# Alfabetizzazione Informatica

Renzo Sprugnoli
Dipartimento di Sistemi e Informatica
Via Lombroso 6/17 - Firenze (Italy)

21 ottobre 2010

# Indice

**1 STRUTTURA FISICA DELL'ELABORATORE** **5**
1.1 Struttura interna . . . . . 5
1.2 Le periferiche . . . . . 9
1.3 Le memorie di massa . . . . . 13

**2 DATI E INFORMAZIONI** **19**
2.1 Dati numerici . . . . . 19
2.2 Le operazioni . . . . . 23
2.3 Numeri e parole . . . . . 27
2.4 Informazioni grafiche e sonore . . . . . 32

**3 TRATTAMENTO DEI DATI** **37**
3.1 Sicurezza delle informazioni . . . . . 37
3.2 La privatezza . . . . . 40
3.3 La compressione dei dati . . . . . 44

**4 L'INTERNO DELL'ELABORATORE** **51**
4.1 La logica delle proposizioni . . . . . 51
4.2 Il linguaggio macchina . . . . . 56
4.3 I linguaggi di programmazione . . . . . 61

**5 ALGORITMI E STRUTTURE DATI** **67**
5.1 Algoritmi di base . . . . . 67
5.2 Le strutture ad albero . . . . . 71
5.3 Complessità e computabilità . . . . . 76

# Capitolo 1

# STRUTTURA FISICA DELL'ELABORATORE

## 1.1 Struttura interna

Gli odierni elaboratori sono stati progettati e costruiti fra la fine degli anni 1930 e l'inizio degli anni 1940, quando Inghilterra e Stati Uniti erano incalzati dalla necessità di progettare macchine da calcolo che potessero eseguire il grandissimo numero di elaborazioni richieste dalle esigenze belliche.

In particolare, in Inghilterra era sentita l'esigenza di mettere a punto un'apparecchiatura che fosse in grado di eseguire l'enorme quantità di combinazioni necessarie a decifrare i codici segreti dei tedeschi. Il progetto COLOSSUS era portato avanti da uno dei padri fondatori della moderna scienza degli elaboratori: Alan Turing. Costui, già nel 1936, aveva inventato le macchine astratte che portano il suo nome e che sono, ancora oggi, la base della teoria della *computabilità*, la disciplina che studia ciò che può essere e ciò che non può essere elaborato meccanicamente. Negli Stati Uniti erano invece più sentiti i problemi del calcolo numerico per la progettazione di grossi impianti bellici, per le scelte strategiche e per la logistica militare e civile.

Figura 1.1: Alan Turing

Qualche anno prima della guerra, Shannon aveva mostrato come il sistema binario permettesse una facile progettazione di calcolatrici elettriche. I suoi studi vennero ripresi e applicati al calcolo elettronico, cioè al calcolo effettuato da particolari circuiti controllati da valvole termoioniche, le valvole che consentivano l'amplificazione e la modulazione dei segnali della radio. Si poterono così ottenere calcolatrici che riuscivano ad eseguire centinaia di operazioni al secondo, una velocità enorme se paragonata a quella della calcolatrici meccaniche allora esistenti.

L'invenzione delle calcolatrici meccaniche risale al 1600. Nel 1624 Schickard, professore di teologia dell'Università di Tubinga, scrisse al suo amico Keplero descrivendogli una macchina da calcolo da lui costruita. Purtroppo, nulla rimane di questa invenzione, che perciò non ha avuto ripercussioni sulla storia successiva. Nel 1642 Pascal, appena diciannovenne, costruì una macchina da calcolo per aiutare il padre, ispettore delle tasse, ad eseguire senza errori le numerose operazioni che doveva effettuare. La macchina, detta *Pascaline*, eseguiva solo somme e sottrazioni. Quarant'anni dopo, Leibniz scoprì il modo per eseguire anche le moltiplicazioni.

Le calcolatrici meccaniche ebbero il momento di massimo sviluppo all'inizio del 1800, quando il matematico inglese Babbage costruì la *Macchina alle Differenze* e progettò la *Macchina Analitica*. Questa, per funzionare, aveva bisogno di una vera e propria programmazione e in questo Babbage fu aiutato da Ada Byron, la figlia del poeta, considerata oggi la prima programmatrice della storia. Può essere curioso ricordare che Babbage brevettò le sue macchine, e questo impedì che nel 1946 potessero essere brevettate le nuove calcolatrici elettroniche. Esse, infatti, sembravano eseguire lo stesso tipo di funzionalità, passando semplicemente dalla meccanica all'elettronica.

L'aspetto noioso del calcolo meccanico è che i numeri vanno impostati a mano e i risultati parziali devono essere ricordati a mente, o scritti su un foglio di carta, per essere nuovamente impostati quando ser-

vono. Questo rende il calcolo meccanico poco sicuro e molto lento, anche se più veloce e attendibile del calcolo manuale. Non si guadagna molto in velocità se da un'operazione al secondo si passa ad un'operazione ogni millesimo di secondo, quando comunque occorrono venti secondi per inserire i dati. Bisogna allora che i dati vengano impostati una volta per tutte all'inizio dei calcoli e che i risultati parziali vengano ricordati dalla stessa macchina, in modo da poter essere riutilizzati quando necessario. Questo impone la presenza di una *memoria*, nella quale inserire i dati di partenza del problema da risolvere, immagazzinare i risultati parziali e finali, che verranno poi scritti in chiaro, ad esempio su una stampante.

Naturalmente, una calcolatrice elettronica deve possedere una serie di circuiti che le permettano di eseguire le operazioni, come somme, sottrazioni, moltiplicazioni, divisioni e radici, per non parlare di logaritmi, esponenziali, seni e coseni. Questi circuiti costituiscono il cuore operativo della calcolatrice e tecnicamente si dicono, nel loro insieme, l'*Unità Centrale di Elaborazione*, o *CPU*. Negli elaboratori di tipo personal, che tutti oggi conosciamo, la CPU è costituita dal microprocessore, un aggeggino di plastica nera con tanti piedini, di un paio di centimetri quadrati di superficie. I grandi elaboratori, noti come *Main-Frame*, hanno una CPU molto più grande e potente, ma che esegue le stesse operazioni e le stesse funzionalità.

Negli anni 1940 e 1950, la CPU, composta da *valvole*, occupava interi armadi e diverse stanze erano necessarie a contenere una sola calcolatrice. Verso la metà degli anni 1950 entrarono in commercio i *transistor* e le dimensioni degli elaboratori diminuirono drasticamente e solo pochi anni più tardi, nei primi anni del 1960, prese avvio la tecnologia dei *circuiti integrati*. Da allora gli elaboratori sono diventati più contenuti: oggi il microprocessore è costituito da un circuito integrato; anche la memoria sfrutta la stessa tecnologia e ciò ha permesso un'estrema miniaturizzazione degli elaboratori, come dimostrano i nostri personal.

I circuiti integrati sono una tecnologia da microscopio, o *nanotecnologia*, come si ama dire oggi. Si usa una lamina di pochi millimetri quadrati (detta *wafer*) di un metallo semiconduttore, come il silicio, il gallio o il germanio. Il metallo, purissimo, viene *drogato*, così si dice, con alcune impurità, che agevolano o impediscono il passaggio dell'elettricità, ed eseguono così le stesse operazioni di base delle valvole e dei transistor. Solo che questi circuiti occupano infinitamente meno spazio, tanto che oggi si riescono a stipare milioni di componenti per centimetro quadrato. E già si parla di nuove tecnologie che permetteranno di arrivare alla singola molecola o al singolo atomo.

Con l'evoluzione tecnologica dalle valvole, ai transistor, ai circuiti integrati, è cresciuta proporzionalmente la velocità degli elaboratori, cioè il numero di operazioni che essi riescono ad effettuare ogni secondo. Dalle poche centinaia di operazioni al secondo negli anni 1940, con l'anno 2000 si è superato il miliardo. Moore[1] ha calcolato che dal 1940 ad oggi la velocità di elaborazione è cresciuta di un fattore 10 ogni 10 anni, con un'impennata esponenziale che non ha riscontro in nessun altro settore tecnologico. Di pari passo sono andati calando i prezzi, ed oggi un personal è pressoché alla portata di tutti. Le sue prestazioni sono milioni di volte superiori a quelle dei più grossi, e costosissimi, elaboratori degli anni '40, '50 e '60.

La memoria per custodire i dati e la CPU per eseguire le operazioni elementari sono due parti essenziali dell'elaboratore, ma un problema complesso ha bisogno, per essere risolto, di decine di migliaia di queste operazioni, disposte in una sequenza opportuna. E' questo il concetto di *programma*, cioè la successione di operazioni che l'elaboratore deve eseguire per risolvere un problema. Ad esempio, per trovare la media di 5 numeri A, B, C, D, E deve eseguire quattro somme:

$$R = A + B;\ S = R + C;\ T = S + D;\ U = T + E.$$

A questo punto deve dividere la somma $U$ per 5, il numero complessivo dei dati. Vedremo più avanti di approfondire questo concetto, ma per il momento accontentiamoci di aver messo in evidenza come un programma sia essenziale all'uso dell'elaboratore.

Prima del 1945, secondo un metodo che risaliva a Babbage, il programma era qualcosa di esterno all'elaboratore. In qualche modo, veniva scritto su schede perforate, su un nastro di carta perforato, oppure impostato con spinotti su un pannello elettrico. Questo rallentava la sua esecuzione, anche se migliorava le prestazioni manuali. Finalmente, nel 1945, von Neumann, un matematico ungherese naturalizzato americano, propose di inserire il programma nella memoria stessa dell'elaboratore. Da allora, tutte queste macchine si sono adeguate al metodo di von Neumann, o del programma interno. L'architettura degli elaboratori, di conseguenza, è stata detta di von Neumann e ancora oggi gli elaboratori funzionano con i dati e il programma nella memoria centrale, mentre le istruzioni sono eseguite una dietro l'altra dalla CPU.

Se l'esecuzione di un programma impegna solamente la memoria e la CPU, occorre però che i dati e il programma siano "immessi" nella memoria, così come alla fine si desidera vedere i risultati che sono

[1] La formulazione originale di Moore si riferiva al numero di transistor per centimetro quadrato nei circuiti integrati

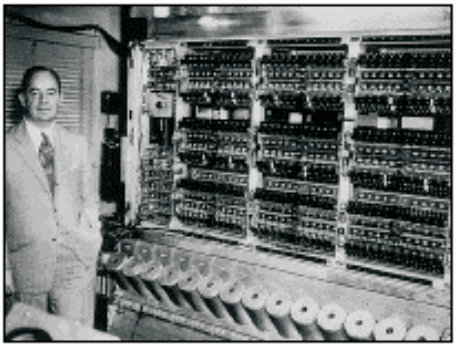


Figura 1.2: John von Neumann

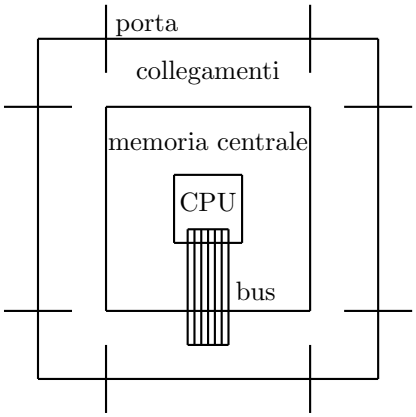



Figura 1.3: Schema di elaboratore

stati ottenuti. C'è bisogno, perciò, di una qualche apparecchiatura che inserisca dati e programma nella memoria e di un'altra apparecchiatura che riesca ad estrarne i risultati.

Esistono molte di queste apparecchiature; esse, però, devono essere collegate alla memoria dell'elaboratore in modo opportuno. Ciò avviene per mezzo di alcuni circuiti, anch'essi elettronici, che fanno giusto da interfaccia tra il cervello dell'elaboratore, cioè la sua memoria, e il mondo esterno. Senza questo, l'elaboratore sarebbe una macchina sorda e muta, del tutto inutilizzabile perché non potremmo comunicare con essa. Questi *circuiti di collegamento* costituiscono la terza e ultima parte dell'elaboratore. Essi vengono detti "*gate*" o *cancelli* o *porte*, e quest'ultimo è il nome più accettato in italiano, anche se sembra riferirsi, in modo troppo specifico, alle prese in cui inserire fisicamente gli spinotti di connessione.

Ribadiamo esplicitamente che si deve distinguere fra i circuiti di collegamento, interni all'elaboratore, e le apparecchiature esterne che, tramite le porte fisiche, vengono a quello collegate. Le apparecchiature, dette *periferiche*, possono essere le più svariate, e diverse periferiche si possono collegare all'elaboratore con lo stesso tipo di porta, cioè di circuiti interni di collegamento. Vedremo più avanti i principali tipi di periferiche, quelle oggi più utilizzate, e come si sono evolute nel tempo. Vedremo anche come, logicamente, avviene il trasferimento dei dati e dei programmi dalla e verso la memoria.

Riassumendo, un elaboratore è costituito da tre parti, che si possono raffigurare a buccia di cipolla. L'Unità Centrale di Elaborazione, o CPU, esegue le istruzioni di base ed è la parte più interna. La memoria centrale è la parte di mezzo, che comunica con la CPU e contiene dati e programmi. Infine, la parte più esterna, è formata dai circuiti di collegamento, che comunicano con la memoria, ma non con la CPU. Sono poi questi circuiti che, attraverso le porte, comunicano con il mondo esterno. La rappresentazione mette in evidenza le comunicazioni interne fra i tre componenti. Queste avvengono per mezzo di una linea di trasmissione bidirezionale, detta bus. Il bus permette lo scambio dei dati fra i circuiti di collegamento e la memoria, e tra la memoria e la CPU.

L'insieme dei circuiti, che costituiscono le varie parti di un elaboratore e le periferiche che lo contornano, formano il cosiddetto *hardware*, cioè la parte fisica. In contrapposizione, i programmi che l'elaboratore esegue e i dati che esso gestisce si dicono il *software*, cioè la parte impalpabile. Con un'iperbole esagerata, ma significativa, lo hardware è stato paragonato al corpo e il software all'anima.

Come abbiamo detto, la struttura appena vista è comune a tutti gli elaboratori, dai più piccoli ai più grandi. Ciò che differenzia le fasce in cui si sogliono suddividere è solo la capacità della memoria, la velocità della CPU e la complessità dei circuiti di collegamento. Ad esempio, la fascia più bassa è costituita dalle *calcolatrici*, cioè gli elaboratori tascabili che ci permettono di fare calcoli numerici e, talvolta, di immagazzinare un certo numero di informazioni. In questa calcolatrici, la CPU è in grado di effettuare le quattro operazioni di base e, in quelle più sofisticate, radici, logaritmi e funzioni trigonometriche. La memoria è limitata a pochi numeri e i collegamenti con l'esterno sono per lo più ristretti alla tastiera numerica e al display.

La grossa limitazione delle calcolatrici è quella di non essere programmabili. Comunque, oggi, sono in commercio alcune calcolatrici parzialmente programmabili, che costituiscono il ponte di passaggio alla fascia successiva, quella dei *calcolatori palmari*. Qui, la CPU è completa di tutte le operazioni aritmetiche e logiche; la memoria è ampia e arriva a qualche milione di informazioni elementari; i collegamenti permettono di comunicare con la rete e con altri elaboratori. Og-

gi i palmari sono da considerare come elaboratori di tipo generale, limitati solo dalle dimensioni, che vogliono essere estremamente ridotte per permetterne l'uso in ogni circostanza. Le dimensioni condizionano anche la tastiera e il display, e sono forse proprio questi i fattori più limitanti.

La fascia successiva è quella dei *personal*. Questi elaboratori sono veramente completi, oggi più potenti di tutti i grandi elaboratori del passato. La CPU è un microprocessore di tipo generale, con velocità di elaborazione paragonabile a quella degli elaboratori più grossi. La memoria supera il miliardo di informazioni elementari e i collegamenti sono possibili con ogni tipo di periferica. Le differenze tra i vari tipi di personal sono dovute alle dimensioni fisiche. I *portatili* sono più piccoli e devono soddisfare particolari caratteristiche di robustezza. Gli *elaboratori da tavolo* sono più grandi e possono essere assemblati secondo i desideri dell'utente. Ciò permette di variarne la potenza e i costi, così che la gamma delle possibilità è pressoché infinita.

Un personal particolarmente veloce e capiente si dice talvolta una *stazione di lavoro*. Questo termine, che in inglese suona *workstation*, è oggi meno usato di qualche anno fa. In una rete si usa l'architettura detta *client/server*. I *client* sono personal che possono lavorare da soli o scambiando informazioni con gli altri client della rete. Una stazione di lavoro si assume il compito di *server*, cioè di elaboratore di collegamento, di supervisione della rete e di centro di elaborazione per i dati comuni o condivisi. Questi dati, aggiornati continuamente, sono disponibili a tutti i client secondo opportune modalità.

Le grandi aziende hanno necessità di calcolo più avanzate. Spesso i collegamenti non possono essere semplicemente locali, ma devono essere a lunga distanza, o *remoti*, come si usa dire. Gli utenti possono essere centinaia, migliaia o decine di migliaia. La struttura client/server è ancora utilizzata, ma l'elaboratore centrale deve essere molto più potente. Esistono vari tipi di macchine che, a seconda delle dimensioni, si dicono minielaboratori, elaboratori medi o grandi. Le funzionalità sono le stesse di tutti gli altri, ma la CPU è particolarmente sofisticata per rendere più veloce l'esecuzione dei programmi e altri processori eseguono operazioni di contorno. Le periferiche son tali da soddisfare le esigenze di tutti gli utenti.

La fascia più elevata, per costi e prestazioni, è quella degli elaboratori *main-frame*. Questi sono i colossi dell'elaborazione elettronica e si possono trovare nelle grandi imprese, nelle istituzioni pubbliche militari e civili, nei centri di ricerca più importanti. Di regola questi elaboratori sono doppi, secondo un'architettura detta *a specchio*. Si immagini infatti che, per un qualche guasto o un qualche inconveniente, si debba interrompere il lavoro di un tale mostro. Decine di migliaia di utenti ne verrebbero a subire le conseguenze, con danni gravissimi. Allora la struttura dell'elaboratore viene duplicata, usando due elaboratori *gemelli*, così da permettere la perfetta ripetizione dei dati e delle istruzioni eseguite, proprio come in uno specchio.

In caso di guasto di una delle due parti, questa viene bloccata e l'elaboratore va avanti con la parte buona. Nessuno degli utenti avverte l'inconveniente. Nel frattempo si provvede a riparare, più celermente possibile, il guasto, sostituendo i componenti difettosi. Appena conclusa la riparazione, si provvede al cosiddetto *allineamento* delle due parti. I dati della parte corretta vengono ricopiati nella parte che si era guastata e, ciò fatto, si riprende l'esecuzione parallela di tutte le istruzioni. Così le due parti proseguono assieme, pronte eventualmente a far fronte ad un'ulteriore interruzione. E' bene osservare che l'odierna tecnologia è molto affidabile e i guasti sono piuttosto rari.

Oggi, non solo i main-frame, ma anche molti server hanno un'architettura a specchio, per le stesse ragioni di sicurezza. Vedremo a suo tempo come l'elaboratore si accorge, autonomamente, di certi errori. La sicurezza dei dati è un problema generale e nessuno ama perdere informazioni che gli sono costate ore o giorni di lavoro. Ogni utente di personal sa, o impara sulla propria pelle, ad eseguire procedure di sicurezza, ad esempio effettuando periodiche copie dei propri dati e dei propri programmi. Tali copie, dette di *back-up*, conservate gelosamente, sono una vera e propria assicurazione sul lavoro svolto.

E' opportuno ricordare che oggi un elaboratore ha, di contorno alla propria CPU, altri *processori* o microprocessori, che pertanto non si possono dire centrali. Tali processori, in parallelo a quello centrale, eseguono compiti specifici. Ad esempio, se un programma deve acquisire molti dati da un'apparecchiatura periferica, la CPU si limita a lanciare il comando di lettura, dopo di che questa viene controllata da un processore indipendente, che porta avanti le molte operazioni necessarie. Intanto la CPU esegue altre istruzioni, ad esempio relative a calcoli che non utilizzano i dati in lettura. Alla fine della lettura, il processore avverte la CPU di aver concluso, dandogli così il benestare a iniziare l'elaborazione dei dati letti. Ciò permette di risparmiare tempo, evitando che la CPU aspetti inoperosa la lettura, di regola molto più lenta dell'elaborazione.

Questi processori ausiliari, per quanto detto, lavorano in modo indipendente rispetto alla CPU e l'uno rispetto all'altro. Tecnicamente, si dice che lavorano in *modalità asincrona*. D'altra parte, tanto la CPU

quanto gli altri processori, sono macchine *sincrone*: le loro operazioni avvengono in successione, scandite da un orologio interno, detto *orologio del sistema*. Tale orologio definisce il ciclo della macchina, e quindi la sua velocità.

Uno dei compiti della CPU è di raccogliere i segnali di inizio e fine lavoro di tutti i processori. Occorre però un programma che gestisca tutte queste informazioni, coordinando le relative operazioni in modo da svolgere correttamente il programma richiesto dall'utente. Questo programma generale è noto come *Sistema Operativo*, e tutti conoscono, almeno di nome, Windows e Linux. Il Sistema Operativo di un elaboratore è altrettanto importante dello stesso hardware, e infatti è venduto insieme a quello. Senza Sistema Operativo un elaboratore è inutilizzabile. Per l'utente sarebbe impossibile coordinare le molteplici attività che un elaboratore svolge. Alcune sono direttamente correlate all'utente, che vuole eseguire le proprie procedure, ma la maggior parte delle azioni dell'elaboratore sono eseguite autonomamente, di corredo alle principali.

Un'elaborazione può procedere, ed eseguire una certa attività, quando tutte le azioni che la devono precedere sono state completate. Il completamente di un'azione specifica, portata a termine da un processore, è segnalato da un comando speciale, detto *interruzione* o, con termine inglese, *interrupt*. Un'interruzione è come lo squillo di una sveglia e dice alla CPU: ho finito quello che dovevo fare; per me è l'ora che tu vada avanti. In Sistema Operativo raccoglie questi segnali e, quando tutto è a posto, procede. Naturalmente, segnalerà un errore, se anche una sola delle azioni termina male.

Oggi, il Sistema Operativo ha anche molte altre attività da svolgere, ed è diventato uno dei programmi più complessi di un elaboratore, in qualche modo molto più complesso dello hardware. Per questo, esso caratterizza un elaboratore al pari del microprocessore, della quantità di memoria centrale e della capacità del disco rigido. Ne vedremo perciò a suo tempo alcune delle principali caratteristiche.

## 1.2 Le periferiche

Come abbiamo detto, un elaboratore senza periferiche sarebbe sordo e muto, e quindi del tutto inutilizzabile. D'altra parte, le apparecchiature che vorremmo collegare all'elaboratore sono pressoché infinite, mentre i circuiti di collegamento, cioè le porte, devono essere previste a priori e devono essere in numero limitato. Ciò comporta che, da una parte, le periferiche devono sottostare ad alcuni standard di comunicazione e, dall'altra, l'elaboratore deve prevedere una serie di modalità che, in funzione della specifica apparecchiatura, traducono, per così dire, i segnali in ingresso in dati elaborabili, e viceversa.

Queste modalità sono realizzate da opportuni programmi, detti *driver*, cioè guide. Essi fanno parte del Sistema Operativo dell'elaboratore. Se pure un driver fa riferimento a periferiche specifiche, queste sono comunque divise in classi a seconda delle funzioni che svolgono e di come le svolgono.

Una più ampia classificazione prevede tre tipi di apparecchiature, a seconda del fatto che esse permettano solo di trasferire dati dal mondo esterno all'elaboratore, permettano solo di trasferire dati dall'elaboratore al mondo esterno, o permettano, infine, tutte e due le modalità di trasferimento. Per rendersi ben conto della terminologia usata, occorre immaginare di essere collocati nella memoria centrale dell'elaboratore, cioè di antropoformizzare la macchina e mettersi dal suo punto di vista. Questo può essere un po' seccante, ma non certo difficile.

Le apparecchiature che portano dati dal mondo esterno alla memoria centrale dell'elaboratore si dicono di *ingresso*, e le operazioni che svolgono si dicono di *lettura*. Pertanto, si parla anche di *periferiche di lettura*. Viceversa, quelle che trasferiscono dati dalla memoria centrale al mondo esterno si dicono di *uscita*, e le operazioni da esse svolte si dicono di *scrittura*. Si parla perciò anche di *periferiche di scrittura*. Naturalmente, le periferiche che trasferiscono i dati in entrambe le direzioni si dicono *periferiche di ingresso e uscita*, oppure *di lettura e scrittura*. Purtroppo, molto usate sono le parole inglesi *input*, per ingresso, e *output* per uscita. Cercheremo, per quanto possibile, di evitare questi termini.

La più tipica, e indispensabile, periferica di ingresso è la *tastiera*. Attraverso la tastiera, noi introduciamo i nostri dati e i nostri programmi nella memoria dell'elaboratore. Quando premiamo un tasto, la tastiera invia una sequenza di impulsi elettrici che, convenzionalmente, individuano il carattere corrispondente al tasto. Ogni carattere, anche lo spazio, ha una sequenza di otto impulsi che lo identificano. Questi impulsi sono codificati in una posizione di memoria, esattamente in un byte, come più avanti spiegheremo con maggiori dettagli. Questa posizione della memoria è stabilita dal Sistema Operativo e propriamente fa parte di una sequenza di posizioni detta *buffer* o, in italiano, *memoria tampone*.

I vari caratteri premuti dall'utente si accumulano nel buffer uno dietro l'altro. L'accumulo di caratteri va avanti fino a che non premiamo il *tasto INVIO*. Questo è un comando di interruzione che avverte il Sistema Operativo della conclusione, almeno parziale, della nostra operazione di immissione dati. Il buffer, allora, passa sotto il controllo del programma che si sta eseguendo. Il buffer viene liberato dei caratteri

immessi ed è pronto ad accettare un'altra sequenza.. In questa maniera, mentre noi effettuiamo l'immissione di altri dati, l'elaboratore può elaborare quelli appena acquisiti. Ad esempio, se stiamo scrivendo una lettera, mentre noi scriviamo la seconda linea, la macchina dà la forma corretta alla prima e la impagina. In realtà, in questi casi, l'elaboratore lavora assai poco, data la sua velocità, ed è costretto ad aspettarci. Come vedremo, riesce di solito ad utilizzare in modo intelligente questi tempi morti.

Le tastiere non sono tutte uguali. La tastiera americana è detta QWERTY, dalla sequenza dei caratteri con cui comincia la parte alfabetica. Quella italiana si dice QZERTY, poiché porta la Z al posto della W. Sulla tastiera italiana, ad esempio, le cifre si scrivono con il tasto delle maiuscole, mentre in quella americana sono minuscole. Oggi, tuttavia, la tastiera italiana si è adeguata a quella americana ed è una tastiera QWERTY anch'essa (vedi Figura 1.4). Anche le cifre sono minuscole e, in generale, i caratteri occupano posizioni analoghe a quella americana. Ricordiamo che la tastiera americana non contiene le vocali accentate, che sono invece presenti in quella italiana. Passare da una tastiera all'altra può presentare qualche difficoltà, almeno all'inizio.

Alcuni caratteri richiedono combinazioni abbastanza complesse di tasti. Oltre al *tasto per le maiuscole*, detto *shift-key* in inglese, esistono due altri tasti di modifica: il *tasto ALT*, per "alternate" ovvero alternativo, e il *tasto CTRL*, per "controllo". Essi permettono di ottenere caratteri particolari, ed anche caratteri speciali che non corrispondono a nessun carattere scrivibile, ma hanno un ruolo particolare. Ad esempio, un carattere di controllo è quello che permette di andare a capo: non si vede, ma ha un preciso effetto. Altri caratteri di controllo sono quelli che determinano il movimento del cursore sullo schermo.

Un'altra periferica indispensabile è il *monitor* o *video*; si tratta di una tipica apparecchiatura di uscita e di regola è usata per far vedere i risultati delle elaborazioni che abbiamo chiesto al nostro elaboratore. Oggi, tutti i monitor hanno caratteristiche grafiche, cioè sono formati da un grande numero di punti, detti *pixel*[2], tipicamente 1024 per 768, cioè circa 800 mila. Illuminati opportunamente, essi possono riprodurre tutte le fonti di caratteri e tutte le immagini. Sul video, possiamo quindi vedere fotografie, diagrammi, disegni e pagine di testo. Una volta i monitor erano molto più limitati e permettevano di visualizzare solo lettere e cifre.

Anche il video, come la tastiera, utilizza un buffer per rendere più veloci le operazioni e poter procedere in parallelo con altre elaborazioni. Il buffer del video contiene, almeno, un'intera schermata o addirittura diverse schermate. Queste vengono composte da speciali programmi di visualizzazione. Per formare una schermata occorre circa un millesimo di secondo. Ogni 20 millisecondi il contenuto del buffer viene trasferito sul video, la cui immagine viene così rinfrescata ed eventualmente aggiornata. In questo modo noi percepiamo 50 immagini al secondo, che ci danno la sensazione della stabilità dell'immagine. Ricordiamo che per questo sono sufficienti 16 immagini al secondo.

Parleremo a suo tempo della formazione dei colori. Per ora, basta rammentare che, durante una seduta, sul video possiamo vedere anche i dati che immettiamo da tastiera. Questo è un servizio speciale, detto *echoing*, offerto dall'elaboratore, molto utile per controllare l'esattezza di ciò che scriviamo. Nonostante questa capacità, che comunque è solo di uscita, il monitor è una periferica di sola scrittura. Per speciali applicazioni, esistono schermi video sensibili, che permettono l'ingresso dei dati, ad esempio se vengono avvicinati o toccati con un dito o una punta smussata. Questi si dicono *touch-scree*n, ma non è il caso dei normali monitor. Tradizionalmente, i monitor sono molto ingombranti e si indicano con la sigla CRT cioè *Cathod Ray Tube*. Oggi hanno ormai preso piede i monitor piatti, a cristalli liquidi o a plasma; anche se sono un po' più costosi sono molto più pratici poiché occupano molto meno spazio.

Un'altra tipica apparecchiatura di uscita è la *stampante*. Più lenta del monitor, permette però di avere documenti permanenti, e non volatili come le immagini sullo schermo. La storia dell'evoluzione delle stampanti è molto interessante e mostra cosa comporta il succedersi di scoperte tecnologiche.

Le prime stampanti erano derivate dalle macchine da scrivere. I caratteri erano formati in rilievo su una catena che scorreva e, quando arrivava il simbolo desiderato, batteva su un nastro inchiostrato stampando il carattere sul foglio di carta. Queste stampanti avevano pertanto un insieme fisso di caratteri. Stampanti più lente utilizzavano palline o margherite intercambiabili. Questo permetteva di avere più fonti di caratteri, ma la stampante si doveva fermare per consentire il cambio.

Si inventarono allora le *stampanti ad aghi*. Gli aghi, battendo sul solito nastro inchiostrato, permettevano di scrivere i vari caratteri. La stampante aveva un proprio programma che le permetteva di far scorrere gli aghi e battere quelli desiderati. In questo modo era teoricamente possibile scrivere ogni carattere in una qualsiasi fonte. La qualità della stampa dipendeva dal numero di aghi a disposizione: più aghi si avevano, più fitti erano i punti stampati e più chiaro risultava il carattere ottenuto.

[2]La parola *pixel* è artificiale e sta per "picture element", cioè *elemento grafico*

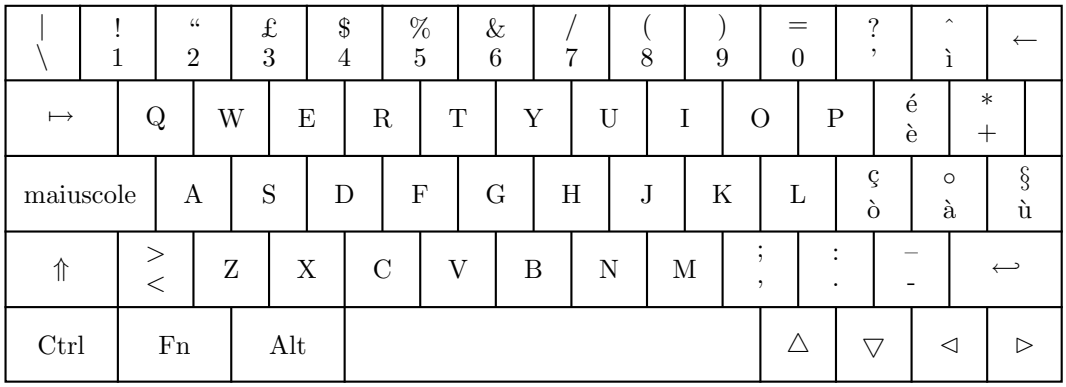



Figura 1.4: La tastiera italiana - QWERTY

Una migliore definizione e meno movimento di parti meccaniche si ottennero con le *stampanti a bolle d'inchiostro*. Queste stampanti schizzavano l'inchiostro sulla carta in modo preciso e in quantità puntiformi. Ciò permise di ottenere qualsiasi carattere, in qualsiasi fonte e dimensione, in modo veloce e pulito. Oggi, questo tipo di stampante, detto a *getto d'inchiostro*, è quello di più larga diffusione. I costi sono molto bassi e la spesa maggiore è legata al consumo di inchiostro. Si ottengono facilmente stampe a colori, e, usando carte lucide speciali, la riproduzione di fotografie. Un aspetto importante, che si tende a sottovalutare, è che queste stampanti sono molto silenziose.

Altrettanto silenziose e più veloci, anche se più ingombranti, sono le *stampanti laser*. Queste stampano tranquillamente dodici o più pagine al minuto, anche fronte-retro. Ricordiamo che il formato standard A4, sul quale sono regolate stampanti e fotocopiatrici, è cm 21 per 29.7, cioè esattamente $21\sqrt{2}$. Il formato A3 è il doppio, mentre il formato A5 è la metà, così che il rapporto fra altezza e base è sempre $\sqrt{2}$.

Oggi esistono stampanti che svolgono funzioni diverse. Hanno incorporato uno scanner, di cui diremo tra poco, e, collegate al telefono o alla rete, oltre che all'elaboratore, permettono di preparare, inviare e ricevere FAX. A sé stanti, poi, possono essere usate come fotocopiatrici.

Lo *scanner* costituisce un altro tipo di periferica. Queste macchine hanno un ripiano sul quale viene posto il foglio, o il libro aperto da scandire. Un raggio di luce scorre il foglio da cima a fondo secondo linee molto ravvicinate tra di loro. Ogni linea, in realtà, è scandita secondo una successione di punti, di modo che l'intero foglio è visto come una sequenza di punti, o *dot*, come vengono chiamati in inglese. Ogni punto è registrato o in bianco e nero oppure a colori, secondo una scala molto ampia. Normalmente, uno scanner esegue questa scansione a 300 *dot per inch* (dpi), cioè 300 punti ogni pollice.

Come è noto, il pollice misura cm 2.54. Una linea di 21 cm corrisponde perciò a 2480 punti. In un foglio si hanno in tutto 3507 linee e di conseguenza $8 \cdot 697 \cdot 360$ punti. Essi costituiscono una fedele riproduzione del contenuto del foglio e possono essere registrati nella memoria dell'elaboratore e riprodotti, a piacere, da una stampante. Questa tecnica permette di memorizzare una qualsiasi immagine o un qualsiasi testo. I 300 punti per pollice possono essere variati a comando. Diminuendoli, si risparmia spazio nell'elaboratore, ma l'immagine risulterà meno accurata. Aumentandoli, si avrà l'effetto opposto. Per immagini particolarmente importanti, come ad esempio la riproduzione di quadri d'autore, si può arrivare a scandire e registrare fino a 3000 punti per pollice.

Lo scanner ci dà la riproduzione fedele di un'immagine, ma se l'immagine è un testo saremmo piuttosto interessati ad ottenere il contenuto del testo stesso, cioè le parole e i segni che lo formano, e non i punti che determinano la scrittura di tali parole. Se, come di regola succede, l'elaboratore ha uno speciale programma detto *OCR* legato allo scanner, questo è possibile. OCR significa *Optical Character Recognizer*, cioè "riconoscitore ottico di caratteri", e il programma è in grado di riconoscere i singoli caratteri, e quindi le parole contenute nel testo. Una volta registrato nell'elaboratore, il testo può essere elaborato, ad esempio da un programma di trattamento dei testi come Word, ed eventualmente stampato in un formato completamente diverso.

Una periferica che oggi ha assunto un ruolo molto importante è il *mouse*, parola che nessuno osa tradurre in italiano. Si tratta di un'apparecchiatura di solo ingresso, e che sta assumendo funzioni sempre più complesse. Di fatto è una *periferica di puntamento*. Questo significa che il cursore ad essa legato può essere spostato ovunque sullo schermo.

Quando si preme un tasto (sinistro o destro) vengono spedite alla memoria dell'elaboratore le coordina-

te del punto dello schermo puntato in quel momento. Questo è tutto quello che fa il mouse, ma le coordinate vengono usate da un programma per innescare un'operazione specifica, diversa se si è pigiato il tasto destro o quello sinistro. Le coordinate dicono all'elaboratore dove sta puntando il mouse; questo conosce il contenuto dello schermo, poiché è lui che lo visualizza, e quindi innesca l'operazione associata a quella zona.

A questo proposito, tipico è l'uso dei *bottoni*. Un bottone è una zona dello schermo individuata, di solito, da un rettangolo o da un cerchio. Nella figura è scritta l'operazione da eseguire, così che *cliccare* il mouse equivale a comandare l'esecuzione dell'operazione. In una schermata possono essere presenti vari bottoni e qualcuno corrisponde a un menù a tendina. Cliccando il mouse, viene mostrato il menù, cioè un elenco di possibili operazioni. Si sposta il mouse su quella che interessa e un nuovo clic innesca l'operazione. Ciò che ora si fa con un clic o due, una volta bisognava scriverlo esplicitamente. Era quindi più facile sbagliare oppure non ricordarsi del nome specifico dell'operazione da eseguire: taglia, cancella o elimina? Stop, esci o termina? Il bottone o il menù suggeriscono l'azione corrispondente, rendendo più semplice la vita dell'utente.

Vista l'utilità, il mouse si va complicando e apparecchiature più o meno equivalenti sono disponibili, ed utilizzate in particolari contesti. Ben noto è il *joy-stick* usato nei videogiochi. Il *touch-pad* sostituisce il mouse nei portatili; infatti il mouse è piuttosto ingombrante e non si addice a questi elaboratori.

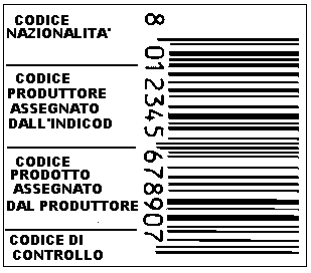



Figura 1.5: Struttura del codice a barre

Una periferica oggi molto diffusa è costituita dai *lettori di codice a barre*. Tutti i prodotti in vendita ai supermercati sono marcati con questi codici (si veda la Figura 1.5), che ormai sono diventati molto familiari, anche se rimangono assai misteriosi. Ogni azienda ha un proprio codice e ne assegna un altro a ciascuno dei suoi prodotti; l'accoppiata dei due codici individua univocamente il prodotto, come il codice fiscale individua il contribuente. Quando si fa la spesa, alla cassa il lettore legge il codice a barre, l'elaboratore abbina a questo codice il prezzo, eventuali sconti e promozioni, e prepara il conto, sperabilmente senza errori. Anche libri, giornali, oggetti vari hanno il loro codice a barre, talvolta addirittura due, a seconda dei circuiti di vendita che possono seguire.

Il codice a barre, nella sua apparente semplicità, è il realtà un codice molto complesso e regolato da convenzioni internazionali. L'aspetto più sofisticato è costituito dal fatto che il codice deve essere letto in condizioni molto difficili e non deve dare adito a possibili errori. Come tutti sappiamo per esperienza, avendolo osservato tante volte, il lettore opera a distanza, il codice può non essere perfettamente spianato o, addirittura, non essere allineato con il lettore. Il codice possiede anche alcune caratteristiche che permettono il controllo automatico della correttezza, così che l'utente viene avvisato se, per qualche motivo, la lettura non è avvenuta regolarmente.

Un'altra apparecchiatura di solo ingresso è il *microfono*, che permette di registrare suoni e voce secondo una tecnica, detta di *campionamento*, che vedremo a suo tempo in qualche dettaglio. Queste registrazioni possono essere ascoltate per mezzo delle *casse*, che sono apparecchiature d'uscita. L'operazione inversa della campionatura converte i dati registrati in suoni o voci, che riproducono quelli registrati. La tecnica è quella stessa usata per i CD musicali e consiste nella *digitalizzazione* dei dati analogici corrispondenti ai suoni. Come le immagini, anche i suoni occupano una grande quantità di memoria.

Si dicono *analogiche* le grandezze che variano con continuità, come le lunghezze, le differenze di potenziale o le temperature. In opposizione, si dicono *digitali* le grandezze che variano in modo discreto. Ne sono esempi i numeri interi e i punteggi. L'elaboratore può trattare solo informazioni digitali, che sono più controllabili e sicure, per le ragioni che vedremo. I CD musicali sono un esempio di digitalizzazione di informazioni analogiche, quali sono i suoni, ovvero la voce e la musica.

Molti elaboratori sono usati come apparecchiatura di controllo, ad esempio per regolare altre macchine; si pensi ad un forno o a una macchina utensile. In questi casi la macchina è connessa all'elaboratore mediante un *convertitore analogico/digitale*, che trasforma quantità continue in quantità discrete, e viceversa. Così una lunghezza, una temperatura o un angolo possono essere trattati come quantità discrete.

Altre periferiche esistono e saranno inventate, legate a sempre nuove applicazioni di cui gli elaboratori man mano acquisiscono il controllo. Anche la facilità d'uso è una notevole spinta a continuare le ricerche in questo settore. Vorremmo però ricordare come, in passato, altre periferiche hanno avuto una parte importante nello sviluppo degli elaboratori. Primo fra tutte il *perforatore di schede*: fino a che le *schede* sono

state un supporto usato per i dati e per i programmi, queste macchine sono state quasi insostituibili. In alternativa, sono stati utilizzati anche i lettori e perforatori di nastri di carta, ma anche questi, oggi, hanno esaurito il proprio compito.

Un settore molto particolare, ma anche molto importante, è quello delle *periferiche per disabili*. Si tratta, per lo più, di apparecchiature di ingresso, ma anche di alcune periferiche d'uscita. Il caso più interessante è il *sintetizzatore della voce*, che permette di trasformare un testo in suoni che simulano la voce umana. Per un cieco, una voce monotona e meccanica è già molto meglio di niente. Per quanto riguarda l'ingresso dei dati, esistono tastiere per i non vedenti, speciali apparecchi per paraplegici e strumenti costruiti appositamente per situazioni particolari e perfino periferiche azionate dal movimento degli occhi. Il caso del fisico Hawking è giustamente famoso, e l'Informatica può essere un mezzo per mettere in contatto col mondo esterno persone che, altrimenti, non potrebbero comunicare.

In questa sezione non ci siamo occupati, di proposito, di quelle che probabilmente sono le periferiche più importanti, almeno dopo tastiera, video e stampante. Ci riferiamo alle così dette *memorie di massa*, quelle memorie che ci permettono di conservare ed utilizzare grandi quantità di dati e di programmi. La memoria centrale, infatti, essendo un'apparecchiatura elettronica, è *volatile*, cioè perde il suo contenuto quando spengiamo l'elaboratore. Abbiamo perciò bisogno di *memorie permanenti*, nelle quali mantenere le nostre informazioni. Queste memorie, dette anche secondarie, si rifanno ad almeno due tecnologie: la registrazione magnetica e quella ottica, attraverso l'uso del laser. Dedicheremo perciò a queste periferiche e al loro utilizzo, tutta la prossima sezione.

## 1.3 Le memorie di massa

La memoria centrale dell'elaboratore è spesso indicata con l'acronimo RAM, che sta per *Random Access Memory*, letteralmente *Memoria ad Accesso Casuale*. Questa sigla non deve portarci fuori strada. L'accesso non è casuale perché fatto a caso, ma la parola "random" sta ad indicare che l'accesso a un dato qualsiasi (e quindi preso a caso) avviene in un tempo costante, indipendente dalla posizione del dato stesso nella memoria. Così, sia che il dato si trovi collocato all'inizio della RAM, nel centro o verso la fine, l'elaboratore riesce a recuperarlo nella stessa quantità di tempo. Questo tempo è veramente infinitesimo e, nei moderni elaboratori, si aggira fra i dieci e i venti nanosecondi, cioè dieci o venti miliardesimi di secondo. Questo tempo è determinato dal *ciclo operativo* della macchina e ne costituisce una caratteristica peculiare.

Questa proprietà della RAM è dovuta al fatto che ogni informazione elementare, cioè ogni byte (come vedremo a suo tempo) ha un preciso indirizzo nella memoria. Specificando tale indirizzo, che poi è semplicemente un numero, i circuiti dell'elaboratore sono in grado di collegarsi immediatamente all'informazione e a trasferirla in un *registro*, cioè in una locazione privilegiata, situata nella CPU. Tecnicamente si dice che l'elaboratore ha avuto accesso al dato. In un brutto italiano, si dice anche che la macchina ha acceduto all'informazione. Il tempo necessario, quei dieci o venti nanosecondi di cui si diceva, è chiamato il *tempo di accesso alla memoria principale*, o alla RAM. Di solito, un dato viene indicato con un nome simbolico, ad esempio $A$, e lasciamo che sia l'elaboratore stesso a scegliere l'indirizzo corrispondente ad $A$, così da poterlo recuperare quando necessario.

Una volta portato in un registro, il dato può essere elaborato. Ad esempio, se è un numero, gli può essere sommato un altro dato, oppure, semplicemente, può essere spostato ad un altro indirizzo di memoria. Come ormai si sa, l'operazione è eseguita dalla CPU, secondo un ordine stabilito dal programma che si sta eseguendo e che è contenuto nella stessa RAM. Purtroppo, sia la RAM, sia i registri usati per l'elaborazione sono costituiti da circuiti elettrici. Come abbiamo detto, appena spengiamo l'elaboratore, il loro contenuto va perduto, e dati e programmi non sono più disponibili. E' per questo che per conservare le nostre informazioni abbiamo bisogno di memorie permanenti, che mantengano dati e programmi per un periodo di tempo indefinito.

Sono queste le *memorie di massa*, apparecchiature esterne all'elaboratore e che perciò ne costituiscono delle periferiche. Per lo più, si tratta di periferiche di ingesso e uscita, poiché noi vogliamo poter *salvare*, cioè scrivere su di esse le informazioni e i programmi prima di spegnere l'elaboratore, ma vogliamo anche *caricare*, cioè leggere e portare nella memoria centrale, le stesse informazioni e gli stessi programmi quando iniziamo una nuova seduta, accendiamo di nuovo l'elaboratore e ci accingiamo a riprendere il nostro lavoro dal punto stesso in cui l'avevamo interrotto. Talvolta, si usano anche memorie di massa di sola lettura, se vogliamo che certi dati non possano essere modificati, ad esempio perché protetti da copyright.

Possiamo dire che, attualmente, esistono due tecnologie per la costruzione e l'utilizzo delle memorie di massa. La prima, e più tradizionale, è quella dei *dischi magnetici*, nei quali le informazioni sono registrate in modo magnetico. Fisicamente, si tratta di dischi, piatti, di varie dimensioni, uniformemente coperti di una sostanza magnetizzabile, di solito ossido di ferro. Le informazioni vengono registrate su circonferenze concentriche, magnetizzando o meno, pic-

colissime porzioni della superficie. Tali porzioni sono tanto piccole da poter essere assimilate a punti. L'eventuale magnetizzazione avviene mediante una *testina* che viene mantenuta a pochi micrometri dalla superficie. La stessa testina può anche rilevare lo stato di magnetizzazione dei punti, e quindi può leggere le informazioni precedentemente registrate.

