.C., che è considerato uno dei primi autori ad aver fatto riferimento a questo concetto scrivendo il libro "Regole di ripristino e riduzione"; dall'altra sono noti numerosi esempi di approccio algoritmico per la soluzione di alcuni problemi



aritmetici. I due esempi più importanti e famosi sono senza dubbio il *crivello di Eratostene* per l'individuazione dei numeri primi e *l'algoritmo di Euclide* per trovare il massimo comune divisore tra due numeri interi.

L'approccio algoritmico che abbiamo testé delineato non è l'unico possibile che consenta di ottenere la soluzione di un problema (il più delle volte di natura matematica) descrivibile secondo un approccio simbolico. Esso è tuttavia quello largamente più usato, e ciò essenzialmente per tre ordini di motivi.
Il primo deriva senza dubbio dal fatto che gli aspetti logico-fondazionali del modello computazionale legato alle procedure effettive furono acquisiti solidamente già nel corso degli anni Trenta, grazie al lavoro dei logici di cui si è accennato precedentemente (Gödel, Church, Turing ecc.). Sulla base di quegli studi si fu in grado di descrivere in termini precisi, cioè in termini matematici, che cosa significhi effettuare una computazione, quali siano i possibili modelli di computazione e quali siano (se esistono) i limiti strutturali all'approccio computazionale. Si riuscì inoltre a stabilire l'equivalenza dei vari modelli proposti, giungendo alla descrizione del modello più generale possibile di elaboratore, la cosiddetta Macchina di Turing (MdT). Essa è in grado di effettuare qualunque computazione, per complessa che possa essere, effettuata da un qualunque elaboratore reale. La definizione della MdT portò inoltre all'enunciazione della celebre tesi di Church-Turing, secondo la quale tutto ciò che si ritiene computabile effettivamente è computabile con una Macchina di Turing. Ricordiamo ancora che le fondamenta teoriche dell'Informatica furono consolidate ben prima che nascessero materialmente i calcolatori, cioè le macchine per eseguire le computazioni in modo efficiente.
Il secondo è legato alla circostanza che tale procedimento è di gran lunga il più immediato e intuitivo. Tutti noi, nella vita quotidiana, adottiamo inconsapevolmente delle strategie di tipo *algoritmico* per risolvere alcuni dei nostri problemi. Per strategia algoritmica intendiamo non solo che essa possa essere esplicitata nei termini precisi di una sequenza di azioni elementari effettive, eseguite le quali si perviene alla soluzione del problema, ma anche che la necessità di dover dare struttura procedurale alla nostra attività ci aiuta a delineare i sotto-problemi che via via siamo chiamati a risolvere, prima di giungere alla soluzione globale. La tecnica algoritmico-procedurale ci offre dunque anche la possibilità di progettare meglio e analizzare con maggior efficacia l'insieme globale delle operazioni elementari da svolgere per conseguire l'obiettivo.
Il terzo ordine di motivi che porta alla popolarità delle procedure algoritmiche è che esse sono state facilmente ed efficacemente cablate nella struttura architetturale delle macchine che devono svolgere materialmente la computazione, cioè degli odierni calcolatori. Vedremo infatti nel seguito che la quasi totalità degli elaboratori oggi impiegati dispone di un'architettura (quella di von Neumann, dal matematico ungherese che per primo la ideò) che si basa sul paradigma di una elaborazione procedurale a passi, di tipo *seriale*, eseguita sequenzialmente lungo la linea temporale da un'unità centrale di elaborazione chiamata *processore*.

Nell'architettura di von Neumann, nota in gergo tecnico anche con la denominazione di *architettura seriale*, il processore esegue una singola istruzione elementare per unità di tempo, e l'intera computazione, anche quando molto complessa, si sviluppa attraverso la concatenazione di un certo numero di passi elementari, basati sul pro-

gramma che descrive l'algoritmo. La soluzione al problema la si ricava come effetto dell'esecuzione dell'ultima istruzione oppure a seguito di un'uscita forzata dal programma, in quanto la soluzione è stata raggiunta prima.

L'architettura seriale si contrappone alle architetture di tipo *parallelo*, nelle quali la soluzione del problema emerge invece dal comportamento collettivo di un insieme di unità computazionali elementari (per esse si usa talvolta il termine metaforico di *neurone*), il cui comportamento e le cui interazioni non sono descrivibili secondo il paradigma della computazione procedurale. Tali unità computazionali sono connesse in una rete, nella quale ciascun neurone può essere potenzialmente connesso a ciascun altro neurone, dotando così la struttura di un massiccio parallelismo che richiama la topologia usata dalle strutture biologiche nella costituzione del cervello, cioè l'organo che esegue le "computazioni" necessarie all'organismo per la risoluzione dei "problemi" legati alla sua sopravvivenza.
La *computazione parallela* ha avuto dei momenti di grande entusiasmo, che si sono però sempre smorzati a causa della mancanza di un vero e proprio modello matematico generale che fosse in qualche misura la controparte di quello scoperto da Gödel, Church e Turing per l'approccio algoritmico-procedurale. A tutt'oggi non è del tutto chiaro se l'approccio parallelo possa, almeno in linea di principio, collocarsi su un piano essenzialmente diverso rispetto a quello tradizionale, anche se si è ragionevolmente portati a ritenere che la tesi di Church-Turing sia valida anche per le computazioni realizzate secondo una modalità operativa parallela. Da questo punto di vista il passaggio da computazione procedurale a computazione parallela corrisponderebbe, essenzialmente, a una semplice mutuazione della *complessità algoritmica* (cioè del numero di passi elementari necessari per eseguire una determinata computazione, fissati che siano i dati iniziali) con la *complessità strutturale* della rete, che in prima approssimazione si può assumere equivalente alla quantità di memoria necessaria per costruire materialmente la rete stessa.

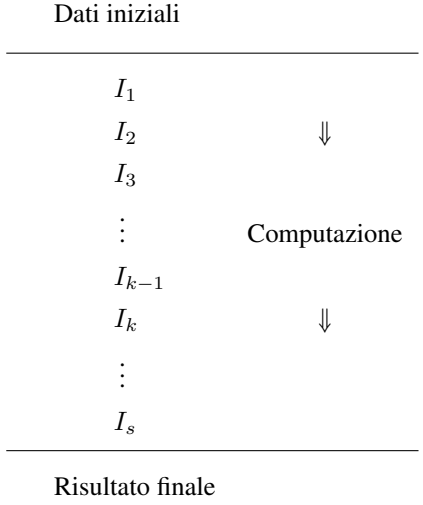
Anche se finora per le computazioni di tipo esatto (cioè quelle che richiedono una soluzione esatta del problema) l'approccio algoritmico si è dimostrato di gran lunga il più efficiente, non si può escludere che in futuro, anche se solo limitatamente a problemi per i quali ci si accontenti di una soluzione approssimata, diventi più conveniente l'impiego delle cosidette *reti neuronali* (artificiali), oppure di altri paradigmi di computazione che hanno suscitato l'interesse degli esperti, quali la *computazione DNA*, gli *algoritmi genetici* e la *computazione quantistica*. A corroborare tale aspettativa c'è comunque il dato di fatto che la natura, nell'organizzare la struttura dei suoi organismi biologici, "ha scelto" il modello parallelo, che gode di una serie di vantaggi derivanti dal fatto che la computazione è sostanzialmente innervata nella struttura della rete. In questo testo ci occuperemo tuttavia del solo approccio algoritmico-procedurale.

In conclusione vale la pena riflettere ancora sulla circostanza, già anticipata nella prefazione, che l'Informatica intesa come studio delle procedure effettive (algoritmi) prescinde dalla tecnologia che s'impiega per l'attuazione delle procedure stesse, al punto che è concepibile, in linea di principio, un'informatica senza calcolatori. La prova di ciò risiede nella comprensione del modello computazionale astratto della macchina di Turing (o di qualunque altro ad esso equivalente) che consente di eseguire una procedura algoritmica facendo uso solo di carta e penna. Poiché sulla base della tesi di Church-Turing tutto ciò che è computabile nel senso comune e intuitivo del termine lo è anche nel senso della Macchina di Turing, si deduce che tutto ciò che siamo in grado di attuare coi nostri moderni calcolatori sarebbe in linea di principio realizzabile in modo effettivo anche solo con carta e penna. Pur tuttavia gli straordinari successi dell'informatica sono stati resi possibili solo da uno sviluppo impetuoso della tecnologia microelettronica, che rappresenta oggi il sostrato fisico sul quale viene cablata l'architettura del calcolatore. Da questo punto di vista il vero salto di qualità lo si ebbe alla fine degli anni '40, quando fu inventato il *transistor*, che diede il via al processo di miniaturizzazione che consente oggi di disporre, sul palmo di una mano, della potenza di calcolo equivalente a quella di grossi elaboratori che, prima di tale invenzione, occupavano intere ali di edifici consumando decine di kilowatt.

2.2 Il concetto di algoritmo

Un calcolatore è un dispositivo che serve per eseguire *programmi*. La loro esecuzione è finalizzata alla risoluzione di *problemi* del mondo reale anche molto complessi ed eterogenei tra loro. La *codifica simbolica*, cioè la possibilità di associare un qualunque significato a un simbolo, coniugata con la possibilità offerta dal calcolatore

di manipolare i simboli in modo logicamente strutturato, consente di trasformare i problemi del mondo reale in *problemi astratti*, di natura logico-simbolica e descrivibili con gli strumenti della matematica. Il problema astratto viene poi risolto in un tempo finito, avvalendosi dell'approccio *procedurale* o *algoritmico* descritto nella sezione 2.1 e che richiamiamo brevemente: si parte da un insieme di informazioni fornite al sistema di elaborazione dall'esterno, i cosiddetti *dati iniziali*, che vengono successivamente elaborati secondo una procedura ordinata a passi, descrivibile in modo preciso ed esauriente come una sequenza *finita* $I_1, I_2, \ldots, I_s$ di *istruzioni elementari*. Il risultato dell'esecuzione dell'istruzione I_k dipende, oltre che dal tipo di istruzione, anche dai dati d'ingresso al passo k, che sono costituiti da eventuali dati esterni e dai dati che derivano dall'esecuzione dell'istruzione I_{k-1} del passo precedente.



Alla procedura in questione viene attribuito il nome di *algoritmo*; la sua esplicitazione rigorosa in un linguaggio comprensibile alla macchina viene chiamata *programma* P, mentre si riserva il termine di *computazione* al processo che consiste nell'esecuzione dell'algoritmo (del programma) a partire dai dati iniziali. Il risultato della computazione, cioè i dati generati in uscita al termine della stessa e che devono essere prodotti in tempo finito, consente di esprimere in modo codificato la soluzione al problema del mondo reale.

L'approccio procedurale-algoritmico non è l'unico che si possa concepire per risolvere problemi di natura logico-simbolica; esistono infatti anche altri *paradigmi* di computazione, quali ad esempio le *reti neuronali*, gli *algoritmi genetici*, la *computazione DNA* e la *computazione quantistica*; tuttavia il metodo procedurale è l'unico basato su un solido *modello di computazione*, legato ai lavori di Church, Turing, Gödel e Kleene, sui quali si è poi sviluppata tutta la *teoria della computazione*; essa stabilisce, tramite rigorosi teoremi matematici, i limiti intrinseci dell'approcio procedurale-algoritmico, distinguendo tra *problemi risolubili* (o *predicati decidibili*) e *problemi non-risolubili* (o *predicati indecidibili*). La successiva *teoria della complessità computazionale* si occupa invece di distinguere, all'interno dei problemi risolubili, i gradi di complessità degli stessi, legati al numero di passi elementari che bisogna svolgere per ottenere la soluzione del problema. In quest'ambito la distinzione è tra problemi *trattabili* in un tempo accettabile (in *tempo polinomiale*) e problemi di fatto *intrattabili*, (almeno quando le dimensioni del problema è sufficientemente grande), poiché richiederebbero un numero *esponenziale* di passi, portando a dei tempi di attesa per la soluzione assolutamente inaccettabili (p.es. $3,2$ milioni di anni).

Nello schema procedurale si ipotizza che le istruzioni elementari siano *immediatamente eseguibili*; l'idea di base, molto comune anche nella pratica quotidiana, è che per risolvere un problema complesso sia necessario attuare una strategia la cui descrizione venga specificata da un certo numero di *passi elementari*. Se per esempio voglio uscire dall'ufficio per andare a prendere l'autobus, dovrò aprire la porta se questa è chiusa, scegliere se andare a destra o a sinistra per raggiungere l'uscita dell'edificio, percorrere il corridoio sino alla scalinata, scegliere se salire o scendere dalla stessa ecc. Immaginiamo ora che mi trovi in un edificio che non conosco, e che per raggiungere la fermata abbia in mano un foglio con le istruzioni sul percorso; anche se non ho idea di come sia fatto l'edificio è sufficiente che esegua alla lettera l'elenco delle istruzioni per arrivare alla fermata. Le istruzioni

elementari (apri la porta, gira a destra, scendi le scale...) devono però essere alla mia portata e immediatamente eseguibili, senza l'ausilio di ulteriori "istruzioni" supplementari.
Se caliamo questo ragionamento nell'ambito dei calcolatori e se teniamo conto che essi sfruttano le tecnologie elettroniche, l'immediata eseguibilità di un'istruzione implica un'attività a livello circuitale realizzata da un *agente di calcolo* $\mathcal{A}$. Le operazioni eseguite da questo dispositivo sono molto semplici, e possono riguardare una gestione dell'informazione a livello di codice ASCII (nel caso si tratti di manipolazione di simboli su tale alfabeto) oppure di blocchi di *byte* che rappresentano dei numeri seconda una delle notazioni usate (complemento a 2, *floating point*,...) o anche una lettura o scrittura dei dati in una memoria ecc. Di conseguenza le capacità di elaborazione di $\mathcal{A}$ sono necessariamente limitate.
Per l'esecuzione dell'istruzione da parte dell'agente di calcolo potrebbe essere necessaria una *memoria* $\mathcal{M}$ di supporto (memoria RAM), per esempio per memorizzare risultati intermedi di una computazione, che può essere arbitrariamente grande.
L'interazione tra l'agente di calcolo $\mathcal{A}$ e il programma P avviene sulla trama di un *tempo discreto*, che viene specificato da un orologio interno del calcolatore (*clock*). Il tempo discreto corrisponde a una cadenza temporale prefissata e costante (p.es. 1 miliardesimo di secondo) in corrispondenza della quale possono essere effettuate le operazioni elementari a carico delle circuiteria. In altre parole il sistema rimane "congelato" tra due istanti di tempo discreto successivi. Si osservi che la modalità di interazione in tempo discreto è alternativa a una modalità in *tempo continuo*, che era invece prerogativa dei vecchi sistemi analogici, nei quali le varie grandezze variano con continuià nel tempo.
Un'altra ipotesi implicita che si fa nella realizzazione di un calcolatore è che l'interazione tra l'agente di calcolo $\mathcal{A}$ e il programma P sia *deterministica*. Ciò significa che, a partire dallo stesso insieme di dati iniziali, l'esecuzione di un insieme specificato di istruzioni (programma) porta sempre allo stesso risultato finale. Tale modalità si contrappone a una computazione nella quale esistano dei meccanismi *aleatori*, in cui sia possibile scegliere tra più di un percorso per la computazione a partire dalle stesse condizioni iniziali.
Per quanto riguarda invece la natura delle istruzioni $I_1, I_2, \ldots, I_s$, possiamo osservare che ogni istruzione I_j deve appartenere a un certo insieme $\mathcal{I} = \{I_1, I_2, \ldots l_k\}$ di istruzioni elementari che la circuiteria di $\mathcal{A}$ è in grado di svolgere. Nella logica dell'approccio procedurale le informazioni elementari, proprio in quanto tali, devono essere necessariamente "poche". Come abbiamo visto, la struttura architetturale acquisita dai vari sistemi porta a distinguere tra due filosofie, quelle $CISC$, che sta per *Complex Instruction Set Computer*, e quella $RISC$, che significa *Reduced Instruction Set Computer*. Nel primo caso le istruzioni dette elementari consentono di svolgere operazioni che sono comunque relativamente complesse e articolate, come la lettura di un dato in memoria, la sua modifica e il suo salvataggio direttamente in memoria tramite una singola istruzione. Si è però osservato che questa impostazione risulta poco vantaggiosa, poiché offre in genere costi maggiori e prestazioni modeste, visto che i tempi di decodifica e di esecuzione sono in generale maggiori anche per le istruzioni più semplici. Nel caso dell'architettura $RISC$ c'è invece un insieme molto ridotto di istruzioni effettivamente elementari (lettura e scrittura in memoria, copia di un dato, somma, sottrazione ecc.), e ciò consente di far lavorare i processori in modo più efficiente. A prescindere dal fatto che si usi un insieme $CISC$ o $RISC$, è ovvio che tale insieme debba avere una dimensione finita, anche se non è però necessario specificare una *limitazione superiore* per k.
Un altro elemento importante da considerare tra le ipotesi del calcolo procedurale-algoritmico è relativa alla dimensione dei dati di ingresso; benché non abbia senso porre alcuna limitazione ad essa, è tuttavia doveroso considerarla come una quantità finita; lo stesso può dirsi anche per la lunghezza della computazione: non si può porre un limite al numero di istruzioni che vengono eseguite prima di arrivare alla fine del programma, visto che tale numero potrebbe anche essere (molto) maggiore di n nel caso il programma contempli dei *cicli*. Tuttavia, affinché il sistema funzioni bene per le finalità per le quali esso è stato costruito e segua lo spirito della definizione di algoritmo, sarebbe auspicabile che la computazione durasse un tempo finito, anche se non limitabile. Come vedremo, però, saremo costretti ad accettare nel modello i casi in cui la computazione incappi in un ciclo infinito o *loop*, dal quale non si può uscire. In tal caso la macchina continua a girare all'infinito, senza pervenire mai all'esecuzione di un'istruzione che porti a uno stop; l'utente interpreta questo fatto come un blocco della computazione e deve forzare dall'esterno l'uscita dal programma oppure, nei casi peggiori, riavviare la macchina, perdendo in entrambi i casi tutto il lavoro svolto. Possiamo allora individuare il seguente elenco delle caratteristiche associate alla nozione informale di algoritmo: All'algoritmo descritto dai punti $1 - 10$ siamo dunque costretti ad aggiungere l'ipotesi di una computazione in qualche caso non terminante; ciò che ne esce rappresenta la proiezione del concetto astratto di

1)	insieme di istruzioni $I_1, I_2, \ldots, I_s$ di lunghezza finita	P (*programma*)
2)	c'è un *agente di calcolo*	$\mathcal{A}$ (circuiteria)
3)	c'è a disposizione della *memoria*	$\mathcal{M}$ (memoria RAM)
4)	$\mathcal{A}$ interagisce con P in *modalità discreta*	(macchina discreta)
5)	$\mathcal{A}$ interagisce con P in *modalità deterministica*	(macchina deterministica)
	Bisogna fissare un limite finito:	
6)	sulla dimensione dei dati d'ingresso?	NO
7)	sulla dimensione dell'insieme $\mathcal{I} = \{I_1, I_2, \ldots l_k\}$ di istruzioni?	NO
8)	sulla dimensione della memoria?	NO
9)	sulla capacità di computazione di A?	SÌ
10)	sulla lunghezza della computazione?	NO
	Tuttavia:	
11)	sono ammesse computazioni con un numero infinito di passi (computazioni non terminanti)	

ALGORITMO + 11) = PROGRAMMA

Figura 2.1: Nozione informale di algoritmo

algoritmo, che porta sempre a una soluzione in un tempo finito, sulla realtà del modello effettivo di computazione, che in certe sfortunate situazioni porta a un programma P che cicla all'infinito.
Senza entrare nei dettagli del problema relativo al punto 11) possiamo dire solo che, se lo escludessimo pretendendo di avere a che fare con un modello di calcolo basato sulle sole computazioni terminanti, allora ci si troverebbe in una situazione in cui alcuni problemi palesemente risolubili mediante approccio procedurale *non* sarebbero risolubili all'interno del nostro modello.

Si osservi che, anche se delineato con una certa precisione dai punti $1 - 10$, il concetto di algoritmo è alquanto sfuggente, poiché in certi casi non appare chiaro se quanto ottenuto rappresenti effettivamente la descrizione di una procedura effettiva che rispetta i precedenti punti $1 - 10$. Come esempio consideriamo la seguente funzione

$$g(n) = \begin{cases} 1 & \text{se esiste una tratta di } \textit{esattamente } n \text{ '5' consecutivi} \\ & \text{nell'espansione di } \pi \\ 0 & \text{altrimenti} \end{cases} \tag{2.1}$$

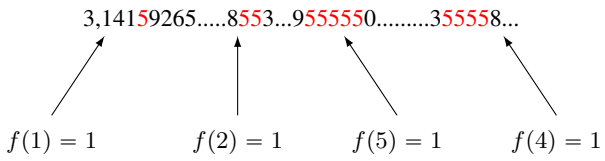
Il significato della funzione è chiaro e, dal punto di vista matematico, non rimane alcuna ambiguità nella definizione della stessa. Il problema è quello di capire se esiste una procedura effettiva per attribuire il corretto valore alla funzione (0 o 1) a ciascuno degli infiniti valori di n.
Per risolvere il problema sembra possibile far uso della serie di Hutton

$$\pi = \sum_{n=0}^{\infty} \frac{(n!\, 2^n)^2}{(2n+1)!} \left[\frac{12}{5} \left(\frac{1}{10} \right)^n + \frac{14}{25} \left(\frac{1}{50} \right)^n \right] \tag{2.2}$$

usando la seguente procedura:

1. genero i decimali di π e controllo i '5';
2. se trovo n '5' consecutivi dichiaro $f(n) = 1$, *altrimenti* dichiaro $f(n) = 0$.

Apparentemente questa sembra essere una procedura effettiva, ma in realtà ci sono dei problemi; supponiamo p.es. che la situazione sia la seguente



$f(3) = 0$ poiché non esiste una tratta di lunghezza 3

Ecco allora che incontreremo prima o poi le tratte di lunghezza $1, 2, 5, 4$ e saremo in grado di attribuire il valore $f(n) = 1$ per questi valori di n. Ma la tratta di lunghezza 3 non comparirà mai, perché non esiste; di conseguenza la nostra procedura non è "effettiva" per il valore $n = 3$, visto che dovremo attendere un tempo infinito senza mai riuscire ad attribuire un valore alla funzione, che invece è definita anche per quel valore; ed essendo definita la risposta corretta deve emergere dopo un numero finito di passi.
Per contro potrebbe darsi che, da certe proprietà teoriche di π a noi ancora non note, sia possibile un giorno derivare un algoritmo che risponde in un numero finito di passi per tutti i valori di n; p.es. se si dimostrasse con argomentazioni teoriche che $f(n) = 1 \ \forall n$, l'algoritmo diventerebbe banale, poiché dovrebbe semplicemente restituire il valore 1 per qualunque valore di n.
Poiché non abbiamo ulteriori elementi, con la funzione (2.1) siamo nella posizione di *non sapere se un algoritmo esiste*.
Consideriamo ora la seguente funzione

$$f(n) = \begin{cases} 1 & \text{se esiste una tratta di } \textit{almeno } n \text{ '5' consecutivi} \\ & \text{nell'espansione di } \pi \\ 0 & \text{altrimenti} \end{cases} \tag{2.3}$$

che è una lieve variazione della funzione precedente. Per come è definita possono succedere due cose

$$\begin{aligned} &\text{o} && g(n) = 1 && \forall n \\ &\text{oppure} \quad \exists k : && \begin{cases} g(n) = 1 & n \leqslant k \\ g(n) = 0 & n > k \end{cases} \end{aligned} \tag{2.4}$$

In entrambi i casi (funzione costante o funzione a gradino) si potrebbe dimostrare che la funzione è computabile; solo che non sappiamo quale delle due sia quella giusta. Con la funzione (2.3) siamo nella posizione di *sapere che esiste un algoritmo, ma di non sapere quale sia*.
Un esempio ancora più estremo è il seguente; prendiamo una delle congetture irrisolte della matematica, p.es. la congettura di Goldbach che stabilisce che ogni numero pari > 2 è somma di due numeri primi, e definiamo

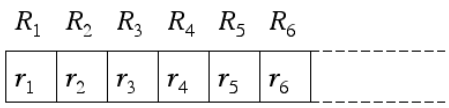
$$h(n) = \begin{cases} 1 & \text{se la congettura di Goldbach è vera} \\ 0 & \text{se la congettura di Goldbach è falsa} \end{cases} \tag{2.5}$$

E' chiaro che la funzione $h(n)$ è in ogni caso una costante, solo che non conosciamo la sua corretta derivazione. Con la funzione (2.5) siamo dunque nella posizione di *sapere che esiste un algoritmo, sapere qual è la sua tipologia*, ma non sapere il valore di uscita.

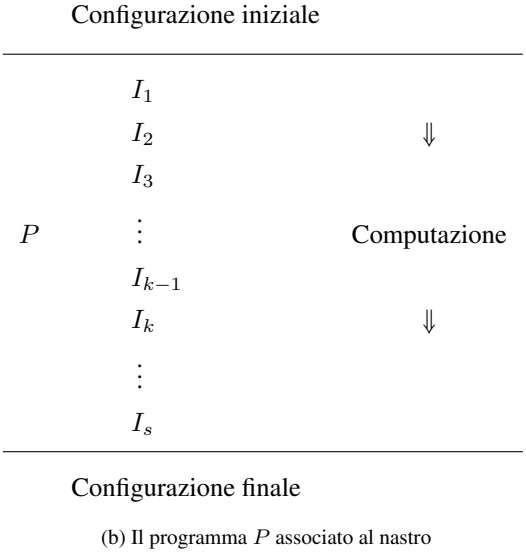
2.3 Il modello RAM

Nel paragrafo 1.3 abbiamo visto che l'informatica nacque sul solco delle riflessioni inerenti gli aspetti logico-fondazionali della Matematica, centrati sulla risoluzione del famoso *Entscheidungsproblem* (Problema della decisione). La sua risoluzione, in senso negativo, da parte di *Alonzo Church* e di *Alan Turing* nel 1936, costituisce il punto di partenza della moderna *Teoria della Computabilità*. Tuttavia il lavoro di Church era molto astratto e di difficile comprensione, mentre il modello della *Macchina di Turing* fece ben presto breccia e divenne in breve tempo il modello di riferimento. Ciononostante, questi approcci alla computazione erano stati creati non per rispondere alle esigenze di modellare il funzionamento di un *computer*, che all'epoca non esisteva ancora, ma per dare soluzione a un problema della logica. Ci si trovò dunque, ben presto, nella spiacevole situazione di avere dei *computer* pienamente funzionanti, associati però a dei modelli di computazione che non avevano nulla a che fare con le linee architetturali del sistema che intendevano rappresentare. Come già ricordato precedentemente, anche i modelli introdotti successivamente (Gödel-Kleene, Gödel-Herbrand-Kleene, Post, Markov) erano tutti basati su procedimenti di calcolo molto astratti, che non trovavano rispondenza diretta con quanto attuato dai *computer* reali. Solo nel 1963 Shepherdson e Sturgis introdussero il modello URM, che prendeva spunto dalla effettiva struttura del calcolatore moderno, basata come abbiamo visto sull'architettura di Von Neumann e sulla memoria RAM. Passiamo ora alla descrizione di una variante del modello URM originale, introdotta da Cutland nel suo celebre libro sulla computabilità [5], che chiameremo nel nostro contesto *modello RAM*.

Il modello RAM è basato su un *nastro di memoria di lunghezza infinita*, che rappresenta una idealizzazione delle memorie del calcolatore, e da un *insieme di istruzioni*, che rappresentano le istruzioni in linguaggio macchina (si veda la figura 2.2a). La memoria è costituita da celle che contengono dei numeri naturali r_i, e dunque $r_i \in \mathbb{N}$. In questo senso c'è una differenza concettuale con i computer reali, che come noto lavorano sulla base di un alfabeto binario; tuttavia tale differenza non è rilevante, poiché una qualunque stringa binaria può essere interpretata come numero intero e qualunque numero intero può avere una rappresentazione binaria. L'aspetto concettualmente rilevante è, semmai, il fatto che ogni cella, contenendo un numero intero non limitabile superiormente, è associata a una quantità d'informazione a sua volta non limitabile.



(a) Il nastro di memoria del modello RAM



(b) Il programma P associato al nastro

Figura 2.2: Gli elementi del modello RAM

Il contenuto delle celle può essere modificato da delle *istruzioni* previste dal modello, che sono state scelte in modo tale da costituire un insieme in qualche senso minimo. Tali istruzioni sono impiegate per realizzare un *programma* $P = I_1, I_2, ..., I_s$ (fig. 2.2b), che a partire da una *configurazione iniziale* del nastro porta a una *configurazione*

finale, dalla quale si ricava il *risultato* della *computazione*; quest'ultima corrisponde all'esecuzione, passo passo, delle istruzioni del programma.

Analizziamo ora le istruzioni, che sono solamente le seguenti quattro:

Istruzione di azzeramento - $Z(n)$ comporta l'azzeramento del contenuto della cella R_n.
Esempio: Se applico l'istruzione $Z(2)$ al nastro della prima riga di figura 2.3 azzero il contenuto della seconda cella (fig.2.3a)

Istruzione di incremento - $S(n)$ comporta l'incremento del contenuto della cella R_n di un'unità.
Esempio: Se applico l'istruzione $S(5)$ al nastro della seconda riga di figura 2.3 incremento di 1 il contenuto della quinta cella (fig.2.3b).

Istruzione di trasferimento - $T(m,n)$ comporta la copia del contenuto della cella R_m nella cella R_n.
Esempio: Se applico l'istruzione $T(2,4)$ al nastro della terza riga di figura 2.3 copio il contenuto della seconda cella nella quarta (fig.2.3c).

Le prime tre istruzioni si chiamano *aritmetiche*, poiché operano manipolando numeri naturali. Usando solamente questo tipo di istruzioni non sarebbe però possibile risolvere problemi con un minimo di interesse pratico, poiché nella vita reale ci si trova sempre nella situazione di dover scegliere strategie diverse a seconda che alcune condizioni siano o meno soddisfatte. In altre parole è necessario introdurre un'istruzione *logica* che ci consenta di aprire percorsi diversi alla computazione.

Istruzione di salto condizionato - $C(m,n,q)$ L'esecuzione di tale istruzione implica il controllo del contenuto delle celle R_m e R_n; se $r_m = r_n$ l'esecuzione del programma continua con l'istruzione I_q, altrimenti si continua con l'istruzione successiva.
Esempio: Se applico l'istruzione $C(1,6,9)$ al nastro della quarta riga di figura 2.3, poiché le celle 1 e 6 hanno lo stesso contenuto, la computazione continua con l'istruzione I_9 del programma (fig.2.3c).

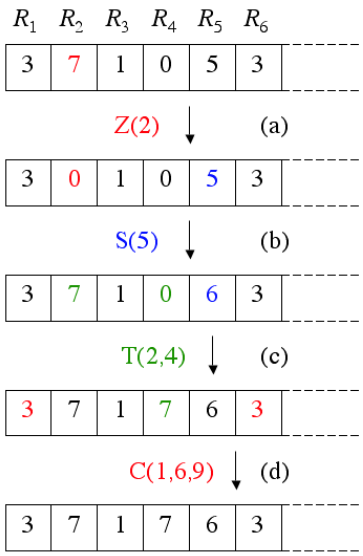


Figura 2.3: Applicazione di alcune istruzioni al nastro di memoria

Vediamo di delineare in modo più preciso i dettagli della computazione in modo che, a partire dal programma e dalla configurazione iniziale, si sappia con precisione quali sono i passi da eseguire e come si ottiene il risultato finale. Lo facciamo evidenziando alcuni punti chiave, che vengono di seguito illustrati.

Computazione - Per eseguire la computazione RAM bisogna fornire un *nastro*, caricato con una *configurazione iniziale* costituita da una sequenza $a_1, a_2, a_3, ...$ di numeri naturali; se $P = I_1, I_2, ..., I_s$ è il programma associato alla macchina, la computazione inizia eseguendo l'istruzione I_1. Dopo averla eseguita la macchina RAM dovrà eseguire la *prossima istruzione* (I_{NEXT}) finché si raggiunge uno STOP (se mai si raggiunge).

Prossima istruzione - Se I_k è l'istruzione corrente, la prossima istruzione da eseguire I_{NEXT} è così definita:

- Se I_k è un'istruzione aritmetica $I_{NEXT} = I_{k+1}$ (fig. 2.4a)
- Se $I_k = C(m, n, q)$ si ha $I_{NEXT} = \begin{cases} I_q & \text{se } r_m = r_n \quad \text{(fig. 2.4b)} \\ I_{k+1} & \text{se } r_m \neq r_n \quad \text{(fig. 2.4c)} \end{cases}$

La computazione ha un flusso regolare quando si ha a che fare con istruzioni aritmetiche, oppure quando si incontra un'istruzione di salto condizionato e vale la condizione $r_m \neq r_n$. Se viceversa si ha $r_m = r_n$ la computazione salta all'istruzione I_q.

I_1	I_1	I_1
I_2	I_2	I_2
.	.	.
$I_k = S(3)$	$I_k = C(3, 5, 9)$	$I_k = C(3, 5, 9)$
$\boldsymbol{I(k+1)}$	.	$\boldsymbol{I(k+1)}$
.	$\boldsymbol{I(9)}$	.
.	.	.
I_s	I_s	I_s

(a) Prossima istruzione nel caso di I_k istruzione aritmetica

(b) Prossima istruzione nel caso di $I_k = C(m, n, q)$ con $r_m = r_n$

(c) Prossima istruzione nel caso di $I_k = C(m, n, q)$ con $r_m \neq r_n$

Figura 2.4: I casi possibili per la prossima istruzione (in grassetto)

STOP della computazione - La computazione si conclude (se si conclude) nel momento in cui si deve eseguire un'istruzione I_v con $v > s$; ciò accade per due possibili motivi:

- È stata eseguita $I_k = I_s$ aritmetica, oppure $I_k = I_s = C(m, n, q)$ con $r_m \neq r_n$ (fig. 2.5a e 2.5b) e la prossima istruzione da eseguire sarebbe $I_v = I_{s+1}$.
- È stata eseguita $I_k = C(m, n, v)$ con $r_m = r_n$ e $v > s$ (fig. 2.5c).

La prima delle due possibilità corrisponde all'interruzione della computazione per mancanza di altre istruzioni, in quanto è stata eseguita l'ultima istruzione del programma; nel secondo caso, invece, si ha un'uscita forzata poiché si è ottenuto il risultato richiesto e non è più necessario procedere oltre con la computazione. Ciò si ottiene richiedendo l'esecuzione di un'istruzione I_v il cui indice v non figura nell'elenco delle istruzioni del programma; ciò porta (convenzionalmente) a una terminazione della computazione. Nell'esempio di figura (2.5c) è stato posto $v = 99$; ciò fa intendere che se vale la condizione $r_m = r_n$, allora ci sarà uno STOP nella computazione, poiché risulta evidente che gli esempi che faremo non avranno mai un numero di istruzioni superiore a 99.
Altro esito possible per la computazione è che essa entri in un ciclo e non giunga mai allo STOP. In questo caso si parla di computazione non terminante o *divergente*.

I_1	I_1	I_1
I_2	I_2	I_2
.	.	.
.	.	$I_k = C(3,5,99)$
.	.	.
$I_s = I_k = S(3)$	$I_s = I_k = C(3,5,9)$	I_{37}
(a) STOP della computazione con I_s=I_k aritmetica	(b) STOP della computazione con I_k=I_s=$C(m,n,q)$ e con $r_m \neq r_n$	(c) STOP della computazione con I_s=I_k=$C(m,n,v)$ e con r_m=r_n e v>s

Figura 2.5: I casi possibili per lo STOP della computazione

Configurazione iniziale - È la configurazione dalla quale si parte per effettuare la computazione. Se $a_1, a_2, ..., a_n$ sono i dati d'ingresso, per convenzione essi vengono posti all'inizio del nastro, lasciando a 0 tutte le altre (infinite) celle di memoria (fig. 2.6).

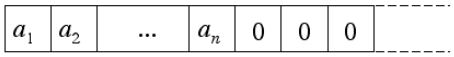


Figura 2.6: Configurazione iniziale del nastro caricato con i dai iniziali $a_1, a_2, ..., a_n$

Configurazione finale - È la configurazione che si ottiene alla fine della computazione. Per convenzione il valore b calcolato dalla computazione è il contenuto della prima cella. Ciò accade ovviamente nel solo caso in cui la computazione termini.

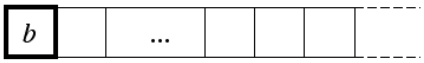
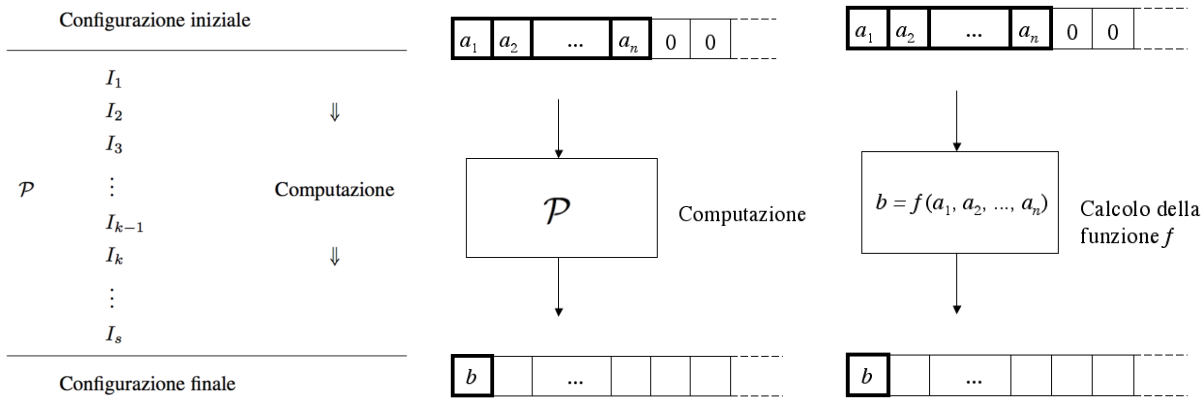


Figura 2.7: Configurazione finale del nastro: il contenuto della prima cella viene considerato il valore calcolato

Convergenza - Indichiamo con la notazione $P(a_1, a_2, ..., a_n)$ la computazione del programma P a partire dalla configurazione iniziale $a_1, a_2, ..., a_n$. Se tale computazione termina diciamo che c'è stata *convergenza*, e scriviamo $P(a_1, a_2, ..., a_n)\downarrow$. Se b è il contenuto della prima cella alla fine della computazione diciamo che la computazione è andata a convergenza su b e scriviamo $P(a_1, a_2, ..., a_n)\downarrow b$. Se la computazione non termina diciamo che si è avuta una *divergenza* e scriviamo $P(a_1, a_2, ..., a_n)\uparrow$

2.4 Computabilità di una funzione

Analizziamo ora il significato della computazione del programma P=$\{I_1, I_2, ..., I_s\}$ da un punto di vista più astratto. Sappiamo che a partire dalla configurazione iniziale del nastro si perviene alla configurazione finale, che corrisponde al *risultato* della computazione, cioè dell'esecuzione, passo passo, delle istruzioni del programma (fig.2.8a). La computazione è quindi un procedimento astratto il quale, a partire da una certa configurazione iniziale $x_1, x_2, ..., x_n$ restituisce in un tempo finito un certo valore y, leggibile sulla prima cella del nastro (fig.2.8b). Ma un procedimento che associ un numero intero y a un'n-pla $x_1, x_2, ..., x_n$ di interi corrisponde al calcolo della funzione $y = f(x_1, x_2, ..., x_n)$, che è una funzione $f : \mathbb{N}^n \rightarrow \mathbb{N}$ (fig.2.8c).



