I **Libri di base** vanno incontro al bisogno di conoscere e di partecipare alle scelte di vita, di studio e di lavoro nel mondo d'oggi. Presentano autori fra i piú esperti in ogni campo di attività e di interesse. Sono scritti e illustrati in modo semplice e chiaro perché tutti possano capire.
La collana è diretta da Tullio De Mauro

Carlo Batini (Pescara, 1949) insegna programmazione dei calcolatori elettronici nell'università di Roma. Oltre a numerose pubblicazioni specialistiche ha scritto testi didattici adottati nelle università.

Ldb ultimi pubblicati *(all'interno l'elenco completo)*

62. **Lo Stato di Israele** di Nicola Garribba
63. **Stalin** di Aldo Agosti
64. **La Terra e le sue risorse** di Edoardo Proverbio
65. **Guida alle tasse** di Leonello Raffaelli
66. **Le frontiere della genetica** di Marcello Buiatti
67. **L'acqua** di Marco Fontana
68. **Nascita dell'industria in Italia** di Roberto Romano
69. **Le basi dell'informatica** di Carlo Batini

Di prossima pubblicazione

La filologia di Lucia Cesarini Martinelli
L'abc degli scacchi di J. Averbach e M. Bejlin
Che cos'è un calcolatore di Giacomo Cioffi

L. 6.000 . . .
ISBN 88-359-2681-5

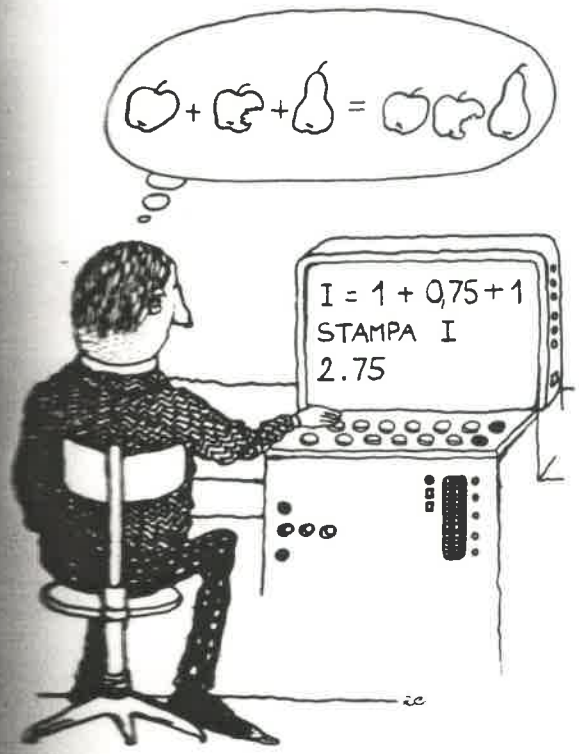
CL 63-2681-1

Libri di base

Le basi dell'informatica

Concetti e metodi per usare bene i calcolatori

di Carlo Batini



Editori Riuniti

Sezione 7. Il sapere: scienze e campi di ricerca
Volume a cura di Mario Carnevale

Carlo Batini

Le basi dell'informatica

Editori Riuniti

Le sezioni dei Libri di base

1. Il mondo: l'universo, gli ambienti, i paesi
2. La storia: epoche ed eventi
3. La donna, l'uomo: corpo, mente e funzioni
4. Arti e comunicazioni: linguaggi e tecniche espressive
5. Economia e lavoro: organizzazione e tecnologie
6. La società: istituzioni e forze politiche e sociali
7. Il sapere: scienze e campi di ricerca
8. Classici, testi, documenti, biografie

via Serchio, 9/11 - 00198 Roma
CL 63-2681-1
ISBN 88-359-2681-5

direttore responsabile Elisabetta Bonucci
cura redazionale di Emilia Passaponti
progetto grafico di Tito Scalbi
impaginazione di Luciano Vagaggini
disegni di Erminia Mercantini
illustrazione di copertina di Rosalba Catamo

INDICE

7 **I. Informatica e vita di ogni giorno**

Come si cerca un numero di telefono, 8. Come si consulta un orario, 10. Come si fa un orario, 15. Come si calcola (in alcuni casi) quanto dobbiamo aspettare un autobus, 18. Conclusioni, 20.

25 **II. Concetti introduttivi**

Algoritmi e programmi, 25. Risorse a disposizione di un calcolatore, 32. Linguaggi macchina e linguaggi programmativi, 33. I grafi di flusso come mezzo di descrizione di algoritmi, 36.

43 **III. La correttezza**

Introduzione, 43. Metodi basati sui dati di test, 49. Metodi formali, 57.

67 **IV. Calcolabilità e non calcolabilità**

Introduzione, 67. Le macchine di Turing, 70. Le macchine a programma, 76. Equivalenza di modelli di calcolo e tesi di Church, 82. Conclusioni, 86.

88 **V. L'efficienza**

Introduzione, 88. Misure di efficienza, 103. L'efficienza è un falso problema?, 109.

112 **VI. Il progetto di algoritmi e programmi**

Introduzione, 112. Uso di un'agenda e di un elenco telefonico, 113. Ricerca del cammino minimo tra due nodi di un grafo, 124. Conclusioni, 138.

139 **VII. Informatica e modelli**

Modelli come approssimazione della realtà, 140. Modelli di progetto e modelli di utilizzo, 144. I rischi dei modelli, 153.

163 **Indice dei termini definiti e degli argomenti**

165 **Letture di altri libri di base**

167 **Altre letture**

I. INFORMATICA E VITA DI OGNI GIORNO

Come si cerca un numero di telefono – Come si consulta un orario – Come si fa un orario – Come si calcola (in alcuni casi) quanto dobbiamo aspettare un autobus – Conclusioni

Ciascuno di noi ogni giorno ha la necessità di usare e produrre un grande numero di informazioni. Informazione è quella che acquisiamo dall'orologio quando leggiamo che ora è, quella che forniamo a un automobilista straniero che ci chiede come fare per andare a San Pietro, quella che scopriamo sulla faccia del medico quando gli chiediamo come sono andate le analisi.

Molte discipline, nate e sviluppatesi in contesti culturali diversi, hanno lo scopo di capire a fondo la natura e i princípi che regolano l'uso e la produzione delle informazioni.

Le scienze del linguaggio, per esempio, studiano, tra gli altri aspetti, il modo in cui è fatto il linguaggio verbale[1], le somiglianze e differenze rispetto agli altri sistemi di comunicazione (immagini, gesti, ecc.), la struttura e l'evoluzione delle varie lingue.

Dell'informazione si occupa in modo specifico una disciplina relativamente recente, chiamata ormai comunemente 'informatica'. Prima di tentarne una definizione, è bene chiarire subito che, come accade in molti altri casi, non è possibile pensare di tracciare un solco preciso che separi ciò che nella ricerca, nella produzione, nella vita di ogni giorno si possa etichettare come informatica e cosa no.

1. Vedi *Guida all'uso delle parole* di Tullio De Mauro, «Libri di base» 3, Roma, Editori Riuniti, 1983[6].

Cominciamo con una prima definizione che piú avanti sarà perfezionata e arricchita. L'informatica è la disciplina che ha lo scopo di 'automatizzare', cioè delegare a strumenti automatici, la soluzione di problemi che, a partire da un insieme di informazioni a disposizione, richiedono la produzione di nuova informazione.

Nella vita di ogni giorno incontriamo una grande varietà di situazioni in cui dobbiamo produrre nuova informazione, ricavandola da altra informazione a nostra disposizione. Se il problema che ci viene posto ci è abbastanza familiare, il processo mentale con cui riusciamo a produrre tale nuova informazione è immediato, addirittura istintivo. Può essere invece molto arduo, addirittura impossibile da risolvere con la nostra razionalità e le nostre conoscenze, se il problema è nuovo e ci viene posto per la prima volta.

In questo capitolo introduttivo, per precisare meglio il significato della definizione appena data e per spiegare le motivazioni e lo scopo del libro, vogliamo partire appunto da alcuni problemi di produzione di informazione che ci si possono presentare nella vita di ogni giorno. L'analisi di questi problemi occuperà quasi tutto il capitolo: solo verso la fine saremo in grado di trarre le conclusioni che ci interessano.

1. Come si cerca un numero di telefono. Supponiamo di dover organizzare un breve viaggio di una giornata, i cui termini specificheremo tra poco. Una delle tappe prevede un appuntamento di lavoro. Per evitare di fare un viaggio a vuoto, è meglio assicurarsi che la persona con cui abbiamo un appuntamento sia effettivamente ad aspettarci. Per verificarlo non ci resta che telefonare. Supponiamo che la persona si chiami Sorgi. Se il numero di Sorgi non ce lo ricordiamo a mente, potremo consultare la nostra agendina telefonica. La pagina dell'agendina che contiene i numeri di persone o enti la cui lettera iniziale va da S a Z è riportata nella fig. 1.

Le informazioni riportate nell'agendina sono organizzate in modo vario. Accanto a righe in cui c'è un cognome e un numero di telefono, altri riferimenti compaiono tramite nomi, magari con le iniziali del cognome puntate. Altre volte il cognome è seguito dall'indicazione del luogo di lavoro; oppure compare la sigla di un'azienda e tra parentesi il cognome della persona che interessa. In un altro caso il numero è

Fig. 1: Esempio di pagina di agendina telefonica.

cancellato con una barra, e il nuovo numero è indicato con una freccia.

In questa pagina della nostra agendina, in ogni caso, Sorgi non c'è. Ci può venire in mente di cercarlo sotto il nome della sua azienda, o sotto il nome del socio. Se ancora non lo troviamo, possiamo fare un tentativo consultando l'elenco telefonico.

In un elenco telefonico è organizzata una massa enorme di informazioni. Al contrario dell'agendina, in cui i cognomi sono ordinati solo per gruppi di lettere, nell'elenco i cognomi sono tutti rigorosamente ordinati uno rispetto all'altro. È in base a questa proprietà che siamo abituati da tempo senza pensarci su, a usare metodi di ricerca molto rapidi. Dunque, dopo esserci procurati l'elenco della città o paese dove abita Sorgi, sappiamo che conviene aprire dapprima l'elenco in una zona presumibilmente vicina a dove si trova il cognome. A seconda del punto in cui abbiamo aperto l'elenco, dovremo successivamente avanzare o arretrare di qualche foglio. Una volta trovata finalmente la pagina giusta, trovato il primo Sorgi non ci resta che scandire le righe una dopo l'altra. Se di Sorgi ce ne sono tanti e noi non ricordiamo il nome

di quello che cerchiamo e sappiamo la via dove abita, converrà scegliere i Sorgi che abitano in quella via. Se ce n'è uno solo allora è probabilmente quello che cerchiamo (a meno che Sorgi abbia deciso di non rendere pubblico il suo numero), se ce n'è piú d'uno siamo nei guai.

Se anche in questo modo non troviamo il numero, possiamo cercare in qualche vecchia agenda, magari portata in cantina come ultimo cimitero prima di essere buttata, oppure telefonare a qualche conoscente comune che sa il numero.

Nel cercare un numero di telefono possiamo dunque utilizzare differenti depositi di informazione: la memoria, l'agenda, l'elenco telefonico, vecchie agende in disuso, qualche nostro amico. Ognuno di questi depositi ha un suo modo di organizzare e rappresentare le informazioni. Tipi di organizzazione e di rappresentazione sono strettamente legati al modo in cui tali informazioni sono piú frequentemente usate.

2. Come si consulta un orario. Supponiamo ora che il nostro viaggio si svolga in Brianza. Partendo da Carugo, nell'arco di una giornata dobbiamo andare prima a S. Eusebio per l'appuntamento di cui si è detto, e poi passare da Milano, vicino al Carrobbio. Non avendo a disposizione una macchina, dobbiamo usare la rete dei trasporti pubblici.

La Regione Lombardia fornisce agli utenti della rete dei trasporti pubblici della Brianza un libretto in cui sono contenute informazioni riguardanti gli orari di tutti i mezzi di trasporto pubblici: treni, autolinee, tramvie. Sulla copertina è riportata una pianta stilizzata della Brianza (vedi fig. 2) che rappresenta tutti i paesi e la rete di trasporti che li collega; a ogni tipo di trasporto corrisponde un diverso colore. In alto a sinistra sono riportati i significati dei vari colori: nella riproduzione della figura 2 non compaiono i colori, ma per il discorso che qui intendiamo fare si tratta di una mancanza di poco conto.

Per arrivare a S. Eusebio partendo da Carugo esistono varie possibilità. Se siamo già abituati da tempo a consultare mappe, usando l'esperienza e la sofisticata tecnologia a nostra disposizione (occhi, cervello) siamo in grado di individuare le alternative in un tempo rapidissimo e ancora una volta senza pensarci su.

Ma se non abbiamo nessuna esperienza di consultazione di

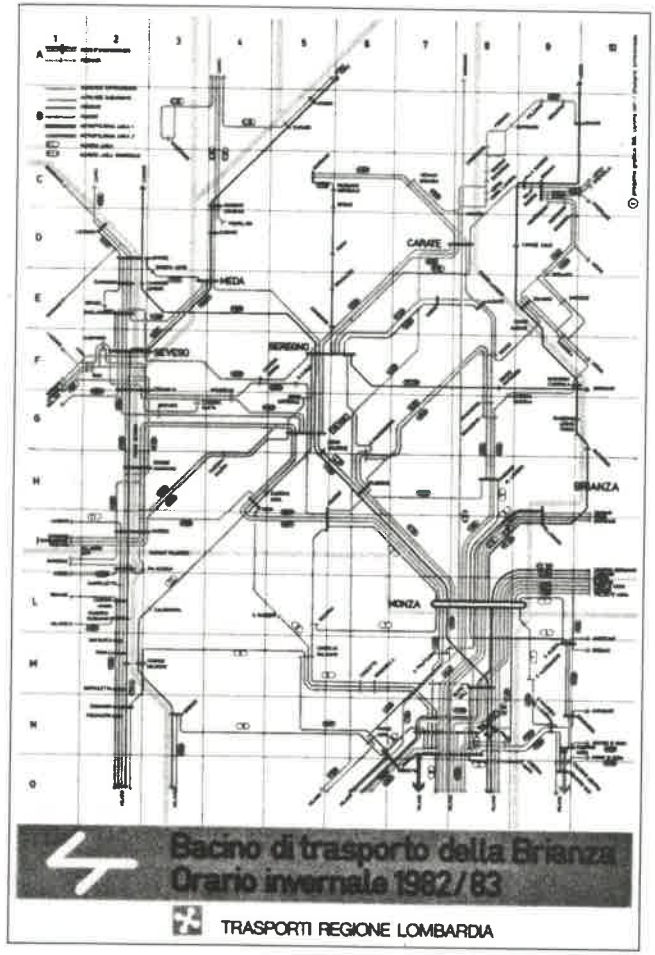


Fig. 2.

mappe, il compito può anche essere molto complicato. Quando ci troviamo di fronte a un problema mai affrontato prima è necessario cercare una strategia, un metodo per risolverlo.

Nel caso della mappa, potremmo partire da Carugo e cercare tutti i rami che collegano Carugo con i paesi adiacenti, magari scartando subito tutti i paesi che sono in direzione opposta a S. Eusebio. Con questo criterio, però, non possiamo avere alcuna certezza di individuare il cammino piú breve: lo facciamo al solo scopo di semplificarci la vita.

Dai paesi adiacenti a Carugo possiamo poi cercare percorsi o cammini che vadano direttamente a S. Eusebio o comunque ci si avvicinino: e cosí via finché non troviamo la soluzione. Un'altra strategia può invece procedere a tenaglia: possiamo cioè alternativamente partire da Carugo andando in avanti e da S. Eusebio andando all'indietro, usando la precedente strategia finché non riusciamo a collegare i due spezzoni di cammino.

Con un po' di pazienza dovremmo farcela: e poi, una volta capito il metodo, la tecnologia fornita dall'occhio e dal cervello ci permette di essere molto rapidi nella esecuzione. Troveremo allora che per arrivare a S. Eusebio ci sono almeno due alternative: o prendere il treno, scendere a Cormano, prendere l'autobus per Cinisello Balsamo e poi un altro autobus per S. Eusebio; oppure il treno fino a Meda, l'autobus fino a Desio, poi il tram fino a Nova e finalmente un autobus fino a S. Eusebio.

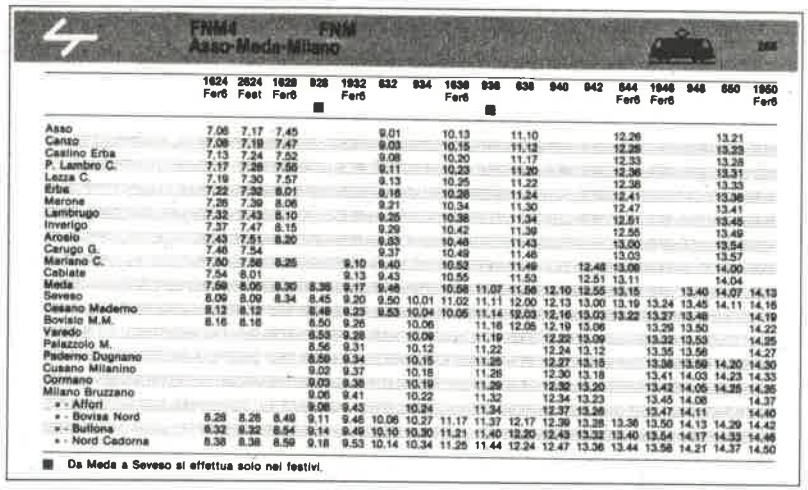
Il problema di partenza era:

trovare il modo piú rapido
per raggiungere S. Eusebio
partendo da Carugo
e utilizzando la rete
dei trasporti pubblici
della Brianza

Per risolverlo abbiamo dovuto prima affrontare un altro problema, in qualche modo legato al primo:

trovare tutti i modi alternativi
possibili per raggiungere
S. Eusebio
partendo da Carugo
e utilizzando la rete
dei trasporti pubblici
della Brianza

L'informazione che abbiamo a disposizione per risolvere questo primo problema è una mappa, rappresentata median-


FNM4 FNM
Asso-Meda-Milano

	1624 Feré	2624 Feat	1628 Feré	928 ■	1932 Feré	632	934	1638 Feré	936 ■	638	940	942	644 Feré	1946 Feré	948	650	1950 Feré
Asso	7.06	7.17	7.45			9.01		10.13		11.10			12.26			13.21	
Cantù	7.08	7.19	7.47			9.03		10.15		11.12			12.28			13.23	
Casllno Erba	7.13	7.24	7.52			9.08		10.20		11.17			12.33			13.28	
P. Lambro C.	7.17	7.28	7.56			9.11		10.23		11.20			12.36			13.31	
Lezza C.	7.19	7.30	7.57			9.13		10.25		11.22			12.38			13.33	
Erba	7.22	7.32	8.01			9.16		10.28		11.24			12.41			13.36	
Merone	7.25	7.39	8.06			9.21		10.34		11.30			12.47			13.41	
Lambrugo	7.32	7.43	8.10			9.25		10.38		11.34			12.51			13.45	
Inverigo	7.37	7.47	8.15			9.29		10.42		11.39			12.55			13.49	
Arosio	7.43	7.51	8.20			9.33		10.46		11.43			13.00			13.54	
Carugo G.	7.48	7.54				9.37		10.49		11.46			13.03			13.57	
Mariano C.	7.50	7.56	8.25		9.10	9.40		10.52		11.49		12.48	13.06			14.00	
Cabiate	7.54	8.01			9.13	9.43		10.55		11.53		12.51	13.11			14.04	
Meda	7.59	8.05	8.30	8.36	9.17	9.48		10.58	11.07	11.58	12.10	12.55	13.15		13.40	14.07	14.13
Seveso	8.09	8.09	8.34	8.45	9.20	9.50	10.01	11.02	11.11	12.00	12.13	13.00	13.19	13.24	13.45	14.11	14.16
Cesano Maderno	8.12	8.12		8.48	9.23	9.53	10.04	10.05	11.14	12.03	12.16	13.03	13.22	13.27	13.48		14.19
Bovisio M.M.	8.16	8.16		8.50	9.26		10.06		11.16	12.05	12.19	13.06		13.29	13.50		14.22
Varedo				8.53	9.28		10.09		11.19		12.22	13.09		13.32	13.53		14.25
Palazzolo M.				8.56	9.31		10.12		11.22		12.24	13.12		13.35	13.56		14.27
Paderno Dugnano				8.59	9.34		10.15		11.25		12.27	13.15		13.38	13.59	14.20	14.30
Cusano Milanino				9.02	9.37		10.18		11.28		12.30	13.18		13.41	14.03	14.23	14.33
Cormano				9.03	9.38		10.19		11.29		12.32	13.20		13.42	14.05	14.25	14.35
Milano Bruzzano				9.06	9.41		10.22		11.32		12.34	13.23		13.45	14.08		14.37
» - Affori				9.08	9.43		10.24		11.34		12.37	13.26		13.47	14.11		14.40
» - Bovisa Nord	8.28	8.28	8.49	9.11	9.46	10.06	10.27	11.17	11.37	12.17	12.39	13.28	13.36	13.50	14.13	14.29	14.42
» - Bullona	8.32	8.32	8.54	9.14	9.49	10.10	10.30	11.21	11.40	12.20	12.43	13.32	13.40	13.54	14.17	14.33	14.46
» - Nord Cadorna	8.38	8.38	8.59	9.18	9.53	10.14	10.34	11.25	11.44	12.24	12.47	13.36	13.44	13.58	14.21	14.37	14.50

■ Da Meda a Seveso si effettua solo nei festivi.

Fig. 3.

te un grafico a due dimensioni e a vari colori. L'informazione che dobbiamo produrre è un elenco di cammini.

Il primo dei cammini individuati è a occhio il piú comodo, ma se vogliamo essere precisi è necessario fare un po' di conti e consultare le tabelle contenute nel libretto. Anche qui, se non abbiamo un minimo di esperienza, dobbiamo individuare un metodo per estrarre dalla massa di informazioni riportate quelle che ci servono.

Prima di tutto dobbiamo trovare nel libretto la tabella oraria di nostro interesse.

La tabella della linea che collega Carugo a Cormano è riportata nella figura 3. Le informazioni sono organizzate in un insieme di colonne: nella prima sono riportate tutte le stazioni della linea, nelle successive, per ogni treno, gli orari di fermata del treno nelle varie stazioni.

Gli orari ferroviari comunemente in uso, a meno di particolarità tipografiche, sono in genere sempre organizzati come la tabella della figura 3. Qualche tempo fa erano in uso anche orari grafici, come quello riportato nella figura 4. In questi orari, come si vede, la tabella è suddivisa in fasce orarie; l'orario di ogni treno è rappresentato con una linea, tanto piú ripida quanto piú il treno è veloce. A ogni fermata la linea si spezza, a indicare il minuto di arrivo e quello di partenza. In questo tipo di orari alcune informazioni sono

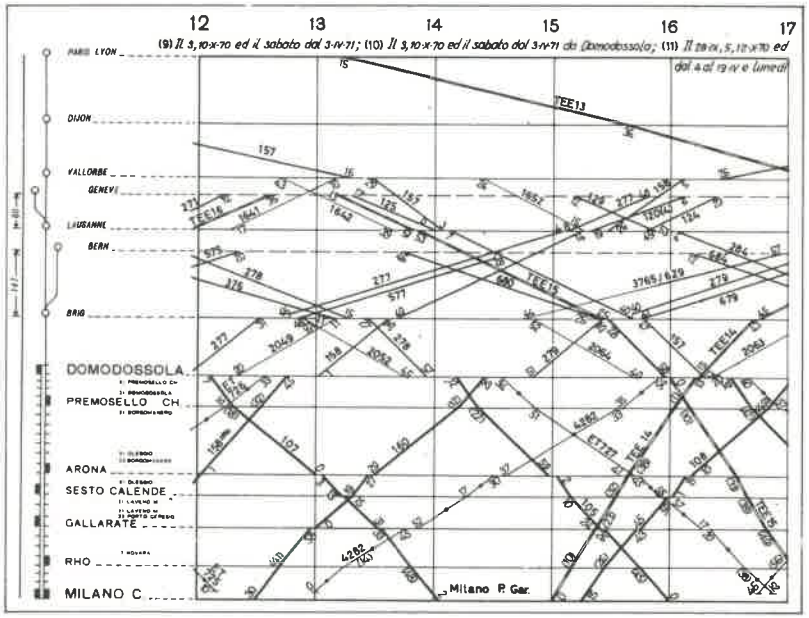


Fig. 4.

rappresentate in maniera efficace, ma certamente poco precisa. Per esempio, la maggiore o minore inclinazione delle linee ci fornisce in maniera visivamente efficace informazioni sulla velocità del treno, ma se vogliamo sapere quanto tempo un treno ferma a una stazione possiamo apprezzare questa informazione solo in maniera imprecisa.

Tornando alla tabella della figura 3, scopriamo che non c'è nessun treno che collega Carugo a Cormano senza scendere. Arrivati a Bovisio alcuni treni proseguono direttamente per Milano Bovisa Nord: ce ne accorgiamo perché da Varedo a Milano Affori non compare nessun numero, e perciò interpretiamo lo spazio bianco come «non fermata».

A questo punto dobbiamo esaminare due alternative possibili. Per esempio scendere a Meda e prendere un altro treno: ci accorgiamo che questo è possibile perché a occhio scopriamo che a Meda fermano tutti i treni; l'altro posto dove fermano tutti i treni è Seveso. L'alternativa è proseguire fino a Milano Bovisa Nord e prendere poi un altro treno nella direzione opposta per Cormano.

Per ciascuna delle precedenti alternative e per l'altro itinerario che abbiamo individuato, dobbiamo pazientemente calcolarci gli orari, tenendo conto delle coincidenze.

Non ci interessa proseguire oltre nell'analisi dell'esempio. I calcoli che ci si presentano sono lunghi e noiosi. Tutti questi calcoli, come quelli che abbiamo osservato in dettaglio, coinvolgono manipolazioni di informazioni di vario tipo, rappresentate secondo diverse convenzioni.

Nei due problemi che abbiamo esaminato, le rappresentazioni delle singole informazioni e la loro organizzazione complessiva ci erano date. Dovevamo prima comprenderle, e, una volta comprese, trovare una strategia per ricavare le informazioni di nostro interesse.

Il problema che segue, invece, è piú insolito del precedente nell'esperienza delle singole persone, e molto diffuso, invece, nelle aziende e grandi organizzazioni. Si tratta di trovare un modo efficace di rappresentare una informazione.

3. Come si fa un orario. Se ci capita spesso di dover percorrere il tratto Carugo-S. Eusebio, conviene una volta per tutte costruire una nuova tabella che riassuma i percorsi pos-

luogo		itinerari					
		orario	tipo mezzo	orario	tipo mezzo	orario	tipo mezzo
Carugo	p.	6.18	treno	7.13	treno	8.34	treno
Meda	a.	6.30		7.27		8.56	
	p.	6.32	treno	7.31	autobus	9.00	autobus
Cormano	a.	7.08					
	p.	7.15	autobus				
Cinisello	a.	7.21					
	p.	7.27	autobus				
Milano B.N.	a.	6.55					
	p.	6.57	treno				
Desio	a.			7.42		9.12	
	p.			7.45	tram	9.14	tram
Nova	a.			7.52		9.20	
	p.			7.55	autobus	9.25	autobus
S. Eusebio	a.	7.35		8.01		9.45	

Fig. 5: Tabella di tutti gli itinerari tra Carugo e S. Eusebio.

sibili e i relativi orari. Dovremo fare un po' di lavoro la prima volta, ma tutte le altre volte che ci serve la tabella sarà già pronta per tutte le possibili esigenze di orario.

Dovremo organizzare la «tabella Carugo-S. Eusebio» in modo tale da rappresentare sinteticamente le informazioni che ci servono e permettere un loro rapido ritrovamento.

Una prima organizzazione è mostrata nella figura 5 (i dati sono inventati). Nella tabella della figura 5 sono riportati in prima colonna tutti i paesi che stanno su almeno un percorso. Ogni successiva coppia di colonne riporta le informazioni significative per un dato itinerario, e cioè gli orari di partenza e arrivo di ogni mezzo di trasporto e il tipo di mezzo utilizzato.

La soluzione della figura 5 contiene fin troppe informazioni che in parte si ripetono. È, come si dice, piuttosto 'ridondante' e anche poco leggibile. È piú sensato pensare a tre tabelle, ciascuna per ogni possibile itinerario. Naturalmente le tre tabelle dovranno contenere la stessa informazione della tabella della figura 5.

luogo		**itinerari**			
		orario	**tipo mezzo**	**orario**	**tipo mezzo**
Carugo	p.	7.13	treno	8.34	treno
Meda	a.	7.27		8.56	
	p.	7.31	autobus	9.00	autobus
Desio	a.	7.42		9.12	
	p.	7.45	tram	9.14	tram
Nova	a.	7.52		9.20	
	p.	7.55	autobus	9.25	autobus
S. Eusebio	a.	8.01		9.45	

Fig. 6: Orario Carugo - S. Eusebio via Nova.

Riportiamo nella figura 6 la tabella per l'itinerario Carugo-Meda-Nova-S. Eusebio. Anche in questa tabella c'è una evidente ridondanza: tutte le colonne relative ai mezzi di trasporto sono uguali. La soluzione piú semplice per eliminare la ridondanza è quella di suddividere la tabella in due tabelle (vedi fig. 7) in cui riportare rispettivamente i tipi di mezzi usati per ogni tratta e gli orari.

tratta	**tipo mezzo**
Carugo-Meda	treno
Meda-Desio	autobus
Desio-Nova	tram
Nova-S. Eusebio	autobus

luogo		**orari**		
Carugo	p.	7.13	8.34	9.12
Meda	a.	7.27	8.56	9.32
	p.	7.31	9.00	9.35
Desio	a.	7.42	9.12	9.55
	p.	7.45	9.14	9.57
Nova	a.	7.52	9.20	10.10
	p.	7.55	9.25	10.14
S. Eusebio	a.	8.01	9.45	10.26

Fig. 7: Tabella dei mezzi usati (in alto) e tabella degli orari (in basso).

La soluzione della figura 7 è sintetica; quanto alla efficacia, se la tabella deve servire a trovare il percorso piú rapido, a una data ora del giorno, la consultazione è ancora un po' scomoda. È meglio avere a disposizione una nuova tabella (vedi fig. 8) in cui sono riportati gli orari di partenza dei collegamenti piú rapidi.

Nei precedenti esempi l'informazione che ci veniva fornita per poter ricavare quella di nostro interesse era già data al massimo livello di dettaglio, e noi dovevamo, con un metodo opportuno, trovare, tra tante informazioni inutili, quelle significative. Nel prossimo esempio, invece, le informazioni sono espresse in modo sintetico da una 'legge generativa'.

ora partenza	**itinerario**
6.18	via Milano Bovisa Nord
7.13	via Nova
8.34	via Nova
. . . . .	

Fig. 8: Tabella dei collegamenti piú convenienti.

4. Come si calcola (in alcuni casi) quanto dobbiamo aspettare un autobus. Continuiamo il nostro viaggio. Una volta raggiunta Milano devo prendere il tram. Arrivato alla fermata mi chiedo quanto tempo devo aspettare. Alcune aziende tramviarie espongono alle fermate tabelle come quella riportata nella figura 9.

linea	orario
34	dalle 6,00 alle 8,45 ogni 15 minuti (1) dalle 9,00 alle 17,50 ogni 10 minuti (2) dalle 18,00 alle 24 ogni 15 minuti (1)
(1) domenica e festivi ogni 20 minuti (2) domenica e festivi ogni 15 minuti	

Fig. 9: Esempio di tabella degli orari di passaggio di un tram.

La tabella della figura 9 fornisce solo la legge generale che stabilisce i passaggi. Il tempo di attesa si ricava sottraendo dal presunto orario del prossimo tram quello segnato dal nostro orologio. Facciamo un esempio. Arrivo alla fermata alle 9,12. La legge generale della tabella dice che dalle ore 9 alle ore 17,30 l'autobus passa ogni 10 minuti. Quindi: 9; 9,10; 9,20; 9,30; ...; 11; 11,10; ecc.; se sono le 9,12, vuol dire che l'ultimo autobus è passato 2 minuti fa e che il prossimo arriverà alle 9,20, fra 8 minuti. Naturalmente, tutto il ragionamento vale a patto che ci fidiamo della puntualità dei tram.

Questa operazione può richiedere un certo tempo, se la legge generativa è un po' complicata. Ma non è questo il caso dell'esempio. Sarebbe probabilmente piú faticoso cercare in una grossa tabella l'orario che ci interessa.

A Los Angeles, per esempio, le informazioni sugli orari sono organizzate come nella figura 10. In questa tabella sono riportati gli orari di passaggio nelle fermate principali della linea 1 nei giorni lavorativi. Quando non è possibile trovare una sintetica legge generativa si deve ricorrere a questi orari, molto poco leggibili.

Per sapere quanto dovrò aspettare a una certa fermata, ovvero quando passerà l'autobus dalla mia fermata, dovrò conoscere la posizione approssimata della fermata rispetto alle due adiacenti riportate in tabella. Per far questo potrò consultare una mappa di Los Angeles (vedi fig. 11, in cui

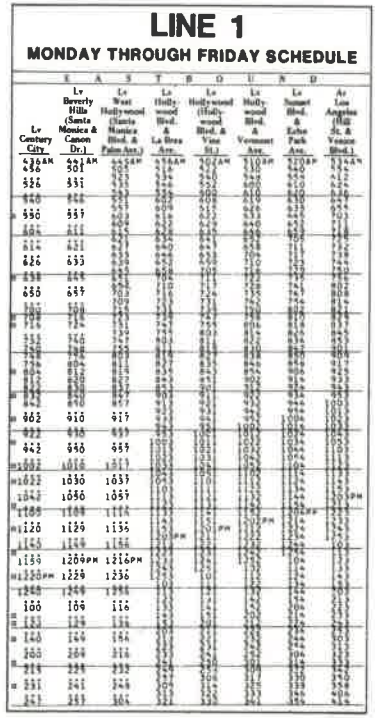

LINE 1
MONDAY THROUGH FRIDAY SCHEDULE

Fig. 10.