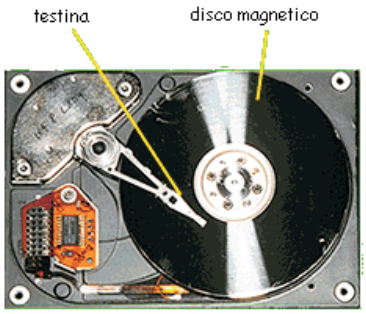



Figura 1.6: Un disco magnetico

Propriamente, il disco magnetico (si veda la Figura 1.6) è il *supporto* che conserva la informazioni e i programmi. La testina, detta di lettura e scrittura per quanto visto, fa parte dell'apparecchiatura che, nel suo insieme, collegata a una porta dell'elaboratore, è in grado di trasferire i dati dal disco alla memoria centrale, e viceversa. Questa periferica si dice *unità a disco magnetico*. Se il disco può essere inserito e tolto a piacimento dall'alloggiamento, l'unità si dice a *disco rimovibile*. Si dice invece a *disco fisso* se questo non può essere tolto dall'alloggiamento.

I dischi variano di capacità in modo molto marcato. Quelli rimovibili vanno da poco più di un milione di informazioni elementari a qualche decina o centinaia di miliardi. Quelli fissi partono da qualche miliardo per arrivare ben oltre i mille miliardi (terabyte). Le dimensioni fisiche non sono molto diverse fra i diversi modelli, se si eccettua il fatto che alcuni tipi di dischi fissi sono costituiti da vari dischi disposti uno sopra l'altro, detti *disk-pack*.

La differenza di capacità è data soprattutto dalla densità con cui le informazioni vengono registrate. La registrazione, come s'è detto, avviene su circonferenze concentriche, dette *tracce*, che possono essere più o meno vicine le une alle altre, così come succede per i punti di registrazione su ogni circonferenza. La dimensione e la vicinanza dei punti sono determinate dalla distanza tra il vertice della testina e la superficie del disco. Ovviamente, più breve è la distanza, più piccoli e più fitti sono i punti. D'altra parte, tale distanza è condizionata da fattori tecnologici: se il disco non è perfetto si rischia un contatto tra testina e superficie. Ciò danneggia il disco che viene, come si suol dire, arato, e non potrà più essere usato.

La prima volta che un disco è usato, esso viene *formattato*. Ciò significa che viene effettuata una prima registrazione di dati nulli che, però, permette di determinare fisicamente le circonferenze di registrazione, di numerarle e di suddividerle in parti più piccole, dette *settori*, che a loro volta vengono numerati. Ogni settore viene a contenere un numero fisso di informazioni elementari. Tale numero varia, a seconda degli elaboratori e del loro sistema operativo, da qualche centinaio a decine di migliaia. Ogni informazione è perciò individuata da tre parametri: il numero della traccia in cui si trova, il numero del settore all'interno della traccia e la posizione all'interno del settore. Quest'ultima quantità si dice *displacement* o, in italiano, *spiazzamento*.

Il settore viene detto talvolta anche *record fisico* e svolge la parte fondamentale nell'operatività dei dischi. In fase di lettura di dati dal disco, un intero settore viene trasferito dal disco stesso alla memoria centrale dell'elaboratore. Similmente, in fase di scrittura, è un intero settore che dalla memoria passa sul disco. Per questo, ogni unità a disco possiede, all'interno della RAM, una zona che può contenere uno o più settori. Tale zona si dice *buffer*, se contiene un unico settore, o *buffer-pool* se ne contiene più di uno. Come abbiamo visto nel caso di ingresso dei dati dalla tastiera, il buffer serve come memoria di transito, o memoria tampone, e sarà all'interno dal buffer che un programma troverà il dato di cui ha bisogno in caso di ingresso, o depositerà il dato in uscita. Tale dato è univocamente determinato dal proprio spiazzamento.

Può essere utile ed istruttivo seguire le operazioni svolte dall'elaboratore quando deve leggere un dato dal disco (vedi Figura 1.7). Le stesse operazioni, in senso contrario, sono eseguite in caso di scrittura, cioè quando il dato deve essere registrato sul disco. Se un programma ha bisogno di un dato presente sul disco, deve conoscerne l'*indirizzo* preciso, cioè il numero di traccia e di settore in cui si trova, nonché lo spiazzamento nel settore. In pratica, il programma passa questo indirizzo al Sistema Operativo, che provvede ad eseguire la lettura, mentre il programma, se questo è il caso, può procedere con altre operazioni. Il Sistema Operativo invia parte dell'indirizzo, cioè numero di traccia e di settore, all'unità a disco, che con questi dati provvede a recuperare il settore desiderato.

La testina di lettura può essere spostata radialmente al disco, in modo che il suo apice operativo si possa posizionare su una qualsiasi traccia del disco. Questo spostamento è la prima operazione svolta dall'unità a disco ed è detta *seeking*. Nel frattempo, il disco ruota velocemente sotto la testina, che può leggere il numero dei vari settori. Appena legge il numero del settore desiderato, comincia la vera e propria operazione di lettura: ogni informazione elementare, che

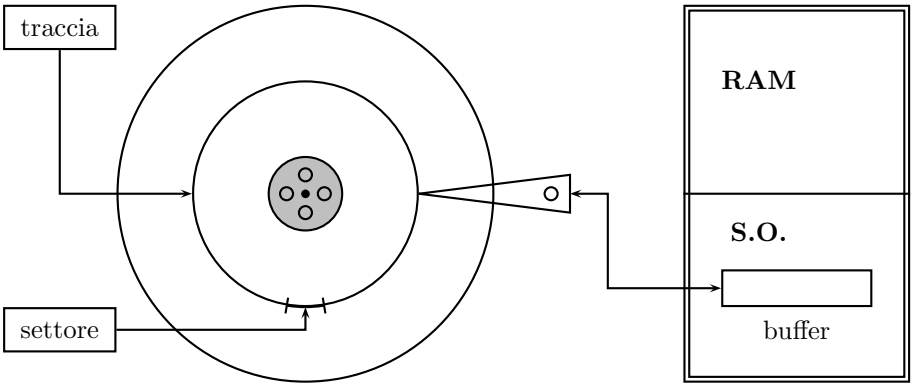



Figura 1.7: Schema di disco magnetico

passa sotto la testina, viene acquisita e trasferita nella memoria centrale, nel buffer riservato. La lettura avviene fisicamente con la rilevazione, da parte della testina, se i punti, che costituiscono l'informazione, sono magnetizzati o meno.

Una volta trasferiti nel buffer, tutti i dati del settore sono disponibili, ed il Sistema Operativo usa lo spiazzamento del dato richiesto dal programma per accedere al dato stesso. Esso viene trasferito nella zona del programma che a lui è riservato e, come ultimissima operazione, il Sistema Operativo invia un messaggio al programma per avvertirlo che il dato è al suo posto e può essere utilizzato. Come s'è detto, questo messaggio si dice tecnicamente interruzione, e il programma, così avvertito, procede ad elaborare il dato acquisito.

Come si vede, la procedura di lettura è abbastanza complicata, ma è bene che la si comprenda per poter rendersi conto della differenza tra l'accesso a un dato nella memoria principale e nella memoria secondaria. L'operazione di scrittura è analoga e presenta solo una differenza importante. Il dato da scrivere, che si trova in una zona controllata dal programma, viene trasferito nel buffer, con uno spiazzamento compatibile con gli altri dati in uscita. La scrittura vera e propria, cioè la registrazione del contenuto del buffer sul disco, non è però immediata. L'operazione viene rimandata a quando il buffer è completo, cioè tutti i dati da scrivere e che esso può contenere sono stati inseriti. Solo allora, un comando all'unità a disco fa spostare la testina sulla traccia da scrivere, la testina aspetta che il settore interessato le passi di sotto e solo allora avviene il trasferimento del contenuto del buffer nel settore.

In quanto tempo avviene la lettura o la scrittura di un settore? I tempi da prendere in considerazione sono tre. Il primo, detto *tempo di seek*, è dato dallo spostamento fisico della testina, e può variare da zero a circa 2 o 3 millisecondi, a seconda che la testina si trovi già sulla traccia da leggere o si debba spostare dal centro alla periferia, o viceversa. Il secondo tempo è quello legato alla rotazione del disco: possiamo essere fortunati e il settore coinvolto è il primo a passare sotto la testina, o, all'opposto, dobbiamo aspettare che il disco compia un giro completo. Se il disco gira a 4800 giri al minuto, un giro completo richiede circa 12 millisecondi, e mezzo giro 6 millisecondi. Questo è detto *tempo di rotazione* ed è il più rilevanti di tutti e tre.

Il terzo tempo, detto *tempo di trasferimento*, è determinato da quanto impiega il settore a passare sotto la testina, e pertanto è una frazione del tempo richiesto da un giro completo del disco. Sommando i tre tempi peggiori, si hanno circa 15 millisecondi per completare un'operazione di ingresso/uscita. Il tempo minimo, invece, è una frazione di millisecondo, quando i tempi di seek e di rotazione sono nulli: il settore richiesto è il primo a passare sotto la testina che già si trovava sulla traccia cercata. In questo caso, il tempo è solo quello di trasferimento. La media di questi valori è di 7 - 8 millisecondi, e questo si dice il *tempo medio di accesso al disco*.

Il tempo medio di accesso a un'informazione su disco è, pertanto, circa un milione di volte più lungo del tempo di accesso alla memoria centrale. Questo fa capire perché si cerchi sempre di usare la memoria centrale molto di più di quelle di massa. Per le ragioni di persistenza che abbiamo visto, all'inizio i dati di un problema saranno sempre nella memoria secondaria. Il programma li leggerà trasferendoli nella memoria centrale e qui li userà ogni volta che gli sono necessari. Nella memoria centrale verranno inseriti, di regola, anche i risultati parziali. Solo i risultati definitivi verranno infine scritti sulla memoria secondaria, proprio per poterli conservare per il tempo che saranno utili, due giorni, un mese o anche molti anni.

Spesso, come nel caso di una grande azienda pubblica o privata, i dati che interessano gli utenti sono decine o centinaia di miliardi e mai possono risiedere tutti assieme nella memoria centrale dell'elaborato-

re. Essi costituiscono una così detta *base di dati* e, di volta in volta, solo una piccola parte di essi serve ad un utente per realizzare una precisa funzionalità. Si pensi, ad esempio, a quando l'elaboratore prepara gli stipendi per il personale e, fatto il primo, ha bisogno dei dati del secondo dipendente, e così via di seguito. In questi casi, solo pochi dati per volta sono utilizzati. Tali dati possono essere modificati, ad esempio registrando che per quel mese il conteggio relativo a quel dipendente è già stato effettuato. Devono perciò essere scritti di nuovo nella memoria secondaria, e poi ignorati.

L'elaboratore, quindi, è costretto ad eseguire continue operazioni di lettura e scrittura dalla memoria secondaria e, semmai, un dato appena registrato, può essere di nuovo richiesto dopo pochi secondi. Per ottimizzare questa continua tiritera, invece di un solo buffer, si usa, come s'è detto, un buffer-pool, cioè una zona di memoria che contiene molti settori del disco.

Man mano che gli utenti richiedono informazioni, i settori interessati vengono trasferiti nel buffer-pool. L'elaboratore è stato istruito in modo intelligente e il Sistema Operativo controlla, prima di ogni lettura, se il settore desiderato è già nel buffer-pool. In tal caso, evita la lettura fisica del settore e si limita a una così detta *lettura logica* o *virtuale*, che consiste nel prendere direttamente i dati dal buffer opportuno.

Le modifiche e le integrazioni dei dati andrebbero registrate nella memoria secondaria con opportune operazioni di scrittura. Qui l'elaboratore agisce con furbizia e, in realtà, apporta le modifiche e le aggiunte alla copia dei settori coinvolti dopo averli portati nel buffer-pool, se già non c'erano. I nuovi dati sono così a disposizione per i programmi che li richiedano, e non sono ancora costati la scrittura fisica sulla memoria secondaria. Si parla, anche in questo caso, di *scrittura logica*, o *virtuale*. Naturalmente, prima o poi, questi settori, nuovi o modificati, andranno scritti fisicamente, ma, siccome c'è un costo temporale, l'elaboratore applica il principio che a pagare e a morire c'è sempre tempo.

In effetti, ogni settore presente nel buffer-pool ha un proprio *indicatore*, o *flag* come si dice in inglese. Inizialmente, quando cioè il settore è trasferito nella memoria centrale, tale indicatore vale zero, ma se il contenuto del settore viene appena modificato, l'indicatore è messo al valore 1, e tale rimarrà fino alla riscrittura sul disco.

Un altro accorgimento usato dall'elaboratore è quello di mantenere i settori del buffer-pool in un preciso ordine. Ogni volta che il settore viene usato, cioè viene letto o modificato, esso viene portato in testa al buffer-pool, sia che venga trasferito dalla memoria secondaria, sia che si trovasse già nel buffer-pool. In tal modo, i settore nel buffer-pool assumono il seguente ordine: in testa quelli usati più recentemente, in coda quelli usati da più lungo tempo. Cosa succede quando un programma ha bisogno di un nuovo settore e il buffer-pool risulta completamente pieno? Uno dei settori presenti va eliminato, lo spazio che occupa diverrà così libero e in esso si potrà trasferire il nuovo settore letto. L'esperienza ha dimostrato che il settore da eliminare dal buffer-pool è quello meno recentemente usato, cioè il settore che si trova in ultima posizione rispetto all'ordine descritto.

Attenzione, però. Il settore può essere eliminato senza danno se e solo se il suo indicatore è zero. Questo infatti significa che esso non è stato mai modificato, e quindi è tale e quale all'originale che si trova sul disco. Se però l'indicatore è uno, la copia nel buffer-pool ha subito almeno una modifica, o anche cento modifiche, e non corrisponde più all'originale sul disco. Questo, perciò, non è più un settore valido, visto che contiene dati vecchi e superati. L'elaboratore, prima di cancellare il settore dal buffer-pool, esegue finalmente la sua scrittura fisica. In tal modo rende permanenti le modifiche apportate e, quando servirà di nuovo, il settore conterrà informazioni perfettamente valide e aggiornate.

Questa tecnica si basa sul principio che, se un dato non serve da tanto tempo, allora è probabile che non servirà ancora a lungo. Essa si dice LRU, cioè del *least recently used sector*, o, in italiano, del *settore usato meno recentemente*. Con poche varianti, è la tecnica usata da tutti gli elaboratori.

Tutto quello che abbiamo detto si applica ad ogni tipo di disco magnetico. Oggi esistono due tipi principali: dischi rimovibili e dischi fissi. Fra i primi, i *floppy disk*, contengono meno di un milione e mezzo di informazioni elementari e servono soprattutto per il trasferimento dei dati fra elaboratori non collegati alla stessa rete. Le *pennette* arrivano a qualche miliardo di byte e si collegano all'elaboratore attraverso una delle porta disponibili. Fra i dischi fissi, il *disco rigido*, o *hard disk*, è la memoria di massa fondamentale di ogni elaboratore, che ne possiede sempre almeno una. Tale disco può contenere diverse centinaia di miliardi di informazioni nella versione per personal, ma arriva a superare i mille miliardi[3] per i grandi elaboratori. Contiene il Sistema Operativo, i programmi e i dati dell'utente. Un qualsiasi inconveniente al disco fisso può essere deleterio per l'elaboratore.

I dischi dei grossi elaboratori, detti disk-pack, hanno di regola diversi piatti e diverse testine, che possono leggere contemporaneamente tutte le tracce parallele, cioè le tracce che costituiscono un *cilindro*. A parte questo, sono del tutto analoghi ai dischi fissi dei personal.

[3] Mille miliardi di byte si dicono "un terabyte". Per questa terminologia si rimanda alla Tabella 2.4

Sono questi gli sviluppi attuali di una tecnologia che risale ai primordi degli elaboratori. I primi, infatti, avevano come memoria principale prima, e memoria secondaria in seguito, un tamburo magnetico, la cui forma si può immaginare. Questa apparecchiatura conteneva al più qualche centinaia di migliaia di informazioni. I primi veri dischi, più grandi di quelli attuali, potevano arrivare a contenere un paio di milioni di informazioni. Analoghi erano i dischi per i primi personal che all'inizio, per la verità, avevano solo un disco rimovibile da 128 kbyte di informazioni. Erano i famosi floppy da 5 pollici e mezzo, poi sostituiti dai tre pollici, più piccoli e affidabili; oggi si preferiscono ormai le pennette, assai più capienti e maneggevoli.

Nel frattempo, ai dischi magnetici si sono affiancati i *dischi ottici*. Prima i CD, cioè *compact disk*, e poi i DVD, ovvero *digital video disk*. Questi dischi sono, fondamentalmente, unità di sola lettura e possono essere registrati una sola volta. Solo con qualche accorgimento possono essere scritti più volte. La tecnica che sta dietro ai CD e ai DVD è legata al laser, e perciò questi dischi si dicono *ottici*.

Il laser è in grado di fornire una luce coerente, che cioè si propaga in linea retta e non si disperde nelle tre dimensioni dello spazio, come fa la luce di una lampadina. Come si è visto, il limite tecnologico dei dischi magnetici è legato alla distanza fra testina di lettura/scrittura e superficie del disco. Per quanto puntiforme sia la testina, la magnetizzazione opera su un cono e quindi un "punto" è tanto più grande quanto maggiore è la distanza. Questo problema non si presenta con il laser, che non ha dispersione spaziale, così che la registrazione e la lettura possono avvenire su punti più piccoli, ovvero con una maggiore densità. A parità di superficie, un disco ottico contiene molte più informazioni di un disco magnetico.

Il laser viene usato in fase di registrazione per modificare la capacità di riflessione di un punto della superficie del disco e in lettura per valutare tale capacità. Questa, pertanto, prende il posto dei due stati di magnetizzazione per i dischi magnetici (vedi Figura 2.4). Un lettore di CD o di DVD è oggi presente su ogni personal. Spesso, si trova anche un masterizzatore, cioè l'apparecchiatura che serve a scrivere, o registrare, uno di questi supporti. Per ora, però, la tecnologia non è in grado di usare CD e DVD come periferiche di ingresso e uscita di tipo generale, in quanto, una volta registrate, le informazioni non possono più essere modificate, o lo possono essere solo con opportuni accorgimenti non sempre attuabili.

# Capitolo 2

# DATI E INFORMAZIONI

## 2.1 Dati numerici

Spesso, e lo faremo anche noi, si tende ad assumere che i termini "dato" e "informazione" siano sinonimi. Questo è falso ed è bene chiarire la differenza tra i due, di modo che, chiarita questa, per abuso di linguaggio si usi l'uno al posto dell'altro, o viceversa.

L'esempio più semplice è costituito dai numeri. Un numero come 12, avulso da qualsiasi contesto, è senz'altro un dato, ma non può essere considerato un'informazione. Se raccogliamo per strada un biglietto con su scritto 12, sarà difficile che possiamo capire il significato di tale numero. Infatti, un'informazione è un dato connesso a un preciso significato, senza il quale il dato rimane tale. Se chiediamo quanti sono i mesi dell'anno, allora la risposta 12 assume il ruolo di vera e propria informazione.

Basta riflettere un attimo per rendersi conto che il dato 12 può riferirsi a infiniti significati: dal risultato del prodotto $3 \times 4$ al numero degli apostoli, dai segni zodiacali alle note in un'ottava. Un caso simile si ha con la parola FRANCO. Ci stiamo riferendo alla moneta di uno stato confinante, all'attore Franchi o al nostro amico Luigi che dice sempre pane al pane e vino al vino? Il concetto di informazione ha una componente semantica che manca al concetto di dato. Questo ha una pura formulazione sintattica, come numero, cioè sequenza di cifre, come parola, ovvero sequenza di caratteri, o come qualche altra rappresentazione, che non possiamo o non vogliamo legare a uno specifico significato.

L'aggettivo "incomprensibile" ci può far capire meglio quanto abbiamo affermato. Ogni cosa che ci si presenta, in quanto tale, è un dato. Quando di qualcosa diciamo che non ci è comprensibile, stiamo semplicemente affermando che, per noi, quel qualcosa non riesce a portarsi a livello di informazione. Rimane, in altre parole, un oggetto o un avvenimento a cui non possiamo attribuire un significato. Tutto è un dato, ma solo alcune cose sono informazione.

Da questo punto di vista, il concetto di informazione è legato alla nostra natura umana e una macchina non può trattare informazioni, ma soltanto dati. In particolare, un elaboratore non elabora informazioni, e molte delle cose dette nella precedente sezione sono formalmente errate. Questo modo di vedere le cose, sostanzialmente corretto, è però un po' troppo restrittivo e va relativamente addolcito.

Per il momento possiamo essere tutti d'accordo nel negare che una macchina possa dare un significato agli oggetti con cui lavora: nello specifico, che un elaboratore possa dare un significato ai numeri e alle parole che elabora. Tuttavia, spesso può rispondere alla domanda "Quanti sono i mesi dell'anno?" e dare la risposta corretta. Per far ciò, occorre che fra i dati a sua disposizione ci sia una connessione tra il dato "mesi dell'anno" e il dato 12.

Ma questo è quasi esattamente quanto avviene nella mente dell'uomo. Basta pensare alla domanda "Quanti erano i mesi nell'antico calendario romano?" Abbiamo chiaro che l'antico calendario romano aveva un certo numero di mesi e conosciamo il numero 10, ma se fra questi due dati non abbiamo un collegamento, non siamo in grado di rispondere. Potremo dire, con un po' di buona volontà, che un dato diviene un'informazione se è collegato ad altri dati, così che l'informazione risulta stabilita proprio da questa connessione. Molti elaboratori sanno, o saprebbero, rispondere alle domande "Qual è la moneta della Svizzera?" o "Come si chiamava l'attore Franchi?" Qualche elaboratore saprebbe perfino rispondere alla domanda "Come si definisce una persona che dice pane al pane e vino al vino?" In questo senso possiamo dire che un elaboratore elabora informazioni.

Nonostante questo, siamo dell'idea che, fondamentalmente, un elaboratore elabora dati, e solo a un certo livello, e con un po' di buona volontà, elabora informazioni. D'altra parte, l'Informatica si occupa di elaboratori a qualsiasi livello e quindi il suo nome, che significa "Elaborazione automatica delle informazioni" è adeguato. Comunque, anche solo per elaborare dati, occorre che tali dati siano in qualche modo definiti. E siccome i dati, come s'è detto, hanno una valenza sintattica, la loro definizione deve essere formale. Tale definizione, come ora vedremo, dipende dal tipo di dato e, fortunatamente, ogni tipo

di dato può essere ricondotto a una forma standard, composta di due soli simboli.

I due simboli che tradizionalmente si considerano sono 0 ed 1. Anche se hanno la forma delle cifre che ben conosciamo, non vogliono avere quel significato: tuttavia, come vedremo e per complicare le cose, possono assumere anche quello. L'uso risale a Leibniz che, nella sua *Characteristica Universalis*, aveva usato 0 per rappresentare il Nulla, 1 per rappresentare Dio. Una sequenza di 0 ed 1 può essere associata a qualsiasi concetto, ottenuto, secondo lui, da un'opportuna combinazione di Essere (cioè, Dio) e Nulla. Oggi, la componente metafisica di 0 ed 1 è quasi completamente scomparsa e i due simboli sono detti *bit*. Essi sono i due dati elementari con i quali si può rappresentare, come si vedrà, ogni altro dato.

La parola bit, benché esista in inglese, nel senso appena detto è artificiale e viene dalla fusione dei termini *BInary digIT*, cioè *cifra binaria*. Questo è giustificato dal fatto che i numeri, nella loro rappresentazione binaria, possono essere rappresentati con queste sole cifre. Tuttavia, il concetto di bit trascende il caso numerico e vedremo più avanti che i bit ci permettono di rappresentare *dati alfabetici*, cioè caratteri, parole e testi; *dati grafici*, cioè linee, curve e colori; e *dati sonori*, cioè suoni, voce e musica. Per questo, il bit è l'unità di informazione elementare, accettata universalmente.

Certamente, i dati più semplici da rappresentare sono i numeri, o meglio i numeri naturali, quelli con cui contiamo: 0, 1, 2, 3, e così via. Lo *zero* è un numero naturale a tutti gli effetti, anche se qualcuno ama sostenere che non serve per contare. In realtà, se in una scatola ci sono quattro caramelle, me ne posso accertare contandole. Ma se nella scatola non c'è nessuna caramella, quante caramelle ci sono? Le ho contate o non le ho contate? E se non le ho contate, come faccio a dire che non ce n'è?

Ma bando a queste insulsaggini e consideriamo la nostra consueta *rappresentazione arabica* in base 10. Essa si dice una *notazione posizionale* e dobbiamo risalire alle nozioni imparate in prima elementare per capire bene. Nella numerazione romana 33 si scrive XXXIII. Ogni X ha valore 10 e ogni I ha valore 1, qualunque sia la posizione che occupa. Nella nostra notazione, il primo 3 vuol dire 30 e il secondo vuol dire soltanto 3. Quindi, ogni cifra assume un significato diverso a seconda della posizione che occupa nel numero. Di qui la specifica di rappresentazione posizionale.

Come abbiamo imparato, le cifre si considerano da destra verso sinistra. La prima si dice cifra delle unità, la seconda delle decine, la terza delle centinaia, poi delle migliaia e così via. Questo modo discorsivo di mettere le cose può essere reso più scientifico introducendo le potenze di 10, cioè della base che determina la rappresentazione. Ricordando che $10^0 = 1$ e $10^1 = 10$, le unità e le decine hanno un'ovvia rappresentazione. Le centinaia corrispondono a $10^2$, le migliaia a $10^3$, e così di seguito:

$$8966 = 8 \times 10^3 + 9 \times 10^2 + 6 \times 10^1 + 6 \times 10^0 =$$

$$= 8 \times 1000 + 9 \times 100 + 6 \times 10 + 6 \times 1.$$

La base 10 è del tutto arbitraria. Secondo gli antropologi deriva dal fatto che abbiamo 10 dita e, per lo stesso motivo, sono note altre basi. In Francia, doveva essere usata la base 20, contando le dita delle mani e dei piedi. Ciò spiegherebbe il nome francese di 80 e 90. I romani forse avevano inizialmente una base 5, dato che V rappresenta una mano aperta ed X le due mani contrapposte. I Babilonesi usavano una misteriosa base 60. In teoria, però, ogni numero può essere usato come *base* per una numerazione posizionale.

Ad esempio, consideriamo la base 7. Per rappresentare i numeri in tale base abbiamo bisogno di sette cifre e possiamo senz'altro considerare le solite cifre arabiche dallo 0 al sei. Il significato del numero 2305 è allora immediato e basta fare i conti richiesti per capire a quale numero decimale esso corrisponde, cioè di che numero si tratta nella nostra rappresentazione:

$$2305_7 = 2 \times 7^3 + 3 \times 7^2 + 0 \times 7^1 + 5 \times 7^0 =$$

$$= 686 + 147 + 0 + 5 = 838_{10}.$$

Il risultato è perciò 838, e quando si voglia render chiaro in quale base un numero è rappresentato, tale base viene scritta come indice del numero stesso.

La stessa cosa può esser fatta con la base 5, e qui il numero 3112 sta a significare il nostro numero 407:

$$3112_5 = 3 \times 5^3 + 1 \times 5^2 + 1 \times 5^1 + 2 \times 5^0 =$$

$$= 375 + 25 + 5 + 2 = 407_{10}.$$

Convenzionalmente, se la base è maggiore di 10, oltre alle 10 cifre decimali, si usano le lettere $A, B, C, D$, eccetera. Ad esempio, in base 12 si ha $A = 10$ e $B = 11$, che sono sufficienti. Così, il numero $A1B$ è il nostro 1463, e anche qui due calcoli bastano a eseguire la trasformazione:

$$A1B_{12} = 10 \times 12^2 + 1 \times 12 + 11 = 1463_{10}.$$

In Informatica, si incontra abbastanza spesso la base 16; dopo le dieci cifre decimali si usano le lettere dalla $A$ alla $F$ con il valore corrispondente:

$$A = 10, B = 11, C = 12, D = 13, E = 14, F = 15.$$

A questo punto si capisce benissimo cosa succede se vogliamo interessarci della base 2. Prima di tutto,

sono sufficienti due cifre e, con la solita convenzione, possiamo prendere 0 ed 1. Si ritrovano i due bit ormai familiari, e questa volta il loro significato è proprio quello di zero ed uno, anche se per il momento al semplice livello di cifre. Un *numero binario* è una sequenza di bit, di solito con la restrizione che il primo a sinistra sia 1. Questa è una pura convenzione e, come nel caso decimale, le cifre 0 a sinistra della prima cifra significativa (cioè, diversa da zero) non contano niente. Nell'elaboratore, capita spesso di considerare numeri binari con una sequenza di zero in testa. Detto questo, il significato di un numero binario discende dalla regola generale e, ad esempio, 1110 vale semplicemente 14:

$$1110 = 1 \times 2^3 + 1 \times 2^2 + 1 \times 2 + 0 = 8 + 4 + 2 = 14.$$

Sarebbe bene imparare a memoria i primi numeri binari, diciamo da 0 fino a 16:

| | | | |
|---|---|---|---|
| 0 | 0000 | 8 | 1000 |
| 1 | 0001 | 9 | 1001 |
| 2 | 0010 | 10 | 1010 |
| 3 | 0011 | 11 | 1011 |
| 4 | 0100 | 12 | 1100 |
| 5 | 0101 | 13 | 1101 |
| 6 | 0110 | 14 | 1110 |
| 7 | 0111 | 15 | 1111 |

Naturalmente, qualche esercizio di conversione è molto utile. Un numero grosso, diciamo di 11 cifre binarie, in base 10 diviene molto più corto:

$$10100010011 = 1\times2^{10}+0\times2^9+1\times2^8+0\times2^7+0\times2^6+$$

$$+0\times2^5+1\times2^4+0\times2^3+0\times2^2+1\times2+1 = 1299_{10}.$$

Questo della lunghezza è l'unico inconveniente dei numeri binari, che risultano circa tre volte più lunghi dei corrispondenti decimali. Infatti, in base 10, con $k$ cifre riusciamo a scrivere tutti i numeri da 0 a $n = 10^k - 1$. Da questo si ricava:

$$k = \log_{10}(n+1).$$

Analogamente, con $h$ cifre binarie si rappresentano i numeri fino a $n = 2^h - 1$, e quindi $h = \log_2(n + 1)$. Con un po' di Algebra, dalla precedente relazione $10^k = n+1$, prendendo i logaritmi in base 2, si ricava:

$$k \log_2 10 = \log_2(n+1) \quad \text{o} \quad k = \frac{\log_2(n+1)}{\log_2 10};$$

allora, il rapporto fra $h$ e $k$ vale:

$$\frac{h}{k} = \frac{\log_2(n+1)}{\log_{10}(n+1)} = \log_2 10 \approx 3.32.$$

A questo punto, se abbiamo un numero scritto in una qualsiasi base, la regola posizionale ci permette di trovarne l'equivalente in base 10, cioè capire qual è il suo significato. Come ai primi tempi dell'Euro, "capire" vuol dire riportare una cifra al nostro più comune modo di pensare. Osservare che 100 Euro sono circa 193 mila lire non ci dice nulla di nuovo, poiché i due valori coincidono, ma ci aiuta a capire l'entità della cifra. Così, dire che 10100 in base 2 è 20 in base 10, ci spiega cosa significa quella sequenza di 0 ed 1.

Naturalmente, in modo immediato, si pone il problema inverso. Se partiamo da uno dei nostri numeri, ad esempio 2004, come se ne può trovare la rappresentazione in un'altra base, diciamo 7 oppure 2? La cosa non è per niente difficile. Il metodo da adottare si dice delle *divisioni successive*, e lo possiamo descrivere per esprimere 2004 in base 7. Impostando la divisione di 2004 per 7, si trova facilmente che il quoziente è 286 e il resto è 2:

| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 0 | 0 | 4 | 7 | | |
| | 6 | 0 | | 2 | 8 | 6 |
| | | 4 | 4 | | | |
| | | | 2 | | | |

Questo resto è particolarmente importante e ricordiamo che, per la definizione della divisione, esso è un numero minore del divisore. In questo caso è un numero compreso tra 0 e sei, cioè una cifra dei numeri scritti in base 7. Continuiamo eseguendo la divisione del quoziente 286 per 7. Si trova 40 come quoziente e 6 come resto. Ancora, 40 diviso 7 dà 5 con resto 5, Infine, dividendo l'ultimo quoziente 5 per 7, si trova 0 con resto 5. Il quoziente 0 è il segnale che il procedimento è finito e possiamo riassumerlo così:

| | | | | | | |
|---|---|---|---|---|---|---|
| 2004 | : | 7 | = | 286 | resto | 2 |
| 286 | : | 7 | = | 40 | resto | 6 |
| 40 | : | 7 | = | 5 | resto | 5 |
| 5 | : | 7 | = | 0 | resto | 5 |

I resti, letti dal basso verso l'alto, formano il numero 5562, che contiene solo cifre relative alla base 7, per quanto abbiamo ricordato. Questo numero rappresenta il nostro 2004 nella base 7. Non possiamo dimostrare la validità del metodo delle divisioni successive, anche se la prova è molto semplice. Limitiamoci a fare la riprova che il risultato è corretto. Se applichiamo la regola posizionale a 5562, e tutto ciò che abbiamo fatto è giusto, dobbiamo ritrovare proprio 2004. Si tratta solo di impostare le operazione ed eseguirle:

$$5562_7 = 5 \times 7^3 + 5 \times 7^2 + 6 \times 7 + 2 =$$

$$= 1715 + 245 + 42 + 2 = 2004.$$

Nel caso della base 2, possiamo semplificare un po' il procedimento delle divisioni successive. Quando

dividiamo un numero $n$ per 2, la divisione è esatta e il resto è zero se $n$ è pari; se invece è dispari, il resto sarà 1. Questo ci dice che il risultato sarà formato di cifre 0 ed 1, come deve essere, e che comunque ogni resto può essere calcolato ad occhio. Parimenti ad occhio possiamo immaginare di essere in grado di eseguire la divisione per 2. Pertanto, la conversione in binario del numero 43 si imposta in questa maniera:

| | |
|---|---|
| 43 | 1 |
| 21 | 1 |
| 10 | 0 |
| 5 | 1 |
| 2 | 0 |
| 1 | 1 |
| 0 | |

Il risultato si legge dal basso verso l'alto.

Anche in questo caso ci dobbiamo fidare del fatto che il procedimento descritto è corretto. Ancora, possiamo eseguire la riprova che consiste di nuovo nel convertire il numero binario trovato nel corrispondente decimale. Questo deve coincidere con il numero di partenza. La conversione dei numeri binari si fa facilmente se mettiamo un 1 sotto la cifra binaria più a destra; quindi, procedendo verso sinistra, ogni volta raddoppiamo. Alla fine, è sufficiente sommare i numeri che si trovano sotto i bit 1 per ottenere il risultato della conversione:

| | | | | | |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 1 |
| 32 | 16 | 8 | 4 | 2 | 1 |

Nel nostro esempio si ottiene $32 + 8 + 2 + 1 = 43$, il numero di partenza, ma il metodo funziona in generale e un po' d'esperienza insegna a raddoppiare ad occhio i numeri, che in definitiva sono sempre gli stessi.

A questo punto, siamo in grado di trattare i numeri a nostro piacimento, passando senza difficoltà da una base all'altra. Naturalmente, se abbiamo un numero in base 5 e vogliamo portarlo in base 3, la cosa più conveniente è portarlo prima in base 10 con la regola posizionale e quindi convertire il risultato in base 3 con il metodo delle divisioni successive. L'unica cosa a cui stare attenti è la conversione in una base maggiore di 10, che comporta l'uso delle cifre speciali $A, B, C$, eccetera, come abbiamo avvertito. Se vogliamo convertire 131 in base 12, la divisione ci dà 10 col resto di 11 e le due cifre del risultato sono, nell'ordine, 10 ed 11. Ricordando che $A = 10$ e $B = 11$, la conversione di 131 è il numero duodecimale $AB$.

Come abbiamo ricordato, la base 2 fu usata dai Leibniz, ma esistono testimonianze che, come codice segreto, era stata usata anche prima. Purtroppo, i lavori di Leibniz rimasero pressoché ignorati fino all'inizio del 1900, quando i logici, sulle orme di Boole, rivalutarono il pensiero di Leibniz nel loro settore. Nell'algebra booleana del calcolo delle proposizioni, 1 rappresenta il valore di verità "vero", mentre 0 rappresenta "falso", logica trasposizione di Dio e del Nulla. Tuttavia, la numerazione binaria rimase più che altro una curiosità fino alla metà degli anni 1930. Nel 1935 Shannon fece vedere come tale rappresentazione fosse conveniente per eseguire le operazioni mediante circuiti elettrici, cosa che più avanti faremo anche noi.

In realtà, la convenienza della numerazione binaria sorge da due fatti molto importanti dal punto di vista pratico. Il primo è che la rappresentazione dei numeri può facilmente avvenire sui supporti più diversi. Shannon considerò il passaggio di corrente in un circuito elettrico come valore 1, e il non passaggio come valore 0. Ciò corrispondeva ai due stati acceso/spento di una lampadina. Nei dischi, abbiamo incontrato le due coppie magnetizzazione/non magnetizzazione e riflessione/non riflessione. Ed anche bucato/non bucato nelle schede e nei nastri di carta. Vedremo ancora puntino bianco e puntino nero e ricordiamo la storica coppia linea/punto dell'alfabeto Morse. E chi più ne ha, più ne metta.

La seconda convenienza è fornita dalla semplicità con cui si eseguono le quattro operazioni elementari di addizione, sottrazione, moltiplicazione e divisione. Il sistema binario offre regole semplicissime che si possono imparare in quattro e quattr'otto. Alle scuole elementari abbiamo faticato un anno e più per apprendere le tabelline, sia quelle della somma $3+2 = 5$ e $7 + 8 = 15$, sia, soprattutto, quelle della moltiplicazione, la famigerata *Tavola Pitagorica*. Nel caso binario, come vedremo fra poco, le stesse tabelline sono banali e si imparano immediatamente. L'esecuzione delle operazioni diviene un gioco da ragazzi, alla portata immediata di tutti, o quasi.

Questa semplicità estrema si riflette immediatamente nel calcolo automatico. Pascal, e dopo di lui Leibniz, erano stati costretti ad usare ruote dentate con 10 denti per poter trattare i numeri decimali. Nel caso binario è sufficiente avere apparecchi che trattano due soli stati. Deriva da questo il successo che ebbe Shannon. Avere operazioni semplici, poi, vuol dire semplificare moltissimo la struttura interna di una macchina da calcolo, la cui complessità riflette direttamente quella delle operazioni. Tutto ciò sopravanza di gran lunga l'inconveniente di dover trattare numeri tre volte più lunghi. In effetti, è più semplice replicare molte volte una macchina semplice, che replicare poche volte una macchina complessa.

I primi calcolatori elettrici ed elettronici avevano l'inconveniente di richiedere, trattare e presentare numeri binari. Occorreva perciò prendere familiarità con la notazione binaria. Oggi, semplici programmi di conversione, che realizzano la regola posizionale

e il metodo delle divisioni successive, permettono di usare tranquillamente i numeri decimali. Anche la calcolatrice tascabile esegue le operazioni in binario, ma noi inseriamo i dati in decimale così come in decimale il display ci mostra i risultati. Le conversioni sono solo il primo e l'ultimo passo di una catena che, per il resto, si svolge completamente nel mondo molto più semplice dell'aritmetica binaria.

## 2.2 Le operazioni

Una volta che abbiamo imparato come si rappresentano i numeri in binario, è bene che ci rendiamo conto di come ciò semplifichi le operazioni da eseguire. Passiamo in rassegna tutte e quattro le operazioni elementari e il confronto tra numeri, che costituiscono la base di tutte le elaborazioni numeriche. Più avanti, per completare quanto detto, accenneremo a come si trattano i numeri negativi e i numeri con la virgola, che richiedono qualche accorgimento particolare, anche se di minima entità. Infine, vedremo come questi numeri si rappresentano nelle memorie dell'elaboratore, sia nella memoria centrale, sia in quelle periferiche. Questa rappresentazione, pur fatta da cifre 0 ed 1, deve tener conto della struttura fisica delle memorie attuali.

Le regole per effettuare la somma tra due numeri decimali e che abbiamo imparato all'inizio delle scuole elementari, sono del tutto generali e, come metodo, valgono indifferentemente per ogni base della notazione posizionale. Naturalmente, $5 + 4$, che fa nove in base 10, non può fare nove in base 7, dove la cifra 9 nemmeno esiste. D'altra parte, il nostro numero 9 in base 7 si scrive 12 e quindi è naturale che poniamo $5 + 4 = 12$ nella base 7. Questa considerazione vale per ogni base e, ad esempio, per la base 5 si ha una semplice tabellina che riassume il valore della somma di due cifre qualsiasi. Essa corrisponde alle regole che impariamo alla scuola per la base 10 ed è sufficiente per eseguire qualunque somma:

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 | 4 |
| 1 | 1 | 2 | 3 | 4 | 10 |
| 2 | 2 | 3 | 4 | 10 | 11 |
| 3 | 3 | 4 | 10 | 11 | 12 |
| 4 | 4 | 10 | 11 | 12 | 13 |

Facciamo un esempio, proprio usando la base 5. Prendiamo due numeri come 1194 e 283, convertiamoli alla base 5 e facciamo la somma. Poiché tale somma, in decimale, vale 1477, alla fine potremo verificare di aver fatto tutto bene convertendo il risultato ottenuto. Col metodo delle divisioni successive si trova $1194_{10} = 14{\cdot}234_5$ e $283_{10} = 2{\cdot}113$. Il primo vero passo è quello di incolonnare i due numeri, come si vede nella Tabella 2.1. Analogamente al caso decimale, l'incolonnamento parte da destra e quando i numeri, come in questa somma, non hanno la stessa lunghezza, la differenza si vede sulla sinistra. Il segno più e l'uguale, la linea di separazione del risultato, fanno parte del folklore, ma aiutano anche ad impostare il calcolo.

| | | | | | |
|---|---|---|---|---|---|
| | | 1 | 1 | | |
| 1 | 4 | 2 | 3 | 4 | + |
| | 2 | 1 | 1 | 3 | = |
| 2 | 1 | 4 | 0 | 2 | |

Tabella 2.1: Somma in base 5

La tabellina dimostra ora tutta la sua utilità. La somma, come al solito, parte da destra e dobbiamo calcolare 4+3. La tabella ci dice 12 (che è il nostro 7) e l'interpretazione è quella che ben conosciamo. Dodici non è il risultato, ma dobbiamo scrivere il 2 e riportare l'1. Il concetto di riporto è di fondamentale importanza e va capito bene. Nella notazione decimale, esso indica che abbiamo superato la decina e che quindi di questo fatto dobbiamo tener conto nella somma delle due cifre successive. Nel caso presente abbiamo superato, per così dire, la cinquina, ma l'effetto è lo stesso: il riporto va a modificare la somma delle due cifre a sinistra. Come pro-memoria, specie quando si fanno le prime addizioni, il riporto si scrive in alto, sopra le nuove cifre da sommare.

A questo punto dobbiamo eseguire $3 + 1$, che fa 4 e a questo dobbiamo aggiungere il riporto. Si ha pertanto $4 + 1 = 10$, che significa scrivi lo 0 e riporta l'1. Di nuovo, scriviamo questo riporto sulle cifre a sinistra e andiamo avanti. Dobbiamo eseguire $2 + 1$ e questo fa tre. Aggiungendo il riporto si trova 4, il che significa: scrivi il 4 e non c'è riporto. Abbiamo ora $4 + 2 = 11$, così che si scrive 1 e si riporta 1. Siamo arrivati alla fine. Dobbiamo sommare 1 con niente e si ottiene 1. Aggiungendo il riporto si ha 2 e questo conclude la nostra operazione. Non rimane che convertire il risultato ottenuto. I calcoli sono abbastanza semplici e si ottiene proprio 1477, il che ci dice che tutto va bene:

$$21402_5 = 2 \times 5^4 + 1 \times 5^3 + 4 \times 5^2 + 0 \times 5 + 2 =$$

$$= 1250 + 125 + 100 + 0 + 2 = 1477_{10}.$$

Nel caso della base 2, le cifre sono solo due e le regole quattro in tutto. Esse si ricordano facilmente, dato che sono del tutto naturali: 0+0 fa zero, 0+1 e

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1 | | 1 | 1 | | | |
| | 1 | 0 | 1 | 1 | 0 | 1 | + |
| | 1 | 0 | 1 | 1 | 1 | 0 | = |
| 1 | 0 | 1 | 1 | 0 | 1 | 1 | |

Tabella 2.2: Somma in base 2

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | 10 | 1 | 10 | | | | |
| 1 | 1 | 0 | 0 | 1 | − | | |
| | 1 | 0 | 1 | 0 | = | | |
| | 1 | 1 | 1 | 1 | | | |

Tabella 2.3: Sottrazione in base 2

1+0 fanno 1, 1+1 fa 0 col riporto di 1:

| | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 10 |

Eseguire un'addizione in binario è allora un'operazione elementare e possiamo vedere un esempio per ribadire i vari concetti. Si voglia valutare $45 + 46$. La conversione in binario ci dà i numeri che devono essere incolonnati: $45_{10} = 101101$ e $46_{10} = 101110$ (si veda Tabella 2.2).

Iniziando da destra, 0+1 e 1+0 danno 1. Poi c'è 1+1 che dà zero con riporto di 1. Ora c'è il passo più difficile, ma 1+1 è ancora 0 con riporto di 1, e il riporto precedente ci dà perciò 1 con il riporto di 1. Manca solo lo 0+0 che dà zero, ma abbiamo il riporto precedente. Infine, 1+1 dà zero col riporto di 1. Questo riporto, come nel caso decimale, va a disporsi come prima cifra del risultato. Abbiamo così il valore finale della somma. La riprova non può mancare e, con la solita regola dei doppi, troviamo che il risultato vale $64 + 16 + 8 + 2 + 1 = 91$, proprio la somma di 45 e 46.

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 64 | 32 | 16 | 8 | 4 | 2 | 1 |

La somma in binario è tutta qui. Le regole legate alla somma delle cifre si sono ridotte drasticamente. In prima elementare ne abbiamo imparate cento, anche se presto ci siamo accorti della proprietà commutativa, scoprendo che 6+7 è uguale a 7+6. Nel caso binario le regole sono solo 3, e questo è un bel po' di risparmio. Ad essere pignoli, considerando il riporto, l'unica difficoltà è data da 1+1, quando abbiamo 1 come riporto dalla somma delle cifre precedenti. Basta però ricordare che il risultato è 1 col riporto di 1.

Come il riporto caratterizza l'addizione, così il prestito costituisce il nucleo della sottrazione. E' facile sottrarre 0 da 0 e da 1, e 1 da 1. Più complicato è sottrarre 1 da 0. Questo non si può fare e allora occorre prendere in prestito una unità da una posizione più a sinistra e trasformare lo 0 in 10, così che $10 - 1$ dà 1. Bisogna ricordare come l'1 che ha offerto il prestito diviene 0, ed ognuno degli 0 intermedi diviene un 1. Un esempio vale più di tanti discorsi. Valutiamo $25 - 10$, come esposto nella Tabella 2.3. Incolonnati i numeri, si comincia al solito da destra. Il primo caso, $1 - 0$, è facile. Ora però dobbiamo fare $0 - 1$. Si prende il prestito dal primo 1 a sinistra che si incontra. I cambiamenti si scrivono il alto per ricordarli meglio. Ora, $10 - 1$ fa 1, ed $1 - 0$ dà 1. Rimane un altro $0 - 1$, ma col prestito possibile si trova ancora 1, così che il risultato finale è 15, come deve essere.

Naturalmente, il problema grosso viene fuori quando il sottraendo è maggiore del minuendo, perché non si può sottrarre un numero più grande da uno più piccolo. Come si è imparato alla Scuola Media, in realtà il risultato è un numero negativo e, per il momento, non abbiamo ancora considerato questo tipo di numeri. Può essere giunto il momento di fare questo passo ulteriore. Cominciamo col dire che, all'interno dell'elaboratore, i numeri sono rappresentati con una quantità fissa di cifre binarie. Questa quantità è di solito di 8, di 16 o di 32 bit, ma esistono anche rappresentazioni più estese:

| | | | | |
|---|---|---|---|---|
| 1 | byte : | 0 | ÷ | 255 |
| 2 | byte : | 0 | ÷ | 65·536 |
| 4 | byte : | 0 | ÷ | 4·294·967·296 |
| 8 | byte : | 0 | ÷ | $1.844 \times 10^{19}$ |

Per non complicare troppo i nostri esempi, ci limiteremo a considerare numeri con 8 bit. Naturalmente, se un numero è composto di 5 bit soltanto, basta premettergli tre bit uguali a zero.