(a) La computazione del programma P corrisponde all'esecuzione delle sue istruzioni

(b) La computazione del programma P dal punto di vista astratto

(c) La computazione del programma P corrisponde al calcolo di una certa funzione f

Figura 2.8: Il significato della computazione RAM

Invertendo i termini della questione possiamo affermare che, volendo *calcolare* la funzione $y{=}f(x_1, x_2, ..., x_n)$ possiamo far uso del programma P. Se si vuole calcolare $f(a_1, a_2, ..., a_n)$, mettiamo i valori $a_1, a_2, ..., a_n$ come configurazione iniziale e facciamo partire la computazione; a seguito della convergenza dobbiamo trovare il valore b nella prima cella. Si osservi che la funzione potrebbe non essere definita per qualche n-pla d'ingresso, per esempio per la $a_1^*, a_2^*, ..., a_n^*$; in tal caso nella tabella della funzione mettiamo un trattino "$-$" al posto del valore della funzione; una funzione di questo tipo viene chiamata *parziale*. È ovvio che in questa circostanza non possiamo far convergere la computazione, perché il valore letto nella prima cella alla fine della stessa verrebbe interpretato come *il* valore della funzione, che invece non è definita. L'unica possibilità che rimane è allora quella di far divergere la computazione nel caso in cui la funzione non sia definita. Se invece la funzione è definita su tutto $\mathbb{N}^n$ non ci sono problemi e ci sarà sempre convergenza; in tal caso la funzione si chiama *totale*.

Definizione 2.2. ***Calcolo di una funzione tramite un programma*** *Sia f una funzione parziale da $\mathbb{N}^n$ in $\mathbb{N}$ e sia P un programma. Diciamo che P RAM-calcola la funzione f se, $\forall a_1, a_2, ..., a_n, b \in \mathbb{N}$*

$$P(a_1, a_2, ..., a_n) \downarrow b \quad \Leftrightarrow \quad \begin{cases} a_1, a_2, ..., a_n \in \mathit{Dom}(f) \\ f(a_1, a_2, ..., a_n) = b \end{cases}$$

il che implica che $P(a_1, a_2, ..., a_n)\uparrow$ se e solo se $a_1, a_2, ..., a_n \notin \mathit{Dom}(f)$

Definizione 2.3. ***Funzione calcolabile*** *Una funzione f si dice calcolabile se esiste un programma P che la RAM-calcola.*

La situazione è dunque quella di figura 2.9; in corrispondenza del dominio di f, cioè laddove la funzione è definita, la computazione converge al valore della funzione. Fuori dal dominio la funzione non è definita e la computazione diverge. Si osservi che una computazione divergente, cioè un *loop* che corrisponde a un blocco del computer, non significa che la funzione non è computabile, ma che la funzione *non è definita* per i corrispondenti valori d'ingresso.

Definizione 2.4. ***Funzione associata a un programma*** *Assegnato un certo programma P, cioè una lista di istruzioni nel linguaggio RAM, e un valore $n \geqslant 1$, se pensiamo all'effetto che P ha su un prefissato insieme di valori*

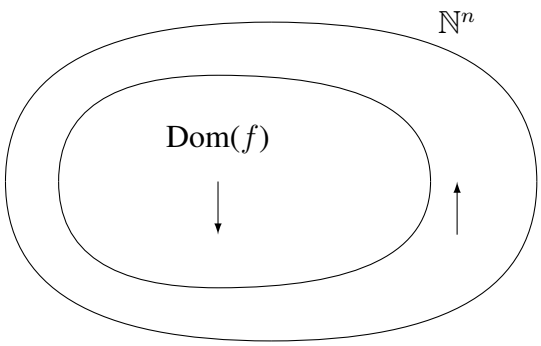


Figura 2.9: Dominio delle funzioni computabili su $\mathbb{N}^n$

iniziali $a_1, a_2, ..., a_n, 0, 0, ...$ *possiamo dedurre che esiste un'unica funzione* n*-aria* $f_P^{(n)}$ *che* P *computa; essa è*

$$f_P^{(n)} \quad = \quad \begin{cases} \textit{l'unico } b \textit{ tale che } P(a_1, a_2, ..., a_n) \downarrow b \\ \textit{se } P(a_1, a_2, ..., a_n) \downarrow \\ \\ \textit{indefinita, se } P(a_1, a_2, ..., a_n) \uparrow \end{cases}$$

D'altra parte è ben noto che diversi programmi possono essere associati alla stessa funzione. Cioè può succedere per due ordini di motivi; il primo è quello di un diverso metodo per calcolare la stessa funzione; il secondo è quello che corrisponde all'aggiunta di istruzioni inutili e ininfluenti sulla computazione. Discuteremo in dettaglio la questione negli esempi che seguiranno.
Un'ultima importante osservazione. Potrebbe sembrare una forzatura il fatto che, per esprimere il funzionamento e l'attività di un calcolatore, si debba ricorrere al linguaggio delle funzioni. In realtà ci si rende subito conto che questo è il linguaggio corretto per rappresentare qualunque attività del computer. Esso è infatti una macchina discreta che lavora in tempo discreto; immaginiamo allora di fare una "fotografia" dei suoi circuiti in un certo istante discreto t_0. Poiché il computer è costituito da milioni di transistors, ciascuno dei quali lavora secondo una logica binaria, in linea di principio potremmo fare una lista ordinata di tali transistors, scrivendo "0" o "1" a seconda che, all'istante t_0, il componente sia nello stato di piena conduzione o di interdizione. Il lungo elenco di zeri e uni corrisponde a un vettore binario, con un numero di coordinate uguale al numero di transistors, che descrive lo *stato* della macchina; indichiamo tale vettore con la notazione $a_1, a_2, ..., a_n$. Se passiamo ora all'istante successivo t_1, ci sarà stata un'evoluzione dello stato a seguito delle istruzioni del programma. Supponiamo per esempio che si stia spostando la freccia del *mouse* sullo schermo; potrebbe allora succedere che in t_0 un certo *pixel*, a 256 livelli di colore, fosse p.es. color bianco, caratterizzato dalla codifica 00001011, che corrisponde al numero 11 nella scala $0...255$. Durante lo spostamento, all'istante t_1 il *pixel* diventa color grigio scuro, e viene codificato dal vettore 11011010 che corrisponde al numero 218. Possiamo allora affermare che la descrizione del funzionamento del *pixel* in questione è basata sul calcolo della funzione $f(a_1, a_2, ..., a_n) = 218$. È ovvio che, secondo questo approccio, serve una funzione per ciascun *pixel*, ma questo è un problema tutto quantitativo, legato alla circostanza che ci sono moltissimi *pixel*. Facciamo ora alcuni esempi di costruzione di programmi elementari che servono per spiegare il funzionamento del modello.

Esempi

Esempio 2.1. Si voglia scrivere un programma che calcola la funzione $f(x) = 0, \ \forall x$.

Si tratta di partire dalla configurazione iniziale $x, 0, 0, \ldots$ per giungere alla configurazione finale $0, \ldots$, qualunque sia il valore di x. Il "programma" è in questo caso costituito dalla sola istruzione $Z(1)$, che porta a zero la prima cella, qualunque sia il contenuto della stessa.

Esempio 2.2. Scriviamo un programma che calcola la funzione $f(x) = 3,\ \forall x$.

Partendo dalla configurazione $x, 0, 0, \ldots$ si deve giungere alla configurazione finale $3, \ldots$, qualunque sia il valore di x. Il programma è

1	$Z(1)$	(porta a zero la prima cella, qualunque sia x)
2	$S(1)$	(incrementa la prima cella)
3	$S(1)$	(incrementa la prima cella)
4	$S(1)$	(incrementa la prima cella)

Si noti che si può ottenere la stessa funzione incrementando per tre volte il contenuto di una qualunque cella diversa dalla prima (p.es. la seconda) e usando l'istruzione $T(m, n)$ per trasferire il risultato

1	$S(2)$	(incrementa la seconda cella)
2	$S(2)$	(incrementa la seconda cella)
3	$S(2)$	(incrementa la seconda cella)
4	$T(2, 1)$	(copia nella prima cella il contenuto della seconda)

Esempio 2.3. Si voglia scrivere un programma che calcola la funzione $f(x) = x + 3,\ \forall x$.
Partendo dalla configurazione $x, 0, 0, \ldots$ si deve giungere alla configurazione finale $x + 3, \ldots$, qualunque sia il valore di x. Il programma è

1	$S(1)$	(incrementa la seconda cella)
2	$S(1)$	(incrementa la seconda cella)
3	$S(1)$	(incrementa la seconda cella)

Esempio 2.4. Si voglia scrivere un programma che calcola la funzione $f(x, y) = x + y,\ \forall (x, y)$.

Partendo dalla configurazione $x, y, 0, \ldots$ si deve giungere alla configurazione finale $x + y, \ldots$, qualunque siano i valori di x e y. Si noti che non possiamo inserire y comandi $S(1)$, poiché non conosciamo a priori il valore di y. Bisogna allora ricorrere alla seguente strategia: si incrementa progressivamente x, in modo che diventi $x + 1$, $x + 2$, $x + 3$, ..., $x + k$,... memorizzando nel contempo il valore corrente di k su una delle celle libere. Quando si giunge al valore k tale che $k = y$, allora nella prima cella c'è $x + y$. Si osservi che il nodo della computazione è il controllo $k \stackrel{?}{=} y$. Se $k \neq y$ dobbiamo continuare a incrementare k; se viceversa $k = y$ allora abbiamo concluso. Lo stato corrente della computaizone è allora

$x+k$	y	k	0	0	$\cdots$

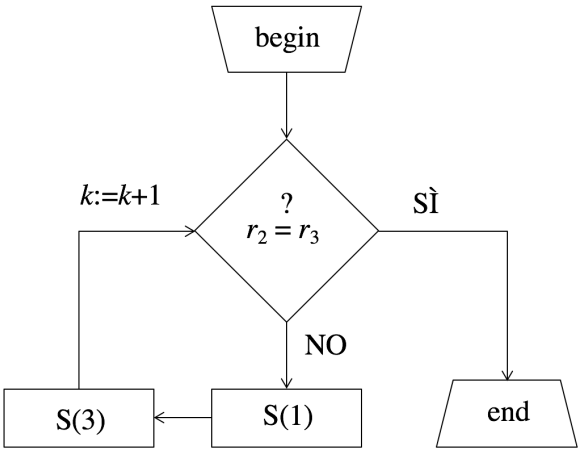
$k \stackrel{?}{=} y$

Per poter intraprendere due percorsi diversi di computazione a seconda che k sia o meno uguale a y, è necessario introdurre l'istruzione di salto condizionato $C(m, n, q)$, con m e n indirizzi di memoria dove sono memorizzati k e y. Si noti peraltro che il controllo deve essere svolto immediatamente, poiché potrebbe accadere che sia $y = 0$, nel qual caso avremmo già terminato. Sarà dunque necessario iniziare con un controllo del tipo $C(2, 3, 99)$, che ci fa concludere la computazione se il contenuto della seconda cella coincide con quello della terza, cioè se $k = y$. Se viceversa $k \neq y$ si deve incrementare k, cioè $k := k + 1$[1] nella prima cella ($S(1)$) e nella terza cella ($S(3)$). A questo punto la nuova configurazione sarà $x + k + 1, y, k + 1, 0, 0, \ldots$, e dovremo rifare il controllo, per verificare se $k + 1 = y$ o meno. Senza inserire una nuova istruzione $C(2, 3, 99)$ nel programma, il controllo può essere fatto tornando alla prima istruzione $C(2, 3, 99)$ mediante un'istruzione di *salto incondizionato* del tipo $C(1, 1, 1)$; poiché è sempre vero che il contenuto della prima cella è uguale a sè stesso, si riaccede alla prima istruzione e si effettua nuovamente il controllo. In questo modo si crea un *ciclo*, dal quale si esce solo quando è verificata la condizione $r_2 = r_3$.

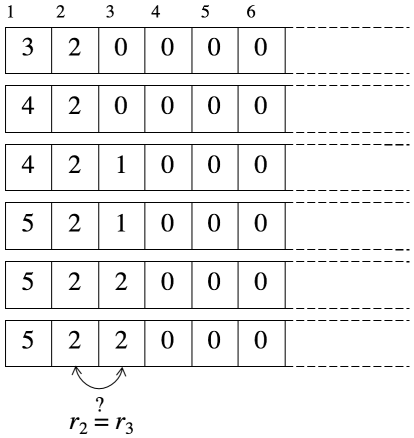
[1] Il simbolo ":=" si legge "diventa".

1	$C(3,2,99)$	(controllo se $k = y$)
2	$S(1)$	($k := k + 1$ nella prima cella)
3	$S(3)$	($k := k + 1$ nella terza cella)
4	$C(1,1,1)$	(salto incondizionato alla prima istruzione)

In figura 2.10a viene riportato il *diagramma di flusso* del programma, che mette in relazione le operazioni effettuate dallo stesso, mentre in figura 2.10b viene riportata la sequenza degli stati di memoria nella somma $3 + 2$.



(a) Diagramma a flusso del programma che calcola la funzione $f(x, y) = x + y$



(b) Successione degli stati per il calcolo della somma 3+2

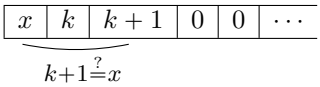
Figura 2.10:

Esempio 2.5. Si voglia scrivere un programma che calcola la funzione $f(x) = x - 1$ sui numeri naturali.

Essa si indica col simbolo $\dot{-}$ e si definisce nel modo seguente

$$f(x) = x \dot{-} 1 = \begin{cases} x - 1 & x > 0 \\ 0 & x = 0 \end{cases} \quad \forall x$$

Per realizzare questa funzione dobbiamo costruire una differenza a partire dalle quattro istruzioni di base, che *non* contemplano alcun tipo di sottrazione. Una possibilità è quella di avere una configurazione corrente del tipo



in modo tale che quando $x = k + 1$, nella seconda cella si trova $k = x - 1$, che deve essere trasferito nella cella iniziale. Poiché la funzione vale 0 quando $x = 0$, la prima cosa da fare è la seguente verifica $x \stackrel{?}{=} 0$; se la risposta è sì allora abbiamo terminato e possiamo uscire, altrimenti si continua col programma. Si tratta allora di porre il $+1$ nella terza cella e fare la verifica $r_1 \stackrel{?}{=} r_3$; se la condizione è soddisfatta si esce, altrimenti $k := k + 1$ e inizia un ciclo.

```
1        C(1, 4, 99)    (controlla se x = 0 usando la quarta cella)
2        S(3)           (pone +1 nella terza cella)
3   ┌──→ C(1, 3, 7)  ┌┄  (controlla se x = k + 1)
4   ┆    S(2)        ┆   (k := k + 1 nella seconda cella)
5   ┆    S(3)        ┆   (k := k + 1 nella terza cella)
6   └┄┄  C(1, 1, 3)  ┆   (salto incondizionato alla terza istruzione)
7        T(2, 1)   ←──┘   (trasferisce il risultato x − 1 nella prima cella)
```

In figura 2.11 viene riportato il *diagramma a flusso* del programma, che mette in relazione le operazioni effettuate dallo stesso.

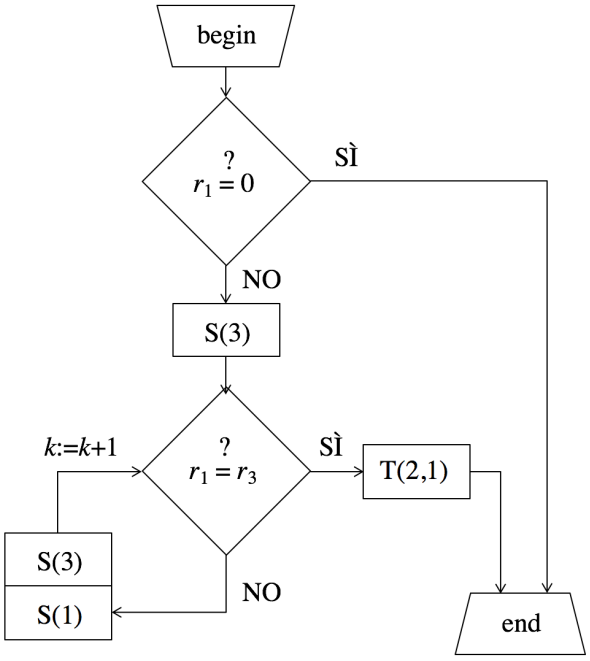


Figura 2.11: Diagramma a flusso del programma che calcola la funzione $f(x) = x \dot{-} 1$

Esempio 2.6. Si voglia scrivere un programma che calcola la funzione $f(x) = x - y$ sui numeri naturali, che si definisce nel modo seguente

$$f(x) = x \dot{-} y = \begin{cases} x - y & x \geqslant y \\ \text{indef} & x < y \end{cases} \quad \forall (x, y)$$

Anche in questo caso si tratta di costruire una differenza a partire dalle quattro istruzioni di base, che *non* contemplano alcun tipo di sottrazione. Quando la funzione è definita si ha sempre $x \geqslant y$, e la differenza $x - y$ corrisponde al numero di volte in cui bisogna incrementare y per arrivare a x. La soluzione è allora quella di avere una configurazione corrente del tipo

x	$y+k$	k	0	0	$\cdots$

$y+k \stackrel{?}{=} x$

in modo tale che, quando $x = y + k$, nella terza cella si trova $k = x - y$, che deve essere poi trasferito nella cella iniziale. Anche in questa caso bisogna iniziare con un controllo, poiché potrebbe succedere che sia $x = y$, nel qual caso sarebbe già tutto finito.

1	$C(1,2,5)$	(controlla se $x = y + k$)
2	$S(2)$	($k := k + 1$ nella seconda cella)
3	$S(3)$	($k := k + 1$ nella terza cella)
4	$C(1,1,1)$	(salto incondizionato alla prima istruzione)
5	$T(3,1)$	(trasferisce il risultato $x - y$ nella prima cella)

In figura 2.12 viene riportato il *diagramma a flusso* del programma, che mette in relazione le operazioni effettuate dallo stesso. Osserviamo che se $x < y$, sarà sempre vero che $x < y + k$, e quindi il confronto tra r_1 e r_2 della prima

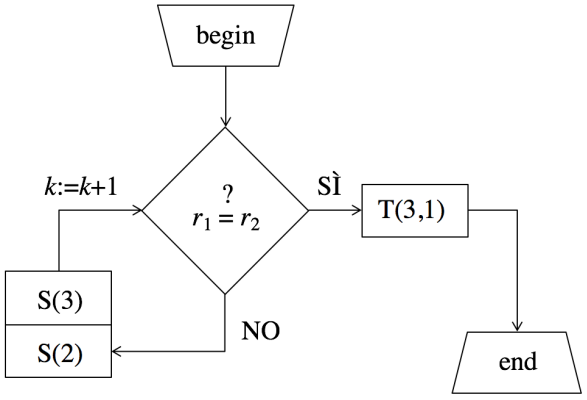


Figura 2.12: Diagramma a flusso del programma che calcola la funzione $f(x) = x \dot{-} 1$

istruzione $C(1,2,5)$ darà sempre esito negativo; non ci sarà quindi la possibilità di uscire dal ciclo per terminare con l'istruzione $T(3,1)$ e la macchina ciclerà all'infinito. Ciò è coerente col fatto che per $x < y$ la funzione risulta non definita.

Esempio 2.7. Scriviamo un programma che calcola la funzione

$$f(x) = \begin{cases} x/2 & \text{se } x \text{ pari} \\ \text{indef} & \text{se } x \text{ dispari} \end{cases} \quad \forall x$$

La configurazione corrente che risolve il problema è $x, k, 2k, 0, 0, \ldots$, in modo tale che quando $x = 2k$ nella seconda cella si ha $k = x/2$. Ogniqualvolta incrementiamo k di un'unità nella seconda cella, dobbiamo incrementare di due unità la terza

1	$C(1,3,6)$	(controlla se $x = 2k$)
2	$S(2)$	($k := k + 1$ nella seconda cella)
3	$S(3)$	($k := k + 1$ nella terza cella)
4	$S(3)$	($k := k + 2$ nella terza cella)
5	$C(1,1,1)$	(salto incondizionato alla prima istruzione)
6	$T(2,1)$	(trasferisce il risultato $k = x/2$ nella prima cella)

Esempio 2.8. Scriviamo un programma che calcola le seguenti funzioni

$$f(x) = \begin{cases} 1 & \text{se } x = 0 \\ 0 & \text{se } x \neq 0 \end{cases} \quad \forall x$$

1	$C(1,2,4)$
2	$Z(1)$
3	$C(1,1,99)$
4	$S(1)$

$$f(x) = \begin{cases} 0 & \text{se } x = 0 \\ 1 & \text{se } x \neq 0 \end{cases} \quad \forall x$$

1	$C(1,2,99)$
2	$Z(1)$
3	$S(1)$

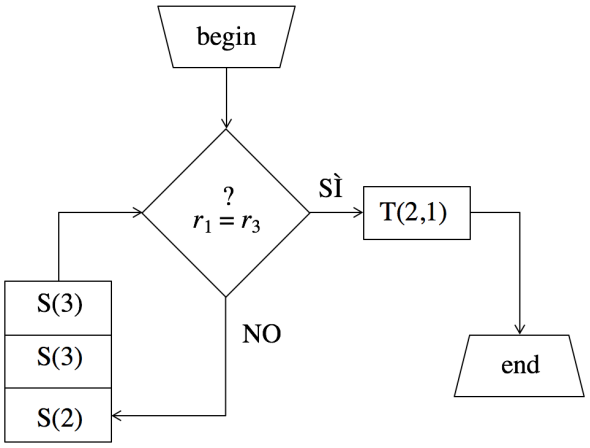


Figura 2.13: Diagramma a flusso del programma che calcola la funzione $f(x) = x/2$

Esempio 2.9. Scriviamo un programma che calcola le seguenti funzioni

$$f(x,y) = \begin{cases} 1 & \text{se } x = y \\ 0 & \text{se } x \neq y \end{cases} \quad \forall (x,y) \qquad\qquad f(x,y) = \begin{cases} 0 & \text{se } x = y \\ 1 & \text{se } x \neq y \end{cases} \quad \forall (x,y)$$

1	$C(1,2,4)$
2	$Z(1)$
3	$C(1,1,99)$
4	$Z(1)$
5	$S(1)$

1	$C(1,2,5)$
2	$Z(1)$
3	$S(1)$
4	$C(1,1,99)$
5	$Z(1)$

Selezione e iterazione col linguaggio del modello RAM

Abbiamo più volte sottolineato il fatto che l'approccio procedurale-algoritmico fa riferimento al concetto di programma P, costituito da un elenco di istruzioni $I_1, I_2, \ldots, I_s$, che devono essere eseguite in sequenza. Questo è l'approccio seguito nell'ambito della programmazione imperativa, dominante rispetto ad altri paradigmi. Ogni istruzione corrisponde a un "comando" che viene impartito alla macchina, e che prevede l'esecuzione di un certo lavoro. Le istruzioni a livello di *linguaggio macchina*, che dà corpo a tali comandi, sono istruzioni molto "povere", nel senso che dovendo essere direttamente eseguibili dal processore non possono prevedere elaborazioni troppo complesse, ma si limitano a operazioni di base del tipo "somma il contenuto di due celle di memoria", oppure "copia il contenuto di un cella in un'altra", "azzera il contenuto di una cella", ecc. Le istruzioni del linguaggio RAM sono proprio di questo tipo, e in tal senso costituiscono un modello molto realistico del funzionamento del calcolatore a livello *hardware*; in linea di principio sarebbe possibile costruire un vero e proprio computer basato su un *assembly* costituito dalle sole quattro istruzioni del linguaggio RAM.
Tuttavia, quando si deve realizzare un programma non si lavora mai a livello di linguaggio macchina, poiché sarebbe inutilmente faticoso, inefficiente e frustrante. Si preferisce invece operare a un livello logico superiore, usando linguaggi come C, C++, Java, Fortran, Pascal, ecc. per i quali le istruzioni elementari consentono di effettuare operazioni logiche più complesse di quanto si possa realizzare in linguaggio macchina, molto vicine alla logica che guida il ragionamento umano. In questo modo c'è anche il vantaggio che i programmi in tali linguaggi sono indipendenti dall'architettura della macchina. Ricordiamo inoltre che la traduzione tra un linguaggio ad *alto livello* di questo tipo e il linguaggio macchina, viene garantita da un programma che si chiama *compilatore*, e che andrà

adattato alle diverse piattaforme architetturali in uso.
Se facciamo riferimento al paradigma più usato di programmazione imperativa, denominato *programmazione strutturata*, possiamo affermare che un programma è solitamente costruito nel seguente modo:

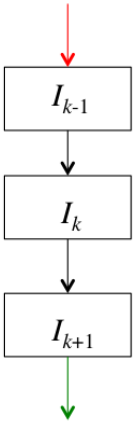
- *una parte dichiarativa*, in cui si dichiarano tutte le variabili del programma e il loro tipo (p.es. variabile intera, variabile carattere, ecc);

- *una parte che descrive l'algoritmo* risolutivo utilizzato, basato sulle istruzioni del linguaggio;
  a loro volta le istruzioni si dividono in:

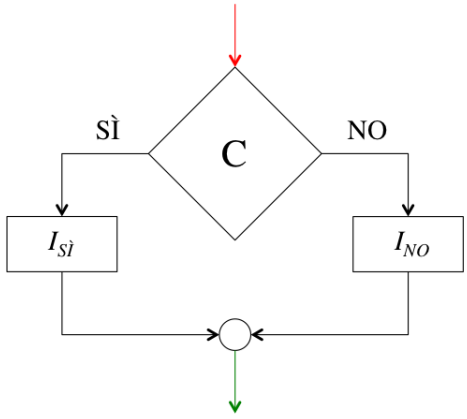
  - istruzioni di lettura e scrittura (scrittura a video, scrittura su disco, lettura da tastiera, ...);
  - istruzioni di assegnamento (del valore a una variabile);
  - istruzioni logiche di controllo (frasi if, while, for, repeat, case, ...).

In un linguaggio strutturato, quale ad esempio il Pascal, ci sono sostanzialmente tre tipi di *strutture logiche di controllo* del programma; esse sono rispettivamente la *Sequenza*, la *Selezione* e l'*Iterazione*. Analizziamole nei dettagli:

Nella **Sequenza** (Fig. 2.14a) le istruzioni sono semplicemente poste in sequenza, una dopo l'altra. Il punto d'ingresso è evidenziato da una freccia rossa, quello di uscita da una freccia verde.



(a) Sequenza di istruzioni

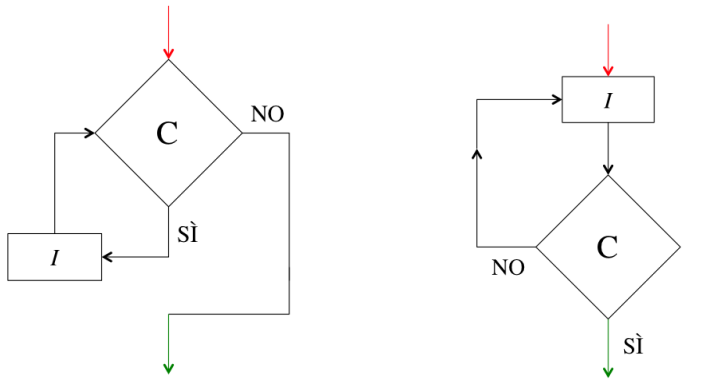


(b) Selezione effettuata sulla base della validità della condizione C; corrisponde a una frase Pascal **if** C **then** $I_{SÌ}$ **else** I_{NO}

Figura 2.14: Strutture logiche di controllo del tipo *sequenza* e *selezione*

Nella **Selezione** (Fig. 2.14b) si procede a una verifica della validità della condizione C, seguendo due strade diverse a seconda che C sia o meno vera; se C è vera (SÌ) si procede con l'esecuzione dell'istruzione $I_{SÌ}$, altrimenti (NO) si esegue l'istruzione I_{NO}. Nel linguaggio Pascal la selezione corrisponde alla frase **if** C **then** $I_{SÌ}$ **else** I_{NO}

Nell'**Iterazione** (Fig. 2.15) si attiva un *ciclo*, controllato dalla realizzazione di una certa condizione prefissata. Il controllo può avvenire in due modi diversi: nel primo la validità della condizione C viene verificata subito; se essa è soddisfatta si esegue I altrimenti si esce. In questo caso si rimane nel ciclo e si continua a eseguire I finché la condizione vale; ciò corrisponde a una frase Pascal del tipo **while** C **do** I (si veda la figura 2.15a). Nel secondo caso viene eseguita subito l'istruzione I e solo successivamente si attua la verifica della condizione C; se essa non è soddisfatta si ri-esegue I altrimenti si esce; ciò corrisponde a una frase Pascal del tipo **repeat** I **until** C (si veda la figura 2.15b). Si osservi che in questo secondo caso l'istruzione I viene eseguita almeno una volta.
Immaginiamo ora che il linguaggio macchina del calcolatore sia basato sul linguaggio del modello RAM, e che si voglia "compilare", cioè tradurre una frase complessa del linguaggio Pascal in una sequenza di istruzioni RAM. Il



(a) Iterazione del tipo **while** C **do** I

(b) Iterazione del tipo **repeat** I **until** C

Figura 2.15: Due tipi diversi di iterazione

punto nodale è chiaramente la costruzione delle strutture logiche di controllo, perché per il resto si tratta solo di assegnazioni o di operazioni di *input/output* dei dati. Si tratta quindi di costruire un programma RAM per ciascuna delle tre frasi **if-then-else**, **while-do** e **repeat-until**, in modo da poter dare le istruzioni corrette a livello macchina quando una di queste frasi venga evocata su un programma di un linguaggio ad alto livello. Cominciamo con la prima.

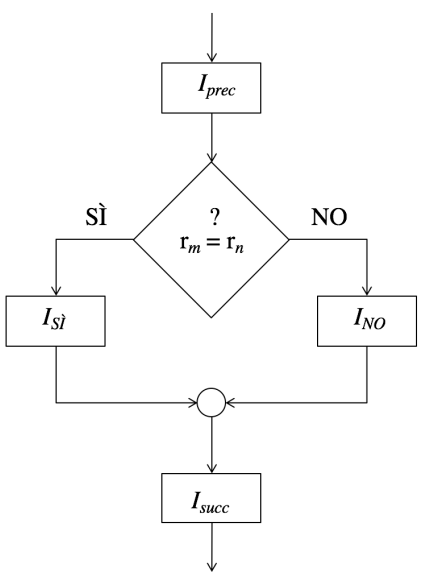
Frase di selezione **if** C **then** $I_{SÌ}$ **else** I_{NO}

Immaginiamo che la frase sia preceduta da una generica istruzione RAM che chiamiamo I_{prec} e seguita da (da una generica istruzione RAM che chiamiamo) I_{succ}. Se abbiamo già eseguito l'istruzione I_{prec} dobbiamo ora scegliere uno dei due percorsi, che portano a una delle due istruzioni $I_{SÌ}$ o I_{NO}, a seconda che la condizione C sia o meno vera. Per effettuare un tale tipo di verifica il linguaggio RAM ci offre l'istruzione $C(m, n, q)$; la condizione C di figura 2.14b deve essere dunque ricondotta a una verifica del tipo $r_m \stackrel{?}{=} r_n$. Se la risposta è sì si va a sinistra e si esegue $I_{SÌ}$; se la risposta è no si va a destra e si esegue I_{NO}. Sotto riportiamo le righe del codice associato.

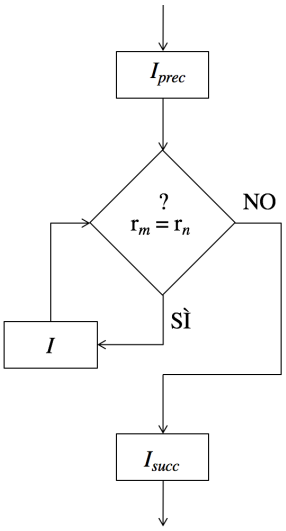
$k-1$	I_{prec}	(esegue l'istruzione precedente)
k	$C(m, n, k+3)$	(controlla se $r_m = r_n$)
$k+1$	I_{NO}	(se $r_m \neq r_n$ si esegue I_{NO})
$k+2$	$C(1, 1, k+4)$	(passa a I_{succ}, senza eseguire anche $I_{SÌ}$)
$k+3$	I_{SI}	(se $r_m = r_n$ si esegue $I_{SÌ}$)
$k+4$	I_{succ}	(esegue l'istruzione successiva)

Frase di iterazione **while** C **do** I

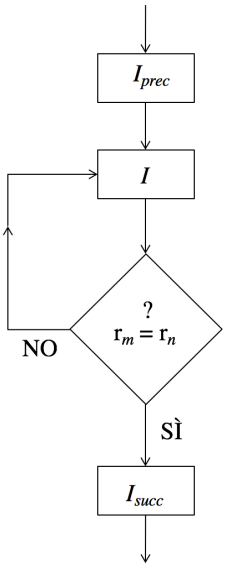
Eseguita I_{prec} si deve verificare la validità della condizione $r_m \stackrel{?}{=} r_n$. Se la condizione è soddifatta si innesca un ciclo che prevede l'esecuzione dell'istruzione I; tale istruzione non può tuttavia seguire immediatamente l'istruzione di controllo, poiché l'istruzione immediatamente successiva a $C(m, n, .)$ è quella che viene attivata nel caso in cui la condizione *non* fosse stata soddisfatta; bisogna dunque fare un salto incondizionato $C(1, 1, k+2)$ alla riga $k+2$. Bisogna poi rifare la verifica $r_m \stackrel{?}{=} r_n$, e per questo si attiva un salto incondizionato $C(1, 1, k)$, che ci riporta al passo k. Finché la condizione $r_m = r_n$ continua a essere soddisfatta si rimane nel ciclo. Nel caso invece la condizione non sia più soddisfatta si deve uscire dal ciclo, saltando direttamente all'istruzione successiva, di riga $k+4$, con un salto incondizionato $C(1, 1, k+4)$.



(a) Diagramma di flusso della frase **if-then-else**



(b) Diagramma di flusso della frase **while-do**



(c) Diagramma di flusso della frase **repeat-until**

Figura 2.16: Diagrammi di flusso delle frasi principali di selezione e di ciclo

$k-1$	I_{prec}	(esegue l'istruzione precedente)
k	$C(m,n,k+2)$	(controlla se $r_m = r_n$)
$k+1$	$C(1,1,k+4)$	(se $r_m \neq r_n$ si deve andare a I_{succ})
$k+2$	I	(se $r_m = r_n$ si esegue I)
$k+3$	$C(1,1,k)$	(salto incondizionato a I_k che chiude un ciclo)
$k+4$	I_{succ}	(esegue l'istruzione successiva)

Frase di iterazione **repeat** I **until** C

Subito dopo aver eseguito I_{prec} si esegue immediatamente anche la I. Poi si attua la verifica della condizione $r_m = r_n$?. Se la condizione non è soddisfatta si apre un ciclo con un'istruzione di salto incondizionato $C(1,1,k)$, che ci riporta alla riga k; finché $r_m \neq r_n$ si rimane nel ciclo eseguendo più volte la I. Quando la C diventa vera si esce dal ciclo, andando all'istruzione successiva.

$k-1$	I_{prec}	(esegue l'istruzione precedente)
k	I	(esegue I)
$k+1$	$C(m,n,k+3)$	(se $r_m = r_n$ si va a I_{succ})
$k+2$	$C(1,1,k)$	(se $r_m \neq r_n$ si apre un ciclo ritornando a I_k)
$k+3$	I_{succ}	(esegue l'istruzione successiva)

La costruzione dei programmi che sono il risultato della traduzione in linguaggio RAM delle tre frasi **if-then-else**, **while-do** e **repeat-until** ci dà la possibilità di creare il raccordo concettuale che esiste tra la scrittura di un programma ad alto livello e l'esecuzione delle corrispondenti istruzioni a seguito della compilazione nel linguaggio macchina.

Capitolo 3

Computabilità e decidibilità

3.1 Predicati e problemi decidibili

In matematica capita spesso di dover decidere se dei numeri godono o meno di una certa proprietà; un esempio potrebbe essere il seguente: fissati due interi x e y decidere se y è un multiplo di x. Costruire un programma per risolvere questo problema significa scrivere un elenco di istruzioni RAM le quali, sulla base di un certo algoritmo risolutivo, partendo dai dati iniziali x e y fornisca una risposta del tipo SI o NO come esito della computazione. Se conveniamo che SI corrisponda a 1 e NO a 0, il tutto si riconduce a computare la funzione seguente

$$f(x,y) = \begin{cases} 1 & \text{se } y \text{ è multiplo di } x \\ 0 & \text{se } y \text{ non è multiplo di } x \end{cases}$$

Di conseguenza possiamo dire che la proprietà o *predicato* 'y è multiplo di x' è effettivamente o *algoritmicamente decidibile*, o più semplicemente *decidibile*, se la funzione $f(x,y)$ è computabile. In generale, supponiamo che $M(\boldsymbol{x})$ sia un predicato n-ario sui numeri naturali $(x_1, x_2, ..., x_n) = \boldsymbol{x}$. La sua *funzione caratteristica* $C_M(\boldsymbol{x})$ si definisce nel modo seguente

$$C_M(\boldsymbol{x}) = \begin{cases} 1 & \text{se } M(\boldsymbol{x}) \text{ vale} \\ 0 & \text{se } M(\boldsymbol{x}) \text{ non vale} \end{cases} \tag{3.1}$$

Definizione 3.1. *Il predicato $M(\boldsymbol{x})$ è* decidibile *se la funzione $C_M(\boldsymbol{x})$ è computabile. $M(\boldsymbol{x})$ è* indecidibile *se $M(\boldsymbol{x})$ non è decidibile.*

Si noti che quando si parla di decidibilità (o indecidibilità) di un predicato, si ha sempre a che fare con la computabilità (o la non computabilità) di funzioni *totali*, poiché la funzione caratteristica è sempre definita.
Alcuni esempi di predicati decidibili sono '$x = y$', '$x \neq y$', '$x \neq 0$', '$x = 2$', 'x divide y', 'x multiplo di y' ecc. e nel linguaggio corrente essi sono spesso descritti come *problemi*; diremo quindi che i corrispondenti problemi sono decidibili. Per ciascuno di questi predicati non è infatti difficile escogitare un programma RAM che computi la corrispondente funzione caratteristica (si vedano gli esempi 2.8 e 2.9 a tal riguardo). In qualche caso (p.es. 'x multiplo di y') la scrittura potrebbe però essere molto lunga e tediosa; vedremo nei prossimi paragrafi un metodo più efficace per stabilire la decidibilità di un predicato, senza dover scrivere materialmente il programma che computa la corrispondente funzione caratteristica.

3.2 Computabilità su altri domini

Poiché il modello RAM tratta solo numeri naturali e tutte le funzioni con la quali abbiamo a che fare sono da $\mathbb{N}^n$ in $\mathbb{N}$, le definizione di computabilità e di decidibilità sono soggette allo stesso tipo di limitazione. Tuttavia queste nozioni sono facilmente estensibili anche ad altri domini (interi relativi, polinomi, matrici, ecc) purché si usi una opportuna *codifica*, secondo lo schema seguente.
Una *codifica* di un dominio $\mathcal{D}$ è una *iniezione* effettiva $\alpha : \mathcal{D} \to \mathbb{N}$; diciamo allora che un oggetto $d \in \mathcal{D}$ è *codificato* dal numero naturale $\alpha(d)$. Supponiamo ora che f sia una funzione da $\mathcal{D}$ a $\mathcal{D}$; allora f è naturalmente codificata da una funzione f^* da $\mathbb{N}$ a $\mathbb{N}$, la quale mappa la codifica di un oggetto $d \in Dom(f)$ alla codifica di $f(d)$. Esplicitamente abbiamo

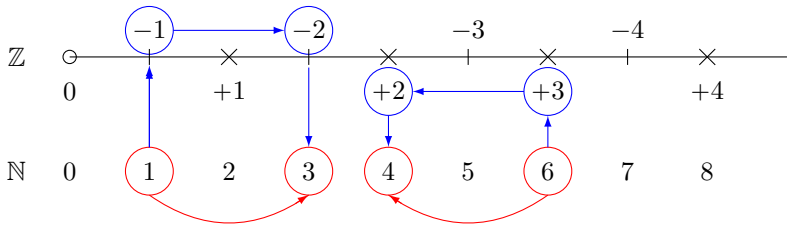
$$f : \mathcal{D} \longrightarrow \mathcal{D} \qquad \mathbb{N} \xrightarrow{\alpha^{-1}} \mathcal{D} \xrightarrow{f} \mathcal{D} \xrightarrow{\alpha} \mathbb{N} \quad (f^*) \qquad f^* : \mathbb{N} \longrightarrow \mathbb{N} \qquad f^* = \alpha \cdot f \cdot \alpha^{-1}$$

Con questa premessa è ora semplice estendere la definizione di computabilità al dominio $\mathcal{D}$, dicendo che f è computabile su $\mathcal{D}$ se f^* è computabile su $\mathbb{N}$.