Fig. 11.

sono riportate le posizioni del capolinea e della prima fermata e calcolare il tempo di passaggio con una proporzione.),

La conclusione che possiamo trarre da quest'ultimo esempio è che la presenza di una regolarità nelle informazioni che riguardano il nostro problema permette di rappresentarle e usarle in modo piú efficace.

Abbiamo cosí concluso l'indagine sulle informazioni elaborate nel nostro itinerario. In tutti i casi esaminati, le informazioni a nostra disposizione si sono rivelate utili ai nostri scopi. Ma sappiamo bene dalla nostra esperienza che non è sempre cosí. Se sull'autobus dobbiamo stabilire a quale fermata scendere e qualcuno ci dice «tra sette fermate», basterà contare le fermate che fa il tram (stando attenti se ne salta qualcuna). Ma se incontriamo una persona che ci dice «deve scendere due fermate prima del capolinea» questa informazione, se non siamo pratici della zona e della fermata, è di ben poco aiuto.

5. Conclusioni. Con i precedenti esempi abbiamo voluto mostrare che la vita di ogni giorno è piena di situazioni in cui noi usiamo o produciamo informazione.

Quando ci troviamo di fronte a un nuovo problema, mai precedentemente affrontato, abbiamo visto (ma soprattutto sappiamo per esperienza) che è necessario trovare quello che per il momento possiamo chiamare un «metodo di risoluzione». Per cercare questo metodo, dobbiamo aver compreso anzitutto la natura profonda delle informazioni a nostra disposizione e come esse si collegano concettualmente tra di loro. Questa prima comprensione spesso non basta, perché il legame tra le informazioni a nostra disposizione e quelle che dobbiamo produrre può essere cosí complesso che risulta molto difficile o impossibile trovare un metodo per riuscire a calcolare tali informazioni.

Questa impossibilità può nascere a ben vedere da due diverse ragioni:

1. da una parte perché non riusciamo a trovare un metodo di risoluzione; e ciò può accadere per i limiti della nostra cultura, della nostra intuizione, della nostra capacità di ragionare;
2. dall'altra perché, pur conoscendo il metodo risolutivo, i limiti tecnologici delle risorse a disposizione ci impediscono di calcolarne la soluzione. L'uomo, visto come macchina per calcolare, ha capacità molto limitate.

I 'calcolatori elettronici' nascono verso la fine della seconda guerra mondiale per rispondere essenzialmente alla seconda esigenza. La guerra, e in particolare la necessità di calcolare rapidamente le orbite dei missili, accelera le ricerche già in corso da qualche anno per produrre i primi prototipi di calcolatori. I primi calcolatori usano una tecnologia molto rudimentale, si rompono facilmente, il loro uso comporta la scrittura di programmi in linguaggi estremamente poveri (nei prossimi capitoli torneremo a lungo sulle parole calcolatore, programma, linguaggio programmativo). Dal momento in cui i primi calcolatori cominciano a diventare prodotti commerciali, si può dire che la ricerca e lo sviluppo hanno avuto due grandi traguardi.

1. Avvicinare il calcolatore all'uomo, migliorando continuamente la tecnologia. Ciò ha dato luogo a uno straordinario, rapido aumento nella efficienza delle prestazioni e della affidabilità e a una altrettanto rapida riduzione dello spazio occupato e dei costi di produzione (vedi piú avanti «Come cambiano i calcolatori elettronici»). Nello stesso tempo c'è stata una continua produzione di nuovi linguaggi programmativi sempre piú potenti e vicini, nelle loro strutture linguistiche, al modo in cui l'essere umano esprime i metodi di risoluzione.
2. Avvicinare l'uomo al calcolatore, proponendo sempre nuovi metodi per guidare l'uomo al progetto di nuovi metodi di risoluzione (ovvero nella analisi di metodi esistenti). Tale progetto, abbiamo visto, è il passo decisivo nella organizzazione di qualsiasi calcolo.

Ora, mentre la tecnologia è evoluta in maniera vertiginosa portando a una drastica riduzione dei costi, piú lenta è stata la produzione di nuove metodologie di progetto e di analisi. Lo scopo principale di questo libro è introdurre il lettore a quell'insieme di metodi che faticosamente stanno prendendo forma in questi anni. In alcuni aspetti questi metodi possono dirsi a uno stadio maturo di evoluzione scientifica, e sono dunque destinati, a breve termine, a entrare nella cultura dell'uomo moderno.

Nell'esaminare questi metodi, ci accorgeremo che essi, concepiti spesso per automatizzare la soluzione di problemi che hanno come oggetto l'uso e la produzione della informazione, ci forniscono in realtà una razionalità anche per una informatica senza calcolatore, l'informatica chiamata «povera» nel bel libro di Giovanni Lariccia, *Le radici dell'informatica*. E ci potremo rendere conto che se la loro formulazione è stata accelerata dalla comparsa del calcolatore elettronico, tuttavia molte idee e strategie in essi contenute sono già presenti nell'approccio dell'informatica povera. In altre parole, se non vi è razionalità nel trattare le informazioni senza il calcolatore, non è detto che il calcolatore risolverà automaticamente i nostri problemi.

Per fare un esempio, se non si è capito nel passato che gli archivi fatti di schede e contenitori cartacei dovevano essere ordinati in modo tale da rendere facile trovare le pratiche

Come cambiano i calcolatori elettronici

Giornali e riviste contengono quasi in ogni numero articoli e pubblicità sul *computer* (parola inglese equivalente alla parola italiana calcolatore). I calcolatori stanno veramente invadendo la nostra vita di tutti i giorni. La tabella 1 può dare una idea molto concreta di questa invasione; fornisce l'andamento delle consegne (per i prossimi anni una previsione) dei soli calcolatori personali (o *personal computers*), calcolatori cioè utilizzabili da un utente per volta, di costo relativamente basso (dai 3 agli 8 milioni a seconda della configurazione) e dotati di programmi che ne permettono le applicazioni piú disparate.

Da quando i calcolatori elettronici sono diventati, dopo i primi risultati sperimentali, una tecnologia prodotta su scala industriale, hanno subíto una evoluzione cosí rapida da non avere pari con nessuna altra tecnologia ad alta diffusione. Per avere una misura quantitativa di questa evoluzione, abbiamo riportato nella tabella 2 l'andamento nel tempo di alcuni tra le piú significative caratteristiche o parametri dei calcolatori.

Tabella 1 **Andamento delle consegne di calcolatori personali**

anno	consegne	anno	consegne
1977	47.000	1982	1.360.000
1978	175.000	1983	2.100.000
1979	284.000	1984	2.900.000
1980	500.000	1985	3.700.000
1981	825.000		

Fonte: *Times*, 24 luglio 1981

Tabella 2 **Andamento nel tempo di alcuni parametri tecnologici dei calcolatori**

		parametri tipici			
generazione	**elemento caratteristico**	**velocità**	**dimensioni**	**affidabilità**	**costo**
prima	valvola	1	1	1	1
seconda	transistore	10	1/100	100	1/10
terza	circuito integrato	100	1/1.000	1.000	1/100
quarta	circuito integrato a larga scala	1.000	1/10.000	10.000	1/1.000

A seconda delle caratteristiche costruttive e delle prestazioni fornite si suole dividere i calcolatori in generazioni: nella tabella 2, l'elemento scelto per caratterizzare le varie generazioni è l'elemento costruttivo basico. La tabella si commenta da sé. Con tale vertiginoso aumento delle prestazioni e riduzione dei costi, le funzioni svolte da un calcolatore che trent'anni fa pesava diverse tonnellate, occupava un intero piano di un edificio, costava un prezzo che soltanto grandi organizzazioni potevano sopportare e aveva ogni giorno qualche guasto, oggi sono svolte da un calcolatore che ha l'ingombro di una macchina da scrivere, non si rompe (quasi) mai e ha prezzi accessibili a piccole organizzazioni e, almeno come tendenza, a vaste categorie di acquirenti singoli.

avendo a disposizione solo il cognome del richiedente informazioni, non lo si capirà neanche nel momento in cui al posto dell'archivio si utilizza un calcolatore. Anzi, l'introduzione del calcolatore può rivelarsi addirittura disastrosa, perché almeno con gli archivi di carta si può sempre inventare sul momento qualche metodo empirico: «Mi pareva che l'altra volta stesse in quell'armadio, tenga anche presente che c'era un foglio rosso un po' stracciato in fondo». Con gli archivi magnetizzati, per nostra stessa decisione, siamo stati espropriati oltre che del metodo per ritrovarla, anche della stessa rappresentazione fisica della informazione, ormai nascosta, muta e inaccessibile.

Prima di intraprendere il discorso sui metodi, avremo bisogno di introdurre nel capitolo II alcuni concetti di base che saranno utilizzati nel resto del libro, quali algoritmo (cioè procedimento di risoluzione di un problema), programma (cioè algoritmo scritto in un linguaggio comprensibile per un calcolatore), linguaggio programmativo. Introdurremo anche un semplice linguaggio per esprimere algoritmi. La comprensione di questo capitolo è essenziale per ciò che segue, ma, se arrivato alla fine del capitolo II il lettore avrà qualche difficoltà di comprensione, non si scoraggi perché i concetti piú importanti verranno ripresi piú avanti.

Nel capitolo III affronteremo il piú cruciale tra i problemi

connessi all'automazione dei problemi (ma fondamentale anche quando siamo noi a eseguire il problema): come si fa a esser sicuri che l'algoritmo (il procedimento di risoluzione) è corretto, cioè risolve proprio il problema che doveva risolvere? Scopriremo un primo limite, in parte teorico, in parte pratico, dell'informatica: è quasi impossibile esser certi di non commettere errori.

Un'altra importante questione connessa all'uso delle macchine per risolvere problemi è affrontata nel capitolo IV, una domanda che è a metà strada tra la scienza e la filosofia: che cosa possono fare le macchine? Scopriremo che tutti coloro che hanno in qualche modo dato una definizione formale del concetto di calcolo algoritmico, hanno fornito modelli equivalenti tra loro, dando in tal modo al concetto stesso una grande solidità nella scienza moderna. E vedremo che, alla luce di tale concetto di calcolo, esistono problemi le cui soluzioni non possono essere calcolate mediante algoritmi.

Ma se è importante sapere se oggi un calcolatore (o un uomo) può in teoria risolvere in modo algoritmico un dato problema, è anche importante sapere se lo può risolvere nella pratica, con uso ragionevole di risorse, con la tecnologia odierna o con quella che possiamo prevedere che avremo a disposizione tra dieci o cent'anni. Nel capitolo V affronteremo perciò lo studio della efficienza degli algoritmi. Troveremo così altri limiti dell'informatica, legati all'esistenza di algoritmi di così complessa soluzione che non abbiamo speranza di risolverli (esattamente) in casi reali con la tecnologia di oggi o del futuro.

I metodi per la correttezza ed efficienza esaminati nei capitoli III e V sono prevalentemente metodi di analisi, nel senso che assumono l'algoritmo come dato e ne valutano una qualità (se è corretto o se è efficiente). Nel capitolo VI affronteremo invece i metodi di progetto, che hanno lo scopo di aiutare il progettista nella sintesi di un algoritmo a partire da un problema. Scopriremo quanto sofisticata debba essere in generale l'attività del progettista per ideare algoritmi, tanto da farlo apparire talvolta come un artista. Nel capitolo VII si esamina il rapporto che sussiste tra le esigenze di chi usa lo strumento di calcolo e le necessarie (o volute) approssimazioni che l'uso di tale strumento comporta. E parleremo perciò dell'uso (e abuso) che in informatica si fa dei modelli.

II. CONCETTI INTRODUTTIVI

Algoritmi e programmi - Risorse a disposizione di un calcolatore - Linguaggi macchina e linguaggi programmativi - I grafi di flusso come mezzo di descrizione di algoritmi

1. Algoritmi e programmi. Dopo la discussione fatta nel capitolo I, possiamo ridefinire l'informatica come la disciplina che studia le *tecnologie* che permettono di automatizzare procedimenti risolutivi di problemi e i *metodi* che permettono di usare in maniera efficace, affidabile ed efficiente tali tecnologie. Cominciamo a vedere quali problemi l'informatica ci aiuta a risolvere.

I problemi a cui qui facciamo riferimento possono essere di natura molto varia. A ogni problema è sempre associata una descrizione delle informazioni a disposizione espressa in genere nella sua prima formulazione in 'linguaggio naturale', ovvero anche in altri tipi di linguaggi, per esempio il linguaggio matematico. Non daremo qui una definizione di problema; preferiamo aiutarci con degli esempi (accanto a quelli già discussi nel capitolo I).

Problema 1. Dati due numeri, calcolarne la somma.

Problema 2. Data un'agendina (o un elenco telefonico), trovare il numero di telefono di una persona.

Problema 3. Data la struttura della rete stradale di una città e le informazioni sui flussi veicolari in ciascuna delle strade in ogni momento della giornata, fornire i tempi di rosso e verde di ciascuno dei semafori presenti nella rete, con

l'obiettivo di ridurre al minimo il tempo di percorrenza complessivo degli itinerari degli utenti della rete.

Problema 4. Scrivere tutti gli n per cui la equazione $X^n + Y^n = Z^n$ ha soluzioni X, Y, Z intere.

Problema 5. Progettare la struttura di un ponte.

Problema 6. Stabilire il deficit della bilancia dei pagamenti italiana in un dato anno.

Riguardo al modo in cui sono formulati tali problemi c'è da osservare che:

1. la descrizione del problema non fornisce in genere alcun metodo per calcolare il risultato dai dati di partenza, o, come si dice riferendosi al calcolatore, dati di ingresso;
2. la descrizione del problema è talvolta ambigua o imprecisa (in molti casi non può certo dirsi formale) nel senso che a essa possono essere associate molteplici interpretazioni. Si pensi al problema 2: che risposta occorre dare quando si cerchi in un elenco il numero di telefono di Paolo Rossi e quindi, probabilmente, esista piú di una persona con quel nome e cognome nell'elenco telefonico?
3. non è addirittura detto che allo stato attuale delle conoscenze o in assoluto sia possibile calcolare i risultati del problema. È questo il caso del problema 4, formulato la prima volta dal matematico francese Pierre de Fermat (1601-1665), per cui non è ancora stata individuata una soluzione;
4. le informazioni in ingresso e quelle fornite come soluzione possono essere di natura molto generale. Nel problema 5 le informazioni a disposizione sono la conformazione del terreno nel luogo dove costruire il ponte, la natura geologica, i dati sui venti e sulle condizioni meteorologiche nella località dove verrà costruito il ponte, ecc., e il risultato dovrà essere la struttura del ponte, espressa dal numero e dalla forma dei piloni, materiale impiegato, ecc. Nel problema 6, come è noto, non è neanche ben chiaro quali siano le informazioni a disposizione e quale sia il loro grado di attendibilità.

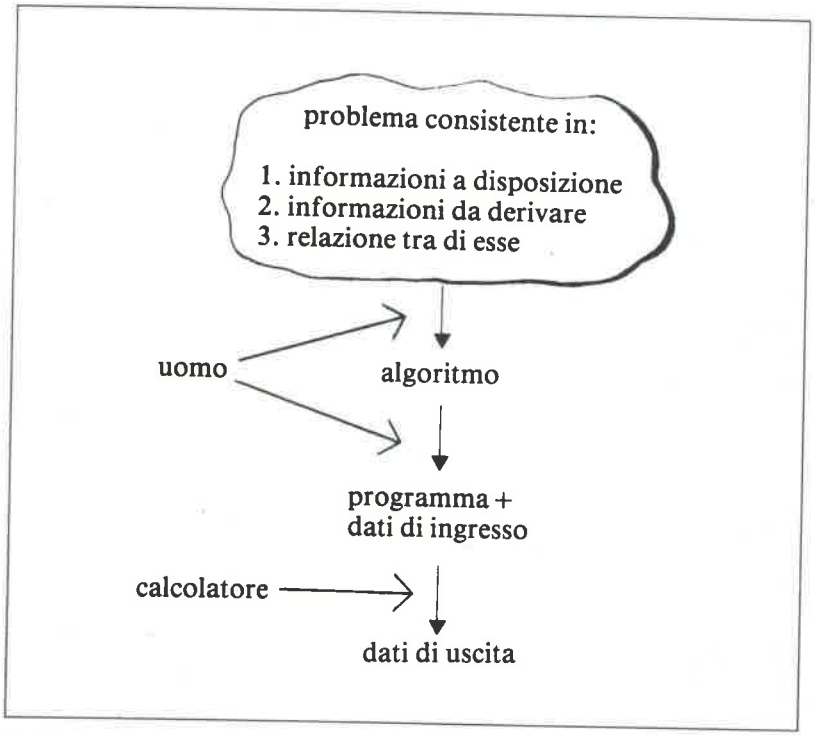


Fig. 12: Schema della automazione nella risoluzione di problemi.

Finora non abbiamo detto nulla su come sono fatti i calcolatori elettronici. Tuttavia dovrebbe essere chiaro che, per poterli utilizzare allo scopo di automatizzare la soluzione di problemi (vedi fig. 12), l'uomo deve sempre intervenire allo scopo di:

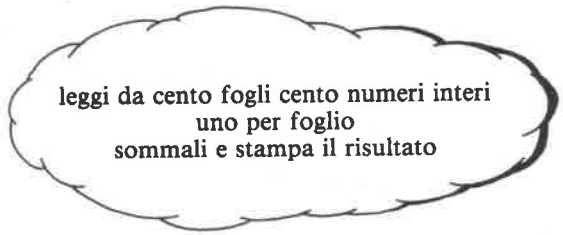
1. individuare (almeno) un procedimento risolutivo del problema, cioè un insieme di regole od operazioni che, eseguite ordinatamente, permettono di calcolare i risultati del problema a partire dalle informazioni a disposizione; chiameremo nel seguito 'algoritmo' un tale procedimento risolutivo;
2. individuare un modo di rappresentare l'algoritmo, le informazioni a disposizione e quelle utilizzate dall'algoritmo nel corso del calcolo in un linguaggio comprensibile per l'elaboratore. Chiameremo 'dati di ingresso' le rappresentazioni (fornite al calcolatore) delle

informazioni a disposizione, 'programma' la rappresentazione dell'algoritmo, 'dati di uscita' le rappresentazioni fornite dal calcolatore come conseguenza del processo di calcolo.

Quando l'uomo utilizza il calcolatore, delega a esso la soluzione di un problema, in cui lui stesso, però, continua ad avere una parte di grande rilevanza. Vogliamo subito far notare, ma riprenderemo questo concetto, che tale processo di delega è un fatto politico di grande importanza nella società odierna. È tale delega che trasferisce allo strumento calcolatore (e a chi lo sa e può usare) uno straordinario potere in un grande numero di processi decisionali che coinvolgono temi come l'organizzazione del lavoro, il controllo sociale, l'economia, la guerra.

Dobbiamo ora approfondire i termini appena introdotti di algoritmo e dato. Per quanto riguarda il concetto di dato, ricordiamo che ciò che lo contraddistingue dal concetto di informazione è il fatto di essere espresso per mezzo di una rappresentazione direttamente elaborabile dal calcolatore. Nel seguito, fino al capitolo VII in cui discuteremo nuovamente il rapporto tra informazioni e dati, non utilizzeremo piú il concetto di informazione, e parleremo sempre di dato, non solo con riferimento ai programmi ma anche con riferimento agli algoritmi. Sul concetto torneremo piú avanti, e cioè nel capitolo VI; per il momento, utilizzeremo nel discorso una classe di dati molto comune, quella dei numeri interi. Approfondiamo invece qui il concetto di algoritmo.

Consideriamo per esempio il problema:



Questo problema può essere risolto abbastanza facilmente da un essere umano, a patto che sappia capire l'italiano e sommare dei numeri. Un possibile algoritmo risolutivo può essere:

Algoritmo somma 100 numeri

1. Poni la variabile CONTATORE al valore 1.
   (Una 'variabile' è un dato il cui valore cambia nel corso della esecuzione dell'algoritmo. A ogni variabile che utilizzeremo nell'algoritmo è associato un nome che identifica univocamente il dato nella descrizione dell'algoritmo. In questo caso, il valore 1 è assegnato alla variabile di nome CONTATORE. Per rappresentare il valore della variabile CONTATORE, l'esecutore potrà utilizzare un foglio, su cui riporterà i diversi valori assunti dal dato).
2. Poni la variabile SOMMA a 0.
3. Leggi da un foglio un valore e assegnalo alla variabile NUMERO. Butta il foglio.
4. Somma le variabili SOMMA e NUMERO e metti il risultato in SOMMA.
5. Se CONTATORE è uguale a 100 vai a 8, altrimenti prosegui.
6. Somma 1 a CONTATORE e metti il risultato in CONTATORE.
7. Vai a 3.
8. Scrivi SOMMA su un foglio.
9. Fermati.

I linguaggi usati per descrivere algoritmi e farli eseguire da calcolatori elettronici, i cosiddetti 'linguaggi programmativi', usano parole standard e regole per la costruzione delle 'istruzioni' piú rigide di quelle permesse dalla lingua che usiamo tutti i giorni per comunicare con altri esseri umani e che chiamiamo linguaggio naturale. Inoltre in tali linguaggi è possibile usare soltanto un limitato numero di istruzioni basiche, ognuna delle quali ha un preciso ruolo nel linguaggio.

Mostriamo nella figura 13 un esempio di programma che risolve il problema della somma di 100 numeri, scritto nel 'linguaggio programmativo Pascal'.

```
PROGRAM ESEMPIO;
VAR SOMMA, NUMERO, I: INTEGER;
BEGIN
    SOMMA:=0;
    FOR I:= 1 TO 100 DO
        BEGIN
            READ (NUMERO);
            SOMMA:=SOMMA+NUMERO;
        END;
    WRITE (SOMMA)
END.
```

Fig. 13: Esempio di programma in linguaggio Pascal.

```
PROGRAMMA ESEMPIO;
LE VARIABILI USATE NEL PROGRAMMA SONO:
    1. SOMMA
    2. NUMERO
    3. I
ESSE SONO INTERE;
COMINCIA:
    1. PONI SOMMA UGUALE A 0;
    2. PER 100 VOLTE ESEGUI
        COMINCIA:
            2.1. LEGGI UN VALORE NELLA VARIABILE
                 NUMERO;
            2.2. AGGIUNGI NUMERO A SOMMA
                 E PONI IL RISULTATO IN SOMMA;
        FINE;
    3. SCRIVI IL VALORE DELLA VARIABILE SOMMA
FINE.
```

Fig. 14: Versione in lingua italiana del programma Pascal.

Il Pascal, come molti altri linguaggi programmativi, utilizza parole chiave prese dal vocabolario inglese: traduciamo allora il programma in una nuova versione, utilizzando al posto di tali parole chiave una forma piú vicina a quella utilizzata nel linguaggio scritto (vedi fig. 14).

Analizziamo il programma della figura 14. La istruzione

```
PROGRAMMA ESEMPIO;
```

dichiara appunto che sta iniziando la descrizione di un programma, chiamato ESEMPIO.

La seconda frase:

```
LE VARIABILI USATE NEL PROGRAMMA SONO:
    1. SOMMA
    2. NUMERO
    3. I
ESSE SONO INTERE;
```

fornisce informazioni sulle variabili usate nel programma. Una variabile, come abbiamo già visto, rappresenta un dato il cui valore può in generale variare nel corso della esecuzione

del programma. Per identificarla, le viene associato un nome, con cui sarà riferita ogni volta che serva nel procedimento risolutivo. Le variabili del programma sono tutte intere, nel senso che i valori che esse possono assumere sono solo interi.

Comincia a questo punto il procedimento risolutivo vero e proprio, e questo è segnalato dalla parola

```
COMINCIA
```

Le istruzioni del programma sono identificate da numeri progressivi. La prima

```
1. PONI SOMMA UGUALE A 0;
```

dice appunto che occorre inizialmente assegnare il valore 0 alla variabile SOMMA.

La seconda istruzione è piú complessa, ed è il nucleo principale del metodo risolutivo:

```
2. PER 100 VOLTE ESEGUI
    COMINCIA:
        2.1. LEGGI UN VALORE NELLA VARIABILE NUMERO;
        2.2. AGGIUNGI NUMERO A SOMMA
             E PONI IL RISULTATO IN SOMMA;
```

Essa dice che per 100 volte devono essere eseguite due istruzioni:

2.1. la prima delle quali legge uno dei 100 valori nella variabile NUMERO.

2.2. la seconda, immediatamente dopo, somma il (vecchio) valore della variabile SOMMA a quello appena letto della variabile NUMERO e assegna il nuovo valore alla variabile SOMMA.

La istruzione

```
3. SCRIVI IL VALORE DELLA VARIABILE SOMMA
```

permette infine di stampare il risultato.

La parola FINE. chiude il programma.

2. Risorse a disposizione di un calcolatore. Per poter comprendere ed eseguire programmi, un calcolatore deve essere organizzato in un insieme di risorse in certo senso analoghe a quelle che utilizza un essere umano quando interpreta ed esegue un algoritmo (vedi fig. 15).

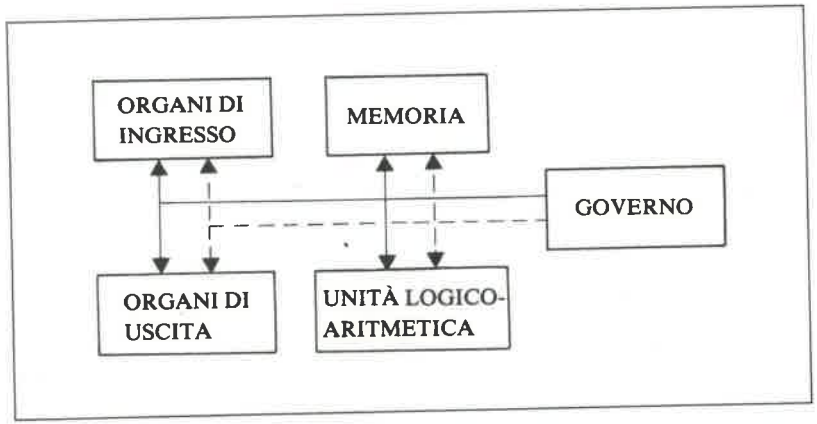


Fig. 15: Schema funzionale di un calcolatore.

Anzitutto, un calcolatore è dotato di organi detti di 'ingresso' e di 'uscita', che hanno lo scopo di permettere la comunicazione tra calcolatore e ambiente esterno[2]. Ricordando quanto detto in precedenza, l'uomo ha bisogno di comunicare al calcolatore:

1. l'algoritmo risolutivo, espresso tramite un programma;
2. i dati su cui il programma opera.

Il calcolatore deve a sua volta essere in grado di comunicare all'uomo i risultati dei calcoli.

Esempi di organi di ingresso sono:

1. una macchina capace di leggere (lettore), schede perforate, su cui istruzioni e dati sono perforati utilizzando opportuni codici;
2. la 'tastiera', simile a una tastiera di una macchina da scrivere.

2. Vedi anche *La rivoluzione elettronica* di Andrea Frova, «Libri di base» 18, Roma, Editori Riuniti, 1981.

Esempi di organi di uscita sono:

1. la 'stampante', che stampa i risultati del problema su fogli di carta;
2. Il 'video', simile a un comune televisore.

Istruzioni del programma e dati di ingresso vengono comunicati al calcolatore tramite linee di comunicazione, schematizzate nel disegno precedente mediante linee continue.

Accanto agli organi di ingresso e uscita, un calcolatore dispone di altri tre tipi di risorse:

1. una 'memoria', dove sono rappresentate le istruzioni del programma, i dati di ingresso e i risultati parziali del calcolo;
2. una 'unità logico-aritmetica', in cui vengono effettuati i calcoli logici e aritmetici connessi alla esecuzione del programma;
3. un 'organo di governo' che ha lo scopo di interpretare le istruzioni del programma e attivare le risorse coinvolte dalla esecuzione della istruzione.

Si noti che in generale l'esecuzione delle istruzioni può coinvolgere trasferimenti di istruzioni e dati tra una qualunque coppia di blocchi che compaiono nello schema della figura 15. Accanto a tali trasferimenti di informazioni, rappresentati nella figura 15 da linee a tratto continuo, occorrono trasferimenti di comandi dall'organo di ingresso agli altri organi, appunto per predisporli alla esecuzione delle azioni coinvolte dalla generica istruzione. Questo secondo tipo di trasferimenti sono rappresentati con linee tratteggiate.

3. Linguaggi macchina e linguaggi programmativi. In questo paragrafo esaminiamo un po' piú da vicino le caratteristiche dei linguaggi programmativi. La domanda a cui ci interessa trovare risposta è: come può un calcolatore comprendere ed eseguire istruzioni in un linguaggio programmativo?

Istruzioni e dati sono rappresentati in un calcolatore tramite cosiddetti 'codici binari', cioè gruppi di simboli appartenenti a un alfabeto di due sole lettere. Un simile 'alfabeto

binario', che usa solo due lettere, è particolarmente comodo proprio per costruire un linguaggio comprensibile per il calcolatore, in quanto questo codice può essere tradotto in un linguaggio fisico in cui ai due simboli corrispondono, per esempio, alta tensione o bassa tensione. Questo linguaggio fisico può essere facilmente interpretato e manipolato dagli organi del calcolatore.

La memoria è suddivisa in parole (o 'celle'), a ciascuna delle quali è associato un indirizzo e in ciascuna delle quali può essere in genere memorizzata una istruzione o un dato espressi appunto in codice binario.

Mediante i circuiti dell'organo di governo, un calcolatore è in grado di interpretare istruzioni che realizzano funzioni molto semplici. L'insieme di tali istruzioni è detto 'linguaggio macchina'.

Un esempio di istruzione in un ipotetico linguaggio macchina è il seguente:

«Somma il contenuto della cella di indirizzo 1000 al contenuto della cella di indirizzo 1001 e metti il risultato nella cella di indirizzo 1002».

Se associamo alla istruzione di somma il codice di istruzione 5 e per comodità usiamo codici decimali anziché binari, l'istruzione potrà essere rappresentata in memoria con la stringa:

All'atto dell'esecuzione di tale istruzione, l'organo di governo decodifica la istruzione, la riconosce come istruzione di somma, predispone il trasferimento dei due valori nell'unità logico-aritmetica dove avviene la somma e successivamente predispone il trasferimento del risultato in memoria.

Altre istruzioni che sono in genere direttamente interpretabili da un calcolatore sono per esempio:

1. salta a eseguire la istruzione che si trova all'indirizzo X;
2. leggi dall'organo di ingresso un dato e trasferiscilo nella cella di indirizzo X;
3. fermati.

Come si può comprendere, scrivere programmi in linguaggio macchina è noioso e complicato, poiché ogni istruzione svolge una funzione molto elementare. Nei primi calcolatori il linguaggio macchina era l'unico linguaggio con cui fosse possibile comunicare con il calcolatore. Nacque allora l'idea di progettare nuovi linguaggi piú potenti e piú vicini al linguaggio con cui ci si esprime normalmente. Questi linguaggi sono comunemente detti 'linguaggi ad alto livello'.

Siccome però il linguaggio macchina è l'unico che può essere compreso dal calcolatore, bisogna che i programmi scritti in questi linguaggi ad alto livello siano tradotti nel linguaggio macchina.

Il problema può essere risolto elaborando una volta per tutte un programma particolare detto 'traduttore' scritto in linguaggio macchina che, a partire dalla conoscenza della sintassi del linguaggio programmativo, cioè l'insieme delle regole per definire frasi del linguaggio e comporle in programmi, e del significato delle frasi, cioè come ciascuna frase si traduca in termini di insiemi di istruzioni in linguaggio macchina, controlla la correttezza sintattica del programma e, in caso di programma sintatticamente corretto, produce il corrispondente programma in linguaggio macchina.

Perciò, tutte le volte che si intende far eseguire un programma scritto in un linguaggio programmativo ad alto livello occorrerà procedere in due fasi (vedi fig. 16).

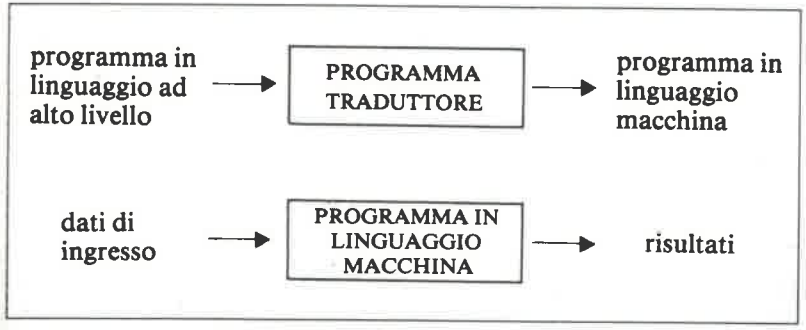


Fig. 16: Fasi del processo di traduzione di un programma.

Oltre ai traduttori, che procedono come visto traducendo tutto il programma in linguaggio macchina allo stesso tempo, è possibile usare 'interpreti', che procedono traducendo ed eseguendo una istruzione per volta.