La scelta degli 8 bit non è casuale. Dall'inizio degli anni 1960, una convenzione, ormai accettata in tutto il mondo, ha definito una unità di misura delle informazioni come composta da 8 bit. Per quanto visto, il bit è una specie di unità naturale, in quanto un'informazione, o un dato, non può essere espresso con meno di un bit. D'altra parte, il bit è un'unità troppo piccola, o almeno lo è diventata da quando le informazioni si contano in milioni e miliardi di bit. La nuova unità è stata detta *byte*, che linguisticamente è un bit con la "i" allungata. I suoi multipli sono indicati nella Tabella 2.4 e valgono un po' di più dei corrispondenti valori decimali.

Un bit può rappresentare due informazioni distinte: passa o non passa corrente; c'è un buco oppure non c'è, e così via. Due bit rappresentano 4 informazioni, corrispondenti alle coppie 00, 01, 10 e 11.

| | | | |
|---|---|---|---|
| kilobyte | kbyte | $2^{10}$ | 1·024 |
| megabyte | Mbyte | $2^{20}$ | 1·048·576 |
| gigabyte | Gbyte | $2^{30}$ | 1·073·741·824 |
| terabyte | Tbyte | $2^{40}$ | 1·099·511·627·776 |

Tabella 2.4: Multipli bel byte

Tre bit rappresentano 8 informazioni, che si ottengono aggiungendo a ciascuna delle precedenti coppie o lo 0 o l'1. In generale, $n$ bit corrispondono a $2^n$ informazioni distinte, e in particolare un byte rappresenta $2^8$, cioè 256 informazioni. Una specifica sequenza di 8 bit si dice una *configurazione di byte* e, vista come numero binario, ha un valore corrispondente: si parte dal byte fatto tutto di cifre 0, che vale 0, a quello fatto tutto di cifre 1, che vale 255.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 53 |
| 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 18 |

Vedendo le cose all'incontrario, possiamo dire che il numero 53 corrisponde a una ben precisa configurazione di byte. Prendiamo tale byte come rappresentazione interna del numero 53 e proviamo a sottrargli 18. Il risultato deve essere ovviamente la configurazione di byte corrispondente a 35. Consideriamo ora il sottraendo 18 ed eseguiamo queste strane operazioni:

1. cambiamo ogni 0 in 1, e viceversa;
2. aggiungiamo 1:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| | | | | | | | 1 |
| 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |

Sommiamo ora il byte così ottenuto a 53:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |

e, miracolo!, il risultato è quel valore 35 che dovevamo ottenere dalla sottrazione. Questo non è, di certo, un vero miracolo, ma è la regola generale per eseguire una sottrazione, cambiandola in una somma.

Il trucco, se così si può chiamare, sta tutto nella rappresentazione dei numeri negativi. Infatti, come ricordiamo dall'Algebra, $53-18$ significa $53+(-18)$, dove $-18$ è l'opposto di 18. Per definizione, si dice *opposto* di 18 quel numero che sommato a 18 dà zero. Lo zero è rappresentato da una qualunque sequenza di cifre 0. Osserviamo allora che, data la rappresentazione di 18, se cambiamo ogni zero in uno, e viceversa, otteniamo un numero che sommato a 18 dà una sequenza tutta fatta di cifre 1. Aggiungendo ancora 1 si ottiene una sequenza tutta fatta di cifre 0, eccetto all'estrema sinistra, dove il riporto genera un 1 indesiderato. Se questo 1 non ci fosse, avremmo trovato un qualcosa che, sommato a 18, ci darebbe 0, cioè l'opposto di 18.

Se rappresentiamo i numeri con un byte, il bit 1 indesiderato occuperebbe la nona posizione a sinistra, e quindi di fatto sta al di fuori del byte. Possiamo perciò tranquillamente ignorarlo. Se siamo soddisfatti di questo, possiamo affermare che l'opposto di 18 si ottiene partendo dallo stesso 18, rappresentato in un byte, cambiando ogni 0 in 1, e viceversa, e alla fine sommando 1 con l'avvertenza di ignorare, se c'è, il riporto all'estrema sinistra. Questo è giusto ciò che avevamo fatto per calcolare $53-18$ e giustifica la correttezza del risultato. Analogamente si trova l'opposto di 39:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 39 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |
| | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| | | | | | | | | 1 |
| $-39$ | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 |

Se proviamo a calcolare $83-39$ si ottiene il risultato corretto, come si vede immediatamente:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 83 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | + |
| $-39$ | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | = |
| 44 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | |

Proviamo ora a fare $25-39$. Usando il valore di $-39$ che abbiamo già trovato, si ottiene un risultato un po' strano, che formalmente si interpreta come il numero decimale 232:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 25 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | + |
| $-39$ | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | = |
| 44 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | |

Ma noi ci aspettiamo un risultato negativo, e cioè $-14$. Vediamo come si scrive $-14$ nella strana notazione appena introdotta:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 14 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| | | | | | | | | 1 |
| $-14$ | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |

Invertendo le cifre 0 con le cifre 1 e sommando 1 al risultato, si ha proprio la configurazione ottenuta dalla sottrazione. Quindi, in definitiva, il nostro conto è corretto. Osserviamo anche che partendo da $-14$, se si invertono le cifre 0 con le cifre 1, e si aggiunge 1, si ottiene proprio 14.

Naturalmente, come sappiamo, $-(-14)$ è uguale a 14, e quindi eseguendo due volte le operazioni di

inversione e di somma di 1, si deve ritornare al numero di partenza. Questo possiamo verificarlo con un nuovo esempio, 84:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 84 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 |
| | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| | | | | | | | | 1 |
| −84 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 |
| | | | | | | | | 1 |
| 84 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |

Rimane però un problema. Quali sono i numeri negativi? Nell'esempio precedente, la configurazione del byte va interpretata come 242 oppure come −14? Qui occorre fare una convenzione: ogni configurazione che ha il bit più a sinistra uguale a 0, è un numero positivo. Quelle con il bit di sinistra uguale ad 1, sono numeri negativi. Per questo, 242 non è 242, ma è il numero negativo −14.

Rappresentando i numeri con un solo byte, il più grande numero positivo è 127, quando lo si legga in base 10.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 127 |

Il più piccolo numero negativo è quello che comincia con 1 ed è composto poi di tutti zero.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | −128 |

Facendo le solite operazioni si ottiene il suo opposto, che corrisponde a −128. Tutto ciò è implicito nella rappresentazione. Per convenzione, si dice che i numeri negativi sono scritti in *complemento a 2*. Per completezza, ricordiamo che si chiama invece *complemento a 1* di un byte (o di un numero in generale) quello che si ottiene scambiando le cifre 0 con le cifre 1. Il lettore può divertirsi a provare che, in base 10, funziona da opposto del numero $n$ il suo complemento a 10: si complementa a 9 ogni cifra e si aggiunge 1.

I numeri interi che vanno da −128 a +127, tutti quelli rappresentabili con un solo byte, sono veramente pochi, e non possono servire che a fare esempi semplici e intuitivi. E' per questo che, nell'elaboratore, si usano diversi byte per rappresentare un singolo numero intero, con la stessa tecnica del complemento a 2 per i negativi. Spesso si usano due byte, che permettono di rappresentare i numeri da $-32{\cdot}768$ fino a $32{\cdot}767$. Per certe applicazioni questi numeri sono già sufficienti, ma non bastano nemmeno per gestire la contabilità familiare, dove, inoltre, oggi occorrono decimi e centesimi. Si passa allora a quattro byte, e questo è lo standard per molti elaboratori, anche se per certe applicazioni scientifiche e finanziare occorre qualcosa di più.

In questi casi, si usano rappresentazioni con un numero maggiore di byte e, soprattutto, c'è la possibilità di avere a disposizione un'*aritmetica indefinita*. In tal caso, ad ogni numero è associata la quantità di byte che servono a rappresentarlo esattamente. Ogni numero intero ha una rappresentazione finita e quindi l'unico limite teorico è costituito dalla quantità totale di RAM che abbiamo a disposizione. Ci sono casi in cui si debbono considerare milioni di cifre, ma noi ci possiamo accontentare di farci calcolare il fattoriale di 100, il numero che si ottiene moltiplicando i primi cento numeri interi. Il risultato è abbastanza impressionante, e troveremmo qualche difficoltà se lo volessimo leggere:

93326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156518286253697920827223758251185210916864000000000000000000000000.

E' giunto il momento di fare un'importante osservazione, sulla quale è bene insistere in modo particolare. Prendiamo una qualsiasi configurazione di byte, ad esempio: 11010011. Se la interpretiamo come numero decimale assoluto, la conversione in decimale ci dice che siamo di fronte a 211. Se però vogliamo interpretare questa quantità come intero relativo, facendo il complemento a 2 si ottiene −45. Quale di queste due interpretazioni è corretta? Dobbiamo prendere atto che entrambe, in linea di principio, sono corrette, e dipende dal contesto se il byte deve essere visto come 211 o come −45. Il byte è un dato, ma bisogna sapere quali informazione esso porta per poterlo interpretare. E' un discorso che abbiamo fatto fin dall'inizio e che dobbiamo avere costantemente presente, anche al di là della convenzione linguistica che tende a confondere dato e informazione.

Questo va bene per noi umani, ma come fa l'elaboratore a capire con quale significato usare questo byte? Come capisce che deve vederlo come 211 e non come −45, o viceversa? Ebbene, l'elaboratore non farà mai una simile distinzione e tratterà il dato secondo le nostri istruzioni, secondo le nostre specifiche. Come vedremo, un byte può avere molte altre interpretazioni: come carattere, come colore, come suono. E' sempre il solito byte e l'elaboratore non ha mezzo per distinguerlo. E' però l'uomo, specificando certe operazioni o certe funzioni piuttosto che altre, a decidere che cosa è quel dato, quale informazione rappresenta. Così, se faccio stampare il nostro byte come numero assoluto, l'elaboratore scriverà 211. Se lo faccio stampare come intero relativo, scriverà −45. E così via, per tutte le altre possibili interpretazioni dello stesso byte.

Per quanto riguarda la moltiplicazione, la cosa più semplice è fare un esempio che mostra appieno la dra-

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | 1 | 0 | 1 | 1 | 0 | × |
| | | | | 1 | 0 | 1 | = |
| | | 1 | 0 | 1 | 1 | 0 | |
| 1 | 0 | 1 | 1 | 0 | = | = | |
| 1 | 1 | 0 | 1 | 1 | 1 | 0 | |

Tabella 2.5: Moltiplicazione in binario

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | |
| 1 | 0 | 1 | | | | | 1 | 0 | 1 | 0 | 1 |
| | | 1 | 1 | 0 | | | | | | |
| | | 1 | 0 | 1 | | | | | | |
| | | | | 1 | 1 | 1 | | | | |
| | | | | 1 | 0 | 1 | | | | |
| | | | | | 1 | 0 | | | | |

Tabella 2.6: Divisione in binario

stica semplicità di questa operazione. Si voglia moltiplicare 22 per 5 (si veda la Tabella 2.5). Dopo aver convertito i due numeri, il prodotto si imposta come nella base 10 e si utilizzano le cifre del moltiplicatore 5 per eseguire man mano le moltiplicazioni elementari. Ma le cifre del moltiplicatore sono solo 0 ed 1. Se la cifra è 1, il risultato è lo stesso moltiplicando, e quindi lo si trascrive nella parte riservata ai prodotti parziali. Quando la cifra del moltiplicatore è zero, anche il risultato è zero, e basta scalare di un posto. L'ultimo prodotto relativo all'1 si scrive ancora sotto, scalando opportunamente. Non rimane ora che sommare i due risultati parziali.

Anche in questo caso la riprova ci conferma la correttezza del procedimento. Siamo stati avvantaggiati dal fatto che la versione binaria di 5 contiene due soli uno. Se ne avesse contenuti quattro, avremmo avuto quattro quantità da sommare. Il calcolatore, di fatto, esegue le somme di due addendi alla volta, man mano che procede nelle moltiplicazioni parziali, e anche noi potremmo fare lo stesso.

E' bene osservare che lo stesso risultato della moltiplicazione si ottiene applicando i prodotti cifra per cifra, cioè utilizzando la Tavola Pitagorica. Questa è veramente elementare e ribadisce quanto abbiamo già detto; gli scolari che dovessero imparare la Tavola Pitagorica in base 2 sarebbero ben fortunati:

| | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

Non rimane ora che vedere cosa succede per la divisione. Essa dipende dal concetto di confronto, che è molto semplice, anche tra numeri binari. Intanto, due numeri sono uguali se e solo se hanno la medesima rappresentazione. Possiamo aggiungere, per eccesso di scrupolo, che eventuali cifre 0 all'inizio non contano. Quando due numeri non sono uguali, intanto è più grande quello più lungo, sempre a meno delle cifre 0 iniziali. Il caso più complesso si ha quando i due numeri, pur avendo la stessa lunghezza sono diversi. Si scandiscono allora da sinistra verso destra e il primo che ha un 1 dove l'altro ha uno 0, quello è il più grande. Infatti, le cifre a sinistra contano le potenze maggiori del 2: sono, come si dice, più *significative*, e incidono maggiormente sulla grandezza del numero.

La divisione si basa tutta sul fatto che il divisore stia o non stia in opportune parti del dividendo. La suddivisione in parti si fa come nel caso decimale e non presenta particolari difficoltà. Si voglia eseguire, ad esempio, 107 diviso 5 (si vede la Tabella 2.6).

Impostata l'operazione, si ha subito che 101 non sta né in 1 né in 11, cioè, è più grande. Sta invece in 110 e questi ci avverte di inserire un 1 nel quoziente e di sottrarre 101 da 110. Questo dà 1 e si abbassa l'1 successivo. Poiché 101 non sta in 11, si scrive 0 nel quoziente e si abbassa lo zero. Di nuovo, il 101 sta nel 110: si scrive 1 nel quoziente e, sottraendo, si trova il resto 1. Si abbassa l'1, ma il 101 non sta nell'11. Si mette 0 nel quoziente e si abbassa l'ultimo 1. Ora il 101 sta nel 111. Si scrive 1 nel quoziente e si ha il resto definitivo di 10. Convertendo il quoziente trovato si ottiene 21, che è proprio il quoziente tra 107 e 5. Anche il resto equivale a 2, e ciò ci permette di concludere che l'operazione è stata eseguita in modo corretto.

Osserviamo esplicitamente che abbiamo fatto una divisione a più cifre, tre per l'esattezza. Nel caso decimale, questo tipo di divisione è la più difficile e spesso si trovano persone di buona cultura che hanno difficoltà a portare avanti queste operazioni. Qui, il fatto che il divisore stia o non stia nel dividendo è di verifica ovvia. Perciò tutto si semplifica e la complessità di trovare quante volte il divisore sta nel dividendo scompare.

Dimostrata così la convenienza della notazione binaria, concludiamo questa lezione con il seguente calembour:

"Ci sono 10 tipi di persone:

- quelle che capiscono il codice binario;
- quelle che non lo capiscono."

## 2.3 Numeri e parole

I numeri naturali, cioè i numeri con cui contiamo, e i numeri interi, cioè i numeri col segno, sono solo

| | | | | |
|---|---|---|---|---|
| 0.29 | × | 7 | = | 2.03 |
| 0.03 | × | 7 | = | 0.21 |
| 0.21 | × | 7 | = | 1.47 |
| 0.47 | × | 7 | = | 3.29 |

Tabella 2.7: Conversione della parte decimale

una piccola parte dei numeri che usiamo. Oggi, con l'Euro, ci siamo di nuovo abituati ai decimi e ai centesimi, cioè ai numeri con la virgola. D'altra parte, le altezze, le distanze e i pesi, essendo quantità continue, o analogiche, richiedono quasi sempre l'uso della virgola e della parte decimale. Il nostro metodo di conversione da base a base è stato fondato sul fatto che, dividendo dividendo, a un certo punto si ottiene un quoziente nullo e un resto. Ma quando si passa ai numeri con la virgola, questo non è più vero e una divisione si può continuare all'infinito. Come si fa allora la trasformazione?

Si prenda ad esempio il numero 38.29 in base 10 e vediamo qual è il suo significato nella notazione posizionale. Le cifre a sinistra della virgola continuano ad indicare unità, decine, centinaia e le altre potenze del 10. A destra della virgola si hanno decimi, centesimi, millesimi, cioè frazioni di potenze di dieci, ovvero potenze negative. Come si ricorderà:

$$\frac{1}{10} = 10^{-1} \qquad \frac{1}{10^2} = 10^{-2}$$

e quindi:

$$38.29 = 3 \times 10^1 + 8 \times 10^0 + 2 \times 10^{-1} + 9 \times 10^{-2}.$$

Per la parte intera, pertanto, continuano a valere le considerazioni ormai acquisite, e volendo passare alla base 7 si procede come al solito. Per la parte decimale, dobbiamo esprimere 29 centesimi in termine di potenze negative del 7.

Procediamo allora come nella Tabella 2.7: prendiamo la parte decimale 0.29 e moltiplichiamola per 7. La parte intera che otteniamo, 2, è la più grande frazione di 7 che si avvicina a 0.29. Infatti, 2 settimi vale circa 0.286, e perciò 2 rappresenta la prima cifra settenaria nella conversione di 0.29. Possiamo allora eliminare il 2 e considerare la parte decimale 0.03. Qui applichiamo le medesime considerazioni e ricaviamo la seconda cifra settenaria, che risulta essere 0. Così la terza è 1 e la quarta è 3. Si scrive allora il risultato ottenuto, che cioè 38.29 equivale a circa 53.2013. Pur potendo continuare, fermiamoci qui.

Il processo potrebbe andare avanti all'infinito, ma, come sappiamo dalla nostra esperienza decimale, questo non è un caso strano. Ad esempio, per calcolare un terzo si ottiene 0 punto 3, 3, 3 e via di seguito. Tanto per ribadire le idee e il procedimento introdotto, vediamo come si esprime 38.29 in base 5. Per la parte intera si ha 123, come già si sa. Per la parte decimale, cominciamo a moltiplicare 0.29 per 5:

| | | | | |
|---|---|---|---|---|
| 0.29 | × | 5 | = | 1.45 |
| 0.45 | × | 5 | = | 2.25 |
| 0.25 | × | 5 | = | 1.25 |
| 0.25 | × | 5 | = | 1.25 |

Si ottengono le cifre cinquali 1, 2, 1, 1 e ci si accorge subito che, da questo punto in avanti, si otterranno tutte cifre 1. Siamo di fronte a un numero cinquale periodico, ma introdurre anche questo concetto ci porterebbe su una strada che non vogliamo percorrere.

Naturalmente, a noi interessa soprattutto la base 2. Per 38 abbiamo la solita conversione e possiamo applicare il metodo delle moltiplicazioni successivi alla parte decimale 0.29:

| | | | | |
|---|---|---|---|---|
| 0.29 | × | 2 | = | 0.58 |
| 0.58 | × | 2 | = | 1.16 |
| 0.16 | × | 2 | = | 0.32 |
| 0.32 | × | 2 | = | 0.64 |
| 0.64 | × | 2 | = | 1.28 |
| 0.28 | × | 2 | = | 0.56 |
| 0.56 | × | 5 | = | 1.12 |

Possiamo tentare anche una specie di verifica, trasformando il risultato ottenuto in decimale. La regola posizionale ci dà una bella espressione e una calcolatrice tascabile ci può aiutare, valutando le frazioni delle potenze di 2 corrispondenti ai bit 1:

$$0\times\frac{1}{2}+1\times\frac{1}{4}+0\times\frac{1}{8}+0\times\frac{1}{16}+1\times\frac{1}{32}+0\times\frac{1}{64}+1\times\frac{1}{128} =$$

$$= \frac{1}{4} + \frac{1}{32} + \frac{1}{128} \approx 0.28906.$$

Questo risultato, assai vicino allo 0.29 di partenza, ci rassicura della bontà della nostra procedura.

A questo punto, dato un qualsiasi numero nella sua forma decimale, siamo in grado di rappresentarlo in una base qualsiasi e, applicando la regola posizionale estesa alle cifre dopo la virgola, anche di eseguire la trasformazione inversa. Questo è soddisfacente dal punto di vista umano, ma non risolve completamente il problema di rappresentare questi numeri nella memoria dell'elaboratore. Infatti, spesso, abbiamo a che fare con numero straordinariamente piccoli o spaventosamente grandi. Ad esempio, possiamo dover trattare la massa di un elettrone (cioè $9.1095 \times 10^{-31}$ kg), e, all'opposto, la massa della nostra intera galassia (cioè, $4 \times 10^{41}$ kg). Questo succede in Fisica e in Chimica, e per quanto riguarda la Matematica, abbiamo visto che numero è 100!.

In questi casi, è bene distinguere la dimensione del numero dalla precisione con cui lo vogliamo trattare. Per esempio, 100 fattoriale è all'incirca uguale a $9.33 \times 10^{157}$. Per molte applicazioni, questa approssimazione è più che sufficiente e, per quanto riguarda le costanti fisiche, è già tanto se se ne conoscono con esattezza quattro o cinque cifre decimali.

Per arrivare a una convenzione comune, prendiamo un numero qualsiasi, ad esempio, 12.34 e portiamolo alla forma 0 punto qualcosa. Nell'esempio abbiamo 0.1234, che torna ad essere proprio lui se lo moltiplichiamo per 100, cioè per $10^2$. Se partiamo invece da 0.0012, questo lo possiamo portare alla forma 0.12, ma lo dobbiamo dividere per 100, cioè moltiplicare per $10^{-2}$. In questa maniera, per 100! si ottiene $0.933 \times 10^{158}$, mentre per la massa dell'elettrone si ha $0.911 \times 10^{-20}$ chilogrammi.

Questa forma convenzionale dei numeri si dice *normalizzata*, o forma *mantissa/esponente*. Propriamente, la *mantissa* è la parte di numero che segue la virgola e che deve cominciare, sempre, con una cifra diversa da zero. L'*esponente*, naturalmente, è quello da mettere alla base 10 per riportare il numero alla sia dimensione esatta. Così, se la mantissa è 314 e l'esponente è 4, il numero è 3140, mentre la stessa mantissa con esponente $-2$ rappresenta il numero 0.00314. L'esponente può ben essere 0, mentre la mantissa è zero solo per il numero 0.

Il numero di cifre che compongono la mantissa si dice la *precisione* del numero. Così, la precisione della massa dell'elettrone è di 7 cifre decimali, anche se la precisione fisica è di sole 5 cifre. Nel caso di 100!, la precisione che conosciamo è di 158 cifre decimali, ma rappresentandolo normalizzato ne limitiamo la precisione a sole 10 cifre decimali. L'elaboratore, per rappresentare nella sua memoria un numero con la virgola usa la forma mantissa/esponente adattata alla base 2. Ad esempio, il numero binario verrà normalizzato: la mantissa è una sequenza di bit e l'esponente quattro è trasformato nell'equivalente binario 100.

Molti elaboratori riservano un byte all'esponente e da 3 a 5 byte alla mantissa. Abbiamo osservato che un byte permette di rappresentare i numeri da $-128$ fino a 127, e quindi la dimensione dei numeri espressi in questa maniera è compresa tra $2^{-128}$ e $2^{127}$. Per capire meglio queste grandezze ricordiamo, come abbiamo già fatto, che $2^{10}$ è quasi uguale a $10^3$ e quindi basta impostare una proporzione per passare alla base 10:

$$128 : x = 10 : 3 \qquad x = \frac{3 \times 128}{10} \simeq 38.$$

Scopriamo così che il byte dell'esponente permette di rappresentare numeri tra $10^{-38}$ e $10^{+38}$. Ciò è sufficiente per la maggior parte delle applicazioni, ma alcuni elaboratori hanno anche rappresentazioni speciali che prevedono due byte per l'esponente. Questo porta la massima dimensione a $2^{32767}$, corrispondente a $10^{9800}$.

Per la mantissa si usano invece 3, 4 o 5 byte. Nel caso di 3 byte, la precisione dei numeri è quella di 24 bit, o meglio di 23 poiché uno indica il segno. Con la stessa proporzione di prima, si trova che la precisione è di circa 7 cifre decimali, il che è abbastanza buono per molte applicazioni, anche scientifiche. Passare a quattro byte significa avere 31 bit di precisione, e questo corrisponde a 9 o a 10 cifre decimali esatte. Oggi la rappresentazione 1 + 4 byte è abbastanza standard, ma per certe applicazioni che richiedono un'altissima precisione, esistono rappresentazioni che arrivano fino a più di 60 byte (cioè circa 500 bit) di mantissa. Questo corrisponde a circa 150 cifre decimali.

Nella terminologia informatica, i numeri rappresentati nel modo descritto, nella forma mantissa/esponente, si dicono *reali*. In effetti, essi hanno poco a che vedere con i numeri reali della Matematica, i quali hanno per lo più un numero infinito di decimali. Come s'è visto, i reali dell'Informatica hanno tutti un numero ben definito di cifre, e quindi sono, tutt'al più, *numeri razionali*. La rappresentazione informatica dei numeri reali è detta in *virgola mobile* o, con terminologia inglese, in *floating point*. Naturalmente, quando tratta un numero, l'elaboratore deve sapere di che tipo è; per le operazioni tra reali usa istruzioni speciali, dette appunto in virgola mobile, che tengono conto della mantissa e dell'esponente. Talvolta, uno speciale microprocessore, detto *coprocessore matematico*, è dedicato a trattare tali numeri.

Con questo, abbiamo praticamente concluso la nostra esposizione della rappresentazione delle informazioni numeriche. Si hanno così numeri assoluti, senza segno, per i naturali; numeri con segno per gli interi relativi; e numeri reali per tutti gli altri. Pertanto, una sequenza di quattro byte può corrispondere a tre numeri sostanzialmente diversi, a seconda di come vengono interpretati. Ma questa interpretazione dipende dall'uso che il programma ne fa o, meglio, che ne fa l'uomo attraverso un programma. Si ritorna così alla distinzione tra dato e informazione. Il dato, costituito dai 32 bit dei 4 byte, può rappresentare informazioni numeriche profondamente diverse. E' bene perciò ricordare che l'uso che ne fa l'elaboratore è determinato dall'uomo, e soltanto dall'uomo.

Ma la storia è appena cominciata. Infatti, le informazioni numeriche sono solo una piccola parte di quelle trattate dall'elaboratore. Storicamente, le prime informazioni dopo quelle numeriche ad essere trattate dall'elaboratore sono state le *informazioni alfa-*

*betiche*. Inizialmente, venivano accettate solo le lettere maiuscole, che avevano lo scopo di rappresentare convenzionalmente istruzioni dell'elaboratore e indirizzi della memoria centrale. Spesso, ad un carattere erano dedicati 6 bit e con 6 bit, come si sa, si rappresentano 64 informazioni. Ci si limitava pertanto alle 26 lettere dell'alfabeto inglese, alle 10 cifre, allo spazio e a qualche segno speciale come la punteggiatura, e poco più. Solo più tardi si cercò di portare nell'elaboratore l'insieme completo dei caratteri che compaiono sulle macchine da scrivere.

I tasti di una macchina da scrivere erano di solito 44 o 46, per un totale di circa 90 caratteri diversi. Già negli anni 1920 erano in uso le *telescriventi*, cioè macchine da scrivere elettriche collegate alla linea telefonica. Scrivendo su una telescrivente a Roma, collegata ad un'altra telescrivente a Milano, i caratteri battuti a Roma venivano trasmessi a Milano, e qui componevano lo stesso testo scritto a Roma. Le filiali di un'azienda potevano così essere collegate con la sede centrale e un giornalista poteva spedire un articolo alla redazione lontana del proprio giornale. Le trasmissioni avvenivano carattere per carattere secondo una convenzione che mandava impulsi elettrici specifici associati a ciascun tasto della telescrivente.

Inizialmente, il codice di trasmissione prevedeva una complessa sequenza di impulsi positivi e negativi per ogni carattere. La sequenza era un multiplo di cinque e prevedeva l'identificazione dell'inizio e della fine della codifica vera e propria di ciascun carattere. Si cercò poi di semplificare e abbreviare, fino a pervenire a un codice di 7 impulsi. Battendo, ad esempio, il tasto della A maiuscola, sulla linea telefonica venivano inviati i sette impulsi $+ - - - - - +$, che arrivati a destinazione venivano interpretati provocando la battuta della leva della A maiuscola. Poiché 7 impulsi corrispondono a 128 diversi codici, si aveva la possibilità di inviare tutti i caratteri scrivibili, 96, più altri che servivano a vari scopi.

Questo codice è conosciuto col nome di *ASCII*, una sigla il cui significato è *American Standard Code for Information Interchange*. Esso codifica i caratteri che si trovano sulla tastiera di una macchina da scrivere o di una telescrivente statunitense. Sono quindi presenti le 26 lettere dell'alfabeto inglese, sia maiuscole che minuscole. Non vi sono invece vocali accentate o altri segni diacritici. Ci sono le cifre e i segni di punteggiatura. C'è il dollaro e ci sono altri segni che oggi si sono imposti in tutto il mondo. Ad esempio, il *cancelletto* o *graticola* #, che significa "numero", la *"e" commerciale* & e la famosa *chiocciola* . In inglese si dice *at-sign* poiché deriva dalla preposizione "at" e indicava il luogo in cui si trovava la telescrivente in trasmissione.

In totale, i caratteri usati e codificati sono 96; gli altri 32 erano utilizzati come caratteri di controllo. Sulla telescrivente indicavano operazioni particolari e non dei veri e propri simboli. Ad esempio, uno denotava l'andare a capo, altri l'inizio e la fine della trasmissione, la richiesta di chi stesse trasmettendo o ricevendo, e così via. Oggi, solo il comando di andare a capo è rimasto, mentre gli altri codici indicano differenti funzioni. Ad esempio, quattro si usano per comandare il movimento del cursore sullo schermo: vai in su, in giù, a destra, a manca. In generale, sia in ingresso che in uscita, vengono adoperati per indicare particolari funzioni legate alla particolare periferica che li manda o li riceve.

All'inizio degli anni 1960, con l'affermarsi del byte, si decise di codificare ogni carattere con un singolo byte. Poiché il byte è composto di 8 bit, si stabilì che il primo bit a sinistra rimanesse uguale a zero. Così il codice ASCII della lettera A maiuscola si ottenne cambiando l'impulso + in 1 e l'impulso - in zero: 01000001. Questa codifica fu accettata da tutti i costruttori di elaboratori, eccetto l'IBM, che aveva un proprio codice, detto *EBCDIC*, che volle mantenere su tutti i propri prodotti. Solo nel 1981, quando decise di intervenire nel mondo dei personal, cambiò anche il proprio atteggiamento e adottò il codice ASCII sui propri personal. Da allora, l'ASCII è diventato il vero standard internazionale.

E' utile sapere (vedere Tabella 2.8) che la codifica ASCII delle lettere va dal codice 65 decimale della lettera A maiuscola al codice 90 della Z maiuscola. Le corrispondenti lettere minuscole hanno un valore maggiore di 32, e quindi partono da 97 per la a minuscola per arrivare a 122 per la z. Le cifre sono codificate ordinatamente da 48 fino a 57. Il codice più basso, 32, è riservato allo spazio, mentre i codici da 0 a 31 corrispondono ai citati caratteri di controllo; ad esempio, l'andare a capo ha codice 15. I codici non compresi tra quelli citati riguardano i segni di interpunzione e gli altri caratteri speciali, come parentesi, dollaro, cancelletto, e via di seguito.

Una parola come ADDENDO è rappresentata da una sequenza di byte, ciascuno corrispondente a una lettera. La doppie vengono memorizzate come tali:

01000001 01000100 01000100 01000101

01001110 01000100 01001111.

Un intero testo, come sequenza di parole e di segni di interpunzione, è codificato nello stesso modo e quindi è una lunga sequenza di byte, ognuno col suo specifico significato. Non occorre dire che la separazione tra le parole è indicata col codice dello spazio, trattato come un carattere alla stregua di tutti gli altri. In questo modo, la Divina Commedia altro non è che una successione di circa cinquecentomila byte

| D. | H. | C. | D. | H. | C. | D. | H. | C. |
|---|---|---|---|---|---|---|---|---|
| 32 | 20 | | 64 | 40 | @ | 96 | 60 | ‘ |
| 33 | 21 | ! | 65 | 41 | A | 97 | 61 | a |
| 34 | 22 | ” | 66 | 42 | B | 98 | 62 | b |
| 35 | 23 | # | 67 | 43 | C | 99 | 63 | c |
| 36 | 24 | $ | 68 | 44 | D | 100 | 64 | d |
| 37 | 25 | % | 69 | 45 | E | 101 | 65 | e |
| 38 | 26 | & | 70 | 46 | F | 102 | 66 | f |
| 39 | 27 | ’ | 71 | 47 | G | 103 | 67 | g |
| 40 | 28 | ( | 72 | 48 | H | 104 | 68 | h |
| 41 | 29 | ) | 73 | 49 | I | 105 | 69 | i |
| 42 | 2A | * | 74 | 4A | J | 106 | 6A | j |
| 43 | 2B | + | 75 | 4B | K | 107 | 6B | k |
| 44 | 2C | , | 76 | 4C | L | 108 | 6C | l |
| 45 | 2D | - | 77 | 4D | M | 109 | 6D | m |
| 46 | 2E | . | 78 | 4E | N | 110 | 6E | n |
| 47 | 2F | / | 79 | 4F | O | 111 | 6F | o |
| 48 | 30 | 0 | 80 | 50 | P | 112 | 70 | p |
| 49 | 31 | 1 | 81 | 51 | Q | 113 | 71 | q |
| 50 | 32 | 2 | 82 | 52 | R | 114 | 72 | r |
| 51 | 33 | 3 | 83 | 53 | S | 115 | 73 | s |
| 52 | 34 | 4 | 84 | 54 | T | 116 | 74 | t |
| 53 | 35 | 5 | 85 | 55 | U | 117 | 75 | u |
| 54 | 36 | 6 | 86 | 56 | V | 118 | 76 | v |
| 55 | 37 | 7 | 87 | 57 | W | 119 | 77 | w |
| 56 | 38 | 8 | 88 | 58 | X | 120 | 78 | x |
| 57 | 39 | 9 | 89 | 59 | Y | 121 | 79 | y |
| 58 | 3A | : | 90 | 5A | Z | 122 | 7A | z |
| 59 | 3B | ; | 91 | 5B | [ | 123 | 7B | { |
| 60 | 3C | < | 92 | 5C | \ | 124 | 7C | \| |
| 61 | 3D | = | 93 | 5D | ] | 125 | 7D | } |
| 62 | 3E | > | 94 | 5E | ^ | 126 | 7E | ∼ |
| 63 | 3F | ? | 95 | 5F | _ | 127 | 7F | Δ |

Tabella 2.8: Codice ASCII: D valore decimale, H esadecimale, C carattere

e una volta registrata in una memoria dell'elaboratore può essere modificata, stampata in particolari formati, manipolata, stravolta o semplicemente letta.

Un sistema di *elaborazione dei testi*, tipo *Word*, permette di immettere nell'elaboratore un intero documento, correggerlo, modificarlo e gestirlo in vari modi. Questo è il grande vantaggio offerto dagli elaboratori alla stesura di relazioni, tesi, racconti e libri, nonché allo studio dei testi letterari. In realtà, le opportunità offerte da questi sistemi sono pressoché infinite. Ad esempio, una funzionalità interessante è quella di mettere automaticamente in ordine alfabetico le parole di un elenco. Per far ciò, l'elaboratore sfrutta proprio la codifica ASCII dei caratteri: il punto fondamentale è che il carattere X precede il carattere Y se e solo se il codice di X è inferiore al codice di Y. Questo, esteso alle parole intere, permette di disporle in ordine lessicografico.

Quando, nel 1981, l'IBM decise di entrare nel mercato dei personal, non solo accettò la codifica ASCII, ma ne propose una estensione. Questa usava i codici, cioè le configurazioni di byte, che hanno un bit 1 nella prima posizione a sinistra. Questi codici corrispondono ai numeri da 128 a 255, che non erano sfruttati dall'ASCII tradizionale. I nuovi codici vennero sfruttati per alcuni caratteri particolari, come simboli matematici e certe lettere greche. Ma l'innovazione principale fu quella relativa ai cosiddetti *caratteri nazionali*. L'IBM voleva ampliare l'uso dei personal a tutto il mondo, e si rendeva conto che molte applicazioni sono legate a testi scritti nella stessa lingua dell'utente.

Ogni lingua europea, però, ha alcuni caratteri particolari. In francese e portoghese abbiamo la C con la cediglia; in tedesco l'umlaut o dieresi su A, O, U; in italiano e francese ci sono gli accenti acuto e grave sulle vocali; in spagnolo la N con la tilde; in olandese la Y con la dieresi, e così via. A tutti questi caratteri fu assegnato un codice dell'estensione, così che un testo, scritto in francese, italiano o tedesco, avesse una precisa rappresentazione nella memoria dell'elaboratore. Senza questi caratteri, si era costretti a scrivere *Citta*' invece di *Città*, e questo va contro consuetudini e tradizioni profondamente radicate e motivate da considerazioni filologiche

Oggi, lo sviluppo dei sistemi di elaborazione dei testi ha portato a ritenere inadeguata la codifica ASCII, quando si vogliano trattare situazioni complesse. Si pensi semplicemente alla gestioni di informazioni multilingue nei commerci e nelle relazioni internazionali, specie quando sono coinvolte lingue scritte in alfabeti diversi. Si è sentita allora la necessità di codici più vasti. Quello che si sta imponendo è il così detto *UNICODE*, che assegna due byte ad ogni carattere. Questo permette di codificare ben $65 \cdot 536$

simboli diversi e infatti abbiamo codici per i caratteri greci, cirillici, giapponesi e tanti altri, con i rispettivi segni diacritici, cioè accenti, sotto e soprascritti, varianti, eccetera.

E' bene ribadire ulteriormente il seguente punto, che già abbiamo discusso durante l'esposizione della rappresentazione dei numeri. Un byte è un dato, e la sua interpretazione è un'operazione umana che, secondo criteri umani, gli assegna uno specifico significato. Così questo byte:

| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

può significare il numero assoluto 149, il numero negativo $-107$, o anche il carattere "ò" secondo la codifica ASCII. Per l'elaboratore l'interpretazione è indifferente, ed è l'utente umano che, a seconda dei suoi intendimenti, ordina all'elaboratore di trattarlo in un modo o nell'altro. Una somma che usi tale byte darà risultati diversi a seconda del tipo di numero si consideri, ed avrà poco senso sommargli 10 se lo si considera un carattere.

D'altra parte, spesso il contesto determina il significato di un dato, e a questa regola non sfugge la memoria dell'elaboratore. Per questo, come s'è detto, si può parlare di informazioni anche nel mondo formale, e quindi a-significante, dell'elaboratore. Talvolta, si può sfruttare la commistione dei dati. Ad esempio, se sommiamo 32 alla lettera "A" maiuscola, otteniamo il codice ASCII della "a" minuscola, e questo è un metodo per trasformare maiuscole in minuscole, e viceversa. D'altra parte, occorre fare un po' d'attenzione e, a proposito di numeri negativi, se sommiamo 75 e 86 si può ottenere, forse non troppo inaspettatamente, il risultato $-95$, alquanto inusitato.

$$\begin{array}{cccccccccc} 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & + \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & = \\ \hline 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & \end{array}$$

## 2.4 Informazioni grafiche e sonore

Il concetto che sta alla base della rappresentazione di entità grafiche è lo stesso che abbiamo visto a proposito dello scanner. L'immagine viene idealmente e fisicamente suddivisa in un grandissimo numero di punti, per ognuno dei quali viene codificato il fatto che esso sia bianco o nero, nel caso di immagini in bianco e nero, oppure il colore, se l'immagine è a colori. Nel primo e più semplice caso, convenzionalmente il bianco è rappresentato dal bit 0, mentre il nero corrisponde al bit 1. Abbiamo visto come una pagina in formato A4, contenente ad esempio una pagina di testo o una lettera, venga ad occupare circa 8 milioni e 700 mila bit, ovvero più di un megabyte. Le immagini a colori occupano molta più memoria, come presto vedremo.

In effetti, prima di interessarci del colore, osserviamo che una codifica in bianco e nero va bene per memorizzare testi, schemi e disegni tecnici, nei quali le lettere e le linee sono nette e ogni punto rilevato può essere classificato come bianco o nero. Se vogliamo però codificare un disegno a matita o una fotografia in bianco e nero (vedere Figura 1.2, a pag. 7), la codifica a 0/1 non è più adeguata, poiché le varie sfumature di grigio vengono estremizzate, così che la riproduzione di ciò che è stato memorizzato risulta molto diversa dall'originale, rendendo netti in modo intollerabile tutti i dettagli. Questo è proprio ciò che succede quando si spedisce una fotografia via FAX, in quanto questo rileva solo i punti bianchi e neri.

In queste circostanze si preferisce utilizzare una macchina - FAX, fotocopiatrice o scanner - che possa rilevare i toni di grigio. Naturalmente, non possiamo più memorizzare un punto con un bit 0/1, ma occorrono più bit per assegnare al punto un codice a diversi valori che indichi la gradazione di grigio che meglio gli compete. Spesso si usa un codice a quattro bit. Poiché con 4 bit si possono indicare 16 valori diversi, il codice permette di memorizzare 16 gradazioni. Di solito si dà il valore 0 al bianco, il valore 1 a un bianco appena sporco, il valore 2 a un inizio di grigio, e così via fino a 14 per un grigio scurissimo, e il valore 15 al nero. Questo permette una riproduzione assai fedele, anche se quadruplica lo spazio di memoria per la registrazione.

Quello dello spazio occupato dalle immagini è un punto dolente. A 300 dpi, in un pollice quadrato (cioè, 6.45 $cm^2$) sono contenuti $90{\cdot}000$ punti e quindi possiamo calcolare quanti bit corrispondono a una fotografia nel formato standard di 10 per 15. Dieci centimetri sono 4 pollici e 15 centimetri sono 6 pollici. Quindi la foto corrisponde a 24 pollici quadrati e questo significa $2{\cdot}160{\cdot}000$ punti. Per trovare il numero di byte dell'occupazione, dobbiamo ora moltiplicare per 4, i bit di ciascuna gradazione di grigio, e poi dividere per 8, onde passare da bit a byte. Il risultato è di 1 milione e 80 mila byte, una quantità che entra appena in un floppy, e siamo ben lontani da un intero foglio A4.

Quando la riproduzione deve essere di qualità superiore, si aumentano i punti da memorizzare e si aumentano le gradazioni del grigio. Si passa ad utilizzare un byte intero per codificare il grigio di un punto e questo permette di avere 256 gradazioni diverse, da 0 per il bianco a 255 per il nero. Lo spazio richiesto raddoppia, e questo non è un inconveniente da poco. Si usano allora particolari metodi di compressione, che permettono di diminuire lo spazio occupato. Si

perde in tal modo una piccola parte delle informazioni relative alla figura, ma ciò non ne compromette una buona riproduzione. Vedremo più avanti di accennare a queste tecniche. Esse sono ancora più importante per le immagini a colori, che richiedono molto più spazio di memoria.

Per le immagini a colori si usano vari tipi di codifica, a seconda dell'accuratezza che si desidera e dell'apparecchiatura di rilevazione e di visualizzazione usata, Il codice più semplice è quello che prevede un byte per ogni punto rilevato. Questo comporta la registrazione di 256 colori diversi, di modo che la rappresentazione risulta abbastanza approssimativa. Ormai, si usa questo codice solo quando non c'è necessità di una buona riproduzione o si usano colori convenzionali, come ad esempio per evidenziare una porzione di testo. Molto più utilizzato è il codice a due byte, che permette di estendere a $65 \cdot 536$ il numero dei colori rappresentabili. Questa è una quantità certamente sufficiente per molte applicazioni.

Tuttavia, la rappresentazione dei colori che è usata più frequentemente è quella che codifica ogni colore con tre byte. Tre byte permettono di distinguere $2^{24}$ colori diversi, e cioè più di sedici milioni. Per cercar di capire come funziona questa codifica, almeno nelle linee generali, dobbiamo discutere alcuni concetti che fanno parte di una scienza detta *cromatologia*. Tutti ci ricordiamo che Newton dimostrò come la luce bianca possa essere scomposta, mediante un prisma, nella successione, o spettro, dei sette colori fondamentali: rosso, arancio, giallo, verde, blu, indaco e violetto. Usando le matite, abbiamo imparato che tre colori, rosso, giallo e blu, permettono di ottenere gli altri colori: ad esempio giallo e rosso danno l'arancione.

Questo metodo è detto tecnicamente *sottrattivo*, in quanto ogni sovrapposizione sottrae qualche colore alla gamma completa che era presente nel bianco iniziale del foglio. Nella pratica, questo metodo è adottato ad esempio dalle stampanti che hanno a disposizione quattro inchiostri dei colori di base e con essi compongono tutti i colori possibili. In realtà, i quattro colori sono il ciano, che è una gradazione dell'azzurro, il magenta, che è un punto del rosso, il giallo e il nero. Nella terminologia inglese, questo metodo si indica con la sigla *CMYK*, cioè Cyan, Magenta, Yellow e blacK. Il nero, naturalmente, serve per i toni scuri e per il colore nero come tale.

Lo schermo del monitor, però, non compone i colori per sottrazione, ma li compone per addizione. Dal punto di vista tecnico, questo significa che ogni punto, o pixel, emette più radiazioni luminose che si sommano e, se le radiazioni fossero tutte quelle dello spettro, noi vedremmo una luce bianca, come quella del sole. In questo caso, è la mancanza di colori che dà il nero, e una radiazione verde, sommandosi al rosso, ci dà la sensazione del giallo. I colori di base cambiano completamente e di norma si considerano il rosso, il verde e il blu. Poiché questi tre colori in inglese suonano Red, Green, Blue, la sigla che identifica la classificazione additiva dei colori si dice *RGB*, e così la chiameremo anche noi.

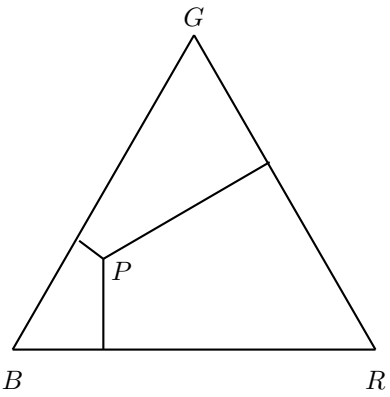



Figura 2.1: Il triangolo dei colori

Ogni colore si può immaginare come la somma di una certa quantità di rosso, di verde e di blu. Ricordiamo che lo spettro luminoso contiene colori puri, e il giallo, ad esempio, ha una lunghezza d'onda di 0.58 millesimi di millimetro. L'arrivo al nostro occhio di radiazioni verdi e rosse, ci dà la sensazione del giallo, ma ciò che vediamo non è il giallo puro. Comunque, questo fatto non è rilevante, e ciò che importa è quello che percepiamo, indipendentemente da come lo vediamo. Inoltre, è ben noto che esiste anche una *psicologia dei colori*, e la stessa lunghezza d'onda viene percepita in modo diverso da persone diverse o in situazioni diverse. Gli accostamenti possono darci la sensazione di vedere colori differenti da quelli che sono in realtà. Tutte queste cose, tuttavia, non interessano la produzione dei colori, se non per i loro aspetti comunicativi.