Esempio 3.1. Sia $\mathcal{D} = \mathbb{Z}$ e consideriamo la funzione $f(x) = x - 1$ su $\mathbb{Z}$. Una possibile codifica per x è la seguente

$$\alpha(x) = \begin{cases} 2x & \text{se } x \geqslant 0 \\ -2x - 1 & \text{se } x < 0 \end{cases} \qquad \alpha^{-1}(n) = \begin{cases} n/2 & \text{se } n \text{ è pari} \\ -(n+1)/2 & \text{se } n \text{ è dispari} \end{cases}$$

Questa codifica corrisponde a un ribaltamento del semiasse negativo delle ascisse di $\mathbb{Z}$ sopra quello positivo, intercalando numeri negativi e positivi. Ciò porta alla seguente situazione, per quanto riguarda la corrispondenza tra elementi di $\mathbb{Z}$ ed elementi di $\mathbb{N}$:



Per costruire la funzione f^* dobbiamo distinguere tra valori pari e dispari di $\mathbb{N}$. Prendiamo p.es. il valore pari 6 in $\mathbb{N}$; nella codifica esso corrisponde a $+3$ in $\mathbb{Z}$, e sappiamo che $+3 - 1 = +2$ in $\mathbb{Z}$, che ha come codifica 4 in in $\mathbb{N}$. Ecco allora che nella funzione f^* a 6 deve corrispondere 4. Ciò significa che $f^* = n - 2$ con n pari. Prendiamo ora un numero dispari su $\mathbb{N}$, p.es. 1; a esso corrisponde -1 su $\mathbb{Z}$, e $-1 - 1 = -2$ su $\mathbb{Z}$, cui corrisponde 3 su $\mathbb{N}$. Ecco allora che $f^* = n + 2$ con n dispari. Il caso dello 0 su $\mathbb{N}$ rimane isolato e a 0 si fa corrispondere 1. Ricapitolando la funzione $f^* : \mathbb{N} \longrightarrow \mathbb{N}$ che si ottiene è la seguente

$$f^*(n) = \begin{cases} 1 & \text{se } n = 0 \\ n - 2 & \text{se } n > 0 \text{ pari} \\ n + 2 & \text{se } n > 0 \text{ dispari} \end{cases} \tag{3.2}$$

Osservazione 3.1. [1] Abbiamo definito la computabilità di una funzione $f : \mathcal{D} \longrightarrow \mathcal{D}$, ma non quella di una funzione $f : \mathcal{D} \longrightarrow \mathcal{E}$, con $\mathcal{D}$ ed $\mathcal{E}$ domini diversi. In tal caso potremmo estendere la definizione introducendo due codifiche $\alpha : \mathcal{E} \to \mathbb{N}$ e $\beta : \mathcal{D} \to \mathbb{N}$, affermando che la f è computabile se lo è la funzione $f^* = \alpha \cdot f \cdot \beta^{-1}$.

3.3 Insieme delle funzioni computabili

Lo scopo dell'analisi che seguirà nei prossimi capitoli è quello di sondare i limiti del metodo procedurale-algoritmico, cioè se esistono problemi (di natura formale) che *non* sono risolubili usando un approccio algoritmico, cioè usando il modello RAM, che sarà d'ora in poi il nostro riferimento per questo tipo di analisi. Sulla base di quanto detto precedentemente è ovvio che un eventuale problema che si dovesse palesare come *non* risolubile nel modello RAM, risulterà non risolubile in ciascuno degli altri modelli alternativi fra quelli rappresentati in tabella 1.13; ciò poiché essi sono tra loro equivalenti nel senso che esprimono la stessa "potenza" computazionale.
Nel nostro ambito parlare di problemi *non* risolubili significa riferirsi a predicati non decidibili, quindi a *funzioni non computabili*; sulla base della definizione 2.3 ciò significa che *non* esiste un programma che la RAM-calcola. Per contro la tesi di Church-Turing, descritta nel paragrafo 1.3, ci dice che tutto ciò che è intuitivamente ed effettivamente computabile lo è nel senso della RAM-computabilità. Ecco allora che dovremo indagare sui seguenti problemi:

1. Quante sono le funzioni computabili? Sono in numero finito o infinito?
2. In che rapporto stanno col numero di *tutte* le possibili funzioni da $f : \mathbb{N}^n \longrightarrow \mathbb{N}$?
3. Esistono funzioni non computabili?
4. Se sì, quante sono?

Il primo punto da affrontare è allora quello di capire quante sono le funzioni computabili.
Poiché, a norma della definizione 2.3, una funzione è computabile se esiste un programma che la RAM-computa, il nostro compito appare alquanto arduo, poiché sembra che si debba procedere come fatto negli esempi 2.1-2.9 del capitolo precedente, cioè dettagliare in modo esplicito un programma per ogni funzione per la quale si vuole dimostrare la computabilità. Un simile procedimento è chiaramente inattuabile; si capisce subito che anche funzioni molto semplici richiederebbero uno sforzo notevole per produrre un programma, che potrebbe essere anche molto lungo e complicato. E' però possibile usare una strategia diversa, che consiste nel costruire funzioni computabili *combinando in modo computabile* altre funzioni computabili più semplici; ciò corrisponde a una prova implicita della computabilità di una funzione, senza la necessità di scrivere concretamente il programma che la RAM-calcola. Studieremo nei prossimi paragrafi i seguenti metodi procedurali di estensione:

Sostituzione - E' la classica sostituzione di funzioni definita in Analisi.

Ricorsione - E' lo strumento più potente e versatile per costruire nuove funzioni a partire da funzioni precedenti.

Minimazione illimitata - E' uno strumento molto più sofisticato, che fu necessario introdurre per catturare anche certi tipi molto speciali di funzioni palesemente calcolabili, ma non ottenibili con l'impiego delle sole sostituzione e ricorsione.

Il nucleo di partenza delle funzioni computabili che useremo come supporto per l'estensione tramite sostituzione, ricorsione e minimazione, viene chiamato *insieme delle funzioni di base*; esso è definito implicitamente dal seguente lemma

Lemma 3.1. *Le seguenti funzioni base sono computabili:*

1. *la funzione nulla* $\mathbf{0}(x) = 0, \ \forall x$;
2. *la funzione successore* $f(x) = x + 1$;

3. *la funzione di proiezione* $U_i^n(x_1, x_2, ..., x_n) = x_i$, *definita per ciascun* $n \geqslant 1$ *e* $1 \leqslant i \leqslant n$.

Dimostrazione. Per dimostrare la computabilità di ciascuna funzione base è sufficiente costruire un programma che la calcola:

1. il programma per calcolare $\mathbf{0}(x)$ è $Z(1)$;
2. il programma per calcolare $x + 1$ è $S(1)$;
3. il programma per calcolare U_i^n è $T(i, 1)$;

In pratica queste funzioni corrispondono all'impiego delle istruzioni aritmetiche $Z(n), S(n), T(m, n)$ del modello RAM. □

Prima di iniziare al descrizione di questi tre strumenti di estensione dell'insieme delle funzioni computabili, dobbiamo però mettere a posto alcune questioni tecnico-pratiche, che se non trattate correttamente potrebbero creare confusione e difficoltà.

Concatenazione di programmi

Nel seguito capiterà spesso di dover concatenare due programmi P e Q, cioè costruire un programma PQ nel quale prima si eseguono le istruzioni di P e successivamente quelle di Q. Se vogliamo che tutto funzioni come previsto, dobbiamo fare attenzione ad alcuni dettagli tecnici.
Sia $P = I_1, I_2, ..., I_s$; una computazione su P si conclude (se si conclude) nel momento in cui si deve eseguire un'istruzione I_v con $v > s$ per qualche valore di v. Se vogliamo che a seguito dell'esecuzione di P venga eseguita la prima istruzione di Q, dobbiamo fare in modo che sia $v = s + 1$. Poiché l'unica istruzione che può creare problemi è la $C(m, n, q)$, nella costruzione di programmi concatenati dobbiamo far riferimento ai soli programmi nei quali in ogni istruzione di salto condizionato si ha $q \leqslant s + 1$. Ciò porta alla seguente

Definizione 3.2. *Un programma* $P = I_1, I_2, ..., I_s$ *si dice in* forma standard *se per ogni istruzione di salto condizionato* $C(m, n, q)$ *si ha* $q \leqslant s + 1$.

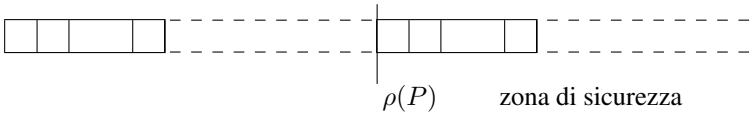
D'ora in poi useremo solamente programmi in forma standard.
Supponiamo ora di dover concatenare due programmi P e Q che sono in forma standard. Un'istruzione di salto condizionato $C(m, n, q)$ di Q, che faccia saltare all'istruzione I_q se $r_m = r_n$, dovrà far saltare all'istruzione I_{q+s} se Q diventa parte della concatenazione PQ; ciò significa che per far funzionare in modo corretto la concatenazione, dovremo sostituire tutte le istruzioni del tipo $C(m, n, q)$ di Q con istruzioni del tipo $C(m, n, q + s)$ di PQ. Otteniamo allora la seguente

Definizione 3.3. *Siano* P *e* Q *due programmi in forma standard, di lunghezza rispettivamente* s *e* t. *La* concatenazione PQ *è il programma* $I_1, I_2, ..., I_s, I_{s+1}, I_{s+2}, ..., I_{s+t}$, *dove* $I_1, I_2, ..., I_s$ *sono le istruzioni di* P *mentre* $I_{s+1}, I_{s+2}, ..., I_{s+t}$ *sono le istruzioni di* Q *nelle quali ogni istruzione di salto* $C(m, n, q)$ *è sostituita dall'istruzione* $C(m, n, q+s)$.

Con queste premesse è ora chiaro che la computazione della concatenazione PQ corrisponde alla computazione su P seguita dalla computazione su Q; si ha inoltre che la configurazione iniziale della computazione su Q corrisponde alla configurazione finale della computazione su P, come richiesto dal concetto di concatenazione di programmi.

Zona di sicurezza

Supponiamo ora di voler costruire un programma Q che abbia P come *subroutine*. In tal caso è necessario individuare una zona di memoria, che non venga modificata dall'esecuzione di P, dove poter memorizzare in sicurezza i dati che dovranno successivamente essere usati da Q alla fine della computazione P. Ciò può essere ottenuto nel modo seguente.
Poiché P ha lunghezza finita, esiste un valore minimo u tale che nessun registro R_v, con $v > u$, viene menzionato in P; in altre parole, se $Z(n)$ o $S(n)$ o $T(m,n)$ o $C(m,n,q)$ è un'istruzione di P, allora $m, n \leqslant u$. Ciò significa che la computazione su P non potrà modificare il contenuto di tutti i registri R_v, con $v > u$; ma vale anche il viceversa: il contenuto dei registri in questione non potrà avere alcun effetto sulla computazione associata a P. In tal modo, nello scrivere il programma Q, potremo memorizzare informazioni importanti e che non devono essere cancellate nella zona di memoria R_v per cui $v > u$; queste informazioni non interferiranno con la *subroutine* P. Denotiamo il numero u con $\rho(P)$. La zona R_v per cui $v > \rho(P)$ è chiamata *zona di sicurezza*.



Notazione di traslazione

L'ultimo dettaglio tecnico che dobbiamo affrontare, prima di definire e usare le procedure di estensione, è di natura notazionale. Supponiamo che P sia un programma in forma standard, che serve a calcolare la funzione $f(x_1, x_2, ..., x_n)$. Se dobbiamo usare P come *subroutine* di un programma Q, potrebbe capitare che i valori iniziali $x_1, x_2, ..., x_n$ con i quali si deve innescare la computazione su P siano memorizzati nelle celle $R_{l_1}, R_{l_2}, ..., R_{l_n}$, invece che nelle prime n celle $R_1, R_2, ..., R_n$, come prevede la convenzione che regola la computazione. Inoltre potrebbe essere necessario depositare l'esito $f(x_1, x_2, ..., x_n)$ della computazione su P nella cella R_l, a disposizione per le successive computazioni, invece che in R_1, come prevede la convenzione. Un'ultimo problema è il seguente: poiché la computazione su P è una *subroutine* del programma Q, potrebbe accadere che le celle $R_1, R_2, ..., R_{\rho(P)}$ siano state alterate dalle precedenti computazioni di Q, e contengano dati non voluti e non conformi alla convenzione usata per la condizione iniziale della computazione su P, che richiede $x_1, x_2, ..., x_n$ nelle prime n celle e 0 in tutte le rimanenti. Per risolvere tutti questi problemi dobbiamo attuare la procedura di figura 3.1, per la quale usiamo la notazione $P[l_1, l_2, ..., l_n \to l]$. L'effetto è quello di computare la funzione $f(x_1, x_2, ..., x_n)$ a partire dalle celle di memoria $R_{l_1}, R_{l_2}, ..., R_{l_n}$ e di trasferire il valore finale della stessa nella cella R_l.

Estensione delle funzioni mediante sostituzione

Un metodo classico per costruire nuove funzioni a partire da funzioni già definite è quello di ricorrere alla composizione o *sostituzione* di funzioni. Se riusciamo a dimostrare che la sostituzione è esprimibile in modo procedurale, allora essa sarà computabile per la tesi di Church-Turing. La conseguenza è la seguente: se applichiamo la sostituzione, che è computabile, a funzioni computabili, le nuove funzioni che otterremo saranno pur esse computabili. Useremo questo approccio anche per ricorsione e minimazione illimitata.
Nel seguito andremo oltre la tesi di Church-Turing, poiché scriveremo in modo esplicito un programma per computare rispettivamente sostituzione, ricorsione e minimazione.

Osservazione 3.2. La computabilità delle funzioni ottenute mediante il procedimento sopra descritto viene certificata solo in modo indiretto, poiché non è più necessario scrivere esplicitamente un programma che RAM-calcola la funzione.

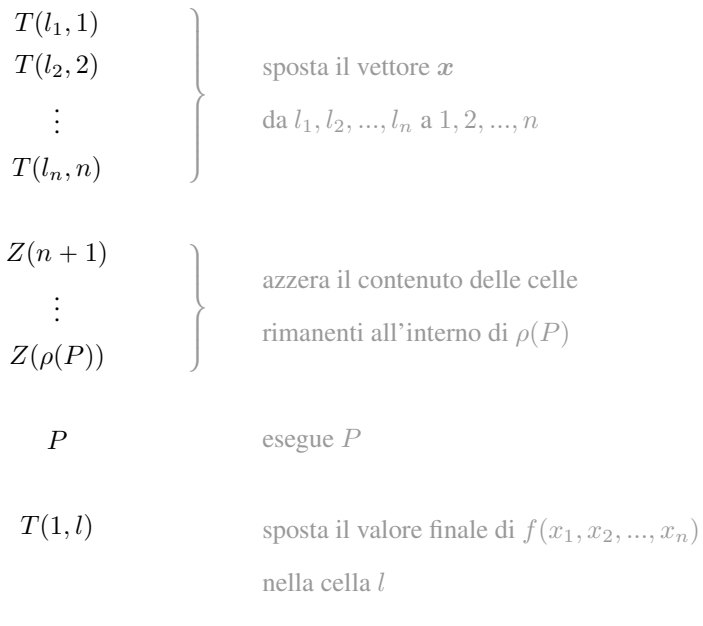


Figura 3.1: Procedura $P[l_1, l_2, ..., l_n \to l]$

Nel teorema seguente dimostriamo la computabilità della sostituzione, e cioè che l'insieme $\mathcal{C}$ è chiuso rispetto alla sostituzione. Nel seguito faremo uso della notazione $\alpha(x) \simeq \beta(x)$ per indicare che, per ogni x, $\alpha(x)$ e $\beta(x)$ sono entrambe definite o entrambe non definite, e se sono definite hanno lo stesso valore.

Teorema 3.1. *Siano $f(y_1, y_2, ..., y_k)$ e $g_1(\boldsymbol{x}), g_2(\boldsymbol{x}), ..., g_k(\boldsymbol{x})$ funzioni computabili, con $\boldsymbol{x} = (x_1, x_2, ..., x_n)$. Allora la funzione*

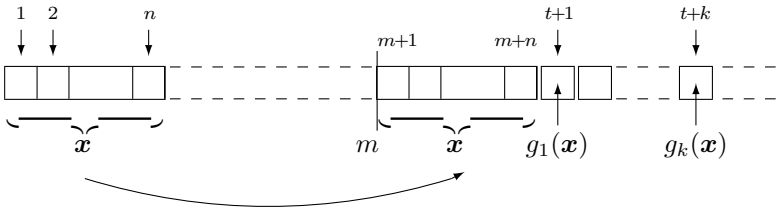
$$h(\boldsymbol{x}) \simeq f(g_1(\boldsymbol{x}), g_2(\boldsymbol{x}), ..., g_k(\boldsymbol{x}))$$

è computabile.

Dimostrazione. Si osservi anzitutto che $h(\boldsymbol{x})$ è definita se e solo se $g_1(\boldsymbol{x}), g_2(\boldsymbol{x}), ..., g_k(\boldsymbol{x})$ sono tutte definite e $(g_1(\boldsymbol{x}), g_2(\boldsymbol{x}), ..., g_k(\boldsymbol{x})) \in Dom(f)$. Se $f, g_1, g_2, ..., g_k$ sono tutte funzioni totali, allora h è totale.
Poiché $f, g_1, g_2, ..., g_k$ sono computabili, esistono i programmi $F, G_1, G_2, ..., G_k$ che le calcolano. Scriveremo un programma H che incorpora la seguente naturale procedura di calcolo della $h(\boldsymbol{x})$: '*Assegnato $\boldsymbol{x}$ usiamo i programmi $G_1, G_2, ..., G_k$ in sequenza per calcolare rispettivamente $g_1(\boldsymbol{x}), g_2(\boldsymbol{x}), ..., g_k(\boldsymbol{x})$. Alla fine usiamo il programma F per calcolare $f(g_1(\boldsymbol{x}), g_2(\boldsymbol{x}), ..., g_k(\boldsymbol{x}))$*'.
Per evitare di perdere informazione utile relativa ai vari stadi della computazione, dovremo conservare $\boldsymbol{x}$ e i valori $g_1(\boldsymbol{x}), g_2(\boldsymbol{x}), ..., g_k(\boldsymbol{x})$, man mano che vengono generati, in zona di sicurezza; essa viene definita fissando il massimo valore dell'indirizzo di memoria affetto da computazioni di uno qualunque dei programmi che dovremo usare, e naturalmente dai valori n e k

$$m = \max(n, k, \rho(F), \rho(G_1), \rho(G_2), ..., \rho(G_k))$$

La zona di sicurezza parte allora dal valore $m + 1$. Conserveremo $\boldsymbol{x}$ nei registri $R_{m+1}, R_{m+2}, ..., R_{m+n}$ e i valori $g_i(\boldsymbol{x})$ nelle celle R_{t+i}, con $t = m+n$ e $1 \leqslant i \leqslant k$. La procedura per computare h prevede allora di copiare $\boldsymbol{x}$ in zona di sicurezza e di calcolare uno dopo l'altro tutti i valori di $g_i(\boldsymbol{x})$, usando i programmi G_i; una volta avuti a disposizione i valori $g_i(\boldsymbol{x})$ per $1 \leqslant i \leqslant k$, si può usare F per calcolare h. In figura 3.2 possiamo vedere la procedura che calcola h.
Si osservi l'uso della notazione $G_i[m + 1, m + 2, ..., m + n \longrightarrow t + i]$ introdotta precedentemente, che prende



$T(1, m+1)$
$T(2, m+2)$
$\vdots$
$T(n, m+n)$

sposta il vettore x in zona di sicurezza

$G_1[m+1, m+2, ..., m+n \longrightarrow t+1]$
$\vdots$
$G_k[m+1, m+2, ..., m+n \longrightarrow t+k]$

calcola $g_i(x)$ e pone il risultato in zona di sicurezza

$F[t+1, t+2, ..., t+k \longrightarrow 1]$

calcola $f(g_1(x), g_2(x), ..., g_k(x))$

Figura 3.2: Programma per RAM-calcolare la sostituzione

come dati in ingresso dei valori che sono memorizzati in una zona diversa da quella prevista dalla convenzione (cioè le prime n celle) e deposita il risultato finale in una cella diversa dalla R_1. Si faccia attenzione al fatto che la procedura illustrata in figura 3.2 è un programma vero e proprio, poiché ogni elemento del tipo $G_i[m+1, m+2, ..., m+n \longrightarrow t+i]$ costituisce una vera e propria *subroutine*. Dunque il programma 3.2 RAM-calcola la funzione h e di conseguenza h è computabile. Resta implicito il fatto che la computazione $H(x)$ terminerà se e solo se termineranno ciascuna delle computazioni $G_i(x)$ $(1 \leqslant i \leqslant k)$ e la $F(x)$. □

Dal teorema 3.1 ricaviamo subito il seguente

Corollario 3.1. *Sia $f(y_1, y_2, ..., y_k)$ una funzione computabile e $x_{i_1}, x_{i_2}, ..., x_{i_k}$ una sequenza di k variabili (anche ripetute) prese dal vettore $x = (x_1, x_2, ..., x_n)$. Allora la funzione*

$$h(x_1, x_2, ..., x_n) \simeq f(x_{i_1}, x_{i_2}, ..., x_{i_k})$$

è computabile.

Dimostrazione. La funzione $h(x_1, x_2, ..., x_n) = h(x)$ può essere espressa nel modo seguente

$$h(x) \simeq f(U_{i_1}(x), U_{i_2}(x), ..., U_{i_k}(x))$$

che è computabile per il punto 3 del lemma 3.1 (computabilità della proiezione) e per la computabilità della sostituzione espressa dal teorema 3.1. □

La computabilità della sostituzione e il corollario 3.1 ci consentono di costruire nuove funzioni mediante riarrangiamento, identificazione di variabili o introduzione di variabili fittizie, e ciò anche nei casi in cui le funzioni $g_1(x), g_2(x), ..., g_k(x)$, sostituite in f, non sono necessariamente dipendenti da tutte le variabili $x_1, x_2, ..., x_n$. Ecco alcuni esempi, nei quali si suppone che la computabilità della somma sia già stata acquisita.

Esempio 3.2.

$$h_1(x_1, x_2) \simeq f(x_2, x_1) \qquad \text{riarrangiamento}$$

$$f(y_1, y_2) = y_1 + y_2 \qquad k = 2 \quad n = 2$$
$$U_2(\boldsymbol{x}) = x_2$$
$$U_1(\boldsymbol{x}) = x_1$$

$$h_1(x_1, x_2) \simeq f(U_2(\boldsymbol{x}), U_1(\boldsymbol{x})) = x_2 + x_1$$

Esempio 3.3.

$$h_2(x_1, x_2) \simeq f(x_1, x_1) \qquad \text{identificazione}$$

$$f(y_1, y_2) = y_1 + y_2 \qquad k = 2 \quad n = 2$$
$$U_1(\boldsymbol{x}) = x_1$$
$$U_1(\boldsymbol{x}) = x_1$$

$$h_2(x) \simeq f(U_1(\boldsymbol{x}), U_1(\boldsymbol{x})) = x_1 + x_1 = 2x_1$$

Esempio 3.4.

$$h_3(x_1, x_2, x_3) \simeq f(x_2, x_3) \qquad \text{variabile fittizia}$$

$$f(y_1, y_2) = y_1 + y_2 \qquad k = 2 \quad n = 3$$
$$U_2(\boldsymbol{x}) = x_2$$
$$U_3(\boldsymbol{x}) = x_3$$

$$h_3(x_1, x_2, x_3) \simeq f(U_2(\boldsymbol{x}), U_3(\boldsymbol{x})) = x_2 + x_3$$

Esempio 3.5.

$$h_4(x_1, x_2, x_3) = x_1 + x_2 + x_3 \qquad \text{estensione della somma}$$

$$f(y_1, y_2) = y_1 + y_2 \qquad k = 2 \quad n = 3$$
$$g_1(x_1, x_2, x_3) = x_1 + x_2$$
$$g_2(x_1, x_2, x_3) = x_3$$

$$h_3(x_1, x_2, x_3) \simeq f(g_1(\boldsymbol{x}), g_2(\boldsymbol{x})) = x_1 + x_2 + x_3$$

Estensione delle funzioni mediante ricorsione

La ricorsione è uno strumento molto potente e versatile per costruire nuove funzioni a partire da funzioni già definite. La sua particolarità è che ciascun valore della nuova funzione viene definito in modo iterativo sulla base di valori precedentemente definiti della funzione stessa, in una sorta di annidamento che si esaurisce nel momento in cui viene fissato il valore d'innesco della funzione al passo iniziale.

Definizione 3.4. *Siano che $f(\boldsymbol{x})$ e $g(\boldsymbol{x}, y, z)$ funzioni, non necessariamente totali. La funzione h è definita per ricorsione su f e g nel modo seguente:*

$$\begin{aligned} &(i) \quad h(\boldsymbol{x}, 0) \simeq f(\boldsymbol{x}) \\ &(ii) \quad h(\boldsymbol{x}, y+1) \simeq g(\boldsymbol{x}, y, h(\boldsymbol{x}, y)) \end{aligned} \tag{3.3}$$

La definizione di un oggetto basata sull'oggetto stesso può suscitare qualche perplessità, ma è proprio su questa introflessione che si basa l'efficacia e la potenza del metodo. Lo schema da seguire è il seguente: se vogliamo conoscere p.es. il valore di $h(\boldsymbol{x}, 3)$, basandoci sulla (ii) dobbiamo conoscere il valore di $h(\boldsymbol{x}, 2)$ e inserirlo in g; ma utilizzando nuovamente la (ii) si vede che per ricavare $h(\boldsymbol{x}, 2)$ bisogna conoscere $h(\boldsymbol{x}, 1)$; l'ultima interazione con la (ii) ci chiede la conoscenza di $h(\boldsymbol{x}, 0)$; è questo il seme d'innesco dell'annidamento ricorsivo, poiché in questo caso la (i) definisce il valore di $h(\boldsymbol{x}, 0)$, che è pari al valore di $f(\boldsymbol{x})$. Conoscendo $h(\boldsymbol{x}, 0)$ ora possiamo risalire nell'annidamento, ricavando successivamente $h(\boldsymbol{x}, 1)$, $h(\boldsymbol{x}, 2)$ e $h(\boldsymbol{x}, 3)$.

Osservazione 3.3. La ricorsione descritta dalla definizione 3.4 riguarda la sola variabile y. Nella sostanza questa variable deve intendersi come la $(n+1)$-esima coordinata del vettore $\boldsymbol{x}$, che viene chiamata y solo per evidenziarla più chiaramente. La ricorsione definita su una singola coordinata del vettore viene detta anche *ricorsione primitiva*. Vedremo nel seguito che esistono forme più sofisticate di ricorsione, non descrivibili mediante ricorsione primitiva.

Osservazione 3.4. A meno che f e g siano entrambe totali, non è detto che h sia totale. Accade infatti che

$$\begin{array}{lll} (\boldsymbol{x}, 0) \in Dom(h) & \text{sse} & \boldsymbol{x} \in Dom(f) \\ (\boldsymbol{x}, y+1) \in Dom(h) & \text{sse} & (\boldsymbol{x}, y) \in Dom(h) \\ & & (\boldsymbol{x}, y, h(\boldsymbol{x}, y)) \in Dom(g) \end{array}$$

La funzione che costituisce un tipico esempio di ricorsione è il fattoriale; pensandoci bene ci si accorge però che la maggior parte delle funzioni aritmetiche elementari sono definibili in modo ricorsivo. Ecco alcuni esempi.

Esempio 3.6.

Fattoriale $h(y) \simeq y!$

(i) $h(0) \simeq 0! = 1$

(ii) $h(y+1) \simeq h(y)(y+1)$

Somma $h(x, y) \simeq x + y$

(i) $h(x, 0) \simeq x + 0 = x$

(ii) $h(x, y+1) \simeq x + (y+1) = (x+y) + 1 = h(x, y) + 1$

Prodotto $h(x, y) \simeq x \cdot y$

(i) $h(x, 0) \simeq x \cdot 0 = 0$

(ii) $h(x, y+1) \simeq x \cdot (y+1) = x \cdot y + x = h(x, y) + x$

Potenza $h(x, y) \simeq x^y$

(i) $h(x, 0) \simeq x^0 = 1$

(ii) $h(x, y+1) \simeq x^{y+1} = x^y \cdot x = h(x, y) \cdot x$

Superpotenza $h(x, y) \simeq \left. x^{x^{x^{\cdot^{\cdot^{x}}}}} \right\} y$ volte

(i) $h(x, 0) \simeq \left. x^{x^{x^{\cdot^{\cdot^{x}}}}} \right\} 0 = 1$

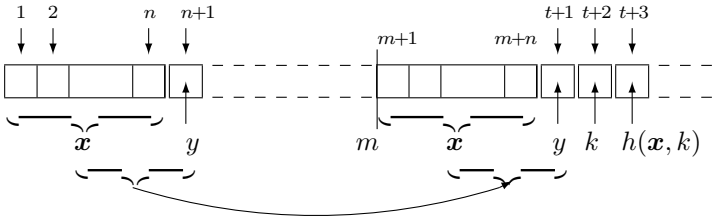
(ii) $h(x, y+1) \simeq \left. x^{x^{x^{\cdot^{\cdot^{x}}}}} \right\} (y+1) = x^{\left. x^{x^{\cdot^{\cdot^{x}}}} \right\} y} = x^{h(x,y)}$

L'idea dell'annidamento ricorsivo appare ben evidente in questi esempi, nei quali appare chiaro che una funzione più complessa (p.es. il prodotto) viene costruita iterando la funzione più semplice (la somma) tramite la funzione g di supporto. Ecco allora che la somma tra x e y si costruisce sommando "1" y volte, con una funzione $g(x, y, z) = z + 1$; il prodotto si ottiene sommando "x" y volte, con una funzione $g(x, y, z) = z + x$; mentre la

potenza si ottiene moltiplicando "x" y volte, con una funzione $g(x, y, z) = z \cdot x$, e così via.
Anche nel caso della ricorsione siamo in grado di costruire una procedura effettiva che la calcola, stabilendo così il seguente

Teorema 3.2. *Siano $f(\boldsymbol{x})$ e $g(\boldsymbol{x}, y, z)$ funzioni computabili, con $\boldsymbol{x} = (x_1, x_2, ..., x_n)$. Allora la funzione $h(\boldsymbol{x}, y)$ definita per ricorsione come nella definizione 3.4 è computabile.*

Dimostrazione. Siano F e G programmi in forma standard per il calcolo di f e g. Scriveremo un programma H per il calcolo della funzione $h(\boldsymbol{x}, y)$ definita per ricorsione dall'equazione (3.3).
Il programma H segue la naturale procedura già descritta in precedenza: si parte dal calcolo di $h(\boldsymbol{x}, 0)$, che si basa sulla computazione di $f(\boldsymbol{x})$ tramite F; poi si calcolano successivamente $h(\boldsymbol{x}, 1)$, $h(\boldsymbol{x}, 2)$, ... , $h(\boldsymbol{x}, k)$ usando G associato alla funzione g; la variabile k è di supporto e serve a controllare la condizione di stop $k = y$.
Fissiamo la zona di sicurezza ponendo $m = \max(n + 2, \rho(F), \rho(G))$, dove il valore $n + 2$ si riferisce all'arietà della funzione $g(\boldsymbol{x}, y, z)$. La prima operazione da fare è quella di memorizzare $\boldsymbol{x}$ e y in zona di sicurezza, cioè $\boldsymbol{x}$ nei registri $R_{m+1}, R_{m+2}, ..., R_{m+n}$ e y nel registro R_{t+1} (poniamo $t = m + n$ per alleggerire la notazione).



Il programma H che calcola la ricorsione è allora il seguente: □

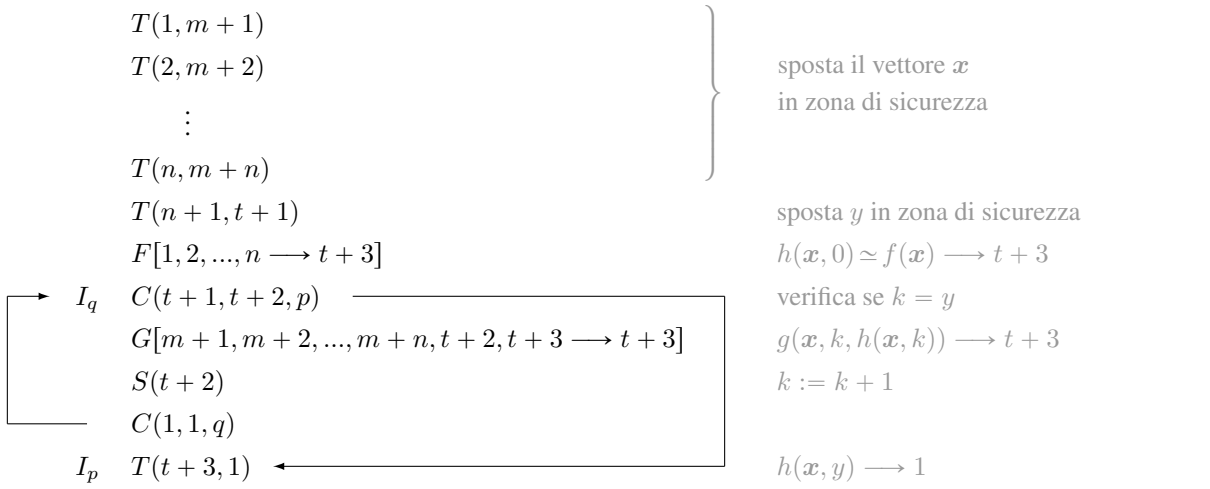


Figura 3.3: Programma per RAM-calcolare la ricorsione

Esempi di funzioni ottenibili mediante sostituzione e ricorsione

L'impiego sinergico dei teoremi 3.2 e 3.3 ci consente di ampliare in modo efficiente e sistematico l'insieme delle funzione RAM-calcolabili a partire dalle funzioni base descritte nel lemma 3.1; inoltre, la computabilità della

sostituzione e della ricorsione ci consente di qualificare computabile una funzione derivata dall'applicazione di queste due tecniche di estensione *senza* dover scrivere effettivamente un programma che le RAM calcola. L'elenco di possibili funzioni che possiamo ottenere sembra potersi estendere all'infinito, perchè come già visto con gli esempi 3.6 relativi alla ricorsione, ogni nuova funzione costituisce un supporto sul quale poter basare la costruzione di un'ulteriore funzione. Vediamo un po' di esempi di uso congiunto di sostituzione e ricorsione.

Teorema 3.3. *Le seguenti funzioni sono computabili:*

(a)	*Somma*	$x + y$	*dim.* si veda l'esempio 3.6
(b)	*Prodotto*	$x \cdot y$	*dim.* si veda l'esempio 3.6
(c)	*Potenza*	x^y	*dim.* si veda l'esempio 3.6
(d)	*Decremento naturale*	$x \dot{-} 1$	*dim.* $0 \dot{-} 1 = 0$
(e)	*Differenza naturale*	$x \dot{-} y$	$(x+1) \dot{-} 1 = x$; per ricorsione $= \begin{cases} x - y & \textit{se } x \geqslant y \\ 0 & \textit{altrimenti} \end{cases}$ *dim.* $x \dot{-} 0 = x$ $x \dot{-} (y+1) = (x \dot{-} y) \dot{-} 1$; per ricorsione e la (*d*)
(f)	*Signum*	$\mathrm{sg}(x)$	$= \begin{cases} 0 & \textit{se } x = 0 \\ 1 & \textit{se } x \neq 0 \end{cases}$ *dim.* $\mathrm{sg}(0) = 0$ $\mathrm{sg}(x+1) = 1$; per ricorsione
(g)	*Signum complementare*	$\overline{\mathrm{sg}}(x)$	$= \begin{cases} 1 & \textit{se } x = 0 \\ 0 & \textit{se } x \neq 0 \end{cases}$ *dim.* $\overline{\mathrm{sg}}(x) = 1 \dot{-} \mathrm{sg}(x)$; per sostituzione, (*e*) e (*f*)
(h)	*Valore assoluto differenza*	$\lvert x - y \rvert$	*dim.* $\lvert x - y \rvert = (x \dot{-} y) + (y \dot{-} x)$; per sostituzione, (*a*) ed (*e*)
(i)	*Fattoriale*	$x!$	*dim.* si veda l'esempio 3.6
(j)	*Minimo*	$\min(x, y)$	$=$ *minimo tra* x *e* y *dim.* $\min(x, y) = x \dot{-} (x \dot{-} y)$; per sostituzione
(k)	*Massimo*	$\max(x, y)$	$=$ *massimo tra* x *e* y *dim.* $\max(x, y) = x + (y \dot{-} x)$; per sostituzione
(l)	*Resto della divisione*	$\mathrm{rm}(x, y)$	$=$ *resto della divisione tra* x *e* y *dim.* omessa; si veda [5] p.36
(m)	*Quoziente della divisione*	$\mathrm{qt}(x, y)$	$=$ *quoziente della divisione tra* x *e* y *dim.* omessa; si veda [5] p.37
(n)	*Divisibilità*	$\mathrm{div}(x, y)$	$= \begin{cases} 1 & \textit{se } x \mid y \\ 0 & \textit{se } x \nmid y \end{cases}$ *dim.* $\mathrm{div}(x, y) = \overline{\mathrm{sg}}(\mathrm{rm}(x, y))$; per sostituzione

Quelli che seguono sono utili corollari riguardanti predicati decidibili.

Corollario 3.2. *Siano* $f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), ..., f_k(\boldsymbol{x})$ *funzioni computabili e* $M_1(\boldsymbol{x}), M_2(\boldsymbol{x}), ..., M_k(\boldsymbol{x})$ *predicati decidi-*

bili, tali che per ogni $\boldsymbol{x}$ valga esattamente uno dei $M_1(\boldsymbol{x}), ..., M_k(\boldsymbol{x})$. Allora la funzione

$$g(\boldsymbol{x}) = \begin{cases} f_1(\boldsymbol{x}) & \textit{se } M_1(\boldsymbol{x}) \textit{ vale} \\ f_2(\boldsymbol{x}) & \textit{se } M_2(\boldsymbol{x}) \textit{ vale} \\ \vdots & \vdots \\ f_k(\boldsymbol{x}) & \textit{se } M_k(\boldsymbol{x}) \textit{ vale} \end{cases}$$

è computabile.