4. I grafi di flusso come mezzo di descrizione di algoritmi. Nei prossimi capitoli avremo spesso bisogno di mostrare algoritmi e ragionare su di essi. A conclusione di questo capitolo, perciò, vogliamo definire un nuovo linguaggio che non può propriamente definirsi un linguaggio programmativo, semplice da comprendere e diffusamente usato per descrivere algoritmi. Anche noi lo useremo nel seguito.

Il linguaggio che chiamiamo dei 'grafi di flusso' utilizza una rappresentazione grafica per descrivere le istruzioni e l'ordine di esecuzione tra di esse. Prima di introdurre tale rappresentazione assumiamo alcune convenzioni riguardo al calcolatore che prendiamo in considerazione. Assumeremo che il nostro calcolatore sia dotato di:

1. un lettore di schede come organo di ingresso. I dati di ingresso al programma sono perforati uno per scheda;
2. una stampante come organo di uscita. I risultati sono stampati dal calcolatore uno per riga di stampa.

Riportiamo nella tabella 3 i simboli usati nel linguaggio dei grafi di flusso per esprimere le istruzioni, e il significato di tali simboli, cioè l'insieme delle azioni svolte quando nel corso della esecuzione dell'algoritmo si incontra il simbolo.

Nella tabella compaiono i termini 'espressione aritmetica' ed 'espressione logica'. Esempi di espressione aritmetica sono:

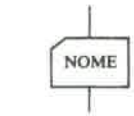
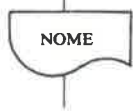
3 + 5 - 2

in cui compaiono i valori costanti (nel seguito le 'costanti') 3, 5, 2 e i segni + e - che sono operatori aritmetici.

SOMMA + NUMERO + 1

in cui accanto alla costante 1 compaiono nomi di variabile. Nel calcolo di questa espressione alle variabili saranno sostituiti i rispettivi valori.

Tabella 3 **Simboli del linguaggio dei grafi di flusso e loro significato.**

simbolo	**significato** *quando s'incontra il simbolo occorre*
	1. Leggere un dato e assegnare il suo valore alla variabile con nome NOME 2. Passare alla scheda successiva
	1. Scrivere il valore della variabile con nome NOME 2. Posizionarsi sulla riga successiva
	1. Calcolare il valore della espressione aritmetica. 2. Assegnare il risultato alla variabile con nome NOME
	1. Calcolare il valore della espressione logica 2. Se la risposta è vera (sí) seguire il ramo destro, altrimenti (no) seguire il ramo sinistro
	Seguire la freccia
	Fermarsi

Le espressioni logiche sono espressioni il cui valore è VERO o FALSO. Per esempio:

3 < 5
(3 è minore di 5)

che ha sempre valore vero, e la

I = 100

il cui valore dipende dal valore della variabile I quando l'espressione viene valutata.

Si noti infine che l'istruzione di stampa nella formulazione della tabella 3 permette di stampare solo valori di variabili.

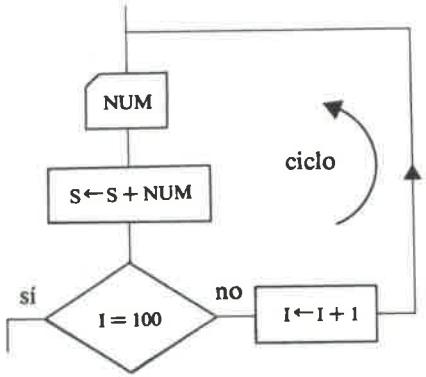
Useremo anche una formulazione del tipo

quando vogliamo stampare una o piú parole, che nella istruzione compaiono racchiuse da apici (in questo caso la parola MESSAGGIO). Un primo esempio di algoritmo scritto nel linguaggio dei grafi di flusso è mostrato nella figura 17.

Vediamo come è organizzato questo algoritmo. Dapprima vengono letti due dati, i cui valori vengono assegnati alle variabili M ed N. Successivamente le due variabili vengono confrontate; se la prima (N) è maggiore della seconda (M) viene stampata, altrimenti viene stampata la seconda. Dunque l'algoritmo legge due numeri e se sono diversi seleziona e stampa il maggiore dei due, se sono uguali stampa uno qualunque dei due.

Come secondo esempio, si veda nella figura 18 un algoritmo per il problema della somma di 100 numeri. Supponiamo che l'algoritmo abbia in ingresso 100 dati rappresentati su 100 schede differenti, uno per scheda.

Si comincia assegnando il valore 1 alla variabile I e il valore 0 alla variabile S. Il nucleo dell'algoritmo è costituito dal ciclo di istruzioni riprodotto nella figura seguente:



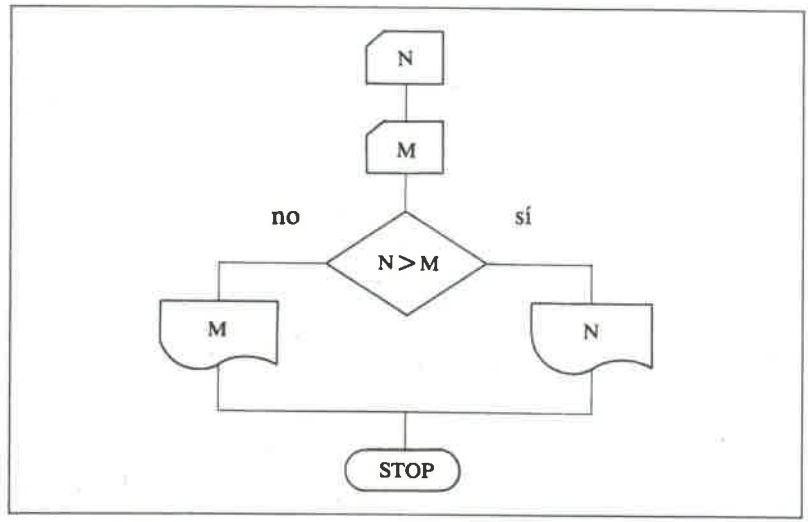


Fig. 17: Primo esempio di algoritmo espresso nel processo dei grafi di flusso.

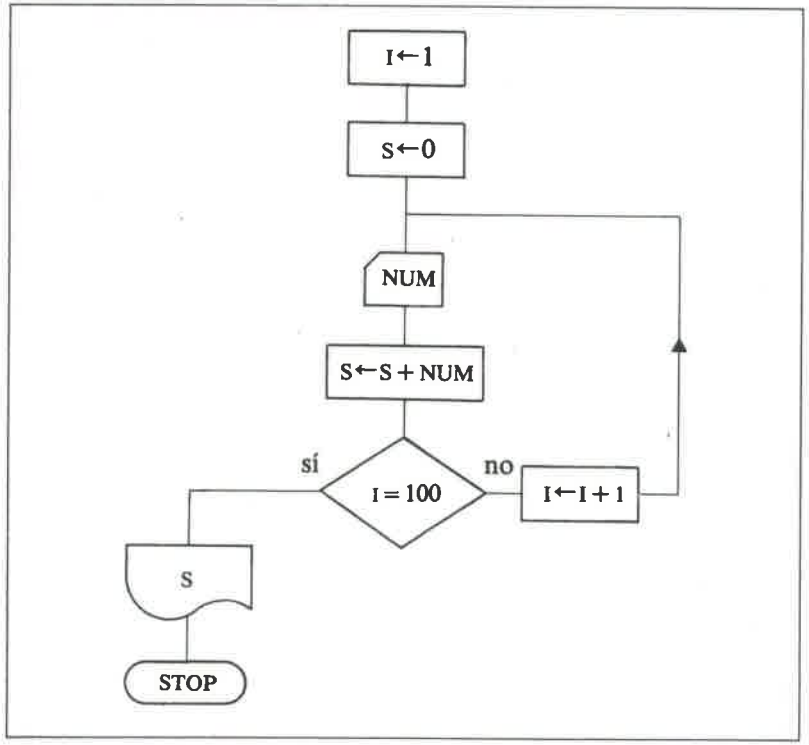
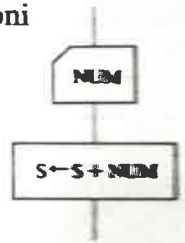


Fig. 18: Algoritmo per la somma di 100 numeri.

Nel ciclo le due istruzioni



vengono eseguite 100 volte, poiché a ogni esecuzione del ciclo il valore della variabile I è aumentato di 1 e il ciclo è abbandonato quando la variabile I assume il valore 100.

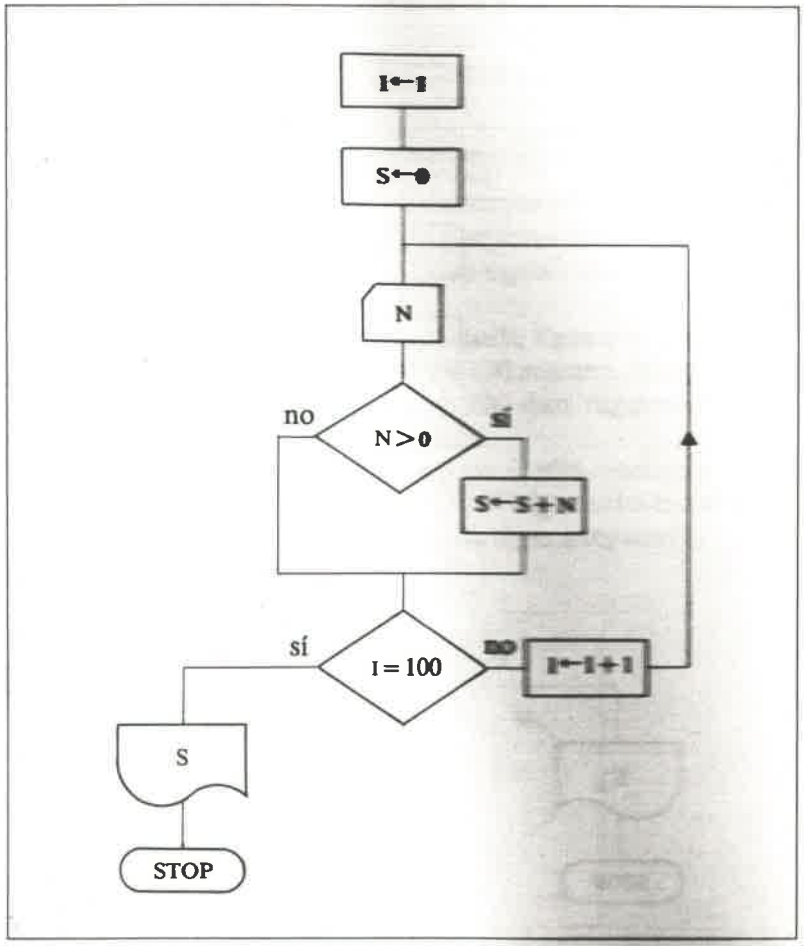


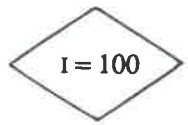
Fig. 19: Algoritmo per la somma dei soli valori positivi.

Per 100 volte dunque viene letto un nuovo valore da una nuova scheda, assegnato alla variabile NUM e subito dopo sommato alla variabile S. Dopo la centesima lettura e somma, come detto, il ciclo viene abbandonato. A questo punto viene stampato il valore di S, cioè la somma dei 100 numeri letti.

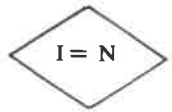
Supponiamo ora di voler modificare il precedente algoritmo in modo tale da sommare solo i numeri positivi tra quelli letti. Otteniamo il programma della figura 19, in cui dopo la lettura del generico valore ci chiediamo se è positivo, e lo sommiamo a S solo in questo caso.

Se invece vogliamo generalizzare l'algoritmo di somma, e costruire l'algoritmo per la somma di n numeri, con n a sua volta letto da scheda, basterà modificare l'algoritmo della figura 19 introducendo all'inizio una istruzione di lettura di n in una variabile, che chiamiamo N:

e sostituire l'istruzione



con l'istruzione



Nei prossimi capitoli faremo diffusamente uso dei grafi di flusso per esprimere algoritmi: talvolta ci distaccheremo dalle rigide regole esposte nella tabella 3, e dentro ai simboli grafici del linguaggio useremo anche frasi in linguaggio italiano per esprimere algoritmi che siamo interessati a descrivere in maniera ancora astratta e non definitiva.

Un'avvertenza importante, prima di concludere. In questo capitolo abbiamo introdotto i concetti di algoritmo e programma come concetti distinti. Nei prossimi capitoli spesso non avremo interesse a distinguere tra i due concetti, che useremo perciò con lo stesso significato. Tenderemo a usare il termine *algoritmo* per mettere in evidenza il fatto che il discorso sui metodi, come abbiamo detto nel capitolo I, può in certa misura fare a meno dell'uso dei calcolatori. Useremo il termine *programma* quando vogliamo fare esplicito riferimento alla descrizione dell'algoritmo espressa in un linguaggio programmativo.

Nel capitolo IV, dedicato alla calcolabilità, saremo interessati a un insieme ristretto di algoritmi (e programmi), richiedendo per essi la proprietà di terminare il calcolo in un tempo finito. Vogliamo subito far notare che questa proprietà non è in generale rispettata nei programmi reali.

Nel capitolo VI la distinzione introdotta in questo capitolo verrà ripresa e motivata. Verrà cioè mostrato come, per semplificare il compito del progettista, possa essere utile in generale distinguere nel progetto due fasi. Nella prima fase si formula il metodo (algoritmo) senza nessun riferimento al linguaggio finale, e nella seconda si procede alla traduzione del metodo in termini del linguaggio (programma).

III. LA CORRETTEZZA

Introduzione – Metodi dei dati di test – Metodi formali

«"Per trovare la via di uscita di un labirinto" recitò infatti Guglielmo, "non vi è che un mezzo. A ogni nodo nuovo, ossia mai visitato prima, il percorso di arrivo sarà contraddistinto con tre segni. Se, a causa di segni precedenti su qualcuno dei cammini del nodo, si vedrà che quel nodo è già stato visitato, si porrà un solo segno sul percorso di arrivo. Se tutti i varchi sono già stati segnati allora bisognerà rifare la strada, tornando indietro. Ma se uno o due varchi del nodo sono ancora senza segni, se ne sceglierà uno qualsiasi, apponendovi due segni. Incamminandosi per un varco che porta un solo segno, ve ne apporremo altri due, in modo che ora quel varco ne porti tre. Tutte le parti del labirinto dovrebbero essere state percorse se, arrivando ad un nodo, non si prenderà mai il varco con tre segni, a meno che nessuno degli altri varchi sia ormai privo di segni".

«"Come lo sapete? Siete esperto di labirinti?"

«"No, recito da un testo antico che una volta ho letto".

«"E secondo questa regola si esce?"

«"Quasi mai che io sappia. Ma tenteremo lo stesso" (da Umberto Eco, *Il nome della rosa*)».

1. Introduzione. Frate Guglielmo, nel libro *Il nome della rosa*, espone una sua teoria su come uscire dal labirinto in cui si è venuto a trovare imprigionato. Appena però il frate che è con lui chiede conferma sulla bontà del metodo, subito le sue certezze vacillano e il metodo che ha in mente diventa un semplice tentativo a cui affidarsi sperando nella buona sorte.

Il metodo esposto da Guglielmo è un insieme ordinato, una sequenza, di istruzioni molto precise, e a buon diritto può essere considerato un algoritmo. Il dubbio espresso nel dialogo riguarda perciò il fatto se l'algoritmo sia effettiva-

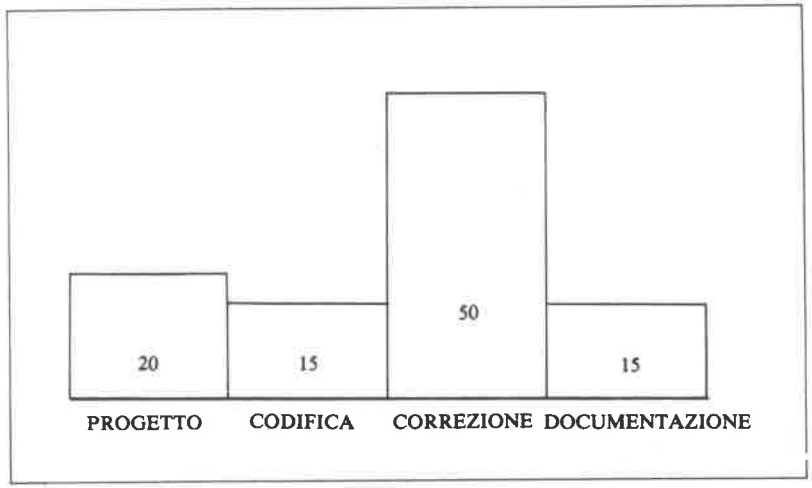


Fig. 20: Distribuzione percentuale dei costi di produzione dei programmi.

mente un algoritmo risolutivo del problema di uscire dal labirinto, o non risolva piuttosto un altro problema, o magari non porti a girare all'infinito attorno a poche stanze, sempre le stesse («Maledizione, siamo tornati al punto di partenza!»).

Dato un problema, trovare un algoritmo risolutivo e descriverlo per mezzo di un programma in un particolare linguaggio programmativo è in genere una attività molto complessa. Gli esempi dati nel capitolo precedente possono dare solo un'idea vaga della quantità di scelte, di valutazioni e di ragionamenti che tale attività comporta.

Sono state fatte statistiche sui tempi medi necessari per eseguire le varie fasi per la realizzazione di un programma. La distribuzione percentuale delle singole fasi è riportata nella figura 20. Accanto alle attività di progetto logico, e codifica, che corrispondono alle attività di progetto dell'algoritmo e di traduzione in un programma di cui abbiamo parlato nel capitolo precedente, abbiamo distinto a parte due aspetti:

1. la 'correzione', attività che ha lo scopo di scoprire gli errori di progetto e appunto correggerli;
2. la 'documentazione' che ha lo scopo di corredare il programma di vari testi esplicativi che spiegano le scelte fatte nel corso del progetto.

Dalla figura emerge chiaramente il grande peso dei costi di correzione: in media, metà del costo (e perciò del tempo) richiesto per produrre un programma è impiegato (o meglio, perso) per correggerlo.

Oggi che i calcolatori hanno sempre maggiori impieghi, capita in pratica ogni giorno che un programma riveli, magari dopo anni e anni di funzionamento corretto, un errore. D'altra parte, i calcolatori stanno ormai sostituendo l'uomo in una miriade di applicazioni in cui l'affidabilità è una caratteristica assolutamente essenziale. Pensiamo ai programmi utilizzati per controllare il funzionamento di processi industriali, di centrali nucleari, di strumenti per la medicina. È noto che in una delle missioni lunari si rivelò a pochi minuti dall'allunaggio un errore nel programma che stava automaticamente controllando la discesa. Un astronauta dovette allora sostituire la macchina effettuando un allunaggio manuale.

Dunque, è molto difficile in generale scrivere algoritmi e programmi per cui si sia ragionevolmente certi che siano corretti. Da qui l'importanza di costruire metodi che cerchino di dare una risposta alle seguenti due domande:

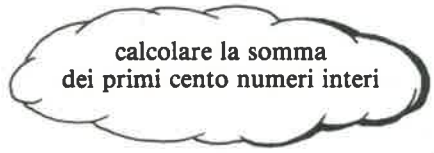
1. dato un problema e un algoritmo, è possibile decidere con certezza che l'algoritmo risolve correttamente il problema?
2. dato un problema e un algoritmo che sappiamo essere scorretto, come si fa a individuare gli errori e correggerli in modo da renderlo corretto?

Nelle pagine che seguono concentreremo l'attenzione sulla prima domanda, e non approfondiremo ulteriormente la seconda.

Anzitutto, è possibile dare una definizione precisa di cosa si intenda per algoritmo corretto?

Come definizione-tentativo, diremo che un algoritmo è corretto se per ogni possibile dato in ingresso fornisce esattamente i risultati previsti dalla descrizione del problema.

Consideriamo per esempio il seguente problema:



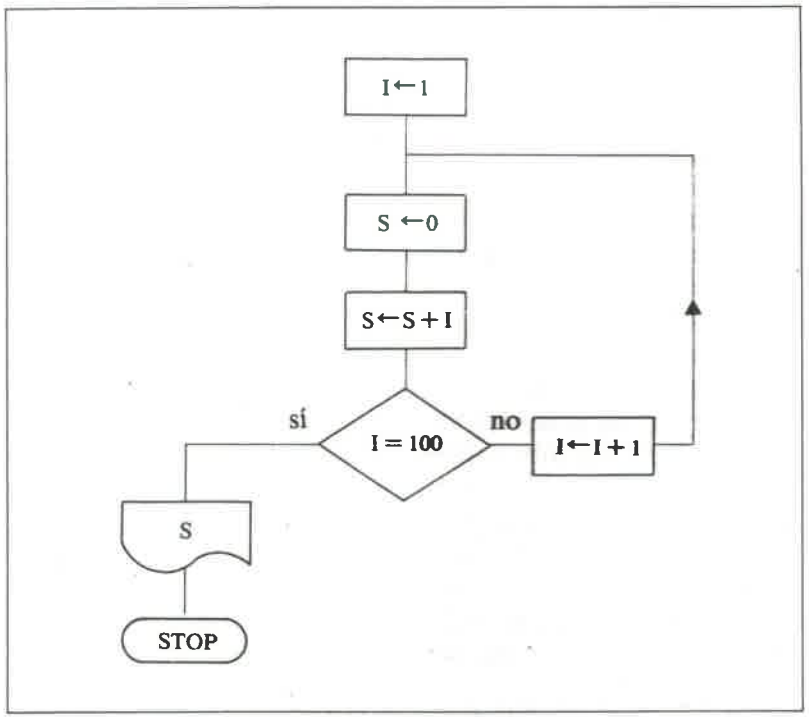


Fig. 21: Algoritmo che dovrebbe fare la somma dei primi 100 numeri interi.

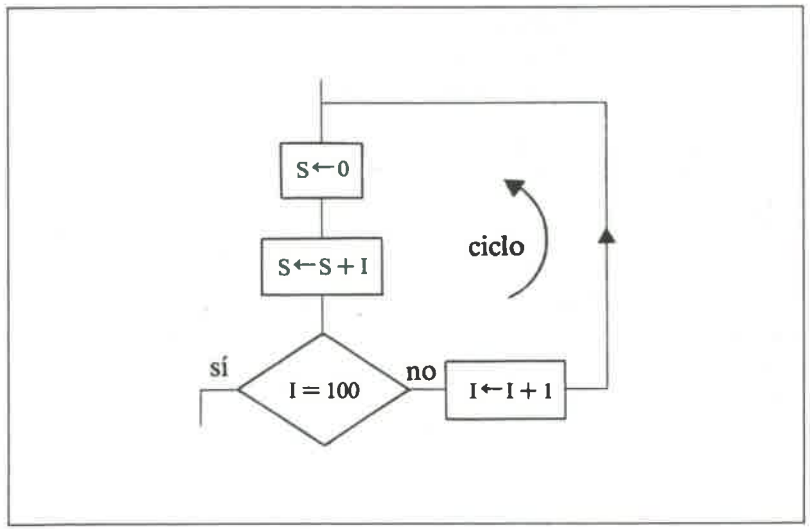


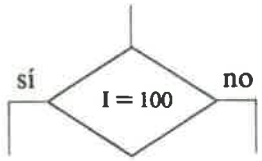
Fig. 22.

e l'algoritmo della figura 21.

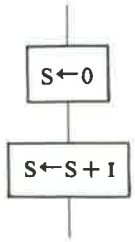
Questo algoritmo, considerato come algoritmo risolutivo del problema descritto sopra, presenta un errore. Quale? Cerchiamo di capirlo osservando bene come è costruito l'algoritmo.

Come si vede il nucleo dell'algoritmo è costituito dal ciclo di istruzioni riprodotto nella figura 22.

La prima volta che è eseguito il ciclo la variabile I vale 1 e ogni volta che il ciclo è eseguito, I è aumentata di 1; il ciclo è abbandonato dopo la sua centesima esecuzione, poiché solo in tale occasione la domanda posta nel blocco



ha risposta Sí. Di conseguenza le due istruzioni



vengono eseguite 100 volte di seguito.

Questo significa che ogni volta che viene eseguita l'istruzione che aggiunge a S il valore I, S ha valore 0. Dunque, al termine della esecuzione dell'algoritmo S vale 100, poiché l'ultimo valore di I viene sommato a S prima di abbandonare il ciclo.

La versione corretta dell'algoritmo è quella riportata nella figura 23, alla pagina seguente, in cui l'istruzione che assegna valore 0 a S è posta fuori dal ciclo.

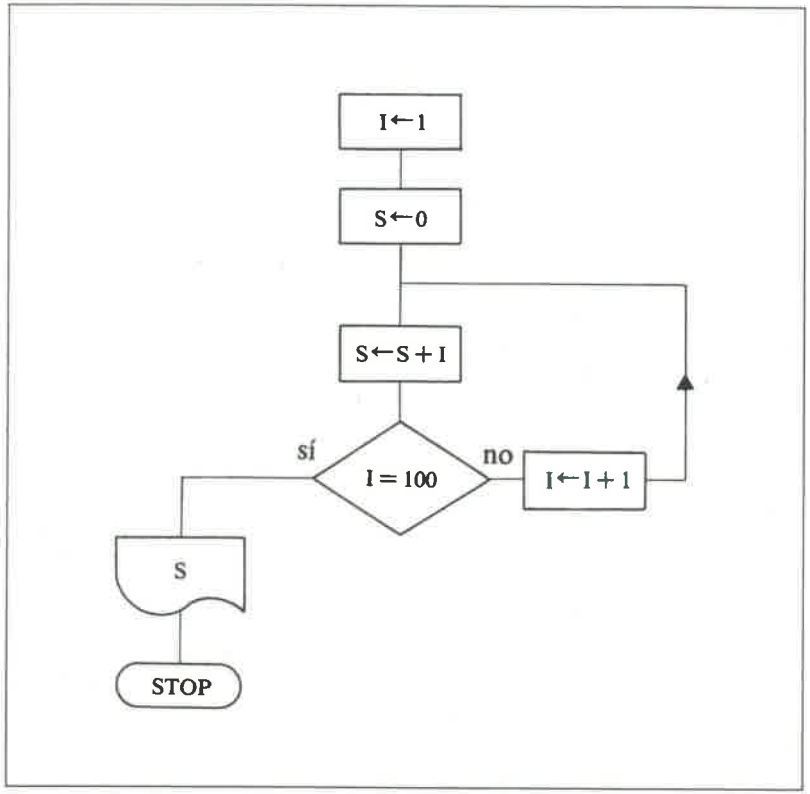


Fig. 23: Versione corretta dell'algoritmo di fig. 21.

Come si vede, anche per provare la correttezza, o la scorrettezza, di un algoritmo molto semplice è necessario fare ragionamenti piuttosto sofisticati.

Vediamo ora i metodi esistenti per tale scopo. I metodi esistenti sono di due tipi:

1. metodi basati sui dati di test;
2. metodi formali.

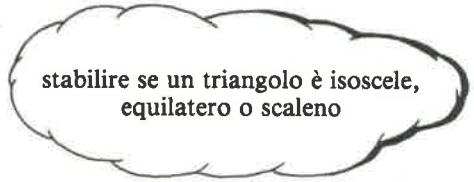
I primi consistono nell'eseguire l'algoritmo e nell'osservarne il comportamento; i secondi provano la correttezza in base a un ragionamento astratto (formale) e non richiedono l'esecuzione dell'algoritmo.

Vediamoli separatamente in dettaglio.

2. Metodi basati sui dati di test. Questi metodi sono da noi utilizzati in tante altre occasioni nella vita di ogni giorno. Quando acquistiamo un televisore ci assicuriamo che funziona accendendolo e spegnendolo, sintonizzandolo su tutti i canali, verificando che l'audio è regolabile, e cosí via; in altre parole per assicurarci che il televisore è realizzato a regola d'arte lo sottoponiamo a una serie di stimoli e verifichiamo che il comportamento sia coerente con il risultato atteso da questi stimoli.

Descriviamo uno di questi metodi tramite un esempio .

Consideriamo il seguente problema:



Si impara dalle elementari, ma è bene ripeterlo: un triangolo equilatero ha tutti i lati della medesima lunghezza, un triangolo isoscele ha due lati della medesima lunghezza e il terzo di lunghezza diversa, un triangolo scaleno ha tutti i lati di lunghezza diversa.

Decidiamo di rappresentare le lunghezze dei lati con numeri interi e, come risposta, di stampare le tre parole *equilatero, isoscele, scaleno*. Le lunghezze dei lati saranno inizialmente lette e memorizzate nelle tre variabili A, B, C.

Un primo algoritmo (avvertiamo subito, scorretto) per risolvere tale problema è riportato nella figura 24, a pagina 50.

L'algoritmo confronta dapprima i lati A e B e successivamente i lati B e C. A seconda degli esiti dei confronti stampa un opportuno messaggio.

Guardando la figura 24 forse il lettore si accorgerà di qualche errore. Lo scopo di un metodo di prova di correttezza è di aiutare il progettista a scoprire sistematicamente tutti gli errori di un algoritmo.

Possiamo decidere di adottare la seguente strategia nella scelta dei dati di test: eseguire l'algoritmo per un insieme di dati che permettono di ottenere tutte le risposte possibili del problema.

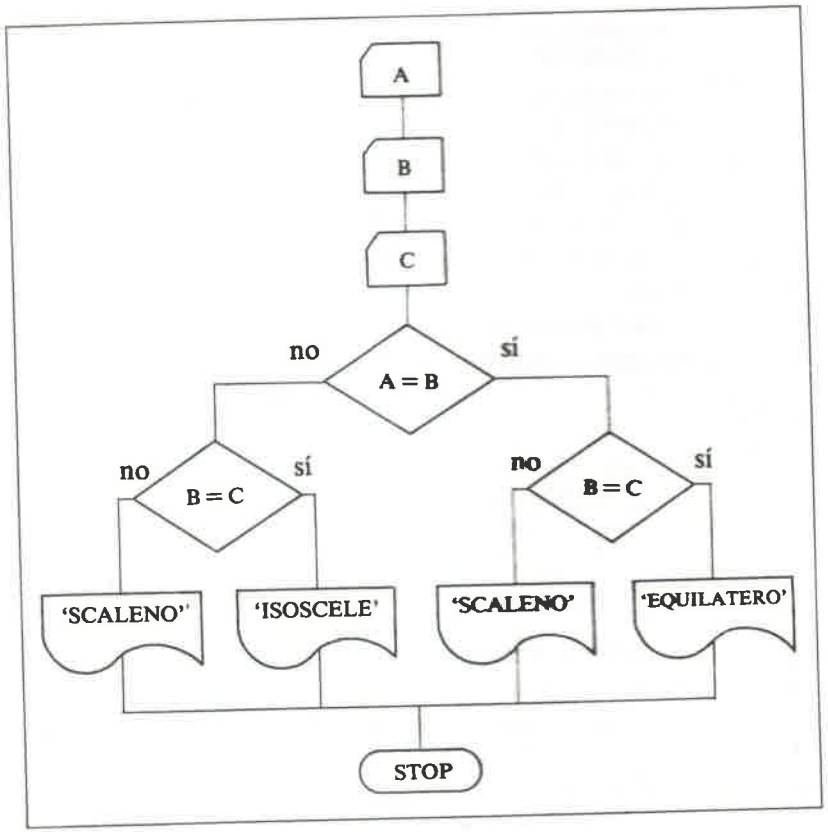


Fig. 24: Algoritmo (scorretto) per l'analisi di un triangolo.

La strategia vede l'algoritmo come una scatola nera (vedi fig. 25) che deve produrre a fronte di certi dati in ingresso, certi risultati, e non si interessa del modo in cui l'algoritmo effettua tale trasformazione.

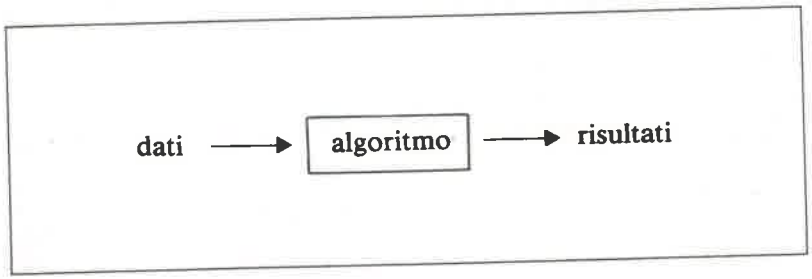


Fig. 25.

Un insieme di dati che rispetta questo criterio è il seguente:

	A	B	C	risposta desiderata
1° insieme	2	2	2	EQUILATERO
2° insieme	2	3	3	ISOSCELE
3° insieme	2	3	4	SCALENO

Si noti che l'algoritmo fornisce risposte corrette per tutti e tre gli insiemi di dati.

Al contrario, se osserviamo l'algoritmo, risulta evidente che la risposta SCALENO nella parte destra del diagramma è sbagliata, perché giunge quando si è verificato che A = B.

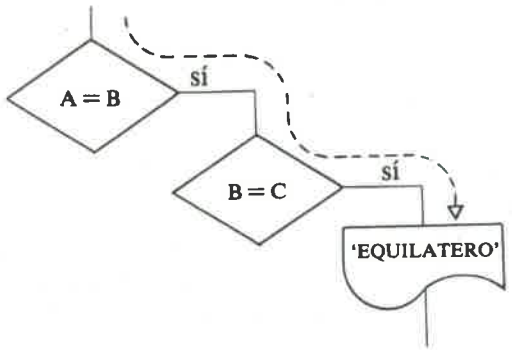
La debolezza del criterio sta proprio nel fatto che l'insieme di dati di test viene individuato senza tener conto di come l'algoritmo funziona.