Se noi disponiamo, idealmente, i tre colori di base nei vertici di un triangolo equilatero, detto *triangolo RGB*, possiamo far corrispondere ogni colore a un punto $P$ del triangolo stesso. La quantità di rosso, di verde e di blu di tale colore è misurata, percentualmente, dalla distanza di $P$ dal lato opposto al vertice del colore. Così, nella Figura 2.1, il punto $P$ corrisponde ad un colore che ha poco rosso, abbastanza verde e molto blu. La nostra sensazione è di un pandizucchero. Naturalmente, ad esempio, se $P$ coincide col vertice $R$, esso corrisponde a zero verde, zero blu e, di conseguenza, tutto rosso, essendo l'altezza del triangolo la distanza del vertice dal lato opposto. Il punto centrale del triangolo, cioè il suo baricentro, corrisponde a uguali quantità di rosso, verde e blu, e

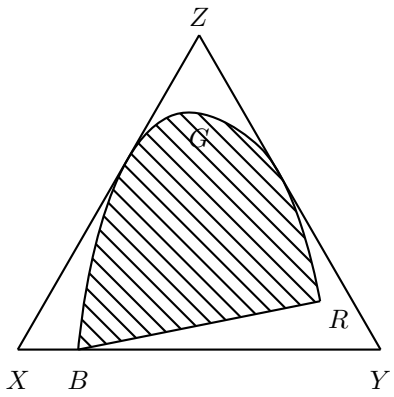



Figura 2.2: Il triangolo $XYZ$

quindi la sensazione che ci procura è quella del bianco. Si noti che il bianco ha perciò coordinate uguali per tutti e tre i colori.

Riassumendo, questo triangolo definisce le coordinate di ciascun colore in termini di rosso, di verde e di blu. La percentuale dei colori di base è, grosso modo, inversamente proporzionale alla distanza del colore considerato dai tre vertici del triangolo. Così, un punto vicino ad $R$ contiene molto rosso e poco verde e blu. Un punto sul lato $RG$ contiene rosso e verde, ma non contiene blu.

Il triangolo è una idealizzazione di come stanno le cose in natura. Queste sono meglio descritte dal *triangolo* $XYZ$, definito nel 1931 dalla *Commissione Internazionale del Colore* (CIE: Commission International d'Eclairage) (vedere Figura 2.2). I vertici sono denotati dalle tre lettere $X$, $Y$ e $Z$ perché, in pratica, non corrispondono a colori reali. Il blu, il rosso e il verde si trovano in posizioni un po' particolari, lungo una linea curva che è nota come *linea spettrale*, cioè la linea dei colori puri che appartengono allo spettro della luce. In altre parole, la linea spettrale corrisponde alle combinazioni di rosso, verde e blu che ci danno le stesse sensazioni dei colori puri.

Un'altra linea importante è il segmento $BR$ tra blu e rosso, corrispondente alle gradazioni del viola. E' detta infatti *linea delle porpore sature* e, assieme alla linea spettrale, determina la zona del triangolo $XYZ$ corrispondente ai colori reali. Ogni punto è determinato dalle tre coordinate $x$, $y$ e $z$ che non coincidono perfettamente con le distanze del punto dai tre lati. Questa convenzione internazionale per la codifica dei colori opera da riferimento di base, anche se nell'elaboratore le cose sono piuttosto diverse.

Nell'elaboratore, ogni colore è rappresentato da tre byte, il primo corrispondente alla quantità di rosso, il secondo alla quantità di verde e il terzo alla quantità di blu. Ad esempio, tre byte con i valori $(255, 0, 0)$ rappresentano il rosso, mentre i valori $(255, 255, 0)$ corrispondono al violetto. Se consideriamo ora tre byte con i valori $(127, 0, 0)$, di nuovo abbiamo quantità nulle di verde e di blu. Siamo perciò sul rosso e ci possiamo chiedere che differenza ci sia col colore $(255, 0, 0)$. Se osserviamo i due colori come vengono riprodotti, ci accorgiamo che il secondo è molto più scuro del primo. In effetti, se manteniamo il verde e il blu a zero, diminuendo il valore del rosso questo diventa sempre più scuro, fino ad arrivare al nero quando si giunge alla terna $(0, 0, 0)$.

Come abbiamo già avuto modo di dire, nella sintesi additiva dei colori, lo 0 corrisponde al nero e l'1 al bianco. L'1 sta a significare completezza di rosso, di verde e di blu, e questo, nella codifica a tre byte, corrisponde alla terna $(255, 255, 255)$. In generale, se partiamo con un colore come ad esempio $(89, 196, 106)$ e poi diminuiamo proporzionalmente i tre colori di base, il colore si scurisce, pur rimanendo dello stesso tono. Viceversa, se aumentiamo proporzionalmente, il colore schiarisce. La percentuale relativa di rosso, verde e blu determina il colore nella sua tonalità. Aumentando o diminuendo le tre quantità, lasciandole nella stessa proporzione, permette di schiarire o scurire il colore, senza però cambiarlo. Molti programmi, come *Photoshop*, permettono di scegliere o definire i colori, passando dal loro codice alla rappresentazione visiva, e viceversa. Questi sistemi, cioè, ci fanno vedere il colore corrispondente a un codice, o ci mostrano il codice legato a un determinato colore.

Oltre alla notazione decimale, come $(255, 0, 0)$, è molto usata la notazione esadecimale, cioè la rappresentazione dei tre colori nella base 16. Questa base ha il vantaggio di essere molto compatta e permette di vedere, in modo quasi immediato, la configurazione in bit dei tre byte del codice. Infatti, ogni cifra esadecimale corrisponde a quattro bit, così che ogni byte può essere rappresentato da due cifre esadecimali. Con un po' di pratica, questa notazione si rivela molto semplice ed utile. Anche l'*HTML* accetta questo modo di definire i colori.

Ricordiamo che il numero 178 si trasforma in base 16 applicando il metodo delle divisioni successive:

$$\begin{array}{rcrcll} 178 & : & 16 & = & 11 & \text{resto } 2 \\ 11 & : & 16 & = & 0 & \text{resto } 11 \end{array}$$

I due resti che si ottengono sono, nell'ordine dal basso verso l'alto, 11 e 2. Ci ricordiamo che, come convenzione, per le cifre superiori a 9 si usano le lettere maiuscole. Quindi si ha $A = 10$, $B = 11$ e così via fino ad $F = 15$. Il numero 178 corrisponde a $B2$. Risulta interessante (si veda a pagina 21) che, se associamo a ciascuna cifra esadecimale il proprio codice binario a 4 bit, la conversione in base 2 del numero di partenza corrisponde alla giustapposizione degli 8 bit delle due

cifre esadecimali. Ad esempio,

$$B2 = 10110010,$$

e questo è in chiaro il contenuto del byte, visto bit a bit.

Con la notazione esadecimale, la codifica del rosso è $FF0000$, quella del bianco $FFFFFF$ e quella del nero 000000. Come si vede, per convenzione, le cifre 0 vanno sempre messe, così che il codice è formato, in tutti i casi, da sei caratteri, ciascuno a rappresentare una cifra esadecimale e, a coppie, il contenuto dei tre byte. L'ordine assunto è, ricordiamo, rosso, verde e blu. Alcuni programmi accettano, oltre alla codifica completa, alcune parole convenzionali inglesi: red, green, blue, yellow, orange, violet. Questi nomi vengono automaticamente convertiti nel codice corrispondente.

L'enorme quantità di spazio richiesta dalla memorizzazione delle immagini a colori, codificate nel modo descritto, ha spinto i ricercatori a studiare metodi che permettano di comprimere questi dati e portare i file corrispondenti a dimensioni più ragionevoli. Questi metodi sono oggi applicati in maniera standard e si ottengono compressioni molto elevate che riducono l'occupazione anche del 50 o del 60 per cento. Tutti i metodi adottati per la compressione delle immagini causano una piccola perdita di informazioni, nel senso che l'immagine riprodotta è un po' meno buona di quella rilevata. Ma questa differenza non influisce sensibilmente nella nostra percezione dell'immagine ed è ben tollerata.

Un processo, in parte analogo, in parte completamente diverso, subiscono i suoni per essere registrati e poi riprodotti dall'elaboratore. Come i colori e le immagini, i suoni sono quantità analogiche (o continue) sia che si tratti di semplici rumori, della voce o della musica sinfonica. Essi perciò, in fase di memorizzazione, devono subire un processo di conversione da analogico a digitale, e un processo inverso di conversione da digitale ad analogico, in fase di riproduzione. Questi processi vengono eseguiti da apposite apparecchiature collegate, in ingresso, a un *microfono* e, in uscita, alle *casse*. Le procedure di conversione sono state studiate ormai da tanti anni e sono state applicate inizialmente alle registrazioni musicali, passando dai dischi tradizionali ai CD.

I dischi tradizionali registrano i suoni, in special modo la musica, incidendo un solco che riproduce, in modo continuo, le onde sonore che arrivano al microfono. Pertanto, la registrazione è di tipo analogico, continuo. In modo analogo, la musica registrata sui nastri magnetici riproduce in modo continuo i suoni prodotti dagli strumenti e dalla voce. Un po' sui generis, ma sempre di tipo analogico, è la riproduzione dei suoni sulla pellicola cinematografica: essi

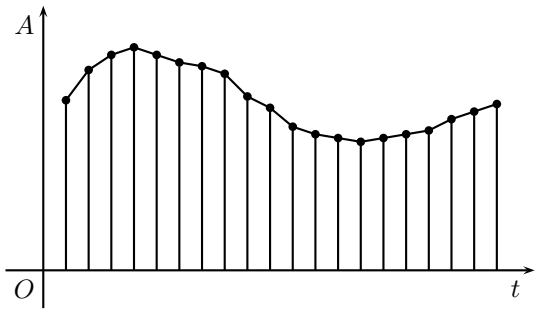



Figura 2.3: Esempio di campionamento

sono trasformati in linee più o meno spesse sui fotogrammi e la riproduzione richiede la trasformazione di informazioni luminose in informazioni sonore. La tecnica adottata dai *Compact Disk* rivoluzionò queste tecniche di registrazione dei suoni.

La registrazione di un CD comprende vari passaggi. Il suono, catturato da un microfono, viene trasformato in segnali elettrici continui. Sono questi che, passando attraverso un *convertitore analogico/digitale* vengono trasformati in numeri e quindi rappresentati da sequenza di bit. Queste sono infine registrate sul disco da un'opportuna apparecchiatura laser. La fase più importante, svolta dal convertitore, si dice *campionamento* o *campionatura* (vedi Figura 2.3). Un dispositivo elettronico scompone il segnale nelle sue componenti e, a intervalli regolari, ne rileva il valore, detto appunto *campione*. Tale valore, convertito in binario secondo una scala opportuna, è la quantità che viene registrata sul CD o, in generale, su una qualche memoria dell'elaboratore.

Il campionamento, avviene secondo modalità ben precise e con una frequenza che, secondo la teoria, è tanto più alta quanto più bassa è la banda del segnale. Questo assicura che, nella fase inversa di riconversione in suono, la riproduzione sia pressoché perfetta. In questa fase, il Compact Disk viene letto, i numeri registrati convertiti nei corrispondenti valori e questi vengono trasformati da discreti in continui, interpolando i punti ottenuti. La curva che si ottiene è praticamente la stessa dell'onda sonora da cui si era partiti, e questo assicura la fedeltà della riproduzione. L'ascolto, naturalmente, avviene per mezzo delle casse o di altre apparecchiature.

La registrazione e la lettura avvengono per mezzo di un raggio laser, e pertanto tali apparecchi sono detti anche *dischi ottici*. Su una base di sostegno si trova una pellicola di alluminio riflettente, protetta da uno strato di plastica trasparente. Il laser sfrutta un raggio di luce coerente, e quindi di grande concentrazione focale. In tal modo può incidere sull'alluminio, e quindi rilevare, delle tacche microscopiche. Hanno infatti dimensioni dell'ordine del micron (si veda la

Figura 2.4), e questo assicura una grandissima quantità di informazioni per centimetro quadrato. Da tale caratteristica deriva il nome di Compact Disk. Purtroppo, le tacche non possono essere modificate, una volta che siano state create, e quindi i CD sono apparecchiature di sola lettura, una volta che siano stati registrati.

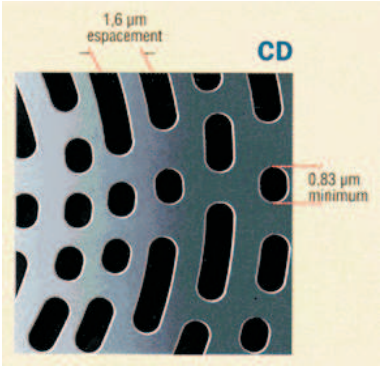



Figura 2.4: Le tacche su un CD

Per lo più, i *CD riscrivibili* hanno solo spazio aggiuntivo per espandersi, abbandonando le parti già registrate e scorrette. D'altra parte, è interesse delle case discografiche, e non solo, di evitare che le registrazioni vengano modificate. Oggi, questa preoccupazione si è estesa a tutti i produttori di testi multimediali, per evidenti problemi di Copyright. Infatti, tanto i CD quanto la loro versione più recente, i *DVD*, sono usati per registrare ogni tipo di informazione, mediante software e apparecchiature opportune, dette *masterizzatori*. Del loro utilizzo come memorie di massa, abbiamo già parlato e, da un punto di vista logico, sono analoghi ai dischi magnetici tradizionali, anche se un po' più lenti.

Con questo abbiamo terminato la nostra esposizione dei vari tipi di informazioni che possono essere registrate ed utilizzate da un elaboratore. Come si è visto, ciò che permette una così ampia varietà, è semplicemente il fatto che tutti i dati presi in considerazione hanno la possibilità di essere rappresentati da sequenze di bit 0/1. Questo fattore comune fa sì che il tipo di informazione, se cioè informazione numerica, alfabetica, grafica o sonora, non sia insita nel dato, ma possa essere trasferita alla parte software: Si ha così la capacità di trattare un'enorme varietà di dati e, come s'è detto, l'elaboratore non è semplicemente un calcolatore, cioè una macchina per eseguire conti, ma permette di gestire ogni tipo di informazione.

# Capitolo 3

# TRATTAMENTO DEI DATI

## 3.1 Sicurezza delle informazioni

Da tempo immemorabile si sta svolgendo una lotta, silenziosa e segreta, fra due modi opposti di rappresentare le informazioni. Da una parte il metodo analogico, dall'altra quello digitale. Abbiamo già visto la differenza e abbiamo ricordato la vecchia registrazione analogica della musica sui dischi di vinile o i nastri magnetici. Ma già ai tempi dei greci, Pitagora dette il primato ai numeri con cui si conta, che, secondo lui, dovevano essere l'anima dell'Universo. Al contrario, Euclide preferiva le grandezze continue, come le lunghezze, le aree, i volumi e gli angoli. Bisogna dire che Euclide vinse e le grandezze analogiche furono preferite, anche se si conservò una piccola sacca di fautori di Pitagora.

In realtà, i matematici non hanno mai preso una posizione definitiva e, con lo spiccato senso pratico che li caratterizza, hanno dato la palma a questo o a quello, a seconda di ciò che fosse più conveniente per risolvere un determinato problema. Ad esempio, Nepero inventò i logaritmi come quantità numeriche, e qualcuno (E. Gunter nel 1623) li usò per inventare il regolo calcolatore, che tratta grandezze analogiche. Le macchine calcolatrici, a partire dalla Pascaline, erano digitali, ma nel 1800 e nel 1900 si costruirono macchinari di calcolo di tipo analogico, per valutare le maree, prevedere il tempo e studiare le migrazioni. Si crearono apparecchiature analogiche per calcolare derivate e integrali, e si pensò che presto le grandezze analogiche avrebbero preso il sopravvento.

Con la nascita degli elaboratori elettronici, tipiche apparecchiature digitali, la superiorità delle macchine analogiche fu messa di nuovo in discussione. Esse mostrarono chiaramente i propri limiti: una scarsa accuratezza nei risultati e, soprattutto, una incertezza di base nelle interpretazione e nella registrazione dei dati. Nei calcoli, la precisione non può superare le tre o quattro cifre decimali. Registrare, come si fa di solito, un dato come lunghezza o come angolo, può dare adito a dubbi e facili contestazioni. Infine, la conservazione di questi dati è difficile e i supporti si deteriorano col passare del tempo. Come esempio, si pensi alla registrazione della musica, che abbiamo già citato: l'accuratezza non può essere spinta più di tanto e il tempo danneggia irrimediabilmente i dischi.

Tutto fluisce e tutto si deteriora, così che qualunque supporto andrà danneggiandosi, per una inevitabile legge naturale. Il vantaggio, che le grandezze discrete hanno su quelle continue, è il fatto che queste ultime sono legate alla struttura fisica del supporto, così che il deterioramento di questo comporta automaticamente il deterioramento delle informazioni. I dati discreti, invece, sopportano facilmente una imprecisione di registrazione e quindi anche quel po' di danni portati dal tempo e dall'incuria. Ad esempio, in molte apparecchiature lo 0 è indicato da una differenza di potenziale di 0 Volt, l'1 da 5 Volt. Tuttavia, è interpretato come 0 qualunque voltaggio tra 0 e 0,7 Volt, e come 1 ogni voltaggio da 3 a 5 Volt. Questo aumenta notevolmente la sicurezza dell'informazione.

Rappresentare un dato con una sequenza di zeri ed uni è pertanto molto più affidabile che rappresentarlo come lunghezza, e se ne può accrescere la precisione aumentando il numero di bit. Questo dell'occupazione dello spazio, è l'unico vero inconveniente delle informazioni digitalizzate, e si cerca di superarlo con apparecchiature e supporti sempre più tecnologicamente evoluti. Abbiamo visto i circuiti integrati, i dischi magnetici e quelli ottici. Di fatto, oggi è possibile comprimere miliardi di informazioni in pochissimo spazio. Tuttavia, non c'è modo di impedire che qualche errore avvenga, e un bit 1 sia interpretato come 0, o viceversa. Si pensi, ad esempio, a un evento fortuito, ma non impossibile, come il passaggio di un raggio cosmico che magnetizza o smagnetizza un punto su un disco magnetico.

In realtà, c'è un'altra situazione nella quale si verificano errori, anzi se ne verificano con una frequenza enormemente più alta che nella registrazione. Si tratta della trasmissione dei dati, sia che questa avvenga su fibra ottica, su linee telefoniche o via radio. Tutti i mezzi trasmissivi sono sottoposti a interferenze, i percorsi sono lunghi e basta un niente su una linea

di trasmissione perché un bit salti. Questo, naturalmente, vale anche per le informazioni analogiche, ma, come abbiamo visto, i dati digitali sono di per sé più sicuri e, cosa molto più importante, un elaboratore può accorgersi quando si sia verificato un errore, può segnalarlo e, in certi casi, può addirittura correggerlo automaticamente.

Questa caratteristica delle informazioni digitali, di poter controllare automaticamente i propri, eventuali errori, è l'arma che ha decretato la vittoria sulle informazioni analogiche, che mancano completamente di una proprietà del genere. L'idea di base è semplice ed è ormai applicata a tutte le informazioni sugli elaboratori, nonché in molti altri contesti.

Partiamo dalla memoria centrale e diciamo subito che un byte, fisicamente, non è costituito da 8 bit, ma da 9. Quello in più, è un bit che l'utente non vede e deve ignorare, ma è di fondamentale importanza. Esso si dice *bit di parità* ed è gestito direttamente dall'elaboratore, che lo modifica e lo controlla secondo precise regole. Esso deve il suo nome alla regola che lo governa: l'elaboratore conta i bit uguali ad 1 del byte considerato e se tale numero è pari, mette nel bit di parità il valore 0; se la somma è dispari, vi mette 1:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | **0** |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | **1** |

.

In questo modo, il numero totale di bit uguali ad 1, nel byte esteso a 9 bit, è sempre un numero pari. Infatti, se negli otto bit tale numero era già pari, aggiungendo 0 non cambia. Se era dispari, l'1 aggiunto fa sì che divenga pari. Nei milioni e milioni di byte che formano la RAM di un elaboratore, vige la ferrea legge che il numero dei bit uguali ad 1 deve essere sempre pari. Questo vale, naturalmente, anche per le memorie secondarie e per i dati che vengono trasmessi su una qualunque linea di trasmissione.

Questa semplice regola fa sì che sia banale, per l'elaboratore, rendersi conto se si è verificato un errore in uno qualsiasi dei byte della RAM. Periodicamente, la RAM è scorsa byte per byte e controllata sulla parità dei propri bit, nella versione estesa a 9. In questo modo si ha un'immediata segnalazione se si è verificato un errore nello hardware e chi gestisce l'elaboratore può prendere i provvedimenti opportuni. Naturalmente, la parte di memoria che contiene il byte difettoso deve essere sostituita. Nel caso di un grosso elaboratore, la regola è, come si è visto, che esista un gemello. Questo continuerà a funzionare, mentre si ripara il danno. Nei personal, purtroppo, non c'è che da portare l'elaboratore a riparare.

Gli stessi controlli possono esser fatti sulla memoria secondaria e sulle trasmissioni. Nel caso delle memorie di massa, l'errore in un byte fa sì che tutto il settore debba essere escluso dalla successiva operatività del disco. Il settore viene marcato come inutilizzabile e non verrà più preso in considerazione dal sistema operativo. Purtroppo, questo può comportare la perdita di un archivio, ma questo può essere il male minore, specie se abbiamo una copia dell'archivio stesso. Quella di fare *copie di sicurezza* dei dischi è una pratica che va sempre seguita. Vi sono apparecchiature apposite per eseguire il *back-up*, cioè il *salvataggio*, del contenuto del disco rigido e abituarsi ad effettuare periodici salvataggi può far risparmiare tanti patemi.

La *trasmissione dei dati* è l'operazione più sottoposta ad errori casuali. Per questo, quando si comincia la spedizione di un archivio, questo viene suddiviso dal Sistema Operativo in tanti blocchi, detti in gergo *pacchetti*. Ogni pacchetto è opportunamente identificato per essere poi assemblato con tutti gli altri al momento dell'arrivo. Quando un pacchetto giunge a destinazione, viene controllato byte per byte e nel suo complesso, on tecniche quali la *checksum* (somma di controllo) o il CRC (*Cyclic Redundancy Checking* = Verifica di ridondanza ciclica). In caso non si rilevino errori, viene mandato un messaggio (o *aknowledgement*) all'elaboratore di partenza per avvertirlo che quello specifico pacchetto è arrivato regolarmente. Se l'*acknowledgement* non ritorna, l'elaboratore provvede a spedire una nuova copia del pacchetto. Solo quando tutti i pacchetti di un archivio sono arrivati regolarmente, l'elaboratore in ricezione disfa i pacchetti e ricompone l'archivio nella sua interezza, e senza errori.

C'è un limite nel metodo che usa il bit di parità: si suppone implicitamente che si sia verificato un solo errore all'interno del byte. E' facile capire che se cambiano due bit, comunque cambino, la parità del byte rimane invariata e il controllo fallisce. Si fa allora l'ipotesi che su uno stesso byte non si verifichino due errori contemporaneamente. Da un punto di vista statistico, questa è un'ipotesi del tutto realistica, specie nella memorizzazione, e non si ritiene necessario introdurre tecniche più sofisticate che permetterebbero di accorgersi anche di due errori. Queste tecniche richiedono l'aggiunta di almeno un altro bit, e questo si ritiene troppo costoso nei confronti del minimo vantaggio che darebbe.

La tecnica del controllo di eventuali errori è oggi molto utilizzata, non solo in Informatica, ma anche in tutti i settori in cui un errore potrebbe causare danni rilevanti, specie quando corrono soldi. L'esempio più noto è quello del codice fiscale. Come si sa, esso è composto di 16 caratteri alfanumerici. I primi tre sono estratti dal cognome; i tre successivi dal nome proprio. C'è poi un gruppo di 5 caratteri corrispondenti alla data di nascita e al sesso: due cifre

per l'anno, una lettera per il mese e due cifre per il giorno. Queste due ultime cifre sono aumentate di 40 per le donne. Infine, ci sono 4 caratteri, una lettera e tre cifre, che indicano il comune di nascita, secondo una tabella convenzionale.

| | | |
|---|---|---|
| Cognome | Sprugnoli | SPR |
| Nome | Renzo | RNZ |
| Data Nascita | 04/03/1942 | 42C04 |
| Sesso | Maschio | – |
| Luogo Nascita | Cogoleto | C823 |
| Controllo | | W |

Il sedicesimo e ultimo carattere non porta alcuna nuova informazione, ma come il bit di parità è un di più che funge da controllo. Questa lettera viene stabilita in funzione di tutti i caratteri che la precedono, secondo regole abbastanza complicate, ma del tutto meccaniche. Un semplice programma dell'elaboratore è in grado di valutare il sedicesimo carattere, una volta che conosca i primi quindici. Questo avviene in fase di creazione del codice fiscale. In caso di utilizzo, l'elaboratore controlla se l'ultimo carattere è quello che dovrebbe essere secondo le regole, e segnala la errore se questa corrispondenza non è verificata. In questo modo ci si rende conto di errori materiali e di possibili truffe.

Anche i codici a barre (vedi Figura 1.5), che abbiamo ricordato, funzionano sullo stesso principio e un errore di lettura è immediatamente segnalato. Così, ad esempio, durante il conto che la commessa ci fa al supermercato, un suono avverte che si è verificato un errore. La commessa ripassa il prodotto e l'errore viene corretto, nell'interesse sia del supermercato, sia del cliente. Purtroppo, un codice di controllo non esiste sulle targhe automobilistiche, per cui sono all'ordine del giorno le contestazioni sulle multe, poiché non è affatto raro che un vigile o un poliziotto sbaglino a trascrivere la targa di un'auto in corsa, creando, anche se in buona fede, un bel po' di inconvenienti.

Queste tecniche di controllo, come si vede, si applicano soprattutto alle informazioni codificate. Infatti, di regola, i codici sono non ridondanti, cioè se cambiamo anche un solo simbolo, cambia il valore del codice. Se BJ 268 TS è una targa automobilistica, lo è anche BJ 298 TS, come lo è BJ 268 TZ, e nessuno può dire che una di questa targhe è sbagliata. Purtroppo (o per fortuna) siamo abituati ad esprimerci e ad intendere con ampi spazi di ridondanza. Ad esempio, possiamo togliere anche molte lettere a una frase scritta e riuscire tuttavia a comprenderla benissimo. La stessa cosa succede quando parliamo e questo è sfruttato sistematicamente dalle trasmissioni telefoniche.

I codici di controllo, come il bit di parità, introducono una ridondanza artificiale in una informazione. In questo modo, se perdiamo qualcosa di quella informazione, la ridondanza introdotta ci fa capire che qualcosa non va, proprio perché sono venute meno le norme che regolavano questa ridondanza. In effetti, come s'è detto, aggiungendo maggiore ridondanza, ad esempio due bit, si possono controllare meglio le informazioni, semmai accorgendoci quando si verificano più errori allo stesso tempo. Alternativamente, possiamo usare bit aggiuntivi per rilevare dove si è verificato l'errore. Questo, immediatamente, ci dice di che errore si tratta. Infatti, se l'errore è in una posizione in cui troviamo scritto 1, lo si corregge scrivendo 0, e viceversa.

In questi casi si parla di *codici autocorrettori* e i metodi per realizzarli sono stati studiati in modo matematico e assai approfondito. Qui possiamo semplicemente far vedere che i codici autocorrettori esistono, dandone un esempio molto semplice. Tanto semplice che non lo si può adottare in pratica, poiché i bit da aggiungere sono troppi; ma, naturalmente, non possiamo permetterci di essere più specifici. Consideriamo allora un gruppo di 8 byte, col loro bit di parità. Immaginiamo di associare a questi 8 byte un byte ulteriore di controllo, che contenga il bit di parità relativo alle varie posizioni del byte. Diciamo che è un bit di parità in verticale:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | **0** |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | **0** |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | **1** |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | **1** |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | **0** |
| 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | **1** |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | **0** |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | **0** |
| **1** | **1** | **0** | **0** | **0** | **1** | **1** | **1** | **1** |

Si osservi che il bit di parità del nono byte è, automaticamente, il bit di parità degli otto bit di parità. Infatti, se si vuole, esso è il bit di parità dell'intero pacchetto. Supponiamo allora che si verifichi un errore, ad esempio, un bit uguale ad 1 divenga uguale a 0. In questo modo, abbiamo un errore di parità sia nel byte sia nella colonna corrispondente al bit sbagliato. L'elaboratore, accorgendosi che la parità non va bene nel byte numero 3 e in corrispondenza della colonna 4, capisce che l'errore si è verificato in questo punto preciso, e per correggerlo deve semplicemente cambiare il valore del bit. Se è 0 lo cambia in 1, come in questo esempio, e se è 0 lo cambia in 1.

Le tecniche di autocorrezione degli errori sono piuttosto costose a causa della ridondanza che si deve introdurre. Quando si usa la semplice rilevazione dell'errore, le azioni da intraprendere sono diverse a seconda di dove l'errore si è verificato. In caso di trasmissione, come s'è visto, la cosa più semplice è

di trasmettere di nuovo il pacchetto incriminato. Se l'errore si è verificato su un disco, si può escludere il settore dalle successive operazioni di lettura e scrittura, a meno che il settore errato non possa essere recuperato. I guai più grossi si hanno quando salta un bit della RAM e allora non c'è da far altro che eliminare il banco di memoria e sostituirlo con uno nuovo.

Talvolta, le manovre di correzione si possono eseguire lasciando l'elaboratore in attività. Più spesso occorre fermare tutto e provvedere a sostituire la parte danneggiata. Questo è particolarmente grave per i grossi elaboratori che, di regola, sono utilizzati contemporaneamente da molti utenti. Costoro sarebbero tutti costretti a sospendere il proprio lavoro, con danni incalcolabili. Per questo abbiamo visto come ci si organizzi con elaboratori gemelli, che vanno avanti di pari passo finché tutto è regolare, mentre continua uno solo se l'altro si guasta. Così si può eseguire la riparazione, riallineare i due elaboratori e continuare, senza che nessuno avverta alcun disagio.

La fase di *riallineamento* dei due elaboratori può essere alquanto complessa. Infatti, mentre l'elaboratore guasto non funzionava, molti dati possono essere cambiati sui dischi dell'elaboratore sano, e sono questi cambiamenti che vanno riportati sulle memorie di massa di quello riparato. Ma un grosso elaboratore possiede migliaia di miliardi di informazioni su disco, e tutte non possono essere ricopiate in tempi ragionevoli. Per non dover ricopiare tutto quanto, però, occorre sapere esattamente dove sono stati apportati i cambiamenti dal momento del danno. Ripartendo dalla situazione e dai dati presenti al momento del guasto, e apportando tutti i cambiamenti avvenuti, si può far coincidere il nuovo stato dell'elaboratore riparato con quello dell'elaboratore che ha continuato a funzionare.

Per questo, ogni elaboratore lavora a *transazioni*, dove per transazione si intende un insieme completo di operazioni necessarie ad eseguire una singola funzionalità. Una transazione non si considera completa se non ha registrato, cioè reso permanenti, tutte le eventuali modifiche ai dati sui dischi effettuate dalle sue operazioni. In tal modo, una transazione o è completa, oppure è come se non fosse mai iniziata. Questo vale da un punto di vista logico, ma fisicamente, quando si verifica un guasto durante una transazione alcuni dati possono essere stati trascritti, altri no. Per tener conto di tutte le possibili situazioni, l'elaboratore registra una serie di informazioni su uno speciale archivio, detto *journal* o *log*, dal nome inglese del giornale, o diario, di bordo.

Due cose, soprattutto, vengono registrate sul journal: la transazione, come è stata richiesta dall'utente, e le copie dei settori che vengono man mano cambiati dalla transazione, giusto prima di subire il cambiamento. La transazione può essere, ad esempio, l'aggiornamento dei dati di un dipendente promosso. Sul *journal* viene registrata la richiesta dell'Ufficio del Personale con i nuovi dati da immettere. Col procedere dell'esecuzione della transazione, quando si deve modificare il contenuto di un settore, prima della modifica esso viene scritto (o salvato) sul journal, in quella che si dice la sua *immagine precedente*, o, con termine tecnico inglese, la *before image*.

Con tutti questi dati sul *journal*, vediamo cosa succede in caso di guasto e quindi di interruzione dell'attività dell'elaboratore. L'altro continua, intanto, a registrare sul journal tutto ciò che succede. Riparato il guasto, l'elaboratore torna indietro sul journal fino a trovare la transazione, durante la quale il guasto si era verificato. Questa transazione non era stata completata, ma parte dei dati da essa modificati potrebbero essere stati registrati. Si ristabiliscono allora all'indietro le *before image* per tornare alla situazione del disco, com'era prima dell'esecuzione della transazione. Si va all'indietro perché un settore potrebbe essere stato modificato più volte, e occorre ripristinare la situazione iniziale.

A questo punto, il disco è riportato allo stato in cui si trovava all'inizio della transazione durante la quale si è verificato l'errore. La transazione viene eseguita da capo e, automaticamente, dopo di lei, vengono eseguite la transazioni successive, quelle che non si erano potute eseguire a causa del guasto. Questa esecuzione, si osservi, è solo parziale, poiché, ad esempio, non occorre inviare i risultati all'utente, che già li ha avuti dall'altro elaboratore. Così, non è necessario aspettare il lento ingresso dei dati da parte dell'utente, in quanto sono già registrati sul journal. Si ha perciò un rapido aggiornamento e presto il ri-allineamento è completo.

Naturalmente, gli inconvenienti possibili sono molti, e vanno dalla mancanza improvvisa di corrente alla rottura fisica di un disco, al sabotaggio. Abbiamo descritto un metodo per minimizzare i disagi degli utenti, ma esso non è certo l'unico e non è quello che si possa applicare in qualunque situazione. Quello che possiamo auspicare è di essere riusciti a dare un'idea di come si cerchi di limitare i danni in situazioni che potrebbero diventare drammatiche, specie oggi, quando la vita e il lavoro dipendono così fortemente dal corretto funzionamento degli elaboratori elettronici, queste macchine così pervasive. Col che, chiudiamo anche questa lezione.

## 3.2 La privatezza

Il concetto di sicurezza dei dati che abbiamo trattato nella lezione precedente è in realtà solo un aspetto

dell'idea di sicurezza che di solito abbiamo. Il deterioramento dei dati, dovuto al tempo, all'usura o alle interferenze nelle trasmissioni, lede l'*integrità* delle nostre informazioni. Ma noi vorremmo che queste fossero anche al sicuro da possibili intrusioni da parte di estranei. Lasciando perdere la curiosità che certi nostri dati potrebbero suscitare, vi sono informazioni che noi riteniamo riservate, e che se venissero conosciute da altri potrebbero recare danno a noi stessi, alla nostra famiglia o alla nostra azienda. Tecnicamente, questi dati si dicono privati, e il fatto che noi non vogliamo che vengano a conoscenza se non di certe persone si qualifica come *privatezza*.

La privatezza è, accanto all'integrità, uno degli aspetti importanti della sicurezza delle informazioni. Sono stati perciò sviluppati vari metodi con lo scopo di assicurare la privatezza dei dati, che è uno dei diritti dei cittadini, della aziende e delle istituzioni, come stabilito per legge. I metodi utilizzati sono di varia natura, e si differenziano a seconda del grado di sicurezza a cui si desidera arrivare e a seconda del fatto che i dati da proteggere si trovino su un elaboratore (e che questo sia o non sia collegato in rete) oppure debbano essere trasmessi su una linea di comunicazione. Come si è visto per l'integrità, trasmettere dati è molto più pericoloso che memorizzarli.

Quando le informazioni private sono contenute su un personal al quale avete accesso solo voi, ad esempio è il vostro portatile, un metodo semplice di protezione è quello di avere una chiave materiale per l'accensione. Di solito esiste un bottone per accendere l'elaboratore, ma è possibile munirlo di un *interruttore a chiave*, così che solo colui che possiede la chiave può accenderlo. Questo è sufficiente per assicurare una normale privatezza alle informazioni, e certo i semplici curiosi saranno scoraggiati dal tentare azioni che potrebbero configurarsi come criminose, quale rubarvi le chiavi. Se i vostri dati, invece, meritassero tali azioni, sarà bene che vengano protetti con i metodi più sofisticati che ora vedremo.

Se non volete munire il vostro elaboratore di una chiave da portarvi in tasca, potete ricorrere alle così dette *parole segrete* o *password*, con un termine inglese ormai entrato nell'uso informatico. Quando l'elaboratore si accende, chiede all'utente di identificarsi, cioè di fargli capire che è proprio lui il legittimo proprietario delle informazioni contenute nelle sue memore. Di solito, l'identificazione avviene digitando il proprio nome e una password, cioè la parola segreta scelta dall'utente e che solo lui conosce, o dovrebbe conoscere. Questa parola viene mascherata affinché non possa essere sbirciata da occhi indiscreti, e se è quella corretta, memorizzata all'interno dell'elaboratore, questo riconosce il suo proprietario e inizia a lavorare. In caso contrario si blocca e non va avanti.

Il concetto di *password* è oggi diventato molto comune e tutti usiamo il PIN per il nostro cellulare e per il Bancomat. Questi apparecchi hanno un doppio controllo, e se sbagliate tre volte consecutive, tutto il meccanismo si interrompe e occorre eseguire qualche procedura di emergenza per poter ripartire. Questo permette di avere password corte, di quattro o cinque cifre. Negli elaboratori, invece, si una una *password* più lunga, che rende praticamente impossibile tentare a caso o provare ad inserire tutte le combinazioni possibili, fino a trovare quella giusta. Certi sistemi generano e provano tutte le *password* possibili, e quindi occorre che esse siano decine di miliardi per rendere non conveniente l'uso di tali procedure, che dovrebbero impiegare anni per scoprire quella corretta.

Poiché andare a caso nel tentare di indovinare una *password* può essere troppo laborioso, le spie provano quelle che, statisticamente, sono più gettonate. Per questo motivo, si sconsiglia di usare come parola segreta (che tanto segreta non sarebbe) la data di nascita, l'indirizzo di casa o il nome della fidanzata, almeno di non cambiarla molto spesso (la fidanzata o il fidanzato, si intende). Anche la pratica di cambiare spesso la *password* è da seguire, poiché non si sa mai se possa essere stata individuata casualmente. Infine, è bene non scriverla su un pezzo di carta, poiché questo potrebbe essere adocchiato e il segreto presto scoperto.

La tecnica della *password* non è solo legata all'accensione dell'elaboratore. In realtà, quando memorizziamo un qualsiasi archivio, programma o insieme di dati che sia, possiamo associargli una parola segreta, anche diversa da archivio ad archivio. Quando noi stessi, o chiunque intenda fare una cosa del genere, prova ad aprire tale archivio, l'elaboratore chiede la password associata, e se non si è in grado di fornirla, l'archivio rimane chiuso ed è impossibile vedere cosa contenga. Naturalmente, non lo si può neppure modificare, così che il suo contenuto rimane integro e segreto. Il problema è di usare password che siano facili da ricordare, ma difficili da immaginare. Anche in questo caso ci si dovrebbe astenere da riportare le password su foglietti, e il problema per chi ha poca memoria è serio.

La protezione della privatezza dei dati mediante *password* è largamente usata, ma ha un inconveniente molto grave, quando le informazioni debbano essere mantenute davvero segrete o quando le si debbano trasmettere attraverso una linea di comunicazione. Se avete messo sul disco del vostro personal la formula della lozione per far ricrescere i capelli, un concorrente sleale può essere tentato di carpirvi il segreto. Se ignora la *password* di accesso, può ben rompere il vostro personal e portarvi via lo *hard-disk*.

Con calma poi metterà in azione i propri informatici perché vadano a vedere cosa c'è scritto sul disco. Per un esperto, non è difficile scorrere il contenuto del disco e trovare la vostra formula, visto che da qualche parte deve pur essere registrata. E purtroppo, il codice ASCII e l'UNICODE sono noti a tutti!

Altrettanto pericolosa, e forse più, è la trasmissione dei dati. Non ci vuole molto, ad una spia, ad intercettare una trasmissione via cavo, e per le trasmissioni via radio (o, come si dice, *wireless*) il gioco è da bambini, visto che in tal caso i dati vagano liberi per l'etere. Il problema consiste ancora nel fatto che le informazioni segrete sono in chiaro, e basta saper leggere, o ascoltare, e a un certo punto ce le troviamo davanti. La soluzione, allora è quella di registrarle e di trasmetterle non così come sono, ma dopo averle mascherate opportunamente.

Da un punto di vista tecnico, questo mascheramento si dice *ciframento* o *cifratura*, e la disciplina che studia i metodi di ciframento (o i modi di romperli) si dice *crittologia*. Il termine *crittografia* è più o meno sinonimo di cifratura e le regole per eseguire la cifratura si dicono di solito cifra o codice. Sicuramente ci si è serviti di metodi criptografici fin dall'antichità, ma il codice più vecchio che si conosca risale a Cesare. Durante le guerre galliche, egli usava un codice molto semplice, che consisteva nello scrivere, al posto di ogni lettera, quella che si trova nell'alfabeto quattro posizioni più avanti. Chi riceveva il messaggio cifrato, doveva semplicemente tornare quattro posizioni indietro. L'alfabeto era considerato circolare, così che la A veniva a seguire la Z e la codifica di Z era la lettera D:

```
GALLIA EST OMNIS DIVISA ...
MEPPOE IZA SQROZ HOCOZE ...
```

(Abbiamo usato l'alfabeto italiano; i romani usavano le lettere K, X ed Y mentre ignoravano la U e la W e identificavano I e J).

Il metodo si può generalizzare e, al posto di ogni lettera, scrivere quella $k$ posizioni più avanti, o più indietro, nell'alfabeto. Tutti ricordiamo che l'elaboratore ribelle in *2001, Odissea nello Spazio* si chiama HAL 9000. Se si prende la lettera successiva, dalla parola HAL si ottiene IBM, e questo spiega l'adozione di questo nome. La crittografia di Cesare si rompe molto facilmente. Anche se non si conosce $k$, si prova prima con la lettera successiva, poi con quella due posizioni più avanti, e così via. Dopo al più 26 prove (o, in italiano, 21) la chiave, cioè il $k$ che costituisce la cifra del metodo, deve per forza venir fuori. Questo fa sì che, oggi, il metodo di Cesare non lo usa più nessuno.

Nella crittografia di Cesare, a lettere uguali corrispondono lettere uguali. Altri semplici metodi di cifratura consistono nel creare una corrispondenza biunivoca arbitraria fra lettere, mantenendo la regola che a lettere uguali corrispondono simboli uguali. Questo è il metodo usato nel biglietto misterioso che il protagonista dello *Scarabeo d'Oro* trova sulla spiaggia. Ed Edgar Allan Poe ci dice come si rompe un codice del genere, certo più complesso di quello di Cesare, poiché coinvolge una scelta arbitraria della corrispondenza tra lettere (o tra lettere e simboli). Il metodo è statistico, e si basa sulla frequenza dei vari caratteri nella lingua in cui il testo originale è scritto. Nello *Scarabeo d'Oro* tale lingua è l'inglese (vedere Tabella 3.1).

| | | | |
|---|---|---|---|
| E | 1268 | F | 256 |
| T | 978 | M | 244 |
| A | 788 | W | 214 |
| O | 776 | Y | 202 |
| I | 707 | G | 187 |
| N | 706 | P | 186 |
| S | 634 | B | 156 |
| R | 594 | V | 102 |
| H | 573 | K | 60 |
| L | 394 | X | 16 |
| D | 389 | J | 10 |
| U | 280 | Q | 9 |
| C | 268 | Z | 6 |

Tabella 3.1: Frequenza delle lettere nei testi inglesi (totale $10{\cdot}003$)

Il testo è il seguente:

```
53++!305))6*;4826)4+.)4+);806*;48!8'60))
85;]8*:+*8!83(88)5*!;46(;88*96*?;8)*+(;485
);5*!2:*+(;4956*2(5*-4)8'8*;4069285);)6!8)
4++;1(+9;48081;8:8+1;48!85;4)485!528806*81
(+9;48;(88;4(+?34;48)4+;161;:188;+?;
```

e il protagonista conta la frequenza dei vari simboli. "8" si trova 33 volte, ";" è presente 26 volte e "4" 19 volte. Gli altri simboli hanno frequenze più basse.

In inglese, la lettera più frequente è la E, seguita dalla T e dalla A. Così, il protagonista del racconto prova a far corrispondere la E, la T e la A ai tre simboli più frequenti del biglietto, e osserva qual è la situazione. In effetti, a questo punto è abbastanza facile accorgersi della presenza dell'articolo, che in inglese esiste solo nella forma THE, rivelando il codice della H. Procedendo con lo stesso spirito, il messaggio è presto decifrato. Il metodo delineato da Poe è proprio quello da usare in generale, e anche questo metodo di cifratura si rompe troppo facilmente per essere adottato in pratica.

Un altro cultore di crittografia fu Augusto. La sua idea è molto interessante. Si consideri come chiave un testo molto lungo; Augusto aveva scelto il primo

canto dell'Iliade "Cantami o Diva del Pelide Achille ...", naturalmente in greco, visto che questa era la sua lingua preferita. Questo testo veniva sottoscritto al messaggio da cifrare e, per ogni lettera di questo, si avanzava nell'alfabeto di quante posizioni erano indicate dalla lettera della chiave. Così, la D avanza di tre posizioni e diventa G, la O avanza di una posizione e diviene P, e così via. Il messaggio, modificato in tale maniera, è quello che viene registrato o trasmesso.

```
DOMANI ATTACCHEREMO ...
CANTAMI O DIVA DEL P...
GPBUOVIALTENBIEVLZO ...
```

Come mostra l'esempio, a lettere uguali corrispondono, di solito, lettere diverse. In questo modo, vengono a perdersi le corrispondenti statistiche sulle frequenze, che rendevano poco sicuro il codice di Cesare. Come si vede, o si immagina facilmente, ad ogni lettera può corrispondere un'altra qualsiasi, senza alcuna preferenza, se si suppone che il testo usato come chiave non abbia una struttura particolare. In pratica, questo codice si può rompere soltanto venendo a conoscenza della chiave, il che dipende solo da chi la possiede. Il codice di Augusto è considerato sicuro, tanto sicuro che su di esso si sono basate, a suo tempo, le trasmissioni tra la Casa Bianca e il Kremlino, sul celebre telefono rosso, all'epoca della Guerra Fredda.

Finché i messaggi segreti devono essere scambiati tra due soli utenti, il metodo di Augusto è perfettamente adeguato, anche se è consigliabile cambiare spesso la chiave di ciframento. Se le persone che vogliono scambiarsi messaggi sono tre, diciamo A, B, C, la chiave che usano A e B non può essere la medesima di quella usata da A e C, perché altrimenti fra A, B, C non ci sarebbero segreti, in quanto ognuno conosce la chiave di codifica. Di solito, questo non è il caso, ed A e B vogliono scambiarsi messaggi senza che C li possa venire a conoscere. E lo stesso deve avvenire tra A e C, e tra B e C. Si richiedono pertanto tre chiavi distinte. Purtroppo, se $N$ sono gli utenti, le coppie possibili che si scambiano le informazioni sono:

$$\frac{N(N-1)}{2} \approx \frac{N^2}{2}.$$

Se gli utenti della rete fossero anche solo un milione, ci vorrebbero cinquecento mila miliardi di chiavi diverse, una cifra davvero esorbitante. E, come si sa, oggi gli utenti di Internet sono centinaia di milioni, il che rende del tutto impossibile un sistema di chiavi ragionevolmente sicuro. D'altra parte, più sono gli utenti della rete, più è necessario assicurare loro che, quando serve, le informazioni che scambiano con gli altri utenti rimangano private. Si pensi ai dati che passano tra impiegati o dirigenti di un'azienda, alle nostre operazioni con la Banca, al commercio elettronico, che deve assicurare al compratore la segretezza dei propri dati e all'organizzazione che vende, così come allo stesso compratore, il riconoscimento e il rispetto del contratto di vendita.

Molti altri metodi crittografici sono stati proposti dopo Augusto e alcuni portano nomi illustri, come quello di Richelieu. Anche nel 1800 e nel 1900 sono state proposto nuove tecniche di cifratura, ma senza cambiare il concetto basilare di chiave e senza cambiare il grado di sicurezza. Tutti questi codici prevedono una chiave di cifratura che è strettamente legata a quella usata per decifrare il messaggio, così che conoscendo l'una si conosce immediatamente anche l'altra. Per questo motivo, tutti questi metodi sono detti *immetrici* e proprio per questo creano problemi di numerosità quando vengano applicati alle nostre mostruose reti di elaboratori. Finalmente, nella prima metà degli anni 1970, vennero inventati metodi diversi, detti *asimmetrici*, poiché la conoscenza della chiave di codifica non implica la conoscenza della chiave di decodifica, e viceversa.