Dimostrazione. La $g(\boldsymbol{x})$ può essere scritta $g(\boldsymbol{x}) = C_{M_1}(\boldsymbol{x})f_1(\boldsymbol{x}) + C_{M_2}(\boldsymbol{x})f_2(\boldsymbol{x}) + ... + C_{M_k}(\boldsymbol{x})f_k(\boldsymbol{x})$, che è computabile per sostituzione e per la computabilità di somma e prodotto. □

Corollario 3.3. (**Algebra di decidibilità**) *Siano $M(\boldsymbol{x})$ e $Q(\boldsymbol{x})$ predicati decidibili. Allora lo sono anche i seguenti predicati*

(a) $'not\ M(\boldsymbol{x})'$
(b) $'M(\boldsymbol{x})$ AND $Q(\boldsymbol{x})'$
(c) $'M(\boldsymbol{x})$ OR $Q(\boldsymbol{x})'$

Dimostrazione. Le funzioni caratteristiche dei predicati sono le seguenti

(a) $'not\ M(\boldsymbol{x})'$: $1 \dot{-} C_M(\boldsymbol{x})$
(b) $'M(\boldsymbol{x})$ AND $Q(\boldsymbol{x})'$: $C_M(\boldsymbol{x})C_Q(\boldsymbol{x})$
(c) $'M(\boldsymbol{x})$ OR $Q(\boldsymbol{x})'$: $\max(C_M(\boldsymbol{x}), C_Q(\boldsymbol{x}))$

e sono computabili se $C_M(\boldsymbol{x})$ e $C_Q(\boldsymbol{x})$ lo sono e per la computabilità di $\dot{-}$, del prodotto e della funzione $\max$ stabilite dal teorema 3.3. □

Dai pochi, ma significativi esempi visti finora si deduce la potenza del metodo di estensione basato sull'impiego sinergico di sostituzione e ricorsione. In effetti quasi tutte tutte le funzioni cui si può pensare sono sempre ottenibili da questo tipo di estensione, anche se, in certi casi, bisogna fare uno sforzo non indifferente per descrivere tali funzioni in un modo idoneo all'applicazione di sostituzione e ricorsione.
Da un punto di vista storico, nella costruzione del concetto di effettiva computabilità, si arrivò al punto di ritenere che l'insieme delle funzioni computabili chiuso rispetto a sostituzione e ricorsione, chiamato insieme $\mathcal{PR}$ delle *funzioni primitive ricorsive*, coincidesse con l'insieme di *tutte* le funzioni computabili. In effetti così non è, ma in un primo momento non fu facile trovare un controesempio di una funzione che *non* fosse derivabile da sostituzione e ricorsione primitiva. Nel prossimo paragrafo mostreremo tale controesempio; dalla sua definizione di evince che tale funzione è palesemente calcolabili, ma si potrebbe dimostrare che non è derivabile da sostituzione e ricorsione primitiva.

Funzioni totali non primitive ricorsive

Verso la fine degli anni '20 due studenti di Hilbert, Gabriel Sudan e Wilhelm Ackermann, impegnati negli studi sull'*Ipotesi del continuo* (si veda il primo problema di Hilbert 1.12), riuscirono a individuare in modo indipendente due funzioni totali che non sono primitive ricorsive. Quella di Sudan, pubblicata nel 1927, è la meno nota, mentre quella di Ackermann venne subito celebrata dagli studiosi più influenti dell'epoca (fra cui lo stesso Hilbert, Bernays e Pèter). In entrambi i casi la struttura della funzione consiste in una sorta di doppia ricorsione.
A titolo di esempio analizziamo la funzione di Ackermann, che è una funzione a tre parametri n, x, y sulla quale viene innestato una sorta di doppia ricorsione, una su n e l'altra su y. I valori $n = 0, 1, 2$ sono associati alle funzioni elementari $x+1$, $x+y$, $x \cdot y$ e costituiscono il seme d'innesco della doppia ricorsione.

Funzione di Ackermann

$$
\begin{array}{rrcl}
 & A(0,x,y) & = & x+1 \\
 & A(1,x,y) & = & x+y \\
 & A(2,x,y) & = & x \cdot y \\
n \geqslant 3 \qquad & A(n+1,x,y) & = & \left\{ \begin{array}{ll} 1 & \text{se } y = 0 \\ A(n,x,A(n+1,x,y-1)) & \text{se } y \geqslant 1 \end{array} \right.
\end{array}
\tag{3.4}
$$

Proviamo a costruire i primi valori della funzione, per prendere confidenza con la struttura sottesa.

$$
\begin{array}{rcl}
A(3,x,y) & = & A(2,x,A(3,x,y-1)) \\
 & = & x \cdot A(3,x,y-1) \\
 & = & x \cdot x \cdot A(3,x,y-2) \\
 & \vdots & \\
 & = & x^y \cdot A(3,x,0) = x^y
\end{array}
$$

$$
\begin{array}{rcl}
A(4,x,y) & = & A(3,x,A(4,x,y-1)) = x^{A(4,x,y-1)} = \\
 & = & x^{x^{A(4,x,y-1)}} = \cdots = \underbrace{x^{x^{\cdot^{\cdot^{\cdot^{x}}}}}}_{y}
\end{array}
$$

$$
\begin{array}{rcl}
A(5,x,y) & = & A(4,x,A(5,x,y-1)) = \underbrace{x^{x^{\cdot^{\cdot^{\cdot^{x}}}}}}_{A(5,x,y-1)} = \\
 & = & \underbrace{x^{x^{\cdot^{\cdot^{\cdot^{x}}}}}\, x^{x^{\cdot^{\cdot^{\cdot^{x}}}}} \cdots x^{x^{\cdot^{\cdot^{\cdot^{x}}}}}}_{A(5,x,y-2)} = \\
 & = & \underbrace{x^{x^{\cdot^{\cdot^{\cdot^{x}}}}}\, x^{x^{\cdot^{\cdot^{\cdot^{x}}}}} \cdots x^{x^{\cdot^{\cdot^{\cdot^{x}}}}}\, x}_{y}
\end{array}
$$

$$
\begin{array}{rcl}
A(6,x,y) & = & A(5,x,A(6,x,y-1)) = \\
 & = & \underbrace{x^{x^{\cdot^{\cdot^{\cdot^{x}}}}}\, x^{x^{\cdot^{\cdot^{\cdot^{x}}}}} \cdots x^{x^{\cdot^{\cdot^{\cdot^{x}}}}}\, x}_{A(6,x,y-1)} = \quad \underbrace{x^{x^{\cdot^{\cdot^{\cdot^{x}}}}}\, x^{x^{\cdot^{\cdot^{\cdot^{x}}}}} \cdots x^{x^{\cdot^{\cdot^{\cdot^{x}}}}}\, \underbrace{x^{x^{\cdot^{\cdot^{\cdot^{x}}}}}\, x^{x^{\cdot^{\cdot^{\cdot^{x}}}}} \cdots x^{x^{\cdot^{\cdot^{\cdot^{x}}}}}\, x}_{A(6,x,y-2)}}_{} = \\
 & \vdots & \\
 & = & \left. \begin{array}{c} \underbrace{x^{x^{\cdot^{\cdot^{\cdot^{x}}}}}\, x^{x^{\cdot^{\cdot^{\cdot^{x}}}}} \cdots x^{x^{\cdot^{\cdot^{\cdot^{x}}}}}\, x}_{} \\ \vdots \\ \underbrace{x^{x^{\cdot^{\cdot^{\cdot^{x}}}}}\, x^{x^{\cdot^{\cdot^{\cdot^{x}}}}} \cdots x^{x^{\cdot^{\cdot^{\cdot^{x}}}}}\, x}_{} \end{array} \right\} y
\end{array}
$$

Non è difficile immaginare la struttura di $A(7, x, y)$, che avrà $A(7, x, y-1)$ al posto dell'y che troviamo nella definizione di $A(6, x, y)$. Si intuisce inotre la crescita esplosiva dei valori della funzione anche per bassi valori di n; se consideriamo p.es. la $A(5, 2, y)$ si ottiene

$A(5,2,0)$	$A(5,2,1)$	$A(5,2,2)$	$A(5,2,3)$	$A(5,2,4)$
1	2	$2^2\}\, 2 = 4$	$2^{2^{2^2}}\Big\}\, 4 = 65536$	$2^{2^{2^{\cdot^{\cdot^{\cdot^2}}}}}\Big\}\, 65536 = ?$

che già per $A(5, 2, 4)$ fornisce un numero incommensurabilmente grande.
La dimostrazione dell'impossibilità di catturare la funzione di Ackermann usando la ricorsione primitiva viene condotta mostrando che essa cresce più velocemente di qualunque funzione primitiva ricorsiva. Tuttavia la funzione è computabile, anche se la dimostrazione di questo fatto è piuttosto complessa; ciò significa che $\mathcal{PR}$, ottenuto da sostituzione e ricorsione primitiva, non copre tutto l'insieme $\mathcal{C}$ delle funzioni computabili. E' necessario introdurre un ulteriore strumento di estensione.

Estensione delle funzioni mediante minimazione illimitata

Per quanto con la sostituzione e la ricorsione primitiva si riesca a generare la stragrande maggioranza delle funzioni note, abbiamo visto che esistono casi patologici, quali la funzione di Ackermann, per i quali l'impiego di questi strumenti non è sufficiente. Bisogna allora far intervenire un ulteriore strumento di estensione, indipendente da sostituzione e ricorsione e da esse non ottenibile, che si chiama *minimazione illimitata* (o semplicemente *minimazione*). La minimazione ci darà la possibilità di raggiungere tutte le funzioni computabili note, nel senso che l'insieme delle funzioni computabili secondo il modello RAM sarà costituito dall'insieme delle funzioni parziali chiuso rispetto a sostituzione, ricorsione e minimazione. Similmente a quanto già fatto per sostituzione e ricorsione, dimostreremo che la minimazione è computabile scrivendo un programma RAM che la computa. Questo significa che ogni funzione ottenuta mediante l'impiego della minimazione potrà essere dichiarata computabile senza la necessità di scrivere uno specifico programma per computarla.
Per introdurre la minimazione in modo più agevole è opportuno fare inizialmente riferimento alla *minimazione limitata*, che ci apprestiamo a definire e che è viceversa derivabile da sostituzione e ricorsione primitiva. Per mostrare tale derivabilità è necessario introdurre due ulteriori funzioni, che sono la *somma* e il *prodotto limitati*, che costituiscono due tecnicismi per meglio gestire le dimostrazioni che verranno.

Definizione 3.5. *Sia $f(\boldsymbol{x}, z)$ una qualunque funzione. Si definiscono per ricorsione le seguenti funzioni $S(\boldsymbol{x}, y)$ e $P(\boldsymbol{x}, y)$:*

Somma limitata

$$S(\boldsymbol{x}, y) = \sum_{z<y} f(\boldsymbol{x}, z) \quad \text{definita per ricorsione come} \quad \begin{cases} \sum\limits_{z<0} f(\boldsymbol{x}, z) = 0 \\ \sum\limits_{z<y+1} f(\boldsymbol{x}, z) = \sum\limits_{z<y} f(\boldsymbol{x}, z) + f(\boldsymbol{x}, y) \end{cases}$$

Prodotto limitato

$$P(\boldsymbol{x}, y) = \prod_{z<y} f(\boldsymbol{x}, z) \quad \text{definita per ricorsione come} \quad \begin{cases} \prod\limits_{z<0} f(\boldsymbol{x}, z) = 1 \\ \prod\limits_{z<y+1} f(\boldsymbol{x}, z) = \left(\prod\limits_{z<y} f(\boldsymbol{x}, z) \right) \cdot f(\boldsymbol{x}, y) \end{cases}$$

Poiché la somma e il prodotto limitati sono definiti mediante ricorsione, è evidente che se $f(\boldsymbol{x}, z)$ è computabile totale lo sono anche $S(\boldsymbol{x}, y)$ e $P(\boldsymbol{x}, y)$.
Descriviamo ora la minimazione limitata; anch'essa è una tecnica di estensione di funzioni, cioè uno strumento per costruire nuove funzioni a partire da altre funzioni precedentemente definite. L'idea di partenza è quella di andare a cercare, tra i valori di una certa variabile z, il più piccolo valore per il quale valga una certa proprietà. Useremo allora la seguente notazione

$$\mu z < y(...)$$

col seguente significato: '*il minimo z minore di y tale che ...*'. La proprietà può essere di qualunque tipo, purché sia descrivibile col linguaggio delle funzioni. Un modo molto comodo è quello di far collassare a zero il valore di una certa $f(\boldsymbol{x}, z)$ nel momento in cui la proprietà che stiamo cercando emerge; in tal modo la minimazione è descrivibile come

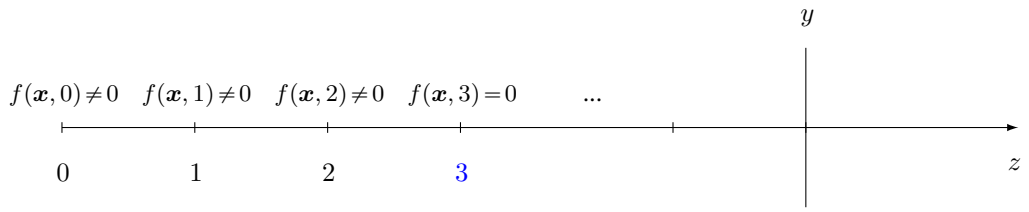
$$\mu z < y(f(\boldsymbol{x}, z) = 0)$$

Per fare in modo che questa espressione sia sempre definita, attribuiamo valore y alla stessa quando un tale z non esiste.

Definizione 3.6. Minimazione limitata *Sia $f(\boldsymbol{x}, z)$ funzione totale computabile. Allora si ha:*

$$g(\boldsymbol{x}, y) = \mu z < y(f(\boldsymbol{x}, z) = 0) = \begin{cases} \text{il minimo } z < y & \text{tale che } f(\boldsymbol{x}, z) = 0, \text{ se tale } z \text{ esiste} \\ y & \text{se tale } z \text{ non esiste} \end{cases}$$

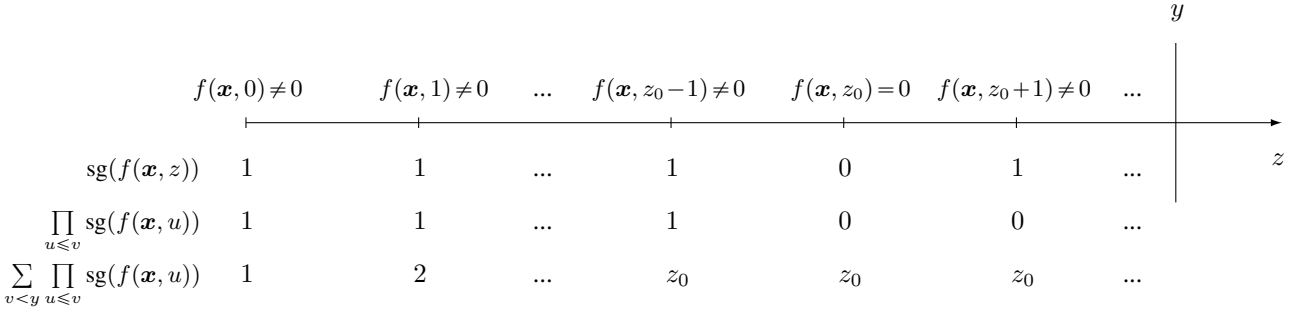
L'operatore $\mu z < y(...)$ viene chiamato anche *operatore di minimazione limitata*.



Si è già accennato al fatto che la minimazione limitata è computabile ed è ottenibile da sostituzione e da ricorsione; d'altra parte non è difficile concepire una procedura per computare $g(\boldsymbol{x}, y)$, poiché basta calcolare successivamente $f(\boldsymbol{x}, 0), f(\boldsymbol{x}, 1), f(\boldsymbol{x}, 2, ...)$ finchè si arriva a $f(\boldsymbol{x}, z) = 0$, oppure ci si scontra con la barriera y, nel qual caso alla funzione viene attribuito il valore y. Tuttavia, se per stabilire la computabilità dell'operatore μ di minimazione limitata scrivessimo un programma RAM, come abbiamo fatto per la sostituzione e per la ricorsione, resterebbe sempre il dubbio che quest'ultima *non* sia ottenibile da sostituzione e da ricorsione primitiva, costituendo così un operatore di estensione indipendente da sostituzione e ricorsione, cioè in grado di estendere in modo *essenziale* l'insieme delle funzioni computabili. Dimostreremo tra poco che così non è.

Teorema 3.4. *Sia $f(\boldsymbol{x}, y)$ una funzione totale computabile; allora anche $\mu z < y(f(\boldsymbol{x}, z) = 0)$ è totale computabile ed è derivabile da sostituzione e ricorsione.*

Dimostrazione. Dobbiamo dimostrare la computabilità ricavandola da sostituzione e ricorsione primitiva. A tal fine usiamo lo schema sotto



Sia z_0 il $\mu z < y(f(\boldsymbol{x}, z) = 0)$; dobbiamo ricavare questo numero in qualche modo. Se consideriamo la funzione $\mathrm{sg}(f(\boldsymbol{x}, z))$ essa vale 0 in corrispondenza di z_0 e vale sempre 1 per $z < z_0$; per $z \geqslant z_0$ può invece valere tanto 0 che 1. Se ora consideriamo il prodotto $\prod_{u \leqslant v} \mathrm{sg}(f(\boldsymbol{x}, u))$ dei primi v termini, il prodotto vale 1 fino a $z_0 - 1$, mentre collassa a 0 da z_0 in poi. Se ora contiamo il numero di 1 che via via incontriamo procedendo da sinistra a destra, usando la formula $\sum_{v<y} \prod_{u \leqslant v} \mathrm{sg}(f(\boldsymbol{x}, u))$, arriviamo al valore z_0, e tale valore rimane costante fino al raggiungimento della barriera y. Ecco allora che si può scrivere

$$z_0 = \mu z < y(f(\boldsymbol{x}, z) = 0) = \sum_{v<y} \prod_{u \leqslant v} \mathrm{sg}(f(\boldsymbol{x}, u))$$

che è computabile per la computabilità delle somme e prodotti limitati e della funzione $\mathrm{sg}(x)$. □

Dalla computabilità della minimazione limitata si possono ottenere anche alcuni risultati sulla decidibilità dei predicati.

Corollario 3.4. *Sia $R(\boldsymbol{x}, y)$ un predicato decidibile. Allora*

(a) *la funzione $f(\boldsymbol{x}, y) = \mu z < y(R(\boldsymbol{x}, z))$ è computabile*

(b) *i seguenti predicati sono decidibili:*

(i) $M_1(\boldsymbol{x}, y) \equiv {}'\forall z < y\ (R(\boldsymbol{x}, z))'$

(ii) $M_2(\boldsymbol{x}, y) \equiv {}'\exists z < y\ (R(\boldsymbol{x}, z))'$

Dimostrazione. Per dimostrare la (a) basta associare la validità del predicato all'uguaglianza a 0 di una qualche funzione, in questo caso della $\overline{\mathrm{sg}}(C_R(\boldsymbol{x}, z)$. La computabilità segue poi dalla computabilità della sostituzione e della minimazione limitata. Per le (b) bisogna invece far vedere che le corrispondenti funzioni caratteristiche sono computabili; ciò deriva dalla computabilità delle somme e dei prodotti limitati. Ecco i dettagli:

(a) $f(\boldsymbol{x}, y) = \mu z < y(\overline{\mathrm{sg}}(C_R(\boldsymbol{x}, z)) = 0)$

(b) (i) $C_{M_1}(\boldsymbol{x}, y) = \prod_{z<y} C_R(\boldsymbol{x}, z)$

(ii) $C_{M_2}(\boldsymbol{x}, y) = \mathrm{sg}(\sum_{z<y} C_R(\boldsymbol{x}, z))$

□

La computabilità della minimazione limitata consente un ulteriore estensione dell'insieme delle funzioni computabili viste nel teorema 3.3

Teorema 3.5. *Le seguenti funzioni sono computabili:*

(p) *Numero di divisori* $D(x)$ $=$ *il numero di divisori di* x $(D(0)=1)$

(q) *Primalità 1* $Pr(x)$ $=\begin{cases} 1 & \textit{se } x \textit{ è primo} \\ 0 & \textit{se } x \textit{ non è primo} \end{cases}$

(r) *Primalità 2* p_x $=$ x*-esimo numero primo* $(p_0=0, p_1=2, p_2=3, ...)$

(s) y*-esimo esponente* $(x)_y$ $=\begin{cases} \textit{l'esponente di } p_y \textit{ nella fattorizzazione in primi di } x, \textit{ per } x,y>0 \\ 0 \textit{ se } x=0 \textit{ o } y=0 \end{cases}$

Dimostrazione. Omesse; si veda [5] p.40. □

Osservazione 3.5. Nel seguito faremo uso della funzione $(x)_y$ per ottenere alcuni importanti risultati teorici; l'effettiva computabilità della funzione, e cioè il fatto di poter ricavare il valore $(x)_y$ da una procedura, giocherà un ruolo determinante.

Abbiamo dimostrato che la minimazione limitatata non costituisce uno strumento di estensione indipendente da sostituzione e ricorsione; non è quindi in grado di estendere in modo essenziale l'insieme delle funzioni computabili. Per fare ciò è necessario modificare la minimazione limitatata, spostando verso l'infinito la barriera y; in questo modo si ottiene la *minimazione illimitatata* che andiamo ora a definire.
Supponiamo che $f(\boldsymbol{x}, y)$ sia una funzione (non necessariamente totale) e tramite essa si voglia definire una funzione $g(\boldsymbol{x})$ nel modo seguente

$$g(\boldsymbol{x}) = \mu y(f(\boldsymbol{x}, y)=0) = \text{il più piccolo } y \text{ tale che } f(\boldsymbol{x}, y)=0$$

in modo tale che se f è computabile lo sia anche g. Ci sono due possibili problemi da affrontare per una corretta definizione di f; il primo è che potrebbe succedere che non esista alcun valore di y per il quale si verifica l'uguaglianza $f(\boldsymbol{x}, y)=0$; il secondo è che, se anche esiste, p.es. $f(\boldsymbol{x}, 3)=0$, potrebbe succedere che $f(\boldsymbol{x}, 1)$ non sia definita. In tal caso l'impiego della naturale procedura di computazione di g, che consiste nel calcolare successivamente $f(\boldsymbol{x}, 0), f(\boldsymbol{x}, 1), f(\boldsymbol{x}, 2), f(\boldsymbol{x}, 3), ...$, porterebbe a una divergenza nel calcolo di $f(\boldsymbol{x}, 1)$. Tenuto conto di tutto ciò possiamo procedere con la seguente

Definizione 3.7. Minimazione illimitata *Assegnata una funzione* $f(\boldsymbol{x}, y)$ *definiamo*

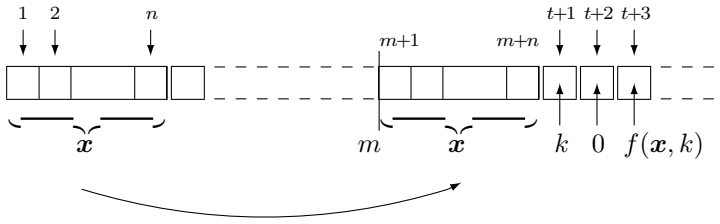
$$g(\boldsymbol{x}) = \mu y(f(\boldsymbol{x}, y)=0) = \begin{cases} \textit{il più piccolo } y \textit{ tale che:} & (i) \quad f(\boldsymbol{x}, z) \textit{ è definita } \forall z \leqslant y, \quad e \\ & (ii) \quad f(\boldsymbol{x}, y)=0 \textit{ se tale } y \textit{ esiste} \\ \textit{indefintita} & \textit{se tale } y \textit{ non esiste} \end{cases}$$

La funzione $\mu y(f(\boldsymbol{x}, y)=0)$ *viene anche chiamata operatore* μ *di minimazione.*

Il prossimo teorema ci mostra che $\mathcal{C}$ è chiuso rispetto alla minimazione.

Teorema 3.6. *Sia* $f(\boldsymbol{x}, y)$ *una funzione computabile; allora anche la funzione* $g(\boldsymbol{x}) = \mu y(f(\boldsymbol{x}, y)=0)$ *è computabile.*

Dimostrazione. Anche in questo caso procederemo come per la sostituzione e la ricorsione, costruendo un programma G che calcola g. Useremo la procedura naturale che consiste nel calcolare successivamente $f(\boldsymbol{x}, k)$ per $k=0, 1, 2, ...$, finché si trova un k per il quale $f(\boldsymbol{x}, k)=0$; ciò è possibile in quanto f è computabile per ipotesi e



quindi esiste un programma F che la computa. Poniamo $m = \max(n+1, \rho(F))$ e $t = m + n$ per alleggerire la notazione. Prima di procedere sposteremo $\boldsymbol{x} = (x_1, x_2, ..., x_n)$ in zona di sicurezza, mantenendo in essa anche i valori correnti di k.

Il programma G è allora il seguente: Si osservi che la procedura diverge se non esiste y tale che $f(\boldsymbol{x}, y) = 0$,

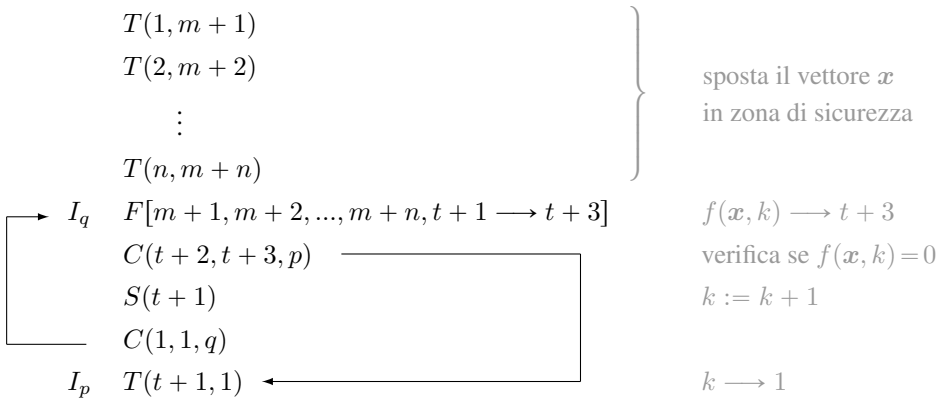


Figura 3.4: Programma per RAM-calcolare la minimazione illimitata

ma anche se $f(\boldsymbol{x}, k)$ non è definita per almeno un valore di $k \leqslant y$ quando $f(\boldsymbol{x}, y) = 0$. Ciò è però coerente con la definzione 3.7. □

Il seguente corollario costituisce l'analogo del 3.4(a) nel caso di minimazione illimitata.

Corollario 3.5. *Sia $R(\boldsymbol{x}, y)$ un predicato decidibile. Allora la funzione*

$$\mu y(R(\boldsymbol{x}, y)) = \begin{cases} \textit{il più piccolo } y \textit{ tale che } R(\boldsymbol{x}, y) \textit{ vale} & \textit{se tale } y \textit{ esiste} \\ \textit{indefinita} & \textit{altrimenti} \end{cases}$$

è computabile.

Dimostrazione. La funzione $g(\boldsymbol{x}) = \mu y(\overline{\text{sg}}(C_R(\boldsymbol{x}, y) = 0))$ è computabile per sostituzione e per la computabilità della minimazione. □

Il corollario che abbiamo appena visto ci offre il destro per descrivere sotto una nuova luce l'operatore μ, che può essere considerato anche come un *operatore di ricerca*. Assegnato infatti il predicato decidibile $R(\boldsymbol{x}, y)$,

la funzione cerca gli y per i quali il predicato vale (se ne esiste almeno uno); e tra tutti individua quello minimo. Il fatto che la minimazione sia una estensione essenziale può essere dedotto anche dalla circostanza che, contrariamente a quanto succede per sostituzione e ricorsione, essa è in grado di generare una funzione parziale a partire da una totale, come si evince dal seguente

Esempio 3.7. Sia $f(x, y) = |x - y^2|$ (funzione totale) e $g(x) \simeq \mu y(f(x, y) = 0))$. $g(x)$ è allora la seguente funzione parziale

$$g(x) = \begin{cases} \sqrt{x} & \text{se } x \text{ è un quadrato perfetto} \\ \text{indefinita} & \text{altrimenti} \end{cases}$$

Poiché l'operatore μ non è derivabile in termini di sostituzione e ricorsione, esso ci consente di estendere in modo essenziale l'insieme delle funzioni computabili. Siamo ora in grado di definire in modo più specifico $\mathcal{C}$; esso è l'insieme delle funzioni ottenibile dalle funzioni base mediante sostituzione, ricorsione e minimazione.

Capitolo 4

Numerabilità delle funzioni computabili

Nei capitoli precedenti abbiamo costruito un modello di computazione - il modello RAM - e abbiamo definito il concetto di funzione computabile, cioè di funzione per la quale esiste un programma che la RAM-calcola (def. 2.3). A partire dalle funzioni base 3.1, usando sostituzione, ricorsione e minimazione, abbiamo via via allargato l'insieme delle funzioni dichiarate computabili, intercettando anche funzioni molto particolari - quali quelle di Sudan e Ackermann - per le quali è richiesto l'uso essenziale della minimazione. Poiché abbiamo dimostrato la computabilità di sostituzione, ricorsione e minimazione - scrivendo un programma RAM che calcola ciascuna di esse - tutte le funzioni ottenute dalle funzioni base mediante uno o più di questi tre operatori (usati in modo singolo o sinergico) sono state dichiarate computabili, senza dover necessariamente scrivere un programma per la loro computazione. Così facendo abbiamo costruito l'insieme $\mathcal{C}$ delle funzioni RAM calcolabili. Sulla base della tesi di Church-Turing questo insieme coincide con l'insieme di tutte le funzioni calcolabili nel senso intuitivo del termine. D'altra parte nessuno è mai riuscito, finora, a esibire l'esempio di una funzione che sia palesemente calcolabile secondo il senso comune, ma che non lo sia nel modello RAM. Se un giorno ciò dovesse succedere sarebbe necessario aggiornare il modello, rendendolo più accurato; tuttavia la probabilità che ciò accada è assai remota, visto che $\mathcal{C}$ coincide esattamente con gli insiemi computabili secondo tutti gli altri modelli (Church, Turing, Gödel-Kleene, ...).
Il prossimo passo è quello di comprendere quante siano queste funzioni computabili (poche, tante, infinite,...) e se sono infinite quale sia l'ordine di questo infinito (numerabile o con la cardinalità dei reali). Bisognerà inoltre porre a confronto la cardinalità delle funzioni computabili con quella dell'insieme di *tutte* le funzioni possibili; se infatti emergesse che l'insieme delle funzioni computabili è un sottinsieme proprio di tutte le possibili, se ne dedurrebbe immediatamente l'esistenza di *funzioni non computabili*. Le risposte che otterremo da questa analisi sono fondamentali per sondare i *limiti intrinseco-strutturali del metodo procedurale-algoritmico*, in altre parole ciò che si può e ciò che non si può fare con i computer.
Si osservi che una limitazione strutturale di questo genere, che potrebbe essere descritta in termini colloquiali affermando che un certo problema *non può essere risolto mediante l'approccio procedurale-algoritmico*, prescinderebbe da elementi di carattere tecnologico, quali disponibilità di memoria RAM, velocità di *clock*, numero di *core* del computer, quantità di memoria *cache* ecc., ma si prefigurerebbe come limite teorico invalicabile alla potenza di calcolo di un qualunque computer passato, presente e futuro.
Prima di iniziare questa analisi, che ci porterà ai risultati concettuali più fecondi della *Teoria della Computabilità*, dobbiamo richiamare alcuni concetti legati alla cardinalità degli insiemi infiniti.

4.1 Cardinalità degli insiemi infiniti

La valutazione della cardinalità di un insieme finito non presenta problemi; si tratta di contare il numero di elementi dell'insieme. Se dobbiamo invece confrontare le cardinalità di due insiemi X e Y, per capire se sono *equipotenti*, si può procedere in due modi: (i) contare gli $|X|$ elementi di X e gli $|Y|$ elementi di Y e verificare se $|X| = |Y|$; (ii) vedere se è possibile creare una corrispondenza biunivoca tra gli elementi di X e quelli di Y. Nel disegno di figura 4.1 si tenta proprio questa strada; l'impossibilità di creare una tale corrispondenza sancisce una differenza nella cardinalità dei due insiemi. Le cose si fanno più incerte quando si lavora con gli

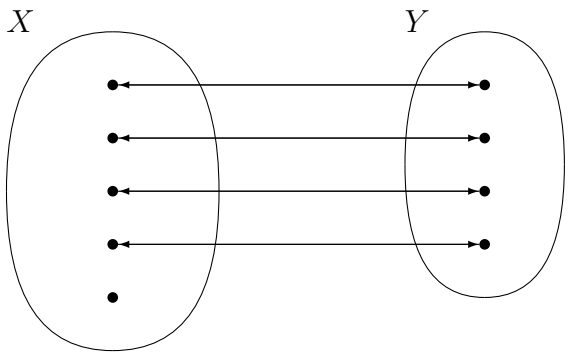


Figura 4.1: Due insiemi finiti non equipotenti

insiemi infiniti. Consideriamo p.es. i numeri naturali da una parte e l'insieme $\mathcal{B}^+$, con $\mathcal{B} = \{0, 1\}$, costituito da tutte e sole le stringhe binarie di lunghezza finita costruite con elementi di $\mathcal{B}$; i primi elementi di $\mathcal{B}^+$ sono $0, 1, 00, 01, 10, 11, 000, 001, ...$ Come si confrontano le cardinalità?
La soluzione venne introdotta da Cantor: *due insiemi infiniti sono equipotenti se è possibile porre in corrispondenza biunivoca gli elementi dei due insiemi*.
In altre parole la cardinalità è definita implicitamente introducendo una *funzione biiettiva*, cioè una funzione che è *iniettiva* e *suriettiva*. Ciò significa che, se X e Y sono due insiemi e $\alpha, \beta \in X$, si deve individuare una funzione f tale che

$$f \text{ iniettiva:} \quad \alpha \neq \beta \Rightarrow f(\alpha) \neq f(\beta)$$
$$f \text{ suriettiva:} \quad f(X) = Y$$

Per poter stabilire la corrispondenza biunivoca nell'esempio di cui sopra, bisogna ordinare gli elementi di entrambi gli insiemi, avendo cura di verificare che nessun elemento sfugga all'ordinamento. Per i naturali si può usare l'ordinamento naturale, $0, 1, 2, 3, ...$; per l'insieme delle stringhe binarie di lunghezza finita si può invece usare la tecnica di elencare prima tutte quelle di lunghezza 1, poi tutte quelle di lunghezza 2, poi tutte quelle di lunghezza 3 e così via, usando l'ordine indotto dalla decodifica della notazione posizionale in base 2 all'interno delle stringhe con pari lunghezza. In questo modo si riesce a enumerare tutte le stringhe, senza perderne alcuna per strada; inoltre ogni stringa ha una specifica posizione all'interno della sequenza, che potrebbe essere calcolata a priori. La corrispondenza che si viene a creare è la seguente:

0	1	2	3	4	5	6	7	8	9	10	...
↕	↕	↕	↕	↕	↕	↕	↕	↕	↕	↕	...
0	1	00	01	10	11	000	001	010	011	100	...

In questo caso non siamo riusciti a esplicitare la corrispondenza con una formula chiusa, che esprima direttamente il valore della funzione; ciò non è però importante dal punto di vista concettuale, poiché l'ordinamento di entrambi

gli insiemi è esplicitato in modo non ambiguo (ordinamento totale in senso stretto).
Osserviamo che la possibilità di ordinare in modo corretto gli elementi di $\mathcal{B}^+$ è legata alla circostanza che siamo riusciti a trovare una strategia per enumerare, cioè *contare*, tutti gli oggetti. Così facendo si stabilisce la corrispondenza biunivoca con $\mathbb{N}$. E' per questo motivo che un insieme equipotente con $\mathbb{N}$ viene chiamato *numerabile*.
Vediamo ora alcuni esempi di insiemi numerabili.

Esempio 4.1.

Numeri pari $\quad \mathbb{P} = \{0, 2, 4, 6, ...\}$
$\mathbb{N} = \{0, 1, 2, 3, ...\}$

In questo caso possiamo scrivere in forma esplicita la funzione f di biiezione come

$$f(x) = x/2 \qquad f^{-1}(n) = 2n$$

con $n \in \mathbb{N}$. Si osservi che $\mathbb{P}$ è un sottinsieme proprio di $\mathbb{N}$; ma poiché esso viene posto in corrispondenza biunovoca con $\mathbb{N}$, i due insiemi infiniti hanno la stessa cardinalità.

Numeri relativi $\quad \mathbb{Z} = \{0, -1, +1, -2, +2, -3, +3, ...\}$
$\mathbb{N} = \{0, \quad 1, \quad 2, \quad 3, \quad 4, \quad 5, \quad 6, ...\}$

$$\alpha(x) = \begin{cases} 2x & \text{se } x \geqslant 0 \\ -2x - 1 & \text{se } x < 0 \end{cases} \qquad \alpha^{-1}(n) = \begin{cases} n/2 & \text{se } n \text{ è pari} \\ -(n+1)/2 & \text{se } n \text{ è dispari} \end{cases}$$

Abbiamo già incontrato questa funzione precedentemente, nell'esempio 3.2.

La numerabilità dell'insieme dei numeri pari e dei numeri relativi risulta intuitiva; meno intuitivo è il fatto che si possa trovare una corrispondenza biunivoca anche tra naturali e razionali; dopotutto gli elementi di $\mathbb{Q}$ sono *densi* sulla retta reale $\mathbb{R}$, nel senso che tra due qualunque elementi $\alpha, \gamma \in \mathbb{R}$ arbitrariamente vicini, con $\alpha < \gamma$, esiste sempre un elemento $\beta \in \mathbb{Q}$ tale che $\alpha < \beta < \gamma$.
La biiezione si scopre subito non appena si realizzi che ogni numero razionale viene identificato in modo univoco dalla coppia di numeri naturali m, n, quella che porta alla frazione m/n. Come sottolineato in precedenza la numerabilità di un insieme deriva direttamente dalla possibilità di *contare* i suoi elementi senza perderne alcuno nella conta; una possibilità è quella di ordinarli rispetto alla somma tra numeratore e denominatore, cioè $m+n$. Procedendo in questo modo possiamo scrivere la tabella doppiamente semi-infinita rappresentata nella figura 4.2.

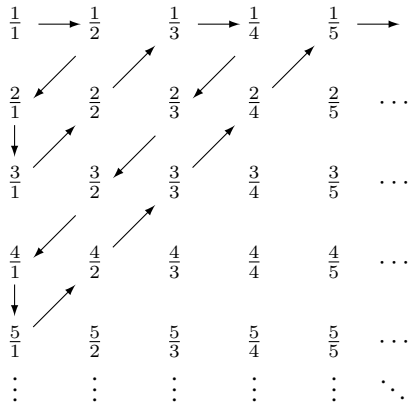


Figura 4.2: Procedura per numerare gli elementi di $\mathbb{Q}$

La tabella contiene tutti i numeri razionali, poiché qualunque sia la coppia m, n il corrispondente razionale si trova sull'incrocio tra m-esima colonna e n-esima riga. Se seguiamo il percorso indicato dalle frecce incrociamo sulla

diagonale tutte le frazioni con $s = m+n$ costante; siamo dunque in grado di ordinare tutte le frazioni per valori successivi di s e di porle in corrispondenza biunivoca con gli elementi di $\mathbb{N}$ secondo la seguente tabella:

$$\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & \ldots \\
\updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \ldots \\
\frac{1}{1} & \frac{1}{2} & \frac{2}{1} & \frac{3}{1} & \frac{2}{2} & \frac{1}{3} & \frac{1}{4} & \frac{2}{3} & \frac{3}{2} & \frac{4}{1} & \frac{5}{1} & \ldots \\
 & \underbrace{\quad}_{s=3} & & \underbrace{\quad\quad}_{s=4} & & & \underbrace{\quad\quad\quad}_{s=5} & & & & &
\end{array}$$

Questo metodo di enumerazione dei razionali ha lo svantaggio di non poter essere espresso con una formula chiusa semplice; nella prossima sezione useremo un altro tipo di corrispondenza biunivoca, che gode di questo vantaggio. Il fatto che l'insieme $\mathbb{Q}$ sia numerabile è un risultato concettuale rilevante e tutto sommato controintuitivo. A questo punto viene la curiosità di capire se anche per $\mathbb{R}$ valga lo stesso risultato. Sappiamo tuttavia che così non è; in questo caso il fatto è abbastanza intuitivo, data la struttura di $\mathbb{R}$ e il fatto che per specificare un numero reale serve una quantità infinita d'informazione - al contrario dei numeri razionali dove basta una coppia di naturali. La dimostrazione di tale fatto è dovuta a Cantor, ed è tanto semplice quanto geniale; è il primo esempio di *dimostrazione per diagonalizzazione*, che useremo spesso nella nostra analisi sui limiti della computabilità.