Consideriamo ora un secondo criterio basato sulla seguente strategia: eseguire l'algoritmo su un insieme di dati che permettono di percorrere tutti i rami del diagramma.

Cerchiamo un insieme di dati di test per questo criterio.

È chiaro, per come è fatto il diagramma, che per rispettare il criterio basterà individuare un insieme di dati che permette di eseguire tutte le possibili istruzioni di stampa.

Partiamo dal caso della unica risposta EQUILATERO. Per produrre la risposta EQUILATERO, i dati dovranno essere tali da far eseguire il seguente cammino:



Un insieme di dati che ha questa proprietà è il seguente:

A = 2
B = 2
C = 2

In questo caso l'algoritmo fornisce risposta corretta.
Un ragionamento analogo si può fare per il caso ISOSCELE. Per il caso SCALENO ci sono due percorsi che possono essere seguiti (vedi fig. 26).

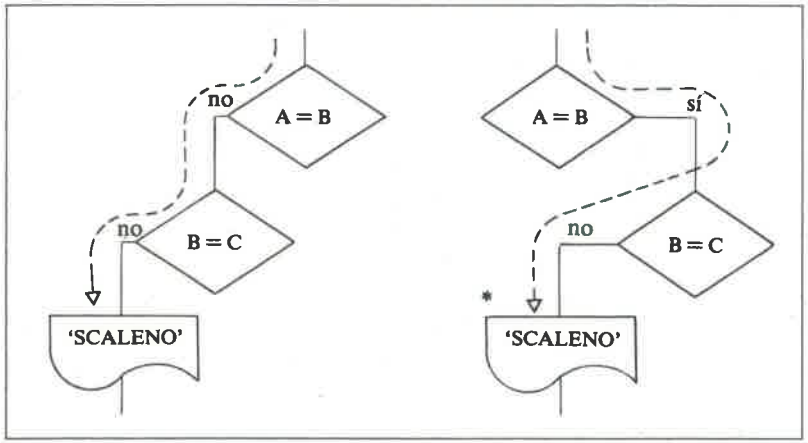


Fig. 26: Cammini da percorrere per fornire la risposta SCALENO.

Un insieme di dati che rispetta il criterio è dunque il seguente:

	A	B	C
1° insieme	2	2	2
2° insieme	2	3	2
3° insieme	2	3	4
4° insieme	3	3	2

È chiaro che, quando l'algoritmo è eseguito sui dati 3, 3, 2 fornisce, facendo un errore, la risposta SCALENO. L'algoritmo è in effetti *sempre* scorretto quando viene prodotta quella risposta, poiché quel punto del programma è raggiunto quando si è verificato che A è uguale a B e B è diverso da C, e dunque il triangolo è isoscele.

Il nuovo criterio ci ha permesso di scoprire un errore, in virtú del fatto che da una parte provoca prima o poi la esecuzione di un qualunque cammino dall'inizio alla istruzione di STOP e dall'altra si è nel caso fortunato di un programma che, quando termina dopo aver eseguito l'istruzione indicata con *, fornisce sempre risposta scorretta.

Nella figura 27 è riportata una nuova versione dell'algoritmo in cui l'errore è stato corretto.

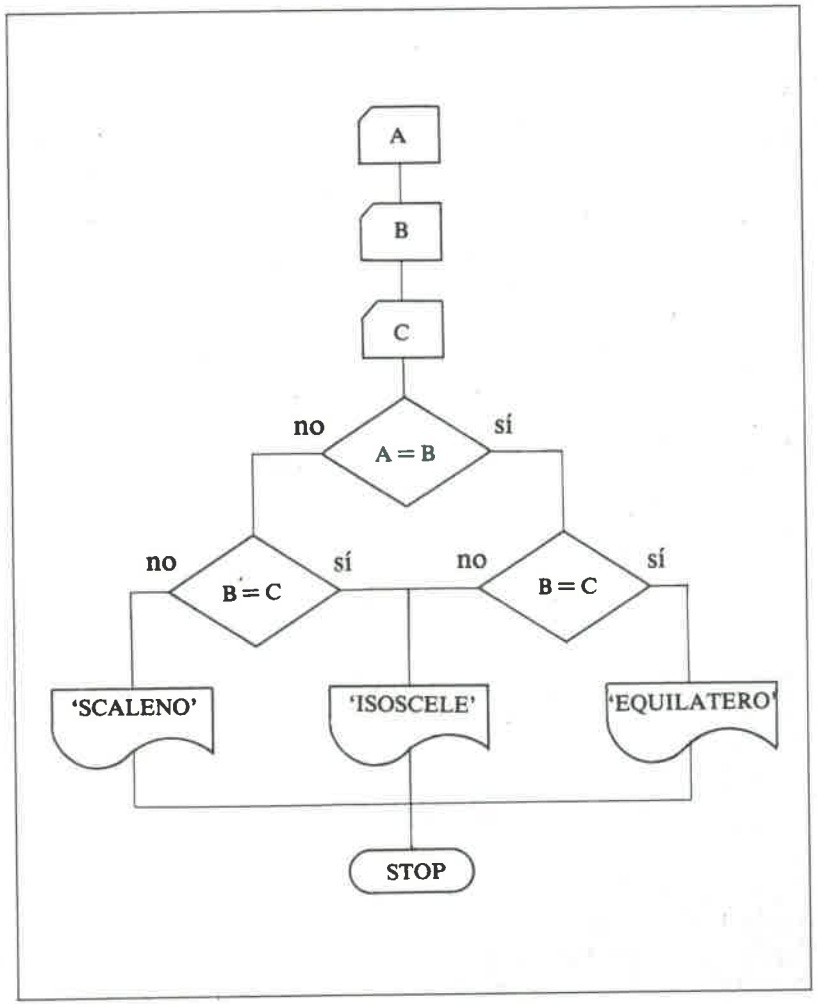


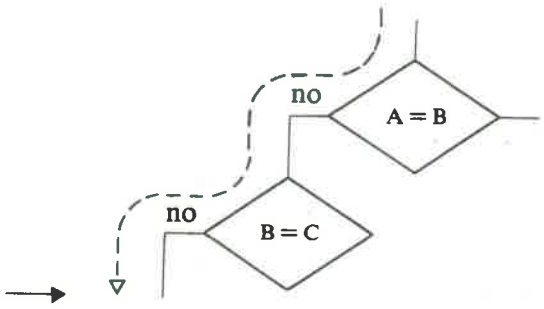
Fig. 27.

Quanto è affidabile il criterio precedente? Che garanzia ci dà sulla effettiva correttezza dell'algoritmo? Purtroppo, nessuna. Basta per esempio osservare che l'algoritmo, sottoposto ai seguenti dati

A = 2
B = 3
C = 2

fornisce scorrettamente risposta SCALENO.

Nel caso specifico il fallimento del test è dovuto alla incoerenza tra tipo di errore commesso nell'algoritmo e dati usati nel test. L'errore commesso nell'algoritmo consiste nello stampare a colpo sicuro il messaggio SCALENO per tutti i dati che dànno luogo al percorso:



Quando si sia arrivati al ramo indicato dalla freccia si può solo affermare che:

1. A e B sono diversi.
2. B e C sono diversi.

Questi due fatti non permettono di dedurre che anche A e C sono diversi: arrivati cioè a quel punto dell'algoritmo noi non sappiamo nulla sulla relazione che sussiste tra A e C! Se invece di scegliere la terna di dati 2, 3, 3 avessimo scelto i dati 3, 2, 3, ci saremmo accorti dell'errore.

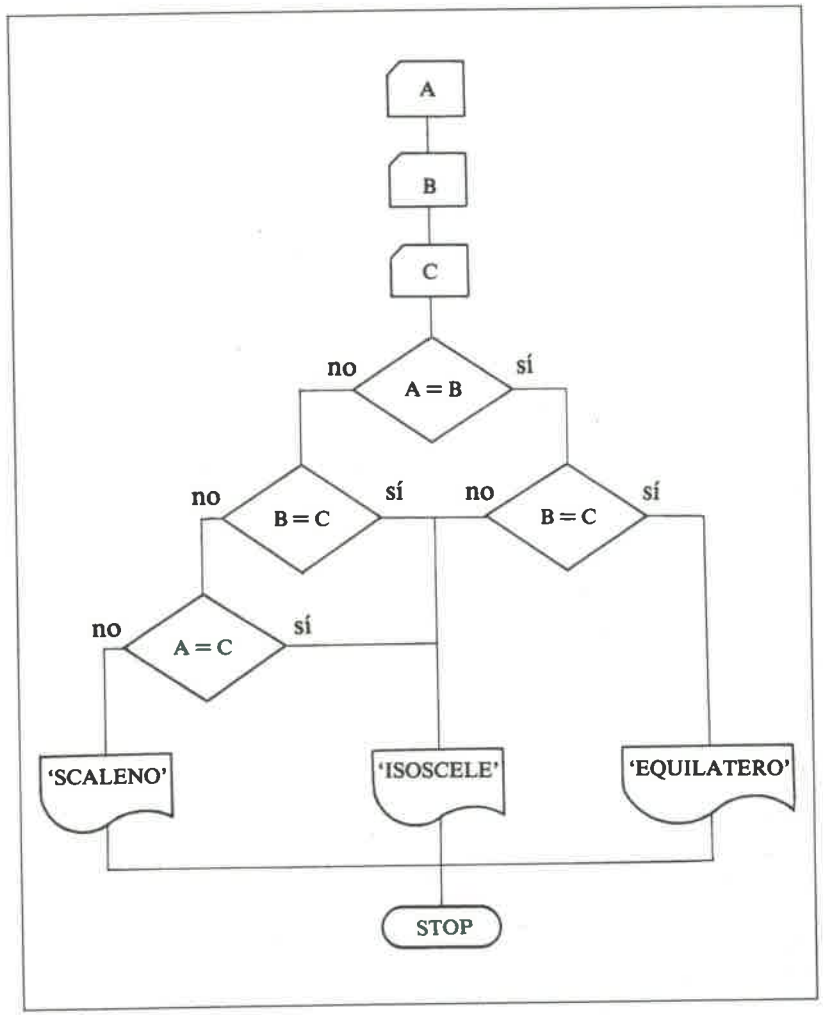


Fig. 28.

Riportiamo nella figura 28 un nuovo algoritmo in cui è stato corretto anche il nuovo errore.

Apparentemente il nuovo algoritmo è corretto. A ben vedere, però, l'algoritmo ha ancora un comportamento scorretto in particolari situazioni, che fino a ora abbiamo trascurato. Se per esempio sottoponiamo all'algoritmo i dati -5, -5, -5, esso fornisce come risposta EQUILATERO, mentre è chiaro che un triangolo con lati -5, -5, -5 non esiste.

Possiamo tener conto di queste situazioni facendo in modo che l'algoritmo, appena letti i dati, verifichi che non abbiano valore negativo. Ma questo non basta: anche un triangolo con lati 2, 2, 1789 non esiste. Per tenere conto di questo altro tipo di situazioni bisognerà verificare le due proprietà tipiche dei lati di un triangolo:

1. la somma delle lunghezze di ogni coppia di lati è sempre maggiore della lunghezza del terzo;
2. la differenza delle lunghezze di ogni coppia di lati è sempre minore della lunghezza del terzo.

Queste ultime osservazioni mostrano che un algoritmo corretto non deve solo dare risposta corretta per i dati di ingresso corretti, ma deve anche segnalare eventuali dati scorretti. Questo aspetto della correttezza non era messo in evidenza nella definizione che abbiamo dato in precedenza.

I criteri che abbiamo mostrato sono tra i piú semplici esistenti per analizzare la correttezza dei programmi mediante uso dei dati di test. Comune a tutti i criteri c'è la stessa filosofia: cercare di ridurre a un numero ragionevole il numero dei test da effettuare in un algoritmo. Abbiamo già visto per i criteri mostrati che non si può mai essere sicuri che i dati per cui è stato effettuato il test siano effettivamente rappresentativi di tutti i dati su cui funziona l'algoritmo.

E, d'altra parte, provare l'algoritmo per tutti i casi possibili, anche se espresso nella sua versione in un linguaggio programmativo, è impresa disperata. Pensiamo al caso semplice di un programma per la somma di due numeri che possono assumere valori compresi tra 0 e 1.000.000. Il numero di esecuzioni richieste per provare il programma in tutti i casi possibili è

$$1.000.000 \times 1.000.000 = 1.000.000.000.000.$$

Pensando a un tempo di 5'' per predisporre il programma ed eseguirlo, una persona che lavorasse giorno e notte (e che avesse vita sufficiente) impiegherebbe 2 milioni di anni a eseguire il test.

Dunque i metodi basati sui dati di test non possono mai

provare che un algoritmo (o programma) è corretto, semmai possono provare la sua scorrettezza! Insomma, essi possono certamente rivelare errori, mentre non possono darci la sicurezza della correttezza.

3. Metodi formali. I metodi formali forniscono una prova logica che l'algoritmo è corretto, in base a ragionamenti e deduzioni effettuati sul testo dell'algoritmo.

Questi metodi possono perciò considerarsi come una applicazione della logica all'informatica. La logica è appunto la disciplina che studia i modi in cui è possibile formalizzare i ragionamenti corretti.

Nel seguito mostreremo un esempio di prova formale di correttezza, riprendendo l'algoritmo della somma dei primi 100 numeri (vedi fig. 29).

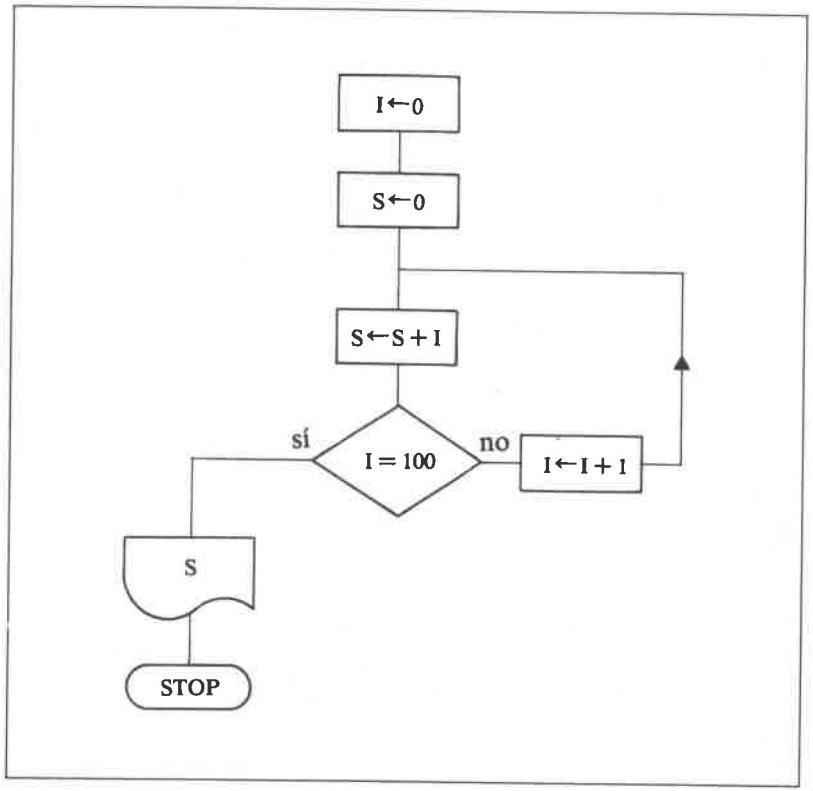


Fig. 29: Algoritmo per la somma dei primi 100 numeri.

Il metodo che seguiremo individua anzitutto la cosiddetta 'asserzione' che deve essere valida alla fine dell'algoritmo, cioè la relazione tra le variabili che esprime in modo formale il legame che esiste tra i dati di ingresso e i risultati.

Una tale asserzione è in questo caso evidentemente:

$$S = \sum_{1}^{100} j$$

La precedente scrittura utilizza il simbolo Σ detto 'di sommatoria', e dice che S deve essere uguale alla somma di un certo numero di valori j, ove j può assumere tutti i valori da 1 a 100.

Riferiamo l'asserzione al punto dell'algoritmo dove deve essere valida, cioè alla fine (vedi fig. 30).

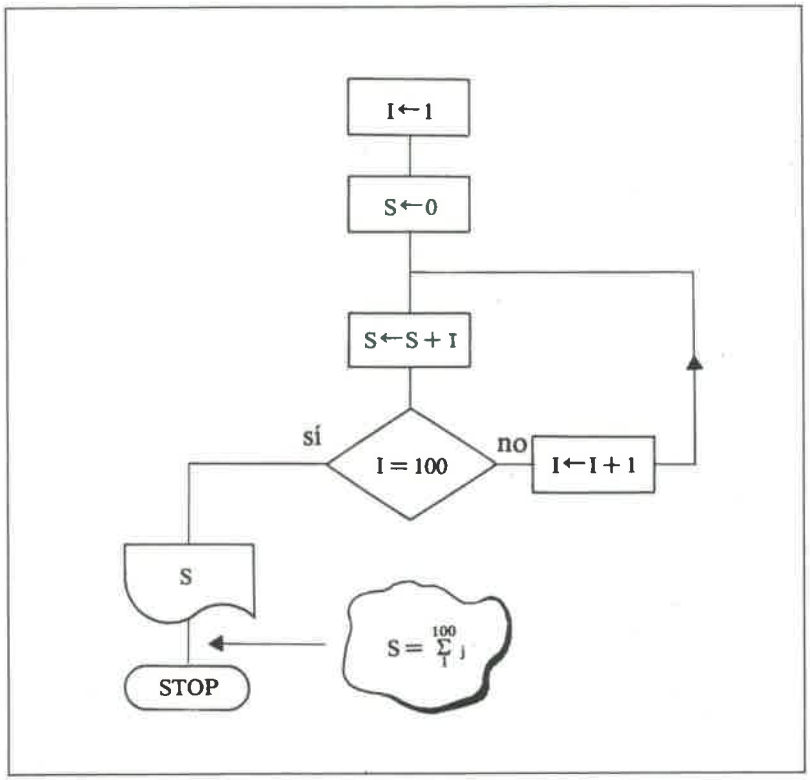
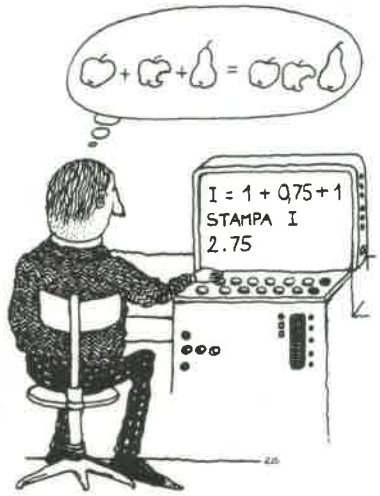


Fig. 30: Algoritmo di fig. 29 con la asserzione di uscita.



Come possiamo provare da un punto di vista formale che, per tutte le possibili esecuzioni dell'algoritmo, è sempre vera la asserzione di uscita? Questo può essere fatto spezzando la prova in una serie di passi intermedi. Possiamo cioè individuare una serie di asserzioni da associare a punti intermedi dell'algoritmo e provare, per ogni asserzione, che la validità di una asserzione in un punto implica la validità della asserzione nel punto successivo. Questa tecnica di prova è tipica nella logica, ed è chiamata 'tecnica di deduzione'. Essa arriva alla prova dell'asserzione finale attraverso una serie di passi in ognuno dei quali utilizziamo le conoscenze a nostra disposizione (nel nostro caso le asserzioni già provate valide e la istruzione da eseguire) per dedurre, appunto, nuova conoscenza. Un classico esempio di ragionamento deduttivo è:

1. Socrate è un uomo
2. ogni uomo è mortale

dunque

3. Socrate è mortale.

È uno dei piú famosi sillogismi del filosofo greco Aristotele (384-322 a.C.).

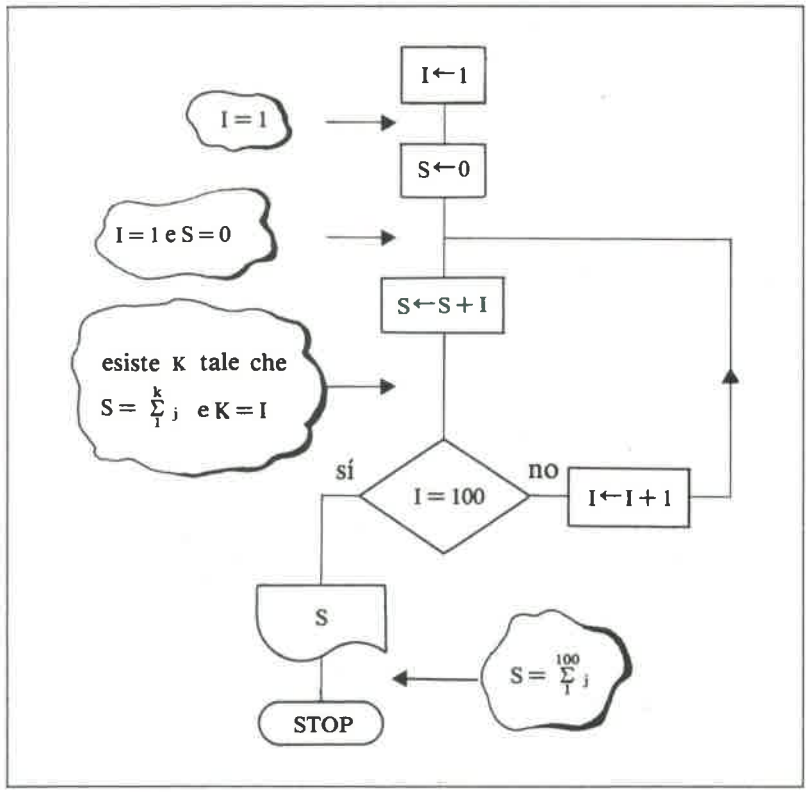
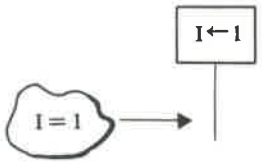


Fig. 31: Asserzioni utilizzate nella prova di correttezza dell'algoritmo della somma dei primi 100 numeri.

Nella figura 31 è riportato un insieme di asserzioni per l'algoritmo della somma.

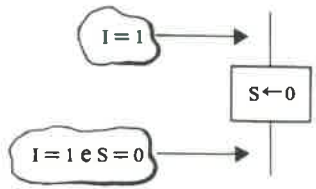
Proviamo ora la validità di ciascuna delle asserzioni.

Passo 1



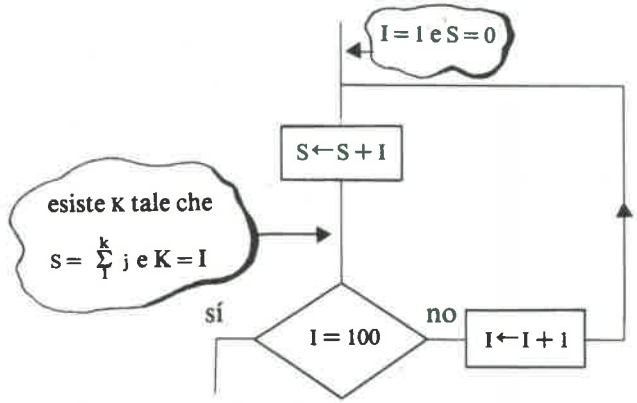
È chiaro che dopo l'esecuzione della istruzione è sempre vero che I è uguale a 1.

Passo 2



Se prima della esecuzione della istruzione era vero che I = 1 e la istruzione assegna il valore 0 a S, è chiaro che dopo la esecuzione della istruzione sarà sempre vero che I = 1 e S = 0.

Passo 3



È il passo piú difficile perché riguarda il ciclo che, al contrario delle altre istruzioni, viene eseguito piú volte.

Come si fa a provare che l'asserzione nel ciclo è valida ogni volta che l'esecuzione passa per quel punto? La tecnica di prova per deduzione non ci aiuta piú: essa funziona quando vi è un insieme finito di possibili storie di esecuzione delle istruzioni che mette in relazione due asserzioni, perché allora con un insieme finito di deduzioni possiamo provare la validità della nuova asserzione. Ma, una volta entrati nel ciclo, è possibile raggiungere la nuova asserzione dopo una, due, tre, un numero qualunque di esecuzioni della istruzione S←S+I.

Possiamo allora usare un'altra tecnica anch'essa tipica nella logica, che chiamiamo 'induzione'. Mostriamo la tecnica di induzione per provare un'asserzione che utilizzeremo nel capitolo V. L'asserzione è:

la somma dei primi n numeri interi è il numero $\frac{n \times (n+1)}{2}$ (cioè $n^2 + n$)

Tra i numeri interi è definito un ordinamento totale, cioè per ogni coppia di numeri n ed m diversi tra loro o è vero n ⟨m o è vero m ⟨ n, ma non tutti e due. Possiamo allora dimostrare l'asserzione in due passi:

1. dapprima proviamo che è vera per n = 0 (ed è facile convincersene);
2. successivamente proviamo che se è vera per un certo numero n, è vera anche per il numero successivo n + 1.

Questa seconda prova richiede qualche ricordo (o nuova nozione) di algebra (inessenziale, peraltro, per la comprensione del resto del libro). La somma dei primi n + 1 numeri può essere scritta come

$$\frac{n \times (n+1) + (n+1)}{2}$$

cioè la somma dei primi n più (n + 1). La precedente espressione può essere riscritta come

$$\frac{(n^2 + n + 2n + 2)}{2}$$

o ancora, ricordando che $(n+1)^2 = n^2 + 2n + 1$,

come

$$\frac{[(n+1)^2 + n + 1]}{2}$$

cioè proprio ciò che volevamo dimostrare.

Nei ragionamenti induttivi si sfrutta la presenza di un ordinamento presente nell'insieme delle affermazioni da provare. Nel nostro caso, ricordiamo, occorre provare che l'asserzione nel ciclo è valida ogni volta che l'esecuzione passa per quel punto. È intuitivo, allora, che l'ordinamento cercato sia basato sul numero delle volte che l'esecuzione dell'algoritmo passa per quel punto.

Applichiamo allora il ragionamento induttivo:

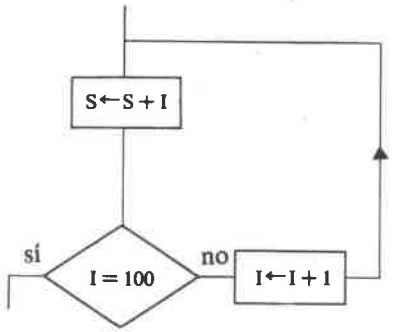
— *la prima volta*

Il valore k richiesto dall'asserzione è in questo caso 1. Infatti per k = 1 $S = \sum_{j=1}^{k} j$ j = 1, cioè il valore appunto calcolato in S la prima volta che è eseguita la istruzione:

— *la generica volta*

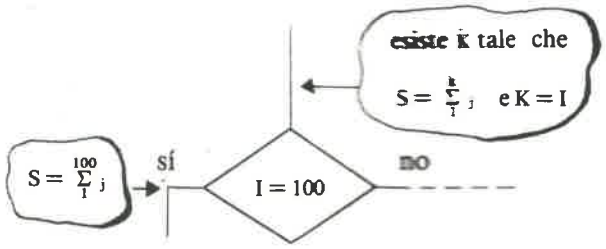
Supponiamo che l'asserzione sia valida per un certo valore di k, per esempio 10. Questo significa che S = 1 + 2 + 3 + ... + 10, ove 10 è l'ultimo valore appena aggiunto ad S ed è anche l'attuale valore di I.

Supponiamo ora che il ciclo di istruzioni



venga eseguito nuovamente, come nel nostro caso. Dopo la nuova esecuzione del ciclo di istruzioni ad S è stato aggiunto il valore 11 (nuovo valore di I) e dunque l'asserzione è ancora vera con k = 11.

Passo 4



È chiaro che quando I è uguale a 100 è ancora vera l'asserzione prima del test nel caso particolare di k = 100. Questo caso particolare è proprio l'asserzione finale.

A questo punto, poiché l'istruzione di stampa non modifica i valori delle variabili, è valida anche l'asserzione finale.

Si noti che il metodo esposto non assicura che l'algoritmo termini, ma solo che, se termina, produce risultati legati ai dati di ingresso da quella data asserzione. Esistono altri metodi per la prova di terminazione che qui non esaminiamo. Possiamo definire la proprietà di 'terminazione' come quella in base alla quale siamo certi che l'esecuzione dell'algoritmo termina in un tempo finito.

Se riesaminiamo la prova, bisogna notare che per dimostrare che l'algoritmo è corretto ci è occorsa molta fatica. Chi ci assicura che oltre che sbagliarci nell'algoritmo non ci possiamo sbagliare anche nel provare che è corretto, e dunque, attraverso due sbagli, provare che è corretto un algoritmo scorretto?

Esaminando con maggior dettaglio il problema della prova formale dei programmi, nei metodi che abbiamo visto sono coinvolti tre aspetti:

1. l'algoritmo;
2. le asserzioni sull'algoritmo;
3. la prova di correttezza.

Per quanto riguarda la prova di correttezza possiamo sperare di costruire un programma che sappia sostituirci nel fare prove di correttezza e delegare a tale programma la prova. Naturalmente di tale programma occorrerà fornire la prova di correttezza!

Per quanto riguarda algoritmo e asserzioni, non può esistere un metodo che permetta di controllare che non ci si è sbagliati due volte. In altre parole, se io voglio scrivere un algoritmo per la somma di 100 numeri, poi scrivo un algoritmo che fa il prodotto di 100 numeri e scelgo delle asserzioni coerenti con il nuovo algoritmo, se genero una prova corretta arriverò alla conclusione che l'algoritmo fa correttamente la somma! È chiaro che la probabilità di sbagliarsi due volte, e tutte e due le volte allo stesso modo, è molto ridotta.

In conclusione, se è vero che i metodi formali non possono dare la certezza della correttezza è anche vero che il tipo di incertezza che lasciano è puramente teorica. Inoltre, l'elaborazione delle asserzioni richiede un notevole sforzo intellettuale; richiede cioè una comprensione profonda del significato dei singoli passi del calcolo che via via vengono composti in strutture di calcolo sempre piú complesse. Di conseguenza, possiamo ben dire parlando, invece che di incertezza, di affidabilità del metodo, che la affidabilità dei metodi formali è di gran lunga superiore a quella dei metodi basati sui dati di test.

Possono pensare le macchine?

Prima di affrontare il problema della calcolabilità, soffermiamoci un attimo sulla domanda: «Possono pensare le macchine?». L'argomento è arduo, e non possiamo approfondirlo. Piuttosto ci sembra utile riportare alcune parti di un articolo di Alan Turing, «Macchine calcolatrici e intelligenza», riportato nel libro a cura di V. Somenzi, *La filosofia degli automi* (Boringhieri).

Il gioco dell'imitazione

Mi propongo di considerare la questione: «Possono pensare le macchine?». Si dovrebbe cominciare col definire il significato dei termini macchina e pensare [...]. Invece di tentare una definizione [...] sostituirò la domanda con un'altra, che è strettamente analoga e che è espressa in termini non troppo ambigui.

Ciò può essere descritto nei termini di un gioco, che chiameremo «il gioco dell'imitazione». Questo viene giocato da tre persone, un uomo (A), una donna (B) e l'interrogante (C), che può essere dell'uno o dell'altro sesso.

Scopo del gioco per l'interrogante è quello di determinare quale delle altre due persone sia l'uomo e quale la donna. Egli le conosce con le etichette X e Y, ed alla fine del gioco darà la soluzione «X è A e Y è B» o la soluzione «X è B e Y è A». L'interrogante può far domande di questo tipo ad A e B:

C: «Vuol dirmi X, per favore, la lunghezza dei propri capelli?»

Ora supponiamo che X sia in effetti A, quindi A deve rispondere. Scopo di A nel gioco è quello di ingannare C e far sí che fornisca una identificazione errata. La sua risposta potrebbe perciò essere:

«I miei capelli sono tagliati "a la garçonne" ed i piú lunghi sono di circa venticinque centimetri».

Le risposte, in modo che il tono di voce non possa aiutare l'interrogante, dovrebbero essere scritte, o, meglio ancora, battute a macchina. La soluzione migliore sarebbe quella di avere una telescrivente che mettesse in comunicazione le due stanze. Oppure le domande e risposte potrebbero essere ripetute da un intermediario.

Scopo del gioco, per il terzo giocatore (B), è quello di aiutare l'interrogante. La migliore strategia per lei è quella di dare risposte veritiere. Essa può anche aggiungere alle sue risposte frasi come «Sono io la donna, non dargli ascolto», ma ciò non approderà a nulla dato che anche l'uomo può fare affermazioni analoghe.

Poniamo ora la domanda «Che cosa accadrà se una macchina prenderà il posto di A nel gioco?». L'interrogante darà una risposta errata altrettanto spesso di quando il gioco viene giocato tra un uomo e una donna?

Queste domande sostituiscono quella originale: «Possono pensare le macchine?» [...].

«Credo che entro circa 50 anni sarà possibile programmare calcolatori con una capacità di memorizzazione di circa 10^9, per fare giocare loro il gioco dell'imitazione cosí bene che un esaminatore non avrà piú del 70% di probabilità di compiere l'identificazione esatta dopo 5 minuti di interrogazione».