L'idea è, in apparenza, relativamente semplice. Sia $M$ un messaggio che un utente $U$ vuole spedire ad un altro utente $V$. Quest'ultimo possiede due chiavi. La prima, detta $p$, è *pubblica*, nel senso che è reperibile in rete ed è quindi a disposizione di tutti. La seconda, detta $s$, è *segreta*, cioè è nota solo all'utente $V$. Le due chiavi sono correlate tra loro, e teoricamente, nota l'una, esiste un metodo noto a tutti che permetterebbe di trovare l'altra, e viceversa. Tuttavia, tale metodo è così maledettamente complicato che anche il più veloce degli elaboratori impiegherebbe qualche centinaio d'anni per ricavare $s$ da $p$, o $p$ da $s$. Questo fa sì che sia praticamente impossibile ricavare $s$, anche se $p$ è disponibile a tutti.

L'utente $U$, una volta procuratasi la chiave pubblica $p$ di $V$, codifica il messaggio $M$ usando tale chiave, e ottiene un messaggio $M'$ che spedisce in rete a $V$. Costui applica la procedura di decodifica usando la propria chiave segreta $s$, e trasforma il messaggio $M'$ nel messaggio originale $M$. Questa decodifica è effettuata quando il messaggio $M'$ è già arrivato all'elaboratore di $V$, così che $M$ è disponibile solo a lui nella sua forma originale. Se un terzo utente $W$ riesce a carpire $M'$ dalla rete, non riuscirà comunque a leggerlo in chiaro, visto che non conosce la chiave segreta $s$. Se $V$ non rivela la sua chiave $s$ e non se la fa scoprire in qualche modo, questo metodo è praticamente sicuro e in circa trent'anni di applicazione nessuno è mai riuscito a romperlo.

Il punto cruciale in questo metodo sta nel trovare le due chiavi $p$ ed $s$. Rivest, a metà degli anni 1970, sviluppò a questo scopo una teoria algebrica. Dato un numero $n$ che sia il prodotto di due numeri primi

$a$ e $b$, le chiavi $p$ ed $s$ sono scelte in modo che la quantità $(ps-1)$ sia divisibile per $(a-1)(b-1)$. Se, come succede, $a$ e $b$ sono numeri con più di cento cifre decimali, questo si riesce a fare in pochi secondi. Questa scelta è motivata dal fatto che se suddividiamo $M$ in tanti blocchi che, interpretati come numeri, siano numeri decimali con circa 200 cifre, trattati col numero $p$ danno luogo ai blocchi del messaggio cifrato $M'$. A loro volta, questi generano di nuovo il messaggio originale $M$ quando siano trattati con la chiave $s$.

Le operazioni di codifica e di decodifica sono pesanti e possono richiedere diversi secondi. Si usano allora tecniche miste: $U$ spedisce a $V$, col metodo detto, una chiave $K$ da usare con la crittografia di Augusto; spedisce poi il suo messaggio usando la chiave $K$, così che $V$ possa decodificarlo velocemente. Ogni volta, se vuole, $U$ può cambiare la chiave $K$, così che il metodo di Augusto risulta del tutto sicuro, ed è più conveniente perché più veloce sia in fase di codifica sia in quella di decodifica. Il procedimento è laborioso, ma per fortuna è completamente portato avanti dagli elaboratori di $U$ e di $V$, di modo che i due utenti quasi non si accorgono di niente e, soprattutto, nulla devono fare se non spedire e leggere il messaggio segreto.

Il metodo, così superficialmente descritto, è detto di ciframento a *chiave pubblica* con riferimento a $p$. Si tratta di un metodo asimmetrico, visto che, in pratica, la conoscenza di una delle due chiavi $p$ o $s$ non permette di conoscere l'altra. Questo porta un vantaggio immediato e molto importante. Ogni utente ha bisogno di due chiavi, quella pubblica e quella segreta. Un milione di utenti, perciò, necessita in tutto di due milioni di chiavi, un bel po' di meno di quei 500 miliardi di chiavi da usare con i metodi di crittografia simmetrica. Ormai, la crittografia a chiave pubblica è entrata nel novero delle applicazioni standard che si trovano su tutti gli elaboratori, personal compresi.

Un altro aspetto importante dei metodi di ciframento a chiave pubblica è che, con la loro adozione, è possibile affrontare un problema cruciale in molte applicazioni, soprattutto in quelle che concernono certificazioni o movimento di denaro. Supponiamo che $U$ debba inviare a $V$ la ricevuta per una somma riscossa. Non basta che gli mandi un semplice messaggio, ma $V$ vuole una firma da parte di $U$, così da poter dimostrare legalmente che $U$ ha ricevuto il suo denaro e nulla ha più da pretendere. Si pone allora il problema della *firma digitale*, un metodo che permetta legalmente (e in modo sicuro) di far ammettere ad $U$: questo messaggio l'ho spedito proprio io, e, nello stesso tempo, di far dire a $V$: questo messaggio l'ha spedito $U$ e non può negare di averlo spedito proprio lui.

La procedura da seguire è, almeno in linea di principio, la seguente. Sia $M$ il testo della ricevuta che $U$ deve spedire a $V$. Questa volta dobbiamo indicare con $p_V$, $s_V$ le chiavi di $V$, e con $p_U$, $s_U$ le chiavi di $U$, poiché servono pure loro. Un punto importante del metodo a chiave pubblica è che codificare con $p$ e poi decodificare con $s$ dà il messaggio in chiaro, come codificare con $s$ e poi decodificare con $p$. Questa è una regola generale della crittografia. Allora, $U$ codifica $M$ con la propria chiave segreta $s_U$ ottenendo $M_U$; quindi codifica questo messaggio con la chiave pubblica $p_V$ di $V$ ottenendo $M'$. Spedisce infine questo messaggio in rete affinché arrivi all'utente $V$. Questi perciò riceve il messaggio $M'$ con la doppia codifica.

$V$ comincia a decodificare $M'$ con la propria chiave segreta $s_V$ e questo riporta il messaggio a prima dell'ultima codifica, cioè ad $M_U$. $V$ decodifica $M_U$ usando la chiave pubblica $p_U$ di $U$, disponibile a tutti, e questo trasforma $M_U$ in $M$, il messaggio in chiaro, che così può essere letto da $V$. Ma attenzione! Codificando $M$ con la sua chiave segreta $s_U$, $U$ ha firmato il messaggio, poiché nessuno, se non lui, conosce $s_U$. E, d'altra parte, se il messaggio non fosse stato codificato con $s_U$, non sarebbe stato possibile per $V$ decodificarlo con $p_U$. Pertanto, se $V$ è riuscito a leggere il messaggio, sa che l'unica persona che gliel'ha potuto spedire è $U$. Questa procedura, perciò, ha lo stesso ruolo di una firma, purché $U$ e $V$ siano gli unici a conoscere le proprie chiavi segrete e l'Ente che le distribuisce sia sufficientemente serio.

In effetti, esistono oggi alcune istituzioni che, dietro pagamento, si sono assunte il compito di distribuire e certificare le chiavi. Un utente $V$ si rivolge a una tale istituzione che provvede a generare $p$ ed $s$. La chiave segreta $s$ è resa nota solo a $V$, mentre la chiave pubblica $p$ è messa a disposizione di tutti in un elenco di chiavi, analogo all'elenco telefonico (o giù di lì). Quando $U$ vuole spedire un messaggio segreto a $V$, cerca in questo elenco la sua chiave pubblica e con essa codifica il messaggio. In tal modo, questo potrà essere letto solo da $V$. La serietà di questi Enti, che potrebbero leggere tutti i messaggi, è assicurata (più o meno) dal fatto che, se ne leggessero anche solo uno e la cosa si risapesse, la loro reputazione si annullerebbe ed essi fallirebbero miseramente.

## 3.3 La compressione dei dati

Oggi la quantità dei dati che possono essere memorizzati all'interno di un elaboratore è enorme e si misura in decine di gigabyte per i personal e in terabyte per i main frame. Naturalmente non esistono praticamente limiti alle memoria intercambiabili, come i CD e i DVD. D'altra parte, come si suol dire, la fame vien mangiando e si cerca di immagazzinare sempre

più dati in spazi sempre più stretti. Questo desiderio di compressione è alimentato anche dal fatto che molte informazioni viaggiano su rete, il che è molto costoso. Comprimere i propri dati vuol dire, allora, abbreviare i tempi di trasmissione e quindi spendere di meno. Ovvero, allo stesso costo, trasmettere molte informazioni in più.

In effetti, comprimere i dati può significare due cose: trovare apparecchiature più piccole e/o più capienti, oppure inventare tecniche che, eliminando la ridondanza dai dati, permettano di memorizzare le stesse informazioni usando meno bit. Il primo è un aspetto tecnologico, che cerca di rendere i bit fisici sempre più piccoli, ma non cambia il loro numero per dato. Abbiamo visto come il floppy e l'hard disk memorizzino i bit nello stesso modo, ma lo hard disk, più preciso, usa punti di magnetizzazione più piccoli, e così registra più dati nella stessa quantità di spazio. Più o meno la stessa cosa avviene nel passaggio da CD a DVD. Noi però non vogliamo entrare in queste questioni di natura fisica e ingegneristica, e ci dedicheremo ora all'altro aspetto.

Come abbiamo osservato, il linguaggio naturale presenta una buona dose di ridondanza, e possiamo togliere varie lettere a una frase, senza per questo impedirne la comprensione:

```
Questa frase si capisce anche se si
          cancella qualcosa
Q e ta fr se s c pisc an he s si c ncel a
                qu lc sa
    Qst frs s kpsc ank se s kncll qlks
```

Ciò è sfruttato dalla stenografia, ma possiamo far sì che anche l'elaboratore ne tragga vantaggio. Abbiamo ricordato che talvolta si introduce una certa ridondanza nei dati per segnalare o correggere automaticamente certi errori. Ma si tratta di un'altra storia, e ciò che realmente si fa sull'elaboratore è questo. Si toglie, per quanto possibile, tutta la ridondanza presente e poi si aggiunge una ridondanza artificiale per controllare gli errori. Sembra un po' assurdo, ma si opera così per sfruttare le capacità dell'elaboratore e comprimere in modo sistematico, efficace ed efficiente.

Riprendiamo la scansione di una pagina di testo. Come s'è visto, questa viene codificata come una sequenza di bit 0, per bianco, e di bit 1, per nero. A 300 dpi si hanno in tutto 2480 punti per linea. Una linea bianca corrisponde perciò a 2480 bit tutti 0. Poiché per scrivere il numero 2480 sono sufficienti 12 bit: $2480_{10} = 100110110000_2$, basterebbe spedire il numero 2480 (eventualmente seguito dall'indicazione che si tratta di bit 0) piuttosto che 2480 bit tutti 0. In effetti, sequenze di bit bianchi si succedono a sequenze di bit neri, e quindi può convenire spedire le lunghezze di queste sequenze, piuttosto che le sequenze stesse. Questo è il metodo di compressione adottato dal FAX e si dice *Run Length Encoding*, cioè codifica mediante la lunghezza delle sequenze.

| | |
|---|---|
| 000000000000000000000000 | 24 |
| 000000000111111000000000 | 9,6,9 |
| 000000111111111111000000 | 6,12,6 |
| 000001110000000011100000 | 5,3,8,3,5 |
| 000011000000000000110000 | 4,2,12,2,4 |
| 000110000000000000011000 | 3,2,14,2,3 |
| 000110000000000000011000 | 3,2,14,2,3 |
| 001100000000000000001100 | 2,2,16,2,2 |
| 001100000000000000001100 | 2,2,16,2,2 |
| 001100000000000000001100 | 2,2,16,2,2 |
| 001100000000000000001100 | 2,2,16,2,2 |
| 001100000000000000001100 | 2,2,16,2,2 |
| 001100000000000000001100 | 2,2,16,2,2 |
| 000110000000000000011000 | 3,2,14,2,3 |
| 000110000000000000011000 | 3,2,14,2,3 |
| 000011000000000000110000 | 4,2,12,2,4 |
| 000001110000000011100000 | 5,3,8,3,5 |
| 000000111111111111000000 | 6,12,6 |
| 000000000111111000000000 | 9,6,9 |
| 000000000000000000000000 | 24 |

Figura 3.1: La lettera "O"

Tecnicamente funziona così. Man mano che lo scanner rileva il contenuto di un foglio, conta le sequenze di bit uguali e le memorizza come numeri di 12 bit. Questo è sufficiente linea per linea. Convenzionalmente, la prima sequenza si suppone bianca e nel caso che sia nera si mette a 0 la prima lunghezza. Le sequenze bianche e nere si succedono, così che i numeri in posizione dispari si riferiscono al bianco (bit 0) e quelle pari al nero (bit 1). Quando termina la scansione di una linea, la sequenza dei numeri viene memorizzata e solo quando il foglio è completo viene spedito per via telefonica. Nella Figura 3.1 mostriamo un piccolo esempio relativo alla lettera "O".

In ricezione, il terminale FAX accetta i vari numeri e li interpreta come lunghezza delle sequenze: la prima di bit 0, la seconda di bit 1, la terza di nuovo di bit 0, e così via. Quando arriva ad aver accumulato 2480 bit, cioè quelli di una linea, sa che questa è completa, ed è pronto a stamparla. In realtà più linee vengono accumulate, a seconda delle possibilità del terminale, e stampate poi contemporaneamente. Questo metodo permette una buona compressione. Se, ad esempio, il foglio è quasi del tutto vuoto, esso si può comprimere in poche centinaia di byte, contro il milione e centomila byte risultanti dalla scansione punto per punto. Di solito si hanno vantaggi un po' più contenuti, ma di regola molto interessanti.

Il Run Length Encoding, purtroppo, va bene so-

lo per documenti in bianco e nero e riconducibili a sequenze di lunghezza non troppo corta. Altrimenti, la codifica delle lunghezze verrebbe ad occupare più spazio delle sequenze memorizzate come tali. Vedremo più avanti cosa si può fare per le immagini a colori. Per i testi e i documenti scritti in ASCII si usano altre tecniche legate alla rappresentazione dei caratteri.

Nella codifica ASCII tutti i simboli corrispondono a un byte, cioè ad otto bit. Tuttavia, una facile osservazione è che, ad esempio, in un testo italiano, non tutti i caratteri hanno la stessa probabilità di apparire. La A è molto frequente mentre, all'opposto, la F è abbastanza rara, per non parlare delle lettere straniera J, K, W, X ed Y. Si veda anche la Tabella 3.1 per la frequenza delle lettere nei testi di lingua inglese. Un'idea semplice è allora quella di usare codici di lunghezza diversa per i vari caratteri: codici brevi per lettere frequenti, codici lunghi per lettere rare. In questo modo si capisce bene che l'occupazione si riduce, anche in modo drastico.

Il punto importante è quello di fare la cosa in modo razionale e, soprattutto, in modo da poter ricostruire il testo nella sua interezza, in ASCII per poterlo leggere facilmente. Questo aspetto non è banale poiché, se ogni carattere ha una codifica di lunghezza diversa, non sappiamo quando i bit di un simbolo sono terminati e comincia il carattere successivo:

$$A = 010, B = 1101, C = 0101, E = 101$$

$$AC = 0101101 = CE.$$

Occorre imporre qualche condizione sulla codifica, e ciò è quanto fece Huffman nel 1952, quando propose il suo metodo per la compressione dei testi. Si dimostra nella *Teoria dell'Informazione* che tale metodo è il migliore fra tutti quelli che usano una codifica dei caratteri a lunghezza variabile. L'idea di Huffman fu quella di stabilire che nessun codice potesse essere la parte iniziale, o *prefisso*, di un altro codice. L'esempio negativo precedente viola questo vincolo ed A è un prefisso di C. Ma se la condizione di Huffman è verificata, non ci possiamo mai sbagliare a decodificare un testo. Stabilito dove comincia la codifica di un carattere, andiamo avanti fino a che non sia completa la codifica di un qualche simbolo. Per la proprietà del prefisso, questo è l'unico codice completo, e quindi possiamo tranquillamente decodificare il carattere. Inoltre, in modo automatico, abbiamo trovato da dove comincia il codice del carattere successivo.

Vediamo come funziona formalmente il metodo, considerando una frase, che abbiamo scelto in modo da avere caratteri più e meno frequenti:

`LA MAMMA AMMANNISCE MAMMOLE`

Calcoliamo la frequenza di ciascuno di essi, compreso lo spazio (indicato dal simbolo $\sqcup$):

$$M = 8, A = 6, \sqcup = 3, N = 2, L = 2,$$

$$E = 2, S = 1, O = 1, I = 1, C = 1.$$

Disponiamoli ora in sequenza, come nella Figura 3.2. L'ordine è arbitrario, e abbiamo scelto semplicemente quello che ci dà una figura meno arruffata. Prendiamo ora le due frequenze più basse, o due fra le frequenze più basse, e mettiamole assieme. Continuiamo nella stessa maniera unendo sempre le due frequenze più piccole:

$$\begin{array}{lll} \zeta = (C, I) & \eta = (O, S) & \theta = (E, L) \\ \delta = (\zeta, \eta) & \kappa = (N, \sqcup) & \gamma = (\delta, \theta) \\ \epsilon = (\kappa, A) & \beta = (\gamma, M) & \alpha = (\epsilon, \beta) \end{array}$$

Si ottiene il cosiddetto *albero di Huffman*, che completiamo mettendo un'etichetta 0 o 1 a tutte le connessioni che partono da ciascun nodo ovale. Il ruolo delle lettera greche sarà chiarito tra breve; se ora seguiamo i cammini che, partendo dal nodo $\alpha = 27$, vanno ai vari caratteri, se ne ottiene la codifica:

| *car.* | *freq.* | *codice* | | *car.* | *freq.* | *codice* |
|---|---|---|---|---|---|---|
| $M$ | 8 | 01 | | $E$ | 2 | 0010 |
| $A$ | 6 | 11 | | $S$ | 1 | 00011 |
| $\sqcup$ | 3 | 101 | | $O$ | 1 | 00010 |
| $N$ | 2 | 100 | | $I$ | 1 | 00001 |
| $L$ | 2 | 0011 | | $C$ | 1 | 00000 |

Non rimane che codificare il nostro testo:

0011111010111010111101110101111100...

Si ha così una sequenza di 79 bit, come si vede facendo i conti dell'occupazione basati sulle varie frequenze:

$$(8 \times 2) + (6 \times 2) + (3 \times 3) + (2 \times 3) + (2 \times 4) +$$

$$+(2 \times 4) + 5 + 5 + 5 + 5 = 79.$$

Il risparmio che si ottiene, si calcola valutando l'occupazione della codifica a lunghezza costante. Per essere onesti, osserviamo che abbiamo dieci caratteri distinti, per cui bastano quattro bit per carattere. In totale, i 27 caratteri del testo occupano 108 bit e il risparmio è di 25 bit, cioè di circa il 24%.

La decodifica del testo è molto semplice, data la proprietà del prefisso, imposta dalla struttura dell'albero. Esso impedisce che qualunque codice possa essere la parte iniziale di un altro. Infatti, ogni codice è un cammino completo dal nodo radice 27 al carattere, e tale cammino è unico. La `L` iniziale, ha un codice univocamente identificabile, ma la decodifica può essere resa automatica. Se associamo l'etichetta di ogni nodo ad una lettera greca (o in qualunque

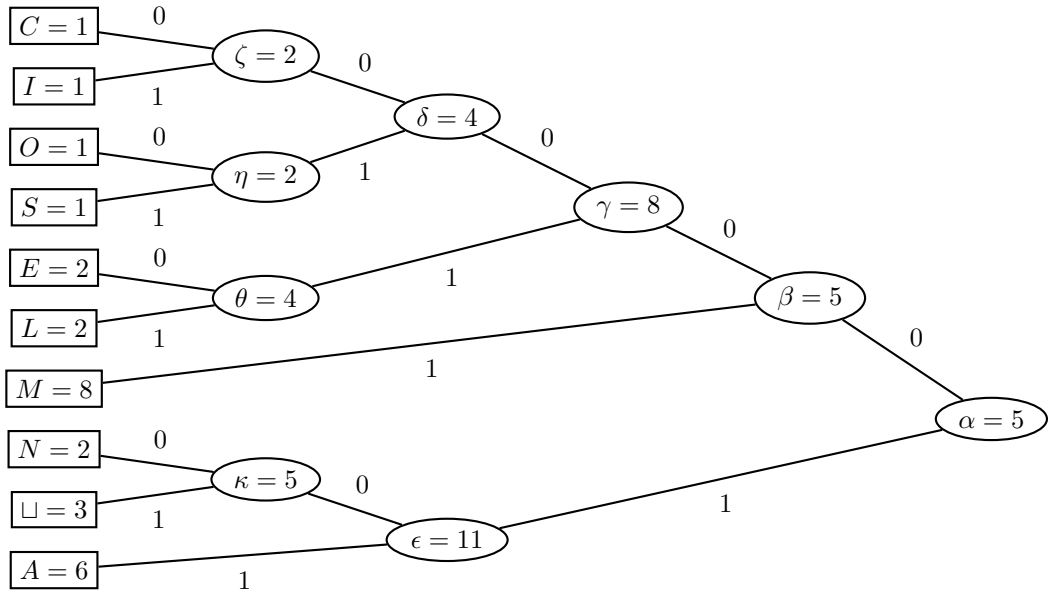



Figura 3.2: L'albero di Huffman

altro simbolo, che diremo *stato*) l'albero si trasforma in un cosiddetto *automa a stati finiti* (si veda ancora la Figura 3.2).

| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | . |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\alpha$ | $\beta$ | $\gamma$ | $\theta$ | $\alpha$ | $\epsilon$ | $\alpha$ | $\epsilon$ | $\kappa$ | $\alpha$ | $\beta$ | $\alpha$ | $\epsilon$ | . |
| | | | $L$ | | $A$ | | | $\sqcup$ | | $M$ | | $A$ | . |

Se scriviamo lo stato iniziale $\alpha$ sotto il primo bit, usiamo tale stato e il bit 0 che gli sta sopra per passare, secondo l'albero/automa, allo stato successivo $\beta$. Questo stato, usando il bit 0 che lo sovrasta, ci porta allo stato $\gamma$. Da questo e dal bit 1, si passa allo stato $\theta$, e infine si arriva alla lettera latina `L`. Questa è la decodifica dei primi quattro bit e si riparte ora dallo stato $\alpha$, da inserire sotto il quinto bit 1. Questo ci porta allo stato $\epsilon$, e da qui alla lettera `A`. Così procedendo si decodifica tutto il testo, in modo completamente meccanico.

Immaginiamo di dove spedire, sulla linea di comunicazione, il nostro testo sulla mamma. Il risparmio del 24% è effettivo, ma allo stesso tempo fittizio. Infatti, chi riceve il testo, deve anche avere l'albero/automa per poterlo decodificare. Esso, perciò, deve essere spedito col testo, e occuperà ben di più dei 25 bit risparmiati. Cos, in realtà, questa codifica è meno compressa del testo inviato in chiaro. Fortunatamente, questo inconveniente è dovuto solo al fatto che il nostro testo è molto corto. Un messaggio molto più lungo avrebbe bisogno di un albero poco più grande del nostro, e quindi lo spazio da esso occupato sarebbe irrilevante rispetto a tutto il testo. E' questa osservazione che, in effetti, rende conveniente il metodo di Huffman.

Per evitare di dover comunicare, insieme al testo, anche l'albero/automa, talvolta si usano codifiche standard. Ad esempio, per i testi in inglese, basta utilizzare un albero costruito con le frequenza standard delle varie lettere nei testi inglesi. Queste frequenze sono state calcolate per le varie lingua, anche in funzione del particolare tipo di testo: libro di narrativa, articolo di giornale, testo scientifico, e così via. Il particolare testo da comprimere non corrisponderà con precisione a uno specifico albero, ma le differenze saranno così esigue che ben poco si perderà nella fase di compressione. Specificando semplicemente l'albero usato, daremo al riceventi l'opportunità di usare lo stesso albero, memorizzato sul suo elaboratore. Evitiamo perciò una trasmissione relativamente costosa.

Come abbiamo accennato, il metodo di Huffman è ottimo rispetto ai metodi che usano una codifica variabile per i singoli caratteri. Nella seconda metà degli anni 1970, furono inventati metodi di compressione che sfruttano un altro principio. I codici hanno tutti la medesima lunghezza, tipicamente 12 bit, ma un codice può rappresentare una sequenza di 1, 2, 3 o anche più caratteri. Se tali sequenze di lunghezza maggiore di 1 sono frequenti, questi metodi permettono di comprimere in modo sostanzioso. Si pensi alla sequenza "CHE " di quattro caratteri. Nella codifica ASCII occupa 32 bit, e quindi una codifica di 12 bit è molto conveniente. In effetti, la sequenza è assai frequente, e ricorre sia per il pronome relativo, sia per la congiunzione, sia come finale di molti sostantivi e di molti aggettivi, come MARCHE, POCHE, MISTICHE e così via.

Il problema, come per Huffman, è quello di trovare

un metodo sistematico per cercare le sequenze più frequenti, cioè un algoritmo per realizzare il compattamento. Due ricercatori israeliani, Lempel e Ziv, proposero vari algoritmi e uno di essi, modificato da un terzo ricercatore, Welch, è stato ampiamente utilizzato, anche in prodotti commerciali di compressione, come il celebre ZIP. Dalle iniziali dei suoi inventori, il metodo è detto LZW. Esso costruisce un dizionario delle sequenze frequenti, che cresce man mano che il testo da comprimere viene scandito e codificato. Questo dizionario contiene l'associazione tra sequenze e codici relativi.

Per dare un'idea di come funziona il metodo, ricordiamo che il dizionario contiene inizialmente tutti i caratteri a disposizione, ad esempio i 256 simboli della codifica ASCII. Man mano che si incontrano, si aggiungono al dizionario le coppie di caratteri, ignorando di conseguenza le coppie impossibili, come in italiano ZM oppure QR. Quando si incontra una coppia già presente nel dizionario, si usa il codice relativo, ottenendo così un risparmio di 4 bit. Inoltre, si aggiunge al dizionario la sequenza composta dalla coppia e dal carattere che la segue. Questa terna, quando sarà incontrata di nuovo, verrà codificata con un unico codice, risparmiando ben dodici bit. Così si va avanti e, come s'è detto, si può arrivare a codificare con un unico codice anche lunghe sequenze, purché occorrano almeno due volte nel testo.

In questo modo, il dizionario dipende dal testo. C'è gente che si diverte a scrivere romanzi nei quali non compare la lettera U: in tal caso, il dizionario non conterrà alcun codice per le sequenze che comprendano almeno una U. Questo fa sì che le sequenze frequenti entrino facilmente nel dizionario, accrescendo la compressione. Questa si aggira di solito intorno al 40% e, in modo automatico, si adatta al testo da comprimere. Usando 12 bit per ciascun codice, si hanno 4096 sequenze codificate, e l'esperienza ha dimostrato che questo è un numero che di solito va bene, e la compressione può addirittura peggiorare se si passa a 13 bit.

Si potrebbe pensare che, come succede col metodo di Huffman, anche per l'LZW occorra memorizzare e spedire il dizionario assieme al testo compresso. Quello che ha di stupefacente questo metodo è che ciò non è vero, e quindi si risparmia in spazio e in tempo di trasmissione. Infatti, il dizionario si può ricostruire durante la fase di decodifica. Si parte con un dizionario contenente i caratteri di base, che, come s'è detto, possono ben essere le codifiche ASCII. Naturalmente, devono essere gli stessi usati per la codifica. Interpretando i codici di base, si ottengono i valori degli altri codici nello stesso ordine con cui si erano ottenuti in codifica. In questo modo, alla fine, il testo è interpretato perfettamente e il dizionario risultante è quello stesso che si era formato durante la codifica. Le prime volte che si usa a mano questo metodo, si ha quasi l'impressione di un procedimento magico.

Altri metodi di compressione sono stati inventati, proposti e anche commercializzati, vista l'importanza che essi rivestono. Tutti e tre i metodi descritti, Run Length Encoding, Huffman ed LZW sono metodi di compressione senza perdita di informazioni. Con questo si intende dire che il testo codificato viene decodificato riproducendo, con esattezza, il testo originale. Nessuna informazione viene perduta. Non sempre, però, c'è la necessità di avere una perfetta corrispondenza fra documento iniziale e documento decodificato. La necessità esiste per un documento scritto, ma è molto meno importante, ad esempio, per una fotografia o per un insieme di suoni. In questi casi, se la tonalità dell'azzurro riprodotta non è perfettamente uguale all'originale, nessuno se ne accorge, se non un esperto del settore.

Con questa idea in testa, si sono proposti vari metodi per la compressione con perdita di informazioni, soprattutto per le immagini e per i suoni. Durante la codifica, alcune informazioni, ritenute meno importanti, vanno perdute e non si riescono a recuperare nella fase di decodifica. Cos, l'immagine o il suono riprodotto, sono solo apparentemente uguali all'originale. La differenza, però, è così poca che normalmente non la si nota nemmeno. Storicamente, queste tecniche si sono applicate per prime alle immagini e poi ai suoni, ma questo ha poca importanza. Ciò che importa è osservare come questi metodi dipendano fortemente dalla Matematica, ed è praticamente impossibile spiegarli se non si hanno conoscenze abbastanza approfondite sulle matrici, sulle trasformate e sulla trigonometria.

Tanto per dare un'idea del metodo più famoso, *JPEG*[1], usato anche nelle macchine fotografiche digitali, possiamo dire questo. L'immagine viene scomposta in tanti quadratini, composti da 8 per 8, cioè 64 puntini colorati, o pixel. Si ha così una matrice 8 per 8, nella quale i colori sono interpretati come numeri. Si applica alla matrice una trasformazione, nota come la *trasformata discreta del coseno* o *DCT* dall'inglese *discreta cosine transform*. Lasciando perdere cosa ciò significhi, diciamo che questa trasformazione ha la capacità di separare il colore fondamentale di alcuni pixel dalle variazioni, di solito piccole, che li differenziano dagli altri. Si pensi a un pezzetto di cielo azzurro, che cambi impercettibilmente verso una tonalità più chiara o più scura.

Fatta questa operazione, se la parte contenente la variazione è composta di valori al di sotto di una certa soglia, essa viene scartata e si memorizzano solo

[1]La sigla significa: *Joint Photographer Expert Group*

i valori degli elementi di base. Questo permette di risparmiare spazio, ma, nel quadratino, può eliminare le differenze tra un pixel e l'altro. Si ha così la perdita di informazioni annunciata, che è tenuta sotto controllo dal valore di soglia usata per lo scarto. Si ottiene una grande compressione quando c'è una zona di colore uniforme: al posto di 64 valori se ne può avere anche uno solo. Si comprime invece poco o niente quando si hanno repentine variazioni di colore. La compressione dipende perciò dal tipo di immagine trattata e può superare il 60% oppure essere estremamente ridotta.

Questo metodo si riferisce alle immagini statiche, come foto e dipinti, e non è adatta alle immagini in movimento, tipo film o trasmissioni televisive. La riconversione in chiaro di ogni immagine richiede un bel po' di tempo, dovuto alla necessità di effettuare, all'inverso, le trasformazioni viste. Questo impedisce la visualizzazioni di immagini con una frequenza tale che il nostro occhio percepisca il movimento continuo, e non veda invece le immagini saltellanti come nei film anni '20. Per avere l'impressione della continuità, la nostra retina deve ricevere almeno 16 immagini al secondo. Nei vecchi film erano 24 e alla televisione sono almeno 25. Per ovviare a questo inconveniente è stato introdotto lo standard *MPEG*[2], che usa JPEG e un ulteriore accorgimento.

Consideriamo come esempio una ripresa cinematografica. Il primo fotogramma è codificato in JPEG. Il secondo è registrato solo per le sue differenze dal primo. Poiché, come si sa, fotogrammi contigui sono molto simili, se non quando cambia la scena, si ha un'ottima compressione e, in fase di decodifica, una conversione in chiaro molto veloce. Si continua così, memorizzando per differenza, col terzo e il quarto fotogramma, fino all'ottavo. Se si continuasse ancora, eventuali discrepanze provocate da questa compressione potrebbero generare notevoli difformità. Pertanto, il nono fotogramma è di nuovo codificato per intero con JPEG. Si riparte così procedendo per differenze per altri sette fotogrammi. In tal modo si arriva a una notevole velocità di decompressione e quindi a una buona visione.

Nel campo dei suoni, metodi di compressione rudimentali erano già stati usati per la telefonia analogica, quella del telefono di casa. Soprattutto i toni molto alti o molto bassi vengono tagliati e spesso capita di non poter riconoscere al telefono la voce di persone che sentiamo continuamente. Naturalmente, compressioni così brutali non si possono fare per la musica e il canto. Nel caso dei CD si usa il campionamento che già è una forma di compressione e quindi introduce delle imperfezioni. Per questo, sui CD non si usano metodi di compressione, limitando il numero di brani che possono esser registrati su un singolo disco.

Oggi sta prendendo piede, specie per le trasmissioni via Internet, ma non solo per esse, una tecnica di compressione dei suoni simile a MPEG per le immagini. Tale metodo è detto *MP3* e si tratta di una tecnica con perdita di informazioni. Come per le immagini, le differenze del suono riprodotto da quello originale sono piccole, ma possono essere colte, per cui non si può dire che il metodo sia eccezionale. Tuttavia, il vantaggio della diffusione tramite Internet e dell'accrescere il numero di brani registrabili, ha portato MP3 a un grande successo presso i giovani.

[2]La sigla significa *Moving Pictures Expert Group*

# Capitolo 4

# L'INTERNO DELL'ELABORATORE

## 4.1 La logica delle proposizioni

Come abbiamo avuto modo di ricordare, i primi studi sulla possibilità di costruire calcolatori elettrici risalgono alla metà degli anni 1930. In particolare fu Shannon a mostrare praticamente come il sistema binario fosse particolarmente adatto a questo scopo. Non possiamo però dimenticare i risultati attenuti da Zuse, in Germania, che purtroppo rimase ignorato fin dopo la guerra mondiale. In realtà, Shannon riprese nei suoi lavori i concetti ben noti della logica formale, la disciplina introdotta da Boole verso la metà del 1800 e che tanto successo aveva avuto alla fine del 1800 e al principio del 1900, così da cambiare radicalmente il nostro modo di vedere la Logica e la Matematica. Da allora, le due scienze sono diventate praticamente indistinguibili l'una dall'altra.

La Logica, introdotta da Aristotele con la formalizzazione dei *sillogismi*, era stata sviluppata dalla *Scolastica* che ne aveva fatto una vera e propria branca della filosofia. La sistemazione portata dai sillogismi sembrava tanto definitiva che per oltre cinque secoli quasi nessuno aveva tentato di criticarla o modificarla. Si prese invece questo compito Gorge Boole, un matematico inglese che nel 1847 pubblicò il libro *Investigazioni sulle leggi del pensiero* nel quale proponeva un approccio alla Logica del tutto diverso. La sua idea era quella di creare un "calcolo" delle proposizioni analogo alle quattro operazioni del calcolo fra numeri. Questa idea riecheggiava l'analoga proposta di Leibniz, ma Boole si dedicò allo sviluppo del suo calcolo riuscendo a dargli una forma effettiva.

Vale forse la pena di dedicare un po' di attenzione al *calcolo delle proposizioni* di Boole, poiché è proprio basandosi su di esso che si costruiscono i nostri calcolatori, secondo le idee di Shannon, e su di esso si fonda la nostra pratica della programmazione. Entrambe sono motivazioni forti per cercare di dare almeno un'idea di ciò che aveva pensato Boole. E' bene, a questo proposito, avvertire subito che il calcolo delle proposizioni è la parte più elementare della *Logica formale*, i cui sviluppi si basano piuttosto sul *calcolo dei predicati*. Ciò dovrebbe rassicurare sul fatto che le cose che stiamo per dire sono alla portata di tutti, basta capire pochi concetti e seguire con attenzione le cose che proporremo.

Il primo punto, e fondamentale, è capirsi su cosa intendiamo per *proposizione*: questa si definisce come una qualsiasi frase che possa essere vera o falsa. Ad esempio "Colombo scoprì l'America" è una proposizione, poiché è una frase vera. Per l'opposto motivo lo è anche "Cento è divisibile per tre". Le domande e le esclamazioni non sono proposizioni e, se non specifico in quale giorno sto parlando, non lo è la frase "Domani è domenica". Una proposizione deve sempre specificare esattamente i propri termini, in modo esplicito o implicito. Ad esempio, "domani" è un riferimento impreciso, se non si sa cosa sia oggi. Specificato questo, la frase precedente diviene una proposizione, e la frase è vera, se oggi è sabato.

Se definisco chiaramente il divano di cui sto parlando, la frase "quel divano è rosso" è una proposizione, che diremo *elementare* poiché non può essere scomposta in proposizioni più semplici. E' elementare anche la proposizione falsa "5 è maggiore di 7", ma possiamo immaginare proposizioni composte come "mangio e bevo", scomponibile nella proposizione "mangio" e in quella "bevo". Il problema del calcolo delle proposizioni consiste nello stabilire la verità o la falsità di una *proposizione composta*, conoscendo la verità e la falsità delle proposizioni elementari che la compongono. Naturalmente, ciò ha senso se possiamo stabilire con esattezza quali sono i possibili modi per legare proposizioni elementari a formare proposizioni composte. Il problema può apparire impossibile, poiché questi legami possono sembrare infiniti.

L'idea di Boole fu di procedere in senso inverso. Intanto osserviamo che se $p$ è una proposizione, tale è anche la sua *negazione*: $\sim p$, che si indica con una tilde e si legge "non è vero che p". Chiaramente, se $p$ è vera allora $\sim p$ è falsa, e viceversa. Ciò può essere formalizzato nella *tabella di verità* della negazione. Simbolicamente, il *valore di verità* "vero" si indica con 1, mentre il valore "falso" si denota con 0. Nella tabella di verità, ad ognuno dei due possibili valori di $p$ si associa il valore di verità della sua nega-

zione. Questa fu la prima "operazione" considerata da Boole, che non mancò di mettere in evidenza la sua analogia con il cambiamento di segno fra numeri, dato che $\sim\sim p = p$ come $--5 = 5$.

| $p$ | $\sim p$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

La seconda operazione considerata da Boole fu la congiunzione "e", che abbiamo usato in "mangio e bevo". Con la congiunzione "e" affermiamo la verità contemporanea delle proposizioni elementari che formano quella composta. Per questo, è facile costruire la tabella di verità. Come ben sappiamo, ci sono quattro combinazioni dei valori di verità di $p$ e di $q$, e il risultato è vero se e solo se sia $p$ che $q$ sono vere. Le quattro combinazioni si scrivono facilmente e l'operazione si dice di *congiunzione*, usando il termine per antonomasia:

| $p$ | $q$ | $p \wedge q$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Boole chiamò la congiunzione il *prodotto logico*, nome suggerito dalla stessa tabella di verità, che potrebbe essere interpretata come il prodotto dei valori di $p$ e di $q$. Questo è molto suggestivo e va citato, anche se oggi l'espressione si usa meno.

La terza operazione, che Boole chiamò *somma logica*, corrisponde alla congiunzione "o", "oppure", nel senso disgiuntivo, per cui oggi è detta *disgiunzione*. Quando diciamo "Oggi mangio con Luigi o con Carlo", se Luigi e Carlo sono amici tra di loro, intendiamo dire che mangeremo con Luigi, oppure se non lo troviamo con Carlo, ma potremmo anche mangiare in compagnia di tutti e due. Talvolta si dice "con Luigi e/o con Carlo", ma l'espressione è piuttosto bruttina. Questo è l'uso disgiuntivo della congiunzione "oppure". La sua tabella di verità è semplice e risulta falsa se e solo se le due le proposizioni componenti sono entrambe false, cioè se non mangio né con Luigi, né con Carlo. All'opposto della congiunzione, la disgiunzione è quasi sempre vera. L'idea di considerarla come somma sta proprio in questa osservazione:

| $p$ | $q$ | $p \vee q$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Per essere precisi, può anche darsi che Luigi e Carlo non si possano sopportare. Se allora dico "Oggi mangio con Luigi o con Carlo", e uso un tono di voce reciso, intendo dire che mangerò con l'uno o con l'altro, ma escludo di poter mangiare con entrambi. Qui "oppure" ha significato esclusivo, come in "prendere o lasciare", "vivo o morto", e la verità di una proposizione esclude che possa esser vera anche l'altra. Come operazione tra proposizioni si chiama *esclusione* e la sua tabella di verità è facile:

| $p$ | $q$ | $p \oplus q$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Boole non la considerò fra le sue operazioni di base, ma la ritroveremo un po' più avanti.

Propriamente, le operazioni tra proposizioni si dicono *connettivi logici*. Negazione, congiunzione e disgiunzione si dicono *connettivi booleani*, e sono presi come base. I Latini distinguevano con congiunzioni diverse i due casi di "oppure". Il senso disgiuntivo era dato da "vel", mentre il senso esclusivo era dato da "aut". Da questo è derivata l'espressione "Dare l'aut-aut", mentre dall'iniziale di "vel" è stato derivato il simbolo della disgiunzione. Quello della congiunzione è stato scelto di conseguenza.

Come dimostra il caso dell'esclusione, i tre connettivi booleani non sono i soli ad essere interessanti. Ci chiediamo allora: quanti e quali sono tutti i possibili connettivi? E di seguito: possiamo esprimerli mediante i tre connettivi booleani, e utilizzando solo loro? La risposta alla prima domanda è un numero intero preciso e la risposta alla seconda è "sì". Vediamo allora come si danno queste risposte.

Un connettivo logico è completamente definito dalla sua tabella di verità. Per capire questo con un esempio, costruiamo la tabella di verità di una proposizione composta, come ad esempio:

$$(p \wedge \sim q) \vee (q \wedge \sim r).$$

| $p$ | $q$ | $r$ | $\sim q$ | $(p \wedge \sim q)$ | $\sim r$ | $(q \wedge \sim r)$ | $\star$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |

Tabella 4.1:
Tabella di verità di: $\star = (p \wedge \sim q) \vee (q \wedge \sim r)$

Poiché abbiamo tre proposizioni elementari $p$, $q$ ed $r$, la tabella conterrà tutte le possibili combinazioni di vero/falso, cioè 0/1, legate a queste proposizioni. Si hanno 8 combinazioni, come si può vedere nella Tabella 4.1. Possiamo ora negare $q$ e costruire la colonna di $p\vee \sim q$ usando quella di $p$, quella di $\sim q$ e la tabella di verità della congiunzione. In modo analogo, costruiamo la negazione di $r$ e poi la colonna relativa a $q\wedge \sim r$. L'ultimo passaggio consiste nell'usare la tabella di verità della disgiunzione per trovare la colonna finale, che costituisce la tabella di verità della proposizione composta di partenza.

Ogni proposizione composta da $k$ proposizioni elementari avrà, in modo analogo, $2^k$ righe corrispondenti a tutte le combinazioni 0/1 delle $k$ proposizioni. Un connettivo è allora definito dai valori 0/1 della sua colonna, cioè tutte le combinazioni di 0/1 delle $2^k$ righe. In totale abbiamo $2^{2^k}$ proposizioni, e quindi $2^{2^k}$ connettivi diversi. Questi, pertanto, sono tutti e soli i possibili connettivi su $k$ proposizioni elementari. Se $k = 2$ si hanno 16 tabelle. Se $k = 3$ si hanno 256 connettivi, e così via. Il punto interessante, che si riesce a dimostrare, è che tutti questi connettivi sono esprimibili mediante le tre operazioni di Boole. In altre parole, usando solo i tre connettivi di negazione, congiunzione e disgiunzione si possono esprimere tutti gli altri.

La dimostrazione non è particolarmente difficile e può valer la pena di descriverla, anche se abbastanza informalmente. Si cominci a considerare una tabella di verità che contenga un solo valore 1. Poiché il nostro scopo è di dimostrare che ogni tabella di verità, e quindi ogni possibile connettivo, si può esprimere con le tre operazioni di base, vediamo che ciò è possibile in questo caso particolare. Poiché sappiamo che una congiunzione è vera se e solo se sono veri i suoi argomenti, occorre far sì che le proposizioni coinvolte siano tutte vere. Questo si ottiene facilmente congiungendo le proposizioni così come stanno quando l'argomento è vero, negandole quando l'argomento è falso. Nell'esempio (1) della Tabella 4.2 si ottiene $p\wedge \sim q \wedge r$, e questa proposizione è vera solo quando $p$ ed $r$ sono vere e $q$ è falsa. Abbiamo così risolto un caso speciale, ed ogni volta che abbiamo una tabella di verità con un solo 1 sappiamo come comportarci.

Passiamo allora a tabelle di verità nelle quali compaiano almeno due valori 1, come nell'esempio (2) della Tabella 4.2. Andiamo con ordine. Se nella tabella ci fosse solo il primo 1, sapremmo come fare e avremmo la nostra proposizione. Così avverrebbe se ci fosse soltanto il secondo 1. A questo punto, però, possiamo ricordare che una disgiunzione è vera, basta che uno dei suoi argomenti sia tale. Se allora eseguiamo la disgiunzione delle due proposizioni ottenute, il risultato sarà un connettivo che è vero in corrispondenza della verità di una qualsiasi delle due proposizioni, cioè proprio il connettivo che volevamo:

| $p$ | $q$ | $r$ | (1) | (2) | (3) |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 |

Tabella 4.2: Esempi per la costruzione

$$(\sim p\wedge \sim q \wedge r) \vee (p\wedge \sim q\wedge \sim r).$$

Un esempio con tre valori 1 nella tabella di verità ribadisce quanto abbiamo visto e non presenta alcuna difficoltà. L'esempio (3) della Tabella 4.2 dà come risultato:

$$(\sim p\wedge \sim q\wedge \sim r) \vee (\sim p \wedge q \wedge r) \vee (p \wedge q\wedge \sim r).$$

Una volta ottenuta la proposizione composta, si può fare la riprova e ricavare la tabella di verità usando, come abbiamo già fatto, le tabelle di verità dei tre connettivi booleani. Questo non fa che confermare il ragionamento fatto, relativo alle condizioni per cui la congiunzione è vera solo se tali sono i suoi argomenti, e la disgiunzione è vera basta che ne sia vero uno solo. E' istruttivo vedere cosa succede per l'esclusione; dalla tabella vista in precedenza si ha:

$$p \oplus q = (\sim p \wedge q) \vee (p\wedge \sim q).$$

Nel caso di "Mangio con Luigi o con Carlo", la formula trovata vuol dire: "Mangio senza Luigi ma con Carlo, oppure mangio con Luigi ma senza Carlo", e questo è appunto ciò che intendevamo dire.

In questa maniera abbiamo quasi risolto il nostro problema: una tabella di verità può contenere zero, uno o più 1. Quindi, non rimane altro che il caso in cui tutti i valori siano 0. Una proposizione come questa, che cioè sia sempre falsa, si dice una *contraddizione*. All'opposto, una proposizione sempre vera si dice una *tautologia*. Chiaramente, la contraddizione è la negazione della tautologia. Questa, però, rientra nel caso delle tabelle di verità con più 1, e quindi sappiamo come trovare l'espressione corrispondente, usando negazione, congiunzione e disgiunzione. Basta ora premettere all'espressione della tautologia una negazione, e il gioco è fatto: abbiamo un'espressione per la contraddizione.

Ecco quindi la conclusione: ogni tabella di verità, e di conseguenza ogni possibile connettivo fra proposizioni, si può ottenere con le tre operazioni di negazione, congiunzione e disgiunzione. I connettivi booleani, pertanto, sono sufficienti ad esprimere tutti gli altri. Come curiosità, si può osservare che essi sono addirittura sovrabbondanti. Infatti, la disgiunzione può essere espressa usando negazione e congiunzione, come si vede facilmente usando le tabelle di verità. E' vero anche che la congiunzione si può esprimere mediante negazione e disgiunzione, e le relative espressioni si dicono *leggi di De Morgan*, il matematico che in parte precedette Boole nello studio della Logica e che fu maestro di Ada Byron:

$$p \wedge q = \sim (\sim p \vee \sim q) \qquad p \vee q = \sim (\sim p \wedge \sim q).$$

I tre connettivi di Boole costituiscono, come si dice, un *insieme universale di connettivi*, proprio nel senso che permettono di realizzare tutti gli altri, cioè tutte le tabelle di verità che possono essere concepite. Le leggi di De Morgan ci dicono che negazione e congiunzione da sole, o anche negazione e disgiunzione, potrebbero essere adoperati allo stesso scopo. Per la verità, nel 1913, il matematico russo/americano Sheffer dimostrò che esistono due particolari connettivi, ciascuno dei quali, da solo, permette di realizzare tutti gli altri. Uno è la negazione della congiunzione, l'altro la negazione della disgiunzione:

$$p \barwedge q = \sim (p \wedge q) \qquad p \veebar q = \sim (p \vee q).$$

I due *connettivi di Sheffer* si dicono anche *NAND* e *NOR*, dove la N sta per "negazione" e AND ed OR sono i nomi inglesi di congiunzione e disgiunzione.