Teorema 4.1. $\mathbb{R}$ *non è numerabile.*

Dimostrazione. Si deve dimostrare che non è possibile stabilire una corrispondenza biunivoca tra $\mathbb{R}$ e $\mathbb{N}$. La dimostrazione procede per assurdo e viene limitata all'intervallo $[0...1]$, poiché se vale per questo sottinsieme, vale per $\mathbb{R}$ a maggior ragione. Si supponga che una tale corrispondenza esista; in tal caso sarebbe possibile costruire una tabella nella quale, accanto a ciascun elemento n di $\mathbb{N}$, verrebbe posto l'elemento $f(n)$ di $\mathbb{R}$ che gli corrisponde nella biiezione. Supponiamo che la situazione sia quella nella seguente tabella:

n	$f(n)$								
0	0,	(7)	1	3	4	1	9	2	$\cdots$
1	0,	4	(2)	0	3	5	5	7	$\cdots$
2	0,	4	2	(2)	7	2	8	2	$\cdots$
3	0,	2	3	0	(4)	6	4	5	$\cdots$
4	0,	9	7	9	5	(1)	0	1	$\cdots$
5	0,	2	1	3	6	9	(6)	0	$\cdots$
6	0,	1	8	5	5	8	1	(3)	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	

L'idea della dimostrazione è quella di far vedere che esiste $r \in \mathbb{R}$ che *non* appartiene alla tabella, cioè tale che $r \neq f(n), \forall n$. Per inviduare r prendiamo gli elementi che stanno sulla diagonale e costruiamo r nel modo seguente:

$$0,\overline{7}\,\overline{2}\,\overline{2}\,\overline{4}\,\overline{1}\,\overline{6}\,\overline{3}\ldots \quad\longrightarrow\quad r = 0,3\,6\,8\,7\,3\,5\,4\ldots$$

Il simbolo $\overline{x}$ significa "*un qualunque intero nell'intervallo* $[0...9]$ *diverso da* x". Una possibile interpretazione di r potrebbe essere allora $r = 0,3687354\ldots$, ma naturalmente ci sono infiniti modi per costruire r. E' evidente che r non può appartenere alla tabella, poiché è diverso da ogni elemento $f(n)$ almeno nella $(n+1)$-esima posizione dopo la virgola. Questo fatto contraddice l'ipotesi che la tabella costituisca una corrispondenza biunivoca. □

Esistono dunque due ordini d'infinito, l'infinito del numerabile e l'infinito del reale. Poiché all'interno del modello RAM possiamo costruire infiniti programmi è interessante capire se la loro cardinalità sia numerabile o

meno. Inoltre, poiché a ogni funzione si possono associare infiniti programmi che la computano, è importante capire se l'ordine delle funzioni sia lo stesso di quello dei programmi. Nella prossima sezione daremo una risposta a questi quesiti.

4.2 Gödelizzazione dei programmi

Il risultato principale che otterremo in questo capitolo è che *l'insieme di tutti i programmi RAM è effettivamente numerabile*. L'enumerazione dei programmi viene chiamata *Gödelizzazione*, in onore di Kurt Gödel che ideò il procedimento di associare a ciascun simbolo e a ciascuna formula ben formata di un linguaggio formale un unico numero naturale nella dimostrazione dei teoremi di incompletezza di cui abbiamo parlato nella sezione 1.2. L'avverbio "effettivamente" sta a significare che è possibile costruire una procedura per attribuire a ogni programma l'intero che gli corrisponde nella biiezione e viceversa. Questa sottolineatura non è superflua, poiché sappiamo che in matematica esistono teoremi non costruttivi, nei quali si dimostra la possibilità di attuare una certa operazione senza necessariamente fornire gli strumenti operativi per realizzarla effettivamente. In questo caso non solo si stabilisce che *esiste* una corrispondenza biunivoca, ma si costruisce una procedura effettiva per attuarla. Cristallizziamo allora questi concetti nelle seguenti definizioni.

Definizione 4.1.

(a) Un insieme X è numerabile se esiste una biiezione $f : X \to \mathbb{N}$.

(b) Una enumerazione $\{x_0, x_1, x_2, ...\}$ di un insieme X è una funzione suriettiva $g : \mathbb{N} \to X$ tale che $x_n = g(n)$.

(c) Se X è un insieme numerabile tramite la biiezione f, esso si dice effettivamente numerabile se f e la sua inversa f^{-1} sono entrambe computabili.

Poiché un programma $P = \{I_1, I_2, \ldots, I_s\}$ è costituito da s istruzioni RAM, per poter enumerare tutti i programmi partiremo dall'enumerazione delle istruzioni. L'insieme $\mathcal{I}$ di tutte le possibili istruzioni è così costituito

$$\begin{aligned} \mathcal{I} = \quad & \{Z(1), Z(2), Z(3), \ldots, \\ & S(1), S(2), S(3), \ldots \\ & T(1,1), T(2,1), T(1,2), \ldots \\ & C(1,1,1),\, C(2,1,1), C(1,1,2), C(1,2,1), \ldots\} \end{aligned} \tag{4.1}$$

L'enumerazione di $\mathcal{I}$ assocerà un intero $\beta(I_j)$ a ciascuna istruzione $I_j \in \mathcal{I}$ del programma secondo la biiezione β. Da questa operazione otterremo una s-pla di interi $\beta(I_1), \beta(I_2), \ldots, \beta(I_s)$, che andrà a sua volta posta in corrispondenza biunivoca con gli elementi di $\mathbb{N}$ tramite una funzione biiettiva $\tau(\beta(I_1), \beta(I_2), \ldots, \beta(I_s))$.
Per poter definire la biiezione finale bisogna allora costruire la β e la τ. La τ sarà una biiezione tra una s-pla di interi e gli elementi di $\mathbb{N}$, mentre la β va costruita tenendo conto che ogni istruzione può avere $1, 2$ o 3 parametri. Nel seguente teorema troviamo le funzioni di cui abbiamo bisogno.

Teorema 4.2. *I seguenti insiemi sono effettivamente numerabili:*

(a) $\mathbb{N} \times \mathbb{N}$

(b) $\mathbb{N}^+ \times \mathbb{N}^+ \times \mathbb{N}^+$

(c) $\bigcup_{k>0} \mathbb{N}^k$, *l'insieme di tutte le sequenze finite di numeri naturali*

Dimostrazione.

(a) Ogni numero naturale può essere scomposto come prodotto della componente pari espressa come 2^m, per qualche m, e di quella dispari espressa come $2n+1$, per qualche n. Consideriamo allora la seguente biiezione $\pi : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$:

$$\pi(m,n) = 2^m(2n+1) - 1 = x \tag{4.2}$$

dove il -1 serve solo a fare in modo che sia $\pi(0,0) = 0$. La funzione π associa in modo univoco un numero naturale x a ciascuna coppia m, n di naturali. Per ricavare m, n da x, cioè calcolare la $\pi^{-1}(x) = (m,n)$, basta risolvere rispetto m e n l'equazione di sopra. Così facendo si ottiene

$$\begin{aligned} m &= (x+1)_1 = \pi_1(x) \\ n &= \frac{1}{2}\left(\frac{x+1}{2^{\pi_1(x)}} - 1\right) = \pi_2(x) \end{aligned} \tag{4.3}$$

cioè

$$\pi^{-1}(x) = (m,n) = (\pi_1(x), \pi_2(x)) \tag{4.4}$$

dove con $\pi_1(x)$ e $\pi_2(x)$ denotiamo le funzioni inverse per ottenere rispettivamente m e n. La biiezione π è effettiva; infatti tanto π che π^{-1} sono computabili, data la computabilità delle funzioni π_1 e π_2 (per la π_1 si veda la (s) del teorema 3.3); le formule chiuse di cui sopra ci danno la possibilità di ricavare $x = \pi(m,n)$ da m e n e viceversa.

(b) Per creare una biiezione ξ tra terne di naturali e $\mathbb{N}$ possiamo annidare la π dentro sè stessa nel modo seguente:

$$\xi(m,n,q) = \pi\left(\pi(m-1, n-1), q-1\right) = x$$

Per ricavare la ξ^{-1} bisogna ricorrere alle equazioni (4.3)

$$\begin{aligned} \pi_1(x) &= (x+1)_1 = \pi(m-1, n-1) \\ \pi_2(x) &= \frac{1}{2}\left(\frac{x+1}{2^{\pi_1(x)}} - 1\right) = q - 1 \end{aligned}$$

e dunque

$$\begin{aligned} m - 1 &= \pi_1\left(\pi_1(x)\right) & m &= \pi_1\left(\pi_1(x)\right) + 1 \\ n - 1 &= \pi_2\left(\pi_1(x)\right) & n &= \pi_2\left(\pi_1(x)\right) + 1 \\ q - 1 &= \pi_2(x) & q &= \pi_2(x) + 1 \end{aligned} \tag{4.5}$$

cioè

$$\xi^{-1}(x) = (m,n,q) = \left(\pi_1\left(\pi_1(x)\right)+1, \pi_2\left(\pi_1(x)\right)+1, \pi_2(x)+1\right) \tag{4.6}$$

Anche in questo caso ξ e ξ^{-1} sono computabili per la computabilità di π_1 e π_2.

(c) Una biiezione $\bigcup_{k>0} \mathbb{N}^k \to \mathbb{N}$ si può definire nel modo seguente:

$$\tau(a_1, a_2, \ldots, a_k) = 2^{a_1} + 2^{a_1+a_2+1} + 2^{a_1+a_2+a_3+2} + \ldots + 2^{a_1+a_2+\ldots+a_k+(k-1)} - 1 = x \tag{4.7}$$

Chiaramente τ è computabile per la computabilità della somma e della funzione esponenziale. Per calcolare $\tau^{-1}(x)$ usiamo il fatto che ogni numero naturale ha un'unica espressione in notazione posizionale in base 2. Dunque, assegnato x possiamo effettivamente individuare $k \geqslant 1$ e $0 \leqslant b_1 < b_2 <, \ldots, < b_k$ tali che

$$x + 1 = 2^{b_1} + 2^{b_2} + \ldots + 2^{b_k}$$

dal quale ricaviamo

$$\tau^{-1}(x) = (a_1, a_2, \ldots, a_k)$$

mediante le equazioni di ricorrenza

$$\begin{aligned} a_1 &= b_1 \\ a_{i+1} &= b_{i+1} - b_i - 1 \qquad (1 \leqslant i < k) \end{aligned} \tag{4.8}$$

□

Vediamo ora un esempio per ogni funzione.

Esempio 4.2.

(a)
$$\begin{aligned}(m,n) &= (2,3)\\ \pi(2,3) &= 2^2(2\cdot 3+1)-1 = 27 = x\\ \pi_1(27) &= (27+1)_1 = (28)_1 = (2^2\cdot 7)_1 = 2 = m\\ \pi_2(27) &= \frac{1}{2}\left(\frac{27+1}{2^2}-1\right) = 3 = n\end{aligned}$$

(b)
$$\begin{aligned}(m,n,q) &= (2,3,4)\\ \xi(2,3,4) &= \pi\left(\pi(1,2),3\right) = \pi(9,3) = 2^9(2\cdot 3+1)-1 = 512\cdot 7-1 = 3583 = x\\ \pi_1(3583) &= (3583+1)_1 = (2^9\cdot 7)_1 = 9\\ \pi_2(3583) &= \frac{1}{2}\left(\frac{3583+1}{2^9}-1\right) = 3\\ m &= \pi_1(9)+1 = (9+1)_1+1 = (2\cdot 5)_1+1 = 1+1 = 2\\ n &= \pi_2(9)+1 = \frac{1}{2}\left(\frac{9+1}{2^1}-1\right)+1 = 3\\ q &= \pi_2(3583)+1 = 3+1 = 4\end{aligned}$$

(c)
$$\begin{aligned}(a_1,a_2,a_3,a_4) &= (2,1,2,3)\\ \tau(2,1,2,3) &= 2^2+2^{2+1+1}+2^{2+1+2+2}+2^{2+1+2+3+3}-1 = 2^2+2^4+2^7+2^{11}-1 =\\ &= 4+16+128+2048-1 = 2195\\ 2195+1 &= 2196 = 2^2+2^4+2^7+2^{11} = 2^{b_1}+2^{b_2}+2^{b_3}+2^{b_4}\\ a_1 &= b_1 = 2\\ a_2 &= b_2-b_1-1 = 4-2-1 = 1\\ a_3 &= b_3-b_2-1 = 7-4-1 = 2\\ a_4 &= b_4-b_3-1 = 11-7-1 = 3\end{aligned}$$

Grazie ai risultati del teorema 4.2 possiamo ora dimostrare il primo risultato rilevante

Teorema 4.3. *L'insieme $\mathcal{I}$ delle istruzioni RAM è effettivamente numerabile.*

Dimostrazione. La dimostrazione viene suggerita immediatamente dalla struttura (4.1) dell'insieme $\mathcal{I}$. Gli elementi dell'insieme possono essere contati per gruppi di 4, in modo tale che la funzione $\beta : \mathcal{I} \to \mathbb{N}$ mappi i quattro tipi di istruzioni in numeri naturali nella forma $4q, 4q+1, 4q+2, 4q+3$ rispettivamente. Usando le funzioni π e ξ precedentemente definite possiamo scrivere

$$\begin{aligned}&\beta(Z(n)) = 4(n-1)\\ &\beta(S(n)) = 4(n-1)+1\\ &\beta(T(m,n)) = 4\,\pi(m-1,n-1)+2\\ &\beta(C(m,n,q)) = 4\,\xi(m,n,q)+3\end{aligned}$$

Poiché π e ξ sono computabili lo è anche β. Per dimostrare che β è una biiezione effettiva bisogna ricavare β^{-1} e dimostrare che anch'essa è computabile. Assegnato x, per ricavare $\beta^{-1}(x)$ bisogna calcolare il resto della divisione di x per 4, cioè i valori q e r tali che $x = 4q+r$, con $0 \leqslant r < 4$. Il valore di r indica di quale tipo sia

l'istruzione, mentre dal valore di q ricaviamo i parametri dell'istruzione nel modo seguente:

$$\text{se } r = 0 \quad \text{allora} \quad \beta^{-1}(x) = Z(q+1) \tag{4.9}$$
$$\text{se } r = 1 \quad \text{allora} \quad \beta^{-1}(x) = S(q+1) \tag{4.10}$$
$$\text{se } r = 2 \quad \text{allora} \quad \beta^{-1}(x) = T(\pi_1(q)+1, \pi_2(q)+1) \tag{4.11}$$
$$\text{se } r = 3 \quad \text{allora} \quad \beta^{-1}(x) = C(m,n,q) \text{ dove } (m,n,q) = \xi^{-1}(q) \tag{4.12}$$

Dunque anche β^{-1} è effettivamente computabile. □

Col teorema 4.3 abbiamo fissato la prima enumerazione, quella degli elementi di $\mathcal{I}$. Facendo qualche calcolo si ricava la seguente corrispondenza:

		r			
		0	1	2	3
u	0	$Z(1)/0$	$S(1)/1$	$T(1,1)/2$	$C(1,1,1)/3$
	1	$Z(2)/4$	$S(2)/5$	$T(2,1)/6$	$C(2,1,1)/7$
	2	$Z(3)/8$	$S(3)/9$	$T(1,2)/10$	$C(1,1,2)/11$
	3	$Z(4)/12$	$S(4)/13$	$T(3,1)/14$	$C(1,2,1)/15$
	4	$Z(5)/16$	$S(5)/17$	$T(1,3)/18$	$C(1,1,3)/19$
	5	$Z(6)/20$	$S(6)/21$	$T(2,2)/22$	$C(2,1,2)/23$
	6	$Z(7)/24$	$S(7)/25$	$T(1,4)/26$	$C(1,1,4)/27$
	7	$Z(8)/28$	$S(8)/29$	$T(4,1)/30$	$C(3,1,1)/31$
	8	$Z(9)/32$	$S(9)/33$	$T(1,5)/34$	$C(1,1,5)/35$
	9	$Z(10)/36$	$S(10)/37$	$T(2,3)/38$	$C(2,1,3)/39$
	⋮	⋮	⋮	⋮	⋮

All'incrocio tra i valori di q e r si trova I/n, dove I è l'istruzione che corrisponde al numero naturale $n = 4q + r$. Sia ora $\mathcal{P}$ l'insieme di tutti i programmi nella forma $P = \{I_1, I_2, \ldots, I_s\}$, costituiti da s istruzioni RAM con s di lunghezza arbitraria e $I_j \in \mathcal{I}$. Vale il seguente

Teorema 4.4. *$\mathcal{P}$ è effettivamente numerabile.*

Dimostrazione. Definiamo una biiezione $\gamma : \mathcal{P} \to \mathbb{N}$ usando le biiezioni β e τ dei teoremi 4.2 e 4.3. Se $P = \{I_1, I_2, \ldots, I_s\}$ si ha

$$\gamma(P) = \tau\left(\beta(I_1), \beta(I_2), \ldots \beta(I_s)\right) = n$$

Poiché β e τ sono biiezioni lo è anche γ; e il fatto che β e τ e le loro inverse β^{-1} e τ^{-1} siano effettivamente computabili garantisce che anche γ e γ^{-1} lo siano. □

Il valore $\gamma(P)$ è chiamato *codifica di P* o *numero di Gödel di P*; definiamo inoltre

$$P_n = \gamma^{-1}(n) = \text{ il programma con numero di Gödel } n \tag{4.13}$$

e diciamo che P_n è l'n-esimo programma. Per come è costruita γ, se $m \neq n$ allora P_m e P_n sono due programmi diversi e viceversa, anche se magari calcolano la stessa funzione. Sappiamo infatti che in generale esistono diversi approcci per risolvere lo stesso problema, cioè diversi algoritmi e quindi diversi programmi. Ma anche usando lo stesso algoritmo, possiamo infarcire il programma di istruzioni inutili ottenendo un altro programma correttamente funzionante; tali istruzioni non incidono sulle soluzioni ottenute, ma formalmente creano un programma diverso. Sottolineamo che, d'ora in poi:

(a) Assegnato un certo programma P possiamo ricavare in modo effettivo il suo numero di Gödel $n = \gamma(P)$

(b) Assegnato un certo numero intero n possiamo ricavare in modo effettivo il programma $P_n = \gamma^{-1}(n)$ che gli corrisponde.

Osservazione 4.1. La funzione $\gamma : \mathcal{P} \to \mathbb{N}$, definita nel contesto del teorema 4.4, non è l'unica biiezione possibile; se ne possono costruire molte altre usando τ e β diverse. L'unica cosa per noi importante è che γ e γ^{-1} siano effettivamente calcolabili. D'ora in poi useremo sempre la γ descritta precedentemente, in modo che, assegnato n, P_n sia univocamente specificato.

La portata di tale teorema è notevole. Poiché sarebbe possibile, in linea di principio, costruire un computer con un *assembly* basato sul modello RAM, risulta evidente che qualunque programma scritto in qualunque linguaggio verrebbe poi compilato dall'assemblatore in una sequenza di istruzioni RAM, cioè in uno dei programmi $P \in \mathcal{P}$. Dunque ogni programma che scriviamo in un qualunque linguaggio ha un proprio numero di Gödel.

4.2.1 Il programma *Jurm* (Java URM)

Vediamo ora qualche esempio di costruzione effettiva del numero di Gödel di un programma e di inversione dello stesso numero. A tal fine può aiutare il software *Jurm* (Java URM), sviluppato da Charlton Rose e disponibile sul sito https://charltonrose.com/archives/1997/jurm
Jurm è un programma scritto in Java, che simula il funzionamento di una *Unlimited Register Machine* (URM), così come descritta nel testo *Computability*, di N. J. Cutland [5] e su questa dispensa come modello RAM. *Jurm* è in grado di associare il numero di Gödel $n = \gamma(P)$ a ogni programma P e viceversa di ricavare il programma $P_n = \gamma^{-1}(n)$ associato a un certo numero n. Nel contempo *Jurm* esegue il programma, assegnati i valori d'ingresso e un eventuale limite opzionale sul numero massimo di passi da svolgere, nel caso in cui la computazione diverga.
Jurm è un programma a riga di comando. La sintassi è la seguente:

`java urm` *program* [*param1* [*param2* [...]]] [`time` *steplimit*]

- *program* può essere un numero intero, che rappresenta il numero di Gödel del programma o il nome di un file di testo contenente l'elenco del programma, caratterizzato dall'estensione .urm

- *param1, param2, ...* sono i valori d'ingresso $a_1, a_2, ..., a_n$ forniti al programma RAM, ovvero i valori iniziali posti nei primi n registri del nastro nel caso si voglia calcolare un funzione n-aria. Tutti gli altri registri sono posti a 0.

- *steplimit*, se specificato, è il numero massimo di passi eseguiti dal programma. Questo parametro è utile nelle situazioni in cui $a_1, a_2, ..., a_n$ non stiano nel dominio e la funzione non sia definita, il che comporterebbe un ciclo infinito.

Jurm genera un riepilogo dell'esecuzione quando il programma simulato si arresta o quando viene raggiunto il limite dei passi *steplimit*, se impostato. Inoltre *Jurm* fornisce in uscita il numero di Gödel e l'indice $\beta(I_k)$ di ciascuna istruzione del programma.

Vediamo alcuni esempi.

Esempio 4.3. Prendiamo come primo esempio il seguente programma

```
1: S(1)          [#1]
2: C(1, 1, 1)    [#3]
```

che cicla all'infinito. I numeri messi tra parentesi quadre sono le codifiche delle istruzioni che si ottengono a partire dalla funzione β; il numero di Gödel programma si ottiene usando la funzione τ:

$$\beta(S(1)) = 4(1-1)+1 = 1$$
$$\beta(C(1,1,1)) = 4\,\xi(1,1,1)+3 = 4\,\pi(\pi(1-1,1-1),1-1)+3 = 4\pi(0,0)+3 = 3$$
$$\gamma(P) = \tau(1,3) = 2^1 + 2^{1+3+1} - 1 = 33$$

Procediamo ora all'inversione

$$\gamma^{-1}(33)+1 = 2+2^5 = 2^{b_1}+2^{b_2} \qquad a_1 = b_1 = 1 \quad a_2 = b_2 - b_1 - 1 = 5-2-1 = 3$$
$$\beta(I_1) = 1 = 4\cdot 0+1 = 4\cdot(1-1)+1 \qquad r = 1 \qquad \rightarrow S(1)$$
$$\beta(I_2) = 3 = 4\cdot 0+3 = 4\cdot\xi(1,1,1)+1 \qquad r = 3 \qquad \rightarrow C(1,1,1)$$

infatti

$$\xi^{-1}(3) = (\pi_1(\pi_1(0))+1, \pi_2(\pi_1(0))+1, \pi_2(0)+1)$$
$$\pi_1(0) = (0+1)_1 = (1)_1 = 0 \qquad \pi_1(\pi_1(0)) = \pi_1(0) = 0 \quad \pi_2(0) = 0$$

Dopo aver calcolato a mano i vari indici proviamo a usare il programma *Jurm* applicato al seguente file, che abbiamo denominato `esempio_4.3.urm` e conservato nella stessa directory di *Jurm*:

```
top:  S (1)
J (1, 1, top)
end:
```

Si osservi che nei file .urm bisogna usare la notazione $J(m,n,q)$ (che sta per *jump*) al posto di $C(m,n,q)$. L'esecuzione è la seguente:

```
java urm esempio_4.3.urm time 100
Resolving labels...done.
Loading program...done.
Executing program...aborted.

Execution summary:
------------------
program:  esempio_4.3.urm
input:  none
output:  unknown
time:  not less than 100 steps

Program listing:
----------------
1:  S (1) [#1]
2:  J (1, 1, 1) [#3]

Program code:
-------------
33
```

L'*output* è dichiarato *unknown* in quanto l'esecuzione corrisponde a un ciclo non terminante, bloccato dopo 100 passi; si noti inoltre la notazione #n, che rappresenta l'indice di ciascuna istruzione, cioè il valore assunto dalla funzione $\beta(I_k)$.

Esempio 4.4. Prendiamo ora

1: T(1,3) [#18]
2: S(4) [#13]
2: Z(6) [#20]

Le codifiche delle istruzioni e il numero di Gödel del programma sono:

$$\beta(T(1,3)) = 4\,\pi(0,2) + 2 = 4(2^0(2 \cdot 2 + 1) - 1) + 2 = 18$$
$$\beta(S(4)) = 4 \cdot 3 + 1 = 13$$
$$\beta(Z(6)) = 4 \cdot 5 + 0 = 20$$
$$\gamma(P) = \tau(18, 13, 20) = 2^{18} + 2^{18+13+1} + 2^{18+13+20+2} - 1 = 9007203549970431$$

Per verificare se i calcoli sono corretti possiamo usare *Jurm* innescandolo col numero di Gödel appena ottenuto; ecco l'esito della computazione:

```
java urm 9007203549970431 time 100
Decoding program...done.
Executing program...done.

Execution summary:
------------------
program:  9007203549970431
input:  none
output:  0
time:  3 steps

Program listing:
----------------
1:  T (1, 3) [#18]
2:  S (4) [#13]
3:  Z (6) [#20]

Program code:
-------------
9007203549970431
```

Si osservi che in questo caso l'opzione *time*, inserita precauzionalemente, non sarebbe stata necessaria in quanto il programma computa la funzione $0(x)$.

Esempio 4.5. Vediamo ora un esempio più impegnativo, usando il programma dell'esempio 2.4 che calcola la somma:

1: C(3,2,5) [#73727]
2: S(1) [#1]
2: S(3) [#9]
1: C(1,1,1) [#3]

Le codifiche delle istruzioni e il numero di Gödel del programma sono:

$$\beta(C(3,2,5)) = 4\,\xi(3,2,5) + 3 = 4\,\pi(\pi(2,1),4) + 3 = 4\,\pi(19,4) + 3 = 4 \cdot 2^{19}(2 \cdot 4 + 1) - 1) + 3 = 73727$$
$$\beta(S(1)) = 4 \cdot (1-1) + 1 = 1$$
$$\beta(S(3)) = 4 \cdot (3-1) + 1 = 9$$
$$\beta(C(1,1,1)) = 4 \cdot \xi(1,1,1) + 3 = 3$$
$$\gamma(P) = \tau(73727,1,9,3) = 2^{73727} + 2^{73727+1+1} + 2^{73727+1+9+2} + 2^{73727+1+9+3+3} - 1$$

Il numero di Gödel che si ottiene in questo caso è enorme; per vedere qual è possiamo far girare il seguente programma `add.urm`, innescandolo p.es. sul calcolo della somma 19+13:

```
top:  J (3, 2, end)
S (1)
S (3)
J (1, 1, top)
end:
```

Ecco l'esito della computazione:

```
java urm add.urm 19 13
Resolving labels...done.
Loading program...done.
Executing program...done.

Execution summary:
------------------
program:  add.urm
input:  (19, 13)
output:  32
time:  53 steps

Program listing:
----------------
1:  J (3, 2, 5) [#73727]
2:  S (1) [#1]
3:  S (3) [#9]
4:  J (1, 1, 1) [#3]

Program code:
-------------
760904883723846201277568630839350619069461069704682767390028120756021019787331003340309684793326301017905920722665014169273224909662565278696717668059656519583460995831417635937328808149067755161185727521357842126185836042501644528990322289831597248441061774215774811762166308725471816609466440386322700929621140572267526301925849109180661690481333824248187054678513534171004405573955408242995022073130345618021637953472849584112004738847447159066946060787526088793330812126857498633595247677060035847586650880342847906189924974492611111069793637089970543181015894033782817768984854706872645163676460854548914610611511068280086054118974528057715964356693632930985691804618383779981127525185815463444386455057812327825457069775996732207243982027971831397081623178627940146177551569831987915326942834061266593272783464122804924447841281549012415816820217693384580041958835885234470
```

573308491225423140276424741383215291418652188888637676127484101576724192182400436227876779051994479290216443503373398705965299199452452805505575052739959498044575812689740890764257491628098463556781324494099089949673984594322978224857757611409807915411781359417414568907696022744346707676201575762346714015767387279230088461324766918920326627861243822260159636964131925526748027031031756362530699007516230627311832985734673479700867710380531532576162135249720830634472122348736264704807068698456986973797276173100428091491620732655957767766998306476734240443739978129556856857140394842276917795578229223200330865228488510412718091861833047906371003743654701290423814116891248514237434855972524827123869513612153628043311590634749920103764651260758049799273433465124201773728411170576478907034465303523431212864797587298432504734663702560640236852348943715500455538950870903667623136255027324961966466274586735504491960089819957109216883939242420713259954071229787207438105447401258066327708667049024621103685335256445766790768199469178426913760209281022741817831740597459228740809003091420962393416937202240164144678614347393264639415292702771415716110411287636117451077143003866735654340869503175967847488180533508343199760020992316423876722832864395055277453380473414743192831605939499934276539076015618616324601364095127102222978841241133707739691273951477524497315720070020147631674052369123801398982291732150730083813676536989542798075732644239638905325343626206052118187567391623463913107011583950632147543565960685261220687086099479430873701321492638241684825475810405809601474617772804493034732820243613877737235866763210576010671351901989610924236498229129828847688548273323568427432922301101640892854370873188421309495338825861585219034818266832843628380246482149455273038638451788375864809751249459256643258883246942654373864674508905588316943307183241501316854832085636619403735663626094419233477810476035680326880525165673525753506062386342068592648942080737663224965752013142921926729002008616943385138577714086937194815572852908894951396141590532344013347138337387824244355739522640601599996582610241729826936764475397026880572638531399781731743122856501507927577249387384964174452803473520267330077544675912479876120676032891606699173774724843322156235219096305064346057133016972962676421832986768302310834899037742348862327246300604794975491863485043884031203074216834722331502042190709145907807968156282601105918742886065392441087256682552002588958798782982016375538806573690187843545281889031353882034692939687368037690461462170430333951348562170429421229873572685329903143008847281218116224558201094918820251834833041447225646612012941252375296998428432748416333569709496531297864169400456702294445328521599621180274794862914183908251692051905181236900555808674994871543857276564585763254876881115009345471604749610429554085725198679761250875751959696115920901653392554883985007175081337967217386961499941468014870100121548728554412987066282206167867224396125791201880416922798070428077147908367714888005633005771755886873395597410134622475944332302341000081195844450576111928877922627789247338965067728644650593383799385713443383923751271579940076626979657802058758447854697954995610563143538553829463021873482497464366591484751484872882922926566014037179716256149137978498811565769315960670087379234446512781105500810046813125543413124363154851795769742015229448244573696519187556816451514037330144462173211299693083535314897060395676919771562987128589443238005773606735196428336047379451045600189825885867890508196409703132289987534121108596305154922193631598348620684277859836032838709747427387047647786875889725190978205952178111010678479661581914201398199661018282492871012930003074618940355701228426694609104917431809135532601339260724050588161285000683940051708723780833870387885893185171613030881861602869563888111019745979576677783012480381869471933007356510024546686550530541818843133181199183617401568086978795831849895575977258400724698037374095619822223506410413356524171489899489841283593331872800213023354396913832409343859773021140254476248683590338975264634488715034874014119236299386478922086386600460514427534712113965195169972627122587238531999836755706513657102688537498953455532618296587555107514758615325005864132783204727491744293925166850605358099563003992792350745298570087645910872265635905845285995770048792654993922195735899027003637982045873297468173729965295738957354995255012761792816242199234060813298907536884036935534902521

7708702782072029160123513084533619223759475715744244576468240556767610271625895З
14330501954331077128731706620251592228611438358387185864292586936195456234709329
7893335303899212133941741695869868770295422436604363591445377142760359436216114З
46034347752147606699229569666667623763365342091861346395554069012960216240952969
57353680432807365516334793627523309794732066551882130876987348025329032344529908
49740069296683792455236975507471294761591155263127189746974833029809558923650113
76846420491092927442890697516351882094441864073716763133919764182941313397458892
29578720475904083835705749815799500019571460044864526410452410066155697460644866
96844871405941524654004427296772653049236298090453599539722798135832301576755983
88073892806053932547384156589711878698327031878414547742740189615054627012039391
15278095585645562999363021761731949598190808590404415960472862410207890377706114
99210413557116863035865060173850027407007072314679850642157277306225820578247057
92535838984052604751792654521331705923864709515511187897772510184899522857631261
86789145420707982975466849247300584075866280831319542242825454050716456191235347
53607915161404512639551894474274953444421163582229661255228073566447625260621629
96535631808363254606611938927834268575337261022550076167959786122591534965971461
42476434568360069675719338741156022841163421877852783924149137115366249828788024
49025638916936397098127976058845016710156079283695173630493297426279552721885905
28247324665794133510513532941582645629875430911931888509314217779967069579883670
98231731809800006945443093772170752925260827441443013672765583802806751787968990
03789553344241449869161546345495360719192480497328384477006744308797302332291123
76124666494034128030814626625265017880674514798694734976456265689064323521114789
43992112297091072403518060116126970787455611666497663016563743053826121162233651
15039357189832125096098137009158360473017307560080298584948629448678700833214352
96838429797554197846311132846901073650086884939426030418664009401824932791339245
16123070507723371022242315132798615310377081061388663783892810345668558136104330
48048189459077230232495197906477087103854421879084643243279541388112257084951797
04976718722684921436479575342177346584176927323552416138467145748228577142102709
09894803044553413191749948560089612170592006540536930232708449758398322451979471
92209507357074221573029980066954050176466214646379778537882364115731031643080922
70071620190323473107668288267852864514847192530815419868309972573031678902504733
76396085008117869894991712271330808420181061999244730214208294022465750274592954
99794072997976957711568656286106788696455995666621982342198562331342144511695797
95378687861040293268280913081688528553901443169111404906770348245445421337388357
47739870089434256021279975743727208508817867284153240081193356587546800241649073
27035905045513119478366593932513063435454165148698683039817123900324230399622095
94132219974615992495293947730217659108919188760840381953341765507490143713857823
00520127313709743510356204118490099281762704377926184188922288092493250040839446
18875654585117383983922469805709597990325474290431189017932741526410517323679227
58416116731292139689268272169953716099852996351190559430607740553652690674880568
17781790979421436922215801545864479742961864097294793781847167858098399891851075
81357902235928502939902002288011730900179920551010501984488862808062672318245316
58554562874398825044699072190253070008327555633565903643386848601769485073364809
93926516954829252305017333170133926697663458025555463185128479377616969862606237
05697473647639170223359994660121272659291045373513674268534668344199574577356099
29233514360308259316375877416698017602163510811906373679724044770355869210158101
64641445378909347189668280294258851441970930597500045335665920771092316957541993
35556720901219469216269545093395709512077609055733842979094825332662264317616579
54919118736011547730701915769097158255928027243652272653591649647103088706238755
74330371070663531281082785846893595129790665525137820002154237472033577866106429
55966032853955586074230901749161697695604327962937562245737153169710659355874637
72048621359256561960239730371426108452091968040414358731096646767245046112537651
98058972753108628063224108747476246917120604627315892198068587708165865965515810
06893965536614716926477190545680403798201406091337939881632112729090330176738862
93205261866479485484284090428705821218654063401206614332571605504391644074826192
09489102792209782817214416604507773814069314298679072770277080146655387763722111

352622623328967569474025241895339718919930326206274611590404099940222637399437356590489073142607417383245978601081107287544159277400413279480477023319757609106865220266270586720915068453672333057343430993527399525078348107897471543859852825327167398310754637529578717890871984589809468174810014871605991500519292626757904760897602092632408697821820831638878735204574488104266407493412456915622165871332326858543032622385496842074798744721701453610968920230473193523087701950365545412126685545274575584524014027040031648445926786806074549182526649664805326889387157770779543601721555284204996417078313919920850613721586955735621307899613360110134524466719247282589764546457406086636508463396484281704375998936552069760711179224201101956780331393953515315116743847135394585567638749461167238485331354721994358474962033919707346172257018872325017854975934060525720106222273824805942209973716282073917592706186123970598585202581889082352111196031996439863471127036201256001958287513139039325192701262827114802647466701251061181325028309200751263209238084779034783689172642593193688485763605447659998824402146756244040957956313185495903193940031327274706663886790762290524745880291635136685615119942334952384060949709528915216973063582710322789020106074408352580721303338696428004928891973264084074864728132935739463627680867980552398000862767944857090121550611638736338822626423856040984327676766118396150975470575255286098205134416869118068855841639636324027756423135468782879833801995251498880690768506886503150682553073907390491426181582743301907001494252229028434788922950443850235298559135874315392004936834092241581394826274188560699265074418773703898311384545077872758505534783618555586937610412714631362715649787321397804880362817856806922384967482160360271141942029334709917255359550769628270207354142349388261997402657884410972759123141451035528412497455968473563047732702036616634388963675509274387495694101141488689638039625122032931611048828971559685896332690994106866746734115130472828035685635387975894858966968378430980050309443658701739945672369170515882737253417981912912430550138515350071011450664326869981981641007206465699527850367562793826613644492348436627009214475415611824722541968397439764310560318709210771397123024427100142967277545507104030815630944961788760672841693485714743162960414873076609971027098251412535991000003445317119842319882802372073328831863095564016690442336652527552846855381802601862782940493869494886052255595116962629186604801570627709422115755312566749063537925095834507968029221946135839090545387304573244024704665245294943314330869973680967080501272868526703068400484385087925030966457060664825670216176167255271443281325768087193835865289682559201731828033873848474971683891384665675441969056008733855141946289988175780848867610497679711525585145959528822771445638277563096773575408886855215187276233545266220907879359947161044495334742671641041048337114976605856260792255329753262162177736417986478358282530652332385963962003357445164093466241452970413713558584633910901353290432519920819799362195985170282871821182151859557642768664867373181995893779981276250958830131419626743048033034914033253074952166012150665330171933830609702337311898083410858860439225549342957835320854997171317672062195894074529867587436930350632121124942412536914939749087716944065877444592446277811142708426712233994126716549888556257224642250054415814043470859581527605423575719757886571571899002079725963697405496091064025273646341119806205449617310324777460503577986120474539030316794033044684630402474117307896031630587507993702464315187988571783997426980417094388318867838625892101353230447149748585152928214451767735077023310851371027869823971158176150470137178698887257817261966370908566487210244576929163890168187727716836828073491972235427886601063668470452804359358461869835447480272514754411087270950180554572856183008804562385434157750868011414006357481296339029531097354703519195902748096118083909931394298022552224441840749664273228144618662001145226779324390520465324167352135207504606634932175912678151034092836629335162155577513211847258958244345015500763538868336686655597074803931565628529482799094125388629720278704577167436286084928031899986061449458298746853985682314589311983076762712773127945868688295112441936097645060833786185585283193767033047051729037436305450142705515118375700597036795923058980109904814595053822210956900281714766428205035982349134353127880804231177003049025192730915258215451267471673827341252875836773928

3189079269299870753380366047994223368319805945084962529226333795722590039122386939266997249023296141279432456283888013990875696604446754999358655992804750092452141086997911213924528368464979483844379779749849774943569612689871033539127105078271400077883997813263830565393532034506477924895023408552816612444643839988135844411635379701821611089844856040220179823318227654733879824958754747587188588279649562934344130167368767743934531406881158701652451743620594857153694737861999922010803573404332953065445241165391351366157011098002798672880823866108980169301386440084028301315423883464345020254855297848287135415463860111131855270549308875186676821296146206423308500518238610560545145012207567453862038115664101588921310072636387281862669083061948005709940450830161017482051575853451333493851645885293046790295129867952269100866725512755319675480230383529486285601397447723193557056044711238548102254941882345465491824296522283956855895559609128478878868482714870682042153698538263045805758788352354988729326802683346254929448726169277073894966299978686518642869785011871728060166375437735422902484572702121195927419748166781044577987622835200842828249009894896664246398724940615377752793632130931566169751077353594218169945649199969846082822435223027646268016643731590857499138877707957230008832489503034995562485991188801464188740071271629623220968848904559097723549623350010837943140038720530548574880720281870603753666446407030914105783924851605202967095612476684651901801307373963539172019492101623779631016704088047257982025540341933702522760823551218798351758602106106538465482760898917074691533607158187001247080003957693413151686108333930246424212384912947025970926565691307365236131858855816755395786933021973626017446197105927470856965101301960244438929719256774261653800992909919077298668641600897223980618978555141395888385373254032121450760248951885407146260023062109719493981234198579098960907130697500677181737693312636527309754302124204202295616456202256763315963572791584836655817824629830095647592883411513600719921599586605343708468557308318259880019465058943631468170700997528818612938011930919598465757866014081435157259352206611898852089674723453284335038840981986151146295061686637042295686224527902884226360209733771425323920271797130887348157279409540916075988166729682068140345548079166736253076755198071925310430908241969320754535559957594540848638196529875092017900336737665267098663870001724484013840244863246729980646682940800738850054392157580378001277927897137598963677901630880886114753013903993072953267745651396863475444386162448349119495431650469434187785867884334386468021294227004047196149164216682408025186862087823033357303794857129720117097538143936017843998603791836095031905766241314928674448249939842096080399121207220550486904031855425797452592265394218869869049141947312385055912001904272185840262390873686579922787865150433013597732733730648807009265121942082455435527199330862154267653402466988248001804053767208222933714154119920904426078678103142378805383619615522500179734955534869450339453201031633323073624515035148931061926232159663909001837181872611254004301071563121962086100687128733941250604427609977310385622810755563417650701404371840272308206429254117959734699885616306155772019542201681866835740251351468497584226330488769589346196860911044316234264333211709198502632397878555332924497357756481713035465794144369444688661217652731066410706359540701636892853529614033988917925999085476272721698447174779231298965096073164434617703839358865988209064573554865085853701754831596026500414026128895481141422594023253023508714117705408981335717301407895157140903912720229472707330885113979877701511776592431477269323007205938263595752483323157767199230390564352103348135155906259129493751649523469569255693560842903342807234900164282170568169819944971293477438857655055778294097222385911360948801310214076435318492092610170628748039444770525331130013939498138326326735197992529525147223617631412623649025333697511367027474537345192888192808932592816079962265694430021187530627925089026573699257601305741404230840764340877808502059814909150252564305969817066897981805356726692225138176130748395642003478211952300396253193265609009990253308945342491129584194854863367973536673822396479747056885908960944490752026127807556906633990133376367437574079764970277026467640404359079864768516133223212400238012332190238704120241883801626097161118893672758706556902117847814422543556463829559893577342533103122053127458617760062078

```
618669545757910719676718359705560782897861523557314500009191689804744196406253135072214924785319357733963299995448727966373691649730447706732990638522429465023821029835390061191077773726692524416144172071006011324918869712549334320422844012858725774933029158261657232042039060110469300510536239215607736068191254116632329015193665011393530350753789669389120487063348718064472517516095604845264930115662517929491371504022313168875707602367581191003203511232778586834638386976384674039017726012029641196168688555843542587504810786985049548436298119157730560147184703370650081394153729693324002671161618186415342824821996343437830980907892266225006116520253248534705858063036487698593759135886654332542332772393984859692139819081581061230701620185631801818539368990016527242840283549197720826566745312636877684846009324659799118896658007283837774928362805361566732107827833280067498634836955296887819320091154814650187042142476705881310722447944642683286387297186803261105721887746165960816473831055401982984812194941406603894434457983021483677438492850643275751651236778954657987082789045473575436470393167750960201155342032335644278118701051847495347953883753033918029893901872888447417633312032191488347462369592552050867441546067360789738632347012393225157107307959827587177226572678046763415081125789267594749589565151018793464355231658893172341568772981985906475257769791027398824218166321081129344258734876563936280305291064332307261697207917153083014655514953728983436269670549644674534477968179154790828684931512009654499981523685283435346310382344769874095951212674194856437256662507151301315449267469046545425944359101202850321937307166913327281425197855320646080524368589366369882054217013764178945073630666479581772865547388971546960062985074148797904921745880363735904456553649918897402519601220048284421167728354450201805032702563228157752701311735056586996788348417219048631842501260688720961327963209201451617729147197449685096462005982745710213319121117087156246520224666743358373434870510381224798762689582174115989559480727400083205785776974567157251036359248369959557384585344543659001804611311954994122410426292888686534601215384067533707514836653528717578353040270472127476690753822033854513264100031899362860261827881400765837524928800545288587519193968191620204107393850095027553136387602180749457378332126438449551342754156156516786712595585192339264003052200228514666612046173988967486671088026643827503840777824128464229873675533105743676071885064334184310789085504459281748652960238171364100266171687196609909030604354765210772244908551825744806728294962480534532717619820574196411322055815288578391730209541176604469986005736835456418037708825602518588210188855819137844780353696648879094677192060496077742986450797477831757807977229879561327543832364900256490416348501524413204839812759638686549366880406020701158142188512427721480717480964895533471153614941348668339159009912870173392008377954973533535028461023648198101932601031888142378985832543551302459939678855341833824934991373336487631780325626263152527518601489953287966492680923451388244486823951584313518692113074285444791246253379122460440710384489787797786209815649434318699407741381527147327742459347773666255329154716974163452061279702552464706803842894781880094672508861913459189621445193467791910263361273888425901382747683355723690058387065313394471840912406662639497836691492076789672987693996967351
```

che è un numero di 22199 cifre!