IV. CALCOLABILITÀ E NON CALCOLABILITÀ

Introduzione - Le macchine di Turing - Le macchine a programma - Equivalenza di modelli di calcolo e tesi di Church

1. Introduzione. Ogni giorno vengono eseguiti in tutto il mondo centinaia di migliaia di programmi che realizzano le piú svariate applicazioni. Accade piú spesso di quanto si creda che un programma, come si suol dire, «entri in ciclo», cominci cioè a un certo punto della esecuzione a eseguire ciclicamente sempre le stesse istruzioni proseguendo, almeno in teoria, all'infinito. Naturalmente questo accade non perché colui che ha scritto il programma abbia inserito tale ciclo di istruzioni, ma piuttosto perché il progettista ha commesso uno o piú errori che hanno dato luogo a una versione sbagliata del programma, in cui appunto si innesca un ciclo infinito. Cerchiamo di capire meglio come questo può accadere.

Il programma della figura 32, che possiamo considerare una versione scorretta del programma di somma di cento numeri, ha un ciclo infinito.

Mentre infatti nel programma della figura 18 il fatto che l'algoritmo esca dal ciclo è assicurato dal continuo incremento della variabile I, in questo caso la variabile I non viene mai modificata, e dunque il ciclo viene eseguito all'infinito. Nel programma della figura 32 l'entrata nel ciclo infinito ha una ragione strutturale e non dipende dai dati letti; il programma, insomma, non termina per nessuna esecuzione, indipendentemente dai dati letti. È chiaro che in generale il fatto che il programma entri in un ciclo potrà dipendere dai dati di ingresso.

Le ragioni per cui un programma entra in ciclo possono essere le piú varie. Di conseguenza, la probabilità che si verifichi una tale circostanza soprattutto in programmi scritti da persone alle prime esperienze di programmazione è molto piú alta di quanto si possa credere.

Potrebbe perciò far comodo disporre di un algoritmo che chiameremo CONTROLLO-CICLI il cui scopo sia quello di verificare se un programma con certi dati di ingresso entri prima o poi in ciclo o no. Per essere piú precisi:

1. detta P la descrizione di un programma espressa dal suo testo in un linguaggio programmativo;
2. detta D la descrizione di un particolare insieme di dati.

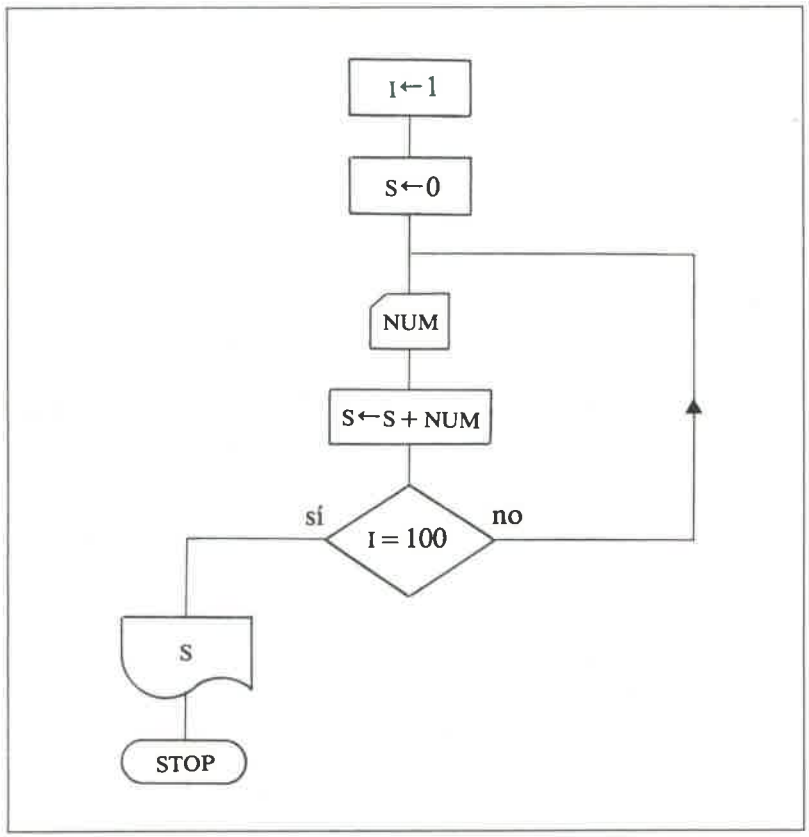


Fig. 32: Programma con ciclo infinito (vedi fig. 18).

L'algoritmo CONTROLLO-CICLI dovrebbe eseguire il calcolo della figura 33.

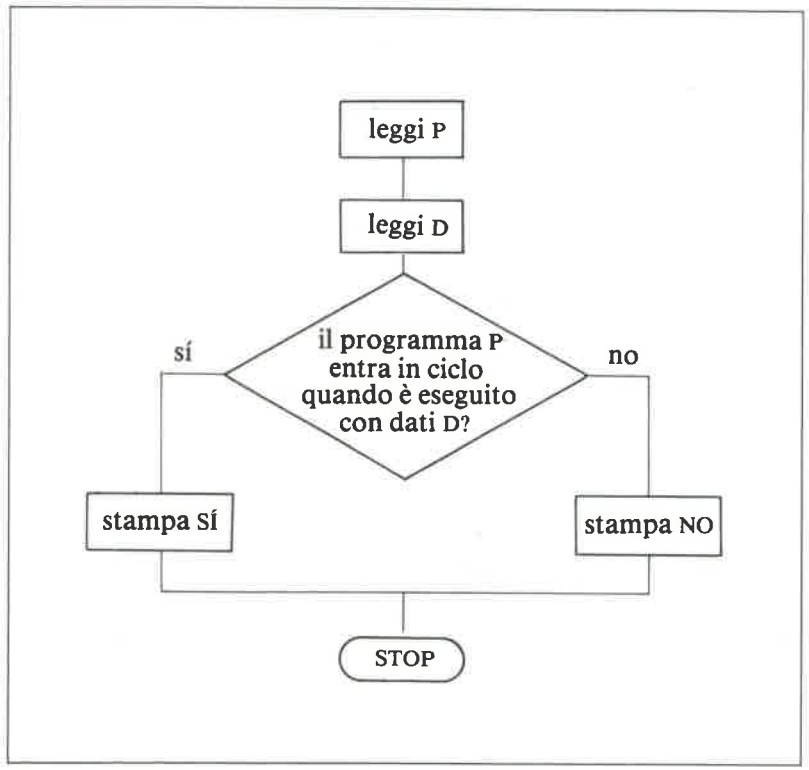


Fig. 33: Struttura dell'algoritmo CONTROLLO-CICLI.

Ebbene, un importante risultato della ricerca in informatica afferma che non esiste un algoritmo che effettui tale calcolo. Detto in altre parole, il risultato afferma che non possiamo sperare che una macchina possa mai aiutarci nel risolvere tale tipo di problema. Questo tipo di calcolo, insomma, non è solo un limite delle attuali macchine, per la semplice ragione che ancora non è stato trovato un algoritmo risolutivo di quel problema. Piú pessimisticamente, nessuna macchina potrà risolverlo, perché un tale algoritmo non esiste.

Per intendere in maniera precisa il significato delle precedenti affermazioni occorre fare diverse considerazioni, che svolgiamo nel seguito.

Anzitutto, per poter indagare sui limiti dei processi di calcolo, e di conseguenza sulle macchine che automatizzano tali processi di calcolo, occorre precisare il concetto stesso di calcolabile[3]. Cosa si intende per calcolare? Si può dare una definizione del concetto di calcolabile che sia operativa, ci permetta cioè di decidere per ogni problema se sia o meno calcolabile, e allo stesso tempo indipendente da un particolare modello di calcolo?

Ora, se riflettiamo bene su quest'ultima affermazione, è chiaro che dare una definizione di calcolabile indipendente da un particolare modello è senza significato, perché è proprio insito nel concetto stesso di algoritmo il fatto di dover essere espresso per mezzo di regole, non ambigue (che esprimono calcoli elementari in un particolare linguaggio), regole la cui interpretazione (non ambigua) presuppone strutturalmente l'esistenza di un modello per il loro calcolo.

L'obiettivo dei prossimi paragrafi è di indagare il concetto di calcolabilità (e non calcolabilità). Partendo dall'ultima osservazione, introdurremo inizialmente uno dei modelli di calcolo che storicamente sono stati proposti per formalizzare il concetto di calcolabile. Successivamente mostreremo come in tale modello esistano problemi non calcolabili. Infine daremo un senso piú generale al problema della calcolabilità, mostrando che tutti i modelli di calcolo che via via sono stati proposti sono tra loro equivalenti.

2. Le macchine di Turing. Vari modelli di calcolo sono stati proposti per esprimere il concetto di calcolabile. Coloro che hanno formulato tali proposte avevano evidentemente uno scopo in comune: quello di formalizzare in modo non ambiguo un insieme di regole di calcolo elementari che costituissero un nucleo su cui basare il concetto di calcolabile. Ognuno ha scelto un modello vicino alla propria esperienza.

Il modello di calcolo che indaghiamo in questo paragrafo, le 'macchine di Turing', è stato proposto da un ingegnere, appunto Alan Turing, nel 1936. L'impronta e la cultura dell'ingegnere sono ben riconoscibili nelle caratteristiche delle macchine di Turing, che ora descriveremo.

3. Su questo argomento vedi *L'infinito* di Lucio Lombardo Radice, «Libri di base» 26, Roma, Editori Riuniti, 1983[2].

Una macchina di Turing, macchina ideale, ha a sua disposizione una memoria costituita da un nastro infinito composto da caselle (vedi fig. 34).

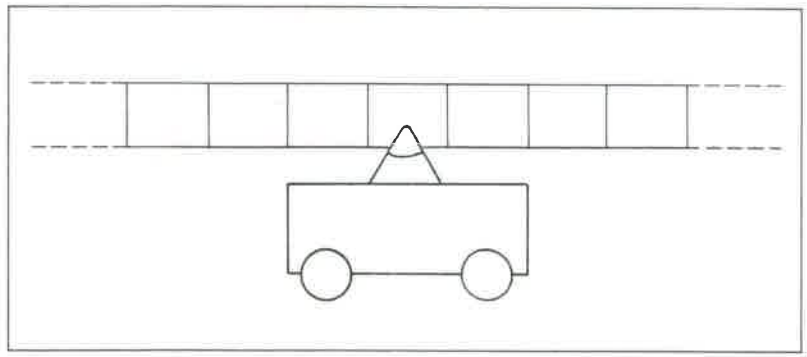


Fig. 34: Una possibile rappresentazione della macchina di Turing.

La macchina può operare sul nastro in tre diverse modalità:

1. può leggere dal nastro il valore scritto in una casella;
2. può scrivere un valore in una casella del nastro. Se nella casella è già scritto un valore, esso viene cancellato e sostituito dal nuovo;
3. fare movimenti elementari di una casella verso destra o verso sinistra.

I simboli che la macchina può leggere o scrivere possono essere qualunque; negli esempi che faremo assumeremo per semplicità che la macchina possa leggere e scrivere due soli simboli: * e 1.

La macchina può trovarsi in ogni momento in uno tra un insieme finito di stati. Chiariremo progressivamente il concetto di 'stato'. Per il momento diciamo che ogni azione che la macchina può compiere: scrittura di un simbolo in una casella, scelta del simbolo, movimento e direzione del movimento, a destra o a sinistra, dipende da due aspetti: il simbolo che la macchina ha letto e lo stato in cui si trova.

A prima vista uno strumento di calcolo con le capacità descritte sembra assai poco potente: le singole operazioni e

funzioni che abbiamo indicato sono veramente elementari. In realtà, come vedremo, è possibile mostrare che una macchina di tal genere è in grado di eseguire almeno tutti i calcoli che è in grado di eseguire un calcolatore elettronico.

Cominciamo con il mostrare che il fatto di poter leggere o scrivere due soli simboli non è un grosso limite.

Se ci limitiamo a considerare problemi che hanno a che fare con numeri interi, per esempio, è sempre possibile codificare un numero intero rappresentando il numero con tante cifre 1 quanto è il valore del numero. Per esempio possiamo rappresentare 7 con la stringa:

1 1 1 1 1 1 1

Questa osservazione ci permette già di mostrare un semplice esempio di macchina di Turing, in grado di riconoscere se un numero è pari o dispari. Supponiamo dunque che la macchina abbia in ingresso il nastro preparato nel seguente modo:

*	1	1	1	1	1	*

. I due asterischi sono posti all'inizio e alla fine del numero che si vuole analizzare. Supponiamo ancora che la testina di lettura della macchina si trovi sul primo 1 a sinistra del numero. Per quanto riguarda poi il modo in cui la macchina comunica il risultato, assumiamo che:

1. se il numero è dispari, la macchina stampa un nuovo asterisco nella prima casella vuota a destra, e si ferma su tale casella.
2. se il numero è pari, non stampa nessun nuovo simbolo e si ferma sulla prima casella vuota a destra.

Descriviamo il comportamento della macchina mediante una tabella, che chiameremo 'tabella di transizione' (vedi fig. 35). In questa tabella, per ogni coppia di possibili stato presente e simbolo letto, sono indicati il nuovo stato, la direzione (eventuale) in cui si muove la macchina e il simbolo (eventualmente) scritto. Assumiamo per ipotesi che la macchina all'inizio del calcolo abbia lo stato «è pari».

stato presente	**simbolo letto**	**direzione**	**simbolo scritto**	**nuovo stato**
è pari	1	→	nessuno	è dispari
è pari	*	→	nessuno	è pari
è pari	vuoto	si ferma	nessuno	è pari
è dispari	1	→	nessuno	è pari
è dispari	*	→	nessuno	è dispari
è dispari	vuoto	si ferma	*	è dispari

Fig. 35: Esempio di tabella di transizione.

Commentiamo la tabella di transizione. La macchina adotta in sostanza la seguente strategia: si sposta a destra ricordando ogni volta se il numero che ha esaminato fino a questo momento è pari o dispari. Quando la macchina raggiunge il simbolo * si sposta a destra ricordando ancora la risposta precedente e a questo punto fornisce il messaggio finale. Infine si ferma.

Come secondo esempio, che descriveremo piú alla svelta, consideriamo una macchina di Turing in grado di sommare due numeri. Anche in questo caso supponiamo che i due numeri siano scritti come stringhe di 1 separati da asterischi

*	1	1	*	1	1	1	*

Supponiamo ancora una volta che la macchina si trovi sul primo 1 a sinistra. A questo punto, una strategia possibile potrà essere quella di spostarsi fino al simbolo * intermedio e continuare a scambiare * con 1 finché la macchina non riconosca il secondo * e dunque concluda il calcolo:

*	1	1	1	*	1	1	*

↓

*	1	1	1	1	*	1	*

↓

*	1	1	1	1	1	*	*

Si può eventualmente prevedere una operazione finale il cui scopo sia di cancellare l'ultimo simbolo *.

Da ciò che abbiamo detto dovrebbe risultare chiaro che nel formalismo delle macchine di Turing ogni macchina corrisponde a uno specifico algoritmo. Naturalmente ci saranno macchine che con particolari dati di ingresso, forniti all'inizio, non terminano mai il loro calcolo. Se ora pensiamo alla idea intuitiva del concetto di calcolabile, è chiaro che una proprietà di tale concetto è la effettiva possibilità di ottenere i risultati a partire dai dati di ingresso in tempo finito. Perciò in questo capitolo, in cui stiamo indagando sulla possibilità di una definizione formale del concetto di calcolabile, useremo il concetto di algoritmo in una accezione diversa, piú restrittiva, di quella data nel capitolo II: assumeremo cioè che un algoritmo, per ogni possibile esecuzione su un qualunque insieme di dati di ingresso, termini sempre il calcolo.

Siamo ora in grado, dopo aver introdotto con qualche dettaglio un modello di calcolo, di affrontare il problema principale di questo capitolo, e cioè il problema della calcolabilità. A questo punto la domanda iniziale diventa:

Esistono problemi per i quali non esiste alcun algoritmo risolutivo, assumendo come modello di calcolo le macchine di Turing?

La risposta è positiva: e lo proveremo dando un esempio di problema per cui appunto non esiste algoritmo risolutivo (nel senso appena specificato).

Il problema cui facciamo riferimento è quello citato all'inizio del capitolo che, riformulato in termini di macchina di Turing, può essere descritto come segue:

«Realizzare una macchina di Turing che sia in grado di decidere, per una qualunque macchina di Turing M e qualunque insieme di dati di ingresso D, se M sottoposta ai dati D termina il calcolo o entra in un ciclo infinito».

Chiamiamo Alt la macchina che stiamo cercando. Stabiliamo anzitutto il modo in cui sono organizzati e forniti i dati in ingresso e i dati prodotti in uscita dalla macchina Alt. Assumiamo che il nastro di ingresso sia configurato inizialmente come segue:

*	descrizione della macchina M	*	dati in ingresso a M	*

Per descrizione della macchina intendiamo quella fornita dalla tabella di transizione (p. 73). Sia la descrizione della macchina M che i dati di ingresso sono forniti mediante un opportuno alfabeto. Se la macchina che stiamo cercando esiste, dovrà terminare scrivendo Sí o No a partire dalla prima casella vuota a destra.

È chiaro che l'ipotesi di partenza implica che in particolare la nostra macchina dovrà poter analizzare una macchina M anche nel particolare caso in cui i dati in ingresso a M siano la sua stessa descrizione.

Apparentemente sembra poco sensato che un programma «legga se stesso» come dato di ingresso. Al contrario, si pensi per esempio a un programma di statistica, che debba calcolare quante volte ogni tipo di istruzione è stata usata in un certo programma; come caso particolare, chiaramente potrà fare la statistica su se stesso.

Nella figura 36 si vede come si presenta il nastro nella nuova situazione, in cui, come si è detto, i dati di ingresso sono la descrizione della macchina.

*	descrizione della macchina M	*	descrizione della macchina M	*

Fig. 36.

Se la macchina Alt esiste, dovrà perciò come caso particolare stabilire per ogni macchina M se M termina avendo come dato di ingresso la propria descrizione.

Ma se Alt esiste, certamente dovrà esistere anche una macchina Alt1 che ha in ingresso una sola copia della descrizione di M.

*	descrizione della macchina M	*

È chiaro, infatti, che l'unica differenza tra Alt e Alt1 sarà che quest'ultima deve semplicemente fare una copia della descrizione di M e poi comportarsi come la macchina Alt.

Modifichiamo a questo punto Alt1 nel seguente modo.

Definiamo una macchina Alt2 che entra in un ciclo infinito quando Alt1 si ferma con risposta affermativa, e si ferma

con risposta negativa, quando anche Alt1 si ferma con risposta negativa.

Alt2 ha un comportamento assai strano. Se infatti le sottoponiamo la descrizione di se stessa, cioè se le chiediamo di stabilire se essa stessa termina avendo come dato in ingresso la sua propria descrizione, allora:

1. nel caso che essa termini avendo come dato di ingresso la sua propria descrizione, allora, per come l'abbiamo costruita, entra in un ciclo infinito;
2. nel caso che essa non termini, si ferma.

È chiaro che siamo entrati in una contraddizione, che ci obbliga a negare l'ipotesi iniziale.

Riassumendo l'argomento esposto fin qui, abbiamo visto che nell'ambito del modello di calcolo rappresentato dalle macchine di Turing esistono problemi per cui *non* esistono algoritmi risolutivi.

A ben vedere, questo risultato non esprime un limite del concetto di calcolo *in assoluto*, quanto nell'ambito di un modello particolare, le macchine di Turing.

Nel corso dell'ultimo secolo molte sono state le proposte volte a definire formalmente modelli di calcolo: è di straordinaria rilevanza notare che tutti questi modelli di calcolo sono stati dimostrati essere tra di loro equivalenti!

Il fatto che tutti i modelli di calcolo proposti si siano dimostrati equivalenti fornisce una grande solidità al concetto di calcolabilità, perché mostra appunto che ogni tentativo di formalizzare questo concetto, partendo da approcci e culture diverse, ha dato luogo agli stessi risultati. Nella parte finale del capitolo trarremo alcune conseguenze da queste affermazioni. Per ora vogliamo mostrare un secondo modello di calcolo, che somiglia piú da vicino a un calcolatore reale, la macchina a programma.

3. Le macchine a programma. Una 'macchina a programma' funziona come un calcolatore, come abbiamo mostrato nel capitolo II. Ha dunque un organo di governo, una unità aritmetica, organi di ingresso e uscita, e una memoria.

La fondamentale differenza strutturale tra un calcolatore reale e una macchina a programma (macchina ideale) è che

nella seconda ogni cella di memoria ha una capacità infinita; in essa è possibile cioè rappresentare qualunque numero o informazione codificata, per grande che sia.

Mentre per la macchina di Turing il meccanismo di calcolo della macchina è stabilito da una tabella di transizione, per la macchina a programma esso è stabilito da un programma, che supporremo memorizzato nell'organo di governo della macchina. Anche per le macchine a programma, come per le macchine di Turing, siamo interessati, nel contesto della calcolabilità, a considerare solo sequenze di calcoli, e quindi programmi, che terminano in un tempo finito.

I programmi che la macchina è in grado di eseguire sono costituiti da un insieme finito di istruzioni.

Ci imbattiamo dunque in un nuovo linguaggio, il linguaggio macchina della macchina a programma, descritto nella tabella 4. In effetti il linguaggio è una via di mezzo tra un comune linguaggio macchina, che usa tipicamente codici binari, e un linguaggio ad alto livello. Per il linguaggio descritto nella tabella 4 valgono le seguenti regole:

1. a ogni istruzione in un programma è associato un numero progressivo;
2. ogni istruzione è composta di due parti di cui:
2.1. il primo descrive il codice operativo della istruzione, cioè un codice che ricorda ciò che la istruzione fa;
2.2. il secondo campo descrive le informazioni che devono essere utilizzate dalla istruzione per poter operare;
3. le celle di memoria sono indicate con nomi simbolici, per esempio A, B, SOMMA;
4. nella tabella per ogni istruzione sono riportati:
4.1. un nome;
4.2. una forma sintattica. La forma sintattica descrive in modo preciso l'istruzione. Nella forma sintattica di una istruzione adottata nella tabella 4 compaiono due tipi di simboli, come mostriamo attraverso l'esempio:

DS nome-di-cella, numero-di-istruzione

a) stringhe di caratteri minuscoli descrivono il simbolo da scrivere mediante il nome della classe a cui appartiene.

Quindi:

numero-di-istruzione

descrive simbolicamente un elemento della classe dei numeri interi, per esempio: 3, 5, 67;

b) gli altri simboli descrivono esattamente il simbolo da scrivere. Per esempio la stringa

DS

indica che a quel punto della istruzione dovrà comparire il simbolo DS. Analogamente per il simbolo ',';

5. l'esempio è appunto un esempio di istruzione reale che rispetta la forma sintattica;
6. il significato descrive l'insieme di attività svolte nella macchina quando quella istruzione è eseguita. Per semplicità è riferito direttamente all'esempio.

Come esempio di programma, mostriamo come nella macchina a programma è possibile decidere se un numero è pari o dispari. Potremo in questo modo confrontare le diverse strategie dei due modelli di calcolo che abbiamo introdotto.

Assumiamo che il numero sia memorizzato nella cella A; se il numero è pari, metteremo 0 nella cella B, se è dispari metteremo 1.

Nella figura 37 riportiamo il programma. Per ogni istruzione sono descritti:

1. il numero progressivo;
2. la istruzione;
3. il significato;
4. un commento che fornisce sinteticamente il ruolo che la istruzione ha nell'algoritmo.

In sostanza il programma continua a sottrarre 1 al valore memorizzato nella cella A finché non arriva a zero; ogni volta che il valore è diminuito di 1 il programma ricorda se il numero finora esaminato sia pari o dispari.

Tabella 4 **Descrizione del linguaggio per la macchina a programma**

nome	forma sintattica	esempio	significato
zero	Z nome-di-cella	Z A	metti 0 nella cella con nome A
successore	S nome-di-cella	S A	aggiungi 1 al contenuto della cella con nome A
vai-a	V num.-di-istruzione	V 5	vai a eseguire la istruzione numero 5
decrementa e-salta	DS nome-di-cella, num.-di-istruzione	DS A,5	se A ha valore 0, vai ad eseguire la istruzione numero 5, altrimenti diminuisci A di 1 ed esegui la prossima.
fermati	F	F	ferma la esecuzione

n.	istruzione		significato	commento
1	Z	B	metti zero in B	ricordati che per ora il numero è pari
2	DS	A,B	se A è zero, vai alla fine del programma	se a questo punto il numero è stato ridotto a zero, in B c'è il valore corretto e dunque il programma termina
3	Z	B	metti zero in B	
4	S	B	metti 1 in B	a questo punto il numero finora esaminato è dispari
5	DS	A,B	se A è zero vai alla fine del programma	se a questo punto il numero è stato ridotto a zero, in B c'è il valore corretto e dunque il programma termina
6	Z	B	metti zero in B	a questo punto il numero finora esaminato è pari
7	V	Z	vai alla istr. Z	continua a sottrarre 1 se necessario
8	F		fermati	il programma è terminato

Fig. 37: Programma per il calcolo di parità nella macchina a programma.

Dopo che la lettura e l'esame di un programma ci ha resi piú familiari col linguaggio, possiamo fare una osservazione importante sul linguaggio stesso.

Notiamo ora che nell'ambito delle istruzioni del linguaggio ve ne è una che non è indispensabile, e cioè la V.

La istruzione V può infatti esprimersi mediante un uso combinato delle istruzioni Z e DS. Ce ne convinceremo subito con un esempio.

La istruzione

```
V 5
```

può esprimersi come

```
Z A
DS A,5
```

Infatti porre 0 in A dà luogo a un sicuro salto al momento della esecuzione della istruzione DS.

Il linguaggio presentato è dunque ridondante nel senso che esistono istruzioni che possono essere espresse per mezzo di altre. Notiamo che questo fatto non è di per sé negativo, poiché un linguaggio è uno strumento di descrizione del pensiero e quando si ha a disposizione un linguaggio piú ricco migliora e diventa piú semplice in genere la possibilità di esprimersi.

Per esempio, nel nostro caso, il fatto di poter utilizzare la istruzione V rende piú semplice e naturale la descrizione della funzionalità che in quel punto del programma occorre effettuare, e cioè «andare a eseguire una istruzione lontana».

Una seconda importante osservazione che possiamo fare a questo punto parte dalla seguente domanda: oltre alla istruzione V è possibile eliminare altre istruzioni dal linguaggio, senza ridurre la sua capacità espressiva, lasciando cioè intatta la sua potenzialità?

La risposta in questo caso è No, e il lettore se ne può facilmente convincere tentando di esprimere ciascuna delle istruzioni rimaste in termini delle altre: ciascuna delle istruzioni «fa qualcosa» che non può essere fatto da nessuna combinazione delle altre.

Non è possibile, per esempio, realizzare la DS per mezzo delle altre istruzioni perché nessuna delle altre dà la possibilità di saltare a una istruzione lontana. E cosí via.

Tornando al programma, vogliamo osservare che anche in esso possiamo trovare una qualche forma di ridondanza. C'è infatti nel programma una istruzione inutile, che può cioè essere eliminata senza alterare l'effetto complessivo del calcolo. Ci riferiamo alla istruzione numero 3, la Z B.

Notiamo infatti che alla istruzione n. 3 si può arrivare da un solo punto del programma, e cioè l'istruzione 2. A sua volta all'istruzione 2 si può arrivare solo dall'istruzione 1 o dall'istruzione 7. Ma sia l'istruzione 1 che l'istruzione eseguita prima della istruzione 7 assegnano 0 alla cella B. Dunque quando viene eseguita la istruzione 3, B è già al valore 0, e non serve a nulla riporla di nuovo a 0! È quindi possibile trasformare il programma in un nuovo programma in cui la istruzione 2 non compare piú (vedi fig. 38).

Naturalmente dovremo aggiornare i numeri delle istruzioni e di conseguenza i numeri nelle istruzioni di salto.

```
1   Z   B
2   DS  A,7
3   S   B
4   DS  A,7
5   Z   B
6   V   2
7   F
```

Fig. 38.

Il ragionamento che ci ha portato alla conclusione che i due programmi sono equivalenti, anche in questo caso semplice, era piuttosto complesso. Sarebbe molto comodo poter scrivere un programma che decidesse se due programmi sono o meno equivalenti, naturalmente per ogni possibile coppia di programmi. Anche questo problema ricade nella classe dei problemi per i quali non si può sperare che esista un algoritmo risolutivo!

Prima di concludere la descrizione di questo secondo modello di calcolo, osserviamo che se invertiamo il significato dei due valori che forniscono le risposte del problema (0 dispari, 1 pari), possiamo ulteriormente semplificare il programma. Avremo allora la versione riportata nella figura 39.

1	S	B
2	DS	A,6
3	Z	B
4	DS	A,6
5	V	1
6	F	

Fig. 39.

4. Equivalenza di modelli di calcolo e tesi di Church. Nei paragrafi precedenti abbiamo introdotto due diversi modelli di calcolo e abbiamo indagato tramite esempi sulle loro strategie di calcolo. In questo paragrafo, vogliamo procedere oltre nella nostra indagine mostrando che, sia pure attraverso prospettive diverse, i due modelli hanno la stessa potenza di calcolo. Generalizzando questo risultato, arriveremo a formulare la tesi espressa alla fine del paragrafo IV, 2, detta 'tesi di Church'.

Proviamo allora anzitutto che, data una qualunque macchina di Turing, è possibile costruire un programma per la macchina a programma che calcola esattamente la stessa funzione.

L'idea che sta dietro la prova è in principio molto semplice: basta costruire una macchina a programma che riesca a simulare la macchina di Turing, sia pure mediante attività elementari «adattate» alla propria struttura e modo di funzionamento.

Anzitutto, abbiamo visto che una macchina di Turing si può descrivere mediante una tabella, ogni riga della quale descrive ordinatamente cinque informazioni:

— stato attuale;
— simbolo letto;
— direzione da prendere;
— nuovo stato;
— simbolo da scrivere.

Per semplificare il nostro ragionamento assumiamo che, nella macchina di Turing che vogliamo simulare, i simboli utilizzati siano le dieci cifre del sistema di numerazione decimale 0, 1, 2, ..., 9. È chiaro che ogni altro simbolo può essere codificato in maniera non ambigua per mezzo di una stringa opportuna in cui compaiono solo questi dieci simboli.

Per una macchina che lavora con al massimo dieci simboli, la tabella di transizione avrà al massimo dieci righe per ogni stato; esprimendoci per mezzo di simboli, al generico stato S_i saranno associate le quintuple:

S_i	0	D_{i0}	NS_{i0}	SS_{i0}
S_i	1	D_{i1}	NS_{i1}	SS_{i1}
...	...	...	...	...
S_i	9	D_{i9}	NS_{i9}	SS_{i9}

Il significato dei simboli della prima riga è il seguente:

— D_{i0} è la direzione da prendere nel caso sia stato letto 0;
— NS_{i0} è il nuovo stato;
— SS_{i0} è il simbolo da scrivere.

Dato che il numero di celle in cui la macchina può aver scritto un simbolo è finito, l'intero nastro può essere descritto per mezzo di tre soli numeri (vedi fig. 40):

Fig. 40: Configurazione del nastro.

— il numero che descrive il simbolo attualmente letto;
— il numero che descrive la stringa di cifre alla sua sinistra;
— il numero che descrive la stringa di cifre alla sua destra.

Per semplificare il ragionamento, supponiamo che tale ultimo numero venga composto leggendo la stringa da destra verso sinistra. Il nastro della figura 40 può essere descritto, con tali convenzioni, dai numeri 1093, 4, 9206.

Supponiamo che tali numeri siano memorizzati nel corso del calcolo in tre variabili chiamate LET, SIN, DES.

A questo punto possiamo simulare la macchina di Turing semplicemente scrivendo per ogni stato un frammento di programma come quello descritto nella figura 41.

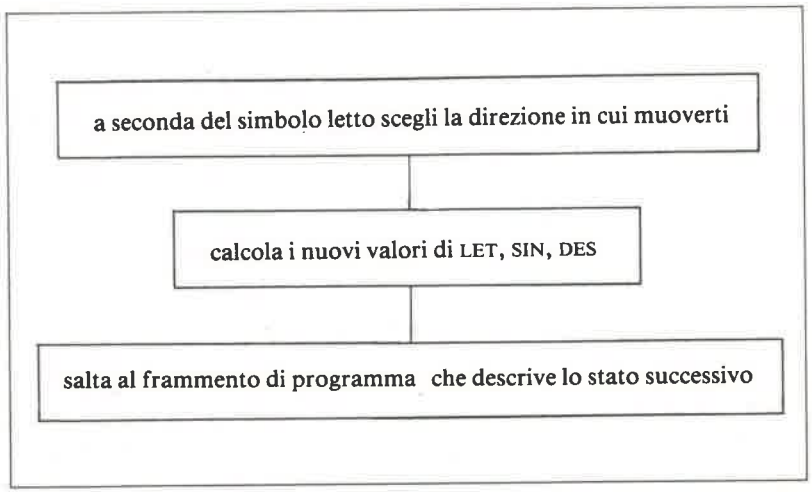


Fig. 41: Programma per la simulazione di una macchina di Turing.

Consideriamo ora separatamente le tre funzioni.

a seconda del simbolo letto scegli la direzione in cui muoverti

Questa funzione può realizzarsi riconoscendo il simbolo letto e saltando, a seconda del simbolo, a due diversi possibili frammenti di programma, associati logicamente alle direzioni sinistra e destra

calcola i nuovi valori di LET, SIN, DES

Chiaramente i nuovi valori di LET, SIN, DES dipenderanno dal simbolo letto e dalla direzione in cui muoversi. A questo punto del programma la direzione è già stata scelta: abbiamo infatti un frammento di programma per la direzione sinistra e uno per la direzione destra.