L'idea di Shannon può essere allora così descritta. Le tabelle della somma e del prodotto di numeri binari, interpretando, come s'è fatto, 1 per vero e 0 per falso, sono a tutti gli effetti connettivi logici. In particolare, la cifra risultato della somma è data dalla tabella dell'esclusione, mentre il riporto e il prodotto sono dati dalla congiunzione. In generale, ogni operazione aritmetica binaria che possiamo inventare è equivalente a una qualche tabella di verità. Quindi, è realizzabile mediante i tre connettivi booleani di negazione, congiunzione e disgiunzione, e solo loro. Se si riuscisse a creare dei circuiti elettrici che realizzano, in qualche modo, questi tre connettivi, combinandoli potremmo eseguire qualsiasi operazione.

Se interpretiamo come 1 il passaggio della corrente in un circuito, e come 0 il non passaggio, i circuiti relativi a congiunzione e disgiunzione erano ben noti. Basta infatti mettere due interruttori in serie per la congiunzione e due interruttori in parallelo per la disgiunzione. Per gli interruttori in serie, occorre che tutti e due siano chiusi perché passi la corrente. Per gli interruttori in parallelo, basta che sia chiuso uno solo, meglio se tutti e due (vedere Figura 4.1). Il criterio sul passaggio della corrente è molto semplice e conveniente. Basta inserire nel circuito una lampadina, così che se passa corrente quella si accende, e rimane spenta ne non passa. Quindi, lampadina accesa vuol dire 1, lampadina spenta 0.

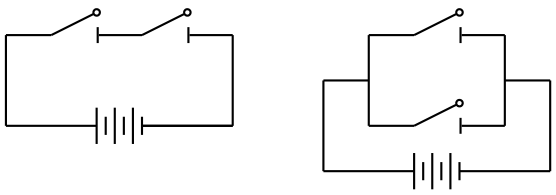


Figura 4.1: Circuiti per congiunzione e disgiunzione

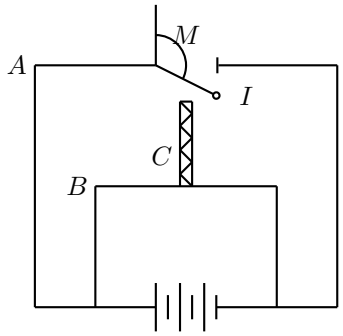



Figura 4.2: Schema di relé per la negazione

Opera un po' più complessa è realizzare la negazione. L'interruttore I è tenuto chiuso da una molla M di modo che nel circuito A passa corrente, quando nel circuito B non ne passa. Quando invece passa corrente nel circuito B, l'elettrocalamita C si magnetizza e fa aprire l'interruttore I superando la forza della molla M. In questo modo nel circuito A non passa più corrente. Questo dispositivo, noto come *relé*, fa sì che nel circuito A passi corrente se e solo se la corrente non passa nel circuito B. Pertanto, il relé realizza l'operazione di negazione. Con questi tre strumenti Shannon poté realizzare circuiti che eseguivano qualunque connettivo, e in particolare sviluppò circuiti per le operazioni aritmetiche in base 2. Il risultato finale appariva come una serie di lampadine che, accese e spente, potevano essere interpretate con il codice binario.

Presto, ai circuiti elettrici si sostituirono quelli elettronici a valvole, e poi alle valvole successero i transistor, ma la logica rimase la stessa. Vediamo, come semplice esempio, di costruire un circuito che realizza l'operazione di esclusione (vedere Figura 4.3). L'ingresso è rappresentato da due fili, uno per $p$ e l'altro per $q$. Possiamo non far passare corrente da nessuno

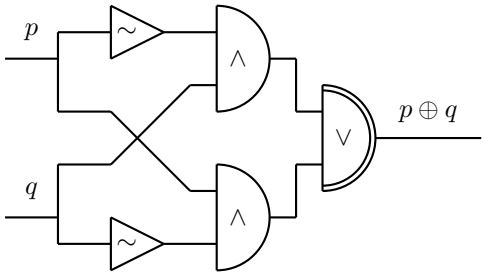



Figura 4.3: Un circuito per l'esclusione

| $p$ | $q$ | $r$ | $p \oplus q \oplus r$ | riporto |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

Tabella 4.3: Tabella per la somma e il riporto

dei due, possiamo far passare corrente in entrambi o possiamo far passare corrente in uno, ma non nell'altro. Deviamo i due fili, in modo da far passare una derivazione attraverso un circuito di negazione. Ora usiamo due circuiti di congiunzione per realizzare $p \wedge \sim q$ e $\sim p \wedge q$. Le uscite di questi due circuiti si combinano ora mediante un circuito che esegue la disgiunzione e l'uscita di questo ci dà il risultato finale. Abbiamo passaggio di corrente se e solo se abbiamo immesso corrente in $p$ o in $q$, ma non in entrambi.

Vogliamo ora progettare un circuito per eseguire la somma e il riporto. Dobbiamo immaginare un circuito in cui entrano tre fili: un bit per ciascuno dei due addendi e il riporto dalla somma parziale precedente. Se il circuito è quello delle unità, cioè quello più a destra, avrà un riporto fisso uguale a zero, ma questo non cambia il progetto generale. Questi tre fili corrispondono a tre proposizioni elementari $p$, $q$, per gli addendi, ed $r$ per il riporto. La tabella di verità si costruisce ora facilmente, come mostrato dalla Tabella 4.3. In realtà, dobbiamo considerare due tabelle: una per la cifra risultato della somma, l'altra per il riporto successivo, che sarà di ingresso per la somma dei due bit che si trovano una posizione più a sinistra.

Vediamo di progettare la parte che calcola la cifra della somma. La proposizione che dobbiamo realizzare è costituita dalla disgiunzione di quattro espressioni che contengono solo negazioni e congiunzioni:

$$p \oplus q \oplus r = (\sim p \wedge \sim q \wedge r) \vee (\sim p \wedge q \wedge r) \vee$$

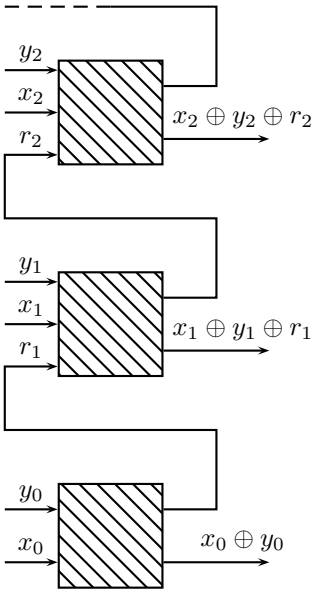



Figura 4.5: Un addizionatore completo

$$\vee(p \wedge \sim q \wedge \sim r) \vee (p \wedge q \wedge r).$$

Come prima (vedere Figura 4.4), deviamo opportunamente i fili di $p$, $q$ ed $r$ per realizzare le prime due congiunzioni, per le quali $p$ è negato (parte bassa della figura). Ogni circuito ha due fili in ingresso e questo ci obbliga a fare diverse congiunzioni. La disgiunzione finale ci dà un risultato parziale. L'altro pezzo del circuito fa cose analoghe, ma interagisce con $p$ e non con la sua negazione. Le congiunzioni sono un po' diverse, ma si arriva alla disgiunzione finale. La disgiunzione delle due disgiunzioni è l'operazione conclusiva. Essa ci dà il risultato voluto, contenuto nel filo in uscita.

In modo del tutto simile si può costruire il circuito che dà il riporto per la somma delle cifre binarie successive. Mettendo insieme i due circuiti, si ottiene un *addizionatore* completo. Possiamo immaginarcelo come una scatoletta nella quale entrano tre fili: il riporto precedente, il bit del primo addendo $x$ e il bit del secondo addendo $y$. La scatoletta fa quello che deve fare e produce due risultati. Questi corrispondono ai due fili in uscita: uno è il bit del risultato, mentre l'altro è il riporto, che andrà ad alimentare la scatoletta successiva. Diverse repliche della nostra scatoletta ci permettono di costruire un addizionatore vero e proprio, che si estende a quanti bit vogliamo.

Normalmente, un addizionatore è composto da 32 scatolette tutte uguali. La prima a destra, quella delle unità tanto per intenderci, parte con un riporto 0, e i riporti successivi si propagano verso sinistra (o, come nella Figura 4.5, verso l'alto). L'addizionatore è solo una delle tante componenti della CPU, che

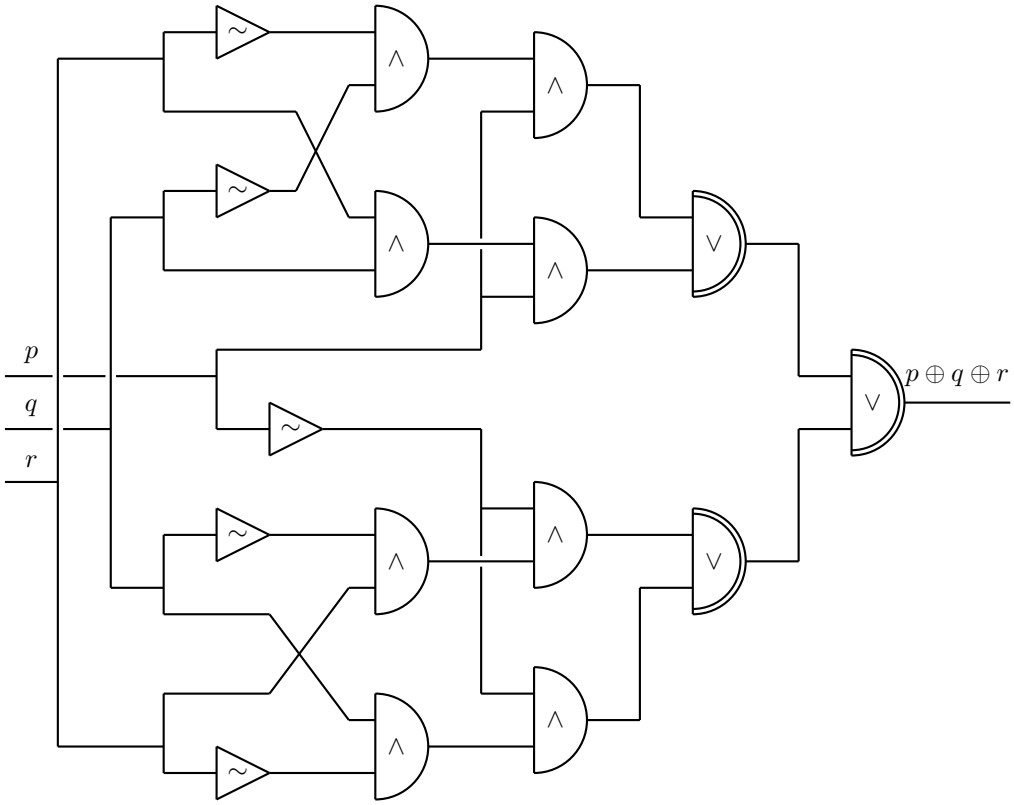



Figura 4.4: Un circuito per l'addizionatore

contiene milioni di transistor, ognuno dei quali esegue una delle operazioni booleane di negazione, congiunzione e disgiunzione. A noi, naturalmente, basta aver capito il principio e non ci vogliamo preoccupare troppo delle tecniche usate effettivamente. Ad esempio, ci sono metodi per semplificare le espressioni che si trovano dalle tabelle di verità con il metodo esposto. Permettono perciò di creare circuiti più semplici e di conseguenza più efficienti.

Nella costruzione di un elaboratore, potrebbe essere conveniente usare anche componenti diverse, insieme o in alternativa ai tre connettivi booleani. Infatti, nella pratica, si usa molto il connettivo di Sheffer NAND, facile da realizzare e assai versatile. Basterebbe da solo, come s'è visto, a costruire qualsiasi altro connettivo, ma si preferisce utilizzarlo assieme ad altre componenti, sempre allo scopo di ottimizzare la costruzione. A questo punto, però, facciamo basta poiché non vogliamo rubare il mestiere agli ingegneri elettronici, per i quali queste cose costituiscono il pane quotidiano.

## 4.2 Il linguaggio macchina

L'idea di Shannon di realizzare le operazioni aritmetiche mediante circuiti elettrici, sfruttando la Logica di Boole, fu portata presto alle estreme conseguenze. I circuiti elettrici permettono di effettuare le operazioni abbastanza velocemente, diciamo un centinaio al secondo. Il movimento meccanico degli interruttori è piuttosto lento e per questo si passò presto ai circuiti elettronici, nei quali il passaggio o meno della corrente è controllato dalle valvole. In questa maniera si superò il migliaio di operazioni al secondo, ma, come s'è osservato, questa velocità non può essere sfruttata se si continuano ad immettere manualmente i dati e i risultati parziali. Questo richiede infatti molti secondi.

Un altro punto importante e che obbliga a rallentare i calcoli, è la necessità di prendere decisioni, che determinano l'effettuazione di processi di calcolo diversi a seconda dei casi. Nel classico esempio della risoluzione di un'equazione di secondo grado $ax^2+bx+c=0$, che tutti ricordano, dopo aver calcolato il discriminante dell'equazione $\Delta = \sqrt{b^2-4ac}$, a seconda che questo sia maggiore, uguale o minore di 0, si devono compiere azioni diverse. Le decisioni devono, apparentemente, essere prese da una persona, ma proprio la Logica di Boole ci dice che possiamo creare un calcolo delle proposizioni che, essendo meccanizzabile, può essere demandato ad una macchina. Quindi, un elaboratore deve essere capace di fare due cose: calcolare e prendere decisioni.

Identificare 1 con vero e 0 con falso - porta in modo semplice a far sì che l'elaboratore possa associare una certa serie di operazioni al verificarsi di una condizione, e una serie completamente diversa al non verificarsi. Basta connettere la prima serie al valore 1 e la seconda al valore 0. Per capire come la cosa funzioni da un punto di vista tecnico, basta ritornare al concetto di von Neumann sul programma interno, cioè contenuto nella memoria dell'elaboratore. Vediamo allora più da vicino come è fatta la memoria e come la si usi per memorizzare dati (cosa che già sappiamo) e programmi.

La memoria si configura come una lunga sequenza di byte. Ogni byte è identificato dalla sua posizione nella sequenza, e tale posizione si dice l'*indirizzo del byte*. Il primo byte occupa la posizione 0, il secondo la posizione 1, e così via. Il byte che occupa la posizione 10 mila è identificato dall'indirizzo 10 mila. Ogni dato, come s'è detto, occupa uno o più byte ed è identificato dall'indirizzo del primo dei suoi byte. Un numero intero occupa di solito 4 byte e quindi se occupa i byte da 10·000 a 10·003, il suo indirizzo sarà proprio 10 mila. Analogamente, un'operazione elementare dell'elaboratore, o *istruzione*, occupa alcuni byte consecutivi della memoria ed è identificata dall'indirizzo del primo byte.

Normalmente, le istruzioni sono eseguite una dopo l'altra. Un particolare registro della CPU, detto *contatore di istruzioni* o *contatore di programma*, contiene l'indirizzo dell'istruzione da eseguire, e mentre la CPU esegue tale istruzione il contatore è automaticamente fatto avanzare all'indirizzo dell'istruzione successiva. Ci sono però alcune istruzioni, dette di *salto* o, in inglese, di *branch* o di *jump*, che modificano il contatore di programma se si è verificata una certa condizione, ad esempio un risultato parziale è 0 oppure è diverso da 0. Possiamo allora dire all'elaboratore di saltare a una certa parte del programma se la condizione si è verificata, di saltare invece ad un'altra, o, semplicemente, di continuare, se non si è verificata.

La tecnica dei salti è alla base della programmazione a livello della macchina, cioè al livello fisico, hardware. Come vedremo, a livello logico si usano altre tecniche, ma queste non si possono applicare alla struttura fisica dell'elaboratore, almeno allo stato attuale della tecnologia. Un'istruzione di salto non fa altro che modificare il contatore di programma, di modo che l'istruzione successiva sarà quella indicata e non quella che segue fisicamente. Il salto è spesso condizionato, cioè è effettuato solo se si è verificata una certa condizione, come ad esempio il fatto che l'elaboratore ha controllato positivamente che un certo valore è zero. Si pensi ancora al delta dell'equazione di secondo grado.

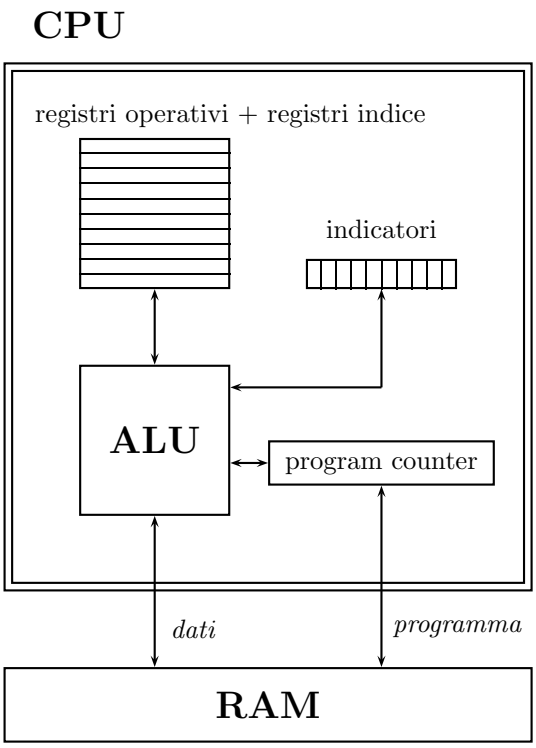



Figura 4.6: Struttura della CPU

In realtà, la CPU non contiene solo i circuiti che eseguono le istruzioni dell'elaboratore. Talvolta, l'insieme di questi circuiti si denota con la sigla *ALU*, cioè *Arithmetic Logic Unit*, per mettere in evidenza che essa serve proprio per le operazioni. Come abbiamo appena visto, la CPU contiene anche il contatore di programma; contiene inoltre un insieme di *registri operativi*, o *accumulatori*, e un gruppetto di *indicatori*. I registri operativi possono essere da due a qualche decina o qualche centinaio. Facciamo il caso semplice che ce ne siano otto e chiamiamoli convenzionalmente con i numeri da 0 a 7. Di solito ogni registro è costituito da 4 byte e si dice operativo perché serve ad eseguire le operazioni, cioè le istruzioni realizzate dalla ALU: le quattro operazioni elementari, le operazioni logiche, i confronti e i salti.

Facciamo l'esempio di una somma. I due addendi si trovano nella memoria centrale e lì dovrà finire anche il risultato. Sarebbe però complesso per l'elaboratore realizzare l'operazione fra due numeri che si possono trovare ovunque nella memoria. Il numero di collegamenti necessari sarebbe inimmaginabile. Si preferisce allora trasferire il primo addendo in un registro, diciamo il 3, sommare al registro il secondo addendo e infine trasferire il risultato, che si trova ancora nel registro, nella posizione a lui riservata:

```
LOAD    3,10000
ADD     3,10004
STORE   3,10020
```

Ogni operazione si svolge con la medesima logica: caricare nel registro, come si dice tecnicamente, il primo argomento dell'istruzione, eseguire l'operazione nel registro e infine scaricare in memoria il risultato ottenuto, che potrà così servire per qualche istruzione successiva. Il registro viene di conseguenza liberato per operazioni successive.

I 32 bit del registro sono, di solito, sufficienti per eseguire le somme e le differenze. Se il risultato è troppo grande, se cioè si è avuto un riporto dalle ultime cifre a sinistra, si dice che si è verificata una condizione di *trabocco*, o di *overflow*, se si vuole usare una parola inglese. Questo fa porre ad 1 il valore dell'indicatore specifico che si trova nella CPU. In altre parole, gli indicatori della CPU non sono altro che bit pronti a segnalare una qualche condizione. L'indicatore di trabocco vale normalmente 0, ma viene posto ad 1 ogni volta che una somma o una sottrazione danno un risultato che non rientra nei quattro byte destinati ai numeri. Un'istruzione di salto può controllare questo indicatore e, se vale 1, portare all'esecuzione di un pezzo di programma che segnali il fatto.

La moltiplicazione e la divisione richiedono l'uso di due registri, convenzionalmente uno pari e uno dispari consecutivi. Se si moltiplicano due numeri di $k$ cifre, si ottiene un numero che ha $2k$ oppure $2k-1$ cifre. Quindi, se i due fattori occupano ciascuno 4 byte, il loro prodotto potrebbe estendersi ad 8 byte. Allora, ad esempio, i registri 2 e 3 si usano in questo modo: caricato il moltiplicando nel registro 3 ed eseguita la moltiplicazione, il risultato può estendersi ai due registri 2 e 3. L'estensione effettiva al registro 2 è un caso di trabocco e come tale è segnalato, altrimenti il risultato è contenuto nel registro 3. Nella divisione, portato il dividendo nel registro 2, l'esecuzione dell'operazione lascia nello stesso registro il quoziente, mentre nel registro 3 si trova il resto.

Fin qui abbiamo parlato di istruzioni che operano su numeri assoluti o su interi negativi. Per i numeri in virgola mobile, cioè in floating point, sono spesso previsti registri particolari. Ciò semplifica l'esecuzione di queste operazioni, in quanto si sa in anticipo che si vuole utilizzare l'aritmetica in virgola mobile. Moltiplicazione e divisione hanno, più o meno, gli stessi problemi visti per i numeri interi. Ricordiamo che, secondo l'Algebra, non possono essere fatte divisioni per 0. Prima di eseguire una qualsiasi divisione, l'elaboratore controlla il divisore e, se esso è zero, mette ad 1 il valore dell'indicatore "divisione per zero". E' così possibile segnalare tale condizione.

L'indicatore di trabocco e quello di divisione per 0 servono a segnalare errori e quindi, in generale, chi programma spera che non si verifichino mai. Vorrà, invece, controllare il verificarsi di certe condizioni, aritmetiche o logiche, Ad esempio, può essere interessato al fatto che un numero sia maggiore, uguale o minore di 0, oppure che due numeri siano uguali, o che il primo sia maggiore o minore dell'altro. Nella CPU ci sono due indicatori: uno per uguale e uno per maggiore. Si può testare il numero contenuto in un registro e tale istruzione metterà ad 1 l'indicatore di uguale se il numero è uguale a 0, e metterà ad 1 l'indicatore di maggiore se il numero è maggiore di 0. In modo analogo, si può confrontare il numero contenuto in un registro col numero contenuto in una certa posizione di memoria.

Se si vuole, è in effetti conveniente pensare al test sul numero contenuto in un registro come al confronto di tale numero con lo 0. Si capisce così che esso è un caso particolare dell'altro. Come si ricorderà, in Matematica ci sono sei relatori: uguale, diverso, maggiore, maggiore o uguale, minore e minore o uguale. I due indicatori sono sufficienti a controllare tutte e sei le condizioni. I due numeri confrontati sono uguali se l'indicatore di uguaglianza vale 1; sono diversi se vale 0. Il primo, cioè il numero nel registro, è maggiore o uguale al secondo se vale 1 l'indicatore di maggiore oppure se vale 1 l'indicatore di uguaglianza. Si rammenti per questo la disgiunzione e s'osservi che i due indicatori non possono mai essere tutti e due uguali ad 1, poiché le due condizioni si escludono a vicenda. Il quadro completo corrispondente ai sei relatori è il seguente:

$$\begin{array}{cc} = & EQ = 1 \\ \neq & EQ = 0 \\ > & GT = 1 \\ \geq & (EQ = 1) \vee (GT = 1) \\ < & (EQ = 0) \wedge (GT = 0) \\ \leq & GT = 0 \end{array}$$

Con questo modus operandi, l'elaboratore riesce a prendere le sue decisioni e a saltare a quella parte del programma corrispondente a una certa condizione. Ad esempio, l'operazione: "Salta al pezzo di programma che comincia all'indirizzo 20 mila se il contenuto del registro 3 è negativo" consisterà di due istruzioni. La prima controlla il contenuto del registro 3 e pone gli indicatori al valore appropriato. La seconda controlla se i due indicatori sono stati entrambi posti a zero. Se questo è il caso, nel registro c'è un numero che non è né uguale né maggiore di 0, quindi è negativo. Cambi allora il contatore del programma a 20 mila. In caso contrario, il numero non è negativo, il contatore di programma è lasciato invariato e l'elaboratore proseguirà con l'istruzione successiva.

Spesso, la CPU possiede uno o più *registri indice*. Questi contengono valori numerici che modificano l'indirizzo del dato da trattare. Ad esempio, se il registro indice $X$ contiene il numero 20, l'istruzione che dice di caricare nel registro 4 il dato della posizione 10 mila, modificato da $X$, in realtà porta nel registro 4 il dato della posizione 10 mila e venti. Questo è assai utile quando si vuole fare la somma di molti dati consecutivi. Basta per questo partire col valore 0 nel registro indice $X$ e quindi incrementarlo, di volta in volta, della lunghezza in byte di ciascun dato. I numeri, come sappiamo, occupano 4 byte e l'incremento sarà pertanto di 4. Alcuni elaboratori hanno un'unica serie di registri che, indifferentemente, servono da accumulatori e da indici.

Ogni istruzione ha un proprio codice, molto spesso individuato da un byte. La relativa istruzione è più lunga perché deve specificare quale o quali registri coinvolgere e quale o quali indirizzi di memoria deve usare come riferimento al dato o ai dati. La codifica delle istruzioni è in buona parte arbitraria, ma va conosciuta quando si voglia programmare l'elaboratore scendendo fino alla scrittura della singola istruzione. Oggi questo non si fa più, ma i primi elaboratori erano programmati proprio in questo modo. L'insieme delle convenzioni sui codici delle istruzioni, l'indicazione dei registri e degli indirizzi di memoria si dice il *linguaggio macchina*, e varia da elaboratore ad elaboratore.

I primi elaboratori venivano programmati direttamente in linguaggio macchina. se la somma aveva codice 01010000, era questa sequenza di bit a dover essere inserita nella memoria dell'elaboratore, seguita dall'indicazione del registro e dell'indirizzo del dato da sommare, anche questi rigorosamente in binario. Per questo, programmare era un'impresa e solo pochi pazzi si potevano dedicare a un lavoro del genere. Lo sforzo di attenzione era enorme, sia nella scrittura di un programma, sia nella sua messa a punto. Individuare un errore era tragico. Ci si riferisce a questa fase della storia degli elaboratori come alla *prima generazione*, e al linguaggio macchina come al *linguaggio della prima generazione*, l'era pionieristica dell'Informatica.

Si studiarono presto modi per rendere la programmazione più umana. Un'idea apparentemente semplice, ma ricca di conseguenze, fu quella di indicare con simboli alfabetici le varie istruzioni. Come ha poi dimostrato il codice ASCII, non è difficile associare alle lettere una precisa sequenza di bit. Inizialmente, si considerarono solo lettere maiuscole, cifre e segni di interpunzione, facendoli corrispondere a sequenze di 6 bit. Questo permetteva di trattare 64 caratteri diversi. Comunque, si poteva associare la parola ADD all'istruzione di somma e, quindi, al codice 01010000. La moltiplicazione alla sigla MPY, cioè "multiply", il caricamento in un registro a LD, cioè "load", e la memorizzazione a ST, cioè "store".

In questa maniera, il programmatore non doveva più ricordare sequenze di bit senza senso come 01010000, ma aveva a disposizione codici facili da tenere a mente e semplici da scrivere. Le istruzioni di salto erano spesso indicate con una B, per "branch", o una J, per "jump". La condizione si inglobava di solito nel codice mnemonico: BEQ significava "Salta se l'indicatore di uguaglianza (EQuality) è uguale ad 1". BGT "Salta se l'indicatore di maggiore (Greater Than) vale 1". E così via, secondo la terminologia inglese imperante. L'istruzione B *tout-court* era il *salto incondizionato*: quando il programma trovava questa istruzione, doveva per forza saltare all'istruzione indicata dalla parte indirizzo. Un tale salto si trovava di regola alla fine di un gruppo di istruzioni legate a una condizione e permetteva di rientrare nel flusso normale del programma.

Un particolare tipo di salto era ed è BS: *branch and save*, cioè, *salta e salva*. Con l'istruzione: BS 8,20000, l'elaboratore salta all'indirizzo specificato, cioè 20 mila, e salva nel registro 8 l'indirizzo dell'istruzione successiva a BS. L'istruzione nella locazione 20 mila salva il contenuto del registro 8, diciamo nella posizione 21 mila, quindi prosegue eseguendo un gruppo G di istruzioni dopo la 20 mila. Ciò fatto, carica di nuovo nel registro 8 il contenuto salvato in 21 mila ed esegue un'istruzione: BR 8, un salto incondizionato all'indirizzo che si trova nel registro 8. Questo riporta l'esecuzione del programma esattamente dove era stato lasciato con l'istruzione salta e salva. In questo modo, è come se l'elaboratore avesse trovato il gruppo di istruzioni G al posto del salta e salva:

```
          ...    ...
19600     BS     8,20000
19604     ...    ...
          ...    ...
20000     ST     8,19996
20004     ...    ...
          {G}
20100     LD     8,19996
20104     BR     8
```

Supponiamo che il gruppo di istruzioni G esegua la radice quadrata del numero contenuto nel registro 1. Se il programma deve eseguire molte volte la radice quadrata, che non è un'istruzione di base dell'elaboratore, questa tecnica fa risparmiare un bel po' di spazio e un bel po' di sforzi del programmatore. Egli, infatti, si deve preoccupare una volta sola di creare e mettere a punto un programma che calcola la radice quadrata. Il gruppo di istruzioni G prende il nome di *sottoprogramma* e costituisce una tecnica utilissima

nella programmazione. Tutte le procedure che calcolano funzioni complicate, quali logaritmi, esponenziali, funzioni trigonometriche, sono realizzate come sottoprogrammi. E il programmatore stesso può costruire propri sottoprogrammi per eseguire procedure che vuole o deve eseguire tante volte.

L'idea di assegnare un codice a ogni istruzione non si fermò qui. Anche i registri ebbero un codice, e soprattutto ebbero un codice le posizioni della memoria. Ad esempio, si può dire all'elaboratore di inserire i dati di un programma a partire dalla locazione 20 mila. Se i dati sono tre numeri interi A, B, C, automaticamente A occuperà i byte da 20·000 a 20·003, B i byte da 20·004 a 20·007, e C i quattro byte successivi. Per sommare le due quantità che si trovano nelle posizioni A e B e memorizzare il risultato in C, servono tre istruzioni. Caricare il valore di A in un registro, diciamo il 3; sommare al registro 3 il contenuto di B; memorizzare in C il contenuto del registro 3, che ora contiene la somma A+B:

```
LD    3,A
ADD   3,B
ST    3,C
```

Si noti che non si può confondere l'istruzione B con la posizione di memoria B, poiché la lettera B viene ad occupare posizioni diverse nell'ambito dell'istruzione: si ha cioè una distinzione sintattica fra i due usi della lettera B. Normalmente, il nome o codice di un indirizzo è composto da una sola lettera oppure da una lettera seguita da altre lettere e cifre. In generale, un nome come A, B, oppure AB7, X3, CODICE o NUMERO indicano, come detto, indirizzi di memoria. Qui possiamo trovare numeri, sequenze di caratteri, o altro. In particolare i nomi possono indicare istruzioni all'interno di un programma. Questo permette di dare riferimenti simbolici alle istruzioni di salto: BEQ MAGG, significa che il programma deve saltare all'istruzione indicata dal nome MAGG se l'indicatore di uguaglianza vale 1. Altrimenti, il programma prosegue con l'istruzione successiva.

Tanto per fare un esempio semplice ma significativo, si supponga di avere tre numeri, $a, b, c$ e di voler sommare $c$ al più grande fra $a$ e $b$. Le istruzioni che dobbiamo far eseguire all'elaboratore sono soltanto cinque. Cominciamo col portare il numero $a$ nel registro 5 e quindi confrontiamolo col numero $b$. Se $a$ (cioè, il registro 5) è maggiore o uguale a $b$, nel registro abbiamo già il numero più grande fra $a$ e $b$ e quindi saltiamo all'istruzione MAGG. Se invece $a$ fosse minore di $b$, si prosegue con l'istruzione successiva che carica $b$ nel registro 5, sostituendolo così ad $a$. In ogni caso, quando si arriva all'istruzione MAGG nel registro 5 troviamo il maggiore tra $a$ e $b$. Non resta che sommare il valore di $c$ e nel registro 5 abbiamo il risultato corretto. E questo è tutto, se con A, B, C indichiamo le posizioni di memoria che contengono i valori $a$, $b$ e $c$:

```
      LD    5,A
      CMP   5,B
      BGE   MAGG
      LD    5,B
MAGG  ADD   5,C
```

Questo modo di affrontare la programmazione è molto più semplice di quanto non lo fosse utilizzare il linguaggio macchina. Il linguaggio, che risulta dall'uso dei simboli per indicare il codice delle istruzioni, i registri e le locazioni di memoria, si dice *linguaggio simbolico*. I linguaggi simbolici hanno costituito una vera e propria rivoluzione nei confronti dei linguaggi macchina, anche se deve esistere un linguaggio simbolico per ogni linguaggio macchina. Si dice allora che i linguaggi simbolici costituiscono i *linguaggi di programmazione della seconda generazione*. Essi si dicono anche *linguaggi assemblatori* o semplicemente *Assembler*. Servono infatti ad assemblare in un tutto unico dati e programmi e perfino programmi di programmatori diversi. Stabilendo opportune convenzioni, si usa la tecnica dei sottoprogrammi per dividere un grosso progetto in più parti e affidarlo a programmatori diversi, accelerando i tempi di realizzazione.

La programmazione Assembler è usata oggi solo in occasioni particolari, come ad esempio per realizzare programmi particolarmente veloci. Programmare in Assembler è come programmare in linguaggio macchina e si possono sfruttare tutte le caratteristiche dell'elaboratore per cercare di ottimizzare un programma. Nonostante la loro velocità, gli elaboratori incontrano spesso problemi che richiedono, per la loro soluzione, un numero enorme di istruzioni. Vedremo più avanti esempi di questi problemi difficili, ma anche problemi più semplici possono richiedere un'estrema attenzione. Quando si programma per il pubblico, non si possono costruire programmi che prolungano l'attesa degli utenti, e la velocità diviene un parametro essenziale di valutazione del software.

I punti deboli dei linguaggi simbolici sono essenzialmente due. Primo, essi cambiano da elaboratore ad elaboratore, poiché riflettono esattamente il linguaggio macchina. Se vogliamo portare un programma da una macchina ad un'altra, esso in pratica deve essere completamente riscritto. Secondo, se abbiamo una formula come $\sqrt{b^2 - 4ac}$, essa deve essere tradotta in una sequenza di operazioni elementari che producano lo stesso risultato. Il passo successivo nella storia dei linguaggi di programmazione è stato quello di crea-

re dei linguaggi indipendenti dal singolo elaboratore e che potessero permettere una programmazione più vicina alla nostra Algebra. Linguaggi, cioè, più facili da apprendere e da utilizzare. Vedremo nella prossima sezione questo tipo di linguaggi, i cosiddetti *linguaggi della terza generazione*.

## 4.3 I linguaggi di programmazione

Nella prima metà degli anni 1950 coloro che si interessavano di programmazione si resero conto che l'Assembler era una vera e propria lingua, era la lingua che permetteva all'uomo di comunicare con l'elaboratore. La lingua comune (italiano o inglese che sia) è specifica dell'uomo, con le sue convenzioni, le sue regole e tutte le ambiguità che l'uso quotidiano ci ha insegnato a dirimere, anche se non sempre. Il linguaggio macchina è la lingua propria dell'elaboratore, l'unica che esso capisca direttamente. Anch'essa ha le proprie regole e le proprie convenzioni, ma è priva di ambiguità. Se ne avesse, l'elaboratore andrebbe in tilt - come si dice: non saprebbe come comportarsi oppure sceglierebbe un'alternativa, non si sa secondo quale regola.

L'uomo, come s'è visto, impara con molta difficoltà il linguaggio macchina, e l'elaboratore non è assolutamente in grado di apprendere una lingua comune. L'Assembler è una via di mezzo, ma favorisce l'elaboratore. Così ci si sforzò di trovare una lingua franca, tipo l'esperanto, che facesse da cavo di connessione tra l'uomo e la macchina. Le caratteristiche formali dovevano essere quelle della lingua naturale, o almeno del linguaggio scientifico e matematico, ma doveva essere evitato qualsiasi tipo di ambiguità. Ciò doveva permettere un colloquio senza fraintendimenti e, quindi, una esecuzione rigorosa dei programmi creati per la soluzione dei problemi.

Uno dei primi linguaggi di questo tipo fu il *COBOL*[1], pensato espressamente per le applicazioni commerciali. Queste richiedono di solito l'analisi e la specifica di tanti casi diversi, ma hanno un apparato matematico molto elementare, per lo più legato alle quattro operazioni di base. D'altra parte, le prime applicazioni degli elaboratori erano state di tipo scientifico, anche se di interesse militare, con largo impiego di logaritmi, esponenziali e funzioni trigonometriche. Lo studio delle applicazioni scientifico/numeriche consigliò presto di pensare a linguaggi nei quali fossero facilmente esprimibili formule algebriche, che potessero essere convertite dall'elaboratore stesso in sequenze di istruzioni del linguaggio macchina. Nacque così il primo *linguaggio algebrico*: il *FORTRAN*.

Il nome FORTRAN sta a significare FORmula TRANslator e in effetti la caratteristica più notevole nei confronti del COBOL era proprio la possibilità che il FORTRAN offriva al programmatore di scrivere le formule più o meno nella forma abituale. Ad esempio, "A+B*C", a parte l'uso dell'asterisco come segno di moltiplicazione, fa quello che uno si aspetta, sommando A al risultato del prodotto di B per C. Questo, è bene osservare, non è del tutto banale, e molti studenti alle prime armi hanno la tendenza ad eseguire la somma A+B e poi a moltiplicare il risultato per C. Questo, come si sa, è scorretto e la convenzione è quella ben nota: in assenza di parentesi, il prodotto ha *precedenza* sulla somma e sulla differenza. Somme e differenze, invece, si eseguono in ordine da sinistra verso destra.

Il FORTRAN permette di scrivere agevolmente anche formula più complesse. Naturalmente, la radice quadrata non può mantenere il suo simbolo caratteristico. Poiché in inglese si dice *square root*, si usa la notazione funzionale: SQRT($x$), abbastanza perspicua di per sé. Il problema più rilevante è quello della divisione, che impone una scrittura su più linee. Questo è quasi impossibile oggi ed era del tutto inammissibile negli anni 1950, poiché l'ingresso dei dati doveva avvenire rigorosamente in linea. Analoghi problemi pongono indici ed esponenti. Così, la divisione si indica con il simbolo tradizionale della barra, e si usano le parentesi per raggruppare, in modo non ambiguo, numeratore e denominatore. Per l'elevamento a potenza si usano due asterischi e gli indici sono messi tra parentesi:

$\sqrt{b^2 - 4ac}$ diviene: SQRT(B**2 - 4*A*C)

Con queste convenzioni, il FORTRAN permette di scrivere formule leggibili dall'uomo e interpretabili dalla macchina. Oggi, in altri linguaggi, si preferisce il simbolo dell'accento circonflesso ai due asterischi e si usano le parentesi quadre al posto di quelle tonde, ma ben poco è cambiato.

Oltre alle formule, il FORTRAN presentava altri aspetti interessanti, e qui vorremmo ricordare il *costrutto iterativo* del DO, che oggi preferiremmo chiamare del FOR, l'uso massiccio dei sottoprogrammi con parametri, e la gestione sofisticata dei dati in ingresso e, soprattutto, in uscita. Quest'ultimo aspetto era, allora, molto importante poiché alleviava il programmatore dalla necessità di creare complicate routine di conversione dei numeri e di strutturazione dei dati in uscita per rendere chiari i risultati ottenuti.

Se si deve effettuare una statistica, ad esempio calcolare la media di cento o mille quantità, il metodo più semplice è quello di memorizzare tali quantità una

[1] Questo nome è una sigla e significa *COmmon Businness Oriented Language*

dietro l'altra, e quindi di accumulare la loro somma in una posizione di memoria, che indichiamo con S. Indichiamo con N il numero dei dati da sommare; allora le istruzione per valutare il totale sono abbastanza semplici:

```
      S=0
      DO 150 I=1,N
      S=S+Q(I)
150   CONTINUE
```

Si comincia azzerando S e quindi si esegue l'istruzione iterativa, indicata in FORTRAN dal comando DO, come s'è detto. Questo richiede di eseguire tutti i comandi (o istruzioni) fino a quello col numero specificato, cioè 150 (un numero arbitrario, ma univoco), incrementando di volta in volta l'indice I. Questo si riferisce ai dati da sommare, che saranno Q(1), Q(2), Q(3) e così via fino a Q(N). Quando si arriva ad N, l'elaboratore capisce che la somma è finita e sa che essa si trova in S. Il programma proseguirà poi con l'esecuzione delle istruzioni successive.

Un sottoprogramma, come abbiamo già visto, è una parte di programma che possiamo isolare e trattare per conto proprio, essendo chiusa in sé medesima. Tipico esempio è un blocco di istruzioni che calcola la radice quadrata di un numero. Questo viene detto *parametro* e varia a seconda di quale numero vogliamo calcolare la radice. Le istruzioni di questo blocco rimangono sempre le stesse e quindi isolarle ha almeno tre vantaggi:

1. possiamo separare, fisicamente e logicamente, il sottoprogramma dal programma e realizzarlo indipendentemente da questo;

2. possiamo risparmiare spazio scrivendo la routine una sola volta, anche se dobbiamo calcolare la radice 133 volte;

3. possiamo utilizzare lo stesso sottoprogramma in altri programmi, che nulla hanno a che fare con quello che stiamo preparando.

Poter suddividere un grosso programma in tanti sottoprogrammi ha due importanti risvolti, uno logico/organizzativo e l'altro pratico. Ogni sottoprogramma può essere studiato per conto proprio, realizzato e gestito senza interferenze con le altre parti del programma. Spesso ciò permette di ottimizzare le prestazioni di ciascun sottoprogramma e ottenere un risultato migliore, globalmente e nello specifico. In pratica, poi, più persone possono essere messe a sviluppare i vari sottoprogrammi, così da rendere molto più veloce la realizzazione di un progetto. Questo infatti può essere portato avanti in parallelo, senza interferenze, risparmiando tempo e sforzi da parte dei programmatori.

D'altra parte, il FORTRAN presentava anche diversi inconvenienti, alcuni dei quali molto importanti. Per cominciare, possiamo ricordare un paio di aspetti poco brillanti, ma abbastanza circoscritti. Il modo di scrivere i programmi, cioè quella che si chiama la *sintassi del linguaggio*, ovvero le sue regole istitutive, era molto rigido, legato al fatto che i programmi venivano scritti sulle schede perforate e da queste passati all'elaboratore. Le istruzioni dovevano partire dalla settima posizione della scheda, mentre se avevano un'etichetta numerica (come il 150 visto sopra), questa doveva partire dalla prima posizione. Si potevano usare solo le lettere maiuscole. Internamente, le strutture dati utilizzabili erano molto semplici, in pratica limitate a vettori e matrici.

Un inconveniente più rilevante era il fatto che un programma non potesse richiamare sé stesso per la valutazione di situazioni man mano più semplici. Questa caratteristica, detta *ricorsività*, è molto utile in varie situazioni. Ad esempio, il fattoriale di un numero N è il prodotto di tutti i numeri interi da 1 fino ad N. Così:

$$4! = 1 \times 2 \times 3 \times 4 = 24,$$

e sarebbe facile scrivere un programma per valutare $n!$ se tale programma avesse la possibilità di utilizzare sé stesso per valori sempre più piccoli dell'argomento. Per definizione si pone:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \times (n-1)! & \text{altrimenti} \end{cases}$$

e quindi il calcolo di $n!$ si riduce a quello di $(n-1)!$; procedendo così fino a che non si arriva a dover valutare 0!, che vale appunto 1. Ad esempio, per calcolare 3!, si procede in questo modo:

$$3! = 3 \times 2! = 3 \times 2 \times 1! = 3 \times 2 \times 1 \times 0!;$$

poiché $0! = 1$, si ottiene il semplice prodotto:

$$3! = 3 \times 2 \times 1 \times 1 = 6$$

e 6 è il risultato finale, ovviamente corretto.

Sicuramente, però, il difetto più grave del FORTRAN è quello di aver mantenuto, pur nella novità del suo approccio, la logica dei salti del linguaggio macchina. Le etichette numeriche, che si potevano premettere alle istruzioni, servivano soprattutto come obiettivo dei salti, o *istruzioni GOTO*. Queste potevano dipendere da una condizione, ma il concetto era il medesimo del linguaggio macchina: abbandonare il normale flusso di esecuzione e passare a una zona diversa del programma. Questo, come s'è visto, era essenziale nella struttura interna dell'elaboratore, ma è anche una tecnica estremamente pericolosa. Infatti,

un salto fa perdere il controllo sul flusso del programma. Saltando in qualche parte delicata, si può sconvolgere completamente la logica di un programma e si possono mescolare parti che, invece, sarebbe stato bene mantenere distinte, sia da un punto di vista logico, sia pratico.

Il FORTRAN è stato il primo dei linguaggi di programmazione della terza generazione, ovvero dei linguaggi algebrici. Presto anche il COBOL si adeguò ai concetti del FORTRAN e in verità venne iniziato uno studio sistematico di quali dovessero essere le caratteristiche di un linguaggio di programmazione. Si era alla metà degli anni 1950 e un lavoro critico enorme fu portato avanti e si svilupparono tante idee che dovevano rivoluzionare la teoria e la pratica della programmazione. Il culmine di tutto ciò fu un linguaggio di programmazione, l'*ALGOL '60*, che pubblicato appunto nel 1960, sconvolse la programmazione e i modi di programmare, anche se, come linguaggio, non è mai stato praticamente usato.

Intanto, l'ALGOL adottò una definizione formale di propri costrutti, che permise un approccio scientificamente privo di ambiguità alla scrittura dei programmi. Il linguaggio definisce la propria sintassi per mezzo della teoria dei *linguaggi formali*, ideata da Chomski alla metà degli anni 1950. Il concetto di *grammatica formale*, soprattutto di *grammatica regolare* e di *grammatica libera dal contesto*, furono gli strumenti che permisero di creare strutture linguistiche formalmente corrette, non ambigue e che, finalmente, si potessero sganciare dal supporto su cui venivano scritte. Queste strutture, inoltre, rendevano chiari i modi di traduzione del programma nel linguaggio macchina, cioè chiarivano formalmente la *semantica computazionale* del programma scritto dall'utente.

Lo studio teorico che aveva portato alla definizione dell'ALGOL chiarì, in modo quasi definitivo, le caratteristiche cui dovevano soddisfare i linguaggi di programmazione. Ad esempio, si capì quale debba essere l'*ambito di validità* dei nomi usati in un programma: nomi dei dati, dei sottoprogrammi, dei parametri, etc.. Specie con la divisione di un progetto in sottoprogrammi, affidati a soggetti diversi, ci si era resi conto che i nomi potevano creare conflitti, se non venivano dati criteri generali per il loro utilizzo. Pietro non può usare il nome A per un certo dato, se Paolo lo usa per un altro, a meno di non dare un significato locale ai nomi usati da un singolo programmatore. All'opposto, sarà opportuno che Pietro e Paolo usino lo stesso nome per un dato che sia a comune tra i loro sottoprogrammi, per evitare che questi non si comprendano l'un l'altro.