Dall'esempio appena trattato risulta chiara la crescita esplosiva dell'indice di Gödel anche per programmi di modestissima complessità e lunghezza. A livello di curiosità, riportiamo nella tabella di figura 4.3 i programmi relativi ai primi 36 numeri naturali. Come si può ben vedere, ci sono diversi programmi che calcolano la stessa funzione; p.es. P_0, P_2 e P_5 calcolano la $0(x)$, P_4, P_{10} e P_{21} calcolano la $1(x)$, mentre P_1, P_{17} e P_{19} calcolano la $x+1$. La presenza del trattino "$-$" indica una funzione non definita per tutti i valori di x.

n	0	1	2	3	4	5	6	7	8	9	10	11
$f(x)$	0	$x+1$	0	x	1	0	0	$-$	0	$x+2$	1	0
P_n	$Z(1)$	$S(1)$	$Z(1)$ $Z(1)$	$T(1,1)$	$Z(1)$ $S(1)$	$S(1)$ $Z(1)$	$Z(1)$ $Z(1)$ $Z(1)$	$C(1,1,1)$	$Z(1)$ $T(1,1)$	$S(1)$ $S(1)$	$Z(1)$ $Z(1)$ $S(1)$	$T(1,1)$ $Z(1)$

n	12	13	14	15	16	17	18	19	20	21	22	23
$f(x)$	0	0	0	x	$-$	$x+1$	0	$x+1$	2	1	1	$-$
P_n	$Z(1)$ $S(1)$ $Z(1)$	$S(1)$ $Z(1)$ $Z(1)$	$Z(1)$ $Z(1)$ $Z(1)$ $Z(1)$	$Z(2)$	$Z(1)$ $C(1,1,1)$	$S(1)$ $T(1,1)$	$Z(1)$ $Z(1)$ $T(1,1)$	$T(1,1)$ $S(1)$	$Z(1)$ $S(1)$ $S(1)$	$S(1)$ $Z(1)$ $S(1)$	$Z(1)$ $Z(1)$ $Z(1)$ $S(1)$	$C(1,1,1)$ $Z(1)$

n	24	25	26	27	28	29	30	31	32	33	34	35
$f(x)$	0	0	1	0	0	0	0	x	0	$-$	$-$	x
P_n	$Z(1)$ $T(1,1)$ $Z(1)$	$S(1)$ $S(1)$ $Z(1)$	$Z(1)$ $Z(1)$ $S(1)$	$T(1,1)$ $Z(1)$ $Z(1)$	$Z(1)$ $S(1)$ $Z(1)$	$S(1)$ $Z(1)$ $Z(1)$	$Z(1)$ $Z(1)$ $Z(1)$	$S(2)$	$Z(1)$ $Z(2)$	$S(1)$ $C(1,1,1)$	$Z(1)$ $Z(1)$ $C(1,1,1)$	$T(1,1)$ $T(1,1)$

Figura 4.3: Programmi RAM relativi ai primi 36 numeri naturali; sono indicate anche le funzioni calcolate.

4.3 Gödelizzazione delle funzioni

Con la numerazione dei programmi garantita dalla funzione γ possiamo ora numerare anche le funzioni associate a questi programmi. A tal fine introduciamo un'importante notazione che useremo per tutto il resto della trattazione.

Definizione 4.2. *Per ogni $a \in \mathbb{N}$ e $n \geqslant 1$ definiamo*

1. $\phi_a^{(n)}$ = *la funzione n-aria computata da $P_a = f_{P_a}^n$ della notazione* (2.4)
2. $W_a^{(n)}$ = *il dominio di* $\phi_a^{(n)} = \{(x_1, x_2, ..., x_n) : P_a(x_1, x_2, ..., x_n) \downarrow\}$
3. $E_a^{(n)}$ = *l'immagine di* $\phi_a^{(n)} = \left\{\phi_a^{(n)}(\boldsymbol{x}) : \boldsymbol{x} \in \mathbb{N}^n\right\}$

Esempio 4.6. Sia $a = 4127$. Il programma associato si ricava invertendo la funzione γ:

$$\begin{aligned}
&\gamma^{-1}(4127)+1 = 2^5+2^{12} = 2^{b_1}+2^{b_2} && a_1 = b_1 = 5 \quad a_2 = b_2 - b_1 - 1 = 12-5-1 = 6\\
&\beta(I_1) = 5 = 4\cdot 1+1 = 4\cdot(2-1)+1 && r = 1 \quad \rightarrow S(2)\\
&\beta(I_2) = 6 = 4\cdot 1+2 = 4\cdot\pi(2-1,1-1)+2 && r = 2 \quad \rightarrow T(2,1)
\end{aligned}$$

poiché $\pi^{-1}(1) = (\pi_1(1), \pi_2(1)) = ((1+1)_1, \pi_2(1)) = \left(1, \frac{1}{2}\left(\frac{(1+1)}{2^1} - 1\right)\right) = (1,0)$

Il programma associato è dunque $P_{4127} = S(2)\,T(2,1)$. Con la notazione appena introdotta si ha che

$$\phi^{(1)}_{4127}(x) = 1 \qquad \forall x$$
$$\phi^{(n)}_{4127}(x_1, x_2, ..., x_n) = x_2 + 1$$
$$W^{(1)}_{4127} = \mathbb{N} \qquad\qquad E^{(1)}_{4127} = \{1\}$$
$$W^{(n)}_{4127} = \mathbb{N}^n \qquad\qquad E^{(n)}_{4127} = \mathbb{N}^+ \text{ se } n > 1$$

Se f è una funzione unaria computabile, allora esiste P tale che $f = \phi_a$ e $a = \gamma(P)$. Diciamo che a è un *indice* per f. Tuttavia, poiché ci sono infiniti programmi che calcolano la stessa funzione f, non possiamo dire che a *è l'*indice di f. Come conseguenza si può affermare che ogni funzione unaria computabile compare nella enumerazione

$$\phi_0 \quad \phi_1 \quad \phi_2 \quad \phi_3 \quad ... \tag{4.14}$$

associata ai programmi $\qquad P_0 \quad P_1 \quad P_2 \quad P_3 \quad ...$

e che l'enumerazione delle ϕ è una enumerazione con ripetizioni. Una simile valutazione si applica anche al caso di funzioni n-arie

$$\phi^{(n)}_0 \quad \phi^{(n)}_1 \quad \phi^{(n)}_2 \quad \phi^{(n)}_3 \quad ...$$

Si osservi che la presenza delle ripetizioni ci impedisce di usare tale enumerazione per creare una corrispondenza biunivoca con gli elementi di $\mathbb{N}$, cioè di poter dichiarare numerabile l'insieme $\mathcal{C}_n$ delle funzioni n-arie. Per risolvere questo problema dobbiamo elidere le ripetizioni dall'elenco.

Teorema 4.5. *$\mathcal{C}_n$ è numerabile.*

Dimostrazione. Partiamo dalla enumerazione $\phi^{(n)}_0\ \phi^{(n)}_1\ \phi^{(n)}_2\ \phi^{(n)}_3\ ...$ che ha ripetizioni, e ricaviamone una senza. L'idea è quella di prendere una dopo l'altra le $\phi^{(n)}_z$, conservando solo le funzioni che non sono sono già apparse in precedenza; in tal modo si eliminano tutte le ripetizioni j tali che $\phi^{(n)}_j = \phi^{(n)}_z$ per qualche valore di z associato a una funzione che è già comparsa. Per risolvere il problema possiamo far uso dell'operatore di minimazione. Sia

$$f(0) = 0$$
$$f(m+1) = \mu z \left(\phi^{(n)}_z \neq \phi^{(n)}_{f(0)}, \phi^{(n)}_{f(1)}, ..., \phi^{(n)}_{f(m)}\right)$$

In tal modo appare che

$$\phi^{(n)}_{f(0)} \quad \phi^{(n)}_{f(1)} \quad \phi^{(n)}_{f(2)} \ ... \ \phi^{(n)}_{f(m)} \ ...$$

è una enumerazione di $\mathcal{C}_n$ senza ripetizioni. □

Osservazione 4.2. Contrariamente a quanto accaduto nei teoremi 4.3 e 4.4, ora non siamo stati in grado di dimostrare l'*effettiva* numerabilità di $\mathcal{C}_n$; questo perché la funzione $f(m)$ che abbiamo appena introdotto non è computabile. Dimostreremo infatti, nel corollario 5.1, che il predicato $'\phi_x = \phi'_y$ è indecidibile. Ciononostante esiste una costruzione di una funzione h, ideata da Friedberg, tale che $\phi^{(n)}_{h(0)} \quad \phi^{(n)}_{h(1)} \quad \phi^{(n)}_{h(2)} \ ... \ \phi^{(n)}_{h(m)} \ ...$ è una enumerazione di $\mathcal{C}_n$ senza ripetizioni. Usando questa costruzione si perverrebbe a una numerabilità effettiva.

Arrivati a questo punto abbiamo acquisito la numerabilità di $\mathcal{C}_n$ per ogni valore dell'arietà $n = 1, 2, 3, ...$ della funzione. Ciò significa che qualunque funzione computabile di qualunque arietà sta nel seguente elenco

$$\begin{array}{ccccccc}
\phi^{(1)}_{f_1(0)} & \phi^{(1)}_{f_1(1)} & \phi^{(1)}_{f_1(2)} & \cdots & \phi^{(1)}_{f_1(m)} & \cdots \\
\phi^{(2)}_{f_2(0)} & \phi^{(2)}_{f_2(1)} & \phi^{(2)}_{f_2(2)} & \cdots & \phi^{(2)}_{f_2(m)} & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \cdots \\
\phi^{(n)}_{f_n(0)} & \phi^{(n)}_{f_n(1)} & \phi^{(n)}_{f_n(2)} & \cdots & \phi^{(n)}_{f_n(m)} & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \cdots
\end{array}$$

e viene identificata dalla coppia di parametri m, n. Poiché

$$\mathcal{C} = \bigcup_{n \geqslant 1} \mathcal{C}_n$$

si ha che l'unione numerabile di insiemi numerabili è numerabile. Per dimostrare la numerabilità di $\mathcal{C}$ è allora sufficiente individuare una corrispondenza biunivoca tra la coppia m, n e i numeri naturali.

Teorema 4.6. *$\mathcal{C}$ è numerabile.*

Dimostrazione. Per ogni n sia f_n la funzione usata nella dimostrazione del teorema 4.5 per individuare una enumerazione senza ripetizioni di $\mathcal{C}_n$. Come biiezione tra la coppia m, n e gli elementi di $\mathbb{N}$ possiamo usare la funzione $\pi : \mathbb{N}^2 \to \mathbb{N}$ introdotta nella (4.2). Definiamo allora la funzione $\theta : \mathcal{C} \to \mathbb{N}$ nel seguente modo

$$\theta\left(\phi^{(n)}_{f_n(m)}\right) = \pi(m, n-1) \tag{4.15}$$

Chiaramente θ è una biiezione. □

Osservazione 4.3. Poichè la θ si basa sulla funzione π, essa è una biiezione effettiva. Se dunque usassimo la costruzione di Friedberg citata nell'osservazione 4.2 potremmo dedurre che anche la $\mathcal{C}_n$ è effettivamente numerabile.

4.4 Funzioni totali e funzioni parziali

Nella descrizione della nozione informale d'algoritmo di figura 2.1 abbiamo sottolineato la necessità di accettare nel modello i casi in cui la computazione incappi in un ciclo infinito o *loop*, dal quale non si può uscire. Una tale situazione non è una per nulla gradita, poiché corrisponde a quello che noi chiamiamo un *baco* del programma, che ovviamente sarebbe meglio evitare. Chiaramente il punto 11) della tabella 2.1 non corrisponde all'idea iniziale di algoritmo, che deve esibire una risposta in tempo finito (anche se non limitabile superiormente). Dalla definizione 2.2 sappiamo che questa situazione corrisponde al calcolo di una funzione parziale al di fuori del proprio dominio, cioè a una computazione divergente.
Storicamente accadde che, in un primo momento, i matematici impegnati nella definizione di computabilità (soprattutto Gödel e Kleene) tentarono di costruire un modello di computazione basato sulle sole funzioni totali - le cosiddette *funzioni totali ricorsive* che indichiamo con il simbolo $\mathcal{R}_0$ (si veda la sezione 6.2). Vedremo ora che, così facendo, ci si troverebbe in una situazione in cui alcuni problemi palesemente risolubili mediante approccio procedurale *non* lo sarebbero all'interno del nostro modello basato sulle sole funzioni totali. Fissata una certa arietà n consideriamo la seguente funzione g

$$\begin{aligned}
&g(0) = 0 \\
&g(m+1) = \mu z\,(z > g(m) \text{ AND } \phi_z \text{ è totale })
\end{aligned}$$

In tal modo appare che

$$\phi^{(n)}_{g(0)} = \varphi^{(n)}_0 \quad \phi^{(n)}_{g(1)} = \varphi^{(n)}_1 \quad \phi^{(n)}_{g(2)} = \varphi^{(n)}_2 \ \ldots \ \phi^{(n)}_{g(m)} = \varphi^{(n)}_m \ \ldots$$

è una enumerazione delle sole funzioni totali n-arie. Consideriamo per semplicità $n = 1$. La tabella sotto rappresentata contiene allora l'insieme di tutte le funzioni totali unarie, calcolate per tutti i valori $0, 1, 2, \ldots$ della variabile d'ingresso x. Il fatto che tutte le funzioni siano totali garantisce che sono definite su tutto il dominio e dunque non esiste alcuna combinazione m, n per la quale si trovi il trattino "$-$" in corrispondenza del valore della funzione $\varphi_n(m)$.

	0	1	2	...	m	...
φ_0	$\varphi_0(0)$	$\varphi_0(1)$	$\varphi_0(2)$	$\cdots$	$\varphi_0(m)$	$\cdots$
φ_1	$\varphi_1(0)$	$\varphi_1(1)$	$\varphi_1(2)$	$\cdots$	$\varphi_1(m)$	$\cdots$
φ_2	$\varphi_2(0)$	$\varphi_2(1)$	$\varphi_2(2)$	$\cdots$	$\varphi_2(m)$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	...
φ_n	$\varphi_n(0)$	$\varphi_n(1)$	$\varphi_n(2)$	$\cdots$	$\varphi_n(m)$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	...

Costruiamo ore la seguente funzione

$$f(x) = \varphi_x(x) + 1$$

E' ben evidente che tale funzione è palesemente calcolabile secondo il senso comune del termine. Infatti, per sapere quanto vale p.es. $f(3)$ basta calcolare il valore di $\varphi_3(3)$ e sommare 1; in pratica ciò corrisponde a innescare il programma associato a φ_3 con il valore 3 e ad aspettare la fine della computazione, che si avrà in un numero finito di passi in quanto φ_3 è totale. Finita la computazione avremo il valore $\varphi_3(3)$ nella prima cella e a esso sommeremo uno. Poichè $f(x)$ è calcolabile, ed è definita per tutti i valori di $\mathbb{N}$, essa è totale e sta da qualche parte nella tabella sopra. Esiste dunque un indice e per la funzione φ tale che

$$f(x) = \varphi_e(x) = \varphi_x(x) + 1$$

Se ora tentiamo di calcolare $f(e)$ troviamo il seguente paradosso

$$f(e) = \varphi_e(e) = \varphi_e(e) + 1$$

Ciò significa che la funzione f, pur essendo palesemente calcolabile secondo il senso comune, *non lo è all'interno del modello che usa le sole funzioni computabili di tipo totale*.
Questa insanabile contraddizione si può risolvere solamente accettando il fatto che la f non sia definita in corrispondenza del valore e.
La conseguenza d questo semplice paradosso è enorme: *non è possibile costruire un modello di computazione che si basi sulle sole funzioni totali, ma bisogna forzatamente introdurre anche le funzioni parziali*.

Osservazione 4.4. Il ragionamento di cui sopra fa per la prima volta uso della *diagonalizzazione di Cantor* o *ragionamento diagonale*, introdotto nel teorema 4.1 per stabilire la non numerabilità di $\mathbb{R}$. Se $\lambda_0, \lambda_1, \lambda_2, \ldots, \lambda_n, \ldots$ è una enumerazione di oggetti, l'idea in generale è quella di costruire un oggetto λ che non sta nella lista usando il motto

"costruisci λ e λ_n diversi in n"

Nel seguito useremo frequentemente il ragionamento diagonale.

4.5 Esistenza di funzioni non computabili

Nella sezione 4.3 abbiamo dimostrato che l'insieme $\mathcal{C}$ delle funzioni computabili è numerabile. Per capire se questo risultato fissi o meno una limitazione all'approccio procedurale-algoritmico dobbiamo indagare sulla cardinalità dell'insieme

$$\mathcal{F} = \bigcup_{n \geqslant 1} \mathcal{F}_n$$

cioè dell'insieme di *tutte* le funzioni del tipo $f : \mathbb{N}^n \to \mathbb{N}, n = 1, 2, 3, \ldots$, con $\mathcal{F}_n = \{f : \mathbb{N}^n \to \mathbb{N}\}$. Se infatti anche $\mathcal{F}$ fosse numerabile, potremmo affermare che con i computer si possono risolvere *tutti* i problemi codificabili nei termini del calcolo di una funzione f da $\mathbb{N}^n$ in $\mathbb{N}$, per qualche valore di n. Viceversa, se si riuscisse a dimostrare che $\mathcal{F}$ ha la cardinalità del reale, se ne dedurrebbe che *esistono funzioni non computabili* (problemi non risolubili - predicati indecidibili). Come vedremo tra poco siamo proprio in questa situazione.

Teorema 4.7. *$\mathcal{F}$ è non numerabile.*

Dimostrazione. La dimostrazione segue il processo di diagonalizzazione introdotto nel teorema di Cantor 4.1. È sufficiente dimostrare il teorema limitatamente a $\mathcal{F}_1$. Ovviamente $\mathcal{F}_1$ è infinito, poiché esso contiene almeno tutte le funzioni costanti del tipo $\boldsymbol{k}(x) = k,\ \forall x$. Supponiamo allora, per assurdo, che $\mathcal{F}_1$ sia numerabile; dunque $\mathcal{F}_1 = \{f_0, f_1, \ldots, f_n, \ldots\}$ contiene tutte le funzioni unarie da $\mathbb{N}$ in $\mathbb{N}$.

	0	1	2	...	n	...
f_0	$f_0(0)$	$f_0(1)$	$f_0(2)$	$\cdots$	$f_0(n)$	$\cdots$
f_1	$f_1(0)$	$f_1(1)$	$f_1(2)$	$\cdots$	$f_1(n)$	$\cdots$
f_2	$f_2(0)$	$f_2(1)$	$f_2(2)$	$\cdots$	$f_2(n)$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\cdots$
f_n	$f_n(0)$	$f_n(1)$	$f_n(2)$	$\cdots$	$f_n(n)$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\cdots$

Costruiremo ora una funzione $g(n)$ che non sta nella tabella, contraddicendo così l'ipotesi assurda. Consideriamo la funzione

$$g(n) = \begin{cases} f_n(n) + 1 & \text{se } f_n(n) \text{ è definita} \\ 0 & \text{se } f_n(n) \text{ non è definita} \end{cases}$$

costruita con il ragionamento diagonale di rendere la nuova funzione diversa nella coordinata n-esima, per ogni n. È evidente che $g(n)$ non può stare nell'enumerazione di $\mathcal{F}_1$; infatti si ha

$$g(n) \neq f_n(n),\ \forall n \ \Rightarrow g \neq f_n,\ \forall n$$

perchè se $f_n(n)$ è definita, le due funzioni hanno valore diverso; se viceversa non è definita, sono diverse in quanto $g(n)$ lo è. Ciò è in contraddizione con l'ipotesi che $\mathcal{F}_1$ contenga tutte le funzioni unarie. □

La dimostrazione di questo teorema può essere intuita anche senza ricorrere al ragionamento diagonale. Per convincercene consideriamo un qualunque numero reale nell'intervallo $[0...1]$ nella forma $0, \ldots$, per esempio

$$x = 0,46279100474766\ldots$$

I numeri $462791004\ldots$ possono essere usati per attribuire un valore a ciascun elemento $f(n)$ della funzione $f : \mathbb{N} \to \mathbb{N}$ secondo lo schema seguente

	0	1	2	3	4	5	6	7	8	$\cdots$
f	$f(0)=4$	$f(1)=6$	$f(2)=2$	$f(3)=7$	$f(4)=9$	$f(5)=1$	$f(6)=0$	$f(7)=0$	$f(8)=4$	$\cdots$

Poiché scegliendo un $x \in [0...1]$ diverso (anche per una sola cifra) si ottiene una funzione diversa e poiché l'intervallo $[0...1]$ ha, a norma del teorema 4.1, la cardinalità di $\mathbb{R}$, si ricava che anche $\mathcal{F}_1$ ha la stessa cardinalità.

I teoremi 4.6 e 4.7, che stabiliscono rispettivamente la numerabilità di $\mathcal{C}$ e la non numerabilità di $\mathcal{F}$, offrono implicitamente un risultato teorico di enorme portata: attestano *l'esistenza di funzioni non computabili*. Anzi, in un certo senso, visto lo scarto che esiste tra $\mathbb{R}$ e $\mathbb{N}$, *quasi* tutte le funzioni sono non computabili. La costruzione diagonale di Cantor ci consente di costruire il primo esempio di funzione non computabile.

Teorema 4.8. *Esiste una funzione unaria totale non computabile.*

Dimostrazione. Consideriamo l'enumerazione $\phi_0, \phi_1, \phi_2, \ldots$ delle funzioni unarie introdotta nella (4.14) e calcoliamo i valori delle funzioni per tutti i valori x d'innesco. Si ottiene la seguente tabella

	0	1	2	$\ldots$	n	$\ldots$
ϕ_0	$\phi_0(0)$	$\phi_0(1)$	$\phi_0(2)$	$\cdots$	$\phi_0(n)$	$\cdots$
ϕ_1	$\phi_1(0)$	$\phi_1(1)$	$\phi_1(2)$	$\cdots$	$\phi_1(n)$	$\cdots$
ϕ_2	$\phi_2(0)$	$\phi_2(1)$	$\phi_2(2)$	$\cdots$	$\phi_2(n)$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ldots$
ϕ_n	$\phi_n(0)$	$\phi_n(1)$	$\phi_n(2)$	$\cdots$	$\phi_n(n)$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ldots$

Ricordiamo che i valori del tipo $\phi_n(m)$ sono assegnati se e solo se la funzione ϕ è definita per la coppia n, m, altrimenti si trova il trattino "$-$". Costruiamo ora una funzione f che è differente da ogni funzione $\phi_0, \phi_1, \phi_2, \ldots$ dell'enumerazione, usando il ragionamento diagonale. Esplicitamente definiamo[1]

$$f(n) = \begin{cases} \phi_n(n) + 1 & \text{se } \phi_n(n) \downarrow \\ 0 & \text{se } \phi_n(n) \uparrow \end{cases}$$

La costruzione di f è tale che, per ogni n, f differisce da ϕ_n in n; infatti se $\phi_n(n)$ è definita, $f(n)$ e $\phi_n(n)$ hanno un valore diverso in n; se invece $\phi_n(n)$ non è definita, f è diversa in quanto essa è definita (e vale 0). Poiché f è diversa da ogni ϕ_n essa non può stare nell'enumerazione di $\mathcal{C}_1$ della tabella ed è quindi non computabile. Chiaramente f è totale. □

Osservazione 4.5. La f del teorema è stata costruita prendendo i valori cerchiati sulla diagonale della tabella sopra, cioè $\phi_0(0), \phi_1(1), \phi_2(2), \ldots$, e cambiandoli sistematicamente con $f(0), f(1), f(2), \ldots$. Si noti che c'è una totale libertà nell'attribuire i valori di f, nel senso che avremmo potuto scrivere

$$f(n) = \begin{cases} \phi_n(n) + a & \text{se } \phi_n(n) \downarrow \\ b & \text{se } \phi_n(n) \uparrow \end{cases}$$

[1] Con un abuso di notazione scriviamo $\phi_n(n) \downarrow$ per intendere che la funzione è definita e quindi la computazione converge; al contrario $\phi_n(n) \uparrow$ indica che la computazione diverge e la $\phi_n(n)$ non è definita.

con a, b che assumono qualunque valore. In questo modo anche

$$f(n) = \begin{cases} \phi_n(n) + 27^n & \text{se } \phi_n(n) \downarrow \\ n^2 & \text{se } \phi_n(n) \uparrow \end{cases}$$

è un'altra funzione non computabile.

4.6 Teorema del parametro

Quando si installa un programma F su un computer si parte da un programma sorgente di tipo generale e, a seconda delle caratteristiche *hardware* della macchina, il programma di installazione fissa alcuni parametri di F in modo da rendere il funzionamento dello stesso compatibile con le caratteristiche della macchina sulla quale F va installato. Supponiamo che prima dell'installazione F abbia $m + n$ parametri liberi e, a seguito dell'installazione, m di questi vengano fissati nel nuovo programma "specializzato" che chiamiamo Q. Il teorema del parametro ci fornisce la possibilità di ottenere l'indice di Gödel di Q, che lavora solo con n parametri d'ingresso. Dal punto di vista della teoria della computabilità, il teorema del parametro è un utile strumento tecnico per dimostrare altri risultati rilevanti nella nostra indagine sui limiti dell'approccio procedurale-algoritmico.
Iniziamo col caso monodimensionale e supponiamo che $f(x, y)$ sia una funzione computabile (non necessariamente totale); se fissiamo il valore di x in modo che sia $x = a$, otteniamo una funzione

$$g_a(y) \simeq f(a, y)$$

che dipende dal valore a. Poiché g_a è computabile, esiste un indice di Gödel e tale che

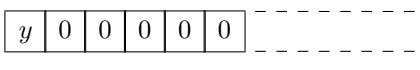
$$\phi_e(y) \simeq f(a, y)$$

con $e = k(a)$ per qualche funzione k. Il teorema del parametro ci consente di ricavare e in modo effettivo da a. Questo è un caso particolare di un teorema noto in letteratura anche con la denominazione di *teorema* $s-m-n$; dimostriamo prima il teorema nel suo caso particolare.

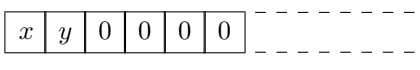
Teorema 4.9. *Sia $f(x, y)$ una funzione computabile. Allora esiste una funzione totale computabile $k(x)$ tale che*

$$f(x, y) \simeq \phi_{k(x)}(y)$$

Dimostrazione. Per ciascun valore prefissato di x, $k(x)$ sarà l'indice del programma Q il quale, data la configurazione iniziale



computa $f(x, y)$. Sia F il programma che computa f. Allora per comporre il programma Q dobbiamo scrivere F preceduto dalle istruzioni che ci consentono di trasformare la configurazione precedente nella seguente



Il programma Q è allora il seguente

e $k(x)$ è l'indice di Gödel del programma Q, che si può ricavare in modo effettivo mediante Gödelizzazione dello stesso. □

$$
\begin{array}{ll}
 & T(1,2) \\
 & Z(1) \\
 & S(1) \\
x \text{ volte} \left\{ \right. & \vdots \\
 & S(1) \\
 & F
\end{array}
$$

Esempio 4.7. Consideriamo la funzione $f(x,y)=x+y$; il programma F che la calcola è quello dell'esempio 2.4 che riportiamo qua di seguito

1	$C(3,2,5)$
2	$S(1)$
3	$S(3)$
4	$C(1,1,1)$

Fissiamo ora $x=3$; il programma specializzato che calcola $f(3,y)=3+y$ è il seguente

1	$T(1,2)$
2	$Z(1)$
3	$S(1)$
4	$S(1)$
5	$S(1)$
6	$C(3,2,10)$
7	$S(1)$
8	$S(3)$
9	$C(1,1,6)$

Usando il programma Jurm possiamo trovare l'indice di Gödel che gli corrisponde; esso $4919883....802687$, che è un numero di 46877 cifre.

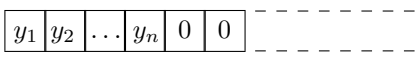
Il teorema del parametro si può generalizzare facilmente al caso in cui x e y siano due vettori $\boldsymbol{x}$ e $\boldsymbol{y}$ di m e n coordinate. Se poi consideriamo che la generica funzione $f(x,y)$ può essere descritta tramite il proprio indice di Gödel, diciamo e, al posto di $f(x,y)$ dobbiamo considerare la funzione $\phi_e^{(m+n)}(\boldsymbol{x},\boldsymbol{y})$ e il problema di trovare, per ogni coppia $e,\boldsymbol{x}$, un intero z tale che

$$\phi_e^{(m+n)}(\boldsymbol{x},\boldsymbol{y}) \simeq \phi_z^{(n)}(\boldsymbol{y})$$

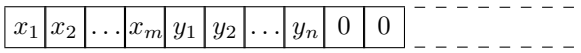
Teorema 4.10. *Per ogni coppia $m,n \geqslant 1$ esiste una funzione computabile totale $(m+1)$-aria $s_n^m(e,\boldsymbol{x})$ tale che*

$$\phi_e^{(m+n)}(\boldsymbol{x},\boldsymbol{y}) \simeq \phi_{s_n^m(e,\boldsymbol{x})}^{(n)}(\boldsymbol{y})$$

Dimostrazione. La dimostrazione è una generalizzazione della precedente. Per ciascun valore prefissato della coppia $(e,\boldsymbol{x})$, $s_n^m(e,\boldsymbol{x})$ sarà l'indice del programma Q il quale, data la configurazione iniziale



computa $\phi_e^{(m+n)}(\boldsymbol{x},\boldsymbol{y})$. Sia P_e il programma che computa ϕ_e. Allora per comporre il programma Q dobbiamo scrivere P_e preceduto dalle istruzioni che ci consentono di trasformare la configurazione precedente nella seguente



Il programma Q è allora quello rappresentato in figura 4.4; $s_n^m(e, \boldsymbol{x})$ risulta essere l'indice di Gödel di Q, che si può ricavare in modo effettivo mediante Gödelizzazione dello stesso. ▫

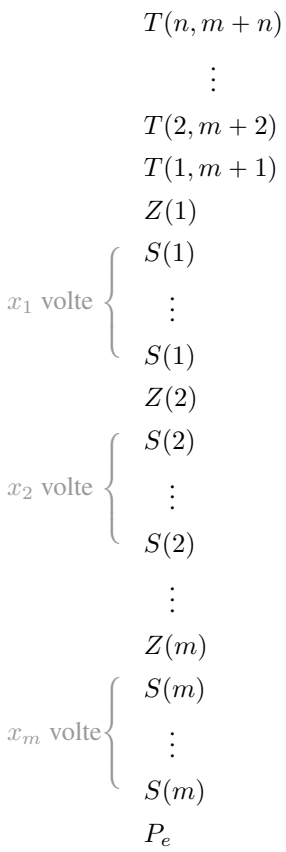


Figura 4.4: Struttura del programma Q il cui indice di Gödel rappresenta la funzione $s_n^m(e, \boldsymbol{x})$.

4.7 Funzione universale

Lo sviluppo delle tecniche di calcolo passò attraverso le fasi descritte nel paragrafo 1.2. In un primo periodo si costruirono delle macchine la cui struttura fisica era basata sul problema da risolvere, p.es. la Pascalina o le Ruote di Leibniz. In una seconda fase, grazie al contributo di Babbage, si passò a un livello superiore nel quale il problema viene risolto modificando il programma, senza dunque ri-definire la topologia costruttiva della macchina. Questa è la modalità classica d'impiego dei computer, nella quale la computazione eseguita dal programma ad hoc, costruito sulla base di un certo algoritmo risolutivo del problema assegnato, porta alla soluzione del problema. Se vogliamo risolvere un problema diverso, dobbiamo costruire un programma diverso.
C'è tuttavia un terzo livello, concettualmente inatteso, che corrisponde alla possibilità di costruire un programma adatto a risolvere qualunque problema; in un certo senso un tale programma, che chiameremo *programma universale* (o *macchina universale*), incorpora tutti i possibili programmi. L'esistenza del programma universale costituisce, assieme alla Gödelizzazione e al teorema s–m–n, la terna di pilastri sulla quale si basano tutti i teoremi rilevanti della Teoria della Computazione, in particolare i teoremi di indecidibilità e della ricorsione; inoltre, esso giustifica teoricamente l'esistenza degli *interpreti*, cioè di programmi in grado di eseguire un programma in un linguaggio di alto livello senza la previa compilazione dello stesso (codice oggetto); cioè corrisponde all'esecuzione

delle istruzioni nel linguaggio usato, traducendole di volta in volta in istruzioni in linguaggio macchina. Vedremo tra poco come sia possibile costruire un programma universale.
Consideriamo la seguente funzione

$$\Psi(x, y) = \phi_x(y)$$

che ha come variabili l'indice di Gödel x di una certa funzione e il valore iniziale y sul quale viene caricata la computazione della stessa funzione. E' evidente che, in un certo senso, la funzione Ψ incorpora tutte le funzioni unarie computabili $\phi_0, \phi_1, \phi_2, \ldots$, poiché per un particolare x la funzione rappresenta ϕ_x; dunque, variando x su tutti i valori possibili la funzione $\Psi(x, y)$ descrive una qualunque funzione computabile. Se estendiamo il ragionamento al caso vettoriale otteniamo la seguente

Definizione 4.3. *Per ciascuna funzione n-aria computabile $\phi_e^{(n)}(\boldsymbol{x})$ si definisce* funzione universale *la seguente funzione $(n+1)$-aria*

$$\Psi_u^{(n)}(e, \boldsymbol{x}) \simeq \phi_e^{(n)}(\boldsymbol{x}) \tag{4.16}$$

Volendo dare una schematizzazione del funzionamento della macchina universale siamo di fronte alla situazione descritta in figura 4.5.

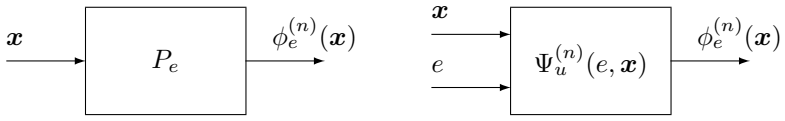


Figura 4.5: Schematizzazione del funzionamento della macchina universale

A priori sembra poco verosimile che una tale funzione possa essere computabile. Tuttavia, ragionando in modo più approfondito, ci si rende subito conto che per poter inglobare tutte le funzioni non è necessario incamerare tutti i programmi, la qual cosa sarebbe palesemente impossibile; è solo sufficiente poter *emulare*, con la funzione universale, la computazione di un qualunque programma. Se si riuscisse a dimostrare la computabilità di Ψ, il *programma universale* P_u sarebbe in grado di emulare il comportamento di qualunque altro programma P_x. Vale allora il seguente

Teorema 4.11. *Per ciascun n la funzione universale $\Psi_u^{(n)}$ è computabile.*

Dimostrazione. Fissato n ed e possiamo stabilire la seguente procedura per calcolare $\Psi_u^{(n)}(e, \boldsymbol{x})$:

1. si decodifica e tramite la funzione γ^{-1}, descritta nella (4.13), ottenendo il programma P_e;
2. si emula la computazione del programma $P_e(\boldsymbol{x})$ passo passo, scrivendo a ogni passo la configurazione corrente del registro e il pedice della prossima istruzione che deve essere eseguita;
3. se e quando la computazione va a convergenza, il valore di $\Psi_u^{(n)}(e, \boldsymbol{x})$ è il contenuto finale del registro R_1, il primo registro della memoria.