Consideriamo il caso di direzione sinistra e riprendiamo l'esempio della figura 40, supponendo che il nuovo simbolo da scrivere sia 7.

È chiaro che i nuovi valori di LET, SIN, DES saranno in questo caso:

LET = 7
SIN = 109
DES = 92064

In generale se n, m, p sono gli attuali valori di LET, SIN, DES, i nuovi valori saranno:

DES = DES + LET
LET = SS_{ik}
SIN = parte intera della divisione di SIN per 10

salta al frammento di programma che descrive lo stato successivo

A questo punto, poiché siamo già su frià su frammenti di programma diversi a seconda del simbolo letto, e poiché lo stato successivo è individuato dallo stato attuale e dal simbolo attuale, basterà fare un salto al frammento di programma opportuno, con la solita tecnica di salto.

Tralasciamo il dettaglio sulla effettiva codifica in termini di macchine a programma, e il lettore si fiderà del fatto che, con qualche sforzo, si riesce a far calcolare a una macchina a programma la divisione per 10 e le altre funzionalità in cui ci siamo imbattuti.

Abbiamo cosí provato che una macchina di Turing può essere simulata da una macchina a programma. Tralasciamo la dimostrazione inversa, perché a questo punto non aggiungerebbe molto al ragionamento.

I precedenti risultati dimostrano che due diversi formalismi proposti per descrivere il concetto di calcolabile sono equivalenti, nel senso che descrivono lo stesso insieme di calcoli. Questo importante risultato è vero per tutti i formalismi che nel corso dell'ultimo secolo sono stati proposti per formalizzare cosa si intenda per calcolabile.

È possibile allora ipotizzare, come ha fatto il matematico americano Alonzo Church, che ciò che intuitivamente si intende per calcolabile sia esattamente l'insieme delle funzioni calcolabili con le macchine di Turing. La affermazione precedente non può essere direttamente provata, nel senso che appunto non si può formalmente definire cosa si intenda per «intuitivamente calcolabile».

Ciò che è possibile fare è proporre un nuovo modello di calcolo e mostrare che i meccanismi di calcolo elementari del formalismo corrispondono a processi eseguibili: nel caso delle macchine di Turing, ad esempio, possiamo di buon grado accettare che il movimento a sinistra o a destra della macchina di una sola casella, la scrittura di un simbolo in una casella, il calcolo del nuovo stato siano processi intuitivamente eseguibili nel mondo fisico.

La tesi di Church afferma appunto che ogni possibile formalismo che noi ci possiamo inventare, e che risponda ai criteri su esposti, è equivalente alle macchine di Turing, e dunque che il concetto di calcolabile è un concetto stabile, maturo della scienza moderna.

Se non è possibile dimostrare la tesi di Church, è possibile dimostrare il suo contrario, qualora qualcuno individuasse un formalismo (intuitivamente calcolabile) piú potente di quelli attualmente noti.

La ricchezza di risultati ottenuti negli ultimi anni fornisce alla tesi di Turing una grande rilevanza nella scienza moderna e, per cosí dire, un grande fascino.

5. Conclusioni. In questo capitolo ci siamo occupati del calcolabile. Nel definire i modelli di calcolo per caratterizzare ciò che è calcolabile, non ci siamo preoccupati di problemi di efficienza. Eppure è chiaro che non affideremmo mai a una macchina di Turing il calcolo degli stipendi dei dipendenti di una ditta, perché ci sarebbe il rischio di arrivare alla fine del mese con la macchina che ancora pazientemente si sposta di casella in casella con lentezza esasperante.

E, indipendentemente dalla natura meccanica della macchina di Turing e dalla scarsa potenza delle singole operazioni che è in grado di compiere, potrebbe darsi che esistano algoritmi che richiedono tempi di soluzione che crescono in maniera rapidissima e impraticabile anche se risolti con macchine sofisticate e all'avanguardia.

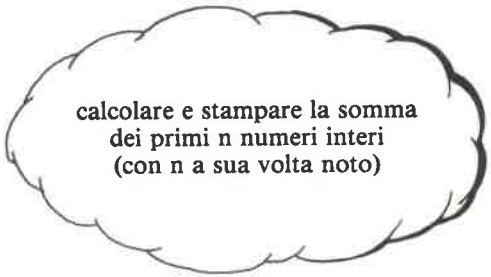
Insomma, è chiaro che se mi servono dei dati in una giornata o nel giro di pochi minuti, non mi consola molto il fatto che per quel problema conosco uno o piú algoritmi risolutivi, se poi mi occorrono sei mesi di calcoli anche usando un calcolatore elettronico.

Per questa ragione, nel capitolo seguente affronteremo l'argomento della misura di efficienza degli algoritmi.

V. L'EFFICIENZA

Introduzione - Misure di efficienza - L'efficienza è un falso problema?

1. Introduzione. Cominciamo a considerare il seguente problema:



Il problema generalizza quello mostrato nel capitolo III dove si trattava di sommare i primi 100 numeri. Un algoritmo risolutivo, quello che ci viene piú naturalmente in mente se non ci pensiamo tanto su e se dobbiamo eseguirlo a mente, consiste nel generare pazientemente nella nostra testa i valori da 1 a n e nel sommarli ordinatamente.

In termini del linguaggio dei grafi di flusso l'algoritmo è mostrato nella figura 42.

Nell'algoritmo la variabile I ha il duplice compito di generare ogni volta il nuovo valore da sommare e di ricordare qual è l'ultimo numero sommato, cosí da poter decidere quando abbandonare il ciclo. La variabile S svolge il solito ruolo di conteggio della somma parziale.

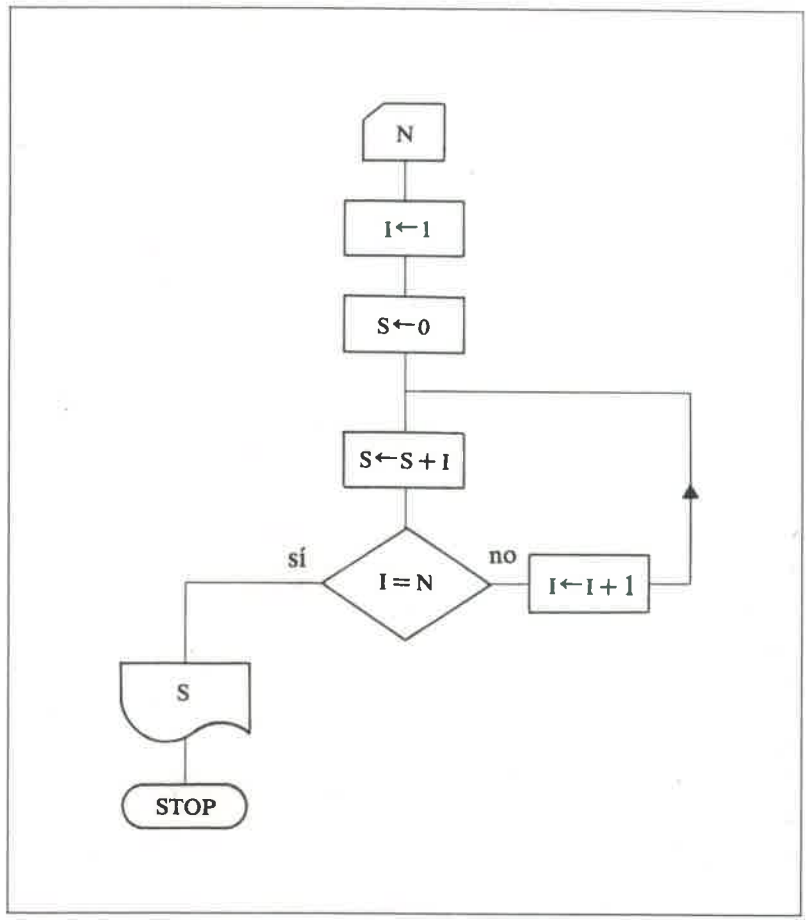


Fig. 42: Primo algoritmo per la somma dei primi n numeri interi.

Se ricordiamo la formula che abbiamo descritto nel capitolo III, ci accorgiamo che il problema può essere risolto in modo molto piú semplice: infatti, la somma dei primi n numeri interi, con n qualunque, vale esattamente

$$\frac{n \times (n+1)}{2}$$

Un secondo algoritmo per il nostro problema è dunque quello mostrato nella figura 43.

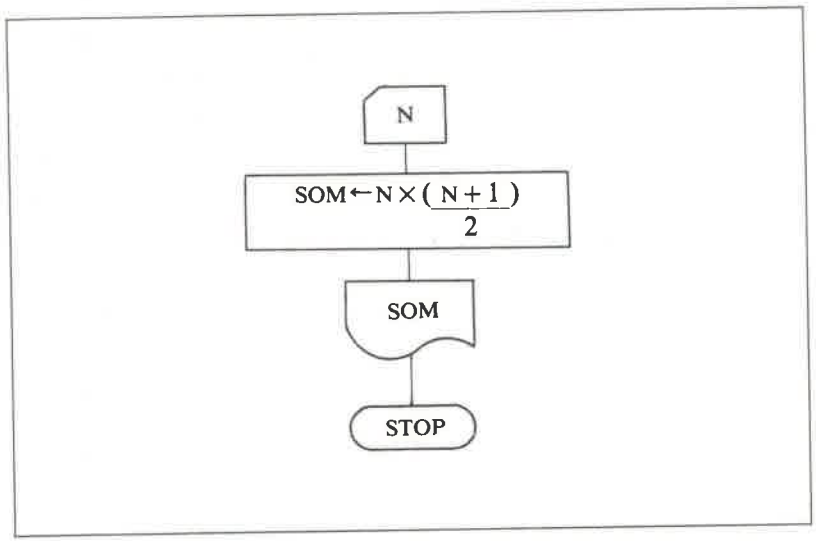


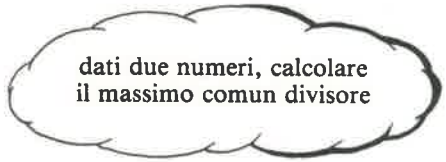
Fig. 43: Secondo algoritmo per la somma dei primi n numeri.

In questo caso dobbiamo eseguire solo tre operazioni aritmetiche, mentre il primo algoritmo richiedeva, come il lettore può verificare, 3 × N somme e N confronti. Dunque, al crescere del valore n dei numeri da sommare, usando il primo algoritmo cresce proporzionalmente il numero delle operazioni da effettuare, mentre usando il secondo il numero di operazioni resta costante.

In base alle considerazioni fatte siamo portati a dire che il secondo algoritmo è piú efficiente del primo. Per ora ci basiamo su una nozione intuitiva di efficienza intesa come capacità dell'algoritmo di calcolare i risultati del problema a partire dai dati di ingresso con limitato uso di risorse. In questo caso la risorsa a cui eravamo interessati era il tempo di esecuzione; un'altra risorsa a cui possiamo essere interessati è per esempio lo spazio di memoria occupato.

Nel problema ora esaminato abbiamo visto che per trovare una soluzione efficiente dovevamo possedere una qualche conoscenza delle proprietà (matematiche) dei dati oggetto del problema. Questo accade anche nel prossimo problema: in piú vedremo che una soluzione apparentemente soddisfacente può essere, con una piú approfondita indagine, ulteriormente migliorata.

Consideriamo il seguente problema:



Il 'massimo comun divisore' di due numeri è il massimo numero intero che divide entrambi con resto 0. Partendo da tale definizione possiamo facilmente individuare un algoritmo per questo problema. Esprimeremo tale algoritmo in un linguaggio ibrido che utilizza prevalentemente il linguaggio naturale per esprimere i singoli calcoli da effettuare e il linguaggio dei grafi di flusso per esprimere le istruzioni di lettura e scrittura e l'ordine tra le operazioni (vedi fig. 44).

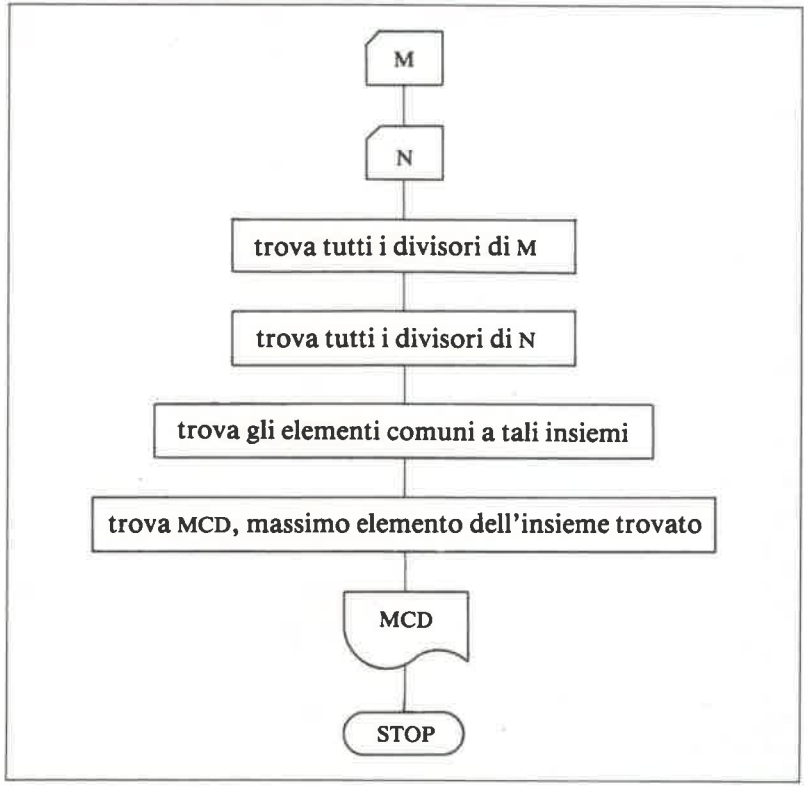


Fig. 44: Primo algoritmo per il problema del massimo comun divisore.

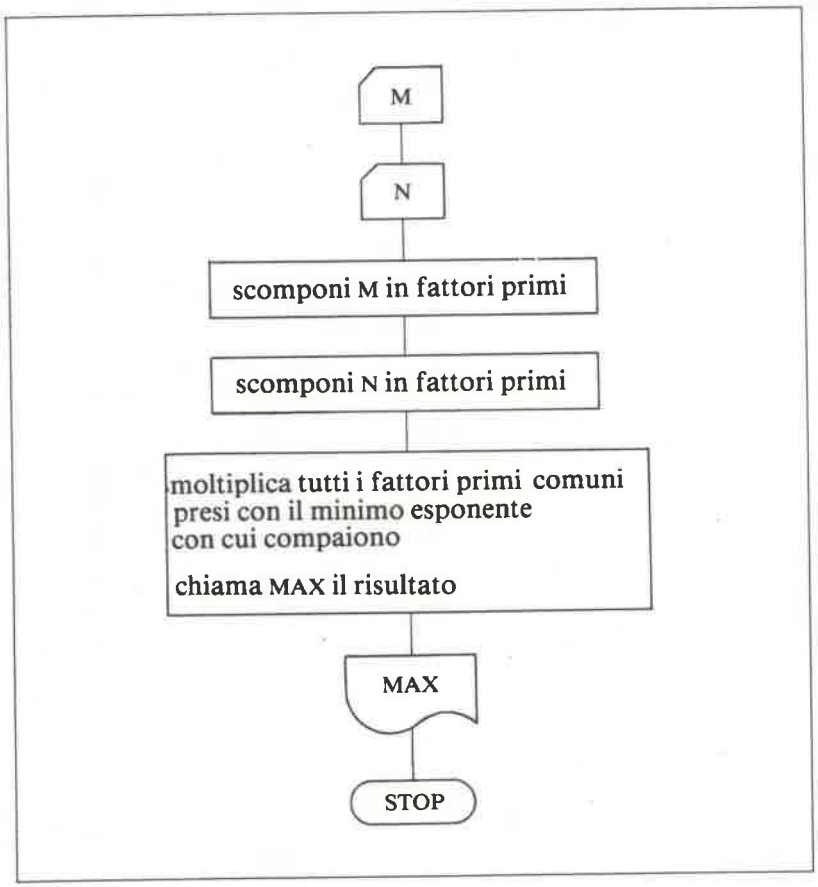


Fig. 45: Secondo algoritmo per il problema del massimo comun divisore.

Un secondo algoritmo, che si ricava ricordando le proprietà del massimo comun divisore studiate alle scuole medie, è riportato nella figura 45.

Confrontiamo qualitativamente i due algoritmi eseguendoli sulla coppia di numeri 70 e 40.

Algoritmo 1

1. Calcolare i divisori di 70
   Otteniamo l'insieme {1,2,5,7,10,35,70}
2. Calcolare i divisori di 40
   Otteniamo l'insieme {1,2,4,5,8,10,20,40}

3. Calcolare i divisori comuni
   Otteniamo l'insieme {1,2,5,10}
4. Calcolare il massimo
   Otteniamo: 10

Algoritmo 2

1. Scomporre 70 in fattori primi
   Otteniamo $70 = 7^1 \times 2^1 \times 5^1$
2. Scomporre 40 in fattori primi
   Otteniamo $40 = 2^3 \times 5$
3. Calcolare il massimo comun divisore
   Otteniamo 10 (uguale a 2×5)

Entrambi gli algoritmi richiedono, anche nel caso piuttosto semplice che abbiamo mostrato, calcoli piuttosto complessi.

Mostriamo ora un algoritmo notevolmente piú semplice da eseguire rispetto ai due precedenti. Tale algoritmo è detto di Euclide, dal nome del matematico greco vissuto intorno al 300 a.C. che per primo lo formulò.

Esso si basa su alcune proprietà del massimo comun divisore che descriviamo nella tabella 5.

Tabella 5 **Proprietà del massimo comun divisore di due numeri utilizzate nell'algoritmo di Euclide**

1. Se m = n	allora	MCD (m, n) = m (oppure n)
2. Se m > n	allora	MCD (m, n) = MCD (m-n, n)
3. Se m < n	allora	MCD (m, n) = MCD (m, n-m)

La prima proprietà dice che se due numeri sono uguali, allora il massimo comun divisore dei due numeri è ovviamente uno qualunque dei due.

La seconda e terza proprietà dicono che se i due numeri sono diversi, il massimo comun divisore dei due numeri è uguale al massimo comun divisore del maggiore dei due e del numero differenza.

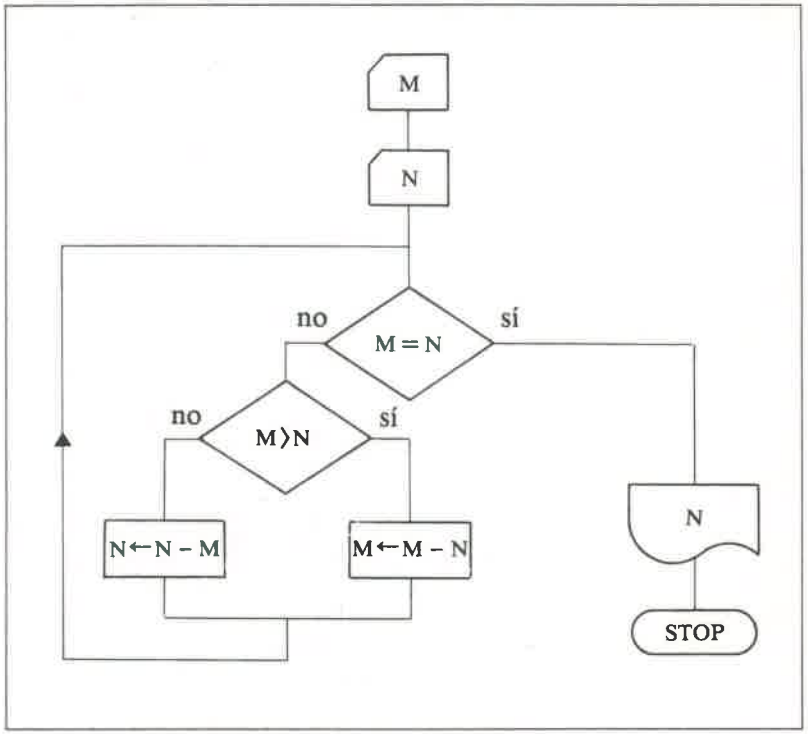


Fig. 46: Algoritmo di Euclide per il problema del massimo comun divisore.

A partire da tali proprietà possiamo ora esprimere il cosiddetto'algoritmo di Euclide' nel linguaggio dei grafi di flusso (vedi fig. 46).

La strategia adottata nell'algoritmo è la seguente.

Letti due numeri, se sono uguali, usando la proprietà 1 della tabella 5 possiamo subito decidere che il massimo comun divisore è uno dei due numeri. Se sono diversi allora

Variabile	Passo					
	1	2	3	4	5	6
M	70	30	30	30	10	10
N	40	40	20	20	20	10

Fig. 47: Coppia di valori memorizzati successivamente in M e N.

Tabella 6 **Proprietà del massimo comun divisore di due numeri**

1. Se m/n = p con resto 0 allora MCD (m,n) = n
2. Se m/n = p con resto r allora MCD (m,n) = MCD(r,n)

confrontandoli possiamo, usando le proprietà 2 o 3, ricondurre il calcolo di MCD(m, n) a uno dei due casi piú semplici MCD(m-n, n) o MCD(m, n-m). Inoltre possiamo usare sempre le stesse due variabili, cioè M ed N, per ricordare i due numeri da confrontare e da aggiornare progressivamente.

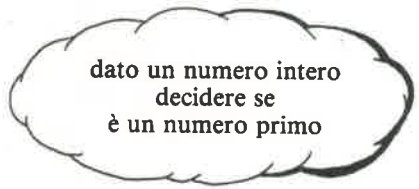
Eseguiamo l'algoritmo partendo ancora dai numeri 70 e 40. Nella figura 47 compaiono i risultati parziali del calcolo.

Si può vedere come il calcolo risulti notevolmente piú rapido rispetto ai precedenti algoritmi. Possiamo dire che, perlomeno nel caso esaminato, l'algoritmo di Euclide è piú efficiente degli altri due algoritmi presentati.

Se però esaminiamo con attenzione l'algoritmo di Euclide, notiamo che anch'esso può essere ulteriormente migliorato. Proviamo ad applicarlo nel caso della coppia di numeri 1000 e 10. Osservando i due numeri è facile arguire immediatamente che il loro massimo comun divisore è 10. Basta infatti osservare che 10 divide esattamente 1000. Partendo da questa osservazione possiamo generalizzare le proprietà della tabella 5 con quelle della tabella 6, in cui facciamo valere per semplicità l'ipotesi che m > n.

Potremmo facilmente passare a descrivere un nuovo algoritmo usando le proprietà della tabella 6, ma ci interessa passare invece a un terzo esempio, che metterà in evidenza un ulteriore aspetto del calcolo della efficienza che finora non è emerso.

Consideriamo il seguente problema:



Come noto, un numero si dice 'primo' se è divisibile con resto 0 solo per 1 e per se stesso.

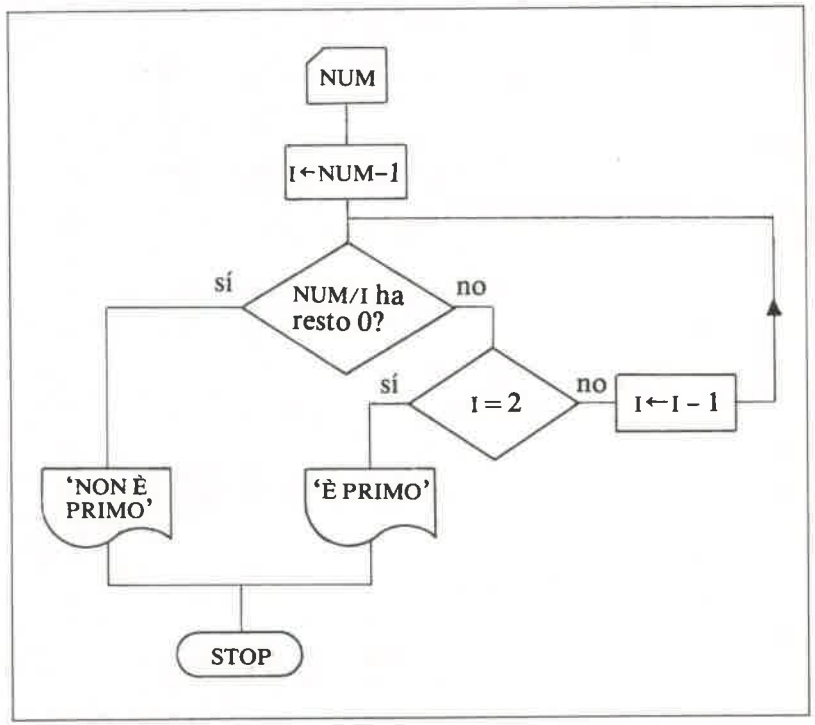


Fig. 48: Algoritmo 1 per verificare se un numero è primo.

In base alla definizione, è facile trovare un primo algoritmo risolutivo del problema (vedi fig. 48). L'algoritmo divide NUM per tutti i numeri interi fino a 2, partendo dal numero immediatamente piú piccolo di NUM.

È facile scoprire una inefficienza nell'algoritmo 1. Per esempio, se NUM è uguale a 31 è inutile iniziare a provare a partire dal numero 30. Si può direttamente iniziare dal numero intero immediatamente piú piccolo di 31/2, cioè 15, poiché tutti i numeri maggiori di 15 non potranno dividere esattamente 31. Possiamo eliminare tale inefficienza dando luogo all'algoritmo della figura 49.

Anche il nuovo algoritmo è inefficiente, nel senso che spesso fa dei calcoli che potrebbe evitare. Infatti, nella ricerca di divisori è inutile partire dalla metà del numero; è sufficiente partire dalla sua radice quadrata (che è in genere un numero molto inferiore alla metà). Dimostriamo la precedente affermazione.

Sappiamo che la radice quadrata r di un numero n è il numero tale che r × r = n. Ora, se un numero n non è primo, può certamente esprimersi come prodotto di due numeri, che chiamiamo n1 ed n2. A questo punto possono darsi due casi:

Caso 1: n1 è uguale a n2.

In questo caso n1 (ed n2) è certamente uguale alla radice quadrata di n, e dunque la divisione di n per la sua radice quadrata rivelerà che non è primo.

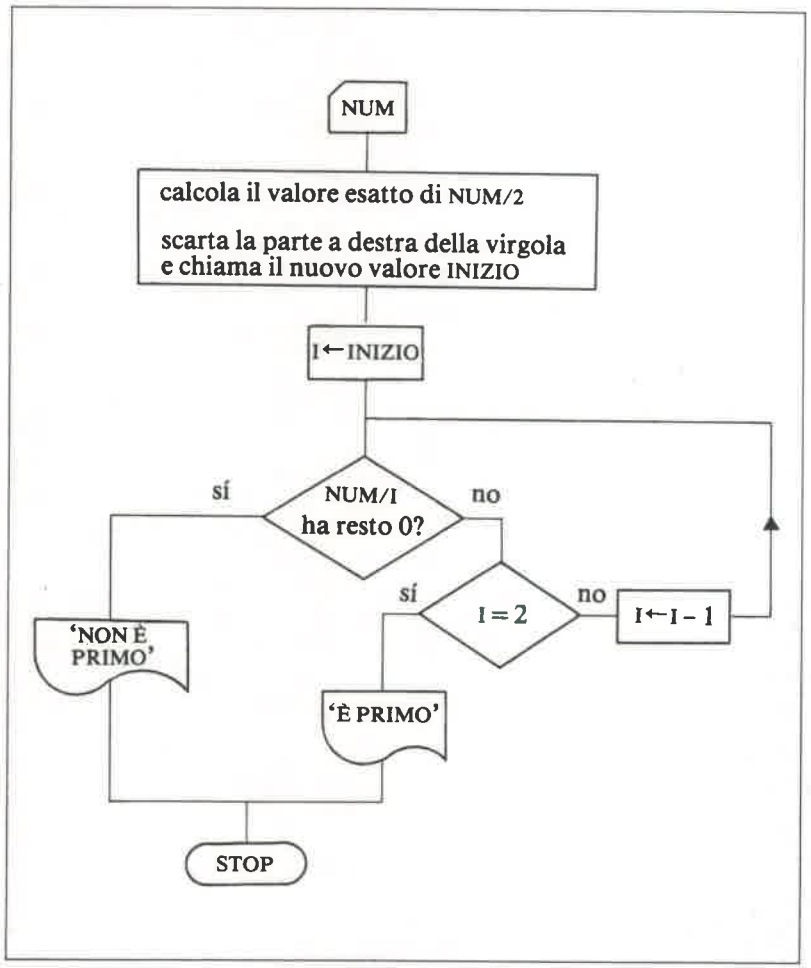


Fig. 49: Algoritmo 2 per verificare se un numero è primo.

Caso 2: n1 è maggiore di n2 (o viceversa, ma la conclusione non cambia).

In questo caso certamente n2 è minore della radice quadrata di n, e dunque, cominciando a dividere n a partire dalla sua radice quadrata, certamente troveremo un numero che lo divide.

La nuova proprietà che abbiamo trovato permette di formulare l'algoritmo 3 (vedi fig. 50).

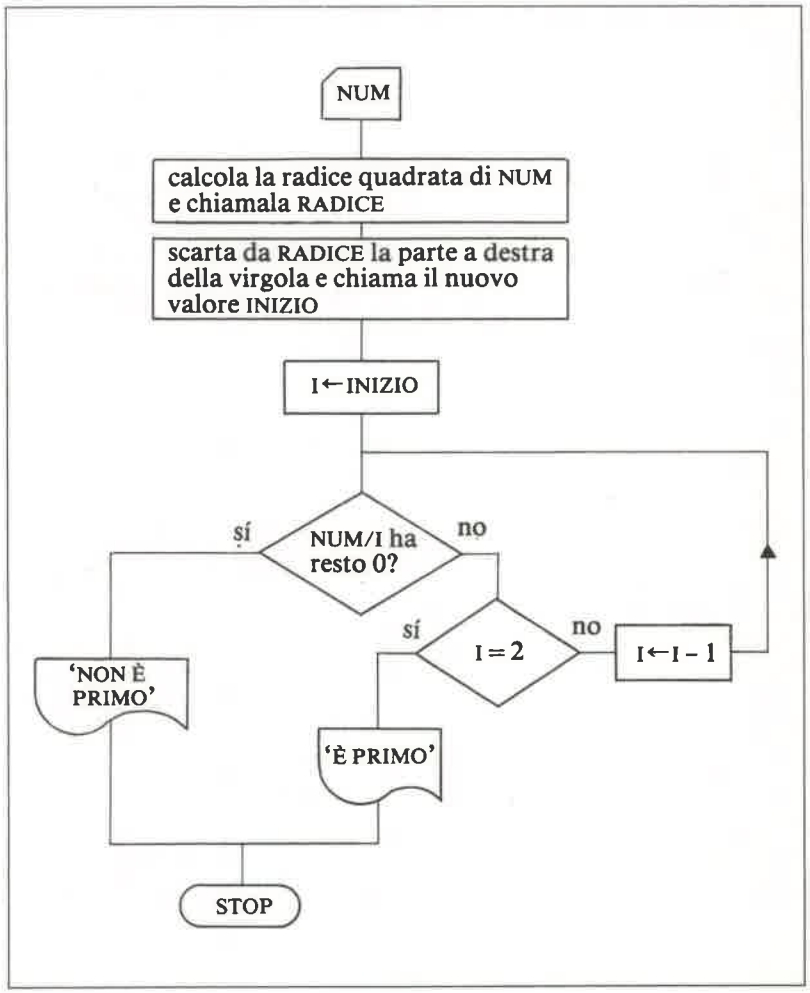


Fig. 50: Algoritmo 3 per verificare se un numero è primo.

Un difetto residuo dell'algoritmo 3 è di cominciare la ricerca dei divisori dal numero piú grande, per scendere man mano fino al numero piú piccolo, il numero 2. È intuitivo infatti che, dato un qualunque numero n, il numero 2 è quello per cui è massima la probabilità che lo divida: la metà dei numeri è divisibile esattamente per 2, un terzo per 3, un quarto per 4, e cosí via. Perché allora non cominciare dai numeri bassi, per poi provare per numeri crescenti? In tal modo, se il numero non è primo, aumentiamo la probabilità di accorgercene prima.

Possiamo perciò costruire un nuovo algoritmo 4.

L'algoritmo 4, che non mostriamo, può essere ulteriormente migliorato se osserviamo che, se il numero non risulta divisibile per 2, è inutile dividerlo poi per tutti gli altri numeri pari: possiamo perciò provare a dividerlo per due e poi per tutti i numeri dispari: chiamiamo questo nuovo algoritmo algoritmo 5. Fermiamoci qua, anche se l'algoritmo potrebbe essere ulteriormente migliorato.

Fissiamo ora l'attenzione sugli algoritmi 3 e 4. I due algoritmi hanno lo stesso comportamento rispetto ai casi peggiori, cioè tutti i casi in cui il numero è primo, e gli algoritmi scoprono tale caratteristica solo dopo avere diviso il numero per i potenziali divisori. Al contrario, se valutiamo le prestazioni dei due algoritmi nel loro comportamento medio su tutti i casi possibili, possiamo senz'altro prevedere, in base alle osservazioni fatte in precedenza, che l'algoritmo 4 è nel caso medio piú efficiente dell'algoritmo 3.