Ma a parte certi aspetti vagamente filosofici, anche se fondamentali per la comprensione dei programmi e per la collaborazione, l'ALGOL chiarì molti punti di tipo più tecnico, relativi alla programmazione. Intanto, eliminò i salti dalla scrittura dei programmi. Per la verità, lasciò il costrutto del GOTO analogo al FORTRAN, ma dimostrò che esso è inutile e invitò i programmatori a lavorare senza salti, sostituendoli con istruzioni condizionali, iterative e ricorsive. Queste possono sostituire completamente i salti ed eliminano quell'aspetto particolarmente pericoloso implicito nei GOTO, e cioè la difficoltà di controllare il flusso corretto di un programma.

Le *istruzioni condizionali* permettono di fare una scelta ed eseguire una specifica parte di un programma in base alla decisione presa. La tipica istruzione condizionale è la IF: viene controllata una condizione (ad esempio, se un certo risultato parziale è zero) e si esegue una serie di istruzioni di macchina se essa è verificata; se ne esegue un'altra in caso contrario:

IF *condizione* THEN *serie1* ELSE *serie2* FI;

Questo evita di introdurre salti a livello del linguaggio, ed eseguita una delle due parti, qualunque essa sia, il programma prosegue con il comando successivo. Ad esempio, per sommare a C il più grande fra A e B si scrive:

IF A > B THEN D=A+C ELSE D=B+C FI;

La IF prende in considerazione condizioni binarie, che cioè possono essere vere o false, ma esistono istruzioni condizionali che lavorano su espressioni a più valori. Ad esempio, un programma può eseguire tre sequenze diverse a seconda che una variabile V assuma uno dei valori 1, 2 oppure 3.

Le istruzioni condizionali sono, comunque, istruzioni che seguono un flusso sequenziale. Abbiamo visto che spesso serve iterare l'esecuzione di certe parti di un programma. A questo serviva il DO del FORTRAN e sulla base di quello si sono studiate le istruzioni iterative. La teoria dell'ALGOL mostrò che esistono almeno due costrutti iterativi, a seconda che si conosca a priori quante volte l'iterazione debba avvenire, oppure la ripetizione sia basata su una qualche condizione che può essere modificata durante la sua esecuzione. Il DO fa parte del primo caso ed oggi, normalmente, è detto *costrutto FOR*, essendo più spesso indicato con questa parola:

S=0; FOR I=1 TO N DO S=S+Q[I] OD;

La parola OD, come prima la parola FI, sono "simpatici" modi per avvertire che il costrutto, DO oppure IF, è terminato. L'altro è detto *costrutto WHILE*, e possiede una variante nota come *costrutto REPEAT*, ma che qui lasciamo perdere. Ad esempio, se si vogliono sommare tutti gli elementi di una sequenza Q[1], Q[2], ..., Q[N] fino a che non se ne trova uno che vale 0, si fa:

```
S=0; I=1;
WHILE Q[I] ≠ 0 DO S=S+Q[I]; I=I+1 OD;
```

Il metodo più sofisticato per ripetere le istruzioni, anzi addirittura interi programmi, è la ricorsività o *ricorsione*. Il seguente esempio calcola il fattoriale di un numero I. La parola FUNCTION indica un sottoprogramma che calcola un valore ben preciso (in questo caso, $I!$); la parola RETURN indica la fine fisica del sottoprogramma:

```
FUNCTION FATTOR(I);
IF I=0 THEN FATTOR=1 ELSE
FATTOR=I*FATTOR(I-1) FI;
RETURN;
```

Questo è il caso del fattoriale, ma la gestione di molte strutture dati, come le liste e gli alberi che vedremo, si fa utilizzando in modo sistematico la ricorsione. Le liste e gli alberi non esistevano, né potevano esistere, in FORTRAN e fecero il loro ingresso sistematico nella programmazione soltanto dopo i chiarimenti portati dall'ALGOL. Per il momento, ricordiamo che il problema della ricorsione non era tanto la sua utilità, o la sua adeguatezza, quanto piuttosto la possibilità di realizzarla in modo efficiente. Si capì che ciò poteva essere fatto utilizzando una struttura dati detta *stack*, o, in italiano, *pila*. La sua realizzazione in termini di linguaggio macchina portò ad una facile implementazione della ricorsione.

La pila prende il suo nome da quella dei piatti: il primo piatto che si usa è l'ultimo che è stato deposto sulla pila. In inglese, questa struttura si dice anche *LIFO*: Last IN, First OUT, cioè l'ultimo arrivato è il primo a uscire. Si pensi al calcolo di 5 fattoriale. Dobbiamo ricordare il 5 e calcolare 4 fattoriale. Per far questo dobbiamo ricordare il 4 e calcolare 3 fattoriale. E così via. Con una pila a disposizione, l'elaboratore può inserirvi 5, 4, 3, 2, 1 nell'ordine (vedi Tabella 4.4). Quando arriva a dover calcolare 0 fattoriale, sa che questo vale 1 e può cominciare ad eseguire le varie moltiplicazioni con gli elementi della pila. Il primo è 1 e calcola $1 \times 1 = 1$. Il secondo, ora diventato primo, è 2 e $1 \times 2 = 2$. Si ha poi il 3 e $2 \times 3 = 6$. Così si arriva a calcolare $24 \times 5 = 120$. Qui la pila termina e questo è il risultato. Naturalmente corretto.

L'ALGOL '60 non fu praticamente usato da nessun programmatore, ma rivoluzionò il modo di concepire e di progettare i linguaggi di programmazione. Il suo figlio più noto è stato il *PASCAL*, che inglobava tutte le caratteristiche dell'ALGOL in un contesto ancora più razionale. Per questo, ancora oggi, il PASCAL rimane insuperato da un punto di vista didattico. In campo aziendale, la lezione dell'ALGOL è stata recepita dal *C*, il linguaggio col quale è stato costruito *UNIX*, il sistema operativo più famoso, anche se non il più usato. Ma, in generale, bisogna dire che tutti i linguaggi di programmazione più moderni si rifanno all'ALGOL per la loro struttura, anche se fanno riferimento a un modo un po' diverso di concepire la programmazione, e cioè la così detta *programmazione ad oggetti*.

Per avere un'idea di cosa si intenda per *oggetto*, si consideri un insieme di dati omogenei o, come si dice, a un *tipo di dati*. Se pensiamo, ad esempio, agli interi, sappiamo che su di essi si possono eseguire certe operazioni, come somme, divisioni o potenze, e non altre, come radici o logaritmi. In modo simile, se pensiamo alle sequenze di caratteri, possiamo concatenarle o isolarne una sottosequenza, ma non possiamo né moltiplicarle né dividerle. Un oggetto è un tipo di dato con le operazioni che ad esso competono. La programmazione ad oggetti parte da oggetti come quelli appena visti e permette di definirne quanti se ne vuole e di complessità qualsiasi. Risolvere un problema - significa allora individuare gli oggetti che lo modellano, cioè le strutture dei dati e le relative operazioni che permettono di gestirli secondo le specifiche del problema realizzate come programmi.

Un problema complesso richiede la costruzione di molti oggetti, alcuni dei quali assai complicati. Ad esempio, se si devono preparare gli stipendi per i dipendenti di un'azienda, potrà esistere un oggetto "dipendente" corredato da una serie di operazioni per valutarne la posizione all'interno dell'azienda, la consistenza familiare, il tipo di tassazione e di ritenute cui deve sottostare, e così via. Un oggetto può avere dei sotto-oggetti come, in questo caso, dirigenti, impiegati, operai, che *ereditano*, cioè condividono, molte delle caratteristiche del dipendente, aggiungendone semmai delle nuove. Non possiamo addentrarci in proprietà troppo specifiche, ma oggi un ben noto linguaggio di questo tipo è *JAVA* che, almeno teoricamente, ha la pretesa di funzionare allo stesso modo su tutti gli elaboratori.

Questo ci riporta al punto di partenza, e cioè al fatto che i linguaggi della terza generazione volevano poter funzionare su ogni elaboratore, al contrario di quelli della prima e della seconda, legati a una macchina specifica. Questo non sempre risulta vero, e certo un punto in cui differiscono è l'ingresso e l'uscita dei dati. Quando, spinti dai sistemi di gestione delle basi di dati, si volle standardizzare anche questo aspetto, nacquero i *linguaggi della quarta generazione*. Questi erano basati sui linguaggi della terza generazione, ma erano arricchiti da comandi per la gestione dei dati, soprattutto quelli residenti nella memoria secondaria. Oggi si confondono con i *sistemi per la gestione delle basi dei dati* e, come struttura generale, non sono diversi dai linguaggi della terza generazione.

Un certo successo hanno trovato anche i *linguag-*

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | 0! | 1 | | | | | |
| | | | | 1! | 1 | 1 | 1 | | | | |
| | | | 2! | 2 | 2 | 2 | 2 | 2 | | | |
| | | 3! | 3 | 3 | 3 | 3 | 3 | 3 | 6 | | |
| | 4! | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 24 | |
| 5! | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 120 |

Tabella 4.4: Andameno della pila per il calcolo di 5!

*gi della quinta generazione*, o *linguaggi dichiarativi*. Nati da un linguaggio molto speciale, il *LISP*, sviluppato alla fine degli anni 1950, sono stati usati, e si usano ancora, per affrontare problemi di *intelligenza artificiale*. Si tratta di programmi che, in vario modo, tentano di simulare il comportamento della mente umana, cioè tentano di essere programmi intelligenti. Molte ricerche sono state fatte in questo settore, e vanno, un po' impropriamente, dal gioco degli scacchi al supporto per le decisioni, dalla diagnosi medica all'apprendimento, per mezzo ad esempio delle *reti neurali*. All'inizio degli anni 1980 si pensò che potessero sostituire tutti gli altri metodi di programmazione. Nel seguito sono stati molto ridimensionati.

Può essere interessante capire cosa si intende per *programmazione dichiarativa*, contrapposta alla *programmazione imperativa* propria delle prime quattro generazioni. Come abbiamo visto dagli esempi, la programmazione tradizionale consiste nell'ordinare all'elaboratore di eseguire la sequenza di istruzioni che gli è stata assegnata: somma A e B, ripeti questa istruzione per 100 volte, e così via. Da questo deriva il termine "imperativa". Per fare un esempio di natura diversa, è come se prendessimo un taxi e, saliti, cominciassimo a dire al conducente: vada dritto fino al semaforo; volti a destra; fatti trecento metri volti a sinistra, e così di seguito. Probabilmente, il tassista vi manderà a quel paese e certo farebbe molto meglio se gli diceste semplicemente che volete andare alla stazione.

Analogamente, la programmazione dichiarativa permette all'utente di "dichiarare" all'elaboratore che cosa intende fare, e lascia che sia la macchina a cercare e trovare la soluzione migliore che soddisfi la richiesta. In luogo di una serie di comandi, l'elaboratore riceve una richiesta e mette in ballo tutta la sua esperienza (cioè, tutto il suo software) per rispondere nel modo migliore possibile. Questo non è banale, poiché significa che l'elaboratore deve conoscere quelle strade che portano alla risoluzione del problema. Come curiosità, ricordiamo che sui linguaggi dichiarativi si basò la celebra sfida informatica giapponese degli anni 1980. Purtroppo, non ebbe il successo sperato.

# Capitolo 5

# ALGORITMI E STRUTTURE DATI

## 5.1 Algoritmi di base

Programmare è un'attività prettamente informatica, ma non dimentichiamo che, in fondo, non si tratta d'altro che di un lavoro di traduzione. Consiste, infatti, nel traslare, in quella lingua franca che è un linguaggio di programmazione, idee originariamente espresse in formule matematiche, in descrizioni operative o, comunque, in procedure di regola piuttosto formalizzate. Per questo, la programmazione è spesso lasciata, anche se non sempre, a persone specializzate, ma non di alto livello, di solito giovani appena assunti. La vera sostanza di un programma sta nel lavoro a monte cha ha portato alla formula o alla procedura formalizzata. Questo lavoro, detto di analisi, è il vero nucleo dell'opera di un informatico.

Quando si pensa di poter affrontare e risolvere un problema per mezzo dell'Informatica, gli analisti informatici lo studiano e cercano di trovare un metodo di soluzione. Questo deve essere ben formulato, senza ambiguità o ridondanze, meglio se per mezzo di equazioni matematiche, e deve portare alla soluzione in un tempo finito, possibilmente ragionevole. Ragionevole vuol dire adeguato al problema. Se si tratta di svolgere i calcoli per le strutture di un grattacielo, possono essere adeguati anche due o tre giorni di computazione. Se si tratta di recuperare i dati di un cittadino, dati da inserire nella carta di identità, cinque secondi sono troppi. Vedremo che esistono problemi per i quali si può arrivare alla soluzione solo con oltre 250 anni di calcoli: questo può avere un interesse teorico, ma non certo pratico.

Un metodo di risoluzione come quello appena descritto si dice un *algoritmo*. Parafrasando, un algoritmo è un metodo che ci porta alla risoluzione di un problema in un tempo finito e che sia formulato in maniera non ambigua (e possibilmente non ridondante). La ricerca e lo studio degli algoritmi è il vero lavoro degli informatici. Un algoritmo ben formulato facilita la fase di programmazione. Inoltre, e questo è un punto molto importante, un algoritmo è indipendente dal particolare elaboratore su cui verrà realizzato il programma corrispondente. Se un giorno si dovesse risolvere lo stesso problema su una macchina nuova usando un nuovo linguaggio, l'algoritmo resterebbe comunque valido e non sarebbe necessario eseguire di nuovo l'analisi del problema. I programmatori possono eseguire il loro lavoro sull'algoritmo esistente, risparmiando tante risorse.

D'altra parte, un problema può avere diverse soluzioni, e quindi si possono pensare vari algoritmi che raggiungono il medesimo scopo. E' naturale allora chiedersi se ce n'è uno migliore e che quindi va preferito agli altri. Sarà tale algoritmo quello che merita di passare alla fase della programmazione e quindi all'uso effettivo. Questo succede abbastanza di frequente, e allora si instaura un processo di analisi che ha lo scopo di individuare l'algoritmo che meglio risolve un determinato problema. L'*Analisi degli algoritmi* è una disciplina ad alto contenuto matematico. Sono stati infatti proposti metodi molto sofisticati e che fanno uso di complessi strumenti matematici. Noi, naturalmente, ci limiteremo a vedere qualche semplice esempio.

Nelle lezioni precedenti abbiamo avuto modo di incontrare molti esempi di algoritmo. Primi tra tutti sono stati gli algoritmi che ci permettono di eseguire le quattro operazioni elementari e che abbiamo abbastanza formalmente specificato per i numeri scritti in base 2. Storicamente, è stato proprio questo il primo uso della parola "algoritmo". Quando nel 1201 Leonardo Fibonacci introdusse in Italia la numerazione arabica, in onore del matematica arabo Al-Kuwarizmi chiamò *algorismi* i metodi di effettuazione delle operazioni elementari. Per secoli, "imparare gli algorismi" volle significare imparare l'aritmetica pratica. Poi, piano piano, il termine si è allargato a indicare qualunque metodo, non ambiguo e finito, di risoluzione di un problema.

Ricordiamo anche, in modo esplicito, gli algoritmi di conversione da una base all'altra. Abbiamo poi fissato un po' la nostra attenzione sugli algoritmi di cifratura delle informazioni e abbiamo insistito sugli algoritmi di compressione. Anche le tecniche di segnalazione degli errori, come quelle di autocorrezione, sono algoritmi. Dappertutto abbiamo incontrato al-

goritmi, poiché dappertutto troviamo problemi. Naturalmente, esistono anche problemi che non siamo riusciti a risolvere e problemi per i quali, addirittura, si può dimostrare che non esistono soluzioni algoritmiche. Questi problemi hanno per lo più un'apparenza innocente, e su di essi torneremo più tardi, perché l'argomento è piuttosto delicato.

Spesso è la Matematica che ci offre metodi per la risoluzione dei problemi. Se questo è il caso, le formule di risoluzione si dicono un *modello matematico* del problema. Ad esempio, un'equazione o un sistema di equazioni, un'equazione differenziale o un sistema di disequazioni con una funzione obiettivo, costituiscono modelli matematici completi di tanti problemi economici, ingegneristici, fisici e biologici. Altre volte occorre introdurre valutazioni e concetti statistici. In generale, molti di questi metodi vanno riuniti in un sistema unico che costituisce, di conseguenza, un modello complesso del problema. Talvolta, purtroppo, possiamo essere ridotti a cercare la soluzione valutando separatamente i vari casi in cui il problema è divisibile, e queste sono le situazioni peggiori, almeno da un punto di vista informatico.

Volendo riassumere in cosa consiste il lavoro di un informatico, possiamo per il momento dire questo. Quando si decide di procedere all'automazione di una certa parte della realtà, grande o piccola che sia, la si comincia a studiare cercando di capire quali sono le quantità da trattare e come devono essere trattate per poter tenere sotto controllo e per poter incidere sulla realtà considerata. I due aspetti - dati o informazioni da una parte, e *procedure* o *funzionalità* dall'altra - sono essenziali e complementari. In particolare, l'aspetto procedurale è quello che guida l'informatico e, in funzione di quello, spesso vengono elaborate particolari strutture per gestire i dati. Inoltre, le procedure devono essere non ambigue ed eseguibili in un tempo ragionevole, cioè devono corrispondere ad algoritmi. Compito dell'informatico è quindi quello di interpretare la realtà in termini di algoritmi.

Usando una terminologia specifica, la prima fase del lavoro di un informatico è quella dell'*analisi della realtà*, cioè del problema che si vuole affrontare. Capite quali sono le quantità coinvolte e le funzionalità necessarie, si passa alla *progettazione*, cioè alla traduzione dei dati e delle procedure in termini algoritmici. Questo deve avvenire come se l'elaboratore non esistesse, o esistesse solo come entità astratta. Il progetto, in questo modo, sarà indipendente dall'elaboratore e continuerà a valere anche se la tecnologia informatica dovesse cambiare radicalmente. Si parla per questo di *progettazione logico/concettuale*. Su questa si basa la fase successiva, detta *progettazione fisica*, che invece adatterà gli algoritmi trovati all'elaboratore che abbiamo a disposizione.

La progettazione fisica permette la suddivisione del lavoro e chiarisce le linee guida di come deve avvenire la realizzazione dei programmi, cioè del software, la loro validazione e infine il loro utilizzo operativo. Naturalmente, la fase operativa è l'obiettivo finale del progetto informatico, ma di regola non ne è l'ultima fase. Il software deve essere continuamente rivisto e aggiornato, sia per correggere eventuali errori sfuggiti a ogni controllo, sia soprattutto per tenerlo al passo con le nuove esigenze che la realtà ci impone nel suo continuo evolversi. Tutto questo processo si chiama spesso *ciclo di vita del software* e su di esso torneremo con maggiori dettagli, quando parleremo dei grandi progetti informatici.

D'altra parte, il ciclo di vita del software si ritrova, pari pari, nei grandi quanto nei piccoli progetti dell'Informatica. Se volete ad esempio gestire il vostro bilancio con un foglio EXCEL, dovete, prima di tutto, capire cosa gestire e come gestirlo: entrate (di solito poche) e uscite per spese e bollette: Volete registrare anche le motivazioni delle spese e degli incassi? Volete trattare qualche piccolo investimento? Le procedure sono di regola semplici somme o sottrazioni, ma volete vedere il saldo dopo ogni operazione? Volete un bel rapporto mensile, trimestrale o annuale? Le necessità e la fantasia giocano un ruolo importante. E' bene fare un progetto un po' più astratto e generale di quello legato a EXCEL, poiché non si sa mai. E una volta realizzato, pensate di non doverlo toccare mai più? Sarete voi stessi a proporvi qualche miglioramento!

Di solito, ogni problema ha il suo algoritmo o i suoi algoritmi di risoluzione. Tuttavia, certi problemi, semplici nella loro formulazione, si ritrovano molto spesso, eventualmente come sottoproblemi di situazioni più complesse. Per la loro frequenza e la loro importanza vengono allora studiati in modo indipendente da un possibile contesto, in modo da arrivare a soluzioni ottimali oppure a soluzioni alternative da adottare in particolari circostanze. Specificamente, in Informatica si considerano due problemi, la *ricerca* e l'*ordinamento* dei dati, come fondamentali, e si dicono *di base* gli algoritmi che li risolvono. E' difficile trovare problemi che non richiedano, in un punto o in un altro, la ricerca di qualche informazione specifica, così come spesso capita di dover ordinare i risultati ottenuti: si pensi a una lista di cento persone e al problema di doverla leggere se non è ordinata per cognome e nome.

Il problema della ricerca è universale. In un'azienda, per soddisfare l'ordine di un cliente occorre cercare il prodotto richiesto in un catalogo. All'Università, per registrare l'esito di un esame occorre ricercare i dati dello studente che l'ha sostenuto. In un ospedale,

| | | |
|---|---|---|
| 1 | ANNA | (4) |
| 2 | BERNARDO | (3) |
| 3 | BRUNO | (4) |
| 4 | CATERINA | (2) |
| 5 | DANIELE | (4) |
| 6 | ELISA | (3) |
| 7 | FRANCO | (4) |
| 8 | LUCIA | (1) |
| 9 | MARIA | (4) |
| 10 | MARTA | (3) |
| 11 | PAOLO | (4) |
| 12 | RACHELE | (2) |
| 13 | RAFFAELLA | (4) |
| 14 | SARA | (3) |
| 15 | VINICIO | (4) |

Tabella 5.1: Un elenco di 15 nomi

per visitare un paziente bisogna ritrovare i risultati delle visite precedenti. E così via. In generale, nell'elaboratore è memorizzato l'elenco dei prodotti, degli studenti o dei pazienti - e occorre scorrere tale elenco fino a trovare ciò che ci interessa. L'elaboratore, con la sua velocità, ci permette di effettuare la ricerca in pochi istanti. Purtroppo, se procediamo in modo ingenuo controllando l'elenco un elemento dopo l'altro, e l'elenco è lungo, i pochi istanti diventano lunghi secondi, e non è piacevole stare ad attendere, specie quando ci si aspetta una grande efficienza.

Si pensi, per fissare le idee, all'anagrafe di una città con un milione di abitanti. Se il signor Zucchelli vuole un certificato, ci possono volere anche 20 secondi per cercare il suo nome - partendo da Abati e Abbienti. In media, mettendo assieme i casi favorevoli e quelli sfavorevoli, occorreranno circa 500 mila prove prima di aver successo con la nostra ricerca. Questo metodo ingenuo si dice *ricerca sequenziale*, e, di fronte a un elenco di $n$ elementi, la media di operazioni che deve fare è circa $n/2$. Un metodo certamente più intelligente e veloce consiste nella cosiddetta *ricerca binaria*. Supponiamo, come d'altra parte abbiamo già fatto, che il nostro elenco sia ordinato alfabeticamente, con Abati all'inizio e Zucchelli alla fine. Vediamo cosa si può fare.

Usiamo come esempio di lavoro un elenco di soli 15 nomi (vedere la Tabella 5.1), e supponiamo di voler cercare MARIA. La ricerca sequenziale confronta MARIA col primo elemento ANNA e, visto che non sono uguali, procede nella ricerca con gli altri 14 elementi. Se l'elenco è ordinato, un metodo più furbo di procedere è quello di confrontare MARIA con l'elemento centrale della tabella, cioè l'ottavo. Se, come in questo caso, MARIA è successivo, sappiamo immediatamente che MARIA non si può trovare nella prima parte dell'elenco: essendo successivo a LUCIA è, a maggior ragione, successivo a tutti gli elementi che precedono LUCIA. In un colpo solo scartiamo metà del nostro elenco. Se MARIA fosse stata precedente all'elemento centrale, avremmo scartato la seconda metà dell'elenco, ma sempre ci saremmo ridotti a cercare tra metà soltanto degli elementi.

Siamo quindi rimasti con gli elementi dal numero 9 al numero 15, e possiamo adottare la stessa strategia, che promette così bene. Se conosciamo la posizione del primo e dell'ultimo elemento di una parte dell'elenco, l'elemento centrale è dato dalla loro semisomma. In questo caso è $(9+15)/2 = 12$, e confrontiamo perciò MARIA con RACHELE: il risultato ci permette di eliminare tutti i nomi da RACHELE in avanti, e limitarci agli elementi dal 9 all'11. L'elemento centrale è ora $(9 + 11)/2 = 10$. Confrontando MARIA e MARTA scartiamo ancora gli elementi 10 ed 11. Non rimane che confrontare MARIA con l'elemento di posizione 9 e trovare finalmente ciò che cercavamo. Abbiamo effettuato 4 confronti in tutto, contro i 9 che sarebbero stati necessari con la ricerca sequenziale.

Questo non è un caso, ed è facile rendersi conto che quattro confronti sono il massimo possibile. Come si vede nella terza colonna della Tabella 5.1, LUCIA richiede un solo confronto e la troviamo al primo tentativo. CATERINA e RACHELE necessitano di due confronti, e così via. Se facciamo la media, troviamo 3.27 confronti, contro gli 8 della ricerca sequenziale. E si osservi che quattro confronti sono sufficienti anche quando l'elemento non si trova nell'elenco, un'eventualità che non si può scartare a priori: il cittadino potrebbe essersi sbagliato o il paziente essere alla prima visita. La ricerca sequenziale, in questo caso, va avanti finché non arriva in fondo, o almeno a superare l'elemento che lo seguirebbe. In pratica, la ricerca binaria riesce a dividere l'elenco in due parti, una delle quali può essere ignorata e l'altra trattata ancora nella stessa maniera. Il problema è così ridotto a problemi più semplici e può essere risolto più facilmente. Questo metodo si dice *divide et impera* e permette di affrontare tanti problemi, che trovano così una soluzione molto efficiente.

Vogliamo allora scoprire qual è il massimo numero di confronti richiesto dalla ricerca binaria quando l'elenco contiene un numero generico $n$ di elementi. Come abbiamo appena osservato, al primo confronto si riesce a trovare un solo elemento, quello centrale. Al secondo confronto se ne trovano due, o, se si preferisce, ci sono due elementi che si riescono a trovare con due confronti. Così, ve ne sono quattro che si trovano con tre confronti ed otto con quattro. Questo significa che vi sono $2^{k-1}$ elementi che si trovano con $k$ confronti. Vi sono pertanto:

$$1 + 2 + 4 + ... + 2^{k-1} = 2^k - 1$$

elementi che si trovano con al più $k$ confronti. Ma

se questa quantità supera $n$, $k$ è il massimo numero di confronti per trovare un qualsiasi elemento. Da questa relazione $2^k - 1 > n$ si ricava $k \geq \log_2(n+1)$. Per definizione, $\log_2(y)$ è quel numero $x$ tale che $2^x = y$. Per $n = 15$ si trova proprio $k = 4$, in accordo con quanto abbiamo appena osservato col nostro esempio.

Il $\log_2(n)$ è una quantità piccola nei confronti di $n$, poiché deve stare all'esponente di 2 per riportarci ad $n$. Ad esempio, se $n$ è un milione, il suo logaritmo è circa 20. Questo significa che per trovare un elemento nella nostra anagrafe di un milione di abitanti sono necessari, al più, 20 confronti, un bel po' di meno di quel mezzo milione che occorrono in media con la ricerca sequenziale. Questo rende la ricerca binaria molto conveniente e con essa, in effetti, qualunque ricerca può essere eseguita in una frazione di secondo. Anche se arrivassimo a un elenco con un miliardo di elementi, il numero di confronti sarebbe comunque limitato a trenta, e questo è davvero un risultato notevole.

Esistono altri metodi per effettuare ricerche veloci, ed uno lo vedremo nella prossima lezione. L'unico vero inconveniente della ricerca binaria è che l'elenco deve essere ordinato, ma la ricerca binaria rimane uno degli algoritmi di base più interessanti ed usati. Preoccupiamoci allora di trovare un algoritmo che ordini gli elementi di un elenco quando questi, come succede normalmente, sono disposti a casaccio.

La tecnica che ora vedremo non è certo tra le migliori, ma ci dà un'idea di come certe cose possono essere fatte in modo molto semplice, anche se non ottimale. Nella prossima lezione vedremo come si possono mettere le cose in modo da prendere due piccioni con una fava: trovare un metodo che permetta di ordinare gli elementi di un elenco in modo veramente veloce e, nel contempo, ci permetta anche di cercare e trovare un elemento in un tempo paragonabile a quello della ricerca binaria.

Si consideri un elenco disordinato. Possiamo immaginarlo composto di numeri, che prendono meno spazio, ma nulla cambierebbe se invece l'elenco fosse composto di nomi:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 25 | 54 | 37 | 48 | 63 | 40 | 22 | 45 | 58 | 32 |

Vogliamo ordinare questi elementi dal più piccolo al più grande. Cominciamo allora a scandire gli elementi dell'elenco, confrontando il primo elemento col successivo e procedendo verso destra. Se il primo elemento è minore del secondo, lasciamo le cose come stanno, altrimenti scambiamo le posizioni dei due elementi. Così operando, possiamo dire di aver fatto un piccolo passo verso l'ordinamento totale, che pur rimane sempre lontano. Nel nostro esempio, 25 e 54 non si scambiano, mentre, procedendo, si devono scambiare di posizione 54 e 37, prima, e 54 e 48 poi. A questo punto, 54 e 63 non si devono scambiare:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 25 | 54 | 37 | 48 | 63 | 40 | 22 | 45 | 58 | 32 |
| 25 | 37 | 54 | 48 | 63 | 40 | 22 | 45 | 58 | 32 |
| 25 | 37 | 48 | 54 | 63 | 40 | 22 | 45 | 58 | 32 |
| 25 | 37 | 48 | 54 | 40 | 63 | 22 | 45 | 58 | 32 |
| 25 | 37 | 48 | 54 | 40 | 22 | 63 | 45 | 58 | 32 |
| 25 | 37 | 48 | 54 | 40 | 22 | 45 | 63 | 58 | 32 |
| 25 | 37 | 48 | 54 | 40 | 22 | 45 | 58 | 63 | 32 |
| 25 | 37 | 48 | 54 | 40 | 22 | 45 | 58 | 32 | 63 |

Man mano che si trovano numeri più grandi, sono questi che vengono a spostarsi verso destra. Quando troviamo il più grande di tutti (anche se noi non sappiamo che è tale), esso si sposta sempre di più fino ad arrivare in fondo all'elenco. E' quello che capita al 63 che, piano piano, arriva in fondo. Come risultato della scansione completa abbiamo due risultati: molti scambi hanno portato a un miglioramento dell'ordine (anche se in misura modesta), e, soprattutto, l'elemento più grande è finito in fondo all'elenco, che è il posto che gli compete.

Effettuiamo allora un'altra scansione, escludendo però l'ultimo elemento, che ormai sappiamo essere a posto. Questa volta è il penultimo elemento a finire nella penultima posizione, e questo va benissimo:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 25 | 37 | 48 | 54 | 40 | 22 | 45 | 58 | 32 | **63** |
| 25 | 37 | 48 | 40 | 54 | 22 | 45 | 58 | 32 | **63** |
| 25 | 37 | 48 | 40 | 22 | 54 | 45 | 58 | 32 | **63** |
| 25 | 37 | 48 | 40 | 22 | 45 | 54 | 58 | 32 | **63** |
| 25 | 37 | 48 | 40 | 22 | 45 | 54 | 32 | 58 | **63** |

Procediamo così alla terza scansione, che possiamo estendere a tutti gli elementi, eccetto gli ultimi due, ormai al loro posto. Il terz'ultimo elemento raggiunge così la posizione che gli compete, e noi possiamo andare avanti con le scansione successive. Ormai, il modo di procedere dovrebbe essere chiaro:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 25 | 37 | 48 | 40 | 22 | 45 | 54 | 32 | **58** | **63** |
| 25 | 37 | 48 | 40 | 22 | 45 | 32 | **54** | **58** | **63** |
| 25 | 37 | 40 | 22 | 45 | 32 | **48** | **54** | **58** | **63** |
| 25 | 37 | 40 | 22 | 32 | **42** | **48** | **54** | **58** | **63** |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **22** | **25** | **32** | **37** | **40** | **42** | **48** | **54** | **58** | **63** |

Man mano, si aumenta il numero di elementi che vanno a finire al proprio posto e si accresce il grado di ordinamento. Spesso, si arriva all'ordinamento completo senza dover fare tutte le scansioni che, teoricamente, dovrebbero durare fino a che non ci siamo ridotti a un solo elemento che, per forza, deve essere il più piccolo. Comunque, l'elaboratore arriva alla conclusione e il nostro elenco risulta perfettamente ordinato.

La *complessità* di un algoritmo è la misura della sua efficienza. Se l'algoritmo deve gestire $n$ dati, la sua

complessità si misura come funzione di $n$. Tornando alla ricerca, se l'elenco contiene $n$ nomi o $n$ numeri, le operazioni da fare aumentano all'aumentare di $n$. Come s'è visto, nel caso della ricerca sequenziale il numero medio di confronti da effettuare è circa $n/2$, e questa è la complessità dell'algoritmo. Nel caso della ricerca binaria, i confronti sono, nel caso peggiore, $\log_2(n)$ e quindi, in media, saranno ancora di meno. Poiché il logaritmo è una funzione che cresce molto lentamente, la ricerca binaria è molto più efficiente della ricerca sequenziale. I numeri che abbiamo visto in precedenza sono molto significativi a questo proposito.

Vediamo allora la complessità del metodo di ordinamento che abbiamo illustrato. Prima di tutto, per meglio intenderci, diamogli un nome. In realtà, esso è noto come *ordinamento a bolle*, in inglese *bubble sort*: se immaginiamo che gli elementi siano le bollicine in un bicchiere di acqua minerale, i loro spostamenti assomigliano a quelli delle bollicine che pian piano vengono a galla; basta ruotare di 90 gradi il nostro elenco. A parte questa nota di fantasia, cominciamo con l'osservare che la prima scansione dell'elenco richiede esattamente $n-1$ confronti: infatti, il primo elemento va confrontato col secondo, questo col terzo, e così di seguito. Qualche volta facciamo anche uno spostamento, e il loro numero è minore di quello dei confronti, così che le operazioni sono al più $2(n-1)$.

La seconda scansione ignora l'ultimo elemento e quindi richiede $n-2$ confronti e, al massimo, $n-2$ spostamenti. La terza scansione cala il conto ad $n-3$ e così via. Alla fine, faremo un unico confronto e, forse, uno spostamento. Abbiamo pertanto:

$$1 + 2 + ... + (n-1) = \frac{n(n-1)}{2}$$

confronti e, se va male, altrettanti spostamenti. Abbiamo quindi, al più, una complessità di $n(n-1)$ operazioni. Quando $n$ è grande (basta pensare a 1000), questa quantità è un po' meno di $n^2$, esattamente 999·000. Pertanto, la possiamo assimilare ad $n^2$ e dire, come si dice, che l'algoritmo di ordinamento a bolle ha una *complessità quadratica*. Raddoppiando $n$, il numero di operazioni diviene quattro volte più grande.

Con la stessa terminologia, la ricerca sequenziale ha *complessità lineare*, e quando il numero degli elementi raddoppia, raddoppia anche il tempo della ricerca. A questo serve il concetto di complessità: a stabilire quanto veloce sia un algoritmo, anche se non si parla di tempi reali, ma solo di aspettative. Tuttavia, la complessità dà l'esatta previsione dei tempi relativi di esecuzione. Da questo punto di vista la ricerca binaria è eccezionale:

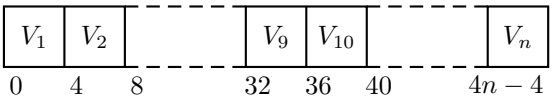
$$\log_2(2n) = \log_2(2) + \log_2(n) = \log_2(n) + 1.$$

Ciò significa che raddoppiando gli elementi dell'elenco, la complessità cresce di una sola operazione! Questa proprietà è caratteristica della *complessità logaritmica*.

Tutti questi esempi ci fanno capire come lo studio della complessità degli algoritmi ci possa dare informazioni importantissime.

## 5.2 Le strutture ad albero

I tre algoritmi che abbiamo visto: ricerca sequenziale, ricerca binaria e ordinamento a bolle, operano tutti su un elenco di elementi che, indifferentemente, possono essere numeri, nomi o altri dati, sui quali, però, si possano eseguire le operazioni di confronto. Un elenco è, dal punto di vista informatico, una sequenza di posizioni contigue nella memoria centrale dell'elaboratore. Ogni elemento corrisponde a un certo numero di byte: quattro per un numero intero, quattro, cinque o sei (o altri numeri, a seconda dell'elaboratore) per i numeri reali, $k$ per i nomi, se $k$ è la lunghezza che abbiamo stabilito a priori: ad esempio 20 se si tratta di nomi o cognomi. In questo modo, l'elaboratore sa calcolare facilmente dove si trova un certo elemento: il decimo intero è nella posizione 36, dato che si parte da 0:




E' proprio il fatto di conoscere la posizione di ciascun elemento ciò che rende agevole all'elaboratore l'esecuzione di questi algoritmi, ovvero dei corrispondenti programmi. Se ad esempio abbiamo un elenco di 8123 nomi di 22 caratteri ciascuno e vogliamo sapere da dove cominciare la ricerca binaria, occorre, come s'è detto, trovare l'elemento centrale (o mediano): $(8123+1)/2 = 4067$. I 4066 elementi che lo precedono occupano $4066 \times 22 = 89{\cdot}452$ byte e quindi si partirà proprio da questo byte (sempre considerando che si deve cominciare da 0) relativamente alla locazione di partenza dell'elenco. Nel gergo informatico, un elenco fatto di elementi contigui, di lunghezza fissa in numero di byte, si chiama un *vettore* ed è la più semplice delle strutture dati.

Si intende per *struttura dati* un'organizzazione dei dati sia in astratto, sia nella memoria centrale dell'elaboratore. Nel primo caso, legato alla formulazione degli algoritmi, si parla di *strutture dati astratte*, mentre nel secondo si parla di *strutture dati fisiche*, e naturalmente ci si riferisce ai programmi. La distinzione è corretta ed utile per gli informatici, ma risulta un po' capziosa per chi non è addetto ai lavori. Pertanto noi la ignoreremo in queste note, e parleremo sempre e solo di semplici strutture dati.

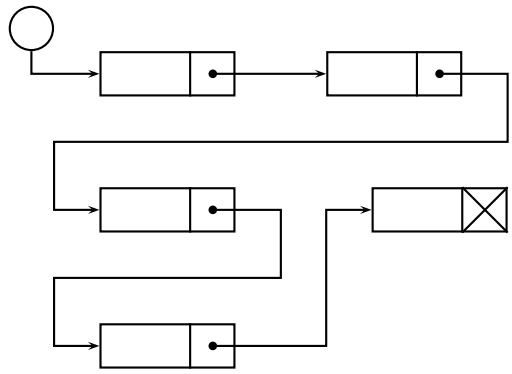


Figura 5.1: Una lista

Un altro esempio di struttura dati è costituito dalle *liste* (vedi Figura 5.1). Una lista, come un vettore, è una sequenza di dati, ognuno di lunghezza fissa predeterminata, ma che risiedono in posizioni non contigue della memoria. Si suppone allora, come d'altra parte succede, che ogni elemento della lista sia costituito da due parti: il dato e il collegamento, o *puntatore*, all'elemento successivo. L'ultimo elemento della lista ha, oltre al dato, un *puntatore vuoto*, cioè un collegamento fasullo che serve però all'elaboratore per capire quando la lista è terminata.

Il vantaggio della lista, nei confronti del vettore, è che i suoi elementi possono essere comunque sparsi nella memoria centrale, e questo dà una grande flessibilità alla struttura, contro l'intrinseca rigidità del vettore. Lo svantaggio, però, è che non è possibile sapere dove si trova un certo elemento, diciamo il ventesimo, proprio perché potrebbe trovarsi ovunque. La lista va scorsa e gli elementi contati, con gran perdita di tempo. Per questo, alle liste non è possibile applicare nessuno degli algoritmi visti, e questo è un bell'inconveniente!

Le liste hanno usi diversi dai vettori, anche se le due strutture sono utilizzate quasi con la stessa frequenza. Accenniamo ad un'applicazione importante, che sfrutta due capacità delle liste: quella di poter crescere indefinitamente e quella di poter inserire facilmente e velocemente un nuovo elemento nella prima posizione. Ciò, infatti, si fa semplicemente cambiando alcuni puntatori. Con una lista si ottiene una facile realizzazione della pila, la struttura dati che, come sappiamo, permette di effettuare il collegamento (o chiamata) fra sottoprogrammi. Inserire un nuovo elemento nella pila equivale ad aggiungere tale elemento nella prima posizione della lista. Usare il primo elemento della pila equivale a togliere (e trattare in qualche modo) proprio l'elemento della lista che si trova in prima posizione.

Naturalmente, le liste hanno molti altri usi. Possiamo addirittura ricordare che il primo linguaggio di programmazione proposto per l'elaborazione dei simboli (e non solo dei numeri, come si faceva di regola) è stato il LISP, che abbiamo già citato in relazione all'Intelligenza Artificiale. LISP era l'acronimo per *LISt Processor*, cioè "Elaboratore di Liste". Infatti, la sua unica struttura dei dati era la lista, ampliata a lista di liste, a lista di lista di liste, e così via. Le operazioni, quindi, erano solo relative alle liste: prendere il primo elemento, togliere il primo elemento, fare un'unica lista di due liste, confrontare due liste. E con questo semplice armamentario si possono risolvere problemi complicatissimi.

Il LISP è rimasto il prototipo dei linguaggi di programmazione rivolti all'Intelligenza Artificiale e all'elaborazione simbolica. L'Intelligenza Artificiale è la disciplina che tenta di far risolvere all'elaboratore problemi che l'elaboratore stesso analizza, al contrario della programmazione tradizionale, nella quale l'elaboratore risolve problemi già analizzati dall'uomo. Molto in auge fino agli anni 1990, ora ha preso strade molto diverse che vanno dai metodi per giocare agli scacchi alla robotica, dal riconoscimento di strutture come voci e immagini all'apprendimento, dal controllo dei sistemi complessi alla diagnosi medica, e così via. Sul LISP sono stati costruiti i linguaggi dichiarativi, che abbiamo ricordato, e che, per la loro struttura, hanno forti gradi di parentela con l'Intelligenza Artificiale.

Una struttura molto interessante, e che pertanto cercheremo di trattare con qualche più dettaglio, è costituita dagli *alberi*. Gli alberi dell'Informatica hanno forse più a vedere con gli alberi genealogici che con gli alberi della natura, ma l'idea comune è quella di avere una radice (o un tronco) da cui si dipartono diversi rami, che a loro volta producono altri rami, e così via, fino a creare una struttura complessa che, però, possa essere scandita, avanti ed indietro, passando di ramo in ramo. In realtà, gli alberi più usati in Informatica procedono di due rami in due rami. Più precisamente, sono costituiti da un nodo iniziale, detto *radice*, e da questa si dipartono due rami, ognuno dei quali raggiunge un nodo, dal quale partono due altri rami, e così di seguito. Si dicono, perciò propriamente, *alberi binari*.

Convenzionalmente, i due rami che partono da ciascun nodo si dicono di destra e di sinistra, e come tali si disegnano. Sempre seguendo un uso ormai consolidato, la radice si disegna in alto e i rami rivolti verso il basso. Questa, naturalmente, è una questione grafica e nulla ha a che fare con la rappresentazione degli alberi all'interno della memoria dell'elaboratore. Prima di descriverla, tuttavia, vogliamo osservare in modo esplicito che certi nodi possono avere solo il ramo di sinistra, altri solo quello di destra. Alcuni, poi, mancano dell'uno e dell'altro, per cui, con loro,

l'albero arresta la sua crescita: questi nodi si dicono *foglie*. Essendo l'albero una struttura finita, se procediamo di nodo in nodo verso il basso, dobbiamo sempre arrivare ad una foglia.

All'interno della memoria, l'albero condivide molte delle caratteristiche delle liste. C'è un nodo iniziale che, come abbiamo detto, si chiama radice. Ogni nodo contiene un'informazione: possiamo avere un albero di numeri, un albero di nomi, uno di cognomi, e così via. Si suppone che tutti i nodi contengano lo stesso tipo di informazione. Il nodo contiene anche due puntatori, che rappresentano i due rami di sinistra e di destra. I puntatori permettono di collegarsi ai nodi successivi, che si dicono *figli* del nodo considerato. Questo, a sua volta, è detto il *padre*. Scopo dei puntatori è quello di permettere una collocazione dei nodi in posti qualsiasi della memoria; così, come e più della lista, un albero binario è una struttura molto flessibile.

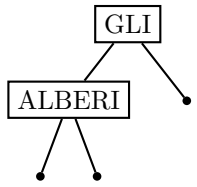
Gli alberi informatici hanno usi molto ampi e differenziati, ma probabilmente quelli più famosi sono gli *alberi binari di ricerca*, caratterizzati da qualche ulteriore proprietà. Il loro nome deriva dal fatto che sono usati per la ricerca veloce di informazioni, ma come vedremo essi servono prima di tutto a memorizzare i dati, e solo di conseguenza a ricercarli. I dati in essi contenuti possono essere modificati, se cambiano, o addirittura cancellati, se non servono più. Infine, essi definiscono implicitamente l'ordine delle informazioni, caratteristica che si ottiene come sottoprodotto della loro costruzione e senza costi aggiuntivi di elaborazione. Questo permette un facile ordinamento, a riprova di quella flessibilità che prima vantavamo.

Per capire come funziona un albero binario di ricerca, la cosa migliore è senz'altro quella di costruirne uno, specificando così l'algoritmo di inserimento dei nuovi nodi, cioè di costruzione. Una volta costruito, sarà allora facile spiegare gli algoritmi di ricerca di una specifica informazione e di ordinamento. Si consideri allora un insieme di numeri o di nomi. Ad esempio, prendiamo le parole iniziali del precedente paragrafo:

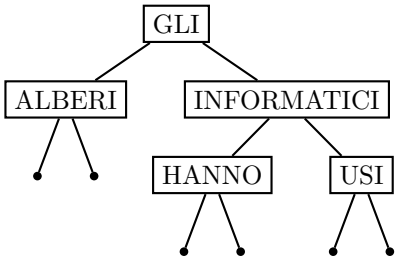
GLI ALBERI INFORMATICI HANNO USI
MOLTO AMPI E DIFFERENZIATI,
MA PROBABILMENTE QUELLI PIU' ...

ed, estraendone man mano le singole parole, costruiamo con esse il nostro albero binario di ricerca. La prima parola è GLI, ed essa va a costituire l'informazione relativa alla radice. Abbiamo così un alberello iniziale, di un solo nodo. Convenzionalmente, i rami fasulli che se ne dipartono si indicano con due segmenti tronchi:


Consideriamo ora la seconda parola ALBERI e vediamo come si inserisce nell'albero. Questo chiarirà la regola che determina la qualifica di "ricerca" a questi alberi binari. La parola ALBERI si confronta con quella contenuta nella radice, cioè GLI. Se, come in questo caso, è minore, si procede lungo il ramo di sinistra; se fosse stata maggiore, avremmo seguito il puntatore di destra. Comunque, il ramo di sinistra non ci porta da alcuna parte, e questo ci dice che ALBERI è una parola nuova da aggiungere alla struttura. Si crea allora un nuovo nodo come termine del ramo di sinistra di GLI, e vi si inserisce la parola ALBERI come informazione. Così il nostro albero cresce e si sviluppa.




In modo analogo, la parola INFORMATICI, confrontata con GLI della radice, ci porta verso destra. Anche questo ramo è vuoto e quindi si inserisce un nuovo nodo che contiene INFORMATICI come informazione. Più interessante è la parola successiva HANNO. Confrontata con la radice ci porta a destra e quindi va messa a confronto con INFORMATICI. Essendo minore ci si sposta lungo il ramo di sinistra. Questo è vuoto, e vi viene perciò aggiunto un nodo con l'informazione HANNO. Proseguendo con USI e partendo sempre dalla radice, si ha l'inserimento di un nuovo nodo a destra di INFORMATICI.