Evocando la tesi di Church-Turing si potrebbe dedurre immediatamente la computabilità della funzione; preferiamo tuttavia impostare una dimostrazione più formale data l'importanza del risultato e il fatto che useremo la traccia di tale dimostrazione anche per ottenere altri risultati successivamente.

Per poter dimostrare la computabilità di $\Psi_u^{(n)}(e, \boldsymbol{x})$ bisogna riuscire a codificare in termini numerico-funzionali le varie fasi della computazione. L'idea generale della dimostrazione è la seguente. Poiché ogni passo

della computazione $P_e(\boldsymbol{x})$ viene descritto: (i) dalla configurazione corrente del registro e (ii) dal pedice della prossima istruzione che deve essere eseguita, si vogliono introdurre due funzioni, c_n e j_n, i cui valori rappresentano rispettivamente la codifica della configurazione corrente del nastro e il pedice della prossima istruzione. Poiché il valore di ciascuna delle due funzioni è un numero naturale, possiamo usare la funzione π introdotta nella (4.2) per creare una corrispondenza biunivoca coi numeri naturali, in modo che la computazione, istante per istante, sia rappresentata da un unico numero naturale $\sigma_n = \pi(c_n, j_n)$. Per stabilire la computabilità di Ψ_u è allora sufficiente dimostrare la computabilità di σ_n.
Costruiamo ora le funzioni c_n e j_n, osservando innanzitutto che, istante per istante, il loro valore dipende: (i) da e, indice di Gödel del programma, (ii) dalla configurazione iniziale $\boldsymbol{x}$ e (iii) dal passo t cui si è giunti fino a quel momento. Le due funzioni sono allora del tipo $c_n(e, \boldsymbol{x}, t)$ e $j_n(e, \boldsymbol{x}, t)$. Per codificare la configurazione corrente delle memorie $R_1, R_2, R_3, \ldots$ della macchina RAM mediante un unico numero naturale, potremmo usare la funzione τ descritta nella (4.8) nell'ambito della Gödelizzazione. Una rappresentazione più compatta deriva dalla possibilità di esprimere, in modo univoco, un qualunque numero naturale come scomposizione in fattori primi (teorema fondamentale dell'aritmetica). Per definire la funzione $c_n(e, \boldsymbol{x}, t)$ possiamo allora usare i valori $r_1, r_2, r_3, \ldots$, contenuti nelle celle $R_1, R_2, R_3, \ldots$ come esponenti dei numeri primi $p_1 = 2, p_2 = 3, p_3 = 5, \ldots$ il cui prodotto rappresenta il valore assunto dalla $c_n(e, \boldsymbol{x}, t)$; ricordiamo che in ogni istante della computazione $P_e(\boldsymbol{x})$ solo un numero finito di celle di memoria contiene un numero $\neq 0$. La funzione che codifica la configurazione è allora la seguente

$$c_n(e, \boldsymbol{x}, t) = 2^{r_1} \cdot 3^{r_2} \cdot 5^{r_3} \cdot \ldots = \prod_{i \geqslant 1} p_i^{r_i} = \begin{cases} \text{la configurazione della computazione } P_e(\boldsymbol{x}) \text{ dopo } t \text{ passi} \\ \text{la configurazione finale, se } P_e(\boldsymbol{x}) \downarrow \text{ in } t \text{ o meno passi} \end{cases} \tag{4.17}$$

mentre per la funzione $j_n(e, \boldsymbol{x}, t)$ possiamo scrivere

$$j_n(e, \boldsymbol{x}, t) = \begin{cases} \text{il pedice della prossima istruzione dopo } t \text{ passi della computazione } P_e(\boldsymbol{x}) \\ 0 \text{ se } P_e(\boldsymbol{x}) \downarrow \text{ in } t \text{ o meno passi} \end{cases} \tag{4.18}$$

A questo punto, disponendo di $c_n(e, \boldsymbol{x}, t)$ e $j_n(e, \boldsymbol{x}, t)$, possiamo rappresentare lo *stato* corrente della computazione $P_e(\boldsymbol{x})$ con un solo numero intero mediante la funzione

$$\sigma_n(e, \boldsymbol{x}, t) = \pi(c_n(e, \boldsymbol{x}, t), j_n(e, \boldsymbol{x}, t)) = \text{lo stato della computazione } P_e(\boldsymbol{x}) \text{ dopo } t \text{ passi} \tag{4.19}$$

Ricordiamo che, assegnata $\sigma_n(e, \boldsymbol{x}, t)$, possiamo risalire alla configurazione di memoria e al pedice della prossima istruzione mediante le funzioni π_1 e π_2 introdotte nella (4.3)

$$c_n(e, \boldsymbol{x}, t) = \pi_1(\sigma_n(e, \boldsymbol{x}, t)) \qquad j_n(e, \boldsymbol{x}, t) = \pi_2(\sigma_n(e, \boldsymbol{x}, t)) \tag{4.20}$$

Lo scopo è ora quello di dimostrare che $\sigma_n(e, \boldsymbol{x}, t)$, e quindi $c_n(e, \boldsymbol{x}, t)$ e $j_n(e, \boldsymbol{x}, t)$, sono computabili. Iniziamo con l'osservare che, se la computazione $P_e(\boldsymbol{x})$ si ferma, lo fa dopo

$$t^* = \mu t(j_n(e, \boldsymbol{x}, t) = 0)$$

La configurazione finale associata è

$$c_n(e, \boldsymbol{x}, t^*) = c_n(e, \boldsymbol{x}, \mu t(j_n(e, \boldsymbol{x}, t) = 0))$$

Il valore della funzione $\Psi_u^{(n)}(e, \boldsymbol{x})$ si ottiene andando a leggere il contenuto r_1 della prima cella della memoria R_1, che corrisponde al primo esponente della scomposizione di $c_n(e, \boldsymbol{x}, t^*)$ in fattori primi; mediante la funzione (s) del teorema 3.3 si ricava allora

$$\Psi_u^{(n)}(e, \boldsymbol{x}) = (c_n(e, \boldsymbol{x}, \mu t(j_n(e, \boldsymbol{x}, t) = 0)))_1 = (\pi_1(\sigma_n(e, \boldsymbol{x}, \mu t(j_n(e, \boldsymbol{x}, t) = 0))))_1 \tag{4.21}$$

La computabilità di $\Psi_u^{(n)}(e, \boldsymbol{x})$ deriva dalla computabilità delle funzioni $(x)_y$ e π_1 e dalla computabilità della sostituzione e della ricorsione. Infatti la $\sigma_n(e, \boldsymbol{x}, t+1)$ si può ricavare per ricorsione nel modo seguente: si

decodifica $\sigma_n(e, \boldsymbol{x}, t)$ usando le relazioni (4.20) e si ottengono $c_n(e, \boldsymbol{x}, t)$ e $j_n(e, \boldsymbol{x}, t)$; se $j_n(e, \boldsymbol{x}, t) = 0$ allora $\sigma_n(e, \boldsymbol{x}, t+1) = \sigma_n(e, \boldsymbol{x}, t)$. Altrimenti scriviamo la configurazione dei registri al passo t, ricavata da $c_n(e, \boldsymbol{x}, t)$ mediante la funzione $(x)_y$

$(c_n)_1$	$(c_n)_2$	$\ldots$	$(c_n)_n$	0	0

Dalla decodifica di e ricaviamo l'istruzione corrente I_j; applicandola alla configurazione $c_n(e, \boldsymbol{x}, t)$ sopra rappresentata otteniamo la nuova configurazione $c_n(e, \boldsymbol{x}, t+1)$. Se $j_n(e, \boldsymbol{x}, t+1)$ è la nuova istruzione da eseguire si ricava

$$\sigma_n(e, \boldsymbol{x}, t+1) = \pi(c_n(e, \boldsymbol{x}, t+1), j_n(e, \boldsymbol{x}, t+1))$$
$$\sigma_n(e, \boldsymbol{x}, 0) = \pi(c_n(e, \boldsymbol{x}, 0), 1) = \pi(2^{x_1} \cdot 3^{x_2} \cdot 5^{x_3} \cdot \ldots \cdot p_n^{x_n}, 1)$$

che è dunque definita per ricorsione a partire dalla configurazione iniziale $x_1, x_2, \ldots, x_n$, per la quale bisogna eseguire l'istruzione I_1. □

Il fatto che la $\Psi_u^{(n)}(e, \boldsymbol{x})$ sia computabile significa che esiste un programma universale P_u che la calcola. Costruire effettivamente tale programma nel caso del modello RAM non è di immediata realizzazione. Si può invece impostare facilmente la struttura della memoria durante il generico passo t della computazione. Si parte dalla configurazione iniziale

e	$c_n(e, \boldsymbol{x}, 0)$	0	0	$\ldots$	0	$\ldots$

che contiene, oltre all'indice di Gödel e, anche la codifica $c_n(e, \boldsymbol{x}, 0)$ della configurazione iniziale $\boldsymbol{x}$ della macchina da emulare, secondo quanto descritto dalla (6.1). Da tale codifica si deduce anche l'arietà n della funzione. Per il generico passo t possiamo riservare la cella R_1 alla memorizzazione del valore assunto da $\Psi_u^{(n)}(e, \boldsymbol{x})$ alla fine della computazione (se c'è convergenza); serve poi dedicare una cella rispettivamente a e, $c_n(e, \boldsymbol{x}, 0)$, n e per i valori correnti di $c_n(e, \boldsymbol{x}, t)$ e $j_n(e, \boldsymbol{x}, t)$ al passo t. Una volta individuata l'istruzione I_j da eseguire, la stessa può essere rappresentata dal proprio indice $\beta(I_j)$ e da un numero dell'insieme $0, 1, 2, 3$ che rappresenta il tipo di istruzione, $Z(n), S(n), T(m, n), C(m, n, q)$, secondo quanto visto nella (4.12). Se I_j ha effetto sulla cella R_k, ecco che $(c_n(e,\boldsymbol{x},t))_k$ ne rappresenta il contenuto.

R_1	e	$c_n(e, \boldsymbol{x}, 0)$	n	$c_n(e, \boldsymbol{x}, t)$	$j_n(e, \boldsymbol{x}, t)$	$\beta(I_j)$	tipo I_j	$(c_n(e,\boldsymbol{x},t))_k$

Osservazione 4.6. La costruzione effettiva della macchina universale risulta più agevole quando si usi il modello di Turing. Ne riparleremo al momento opportuno.

Dalla dimostrazione del teorema 4.11 sulla computabilità della funzione universale si ricava immediatamente il seguente corollario

Corollario 4.1. (**Computazione in tempo finito**) *Per ogni $n \geqslant 1$ i seguenti predicati sono decidibili*

$$(a) \quad {}'P_e(\boldsymbol{x})\downarrow \text{ in } t \text{ o meno passi}'$$
$$(b) \quad {}'P_e(\boldsymbol{x})\downarrow y \text{ in } t \text{ o meno passi}'$$

Dimostrazione.

$$(a) \quad {}'P_e(\boldsymbol{x})\downarrow \text{ in } t \text{ o meno passi}' = {}'j_n(e, \boldsymbol{x}, t) = 0'$$
$$(b) \quad {}'P_e(\boldsymbol{x})\downarrow y \text{ in } t \text{ o meno passi}' = {}'j_n(e, \boldsymbol{x}, t) = 0 \text{ AND } (c_n(e, \boldsymbol{x}, t))_1 = y'$$

□

La computabilità della funzione universale, coniugata con l'uso del teorema $s-m-n$, porta anche ai seguenti risultati

Teorema 4.12. *Esiste una funzione totale computabile $s(x,y)$ e una $r(x,y)$ tali che*

$$(a) \quad \phi_{s(x,y)} \simeq \phi_x \cdot \phi_y$$

$$(b) \quad \phi_{r(x,y)} \simeq \phi_x \circ \phi_y$$

Dimostrazione. $(a) \quad f(x,y,z) \simeq \phi_x(z) \cdot \phi_y(z) \overset{1}{\simeq} \Psi_u(x,z) \cdot \Psi_u(y,z) \overset{2}{\simeq} \phi_{s(x,y)}(z)$

dove la 1 deriva dalla computabilità della funzione universale e la 2 dal teorema $s-m-n$. In modo analogo si ha

$$(b) \quad g(x,y,z) \simeq \phi_x(\phi_y(z)) \overset{1}{\simeq} \phi_x(\Psi_u(y,z)) \overset{1}{\simeq} \Psi_u(x,\Psi_u(y,z)) \overset{2}{\simeq} \phi_{r(x,y)}(z) \qquad \square$$

Capitolo 5

Indecidibilità, semidecidibilità e parziale ricorsività

Il capitolo 4 ha messo in luce un risultato molto profondo e di natura positiva, cioè la possibiltà di contare tutti i possibili programmi nel linguaggio RAM e di porli in corrispondenza biunivoca con i numeri naturali. Abbiamo inoltre verificato che la biiezione con gli elementi di $\mathbb{N}$ vale anche per l'insieme delle funzioni computabili. Per contro, questi risultati positivi aprono le porte al primo risultato negativo della teoria, cioè l'esistenza di funzioni non computabili, sancita implicitamente dalla circostanza che, al contrario delle funzioni computabili, l'insieme di tutte le funzioni ha la cardinalità di $\mathbb{R}$; il teorema 4.8 individua poi concretamente un esempio di tali funzioni. L'esistenza di funzioni non computabili pone un chiaro limite teorico al metodo procedurale-algoritmico, anche se si può avere la sensazione che tale limite sia di natura talmente astratta da non riguardare concretamente il mondo reale.
In questo capitolo mostreremo, al contrario, che ci sono ricadute importanti anche nel mondo reale di chi opera quotidianamente con le tecnologie informatiche. Questi risultati verranno espressi mediante alcuni importanti teoremi di *indecidibilità*; uno di questi, il teorema di Rice, offre una chiave di lettura molto negativa, nel senso che stabilisce che la totalità dei predicati che riguardano operazioni "interessanti" su funzioni calcolabili sono indecidibili.

5.1 Principali risultati di indecidibilità

Il primo risultato che presentiamo ha come oggetto la valutazione della totalità di una funzione. Le ricadute operative sono enormi. Immaginiamo che in un'azienda che produce *software* venga presentato un nuovo programma P_x che calcola la funzione Φ_x; come noto il problema più rilevante alla presentazione di un nuovo pacchetto *software* è la possibile presenza di bachi che facciano bloccare la computazione in certe condizioni particolari. I bachi sono essenzialmente di due tipi: *i*) risultato non corretto, cioè $P_x(\boldsymbol{y}) \downarrow c \neq b$, con $f(\boldsymbol{y}) = b$ risultato atteso; *ii*) divergenza, cioè $P_x(\boldsymbol{y}) \uparrow$ in corrispondenza di una configurazione d'ingresso per la quale la funzione avrebbe dovuto essere definita. Questo secondo caso è il più odioso dal punto di vista operativo, poiché congela la computazione e blocca il computer. È evidente che i bachi del secondo tipo esistono solo se abbiamo a che fare con funzioni parziali, e quindi poter decidere (in modo algoritmico) se una certa funzione ϕ_x è totale ci darebbe la possibilità di verificare i programmi rispetto a quest'ultima ipotesi. Dal punto di vista matematico ciò corrisponderebbe a verificare se esistono configurazioni d'ingresso che *non* stanno nel dominio della funzione Φ_x calcolata da P_x. La teoria ci dice che purtroppo non è possibile costruire questo ipotetico programma P_T di

verifica, che risponda SÌ o NO a seconda che ϕ_x sia o meno totale. Questo è solo il primo di una lunga serie di risultati negativi, che culmineranno col teorema di Rice.

Teorema 5.1. (**Indecidibilità della totalità**) *$'\phi_x$ è totale$'$ è indecidibile.*

Dimostrazione. Sia

$$f(x) = \begin{cases} 1 & \text{se } \phi_x \text{ è totale} \\ 0 & \text{se } \phi_x \text{ non è totale} \end{cases}$$

la funzione caratteristica del predicato. Dobbiamo dimostrare che essa non è computabile. La dimostrazione procede per assurdo nel seguente modo: costruiamo una funzione $g(x)$ sicuramente non computabile e facciamo vedere che se $f(x)$ fosse computabile lo sarebbe anche $g(x)$. La $g(x)$ che prendiamo è la seguente

$$g(x) = \begin{cases} \phi_x(x) + 1 & \text{se } \phi_x \text{ è totale} \\ 0 & \text{se } \phi_x \text{ non è totale} \end{cases}$$

Osserviamo che $g(x)$ è totale; tuttavia essa non può coincidere con alcuna delle ϕ_x totali, in quanto diversa nella posizione $\phi_x(x)$ (in x ϕ_x vale $\phi_x(x)$, mentre $g(x)$ vale $\phi_x(x) + 1$); d'altra parte non può neanche coincidere con alcuna delle ϕ_x parziali, in quanto $g(x)$ è totale. Dunque $g(x)$ non è computabile. Se dunque $f(x)$ fosse computabile, si potrebbe usare la computabilità della funzione universale e scrivere

$$g(x) = \begin{cases} \Psi_u(x,x)+1 & \text{se } f(x)=1 \\ 0 & \text{se } f(x)=0 \end{cases}$$

Ciò consentirebbe di calcolare la $g(x)$ tramite la $f(x)$. Dunque $f(x)$ non è computabile. □

Osservazione 5.1. Stabilire la totalità della funzione ϕ_x richiederebbe un numero infinito di verifiche, una per ciascun valore y della $\phi_x(y)$. Infatti non si può mai escludere che, anche se solo per un valore esageratamente grande di y, capiti che $\phi_x(y)$ è indefinita. Poichè tale verifica andrebbe fatta in tempo finito, se ne deduce l'impossibilità di decidere il predicato corrispondente. Vedremo nel seguito criteri più flessibili di decidibilità.

Il prossimo teorema è il più famoso tra tutti, poiché si tratta del problema della *fermata della macchina di Turing* o *Halting Problem*, descritto per la prima volta da Turing nel suo articolo del 1936. In questo articolo Turing riduce l'*Halting Problem* alla questione dell'esistenza di un "algoritmo" in grado di risolvere l'*Entscheidungsproblem*.

Teorema 5.2. (**Halting problem**) *$'\phi_x(y)\downarrow'$ è indecidibile.*

Dimostrazione. Inizieremo dimostrando che $'\phi_x(x)\downarrow'$ è indecidibile. Sia

$$f(x) = \begin{cases} 1 & \text{se } \phi_x(x)\downarrow \\ 0 & \text{se } \phi_x(x)\uparrow \end{cases}$$

la relativa funzione caratteristica. Dobbiamo dimostrare che essa non è computabile. Ragioniamo per assurdo e supponiamo che lo sia. Consideriamo la seguente funzione

$$g(x) = \begin{cases} \uparrow & \text{se } \phi_x(x)\downarrow \quad (f(x)=1) \\ 0 & \text{se } \phi_x(x)\uparrow \quad (f(x)=0) \end{cases}$$

Dalla computabilità della f seguirebbe allora la computabilità della g. Se g è computabile essa ha un indice di Gödel, diciamo e; dunque $g = \phi_e$. Se ora calcoliamo la funzione sul proprio indice, cioè $g(e) = \phi_e(e)$, ricaviamo

$$g(e) = \phi_e(e) = \begin{cases} \uparrow & \text{se } \phi_e(e)\downarrow \\ 0 & \text{se } \phi_e(e)\uparrow \end{cases}$$

che è chiaramente un assurdo. Dunque f non è computabile e $'\phi_x(x)\downarrow'$ è indecidibile. Consideriamo ora il predicato generale $'\phi_x(y)\downarrow'$; la sua funzione caratteristica è

$$f^*(x,y) = \begin{cases} 1 & \text{se } \phi_x(y)\downarrow \\ 0 & \text{se } \phi_x(y)\uparrow \end{cases}$$

e si deve dimostrare che non è computabile. Tale risultato deriva direttamente dall'indecidibilità di $'\phi_x(x)\downarrow'$; se infatti $'\phi_x(y)\downarrow'$ fosse decidibile, la sua funzione caratteristica sarebbe computabile, e quindi la si potrebbe calcolare anche per $y = x$; si potrebbe cioè calcolare $f^*(x,x) = f(x)$, che invece non è calcolabile. □

Osservazione 5.2. La parte finale della dimostrazione è un primo esempio della tecnica di *riduzione*, nella quale si fa vedere che un certo problema $M(x)$ è difficile almeno quanto un altro problema di riferimento $R(x)$, per il quale si sa che non ci sono soluzioni; dunque non ce ne sono neanche per $M(x)$. In questo caso abbiamo dimostrato che se esistesse una soluzione per $M(x) =' \phi_x(y)\downarrow'$, allora ce ne sarebbe una anche per $R(x) =' \phi_x(x)\downarrow'$, che invece è indecidibile; dunque $'\phi_x(y)\downarrow'$ è difficile almeno quanto $'\phi_x(x)\downarrow'$. Si dice allora che il problema $R(x) =' \phi_x(x)\downarrow'$ è stato *ridotto* al problema $M(x) =' \phi_x(y)\downarrow'$.

La portata operativa dell'indecidibilità della totalità, epressa dal teorema 5.1 e dell'indecidibilità della fermata della macchina di Turing, relativa al teorema 5.2, è notevole. Se l'indecidibilità di $'\phi_x$ è totale$'$ implica che non si può costruire un programma di verifica per tutti gli elementi di $\mathbb{N}$, il teorema appena dimostrato *esclude* tale possibilità anche per specifici valori $y \in \mathbb{N}$. Si noti che tale risultato non implica che, per certe coppie specifiche di x e y, non si sia in grado di sapere a priori cosa succede; p.es. il programma P_7 di tabella 4.3, costituito dall'unica istruzione $C(1,1,1)$, sappiamo che genera un *loop* qualunque sia il valore y d'ingresso. Quello che il teorema 5.2 esclude è che esista una procedura che possa decidere *qualunque* sia la coppia x, y. Il risultato espresso dal teorema 5.1 è, se vogliamo, ancora più severo: non esiste una procedura generale per verificare la correttezza di un programma, cioè se esso sia libero da *loop*, anche rinunciando al grado di libertà che consente di scegliere le coppie x, y in modo arbitrario.

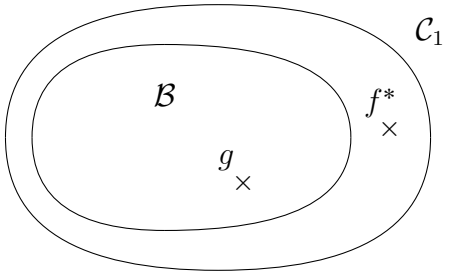
L'ultimo risultato di indecidibilità che dimostreremo è probabilmente quello che presenta le limitazioni più severe. L'idea è che non si possa decidere in merito all'appartenenza di una certa funzione ϕ_x a un sottinsieme proprio non degenere dell'insieme delle funzioni computabili; dal punto di vista operativo ciò corrisponde ad affermare che non si può conoscere in modo algoritmico se una certa funzione possiede o meno determinate specifiche.

Teorema 5.3. (**Teorema di Rice**) *Sia $\mathcal{B} \subseteq \mathcal{C}_1$, con $\mathcal{B} \neq \varnothing, \mathcal{C}_1$. Allora il predicato $'\phi_x \in \mathcal{B}'$ è indecidibile.*

Dimostrazione. L'idea generale della dimostrazione è quella di ridurre il problema $'\phi_x(x)\downarrow'$ al problema $'\phi_x \in \mathcal{B}'$, in modo che se fosse possibile risolvere quest'ultimo si risolverebbe automaticamente anche il primo, che invece sappiamo non essere decidibile. Per attuare questa strategia dobbiamo far uso di due funzioni di supporto, una qualunque g che scegliamo noi e che sta in $\mathcal{B}$ e una speciale f^* che sta fuori. Consideriamo come f^* la funzione non definita dappertutto. Dal corollario 3.3 sull'algebra di decidibilità sappiamo che $'\phi_x \in \mathcal{B}'$ è decidibile se e solo se $'\phi_x \in \mathcal{C}_1 \backslash \mathcal{B}'$ lo è; dunque possiamo assumere, senza perdita di generalità, che la funzione f^* non appartenga a $\mathcal{B}$; se ciò non fosse vero potremmo dimostrare il teorema per $\mathcal{C}_1 \backslash \mathcal{B}$. Scegliamo allora una qualunque $g \in \mathcal{B}$ e consideriamo la seguente funzione

$$f(x,y) = \begin{cases} g(y) & \text{se } \phi_x(x)\downarrow \\ \uparrow & \text{se } \phi_x(x)\uparrow \end{cases}$$

La $f(x,y)$ è chiaramente computabile; dunque per il teorema $s-m-n$ esiste una funzione totale computabile



$k(x)$ tale che $f(x,y) \simeq \phi_{k(x)}(y)$. Si osservi ora che

$$\begin{array}{lcll} \phi_x(x)\downarrow & \Rightarrow & \phi_{k(x)} \simeq g & \Rightarrow \quad \phi_{k(x)} \in \mathcal{B} \\ \phi_x(x)\uparrow & \Rightarrow & \phi_{k(x)} \simeq f^* & \Rightarrow \quad \phi_{k(x)} \notin \mathcal{B} \end{array}$$

Per risolvere il problema $'\phi_x(x)\downarrow'$ sarebbe dunque sufficiente sapere se $\phi_{k(x)}$ è uguale alla funzione g o alla f^*, cioè sapere se $\phi_{k(x)}$ appartiene o meno a $\mathcal{B}$; se dunque $'\phi_x \in \mathcal{B}'$ fosse decidibile, sarebbe decidibile anche $'\phi_x(x)\downarrow'$, che invece non lo è. Abbiamo dunque ridotto il problema $'\phi_x(x)\downarrow'$ al problema $'\phi_x \in \mathcal{B}'$ usando la funzione computabile $k(x)$; dunque $'\phi_x \in \mathcal{B}'$ non è decidibile. ▫

Dal teorema di Rice segue immediatamente il seguente corollario

Corollario 5.1.

$$\begin{array}{lll} (a) & '\phi_x = \mathbf{0}' & \textit{è indecidibile} \\ (b) & '\phi_x = \phi'_y & \textit{è indecidibile} \end{array}$$

Dimostrazione. Per la (a) basta scegliere $\mathcal{B} = \{\mathbf{0}\}$. Per la (b) si opera per riduzione; se $'\phi_x = \phi'_y$ fosse decidibile, lo sarebbe anche $'\phi_x = \phi'_c$, con c indice tale che $\phi_c = \mathbf{0}$, e dunque sarebbe decidibile $'\phi_x = \mathbf{0}'$, che non lo è. ▫

I risultati negativi associati al teorema di Rice si possono esprimere in modo sintetico affermando che non è possibile decidere se una certa funzione possieda o meno una certa proprietà, a meno che questa non sia banale, cioè riconducibile ai casi $\mathcal{B} = \varnothing$ oppure $\mathcal{B} = \mathcal{C}_1$. Per fare un esempio chiarificatore supponiamo che si voglia verificare se un certo programma P_e possieda o meno una certa specifica s, incarnata dalla funzione f_s; se $\mathcal{B} = \{x \in \mathbb{N} : \phi_x = f_s\}$ è l'insieme degli indici di Gödel associati alla specifica, si ha che $'\phi_e \in \mathcal{B}'$ è indecidibile, e dunque non possiamo sapere se il nostro indice e è uno di quelli per il quale vale la proprietà.

5.2 Il teorema di ricorsione

Dopo tanti risultati negativi, che sanciscono l'impossibilità di risolvere importanti problemi del mondo reale, presentiamo ora un risultato positivo. Si tratta del *teorema di ricorsione* di Kleene, detto anche *teorema del punto fisso*. Oltre a essere uno strumento teorico di grande importanza per la teoria avanzata della computazione, costituisce una giustificazione teorica del processo di ricorsione introdotto nella sezione 3.3. Una funzione $h(\boldsymbol{x}, y)$ viene definita per ricorsione sulla base dell'equazione 3.3, che qua riportiamo per comodità del lettore

$$\begin{array}{ll} (i) & h(\boldsymbol{x}, 0) \simeq f(\boldsymbol{x}) \\ (ii) & h(\boldsymbol{x}, y+1) \simeq g(\boldsymbol{x}, y, h(\boldsymbol{x}, y)) \end{array} \tag{5.1}$$

Poiché la $h(\boldsymbol{x}, y{+}1)$ si costruisce a partire dalla $h(\boldsymbol{x}, y)$ al passo precedente, le due funzioni sono concettualmente due oggetti distinti; se scriviamo $h(\boldsymbol{x}, y) \simeq \phi_e(\boldsymbol{x}, y)$, la $h(\boldsymbol{x}, y{+}1)$ può essere interpretata come una funzione il cui indice di Gödel dipende, tramite una certa funzione t, dall'indice e relativo alla $h(\boldsymbol{x}, y)$; in tal modo possiamo scrivere

$$(i) \quad \phi_{t(e)}(\boldsymbol{x}, 0) \simeq f(\boldsymbol{x})$$
$$(ii) \quad \phi_{t(e)}(\boldsymbol{x}, y+1) \simeq g(\boldsymbol{x}, y, \phi_e(\boldsymbol{x}, y))$$

Se ora accade che si realizzi la condizione $\phi_e = \phi_{t(e)}$, tale funzione sarà la soluzione dell'equazione di ricorsione, consentendo di giustificare in modo teorico la sua esistenza e consistenza. Vale allora il seguente

Teorema 5.4. (**Teorema di ricorsione**) *Sia assegnata una funzione t totale e computabile; allora esiste e tale che*

$$\phi_e = \phi_{t(e)}$$

Dimostrazione. Sia u intero qualunque; costruiamo una funzione $\Theta(u, \boldsymbol{x})$ nel modo seguente: calcoliamo $\phi_u(u)$ e se $\phi_u(u)\downarrow$ allora lo usiamo come indice di Gödel

$$\Theta(u, \boldsymbol{x}) = \begin{cases} \phi_{\phi_u(u)}(\boldsymbol{x}) & \text{se } \phi_u(u)\downarrow \\ \uparrow & \text{se } \phi_u(u)\uparrow \end{cases}$$

Per il teorema $s{-}m{-}n$ esiste $g(u)$ totale computabile tale che

$$\phi_{g(u)}(\boldsymbol{x}) = \begin{cases} \phi_{\phi_u(u)}(\boldsymbol{x}) & \text{se } \phi_u(u)\downarrow \\ \uparrow & \text{se } \phi_u(u)\uparrow \end{cases} \tag{5.2}$$

Sia ora t una qualunque funzione totale computabile; usando la g costruiamo la sua composta $t \circ g$; poichè la composizione di due funzioni totali computabili è totale computabile, esiste un indice di Gödel v tale che

$$\phi_v \simeq t \circ g$$

Calcolando ora la 5.2 in v si ottiene

$$\phi_{g(v)} \simeq \phi_{\phi_v(v)} \simeq \phi_{t \circ g(v)}$$

e dunque $e = g(v)$ è il punto fisso. □

Dal teorema del punto fisso seguono immediatamente i seguenti corollari

Corollario 5.2. *Per ogni funzione $f(x, \boldsymbol{y})$ computabile esiste e tale che*

$$f(e, \boldsymbol{y}) \simeq \phi_e(\boldsymbol{y})$$

Dimostrazione. Per il teorema $s{-}m{-}n$ esiste $t(e)$ tale che

$$f(e, \boldsymbol{y}) \simeq \phi_{t(e)}(\boldsymbol{y}) \simeq \phi_e(\boldsymbol{y})$$

dove l'ultima uguaglianza deriva dal teorema di ricorsione. □

Corollario 5.3. *Esiste e tale che*

$$e = \phi_e(\boldsymbol{y})$$

Dimostrazione. Basta prendere come $f(e, \boldsymbol{y})$ la funzione costante e, cioè $f(e, \boldsymbol{y}) = e$, e usare il corollario precedente. □

Il corollario 5.3 ha un'interpretazione molto suggestiva; la funzione $\phi_e(\boldsymbol{x}) = e$ assume un valore costante per tutti i valori di $\boldsymbol{x}$ in ingresso, e tale valore coincide col proprio indice di Gödel; in altre parole la funzione ϕ_e, in un certo senso, *genera sé stessa*. Nell'ambito della programmazione ciò corrisponde a un programma che si autoreplica, cioè che genera il proprio listato. E' lecito interrogarsi sulla possibilità di costruire concretamente tali programmi, noti col nome di *Quine*, e il corollario 5.3 ci dà una risposta positiva in tal senso. Il programma sotto rappresentato è un Quine per il Pascal, ma si possono ottenere per tutti i linguaggi; in esso vengono definite varie costanti stringa, la composizione delle quali ricostruisce il listato del programma. L'esecuzione corrisponde a un'unica istruzione di scrittura, che costruisce nel modo corretto il listato del programma.

```
program s;
const p='program s;const p=';
a='a';
aa=''';';
aaa='a=''';
aaaa='''';
aaaaa='begin write(p,aaaa,p,aa,aaa,a,aa,a,aaa,aaaa,aa,aa,
a,a,aaa,aaa,aaaa,aa,a,a,a,aaa,aaaa,aaaa,aa,a,a,a,a,aaa,
aaaaa,aa,aaaaa)end.';
begin
write(p,aaaa,p,aa,aaa,a,aa,a,aaa,aaaa,aa,aa,a,a,
aaa,aaa,aaaa,aa,a,a,a,aaa,aaaa,aaaa,aa,a,a,a,a,
aaa,aaaaa,aa,aaaaa)
end.
```

5.3 Parziale decidibilità

Abbiamo visto dal teorema 5.2 che il predicato $'\phi_x(x)\downarrow'$ non è decidibile, poiché la sua funzione caratteristica non è computabile. L'indecidibilità è legata alla necessità di decidere in tempo finito se esiste un ciclo infinito per certi valori della variabile d'ingresso. Se rimuoviamo questo vincolo, otteniamo la seguente funzione caratteristica

$$f(x) = \begin{cases} 1 & \text{se } \phi_x(x)\downarrow \\ \uparrow & \text{se } \phi_x(x)\uparrow \end{cases} \tag{5.3}$$

che è invece computabile. In questo caso si ottiene una risposta nel solo caso in cui ci sia convergenza, mentre se c'è divergenza la procedura di verifica cicla all'infinito. Una procedura di questo genere si chiama *procedura di decisione parziale* e il predicato associato si dice *predicato parzialmente decidibile* o *semidecidibile*.

Definizione 5.1. *Un predicato $'M(x)'$ si dice* parzialmente decidibile *se la funzione*

$$f(x) = \begin{cases} 1 & \textit{se } 'M(x)' \textit{ vale} \\ \uparrow & \textit{se } 'M(x)' \textit{ non vale} \end{cases}$$

è computabile. Questa funzione è chiamata funzione caratteristica parziale. *Se $'M(x)'$ è parzialmente decidibile ogni algoritmo per computare f si chiama* procedura di decisione parziale per $'M(x)'$.

Vediamo alcuni esempi.

Esempio 5.1.

1. Il predicato $'\phi_x(x)\downarrow'$ relativo al problema 5.2 della fermata della macchina di Turing è parzialmente decidibile; come già affermato, la funzione semicaratteristica (5.3) è infatti computabile.

2. Qualunque predicato decidibile è semidecidibile: basta costruire una procedura che generi un ciclo infinito quando il valore della funzione caratteristica è 0.

3. Il predicato $'\phi_x(x)\uparrow'$ *non* è neanche semidecidibile. Infatti la sua funzione caratteristica

$$f(x) = \begin{cases} 1 & \text{se } \phi_x(x)\uparrow \\ \uparrow & \text{se } \phi_x(x)\downarrow \end{cases}$$

ha un dominio che differisce dal dominio di qualunque funzione computabile unaria, in quanto $f(x)$ è definita per i valori di x per i quali la corrispondente $\phi_x(x)$ non è definita ed è invece indefinita quando la corrispondente $\phi_x(x)$ è definita; per comprendere la situazione può aiutare la figura sotto

	0	1	2	$\dots$	n	$\dots$
ϕ_0	$\phi_0(0)\downarrow$	$\phi_0(1)$	$\phi_0(2)$	$\cdots$	$\phi_0(n)$	$\cdots$
ϕ_1	$\phi_1(0)$	$\phi_1(1)\uparrow$	$\phi_1(2)$	$\cdots$	$\phi_1(n)$	$\cdots$
ϕ_2	$\phi_2(0)$	$\phi_2(1)$	$\phi_2(2)\downarrow$	$\cdots$	$\phi_2(n)$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\dots$
ϕ_n	$\phi_n(0)$	$\phi_n(1)$	$\phi_n(2)$	$\cdots$	$\phi_n(n)\uparrow$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\dots$
f	$\uparrow$	1	$\uparrow$	$\cdots$	1	$\cdots$

Dunque $f(x)$ non è computabile.

4. Sia $'\phi_x$ è totale$'$ che $'\phi_x$ non è totale$'$ *non* sono semidecidibili. Si veda l'esercizio $6.14.7$ del Cutland [5].

Con riferimento al legame che esiste fra totale e parziale decidibilità si può dedurre il seguente

Teorema 5.5. *Un predicato $'M(x)'$ è decidibile se e solo se sia $'M(x)'$ che $'$NOT $M(x)'$ sono parzialmente decidibili.*

Dimostrazione. Se $'M(x)'$ è decidibile, per il corollario 3.3 lo è anche $'$NOT $M(x)'$ e dunque sono entrambi semidecidibili per il punto 2 dell'esempio 5.1. Se viceversa $'M(x)'$ e $'$NOT $M(x)'$ sono parzialmente decidibili, allora esiste un programma F che si ferma quando $'M(x)'$ è vero e un programma G che si ferma quando $'$NOT $M(x)'$ è vero, cioè

$$F(x)\downarrow \iff \text{se } 'M(x)' \text{ vale}$$
$$G(x)\downarrow \iff \text{se } '\text{NOT } M(x)' \text{ vale}$$

Dunque per ogni x o si ferma la computazione $F(x)$ oppure si ferma la computazione $G(x)$, ma non entrambe. Un algoritmo per decidere $'M(x)'$ è allora il seguente: fissato x facciamo partire le due computazioni $F(x)$ e $G(x)$ a passi alterni: se si ferma prima $F(x)$ allora vale $'M(x)'$; se si ferma prima $G(x)$ allora vale $'$NOT $M(x)'$. □

5.4 Insiemi ricorsivi e ricorsivamente enumerabili

Esiste una stretta connessione tra predicati unari sui numeri naturali e sottinsiemi di $\mathbb{N}$; in corrispondenza di ogni predicato $'M(x)'$ abbiamo infatti l'insieme $A = \{x :' M(x)' \text{ vale}\}$ chiamato *estensione* di M (ovviamente

potrebbe essere che $A = \varnothing$). Viceversa, a un certo insieme $A \subseteq \mathbb{N}$ si può far corrispondere il predicato $'x \in A'$. Sulla base di una tale corrispondenza chiamiamo *ricorsivi* gli insiemi che sono associati a predicati decidibili. In altre parole un insieme è ricorsivo se si può decidere sull'appartenenza di un suo elemento.