È importante notare che in molte circostanze ciò che interessa dell'algoritmo è il suo comportamento rispetto al caso peggiore. Questo accade in tutte le utilizzazioni dei calcolatori cosiddette in tempo reale, in cui cioè vogliamo che la risposta del calcolatore, qualunque siano i dati che gli abbiamo fornito, arrivi entro un tempo prefissato, significativo per la applicazione. Per esempio, in un sistema di controllo di un altoforno, quando si verifichi un malfunzionamento c'è un tempo limite di reazione al guasto oltre il quale subentrano rischi di malfunzionamenti piú gravi.

Facciamo il punto della situazione per quanto riguarda il problema della efficienza. Fino a ora, nel confrontare gli algoritmi degli esempi precedenti abbiamo assunto una definizione di efficienza basata sul numero di calcoli effettuati

	10	15	20	25	30
A1	29	72	131	209	296
A2	9	23	41	65	86
A3	13	25	38	52	70
A4	11	21	32	44	54
A5	7	12	19	26	32

Fig. 51.

dall'algoritmo (quindi ancora piuttosto generica). Abbiamo notato poi che ci sono almeno due tipi di efficienza cui possiamo essere interessati, quella nel caso medio e quella nel caso peggiore, e per quanto riguarda il confronto tra algoritmi, lo abbiamo effettuato o in base a poche prove (algoritmi per il calcolo del massimo comun divisore) o in base ad alcune analisi sulla proprietà dei dati osservati.

Possiamo migliorare la nostra indagine se osserviamo il comportamento degli algoritmi eseguendoli per un campione significativo dei dati di ingresso. Non ha infatti senso, come già abbiamo osservato nel capitolo sulla correttezza, eseguirli per tutti i possibili valori di ingresso. Possiamo essere aiutati in questa valutazione dall'uso di strumenti automatici. Nel caso degli algoritmi per verificare se un numero è primo, un'idea ragionevole può essere quella di eseguire gli algoritmi per insiemi di numeri interi via via piú grandi e confrontarli dal punto di vista del numero complessivo dei tentativi che ciascuno fa nella ricerca di divisori esatti.

A tale scopo possiamo scrivere un programma il cui scopo è fare proprio questi conteggi. I risultati piú significativi sono riportati nelle tabelle delle figure 51 e 53 e nei corrispondenti grafici delle figure 52 e 54 a pp. 101-103.

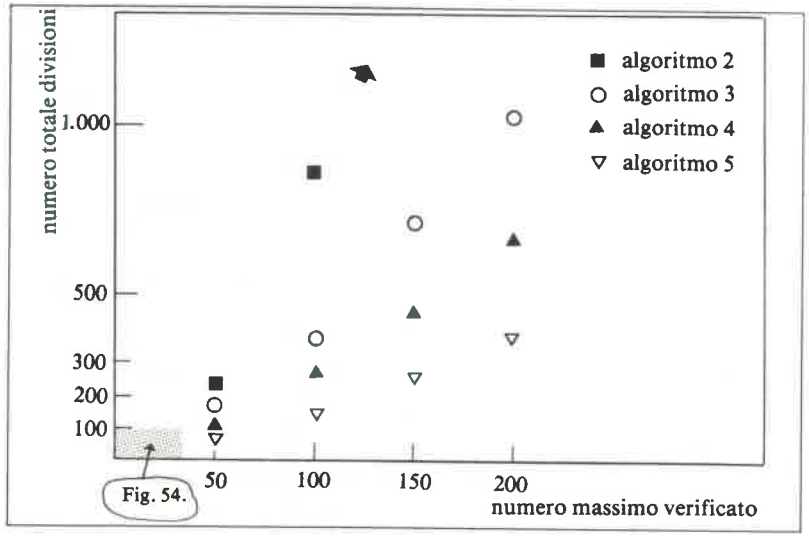


Fig. 52.

La tabella della figura 51 riporta il numero dei tentativi fatti dai cinque algoritmi per i primi 10, 15, 20, 25, 30 numeri. La tabella della figura 53 riporta i tentativi per i primi 50, 100, 150, 200 numeri naturali. La prima tabella fornisce quello che viene chiamato il 'comportamento nel transitorio', la seconda il comportamento detto 'asintotico'. Nel comportamento asintotico i fattori che influenzano le prestazioni dell'algoritmo (nel nostro caso le proprietà dei numeri naturali e la strategia usata dall'algoritmo) tendono ad avere una influenza sempre piú stabile al variare del campione esaminato. Nel transitorio può accadere invece che alcune proprietà abbiano maggiore influenza, e quindi al variare del campione modifichino anche sostanzialmente le indicazioni che fornisce il comportamento asintotico.

Nel nostro caso, per esempio, quando i numeri sono piccoli tendono ad avere divisori che differiscono di poco dalla metà del numero: questa è la ragione del miglior comportamento dell'algoritmo 2 rispetto all'algoritmo 3 per piccoli campioni.

Come abbiamo detto a proposito del caso medio e del caso peggiore, in una utilizzazione reale dell'algoritmo può essere piú interessante il suo comportamento nel transitorio, se i

	50	100	150	200
A1	838	3334	7541	13474
A2	238	884	1991	3574
A3	149	413	766	1164
A4	109	260	451	673
A5	64	149	255	376

Fig. 53.

dati per cui viene utilizzato usualmente sono in genere in tale fascia. Perciò qualche volta può essere preferito un algoritmo a un altro che si comporta meglio del primo asintoticamente, ma peggio nel transitorio.

Si noti ancora che il metodo che abbiamo suggerito (eseguire l'algoritmo per un campione significativo di dati e verificarne il comportamento) può essere in generale faticoso o addirittura impossibile da attuare. Già nel caso del problema del massimo comun divisore le prove da fare per tutte le coppie dei primi n numeri sono n al quadrato. Per il problema di ordinare n numeri inizialmente non ordinati le prove relative a numeri scelti comunque tra i primi m interi sono m elevato alla n. Infatti per ogni numero abbiamo m scelte e i numeri sono in totale n. Il numero delle prove da fare cresce dunque in modo estremamente rapido al crescere di m ed n. Abbiamo quindi bisogno, accanto a questi metodi che possiamo chiamare empirici, di metodi che non richiedano la esecuzione dell'algoritmo. Come si vede, siamo in una situazione molto simile a quella vista per i metodi per la prova di correttezza.

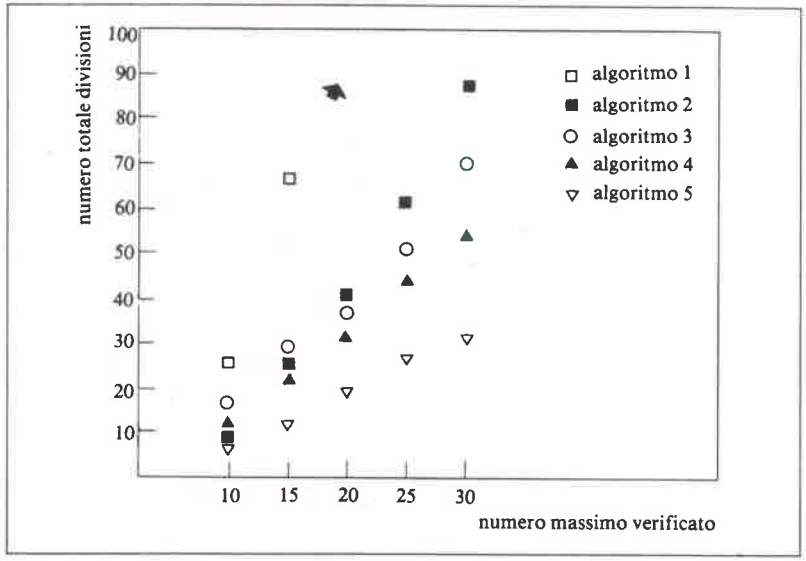


Fig. 54.

2. Misure di efficienza. I problemi dei quali in linea di principio siamo interessati ad automatizzare la risoluzione possono suddividersi in tre categorie:

1. problemi per i quali sono al momento noti un certo numero di algoritmi risolutivi;
2. problemi per i quali è noto che non esistono algoritmi risolutivi;
3. problemi per i quali non si sa se esistano o non esistano algoritmi risolutivi.

Nel capitolo precedente ci siamo interessati del secondo tipo di problemi; in questo capitolo siamo evidentemente interessati alla prima categoria.

Consideriamo dunque un problema per cui siano noti vari algoritmi risolutivi (vedi fig. 55).

Tali algoritmi differiscono in genere, e spesso sensibilmente, dal punto di vista della efficienza. Per poterli confrontare da tale punto di vista, abbiamo bisogno di precisare la definizione intuitiva di efficienza che abbiamo dato all'inizio del

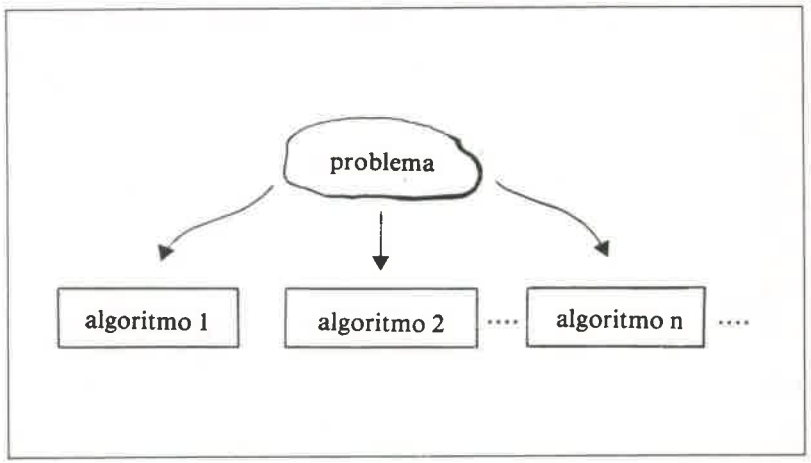


Fig. 55: Dato un problema a un dato stadio della conoscenza sono in generale noti vari algoritmi risolutivi.

capitolo, e fornirne una qualche misura. Parliamo ora appunto di misure di efficienza.

Le risorse il cui uso siamo interessati a limitare sono sostanzialmente di due tipi:

1. tempo impiegato per eseguire l'algoritmo;
2. spazio di memoria necessario per l'esecuzione.

I precedenti esempi si riferivano tutti a un confronto in termini del tempo impiegato.

Riguardo ai due problemi precedenti, non interessa tanto conoscere il tempo di esecuzione dell'algoritmo per uno specifico insieme di dati di ingresso, quanto ricavare una legge di comportamento generale dell'algoritmo. Ci interessa la misura del tempo necessario ad eseguirlo in funzione di una qualche misura o dimensione dei dati di ingresso. Tale dimensione ha lo scopo di caratterizzare ciò che possiamo chiamare la loro complessità.

Per il problema della somma di n numeri, tale dimensione può essere data evidentemente dal numero n dei numeri da sommare. Per il problema del massimo comun divisore può essere assunta come misura della complessità dei due dati il numero delle loro cifre.

Possiamo poi osservare che è poco interessante in generale (e probabilmente, in generale, molto complicato) calcolare il tempo reale effettivo di esecuzione: in molte situazioni ci accontentiamo di una sua valutazione approssimata, che sia però semplice da calcolare e allo stesso tempo significativa. Lo stesso vale per lo spazio fisico di memoria occupato. Possiamo allora concentrare l'attenzione sulle istruzioni piú eseguite dell'algoritmo, trascurando le altre, e assumere perciò come misura della efficienza (o, con termine piú usato, misura della complessità) dell'algoritmo il numero delle istruzioni piú eseguite, espresso in funzione della dimensione dei dati di ingresso.

In base poi a quanto emerso dagli esempi precedenti, possiamo distinguere tra misura nel caso medio e misura nel caso peggiore. Infine, poiché una valutazione della misura nel transitorio può essere in generale molto complessa da ottenere, possiamo fare solo riferimento a una misura asintotica, in cui appunto non si tiene conto del transitorio. Si dovrà perciò parlare di misure di complessità, a seconda del particolare contesto in cui si vuole effettuare la valutazione.

Si noti ancora che nelle precedenti misure abbiamo implicitamente assunto che a tutte le istruzioni sia associato lo stesso tempo di esecuzione. Come seconda approssimazione, potremmo associare alle varie operazioni primitive un peso proporzionale al tempo che si suppone necessario per l'esecuzione.

L'analisi di complessità richiede in generale sofisticati strumenti matematici. Nel seguito mostriamo alcuni casi in cui la semplice natura degli algoritmi permette di analizzare la complessità con considerazioni elementari.

Calcoliamo la complessità nel caso dell'algoritmo della somma di n numeri introdotto nel capitolo II che scriviamo in versione estesa nella figura 56.

Le istruzioni piú eseguite sono in questo caso quelle interne al ciclo, e sono eseguite un numero di volte pari a n. In questo caso semplice, poi, la misura nel caso peggiore e quella nel caso medio coincidono, perché il numero di operazioni eseguite dipende solo da n ma non dal particolare ingresso. Infine, comportamento nel transitorio e comportamento asintotico coincidono, per la stessa ragione. La misura di complessità ha dunque in questo caso valore n.

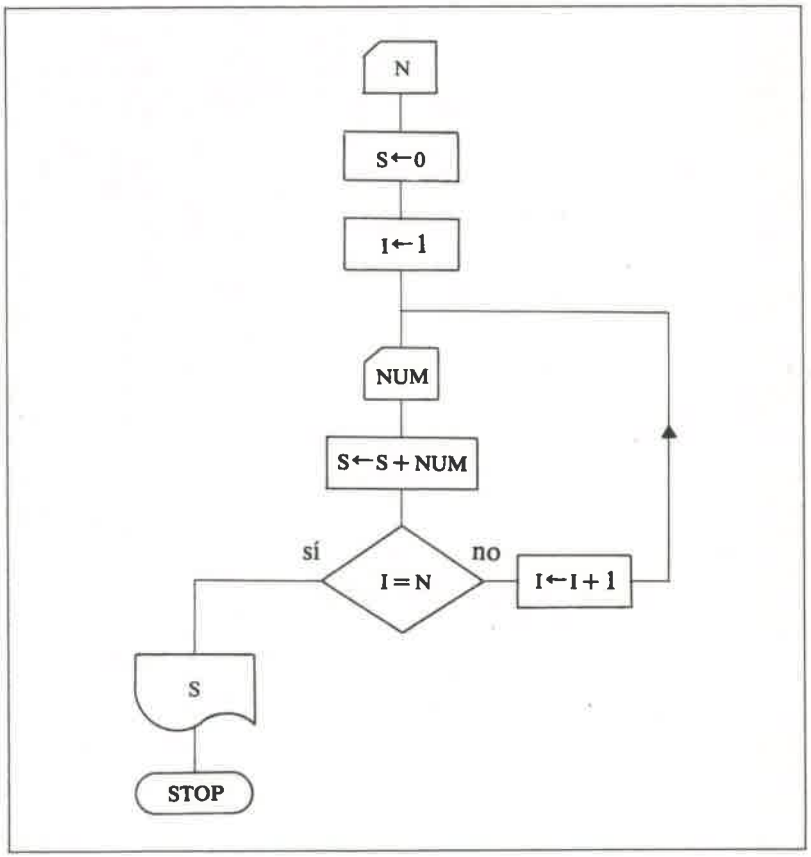
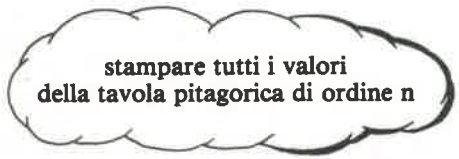


Fig. 56: Algoritmo per la somma di n numeri.

Consideriamo ora un secondo problema:



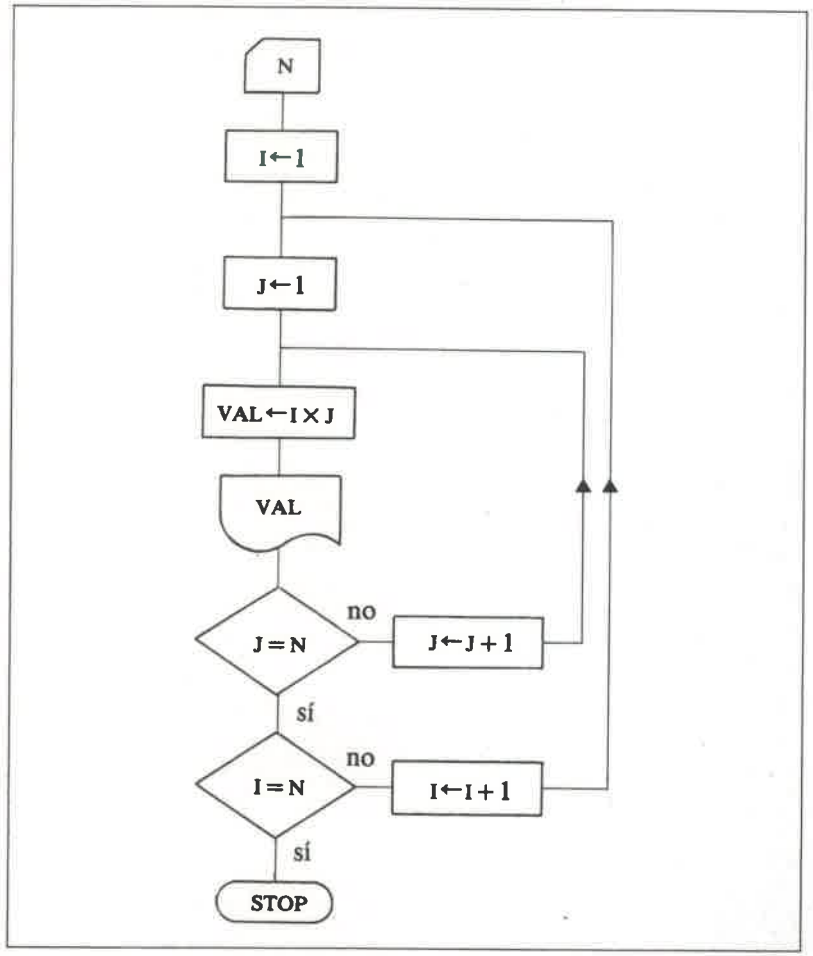
Con «ordine di una tavola pitagorica» intendiamo il massimo numero che compare nelle righe e colonne della tavola. Riportiamo per esempio nella figura 57 la tavola pitagorica di ordine 5.

Un algoritmo che, letto n, genera tutti i numeri della tavola pitagorica di ordine n compare nella figura 58. Il lettore

Fig. 57: Tavola pitagorica di ordine 5.

1	2	3	4	5
2	4	6	8	10
3	6	9	12	15
4	8	12	16	20
5	10	15	20	25

Fig. 58: Algoritmo per il calcolo della tavola pitagorica di ordine n.



1. Leggi n;
2. Leggi n numeri in memoria;
3. Somma gli N numeri e metti il risultato in SOMMA;
4. Stampa SOMMA;
5. FERMATI.

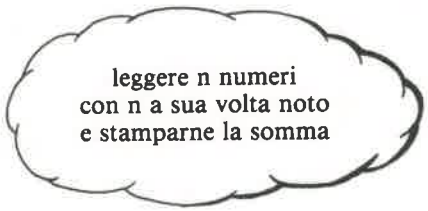
Fig. 59: Algoritmo 1 per la somma di n numeri.

può notare che questo è il primo algoritmo che presentiamo con due cicli nidificati (uno dentro l'altro); questo aspetto strutturale ne rende un po' piú impegnativa la comprensione. Si noti che per ogni valore di I tra i ed n viene eseguito n volte il ciclo piú interno, con J che assume a sua volta a ogni esecuzione del ciclo i valori da i a n. Dunque, vengono dapprima generati i multipli di 1, poi quelli di 2, e cosí via.

Per questo algoritmo è ragionevole assumere come dimensione dei dati di ingresso il valore di N, che, ricordiamo, è l'ordine della tavola. Le istruzioni piú eseguite si trovano ancora una volta dentro il ciclo piú interno (in questo caso dentro a due cicli) e quindi la misura di complessità (del tempo di esecuzione) ha come valore n^2.

Vediamo ora un semplice esempio in cui siamo interessati a confrontare lo spazio di memoria.

Partiamo dal problema:



Di tale problema esistono almeno due algoritmi risolutivi (vedi figg. 59 e 60).

1. Leggi n;
2. Metti 0 in SOMMA;
3. Per N volte esegui:
   3.1. Leggi un numero
   3.2. Sommalo a SOMMA e metti il risultato in SOMMA;
4. Stampa SOMMA;
5. FERMATI.

Fig. 60: Algoritmo 2 per la somma di n numeri.

Per il primo possiamo dire che per memorizzare gli n valori da sommare sono necessarie almeno n parole di memoria.

Per il secondo, solo uno degli n numeri è letto ogni volta e memorizzato, e dunque è necessaria una sola cella.

Dal punto di vista della occupazione di memoria il secondo algoritmo è dunque piú efficiente del primo poiché invece di memorizzare prima tutti gli n valori e poi leggerli, ne legge uno per volta e lo somma subito dopo. Per quanto riguarda la complessità (relativamente alla occupazione di memoria) l'algoritmo 1 ha complessità costante, l'algoritmo 2 ha complessità n.

I casi esaminati sono estremamente semplici da analizzare. All'aumentare della complessità strutturale dell'algoritmo diventa in genere un compito molto arduo fornire misure di complessità. Gli stessi problemi del massimo comun divisore e dei numeri primi, che abbiamo analizzato in questa sezione, sono molto complessi da analizzare. Lo studio dei metodi per valutare le misure di complessità degli algoritmi è ancora uno dei campi di ricerca piú aperti della informatica.

3. L'efficienza è un falso problema? Il problema della efficienza degli algoritmi è cruciale nelle applicazioni in tempo reale, in cui la risposta dell'algoritmo non può farsi attendere oltre un certo tempo, perché altrimenti diventa inutile. A parte questi problemi, l'individuazione di algoritmi efficienti

classe di complessità	valore di n 10	20	30	40	50	60
n	0,00001 sec	0,00002 sec	0,00003 sec	0,00004 sec	0,00005 sec	0,00006 sec
n^2	0,0001 sec	0,0004 sec	0,0009 sec	0,0016 sec	0,0025 sec	0,0036 sec
n^3	0,001 sec	0,008 sec	0,027 sec	0,064 sec	0,125 sec	0,216 sec
n^5	0,1 sec	3,2 sec	24,3 sec	1,7 min	5,2 min	13 min
2^n	0,001 sec	1 sec	18 min	12,7 giorni	35,7 anni	366 secoli
3^n	0,059 sec	58 min	6,5 anni	3855 secoli	2×10^8 secoli	10^{13} secoli

Fig. 61: Confronto del tempo di esecuzione di alcune classi di complessità in funzione di alcune dimensioni di ingresso.

è stato nel passato visto da molti come il problema principale da risolvere nel progetto di algoritmi e programmi. Si aveva molta considerazione per i programmatori che con qualche trucco di programmazione riuscivano a diminuire di qualche decimo di secondo il tempo di esecuzione di un programma.

Ora la tecnologia sta facendo passi da gigante e dunque il vero problema non è tanto limitare il piú possibile il tempo di calcolo o l'occupazione di memoria, quanto trovare rapidamente algoritmi che abbiano misure di complessità ragionevoli. Per precisare meglio cosa si possa intendere per complessità ragionevole, si osservi la tabella nella figura 61.

Nella tabella viene mostrato per alcuni dei valori di complessità (o classi di complessità) piú significativi e frequenti, come varia il tempo effettivo di calcolo al variare della dimensione dei dati in ingresso, assumendo un tempo di esecuzione di un microsecondo (10^{-6} secondi) per la singola istruzione.

Dalla tabella si vede chiaramente che i valori di complessità piú critici sono quelli esponenziali, quelli in cui cioè la dimensione dell'ingresso compare come esponente nel valore di complessità.

	massima dimensione di ingresso di un problema risolubile in un'ora		
classe di complessità	T con l'attuale tecnologia	T con un calcolo 100 volte piú veloce	T con un calcolo 1.000 volte piú veloce
n	N1	100 N1	1000 N1
n^2	N2	10 N2	3 1,6 N2
n^3	N3	4,6 N3	10 N3
n^5	N4	2,5 N4	4 N4
2^n	N5	N5 + 6,6	N5 + 9,9
3^n	N6	N6 + 4,2	N6 + 6,3

Fig. 62: Effetto del miglioramento della tecnologia sul tempo di esecuzione degli algoritmi per varie classi di complessità.

Riferendoci ancora alla vertiginosa evoluzione della tecnologia, potremmo pensare che anche per la classe degli algoritmi esponenziali sia solo questione di tempo, e che di qui a pochi anni anche il piú complicato algoritmo possa essere eseguito su un calcolatore superpotente nel giro di pochi secondi.

La tabella della figura 62 scoraggia questa prospettiva. Nella tabella viene mostrato come vari per le stesse classi di complessità della figura 60 la piú grande dimensione di ingresso di un problema risolvibile in un'ora se avessimo a disposizione un calcolatore 100 e 1000 volte piú veloce di quelli attuali. Anche qui si vede che per le classi esponenziali il miglioramento è assolutamente trascurabile.

Gli algoritmi con complessità esponenziale sono dunque, e rimarranno sempre, la autentica bestia nera dei calcolatori.

VI. IL PROGETTO DI ALGORITMI E PROGRAMMI

Introduzione - Uso di un'agenda e di un elenco telefonico - Ricerca del cammino minimo tra due nodi di un grafo - Conclusioni

1. Introduzione. In questo capitolo approfondiamo l'aspetto progettuale dell'informatica. Anche se fino a ora abbiamo utilizzato problemi e algoritmi per la loro risoluzione di struttura piuttosto semplice, il lettore si sarà reso conto che il progetto di un algoritmo è un'attività in generale assai complessa, che richiede molta creatività e capacità analitica.

Nel progetto di un algoritmo vi sono due tipi di attività che si influenzano a vicenda e che svolgono un ruolo decisivo: la scelta delle strutture di dati utilizzate; la scelta delle operazioni da effettuare su tali strutture, e l'ordine con cui effettuare tali operazioni.

Fino a ora il problema della scelta delle strutture di dati non è emerso poiché abbiamo scelto problemi (e algoritmi) che esprimevano i dati per mezzo di variabili ognuna delle quali rappresentava un singolo numero intero. Se però, per esempio, si torna a considerare l'algoritmo della fig. 59 che abbiamo riprodotto in alto si vede che per descriverlo nel linguaggio dei grafi di flusso presentato nel capitolo II avremmo bisogno di un numero di variabili pari a n tutte diverse tra loro. Questa scelta è chiaramente pesante e non sopportabile.

Nei linguaggi ad alto livello, perciò, accanto a variabili che rappresentano valori singoli, è possibile definire e usare variabili che descrivono strutture di dati, cioè insiemi di dati in relazione tra loro. Nasce allora il problema di individuare dei metodi che permettano di semplificare e orientare da una

1. Leggi n;
2. Leggi n numeri in memoria;
3. Somma gli N numeri e metti il risultato in SOMMA;
4. Stampa SOMMA;
5. FERMATI.

Algoritmo 1 per la somma di n numeri.

parte la scelta delle strutture di dati che piú efficacemente ed efficientemente rappresentano i dati di ingresso del problema e i risultati intermedi e finali, e dall'altra, come detto, di orientare la scelta delle operazioni da effettuare e l'ordine con cui farle.

Vedremo che può essere conveniente suddividere questo processo di scelta in due passi, nel primo dei quali procedere alla sintesi di un algoritmo prescindendo dalla scelta delle 'strutture di dati', e nel secondo scegliere le strutture tra quelle disponibili nel linguaggio programmativo a disposizione ed esprimere l'algoritmo in termini di tali strutture. Potremo cosí precisare e giustificare la distinzione tra i concetti di algoritmo e programma fatta nel capitolo II.

In questo capitolo, ancor piú che negli altri, conduciamo l'indagine mediante una analisi di casi. Esamineremo cioè alcuni casi significativi di problemi, e durante l'analisi metteremo in evidenza gli aspetti dei metodi di progetto di algoritmi cui siamo interessati.

2. Uso di un'agenda e di un elenco telefonico. Riconsideriamo i problemi legati all'uso di un archivio per ricordare e ritrovare numeri di telefono. Abbiamo già notato che per questo problema nella vita reale vengono usate due diverse tecnologie, la *agenda telefonica* e l'*elenco telefonico*. Esaminiamone l'organizzazione in dettaglio.

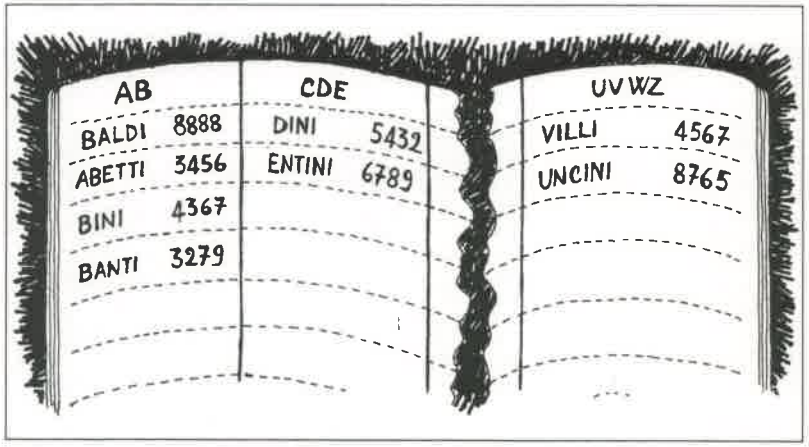


Fig. 63: Struttura di una agendina telefonica.

L'agenda telefonica. Prendiamo in esame quelle agendine telefoniche tascabili che si possono acquistare presso i cartolai, ovvero vengono regalate dai settimanali ad alta tiratura verso la fine dell'anno.

Nella figura 63 ne abbiamo schematizzato la struttura.

Assumeremo per semplicità nel seguito che le persone ai cui numeri di telefono siamo interessati siano sempre riferite nell'agendina tramite i cognomi. Cognomi e numeri di telefono sono raggruppati a seconda delle iniziali dei cognomi. Per esempio nella prima pagina dell'agendina della figura 63 compaiono tutti i cognomi la cui prima lettera è A o B, nella seconda pagina i cognomi la cui prima lettera è C, D, E, e cosí via.

Esaminando le singole pagine si nota poi che all'interno di ogni gruppo i cognomi compaiono in ordine casuale. L'agenda è stata riempita all'inizio dell'anno senza badare all'ordine alfabetico rigoroso. Via via che dobbiamo inserire un nuovo cognome lo mettiamo dove c'è posto. Questo è il modo usuale con cui le agendine vengono riempite.

Le operazioni tipiche che effettuiamo nel corso dell'anno sull'agendina sono:

1. ricerca di un numero di telefono, dato il cognome;
2. inserimento nuovo cognome e numero di telefono;
3. cancellazione di un cognome e numero di telefono.

Vogliamo mostrare ora come la scelta della struttura descritta influenzi gli algoritmi di ricerca e inserimento.

Un possibile algoritmo per il problema della ricerca è mostrato nella figura 64.

A prima vista il precedente algoritmo sembra corrispondere al metodo che adoperiamo normalmente quando consultiamo un'agenda. Eppure, se osserviamo con attenzione il nostro comportamento quando usiamo un'agenda, ci accorgiamo che la fase di ricerca della pagina è in genere effettuata con un metodo piú efficiente. Se cerchiamo un cognome che comincia per S, non cominciamo dalla prima pagina, quella che contiene i cognomi che cominciano per A e B, proseguendo poi, sequenzialmente, fino alla S. Sappiamo che la S sta verso la fine dell'alfabeto e dunque apriamo l'agenda verso la fine, andando poi, a seconda dei casi, un po' piú avanti o un po' piú indietro, finché non troviamo la pagina richiesta.

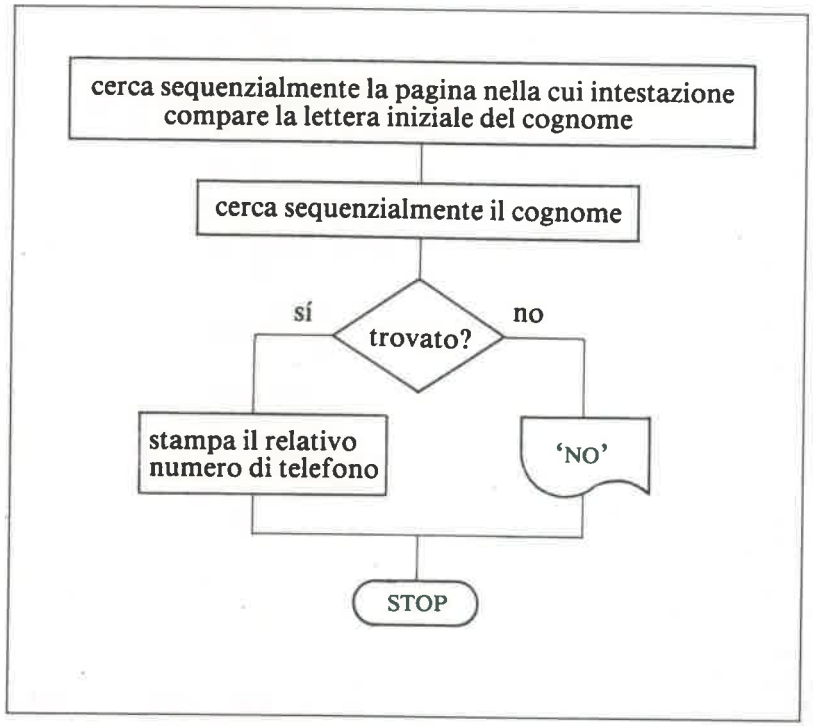


Fig. 64: Primo algoritmo per la ricerca di un numero di telefono in un'agenda.