Seguiamo ora nella Figura 5.2 l'inserimento di MOLTO. La parola è maggiore sia di GLI, sia di INFORMATICI. Arrivati ad USI, troviamo però che è minore e il suo posto è perciò alla sinistra di questo nodo. L'inserimento delle altre parole segue le medesime regole e questo fa crescere l'albero, che si sviluppa man mano occupando zone sempre nuove della memoria.

Il vantaggio degli alberi, come quello delle liste, è di crescere senza restrizioni relative alla posizione

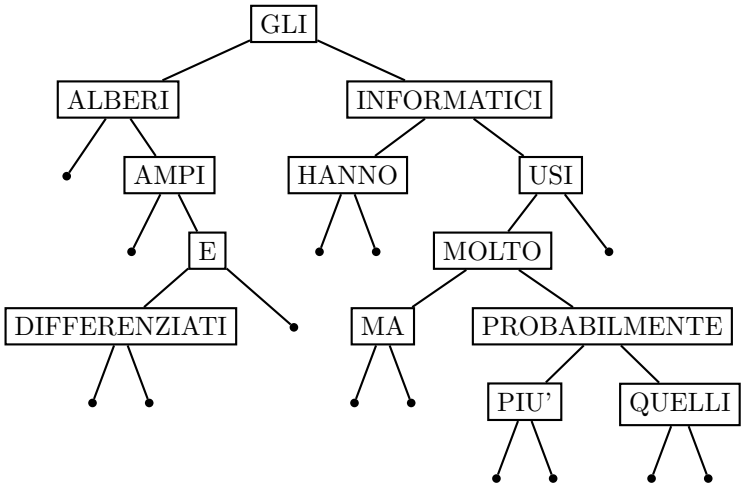



Figura 5.2: Un albero binario di ricerca

di memoria che può utilizzare. In un vettore, invece, se abbiamo deciso di inserire cento elementi, non ne possiamo inserire 101, poiché potremmo andare a toccare posizioni della memoria riservate ad altri dati. La gestione della memoria libera, richiesta da liste ed alberi, è abbastanza complessa, ma esistono appositi programmi che sollevano l'utente da tale incombenza. Una volta costruito, l'albero rivela le sue molteplici caratteristiche, delle quali ci limitiamo a discutere e commentare la ricerca e l'ordinamento.

La ricerca di un elemento segue, pari pari, lo schema descritto per l'inserimento. Basta pensare un attimo, e si capisce che come l'abbiamo aggiunto, così possiamo recuperare un qualsiasi elemento. Prendiamo il caso di PIU'. Lo confrontiamo con la radice GLI ed, essendo più grande, procediamo verso destra. Il secondo confronto è con INFORMATICI, di cui PIU' è maggiore. Andando ancora verso destra si trova USI e quindi, finalmente, si procede verso sinistra. Qui c'è MOLTO e PIU' è di nuovo maggiore. Procedendo verso destra si trova PROBABILMENTE e quindi si va a sinistra. L'ultimo confronto è con PIU' e questo ci dice che la parola è presente nel nostro testo. Basta provare con FRETTA, arrivare al nodo con E, e accorgersi che la parola, invece, nel nostro albero non è presente.

Prima di passare ad esporre il metodo per l'ordinamento legato agli alberi binari di ricerca, vediamo un po' perché questa struttura è considerata tanto interessante. Cominciamo col ricordare che si dice *livello* di un nodo la sua distanza dalla radice, più 1. In questa maniera, il livello della radice GLI è 1, il livello di HANNO è 3 e quello di PIU' è 6. E' ovvio che quando cerchiamo un termine che si trova nell'albero, eseguiamo tanti confronti quanto è il livello del nodo in cui la parola si trova. Quindi, il livello è una misura di quanto velocemente possa essere eseguita la ricerca. Un albero sarebbe perfetto se fosse così costruito: un nodo, cioè la radice, al livello 1; due nodi al livello 2, quattro nodi al livello 3 e, in generale, $2^{k-1}$ nodi al livello $k$. Purtroppo, come si vede dall'esempio, questa è solo una situazione ideale.

Immaginiamo comunque per un momento di trovarci di fronte all'albero perfetto. Poiché:

$$1 + 2 + ... + 2^{k-1} = 2^k - 1,$$

se l'albero si estende per $k$ livelli, esso contiene questo numero di nodi, o parole. Se indichiamo con $n$ il numero dei nodi dell'albero, abbiamo $2^k - 1 = n$, ovvero $k = \log_2(n + 1)$. Poiché, come si è osservato, $k$ è il numero dei livelli dell'albero, esso rappresenta anche il massimo numero di confronti necessari a trovare una qualsiasi parola presente nell'albero. Come ricordiamo, questo è anche il massimo numero di confronti che si hanno nella ricerca binaria. Pertanto, la ricerca binaria così efficiente, è equivalente alla ricerca in un albero binario, purché questo sia perfetto, o, come si dice, *perfettamente bilanciato*.

In generale, purtroppo, come nel nostro precedente esempio, quando si costruisce praticamente un albero binario, esso non viene fuori così ben bilanciato. Senza voler definire esattamente questa parola, diciamo che un albero è *bilanciato* (senza specificare "perfettamente", che ha il significato preciso che si è visto) quando non *pende* troppo né a sinistra, né a destra. Si può dimostrare, anche se noi non lo facciamo, che inserendo in un albero una sequenza di parole prese a caso (come le parole di un testo, lette nell'ordine della frase) il risultato è proprio un albero (abbastanza) bilanciato. Gli alberi bilanciati sono importanti perché il loro livello medio, cioè il numero medio di confronti che devono essere fatti per recuperare una parola qualsiasi, è vicino a $\log_2(n)$, anzi è circa $1.386 \log_2(n)$.

In pratica, questa formula vuol dire che la ricerca di un elemento all'interno di un albero binario è molto efficiente. Ad esempio, se $n$ è un milione, la media dei confronti è all'incirca 28, non troppo discosta dai 20 confronti della ricerca binaria, e ben lontana dai 500 mila confronti della ricerca sequenziale. Un qualsiasi elemento, comunque, si trova in una frazione di secondo, e questo mette in rilievo come la struttura ad albero sia paragonabile alla ricerca binaria, pur non avendo richiesto, come quella, l'ordinamento dei dati, operazione che, come abbiamo visto, richiede un bel po' di lavoro per essere portata a termine. In effetti, come ora vedremo, è la costruzione stessa dell'albero binario che crea una sorta di ordinamento delle parole inserite nella struttura.

L'osservazione di base, relativa alla connessione fra alberi binari e ordinamento, è la seguente: quando si effettua l'inserimento, spostiamo tale inserimento verso sinistra, se l'elemento è minore della parola nel nodo, lo spostiamo verso destra, se è maggiore. Questo implica che, rispetto ad ogni nodo, gli elementi che inseriamo a sinistra sono tutti più piccoli della parola nel nodo, quelli che inseriamo a destra sono più grandi. Quindi, in un certo senso, gli elementi di un albero sono già implicitamente ordinati. Basta osservare nel nostro albero la situazione relativa alla radice che separa a sinistra le parole che precedono GLI e a destra quelle che lo seguono. E questo vale, come si vede subito, per tutti i nodi dell'albero, e su questo ora ci baseremo.

Se fissiamo ancora la nostra attenzione sulla radice, e riprendiamo la proprietà appena ricordata, possiamo dire che si ottiene un ordinamento completo delle parole del nostro albero procedendo così:

1. si ordinano gli elementi a sinistra della radice;
2. si considera la parola contenuta nella radice;
3. si ordinano gli elementi a destra della radice.

L'insieme dei nodi che si trovano a sinistra di un nodo assegnato si dicono il suo *sottoalbero di sinistra*; naturalmente, quelli che si trovano a destra costituiscono il suo *sottoalbero di destra*. La dizione "sottoalbero" è appropriata, nel senso che si tratta proprio di un albero binario di ricerca, ed è solo una parte dell'albero completo. L'ordinamento di questo, allora, consiste nell'ordinare il sottoalbero di sinistra, inserire la parola della radice e ordinare il sottoalbero di destra.

Messe le cose in questa maniera, la procedura di ordinamento si presenta come una procedura ricorsiva, il concetto che abbiamo introdotto a suo tempo. Un processo ricorsivo riconduce l'elaborazione a entità sempre più piccole, fino ad arrivare ad entità banali per le quali si è definito esplicitamente il comportamento. Avevamo visto il calcolo per il fattoriale, che usa la moltiplicazione fino a ridursi a calcolare il fattoriale di 0, definito come 1. A questo punto l'elaborazione si completa con l'esecuzione delle varie moltiplicazioni. Qui, analogamente, partendo dall'albero completo, si passa all'ordinamento dei due sottoalberi, di sinistra e di destra, fino a ridurci alle foglie che, non avendo alcuno dei due sottoalberi, producono semplicemente la parola in esse contenute:

Possiamo seguire cosa succede per il sottoalbero di sinistra del nostro albero di esempio. Il nodo ALBERI è da vedere come radice del sottoalbero. Ordinare il suo sottoalbero di sinistra non produce nulla, poiché non ha elementi. Si scrive allora la parola ALBERI che risulta la prima dell'ordinamento, e si passa al suo sottoalbero di destra. La radice AMPI ci rimanda al sottoalbero di sinistra, che però è vuoto. Si scrive allora la radice e si procede col sottoalbero di destra; la radice E ci rimanda al sottoalbero di sinistra, contenente un solo nodo. Questo è il caso che trattiamo direttamente, nel senso che in questa situazione si scrive semplicemente la parola contenuta nel nodo. Tocca poi alla radice corrente E, che rimanda successivamente al suo sottoalbero di destra. Questo è vuoto e termina così il sottoalbero di sinistra di GLI, che entra ora nell'elenco:

ALBERI
AMPI
DIFFERENTI
E
GLI

Si è ottenuto così l'ordinamento dei primi termini dell'elenco. In modo analogo, si ordinano gli elementi del sottoalbero di destra, e questo completa l'ordinamento. La procedura ricorsiva si riduce a tre frasi, o istruzioni, in un linguaggio di programmazione di tipo algebrico, tipo PASCAL o C:

```
PROCEDURE ORDINA(R);
IF R.SINISTRA ≠ VUOTO
      THEN ORDINA(R.SINISTRA);
SCRIVI(R.INFO);
IF R.DESTRA ≠ VUOTO
      THEN ORDINA(R.DESTRA);
RETURN
```

Questo può essere considerato un vero e proprio programma. Il parametro R indica la radice del sottoalbero che si sta considerando; il programma stesso verrà richiamato ponendo in R la radice dell'intero albero, così da dare inizio all'ordinamento. Si suppone che R contenga tre informazioni: R.INFO, il valore del termine contenuto nel nodo (nel nostro caso, una parola), R.SINISTRA, il puntatore verso il sottoalbero di sinistra, e R.DESTRA, il puntatore verso il

sottoalbero di destra. Il valore VUOTO indica che il corrispondente sottoalbero non esiste. Il lettore vorrà scusare questo ingresso, un po' repentino, negli aspetti più tecnici della programmazione.

In JAVA la cosa è un po' più complicata. Si può osservare che le tre istruzioni sono eseguite per ogni nodo dell'albero binario, eventualmente saltando qualche pezzo se un sottoalbero non esiste. Pertanto, il numero di operazioni cresce in modo lineare rispetto al numero di elementi, o di nodi, presenti nell'albero. In altre parole, le operazioni raddoppiano se raddoppia il numero dei nodi, e quindi delle parole. Questo è il meglio che si possa fare per ordinare gli elementi di un elenco.

In realtà, l'albero sfrutta il fatto che l'ordinamento è stato eseguito già in fase di inserimento. La costruzione dell'albero è, in effetti, l'operazione più costosa, ma se l'albero è bilanciato è circa $1.386n\log_2(n)$, cioè poco più che lineare. Questo dimostra, ancora una volta, quanto gli alberi siano efficienti come struttura. Anche la cancellazione di un elemento si può fare in tempo logaritmico come la ricerca, ma ci guardiamo bene dal presentare qui l'algoritmo corrispondente, anche se niente affatto complicato. Tutti questi fatti fanno ritenere gli alberi una delle strutture più utili e versatili dell'Informatica, anche se esistono tecniche che, caso per caso o funzionalità per funzionalità, possono essere un po' migliori.

Gli alberi vanno bene fino a che sono bilanciati. Essi possono degenerare in una semplice lista se, ad esempio, inseriamo gli elementi nel loro ordine naturale. In tal caso, la capacità di dividere il problema in due, tipico dell'albero, non può essere più sfruttata e ciò rende quasi inservibili gli alberi sbilanciati. Per questo sono state inventate procedure che permettono agli alberi di crescere in modo bilanciato in tutte le circostanze. Ma qui le cose si complicano un po' e noi siamo costretti a lasciar perdere, nella speranza di aver dato un'idea abbastanza chiara di cosa sono gli alberi binari di ricerca e quali sono alcune delle loro molteplici applicazioni nell'Informatica.

## 5.3 Complessità e computabilità

Gli algoritmi, che abbiamo presentato sui vettori, sulle liste e sugli alberi, sono considerati abbastanza semplici, almeno dal punto di vista dell'elaboratore. Vediamo di capir bene cosa si vuole affermare con una frase del genere. Chi viene a conoscere per la prima volta l'algoritmo di ordinamento legato agli alberi, può trovare più o meno difficoltà a capirlo, cioè a ripetere correttamente la procedura su altri alberi di ricerca. Qualunque sia stato il grado di comprensione richiesto, una volta scritto il programma relativo, l'elaboratore impiegherà sempre un numero di operazioni proporzionale ad $n$, se $n$ sono i nodi dell'albero. Pertanto, se il numero dei nodi raddoppia o triplica, così raddoppierà o triplicherà il tempo richiesto dall'ordinamento.

I moderni elaboratori elettronici si sono acquistati la fama di macchine velocissime, che riescono ad affrontare e risolvere tutti i problemi che la realtà ci propone. Non solo essi ci facilitano il lavoro, ma la nostra fiducia è così grande che ci immaginiamo cose grandiose: con la buona volontà e con qualche sforzo potremo presto liberarci di tanti lavori inutili e, soprattutto, potremo trovare una soluzione a tanti problemi che ci affliggono. Questo, purtroppo, è ben lontano dalla verità per due motivi molto profondi:

1. vi sono problemi dei quali conosciamo benissimo le soluzioni possibili, ma che l'elaboratore affronta solo con estrema difficoltà;

2. esistono problemi che proprio non sono risolubili in termini algoritmici, e quindi non sono risolubili né con gli elaboratori di oggi, né con quelli che saranno costruiti nel futuro.

Vediamo di affrontare il primo aspetto, quello dei problemi risolubili, ma difficili da trattare da parte dell'elaboratore. Abbiamo detto a suo tempo che un algoritmo è un metodo per la risoluzione di un problema, metodo che sia formulato in modo non ambiguo e che ci porti alla soluzione in un tempo finito. La condizione sulla non ambiguità delle regole dell'algoritmo ha il significato chiaro di rendere la sua esecuzione alla portata di una macchina che, ad ogni istante, deve sapere che cosa fare. Nella nostra concezione, ogni algoritmo è meccanizzabile, almeno in principio. La condizione sul tempo finito è altrettanto ragionevole, poiché ben difficilmente potremmo essere soddisfatti da una risposta che mai non verrà.

Gli algoritmi che abbiamo incontrato fin qui (eccetto uno) sono di agevole soluzione da parte dell'elaboratore, una volta che gli abbiamo fornito l'algoritmo opportuno sotto forma di programma. Ricerca e ordinamento sono algoritmi visti recentemente, ma gli algoritmi di compattamento e quelli per l'esecuzione delle funzioni elementari sono esempi altrettanto validi, anche se li abbiamo trattati all'inizio di queste note. L'unica eccezione è costituita dal metodo di scomposizione di un numero in fattori primi: abbiamo osservato che esso è sempre risolubile, ma abbiamo anche avvertito che il tempo di risoluzione, ancorché finito, può essere dell'ordine di centinaia di anni per numeri grandi (diciamo, duecento cifre) ma non certo astronomici.

Occorre intenderci bene quando diciamo che un problema è risolubile. Questa affermazione può avere

un significato positivo e uno negativo. Il significato è positivo quando intendiamo dire che si conosce un metodo, cioè un algoritmo, per trovarne la soluzione (quando esiste), e per stabilire che non c'è soluzione, quando non esiste. Si pensi al vecchio esempio della risoluzione delle equazioni di secondo grado. Il significato è invece negativo quando intendiamo dire che conosciamo una dimostrazione sulla non risolubilità del problema. Ad esempio, dai tempi di Pitagora sappiamo che non esiste alcuna frazione che sia uguale alla radice quadrata di 2. Nel 1800 si dimostrò l'impossibilità di tanti problemi, fra i quali è famoso quello della quadratura del cerchio.

Un problema che sia sempre risolubile in questo senso esteso si dice tecnicamente *decidibile*, e la parola sta a indicare il fatto che, in ogni caso, sappiamo decidere se il problema ha soluzione (e, quindi, quale essa sia) o non ha soluzione. I problemi decidibili sono quelli che ci aspettiamo di incontrare, anche se, purtroppo, esistono problemi indecidibili, di cui parleremo più avanti. Fra i problemi decidibili, alcuni sono più facili da risolvere, altri più difficili. Con questo, vogliamo ribadire, non ci si riferisce alla difficoltà umana di trovare un algoritmo, ma piuttosto alla difficoltà che può avere un elaboratore ad arrivare alla soluzione, una volta che gli si sia fornito l'algoritmo.

Come abbiamo visto, per trovare un elemento all'interno di un albero binario l'elaboratore impiega una frazione di secondo. Per costruire un albero, l'elaboratore impiega un po' più di tempo, indicato da quella che abbiamo detto la sua complessità, cioè $1.386n\log_2(n)$. Raddoppiando il numero di elementi, la costruzione richiede un po' più del doppio di tempo, ma questo non è particolarmente grave. Ad esempio, l'esecuzione dell'ordinamento a bolle richiede circa $n^2/2$ operazioni, e quindi se raddoppia il numero di elementi, il tempo cresce di 4 volte.

Ma le situazioni veramente complicate vengono fuori da problemi che, spesse volte, hanno una forma molto banale. Ma questa banalità si dimostra essere un'illusione e tali problemi impegnano così tanto l'elaboratore che sono stati detti *intrattabili*. L'esempio forse più famoso è il così detto *problema del commesso viaggiatore*. Questo problema si presenta in molte occasioni e sotto formulazioni diverse. Ad esempio, ne esiste una versione collegata ai problemi di connessione in rete. Tuttavia, la formulazione meno tecnica, e che dà il nome al problema, è la seguente.

Nella citta di X vive un commesso viaggiatore che all'inizio di ogni settimana (o di ogni mese) è abituato a mettere a punto il piano delle proprie visite nel periodo in questione. Egli sa quali città deve visitare per il suo lavoro, sa le spese che deve sostenere per gli spostamenti, e desidera trovare il percorso ottimale, cioè il percorso che, partendo da X, lo riporta ad X dopo aver visitato tutte le città previste, ma con la minor spesa possibile. Questo, naturalmente, gli farà guadagnare di più.

Se le città da visitare sono solo due, A e B, i percorsi possibili sono XABX e XBAX, e quindi il problema ha una soluzione banale, che consiste nel calcolare il costo dei due percorsi e scegliere quello che costa meno. Se le città da visitare sono 3, le cose cominciano a complicarsi e i percorsi possibili sono sei:

XABCX, XACBX, XBACX,
XBCAX, XCABX, XCBAX.

Essi corrispondono alle possibili permutazioni delle tre città da visitare A, B, C. Il termine "permutazione" indica ogni possibile ordinamento degli oggetti considerati. Se le città sono 4, le permutazioni possibili sono 24, ma con un po' di buona volontà il nostro commesso viaggiatore può ancora trovare la soluzione ottima. Se poi sa un po' programmare, è abbastanza semplice scrivere un programma per trovare tutte le permutazioni e calcolarne il costo.

Vediamo di capire quante sono le permutazioni da considerare per il nostro problema, se le città da visitare sono $n$. Pensiamo di dare un ordine a queste città. In prima posizione possiamo inserire una qualsiasi delle $n$ città, cioè ci sono $n$ possibilità. In seconda posizione possiamo inserire una qualsiasi cita, eccetto quella messa in prima posizione. Quindi, per ognuna delle $n$ città in prima posizione, ve ne sono $n-1$ in seconda per un totale di $n(n-1)$ possibilità. In terza posizione possiamo ancora inserire tutte le città, eccetto quelle che abbiamo ormai messo in prima e seconda posizione. Le possibilità sono allora $n(n-1)(n-2)$. Continuando a ragionare così, si va avanti fino a che non rimaniamo con l'ultima città, e allora la inseriamo nella permutazione, e abbiamo finito.

Questo ragionamento ci dice che le possibili permutazioni sono:

$$n(n-1)(n-2)...2\times 1 = n!$$

una nostra vecchia conoscenza, il fattoriale di $n$. I valori sono in accordo con il 2, il 6 e il 24 che abbiamo trovato per $n = 2, 3, 4$. Purtroppo il fattoriale è una quantità che cresce molto rapidamente. Ad esempio, $10! = 3{\cdot}628{\cdot}800$, e quanto a 20! siamo sull'ordine di $10^{18}$. Se il nostro commesso viaggiatore dovesse visitare 20 città, il suo elaboratore dovrebbe studiare questo bel po' di permutazioni. Trattandone una ogni milionesimo di secondo, impiegherebbe $10^{12}$ secondi, e cioè circa 35 mila anni, e questo non è particolarmente esaltante. Ci fa però capire qual è il problema a cui siamo di fronte.

Si potrebbe pensare che, cercando cercando, si riusciranno a trovare algoritmi che non devono passare

in rassegna tutte le permutazioni possibili. Purtroppo, fino ad oggi, nulla si è potuto fare in questa direzione, anche se di studi ne sono stati fatti a iosa. Anzi, ci sono motivi per pensare che il metodo ingenuo sia l'unico possibile.

Verso il 1730, il matematico inglese Stirling trovò una formula approssimata per il fattoriale:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n.$$

Essa mostra che il numero di permutazioni cresce in modo più che esponenziale: infatti, troviamo la $n$ ad esponente. Un problema si dice *intrattabile* se la sua complessità è almeno esponenziale. Si dice invece *trattabile* se la sua complessità è al più polinomiale, cioè $n$ è alla base e non all'esponente. Tali sono la ricerca sequenziale e binaria, l'ordinamento a bolle e quello relativo agli alberi.

Ci sono motivi molto seri, cioè molto matematici, per fissare il limite dei problemi trattabili alla complessità esponenziale, ma noi ci accontenteremo di dare una giustificazione intuitiva. Se il programma del commesso viaggiatore riuscisse a trattare un milione di permutazioni al secondo (il che è già poco probabile) in un'ora di calcolo potrebbe risolvere il problema con 13 città. Non sono molte, ma per il momento accontentiamoci. Nella stessa ora, possiamo supporre che il programma della ricerca sequenziale riesca a fare 3600 ricerche nell'elenco dell'anagrafe di una città con un milione di abitanti. E ancora, il programma di ordinamento a bolle riesca a ordinare 3600 elenchi di 1000 elementi ciascuno.

Il grande scienziato Valentino Coconti inventa un nuovo elaboratore, mille volte più veloce di tutti gli elaboratori esistenti. Subito, la nuova perla informatica viene impegnata sui tre problemi precedenti e si valuta cosa riesca a fare in un'ora di elaborazione. E' chiaro che la ricerca sequenziale, di complessità lineare, da 3600 ricerche passerà a farne mille di più, cioè tre milioni e 600 mila, con un bel miglioramento. Per l'ordinamento a bolle, di complessità quadratica, si ha un incremento da 3600 elementi a oltre 110 mila, e anche questo è un miglioramento degno di nota. Valentino Coconti si frega le mani soddisfatto, osservando questo successo, e si prepara a contare il contante che gli verrà dalla sua invenzione.

Quando però passa al problema del commesso viaggiatore, l'entusiasmo di Valentino si smorza alquanto. Per passare da 13 a 14 città, il tempo aumenta di 14 volte, ovvero, con la nuova macchina, si sfruttano 14 dei 1000 incrementi a disposizione. Per passare da 14 a 15 si sfruttano $14 \times 15 = 210$ dei mille incrementi, e per passare da 15 a 16 ne occorrerebbero $210 \times 16 = 3360$, più di tre volte le possibilità del nuovo elaboratore. In altre parole, per arrivare a 16 città, l'elaboratore dovrà lavorare per 3 ore e 20 minuti, e questo non è affatto entusiasmante. Quindi, il problema del commesso viaggiatore rimane al di là delle capacità della nuova invenzione, per quanto eccezionale essa possa apparire. Questo è il guaio di tutti i problemi di complessità esponenziale.

Quando un algoritmo di risoluzione di un problema ha complessità esponenziale, l'aumento di poche unità nella dimensione del problema comporta una crescita enorme nel numero di operazioni da effettuare e, quindi, a un'impennata nel tempo di elaborazione. E' questo anche il caso della scomposizione di un numero in fattori primi, un altro problema intrattabile. Naturalmente, ce ne sono tanti altri, ma come abbiamo visto la difficoltà di scomporre un numero in fattori è sfruttata nei metodi di ciframento a chiave pubblica. La conoscenza che tutti hanno del numero $N = xy$ usato come base del metodo, dovrebbe essere accompagnato dalla conoscenza di uno almeno dei suoi fattori; ma questo non è banale, poiché è troppo difficile trovare uno qualsiasi dei fattori $x$ o $y$.

Il fatto che ci siano problemi intrattabili, cioè veramente difficili da affrontare con l'uso dell'elaboratore, non esaurisce la nostra storia. Potremmo infatti pensare (cosa che avviene spesso) che sì, è vero, esistono problemi duri anche per le nostre macchine, ma comunque la nostra intelligenza sia capace di trovare un metodo di risoluzione a qualunque problema le vogliamo porre. Talvolta la soluzione sarà valida solo in linea di principio, ma una soluzione si troverà. In effetti, questo è l'atteggiamento ottimista, sviluppato dalla scienza moderna, che ci pervade: noi siamo convinti che col tempo i nostri sforzi intellettuali ci permetteranno di affrontare e risolvere ogni tipo di problema. Incontreremo anche certi problemi che, con le nostre misere forze, saranno risolvibili solo allo stato embrionale; ma questo dipende dal nostro essere limitati nello spazio e nel tempo, non dai problemi o dalla nostra intelligenza.

In realtà, la situazione è piuttosto più complicata. A suo tempo abbiamo definito un algoritmo come un qualsiasi metodo che risolva un determinato problema in modo non ambiguo e in tempo finito. Purtroppo, non di tutti i problemi conosciamo algoritmi di risoluzione. Per fare un paio di esempi tratti dalla Matematica, ricordiamo:

1. si può osservare che esistono coppie di *numeri primi gemelli*, cioè che differiscono di due unità: 5 e 7, 29 e 31, e così via. E' stata fatta l'ipotesi che i primi gemelli siano infiniti, ma nessuno è mai riuscito a dimostrare;

2. verso la metà del 1700 il matematico russo Goldbach notò che i numeri pari si ottengono come somma di due numeri primi: $100 = 87 + 13$, $32 = 25 + 7$, e così via. Scrisse ad Eulero facendo

l'ipotesi che questa proprietà valesse per tutti i numeri pari, cosa che lui non sapeva dimostrare. Neanche Eulero riuscì a trovare una dimostrazione, come non c'è riuscito nessuno dopo di lui.

Un modo ingenuo di affrontare questi problemi, in particolare l'ipotesi sui numeri gemelli, potrebbe essere quello di dire: Proviamo! Facciamo, cioè, un elenco di tutti i possibili numeri gemelli a cominciare da 3 e 5. Se i numeri gemelli sono infiniti, il nostro elenco cresce e cresce e pian piano produrrà tutte le coppie possibili. Ma se le coppie sono in numero finito, diciamo $n$, arrivati a cercare la $(n+1)$-esima coppia la ricerca va avanti, va avanti, ma non si troveranno mai i due gemelli che non ci sono. Abbiamo instaurato, come si dice, una *procedura infinita*.

Non sappiamo se questa eventualità si presenta davvero, ma dobbiamo, per forza di cose, tener di conto che i metodi di risoluzione dei problemi possono incorrere in procedure infinite, e allora non li possiamo più classificare come algoritmi. Si potrebbe formalizzare questa osservazione passando, ad esempio, dagli algoritmi ai programmi. E' facile scrivere un programma che "cicla" oppure uno che cicla per certi valori e termina per certi altri:

K=5; WHILE K≠ 0 DO K=K+1 OD;

Ma lasciamo perdere e accontentiamoci di un'esposizione non formale.

La classe dei metodi di risoluzione dei problemi è allora più vasta di quella degli algoritmi, che devono sottostare alla condizione di arrivare sempre alla fine della loro elaborazione. Se è chiaro questo concetto, dovrebbe essere altrettanto chiaro il seguente. Cominciamo col chiamare *processo* un qualsiasi metodo di risoluzione di un problema; osserviamo quindi che ogni processo può essere descritto con una certa sequenza di frasi, in una lingua che possiamo fissare una volta per tutte, diciamo l'italiano. Questa descrizione deve essere finita, altrimenti non sarebbe una descrizione, anche se la sua esecuzione potrebbe richiedere un tempo infinito. Possiamo allora ordinare in un elenco tutti i processi $P_n$: cominciamo da quelli che hanno la descrizione più corta e procediamo coi più lunghi. A parità di lunghezza, mettiamo prima quelli alfabeticamente inferiori, come in un dizionario.

Poiché le descrizioni di data lunghezza sono in numero finito, in linea di principio il nostro elenco non presenta difficoltà di costruzione. Poiché stiamo studiando un problema teorico, non ci dobbiamo preoccupare se tale costruzione si protrarrà per anni e secoli e se contiene processi ripetuti o privi di senso. Questo elenco è ciò che ci interessa, ma facciamo ora un'osservazione. Sarebbe interessante poter dare un analogo elenco per gli algoritmi, cioè per quei processi che ci danno sempre e comunque un risultato. Potremmo così dire qual è il millesimo algoritmo e qual è il milionesimo.

Ora la cosa si fa più difficile, perché dovremmo scegliere le descrizioni di quei processi che terminano sempre, e questo non è banale poiché dovremmo verificare una cosa del genere e, come per l'ipotesi di Goldbach potremmo incappare in qualche computazione infinita, che ci lascerebbe a metà dell'opera. Supponiamo comunque di poter costruire un elenco del genere, e siano $A_1, A_n2, \ldots$ i vari algoritmi.

Facciamo ora un lavoro ulteriore. Ordiniamo i possibili ingressi ai vari algoritmi: numeri, parole, immagini codificate, eccetera. Procedendo ancora per lunghezza e, a parità di lunghezza, in ordine alfabetico, si arriverà a un elenco completo. Siano $I_1, I_2, \ldots$ questi argomenti. Possiamo allora costruire una matrice infinita che, in corrispondenza ad ogni algoritmo $A_n$ e ad ogni possibile argomento $I_m$, ci dà il risultato $U_{n,m}$. Questo è possibile, essendo le computazioni degli algoritmi tutte finite.

| | $I_1$ | $I_2$ | $I_3$ | $I_4$ | $\cdots$ |
|---|---|---|---|---|---|
| $A_1$ | $U_{1,1}$ | $U_{1,2}$ | $U_{1,3}$ | $U_{1,4}$ | $\cdots$ |
| $A_2$ | $U_{2,1}$ | $U_{2,2}$ | $U_{2,3}$ | $U_{2,4}$ | $\cdots$ |
| $A_3$ | $U_{3,1}$ | $U_{3,2}$ | $U_{3,3}$ | $U_{3,4}$ | $\cdots$ |
| $A_4$ | $U_{4,1}$ | $U_{4,2}$ | $U_{4,3}$ | $U_{4,4}$ | $\cdots$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\ddots$ |

Definiamo ora un algoritmo $A$ nel modo seguente: quando viene applicato al $k$-esimo ingresso produce un valore diverso da $A_k(I_k)$, cioè del $k$-esimo algoritmo applicato al $k$-esimo argomento. In altre parole, se $A_k(I_k) = U_{k,k}$, il valore di $A(I_k) = N_k \neq U_{k,k}$. Ad esempio, se $U_{k,k}$ è il numero $p$, allora poniamo $N_k = p + 1$; oppure, se $U_{k,k}$ è una frase $F$, poniamo $N_k$ =\$F\$. In questa maniera, il modo di comportarsi di $A$ è ben definito, non ambiguo, e produce sempre un risultato in un tempo finito: quello di andare a veder quanto vale $U_{k,k}$ oppure di trovare questo valore eseguendo $A_k(I_k)$. Quindi, $A$ è un algoritmo.

Poiché la nostra matrice elenca tutti gli algoritmi, fra di essi si deve trovare anche $A$. Tuttavia, $A$ non può essere $A_1$ poiché $A(I_1) = N_1$ che per definizione è diverso da $A_1(I_1)$. $A$ non può essere $A_2$ perché $A(I_2) = N_2$, per definizione diverso da $A_2(I_2)$. E così $A$ non può essere nessuno degli $A_k$.

Ma questo è semplicemente assurdo, a meno che non sia falsa l'ipotesi da cui siamo partiti, e che cioè l'elenco degli algoritmi potesse essere costruito. A rigore, non importa nemmeno poter costruire la matrice di tutti i risultati. Basta, per ogni valore di $k$, sapere qual è l'algoritmo $A_k$, qual è il $k$-esimo ingresso $I_k$, e quindi calcolare $A_k(I_k)$, per prendere poi un valore diverso. Ma l'elenco degli ingressi si costrui-

sce come s'è visto e la computazione è sempre finita perché si stanno considerando algoritmi. Di conseguenza, è giocoforza ammettere che la cosa impossibile è proprio la costruzione dell'elenco di tutti gli algoritmi. Questa è la nostra conclusione e pertanto siamo di fronte a un problema ragionevolmente semplice e ben posto per il quale non può esistere alcun algoritmo di risoluzione.

La costruzione dell'elenco di tutti gli algoritmi non può essere effettuata in termini finiti, cioè con la specifica di un algoritmo, e questo, si badi bene, non vuol dire che il problema non abbia soluzione. Infatti, come abbiamo visto, potremmo dire che l'elenco degli algoritmo è un sottoelenco di quello di tutti i processi. E questo elenco, invece, ha una costruzione abbastanza banale.

Come conseguenza di questo risultato, possiamo dimostrare che esistono altri problemi che non sono risolubili algoritmicamente, pur essendo ben posti. Ad esempio, non esiste alcun algoritmo $T$ che, posto di fronte alla descrizione di un processo $P_n$ e a un suo possibile ingresso $I_k$, ci dica se la computazione sarà finita o infinita. Questo problema è noto come *problema della fermata* e vediamo come si dimostra.

Supponiamo che l'algoritmo $T$ esista e ci dica in tempo finito se il processo $P_n$ termina sull'ingresso $I_k$. Costruiamo allora la matrice di tutti i processi e di tutti gli argomenti, dove inseriamo un simbolo speciale, diciamo la gratella, ogni volta che $T$ ci dice che il processo innesca una computazione infinita. Nelle altre caselle inseriamo il risultato effettivo. In questa maniera, la matrice può essere costruita in maniera algoritmica, poiché così si costruiscono le sue componenti. Ma l'elenco dei $P_n$ diviene l'elenco di tutti e soli gli algoritmi $A_n$, poiché abbiamo trasformato ogni processo in algoritmo. Sappiamo però che l'elenco degli algoritmi non si può costruire ed è quindi assurdo che si possa pensare di costruirlo in questo modo indiretto. La conclusione è che $T$ non può esistere.

Il fatto stesso che esistano problemi indecidibili ci deve convincere che la nostra intuizione sbaglia quando ci porta a credere che tutti i problemi formulabili correttamente abbiano una soluzione in termini di algoritmi. In effetti, la costruzione dell'elenco di tutti i processi ci dice che possiamo definire per bene, in modo comprensibile e non ambiguo, questa classe molto ampia. La cosa sconvolgente (o, almeno, sconvolgente per i logici) è che restringendo la classe dei processi a quella degli algoritmi, si perde qualcosa che riteniamo importante, la capacità di riconoscere la caratteristica che più ci preme, e cioè la finitezza delle possibili elaborazioni.

# Indice analitico


accumulatore, 57
acknowledgement, 38
addizionatore, 55
addizione, 24
Al-Kuwarizmi (c. 770 – c. 840), 67
albero, 72
albero bilanciato, 74
albero binario, 72
albero binario di ricerca, 73
albero di Huffman, 46
albero perfettamente bilanciato, 74
ALGOL '60, 63
algorismo, 67
algoritmo, 67
algoritmo di base, 68
allineamento, 8
ALU, 57
ambito di validità, 63
American Standard Code for Information Interchange, 30
analisi degli algoritmi, 67
analisi della realtà, 68
apparecchiatura di ingresso, 9
apparecchiatura di ingresso e uscita, 9
apparecchiatura di uscita, 9
architettura a specchio, 8
architettura client/server, 8
Aristotele (384 a.C. – 322 a.C.), 51
arithmetic logic unit, 57
aritmetica indefinita, 26
ASCII, 30
Assembler, 60
at-sign, 30
Augusto (63 a.C. – 14 d.C.), 42
automa a stati finiti, 47

Babbage, Charles (1791 – 1871), 5, 6
back-up, 8, 38
base, 20
base di dati, 16
before image, 40
binary digit, 20
bit, 20
bit di parità, 38
Boole, George (1815 – 1864), 51
bottone, 12
branch, 57
bubble sort, 71
buffer, 9, 14
buffer-pool, 14, 16
Byron, Augusta Ada (1815 – 1852), 5, 54
byte, 24

C, 64
calcolatore palmare, 7
calcolatrice, 7
calcolo dei predicati, 51
calcolo delle proposizioni, 51
campionamento, 12, 35
campionatura, 35
campione, 35
cancelletto, 30
cancello, 7
carattere nazionale, 31
caricare, 13
casse, 12, 35
cathod ray tube, 10
CD, 17
CD riscrivibile, 36
Cesare, Caio Giulio (102 a.C. – 44 a.C., 42
Characteristica Universalis, 20
checksum, 38
chiave pubblica, 43
chiave segreta, 43
chiocciola, 30
Chomski, Noam (1928 – ), 63
ciano, 33
ciclo di vita del software, 68
ciclo operativo, 13
cifra binaria, 20
ciframento, 42
ciframento a chiave pubblica, 44
cifratura, 42
cifre significative, 27
cilindro, 16
circuito di collegamento, 7
circuito integrato, 6
cliccare, 12
client, 8
CMYK, 33
codice a barre, 12
codice autocorrettore, 39
collegamento remoto, 8
Colossus, 5

commissione internazionale del colore, 34
compact disk, 17, 35
complemento a 1, 26
complemento a 2, 26
complessità di un algoritmo, 70
complessità lineare, 71
complessità logaritmica, 71
complessità quadratica, 71
configurazione di byte, 25
congiunzione, 52
connettivi booleani, 52
connettivi logici, 52
connettivo di Sheffer, 54
contatore di istruzioni, 57
contatore di programma, 57
contraddizione, 53
convertitore analogico/digitale, 12, 35
copia di back-up, 8
copia di sicurezza, 38
coprocessore matematico, 29
costrutto FOR, 63
costrutto iterativo, 61
costrutto REPEAT, 63
costrutto WHILE, 63
CPU, 6
CRC, 38
crittografia, 42
crittologia, 42
cromatologia, 33
CRT, 10

dati alfabetici, 20
dati grafici, 20
dati numerici, 20
dati sonori, 20
dato, 19
DCT, 48
de Morgan, Augustus (1806 – 1871), 54
decidibile, 77
diario di bordo, 40
digital video disk, 17
digitalizzazione, 12
disco fisso, 14
disco magnetico, 13
disco ottico, 17, 35
disco rigido, 16
disco rimovibile, 14
disgiunzione, 52
disk pack, 14, 16
displacement, 14
divide et impera, 69
divisione, 27
dot, 11
dot per inch, 11
dpi, 11
driver, 9

drogare, 6
DVD, 17, 36

e commerciale, 30
EBCDIC, 30
echoing, 10
elaboratore da tavolo, 8
elaboratore di testi, 31
elaboratore personal, 8
elaboratori gemelli, 8
elemento grafico, 10
ereditare, 64
esclusione, 52
esponente, 29
Euclide ($\sim$ 300 a.C.), 37
Eulero, Leonhard (1707 – 1783), 78
EXCEL, 68

FAX, 32
Fibonacci, Leonardo (1180 – 1250), 67
figlio, 73
firma digitale, 44
flag, 16
floating point, 29
floppy disk, 16
foglia, 73
forma mantissa/esponente, 29
forma normalizzata, 29
formattazione del disco, 14
FORTRAN, 61
funzionalità, 68

gate, 7
gemelli, 78
giornale di bordo, 40
Goldbach, Christian (1690 – 1764), 78
grammatica formale, 63
grammatica libera dal contesto, 63
grammatica regolare, 63
grandezza analogica, 12
grandezza digitale, 12
graticola, 30

hard disk, 16
hardware, 7
Hawking, Sthephen William (1942 – ), 13
HTML, 34

immagine precedente, 40
indicatore, 16, 57
indirizzo, 14
indirizzo di un byte, 57
Informatica, 19
informazione, 19
informazione alfabetica, 30
ingresso, 9
insieme universale di connettivi, 54

integrità, 41
intelligenza artificiale, 65, 72
interrupt, 9
interruttore a chiave, 41
interruzione, 9, 15
intrattabile, 77, 78
istruzione, 57
istruzione condizionale, 63
istruzione di salto, 57
istruzione GOTO, 62

Java, 64
journal, 40
joy-stick, 12
JPEG, 48
jump, 57

Kepler, Johannes (1571 – 1630), 5

least recently used, 16
legge di Moore, 6
leggi di de Morgan, 54
Leibniz, Gottfried (1646 – 1716), 5, 20, 22, 51
lettore di codice a barre, 12
lettura, 9
lettura fisica, 16
lettura logica, 16
lettura virtuale, 16
LIFO, 64
linea delle porpore sature, 34
linea spettrale, 34
linguaggio algebrico, 61
linguaggio assemblatore, 60
linguaggio della prima generazione, 59
linguaggio della quarta generazione, 64
linguaggio della quinta generazione, 65
linguaggio della seconda generazione, 60
linguaggio della terza generazione, 61
linguaggio dichiarativo, 65, 72
linguaggio formale, 63
linguaggio macchina, 59
linguaggio simbolico, 60
Linux, 9
LISP, 65, 72
lista, 72
livello di un nodo, 74
log, 40
Logica formale, 51
LRU, 16

macchina alle differenze, 5
macchina analitica, 5
macchina di Turing, 5
macchina sincrona, 9
magenta, 33
main-frame, 6, 8
mantissa, 29
masterizzatore, 17, 36
memoria, 6
memoria ad accesso casuale, 13
memoria di massa, 13
memoria permanente, 13
memoria tampone, 9
memoria volatile, 13
metodo additivo, 33
metodo asimmetrico di cifratura, 43
metodo delle divisioni successive, 21
metodo simmetrico di cifratura, 43
metodo sottrattivo, 33
microfono, 12, 35
microprocessore, 6, 8
modalità asincrona, 8
modello matematico, 68
moltiplicazione, 26
monitor, 10
monitor piatti, 10
Moore, Gordon (1929 – ), 6
mouse, 11
MP3, 49
MPEG, 49

NAND, 54
nanotecnologia, 6
negazione, 51
Newton, Isaac (1642 – 1727), 33
nodo figlio, 73
nodo padre, 73
NOR, 54
notazione posizionale, 20
numeri interi, 27
numeri naturali, 27
numeri periodici, 28
numeri primi gemelli, 78
numero binario, 21
numero razionale, 29
numero reale, 29

OCR, 11
oggetto, 64
opposto, 25
optical character recognizer, 11
ordinamento a bolle, 71
orologio del sistema, 9
overflow, 58

pacchetto, 38
padre, 73
parametro, 62
parola segreta, 41
Pascal, 64
Pascal, Blaise (1623 – 1662), 5, 22
Pascaline, 5

password, 41
pennetta, 16
perforatore di schede, 12
periferica, 7, 9
periferica di lettura, 9
periferica di lettura e scrittura, 9
periferica di puntamento, 11
periferica di scrittura, 9
periferica per disabili, 13
personal computer, 8
Photoshop, 34
pila, 64
Pitagora (∼ 500 a.C.), 37, 77
pixel, 10, 33
Poe, Edgar Allan (1809 – 1849), 42
porta, 7
portatile, 8
precedenza, 61
precisione, 29
prefisso, 46
prima generazione, 59
privatezza, 41
problema decidibile, 77
problema del commesso viaggiatore, 77
problema dell'ordinamento, 68
problema della fermata, 80
problema della ricerca, 68
problema intrattabile, 78
problema trattabile, 78
problemi intrattabili, 77
procedura, 68
procedura infinita, 79
processo, 79
processore, 8
prodotto logico, 52
progettazione, 68
progettazione fisica, 68
progettazione logico/concettuale, 68
programma, 6
programmazione ad oggetti, 64
programmazione dichiarativa, 65
programmazione imperativa, 65
proposizione, 51
proposizione composta, 51
proposizione elementare, 51
psicologia dei colori, 33
puntatore, 72
puntatore vuoto, 72

qwerty, 10
qzerty, 10

radice, 72
RAM, 13
random access memory, 13
rappresentazione arabica, 20
record fisico, 14
registro, 13
registro indice, 59
registro operativo, 57
relé, 54
rete neurale, 65
RGB, 33
riallineamento, 40
ricerca binaria, 69
ricerca sequenziale, 69
Richelieu (1585 – 1642), 43
riconoscitore ottico di caratteri, 11
ricorsione, 64
ricorsività, 62
Rivest ( – ), 43
RLE, 45
run length encoding, 45

salta e salva, 59
salto incondizionato, 59
salvare, 13
salvataggio, 38
scanner, 11
scheda, 12
Schickard, Wilhelm (1592 – 1635), 5
Scolastica, 51
scrittura, 9
scrittura fisica, 16
scrittura logica, 16
scrittura virtuale, 16
seeking, 14
semantica computazionale, 63
server, 8
settore, 14
settore usato meno recentemente, 16
Shannon, Claude (1916 – 2001), 5, 22, 51, 54, 56
Sheffer, Henry Maurice (1883 – 1964), 54
shift-key, 10
sillogismo, 51
sintassi del linguaggio, 62
sintetizzatore della voce, 13
sistema operativo, 9
sistema per la gestione delle basi di dati, 64
software, 7
somma logica, 52
sottoalbero di destra, 75
sottoalbero di sinistra, 75
sottoprogramma, 59
sottrazione, 24
spiazzamento, 14
stack, 64
stampante, 10
stampante a bolle, 11
stampante a getto d'inchiostro, 11
stampante ad aghi, 10
stampante laser, 11

stato, 47
stazione di lavoro, 8
Stirling, James (1692 – 1770), 78
struttura dati, 71
struttura dati astratta, 71
struttura dati fisica, 71
supporto, 14

tabella di verità, 51
tastiera, 9
tasto ALT, 10
tasto CTRL, 10
tasto delle maiuscole, 10
tasto INVIO, 9
tautologia, 53
tavola pitagorica, 22, 27
telescrivente, 30
tempo di accesso alla memoria principale, 13
tempo di rotazione, 15
tempo di seek, 15
tempo di trasferimento, 15
tempo medio di accesso al disco, 15
teoria dell'informazione, 46
teoria della computabilità, 5
testina di lettura e scrittura, 14
tipo di dati, 64
touch-pad, 12
touch-screen, 10
trabocco, 58
traccia, 14
transazione, 40
transistor, 6
trasformata discreta del coseno, 48
trasmissione dei dati, 38
trattabile, 78
triangolo $RGB$, 33
triangolo $XYZ$, 34
Turing, Alan (1912 – 1954), 5

UNICODE, 31
unità a disco magnetico, 14
unità centrale di elaborazione, 6
UNIX, 64
uscita, 9

valore di verità, 51
valvola, 6
verifica di ridondanza ciclica, 38
vettore, 71
video, 10
virgola mobile, 29
von Neumann, John (1903 – 1957), 6, 57

wafer, 6
Windows, 9
wireless, 42
Word, 31
workstation, 8

zero, 20
Zuse, Konrad ( – 1954), 51