Definizione 5.2. *Sia A un sottinsieme di $\mathbb{N}$. La sua funzione caratteristica è definita nel modo seguente*

$$C_A(x) = \begin{cases} 1 & \textit{se } x \in A \\ 0 & \textit{se } x \notin A \end{cases}$$

Allora A è detto ricorsivo *se la C_A è computabile o, equivalentemente, se $'x \in A'$ è decidibile.*

Esempio 5.2. I seguenti insiemi sono ricorsivi

1. $\mathbb{N}$.
2. $\mathbb{P}$, l'insieme dei numeri pari.
3. Ogni insieme finito è ricorsivo.
4. L'insieme dei numeri primi.

Esempio 5.3. I seguenti insiemi non sono ricorsivi

1. $\{x : \phi_x$ è totale$\}$ (teorema 5.1)
2. $\{x : \phi_x(x)\downarrow\}$ (teorema 5.2)
3. $\{x : \phi_x = \mathbf{0}\}$ (corollario 5.1)

Anche in questo caso si può allentare il vincolo di decidibilità accettando una decidibilità solo parziale; in tal caso i predicati saranno semidecidibili. Vale allora la seguente

Definizione 5.3. *Sia A un sottinsieme di $\mathbb{N}$. Si dice che A è* ricorsivamente enumerabile *(r.e.) se la seguente funzione* semicaratteristica

$$f(x) = \begin{cases} 1 & \textit{se } x \in A \\ \uparrow & \textit{se } x \notin A \end{cases}$$

è computabile o, equivalentemente, se $'x \in A'$ è parzialmente decidibile.

Esempio 5.4. I seguenti insiemi sono ricorsivamente enumerabili:

1. Ogni insieme ricorsivo è ricorsivamente enumerabile (cfr. esempio 5.1-2).
2. $\{x : \phi_x(x)\downarrow\}$ (cfr. esempio 5.1-1).

Esempio 5.5. I seguenti insiemi non sono ricorsivamente enumerabili:

1. $\{x : \phi_x(x)\uparrow\}$ (cfr. esempio 5.1-3).
2. $\{x : \phi_x$ è totale$\}$ (si veda il teorema successivo)
3. $\{x : \phi_x$ non è totale$\}$ (si veda il corollario 2.17 del Cutland [5])

Teorema 5.6. *$\{x : \phi_x$ è totale$\}$ non è ricorsivamente enumerabile.*

Dimostrazione. Sia per assurdo $f(x)$ una funzione che enumera le funzioni totali; ciò significa che

$$\phi_{f(0)}, \phi_{f(1)}, \ldots, \phi_{f(x)}, \ldots$$

è una enumerazione delle funzioni totali computabili. Mediante un ragionamento diagonale possiamo allora costruire una funzione totale computabile g, diversa da qualunque funzione nell'enumerazione, nel modo seguente

$$g(x) = \phi_{f(x)} + 1$$

Poiché $g(x)$ è totale computabile, dovrebbe stare nell'elenco; ma $g \neq \phi_{f(m)}\ \forall m$ e dunque g è diversa da ogni funzione dell'elenco. Da cui la contraddizione. □

Osservazione 5.3. Da un punto di vista operativo un insieme ricorsivamente enumerabile è un insieme $A = \{h(0), h(1), h(2), ...\}$ che può essere enumerato in modo effettivo, cioè nel quale gli elementi possono essere contati tramite una funzione totale computabile h. Un altro modo per esprimere questo concetto è che un insieme ricorsivamente enumerabile è *effettivamente generabile* mediante una procedura algoritmica. La procedura genererà, di tanto in tanto e non necessariamente ad intervalli di tempo regolari, un numero da aggiungere alla lista

$$\begin{aligned} h(0) &= 1^\circ \text{ numero generato dalla procedura} \\ h(1) &= 2^\circ \text{ numero generato dalla procedura} \\ h(2) &= 3^\circ \text{ numero generato dalla procedura} \\ &\vdots \\ h(n) &= (n+1)^\circ \text{ numero generato dalla procedura} \end{aligned}$$

Esempio 5.6. L'insieme degli x tali che esiste una tratta di x cifre '5' consecutive nella scomposizione di π è ricorsivamente enumerabile (cfr. (2.1)). Usando la serie di Hutton 2.2, che genera una alla volta tutte le cifre di π, si controlla quando escono dei '5'. Quando esce una tratta di '5' di lunghezza x la metto nell'insieme.

Capitolo 6

Altri modelli di computazione

Nei capitoli precedenti abbiamo definito e impiegato il modello RAM per costruire il nucleo essenziale della teoria della computazione. Esso è costituito dalla Gödelizzazione dei programmi e delle funzioni computabili, dall'esistenza di funzioni non computabili, dal teorema $s-m-n$, dall'esistenza della funzione universale e dai principali risultati di indecidibilità e di decidibilità parziale. Il modello RAM, che ha il pregio di rappresentare molto da vicino l'architettura effettiva dei computer moderni, è stato usato come modello di supporto per l'elaborazione dei vari teoremi. Sappiamo però, dalla sezione 1.3 (si veda la figura 1.13) che furono molti i modelli di computazione, tra loro equivalenti, presentati all'inizio del '900. Abbiamo anticipato che esistono teoremi di equivalenza tra tutti questi modelli e ciò porta alla conclusione che essi sottendono un'unico insieme di funzioni calcolabili, che corrisponde all'insieme $\mathcal{C}$ delle funzioni calcolabili secondo il modello RAM; sulla base della tesi di Church-Turing esso viene fatto corrispondere all'insieme $\mathfrak{C}$ delle funzioni computabili nel senso intuitivo del termine.

In questo capitolo analizzeremo brevemente alcuni di questi modelli e daremo un'idea di massima su come organizzare la dimostrazione di un teorema di equivalenza. Descriveremo il modello di Gödel-Kleene - delle *funzioni μ-ricorsive* e delle *funzioni parziali ricorsive* - le macchine di Turing e di Post e le macchine a memorie *push-down*. Il modello di Gödel-Kleene, in particolare, è utile dal punto di vista didattico, perché costituisce l'ossatura teorica del modello RAM e come vedremo c'è una forte connessione tra i due. Per quanto riguarda invece la celeberrima *macchina di Turing*, passeremo in rassegna anche le molteplici definizioni che si trovano in letteratura, che variano a seconda del contesto e dell'operatività che si vuol far assumere a tale modello. In ultima battuta analizzeremo brevemente la *macchina di Post*, la quale servirà da supporto alla *macchina con memorie push-down* (o *automa a pila*), che a sua volta rappresenta l'anello di congiunzione fra il mondo della *Turing completezza*, costituito da qualunque apparto formale in grado di simulare il comportamento della macchina di Turing, e il mondo degli *automi*, che si situano a un livello inferiore quanto a potenza computazionale.

6.1 Funzioni μ-ricorsive e funzioni parziali ricorsive

Nella prima fase della storia della computabilità si riteneva che sostituzione e ricorsione fossero sufficienti per poter costruire l'insieme di tutte le funzioni che noi oggi chiamiamo computabili. Questo approccio portò alla definizione delle cosiddette *funzioni primitive ricorsive*, di cui abbiamo già fatto cenno nella sezione 3.3.

Definizione 6.1. *La classe $\mathcal{PR}$ delle* funzioni primitive ricorsive *è la più piccola classe di funzioni (totali) che contiene le funzioni base $\mathbf{0}(x), x+1, U_i^n(\boldsymbol{x})$ ed è chiusa rispetto alle operazioni di sostituzione e ricorsione.*

Tutte le funzioni computabili ottenute nei primi paragrafi della sezione 3.3 del capitolo 3, a partire dalle funzioni base, sono primitive ricorsive, poiché in questo caso non è stata utilizzata la minimazione illimitata. Inoltre, poiché le funzioni base sono totali e la sostituzione e la ricorsione, alimentate con funzioni totali, portano ad altre funzioni totali, tutte le funzioni così generate sono totali e dunque $\mathcal{PR}$ contiene solo funzioni totali. Infine si osservi che $\mathcal{PR}$ è chiuso anche rispetto alle somme e prodotti limitati e alla minimazione limitata. Dunque la classe delle funzioni primitive ricorsive è piuttosto estesa. Esistono tuttavia funzioni totali computabili che non sono primitive ricorsive; un esempio è dato dalla funzione Ackermann introdotta nella sezione 3.3 e definita dalla (3.4). Ciò implica che, sebbene le funzioni primitive ricorsive formino una classe molto estesa, esse non includono tutte le funzioni computabili e quindi non possono essere considerate come una possibile caratterizzazione della nozione informale di computabilità.

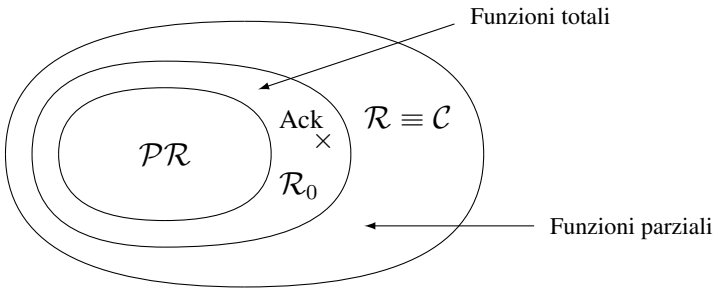
L'introduzione dell'operatore μ, applicato al solo caso delle funzioni totali, porta alla definizione della seguente classe

Definizione 6.2. *La classe $\mathcal{R}_0$ delle* funzioni μ-ricorsive *(o* totali ricorsive*) è la più piccola classe di funzioni totali che contiene le funzioni base $\mathbf{0}(x), x+1, U_i^n(\boldsymbol{x})$ ed è chiusa rispetto alle operazioni di sostituzione, ricorsione e minimazione; l'uso dell'operatore μ di minimazione è consentito solo se da esso vengono generate funzioni totali.*

Si è già discussa la limitazione che deriverebbe dall'uso delle sole funzioni totali nell'ambito della sezione 4.4; in tal modo si avrebbe un modello che non è in grado di catturare la computabilità di alcune semplici funzioni, che sono computabili nel senso comune del termine. E' allora necessario introdurre anche le funzioni parziali. Ciò porta alla seguente

Definizione 6.3. *La classe $\mathcal{R}$ delle* funzioni parziali ricorsive *è la più piccola classe di funzioni parziali che contiene le funzioni base $\mathbf{0}(x), x+1, U_i^n(\boldsymbol{x})$ ed è chiusa rispetto alle operazioni di sostituzione, ricorsione e minimazione; l'uso dell'operatore μ di minimazione è consentito senza alcuna limitazione.*

Come vedremo tra poco questa classe sottende lo stesso insieme di funzioni calcolabili espresse dal modello RAM. Il rapporto che esiste fra $\mathcal{PR}$, $\mathcal{R}_0$ e $\mathcal{R}$ è illustrato nella seguente figura



Teorema 6.1. $\mathcal{R} \equiv \mathcal{C}$

Dimostrazione. Per dimostrare che $\mathcal{R} \equiv \mathcal{C}$ dobbiamo far vedere che valgono le due relazioni $\mathcal{R} \subseteq \mathcal{C}$ e $\mathcal{C} \subseteq \mathcal{R}$. La prima deriva dalla RAM computabilità delle funzioni base, della sostituzione, della ricorsione e della minimazione (teoremi 3.1, 3.1, 3.2 e 3.6).
Per dimostrare che $\mathcal{C} \subseteq \mathcal{R}$ bisogna far vedere che ogni computazione basata sul modello RAM può essere espressa in termini di una funzione parziale ricorsiva. Sia allora $f(\boldsymbol{x})$ la funzione calcolata da un certo programma $P =$

$I_1, I_2, \ldots, I_s$ e consideriamo le seguenti funzioni associate alla computazione di $P(\boldsymbol{x})$

$$c(\boldsymbol{x}, t) = \begin{cases} \text{il contenuto della cella } R_1 \text{ dopo } t \text{ passi della computazione } P(\boldsymbol{x}) \\ \text{il contenuto finale della cella } R_1, \text{ se } P(\boldsymbol{x}) \downarrow \text{ in } t \text{ o meno passi} \end{cases} \tag{6.1}$$

mentre per la funzione $j(\boldsymbol{x}, t)$ possiamo scrivere

$$j(\boldsymbol{x}, t) = \begin{cases} \text{il pedice della prossima istruzione dopo } t \text{ passi della computazione } P(\boldsymbol{x}) \\ 0 \text{ se } P(\boldsymbol{x}) \downarrow \text{ in } t \text{ o meno passi} \end{cases} \tag{6.2}$$

Chiaramente $c(\boldsymbol{x}, t)$ e $j(\boldsymbol{x}, t)$ sono funzioni totali. Se $f(\boldsymbol{x})$ è definita, allora $P(\boldsymbol{x})$ converge dopo

$$t_0 = \mu t(j(\boldsymbol{x}, t) = 0)$$

passi e il valore finale della funzione è

$$f(\boldsymbol{x}) = c(\boldsymbol{x}, t_0) = c(\boldsymbol{x}, \mu t(j(\boldsymbol{x}, t) = 0))$$

che è definita solo se $P(\boldsymbol{x}) \downarrow$. Per mostrare che $f(\boldsymbol{x})$ è parziale ricorsiva, basta mostrare che anche $c(\boldsymbol{x}, t)$ e $j(\boldsymbol{x}, t)$ lo sono; e questo è immediato poiché è sufficiente simulare la computazione fino al passo t; si veda a tal proposito anche la dimostrazione della computabilità della funzione universale 4.11, che usa funzioni simili. □

Osservazione 6.1. Alla luce delle definizioni 6.2 e 6.3 si osservi la stretta relazione che esiste col modello RAM. Le funzioni parziali ricorsive sono basate sulle funzioni base $\mathbf{0}(x), x+1, U_i^n(\boldsymbol{x})$ e sulla sostituzione, ricorsione e minimazione; il modello RAM si appoggia a tale modello, visto che le tre istruzioni aritmetiche - $Z(n), S(n)$ e $T(m, n)$ - rappresentano la proiezione a livello software delle funzioni base. L'istruzione $C(m, n, q)$ è necessaria per poter introdurre le condizioni e la successiva costruzione della sostituzione, della ricorsione e della minimazione a livello di programma RAM completa il parallelismo tra modello RAM e modello delle funzioni parziali ricorsive.

6.2 La macchina di Turing

La definizione di computabilità proposta da Turing nel 1936 si basa sull'attuazione del concetto di algoritmo da parte di un agente umano. Egli sta seduto alla scrivania, ha una pila di fogli di carta (teoricamente infinita) alla sua destra, una (teoricamente infinita) alla sua sinistra e un foglio sulla scrivania; possiede inoltre una matita e una gomma. Su ogni foglio è scritto un carattere che appartiene a un certo alfabeto finito Γ. La computazione vera e propria consiste in una successione - potenzialmente infinita - di azioni elementari che l'agente è in grado di svolgere e che appartiene a uno dei seguenti tre tipi:

1. leggere il simbolo α scritto sul foglio sulla scrivania; cancellare α e scrivere un altro simbolo β; è anche ammesso che sia $\beta = \alpha$;
2. prendere il foglio, posizionarlo sulla pila di destra e prelevare un foglio dalla pila di sinistra, ponendolo sulla scrivania;
3. prendere il foglio, posizionarlo sulla pila di sinistra e prelevare un foglio dalla pila di destra, ponendolo sulla scrivania.

L'azione effettivamente attuata dall'agente umano dipende (i) dal simbolo α letto sul foglio che sta sulla scrivania e (ii) dallo *stato* corrente q nel quale si trova l'agente, che viene preso da un insieme finito Q di stati possibili.

Da un punto di vista operativo l'agente deve disporre delle informazioni necessarie quale si specifica quali delle azioni 1-2-3 egli debba attuare quando, stando nello stato q, egli sta leggendo la lettera α sul foglio; viene inoltre specificato quale sia il nuovo stato q* che si raggiunge dopo aver eseguito una delle operazioni sopra specificate. La computazione inizia specificando il primo foglio da analizzare e lo stato iniziale della macchina; nel seguito procede sulla base delle istruzioni impartite dalla tabella. Con questo approccio alla computazione l'algoritmo risiede implicitamente nella tabella delle istruzioni e l'esito finale della computazione dipende, oltre che dalla tabella, dalle cndizioni iniziali e dal contenuto del nastro. Lo schema appena descritto si chiama *macchina di Turing* (MdT).

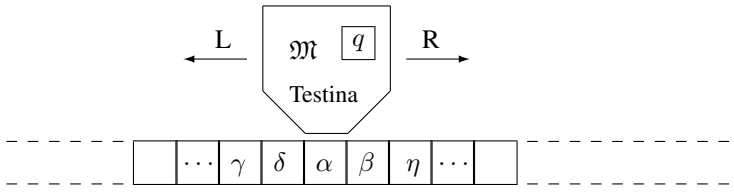


Figura 6.1: Modello della macchina di Turing

In figura 6.1 viene riportato il modello della macchina di Turing $\mathfrak{M}$; esso consta di un nastro infinito a destra e a sinistra, costituito da celle di memoria che possono memorizzare solo lettere che appartengono a un alfabeto finito Γ. Questa è la differenza più rilevante, dal punto di vista concettuale, rispetto al modello RAM, il cui nastro può invece memorizzare un intero arbitrariamente grande. Per ogni computazione convergente su $\mathfrak{M}$ verrà usata solo una parte finita del nastro, anche se in generale non sappiamo in anticipo quale e quanto grande sia. In qualsiasi momento ogni cella del nastro o è vuota o contiene un simbolo α_i dell'alfabeto Γ di $\mathfrak{M}$. Senza perdita di generalità possiamo supporre di rappresentare il vuoto col simbolo $\Delta = \alpha_0$ e di incorporarlo all'interno dell'alfabeto, in modo che sia $\Gamma = \{\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_n\}$.
$\mathfrak{M}$ ha una testina di lettura che serve per (i) leggere un simbolo memorizzato, (ii) cancellare un simbolo letto (iii) scrivere un simbolo sul nastro. $\mathfrak{M}$ è allora in grado di eseguire i seguenti tre tipi di operazioni semplici sul nastro:

1. cancellare il simbolo nella cella che viene scansionata e sostituirlo con un altro simbolo dall'alfabeto Γ;
2. spostare la testina di lettura di una cella a destra di quella scansionata (o, equivalentemente, spostare il nastro di una cella a sinistra);
3. spostare la testina di lettura di una cella a sinistra di quella scansionata (o, in modo equivalente, spostare il nastro di una cella a destra).

La scala dei tempi sui quali si basa il funzionamento della macchina $\mathfrak{M}$ è discreta e in ogni istante di tempo $\mathfrak{M}$ sta in uno specifico stato q tra quelli appartenenti a un insieme finito $Q = \{q_1, q_2, \ldots, q_m\}$. A seguito delle operazioni effettuate, seguendo le istruzioni della tabella, si può avere un cambiamento di stato. L'azione intrapresa da $\mathfrak{M}$ in ogni istante dipende dallo stato corrente di $\mathfrak{M}$ e dal simbolo letto in quel momento dalla testina. Questa dipendenza è descritta da una tabella $\mathfrak{T}$ che consiste in un insieme finito di quadruple, ognuno delle quali assume una delle seguenti forme:

$$q_j\ \alpha_i\ \alpha_k\ q_r$$
$$q_j\ \alpha_i\ R\ \ q_r$$
$$q_j\ \alpha_i\ L\ \ q_r$$

La quadrupla $q_j\ \alpha_i\ \beta\ q_r$ specifica l'azione che deve essere eseguita da $\mathfrak{M}$ quando, stando nello stato q analizza il simbolo α_i:

1. Agire sul nastro nel modo seguente:

(a) se $\beta = \alpha_k$ cancellare α_i e scrivere α_k nella cella scansionata;

(b) se $\beta = R$ spostare la testina di lettura di una cella a destra;

(c) se $\beta = L$ spostare la testina di lettura di una cella a sinistra;

2. Acquisire lo stato q_r

La tabella $\mathfrak{T}$ dev'essere costruita col vincolo che per ogni coppia $q_j\alpha_i$ vi sia al massimo una quadrupla della forma $q_j\alpha_i\beta q_r$, altrimenti ci sarebbe ambiguità su ciò che $\mathfrak{M}$ deve fare successivamente. Una volta specificati lo stato iniziale q_j e la posizione della testina, si può iniziare la computazione leggendo il contenuto α_i della cella e operando seguendo le istruzioni di $\mathfrak{T}$; se essa porta alla nuova configurazione $q^*\alpha^*$ bisogna cercare all'interno di $\mathfrak{T}$ la quartupla che inizia con $q^*\alpha^*$, eseguire l'istruzione e proseguire nella computazione. La computazione termina solo quando $\mathfrak{M}$ sta in uno stato $\overline{q}$, sta scandendo il simbolo $\overline{\alpha}$, ma non esiste alcuna quadrupla della forma $\overline{q}\ \overline{\alpha}\ \beta\ q_r$ in $\mathfrak{T}$; cioè non c'è quadrupla che specifichi cosa fare dopo.

Da un punto di vista astratto la tabella $\mathfrak{T}$ esprime in modo esplicito i valori della seguente *funzione di transizione* δ

$$\delta : Q \times \Gamma \to Q \times (\Gamma \cup \{L, R\}) \tag{6.3}$$

che ci consente di definire in modo astratto una MdT nel modo seguente:

Definizione 6.4. *Una* macchina di Turing *è una quartupla* $(Q,\ \Gamma,\ \delta,\ q_1)$ *dove*

1. $Q = \{q_1, q_2, \ldots q_m\}$ *è l'insieme degli stati;*
2. $\Gamma = \{\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_n\}$ *è l'alfabeto del nastro;*
3. $\delta : Q \times \Gamma \to Q \times (\Gamma \cup \{L, R\})$ *è la funzione di transizione;*
4. $q_1 \in Q$ *è lo stato iniziale.*

Bisogna inoltre fissare una convenzione per la posizione iniziale della testina in lettura.

In generale una MdT può essere usata secondo diverse modalità e finalità di funzionamento; le più rilevanti sono:

1. calcolare funzioni da $\mathbb{N}^n \to \mathbb{N}$ (MdT intesa come *calcolatore*);
2. riconoscere una stringa (MdT intesa come *accettore*);
3. decidere un predicato (MdT intesa come *decisore*);

6.2.1 Macchina di Turing per calcolare una funzione

Esempio 6.1. Sia $Q = \{q_1, q_2\}$, $\Gamma = \{\Delta, 0, 1\}$ dove Δ rappresenta il simbolo di vuoto. La tabella di transizione è

q_1	0	R	q_1
q_1	1	0	q_2
q_2	0	R	q_2
q_2	1	R	q_1

che può essere scritta anche nella seguente forma

	0	1
q_1	R, q_1	$0, q_2$
q_2	R, q_2	R, q_1

che è quella di una *matrice di transizione*. Questa seconda modalità di assegnazione ci consente di esprimere il funzionamento della macchina anche mediante il *grafo finito orientato* di figura 6.2. Il grafo si legge in questo modo: se $\mathfrak{M}$ è nello stato q_1 e sta leggendo 0, dalla matrice di transizione sappiamo che la testina deve spostarsi a

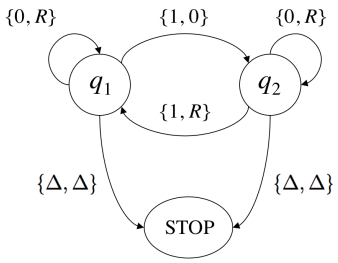


Figura 6.2: Grafo finito orientato associato alla macchina di Turing dell'esempio 6.1

destra (R) e deve rimanere in q_1; dal grafo questa azione è rappresentata dall'arco richiuso su q_1 ed etichettato con $\{0, R\}$; se invece si legge 1 la testina rimane ferma, ma sostituisce 1 con 0 cambiando stato e ponendosi in q_2; sul grafo questa azione è rappresentata dall'arco che va da q_1 a q_2 ed etichettato con $\{1, 0\}$.
Vediamo ora come evolve la computazione supponendo di usare la stringa 111111 come stringa iniziale. La testina è posizionata in corrispondenza del primo 1 a sinistra della stringa iniziale; i simboli Δ stanno a sinistra e a destra della stringa e riempiono tutto il nastro. Nella tabella sotto, la posizione della testina corrisponde a quella del simbolo indicante lo stato (q_1 nella condizione iniziale per $t = 0$). Ogni riga del nastro rappresenta un istante di tempo.

			q_1							
$t = 0$	$\ldots$	Δ	1	1	1	1	1	1	Δ	$\ldots$
			q_2							
$t = 1$	$\ldots$	Δ	0	1	1	1	1	1	Δ	$\ldots$
				q_2						
$t = 2$	$\ldots$	Δ	0	1	1	1	1	1	Δ	$\ldots$
					q_1					
$t = 3$	$\ldots$	Δ	0	1	1	1	1	1	Δ	$\ldots$
					q_2					
$t = 4$	$\ldots$	Δ	0	1	0	1	1	1	Δ	$\ldots$
						q_2				
$t = 5$	$\ldots$	Δ	0	1	0	1	1	1	Δ	$\ldots$
							q_1			
$t = 6$	$\ldots$	Δ	0	1	0	1	1	1	Δ	$\ldots$
							q_2			
$t = 7$	$\ldots$	Δ	0	1	0	1	0	1	Δ	$\ldots$
								q_2		
$t = 8$	$\ldots$	Δ	0	1	0	1	0	1	Δ	$\ldots$
									q_1	
$t = 9$	$\ldots$	Δ	0	1	0	1	0	1	Δ	$\ldots$

Nella prima riga per $t = 0$ la testina è su q_1 e legge 1; si deve cambiare stato (q_2) e scrivere 0 al posto di 1. Per $t = 1$ la macchina è in q_2 e la testina legge 0; la testina deve allora andare a destra (R) e la macchina rimane in q_2; e così via. L'attività svolta dalla macchina è quella di intercalare uno 0 con un 1; in tal modo se gli 1 della stringa iniziale sono $x + 1$, il numero di 1 finali sul nastro corrisponde a $f(x) = \lceil x/2 \rceil$.

Esempio 6.2. Sia $Q = \{q_1, q_2\}$, $\Gamma = \{\Delta, 0, 1\}$ e si voglia calcolare la funzione $f(x, y) = x + y$. Per quanto visto precedentemente, in merito alle convenzioni da usare per il calcolo di una funzione, la stringa iniziale deve contenere $x + 1$ e $y + 1$ simboli 1, separati dal simbolo Δ; il programma deve semplicemente togliere due 1, in modo che alla fine della computazione rimangano $x + y$ simboli 1.

La tabella di transizione è per calcolare questa funzione è fatta nel seguente modo

q_1	1	Δ	q_1
q_1	Δ	R	q_2
q_2	1	Δ	q_3
q_2	Δ	R	q_2

mentre la matrice di transizione associata è

	1	Δ
q_1	Δ, q_1	R, q_2
q_2	Δ, q_3	R, q_2

In figura 6.3 c'è il grafo corrispondente. Vediamo ora come evolve la computazione.

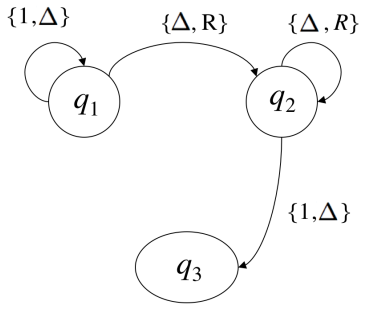


Figura 6.3: Grafo finito orientato associato alla macchina di Turing dell'esempio 6.2

Supponiamo di voler calcolar $2 + 1$; la testina è posizionata in corrispondenza del primo 1 a sinistra della stringa iniziale $111\Delta11$; anche in questo caso a sinistra e a destra della stringa iniziale ci sono dei simboli Δ che riempiono tutto il nastro.

$$
\begin{array}{lcccccccccc}
 & & & & q_1 & & & & & & \\
t=0 & \ldots & \Delta & 1 & 1 & 1 & \Delta & 1 & 1 & \Delta & \ldots \\
 & & & & q_1 & & & & & & \\
t=1 & \ldots & \Delta & \Delta & 1 & 1 & \Delta & 1 & 1 & \Delta & \ldots \\
 & & & & & q_2 & & & & & \\
t=2 & \ldots & \Delta & \Delta & 1 & 1 & \Delta & 1 & 1 & \Delta & \ldots \\
 & & & & & & q_3 & & & & \\
t=3 & \ldots & \Delta & \Delta & \Delta & 1 & \Delta & 1 & 1 & \Delta & \ldots
\end{array}
$$

Nella prima riga per $t = 0$ la testina è su q_1 e legge 1; si scrive Δ e si rimane nello stesso stato. Per $t = 1$ la macchina è sempre in q_1, ma la testina legge Δ; la testina deve allora andare a destra (R) e cambiare stato in q_2. Nell'ultimo passo della computazione la testina si sposta a destra e cambia stato in q_3; a questo punto la macchina si ferma, poiché la combinazione $q_3, 1$ non è presente nella tabella. Sono stati così eliminati i due simboli 1 e il risultato finale corrisponde al numero dei simboli 1 rimasti sul nastro, che sono 3.

6.2.2 Macchina di Turing per riconoscere una stringa

6.2.3 Macchina di Turing per decidere un predicato

6.3 La macchina di Post

6.4 La macchina a memorie *push-down*

Capitolo 7

Automi non deterministici

Un *automa a stati finiti non-deterministico* (NFA) è una quintupla $\mathcal{A}_N = \langle Q, \Sigma, \delta_N, q_0, F_N \rangle$, dove $Q = \{q_0, q_1, ..., q_{m-1}\}$ è l'insieme degli stati, Σ è l'alfabeto d'ingresso, q_0 è lo stato iniziale e $F_N \subseteq Q$ è l'insieme degli stati finali. La funzione di transizione δ_N è ora definita

$$\delta_N : Q \times \Sigma \to \wp(Q)$$

dove $\wp(Q) = \{P_1, P_2, ..., P_{2^m}\}$ è l'insieme delle parti di Q. Si osservi che è ora ammesso $\delta_N(q, a) = \emptyset$ per qualche $q \in Q$ e $a \in \Sigma$. Anche per gli NFA, dalla funzione δ_N si ottiene in modo univoco la funzione

$$\widehat{\delta}_N : Q \times \Sigma^* \to \wp(Q)$$

nel modo seguente

$$\begin{cases} \widehat{\delta}_N(q, \Lambda) = \{q\} \\ \widehat{\delta}_N(q, xa) = \bigcup_{p \in \widehat{\delta}_N(q,x)} \delta_N(p, a) \end{cases} \tag{7.1}$$

Una stringa x è *accettata* da un NFA $\mathcal{A}_N = \langle Q, \Sigma, \delta_N, q_0, F_N \rangle$ se

$$\widehat{\delta}_N(q_0, x) \cap F_N \neq \emptyset$$

Il *linguaggio accettato* da A_N è l'insieme delle stringhe accettate, ovvero:

$$\mathcal{L}(A_N) = \left\{ x \in \Sigma^* : \widehat{\delta}_N(q_0, x) \cap F_N \neq \emptyset \right\}$$

Mentre la rappresentazione mediante tabella degli automi non deterministici è profondamente diversa da quella per i deterministici (in ogni casella si deve inserire ora un insieme di stati), la rappresentazione a grafo rimane pressoché immutata. L'unica differenza è che da un nodo possono uscire più archi (o nessun arco) etichettati con lo stesso simbolo. Dal punto di vista del modello con testina e nastro, il non-determinismo va invece immaginato come la possibilità contemporanea di attuare la computazione per ognuno degli stati raggiunti. Si aprono dunque svariate computazioni virtuali parallele.
Dimostreremo ora che i linguaggi accettati dai DFA e dagli NFA coincidono. Poiché un DFA si può vedere come un NFA in cui $\delta(q, a)$ restituisce sempre insiemi costituiti da un solo stato (detti anche *singoletti*), si ha che ogni linguaggio regolare è un linguaggio accettato da un qualche NFA; dobbiamo allora stabilire il risultato inverso.

Teorema 7.1. *Sia $\mathcal{A}_N = \langle Q, \Sigma, \delta_N, q_0, F_N \rangle$ un automa NFA. Esiste allora un automa DFA $\mathcal{A}_D$ tale che $\mathcal{L}(\mathcal{A}_D) = \mathcal{L}(\mathcal{A}_N)$.*

Dim. Definiamo $\mathcal{A}_D = \langle \wp(Q), \Sigma, \delta_D, \{q_0\}, F_D \rangle$, dove

$$\begin{aligned} \wp(Q) &= \{P_1, P_2, ..., P_{2^m}\} \\ \Sigma &= \text{ alfabeto d'ingresso} \\ \{q_0\} &= \text{ stato di partenza} \\ F_D &= \{P_i | P_i \cap F_N \neq \emptyset\} \\ \delta_D(P,a) &= \bigcup_{p \in P} \delta_N(p,a) \qquad \text{con } P \in \wp(Q) \end{aligned} \tag{7.2}$$

Si deve dimostrare, per induzione sulla lunghezza della stringa di input x, che

$$\widehat{\delta}_N(q_0, x) = \widehat{\delta}_D(\{q_0\}, x) \tag{7.3}$$

Per $|x| = 0$ il risultato è banale, poiché $x = \Lambda$.
Supponiamo che l'ipotesi induttiva valga per tutte le stringhe x tali che $|x| \leqslant m$. Sia xa una stringa di lunghezza $m+1$. Allora, tenendo conto che per un DFA la funzione di transizione viene definita come

$$\begin{cases} \widehat{\delta}_D(q, \Lambda) = q \\ \widehat{\delta}_D(q, xa) = \delta_D\left(\widehat{\delta}_D(q,x), a\right) \end{cases} \tag{7.4}$$

si ha

$$\begin{aligned} \widehat{\delta}_D(\{q_0\}, xa) &= \delta_D\left(\widehat{\delta}_D(\{q_0\}, x), a\right) && \text{per la definizione (7.4) di } \widehat{\delta}_D \\ &= \delta_D\left(\widehat{\delta}_N(q_0, x), a\right) && \text{per l'ipotesi induttiva} \\ &= \textstyle\bigcup_{p \in \widehat{\delta}_N(q_0,x)} \delta_N(p,a) && \text{per la definizione (7.2) di } \delta_D(P,a) \\ &= \widehat{\delta}_N(q_0, xa) && \text{per la definizione (7.1) di } \widehat{\delta}_N(q_0, xa) \end{aligned}$$

E' dunque dimostrata la (7.3). Il teorema segue ora dal fatto che:

$$\begin{aligned} x \in \mathcal{L}(\mathcal{A}_N) \quad \text{sse} \quad & \widehat{\delta}_N(q_0, x) \cap F_N \neq \emptyset && \text{definizione di linguaggio NFA} \\ \text{sse} \quad & \widehat{\delta}_D(\{q_0\}, x) \cap F_N \neq \emptyset && \text{per l'ipotesi induttiva} \\ \text{sse} \quad & \widehat{\delta}_D(\{q_0\}, x) \in F_D && \text{per la definizione di } F_D \\ \text{sse} \quad & x \in \mathcal{L}(\mathcal{A}_D) && \text{per la definizione di linguaggio DFA} \end{aligned}$$

◯

Affrontiamo ora il cosiddetto *lemma di espansione* (o *pumping lemma*) per i linguaggi regolari, che costituisce una condizione necessaria affinché un linguaggio sia regolare. Informalmente si può dire che, in un linguaggio regolare, tutte le parole sufficientemente lunghe possono essere espanse, cioè avere una sezione centrale della parola ripetuta un numero arbitrario di volte, per produrre una nuova parola che appartiene ancora al linguaggio. La dimostrazione del teorema sfrutta il cosiddetto *principio della cassettiera*: se $m+1$ oggetti sono distribuiti in m cassetti, necessariamente ci deve essere almeno un cassetto che contiene più di un oggetto.

Teorema 7.2. *(Pumping lemma) Sia $\mathcal{L} = \mathcal{L}(\mathcal{A})$, dove $\mathcal{A}$ è un automa a m stati. Sia $x \in \mathcal{L}$, con $|x| \geqslant m$. Allora si può scrivere $x = uvw$, dove $v \neq \Lambda$ e $uv^{[i]}w \in \mathcal{L}$ per tutti i valori $i = 0, 1, 2, 3, ...$*

Dim. Poiché x consiste di almeno m simboli, $\mathcal{A}$ deve passare attraverso almeno m transizioni di stato mentre scansiona x; contando lo stato iniziale ciò richiede almeno $m+1$ stati non necessariamente distinti. Ma

poiché in tutto ci sono solo m stati, concludiamo (principio della cassettiera) che $\mathcal{A}$ deve essere in almeno uno stato più di una volta. Sia q uno stato in cui $\mathcal{A}$ si trova almeno due volte. Possiamo allora scrivere $x = uvw$, dove

$$\widehat{\delta}(q_0, u) = q \tag{7.5}$$

$$\widehat{\delta}(q, v) = q \tag{7.6}$$

$$\widehat{\delta}(q, w) \in F$$

Ciò significa che $\mathcal{A}$ arriva nello stato q per la prima volta dopo aver scritto l'ultimo simbolo di destra di u e poi dopo aver scansito l'ultimo simbolo di v. Poich questo "loop" può essere ripetuto un numero arbitrario di volte (anche 0), è chiaro che

$$\widehat{\delta}(q_0, uv^{[i]}w) = \widehat{\delta}(q_0, uvw) \in F$$

◯

Il lemma di espansione può essere usato per dimostrare che un certo linguaggio non è regolare. Usiamolo con riferimento a $\mathcal{L} = \{a^n b^n | n \geqslant 0\}$.
Supponiamo per assurdo che $\mathcal{L}$ sia regolare e sia m il numero di stati dell'automa $\mathcal{A}$ che genera il linguaggio. Scegliamo ora $x = a^m b^m$. Poiché x è un elemento di $\mathcal{L}$ e ha una lunghezza superiore a m, il lemma di estensione garantisce che esso può essere diviso in tre parti, $x = uvw$, in modo che per ogni $i \geqslant 0$ si ha $uv^{[i]}w \in \mathcal{L}$. Verificheremo ora che questo risultato è impossibile scegliendo $i = 2$. Ci sono tre possibili casi:

1. La stringa v consiste solo di 0. In questo caso, la stringa $uvvw$ ha più 0 che 1 e quindi non è un membro di $\mathcal{L}$.

2. La stringa v consiste solo di 1. Stesso ragionamento del caso precedente.

3. La stringa v consiste di 0 e di 1. In questo caso, la stringa $uvvw$ può avere lo stesso numero di 0 e di 1, ma ci saranno alcuni 1 prima di 0, e quindi la struttura della stringa non è quella prevista.

Bibliografia

[1] D. Antolini, comunicazione personale AA 2019/20.

[2] A. D'Amore, "I circuiti di commutazione", dispense.

[3] G. Bucci, "Calcolatori elettronici - Architettura e organizzazione", Mc Graw Hill Education, 2014.

[4] A. Clements, "Principle of computer hardware", Oxford University Press, 2005.

[5] N. Cutland, "Computability: an introduction to recursive function theory", Cambridge University Press, 1980.

[6] J.L. Casti, W. De Pauli, "Gödel, A Life of Logic", Perseus Publishing.

[7] M. Davis, "Il Calcolatore Universale", Adelphi, Biblioteca Scientifica 35.

[8] M. Davis, "Is Mathematical Insight Algorithmic?",
http://cs.nyu.edu/cs/faculty/davism/penrose.ps

[9] M. Davis, "How Subtle is Gödel's Theorem",
http://cs.nyu.edu/cs/faculty/davism/penrose2.ps

[10] J.J. Gray, "The Hilbert Challenge", Oxford Univ. Press.

[11] F. Fabris, "Teoria dell'Informazione, codici, cifrari", Bollati Boringhieri, Torino, 2001.

[12] G.O. Longo, "Il nuovo Golem - Come il computer cambia la nostra cultura", Edizioni Laterza.

[13] Z. Manna, "Teoria Matematica della Computazione", Bollati Boringhieri, Torino, 1978.

[14] J. Roulston, "A Rough History of Computing",
www.gadae.com/news/Papers/A Rough History of Computing.pdf

[15] R. Spelta, www.storiainformatica.it

[16] www.wikipedia.org

Elenco delle figure