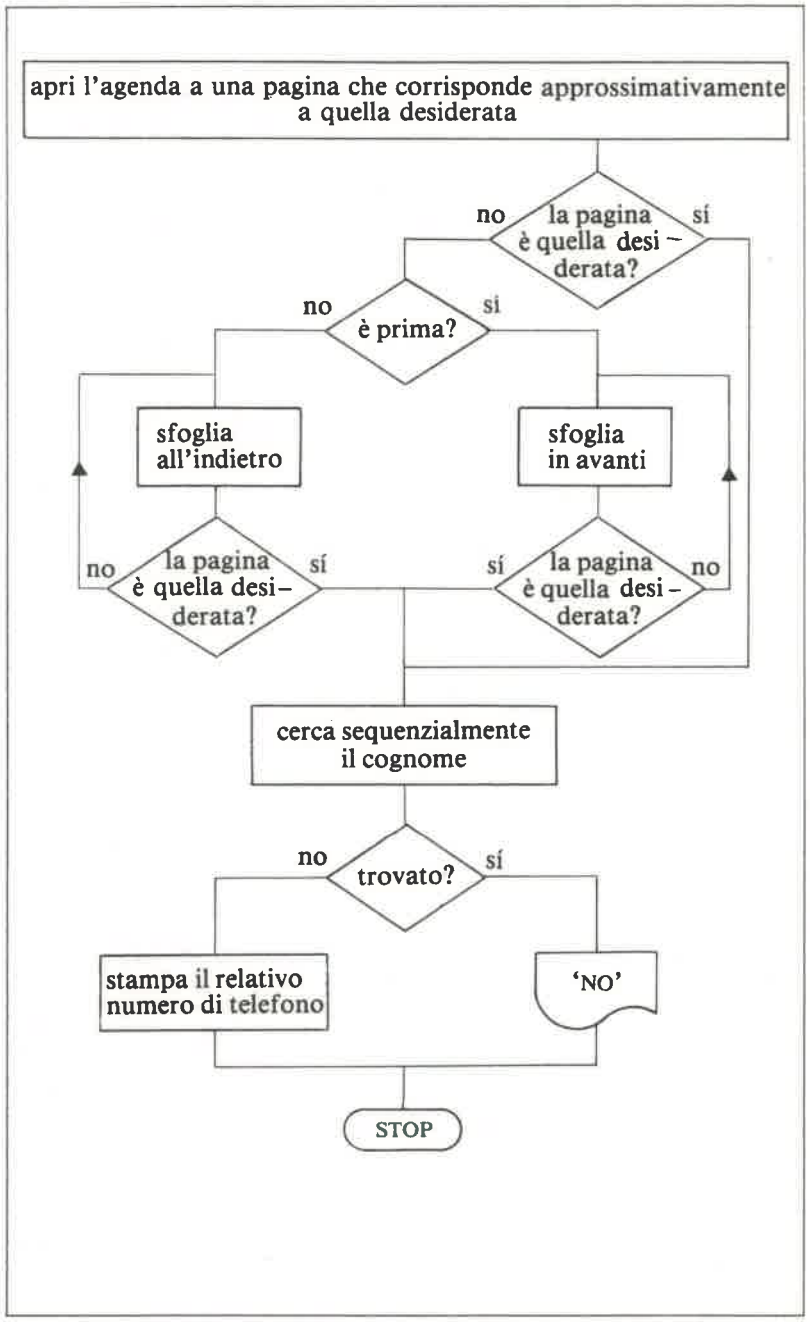


Fig. 65: Nuovo algoritmo per la ricerca di un numero di telefono in un'agenda.

Possiamo perciò riformulare l'algoritmo in una versione piú efficiente, mostrata nella figura 65.

Si noti che nel precedente algoritmo abbiamo usato un differente grado di dettaglio nell'esplicitare le attività della ricerca della pagina e della ricerca del cognome nella pagina, e siamo stati piú precisi nella prima attività.

Ancora, è da notare che nell'algoritmo per tre volte compare la domanda:

La pagina è quella desiderata?

Sarebbe possibile unificare le tre domande in una sola, semplificando cosí la struttura dell'algoritmo? La risposta è sí, e un algoritmo che rispetta tale proprietà è mostrato nella figura 66.

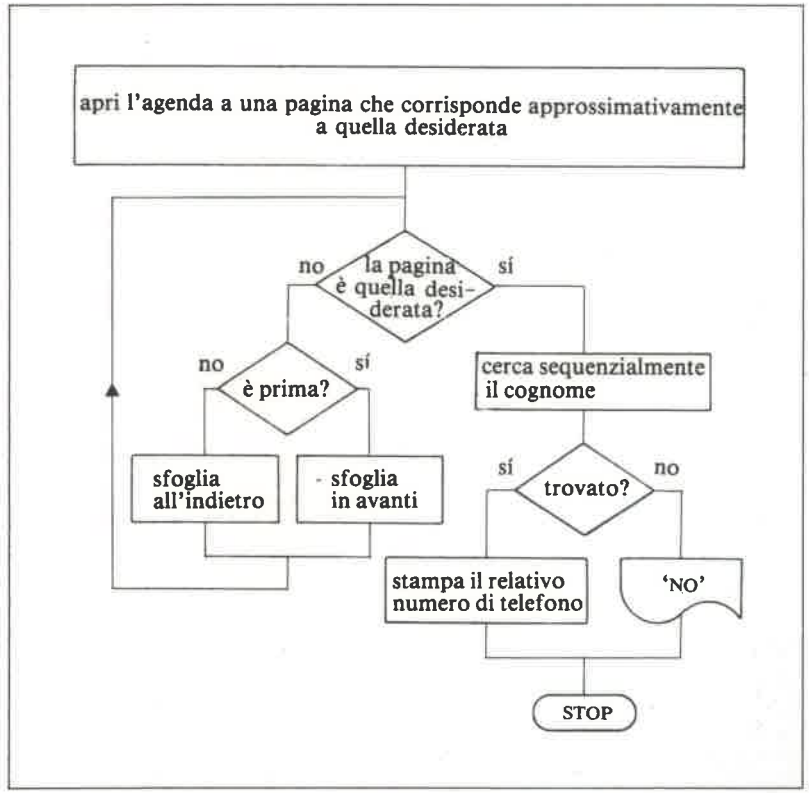


Fig. 66: Un algoritmo equivalente al precedente, con struttura piú semplice.

Il nuovo algoritmo è equivalente al precedente, nel senso che a partire dagli stessi dati in ingresso fornisce la stessa risposta.

Tuttavia, anche se ha una struttura piú semplice, è meno chiaro e meno efficiente del precedente. È meno chiaro perché per eliminare la domanda sulla pagina desiderata propone ogni volta una domanda piuttosto artificiosa: è prima? È meno efficiente perché a ogni nuova pagina dobbiamo ora rispondere a due domande.

La chiarezza è una qualità essenziale degli algoritmi, spesso purtroppo sottovalutata. Quanto piú è facilmente comprensibile la strategia che l'algoritmo adotta, tanto piú è semplice individuare e correggere eventuali singoli passaggi sbagliati. Nello stesso tempo di fronte a variazioni nelle richieste del problema sarà piú facile apportare le modifiche necessarie.

Passiamo ora a esaminare il problema dell'inserimento di un nuovo cognome. L'algoritmo che tipicamente usiamo è, in prima approssimazione, quello della figura 67.

Con la dizione «cerca la pagina» abbiamo indicato in forma sintetica l'algoritmo di ricerca di pagina descritto in dettaglio nelle figure precedenti.

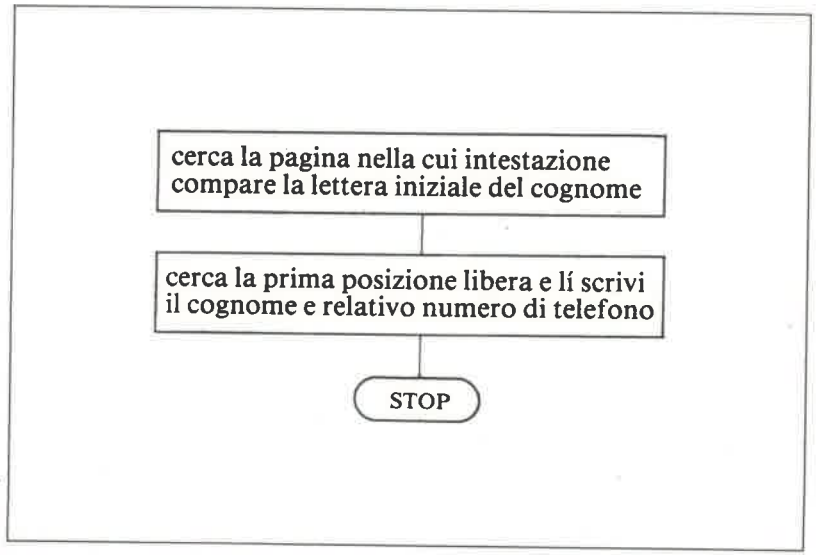


Fig. 67: Prima versione di un algoritmo di inserimento di un numero in un'agenda.

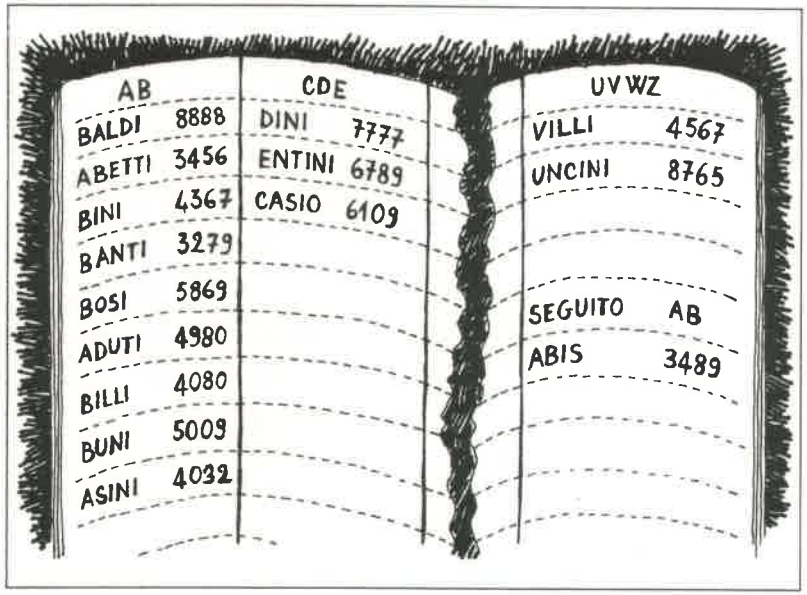


Fig. 68: Esempio di agenda riempita secondo i nuovi criteri.

L'algoritmo della figura 67 non prevede il caso che non vi sia spazio per il nuovo cognome, ossia che la pagina corrispondente sia già tutta occupata. Per poter tener conto di questo caso, dobbiamo complicare un po' l'organizzazione dei dati della agenda. Possiamo per esempio partire dalla ipotesi che:

1. nel corso dell'anno accada un certo numero di volte che, dovendo aggiungere un nuovo numero di telefono, la pagina risulti invece piena;
2. esista un certo numero di pagine che per tutto l'anno rimane piuttosto vuoto.

Partendo da queste ipotesi possiamo organizzare l'agenda come segue: ogni qualvolta che una pagina è tutta piena, si cerca una nuova zona, che diventa l'estensione della pagina ormai piena (vedi fig. 68).

La nuova organizzazione permette in generale di rappresentare piú cognomi della precedente, a prezzo di una maggiore complicazione dell'algoritmo di ricerca.

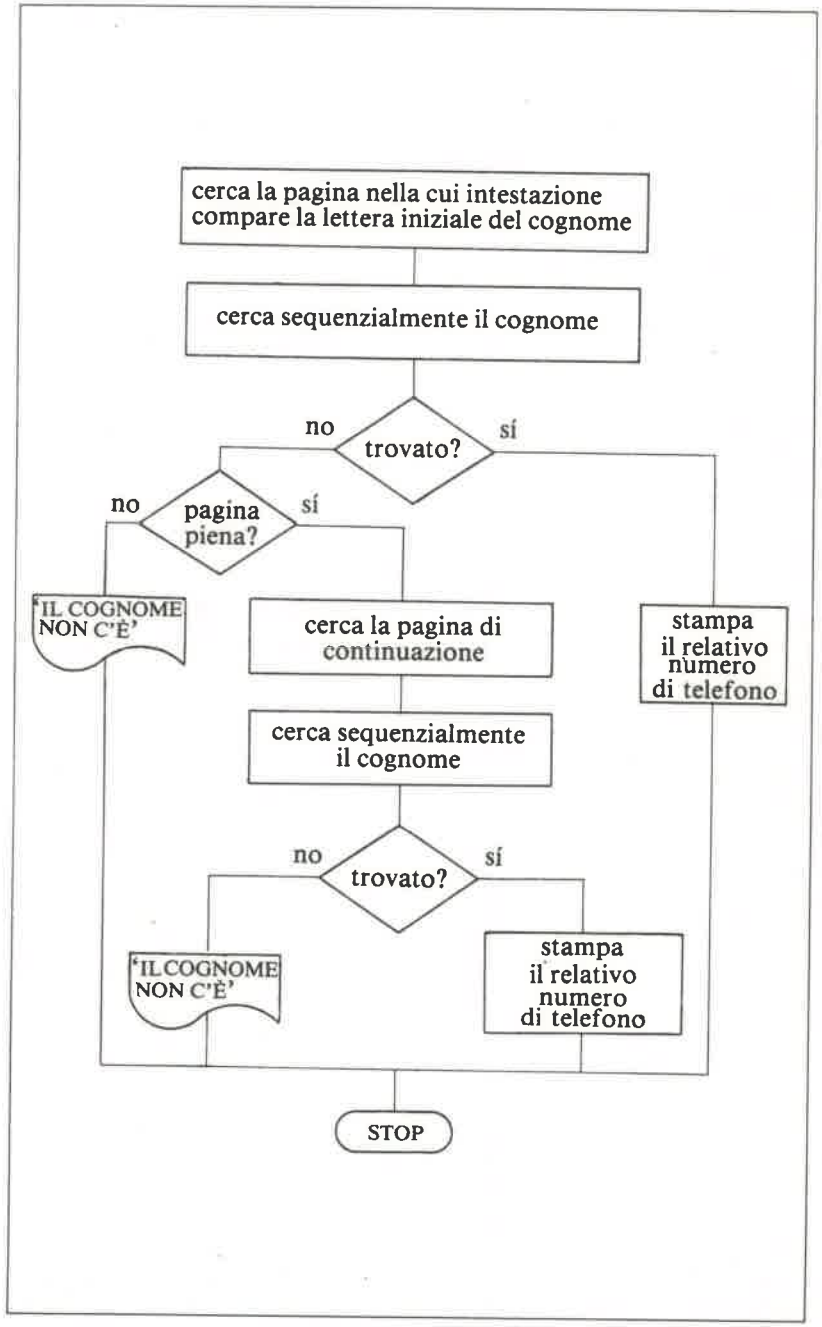


Fig. 69: Algoritmo di ricerca modificato.

Un algoritmo di ricerca, con la nuova organizzazione, è quello della figura 69.

L'esempio che abbiamo esaminato mostra che:

1. dato un problema esistono vari modi di organizzare i dati relativi al problema e
2. gli algoritmi risolutivi del problema dipendono strettamente dalla struttura scelta per i dati.

Queste importanti osservazioni sono confermate dall'esame della seconda struttura di dati tipicamente usata per rappresentare cognomi e numeri di telefono, l'elenco telefonico.

L'elenco telefonico. L'elenco telefonico, cosí come l'agenda, rappresenta informazioni riguardanti persone e loro numeri di telefono. Poiché, al contrario dell'agenda, nell'elenco sono rappresentati i numeri di telefono di tutte le persone di una città (meglio, di quasi tutti: un certo numero di persone, per ragioni di tranquillità e sicurezza, non permette che il suo numero sia riportato nell'elenco) per poter identificare una persona in una città è necessario riportare oltre al cognome anche il nome e l'indirizzo.

Cosí come per l'agenda, esaminiamo ora il modo in cui sono tipicamente organizzati i dati in un elenco telefonico.

In un elenco i dati sono ordinati per cognome; tra le persone con cognomi uguali, sono ordinati per nome. Inoltre a ogni pagina è associata una intestazione formata da tre lettere che costituiscono le prime tre lettere del primo cognome della pagina:

CAS

CASIROLI Ing. Rosario, 15 v. Tabarrini ---- 794 50 11
CASLI Isce, 35 p. Aruleno Celio Sabino ---- 747 26 10
CA.SMA.RA. Cappelleria, Smacchiatoria
Rammendi 102/a p. S. Bernardo ---------- 46 14 91
CASMIRO Paolina, 92 v. Pian 2 Torri ----- 528 32 30
CASNEDI dr. Maria Sofia, 16 v. Fornaci --- 63 58 16
CASO Angela, 25 v. Torrita Siena ---------- 523 84 48
» dr. Antonio, 75 v. Mona ---------- 591 70 63
» Arsico, 18 v. Pienza ---------- 810 64 01

Questa informazione supplementare, insieme all'ordinamento dei cognomi, facilita notevolmente l'operazione di ricerca di un numero di telefono. Riportiamo un possibile algoritmo per tale operazione nella figura 70.

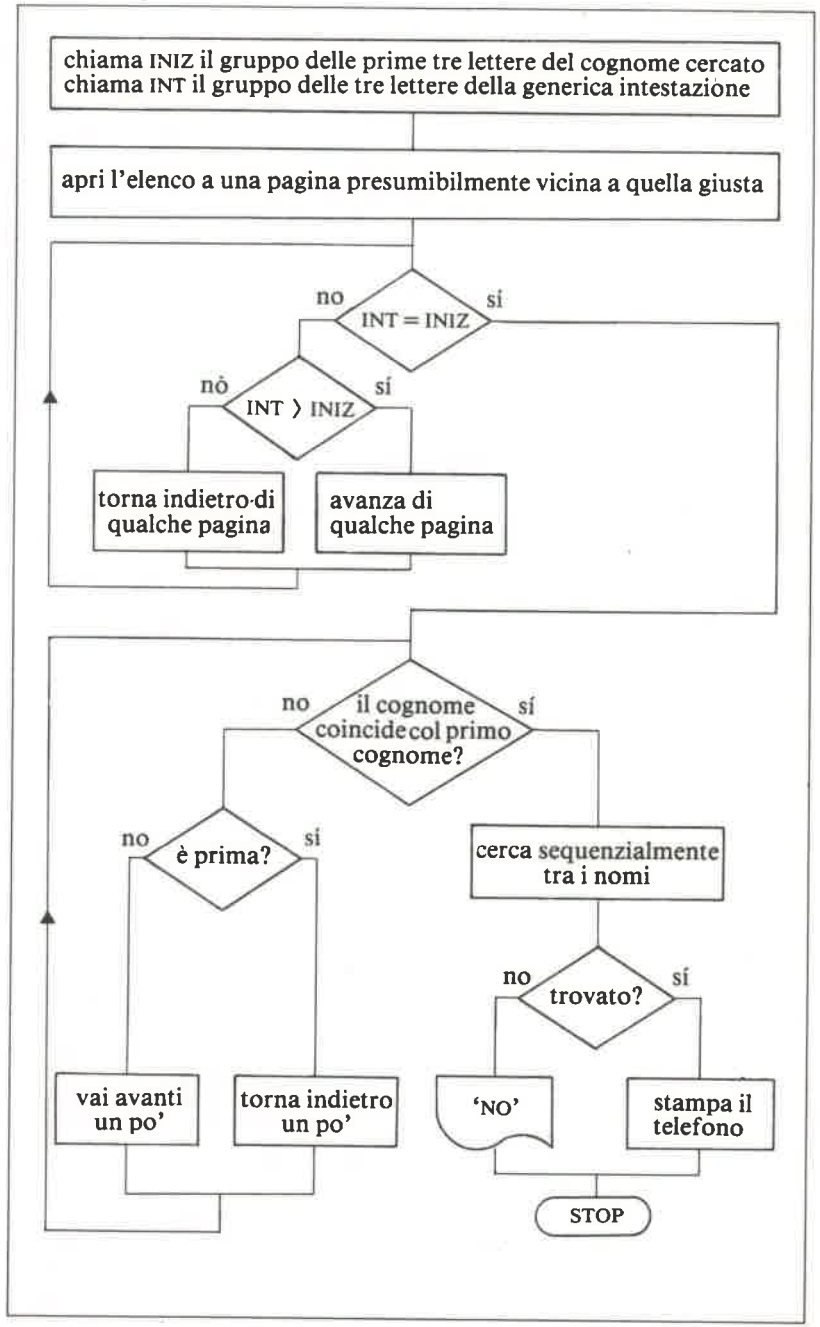


Fig. 70: Algoritmo di ricerca di un numero di telefono in un elenco telefonico.

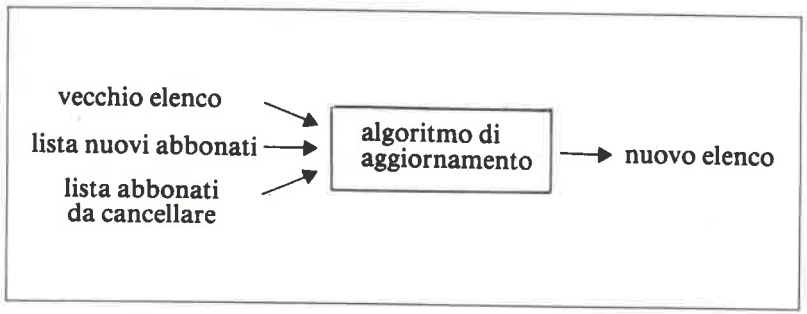


Fig. 71: Dati di ingresso e di uscita all'algoritmo di aggiornamento annuale dell'elenco telefonico.

L'algoritmo presenta una struttura piú complicata dell'analogo per l'agenda, ma è certamente molto piú efficiente in termini di tempo di esecuzione. Nella ricerca della pagina, infatti, le pagine vengono sfogliate a gruppi, e nella ricerca del cognome si procede per campione, e si risparmia cosí un bel po' di tempo.

La maggiore efficienza della operazione di ricerca di un numero di telefono (operazione realizzata migliaia di volte ogni giorno in una grande città) è naturalmente ottenuta a prezzo di una maggiore complicazione della struttura e di altre operazioni, come quella di aggiunta di un nuovo cognome.

L'operazione di aggiunta di un nuovo cognome, insieme a quella di cancellazione, viene fatta una volta per tutte in una certa epoca dell'anno in cui si prepara l'elenco dell'anno nuovo. In questo caso l'algoritmo ha come dati di ingresso (vedi fig. 71) il vecchio elenco e due liste relative agli abbonati da inserire per la prima volta e quelli da cancellare.

Dagli esempi mostrati emerge una importante caratteristica del progetto di algoritmi: la struttura scelta per rappresentare i dati coinvolti dal problema e la struttura dell'algoritmo si influenzano profondamente.

Colui che progetta un algoritmo, il progettista, deve conoscere a fondo caratteristiche e proprietà di un vasto numero di strutture di dati per poter comprendere quali strutture possono essere significative per una data classe di problemi e quali tra queste scegliere in base alle caratteristiche specifiche di un dato problema.

3. Ricerca del cammino minimo tra due nodi di un grafo. Quando si deve affrontare il progetto di un algoritmo per un problema complesso, conviene scomporre il problema iniziale in una serie di problemi via via piú semplici, esprimendo tali problemi in maniera ancora astratta, per esempio usando il linguaggio naturale. Con questa strategia generale viene garantita una maggiore semplicità e allo stesso tempo una maggiore affidabilità al progetto stesso.

In questo paragrafo mostreremo una applicazione di tale strategia generale. In particolare vogliamo mostrare che nel progetto di un algoritmo è possibile descrivere l'algoritmo senza tener conto, almeno inizialmente, della scelta delle strutture di dati che è possibile definire nel particolare linguaggio a disposizione. Tale scelta può essere fatta successivamente, dopo che ci si è fatti una chiara idea della struttura dell'algoritmo espresso in termini appunto ancora astratti, indipendenti dal linguaggio programmativo.

In questo contesto dunque i concetti di algoritmo e programma sono ben distinti.

Si consideri il seguente problema:

La rete di strade di cui si parla nel problema può essere la rete delle strade di una città, ovvero di una regione, di una nazione, di un continente, o della intera Terra. Il problema che vogliamo indagare è un problema reale per molti di noi nella vita quotidiana, quando la mattina dobbiamo raggiungere il posto di lavoro usando un nostro mezzo di trasporto.

Ai fini del nostro problema possiamo rappresentare la rete stradale di una città mediante un grafo (vedi fig. 72) costituito da nodi e rami; i nodi rappresentano gli incroci tra le strade (ciascuno identificato da un numero), i rami rappresentano i segmenti di strada compresi tra due incroci: a ogni ramo è associata la distanza del corrispondente segmento, espressa in metri.

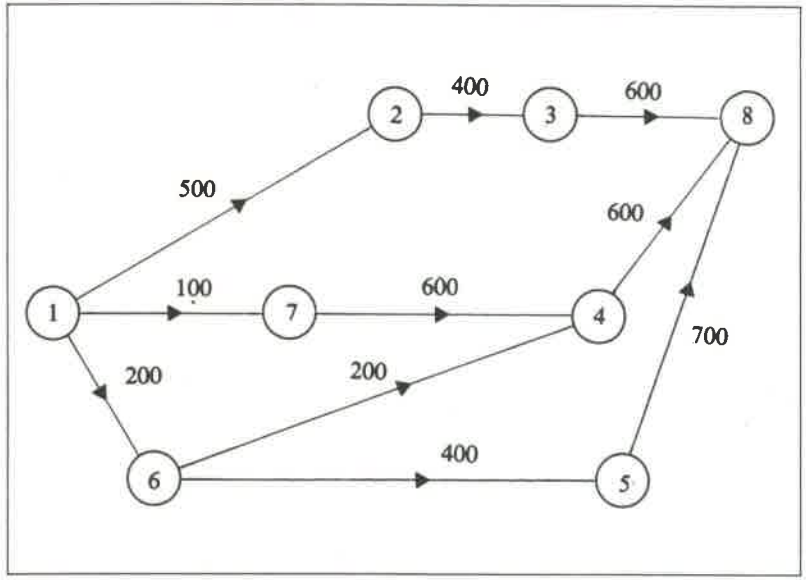


Fig. 72: Esempio di rete stradale schematizzata come grafo.

Supponiamo nel seguito di risolvere il problema cercando i cammini a partire dal nodo 1.

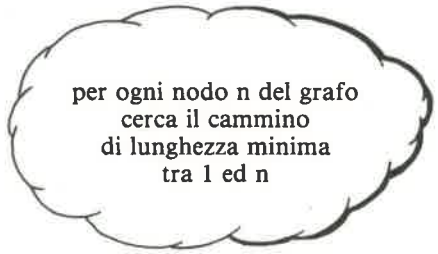
Nella figura 73 riportiamo l'elenco dei cammini con lunghezza minima dal nodo 1 a tutti gli altri nodi e le loro lunghezze, in ordine di lunghezza crescente.

	cammino	lunghezza
1.	1 7	100
2.	1 6	200
3.	1 6 4	400
4.	1 2	500
5.	1 6 5	600
6.	1 2 3	900
7.	1 6 4 8	1000

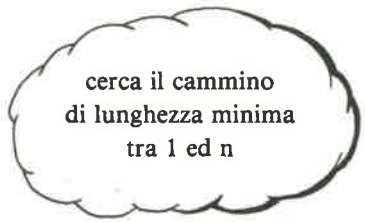
Fig. 73: Elenco dei cammini.

Possiamo a questo punto passare a ragionare su un algoritmo risolutivo del problema posto nell'esempio. Dobbiamo cercare qualche tecnica che ci permetta di scomporre il problema in sottoproblemi, ognuno dei quali sia piú facile da risolvere del problema originario.

Una prima versione di un algoritmo tentativo può essere la seguente:



Possiamo dire effettivamente di aver ridotto il problema di partenza a un problema un po' piú semplice e cioè



Un algoritmo per tale problema può per esempio trovare tutti i cammini tra 1 ed n e tra essi scegliere il minimo. Piuttosto che doversi ricordare tutti i cammini tra 1 ed n potremmo generarli uno per volta e confrontarli con quello che fino a quel momento era il minimo. Resta in questo caso il problema di generare ogni singolo cammino tra 1 ed n. Un algoritmo stupido che è in grado di fare ciò può, per esempio, generare tutte le potenziali sequenze di nodi tra 1 ed n scegliendo in tutti i modi possibili nell'insieme dei nodi, e verificare successivamente per ogni cammino potenziale generato che sia veramente un cammino tra 1 ed n.

Mettendo insieme in maniera piú precisa tutte le precedenti idee otteniamo la formulazione di figura 74.

1. Dai alla variabile MAX un valore molto grande.
2. Per ogni possibile sequenza di nodi scelta nel grafo:
   2.1. Se la sequenza è un possibile cammino tra 1 ed n allora
      2.1.1. Se la sua lunghezza è minore di MAX allora
         2.1.1.1. Aggiorna MAX con il nuovo valore
         2.1.1.2. Ricordati la nuova sequenza come potenziale minima.

Fig. 74: Un tentativo di algoritmo per il problema del cammino minimo fra due nodi.

Ragionando un momento su questo algoritmo ci accorgiamo che abbiamo intrapreso una strada piuttosto disastrosa dal punto di vista della efficienza. Infatti tutte le possibili sequenze di nodi scelte tra k sono 2^k, numero che cresce molto rapidamente all'aumentare di k, facendo ricadere l'algoritmo in una di quelle classi «complicate» che abbiamo individuato nel capitolo V. Dunque occorre sospendere la ricerca e cercare qualcosa di meglio.

Per fortuna qualcosa di meglio esiste ed è un algoritmo molto noto, detto 'algoritmo di Dijkstra', dal nome dell'autore.

Vogliamo qui di seguito introdurre l'algoritmo di Dijkstra per mostrare al lettore quanto sofisticato debba essere a volte il ragionamento da fare per riuscire a generare algoritmi veramente efficienti.

La strategia base dell'algoritmo di Dijkstra consiste nell'individuare le lunghezze dei cammini minimi in ordine di lunghezza crescente. Partiamo inizialmente dal nostro esempio e cerchiamo di calcolare «a mano», raffinando man mano il ragionamento e allo stesso tempo abbozzando un algoritmo, l'insieme delle lunghezze dei percorsi minimi, in ordine di lunghezza crescente.

Cerchiamo anzitutto quale nodo può essere raggiunto dal nodo 1 col percorso piú breve: otterremo in tal modo la lunghezza piú breve, rispettando cosí la strategia esposta. Il

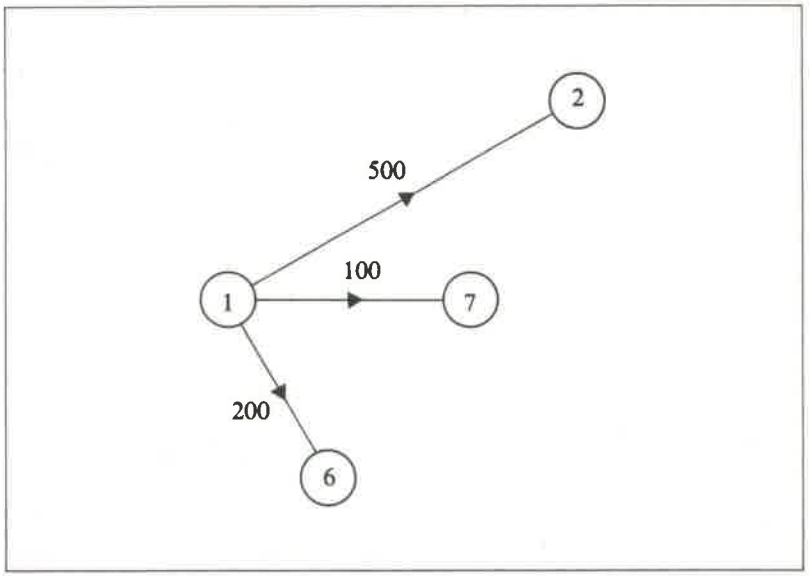


Fig. 75: Cammini coinvolti nella scelta del primo nodo.

nodo dovrà evidentemente essere scelto tra i nodi direttamente collegati con 1 (vedi fig. 75).

Come scegliere il nodo col percorso piú breve nell'insieme {2, 6, 7}? Basta evidentemente cercare quello con distanza minima da 1, e cioè 7.

Occorre ora cercare un nuovo nodo; è facile mostrare che vale la seguente proprietà:

Fatto. Il nuovo nodo va cercato tra i nodi candidati rimasti (nel nostro caso {2, 6}), ovvero tra i nodi adiacenti al nodo scelto (7) non ancora candidati o scelti (4).

Proviamo il fatto. Il nodo 4 va certamente messo nell'insieme dei candidati, perché potrebbe avere una distanza tale da 7 che la somma delle distanze dei rami tra 1 e 4 è ancora minore delle distanze di 2 e 6 da 1.

Assumiamo ora per assurdo che esista un altro nodo, diciamo k, che non è adiacente a 7 e che ha distanza da 1 minore delle distanze in {2, 6, 4}. Chiaramente tutti i nodi nel percorso tra 1 e k hanno distanza minima da 1 minore della distanza (minima) tra 1 e k, e dunque in base alla strategia enunciata devono essere già stati scelti. Ma l'unico nodo finora scelto è 7, e noi abbiamo escluso che 7 sia in un percorso tra 1 e k. Questo argomento è valido in generale, e potrà essere utilizzato in ogni passo di scelta.

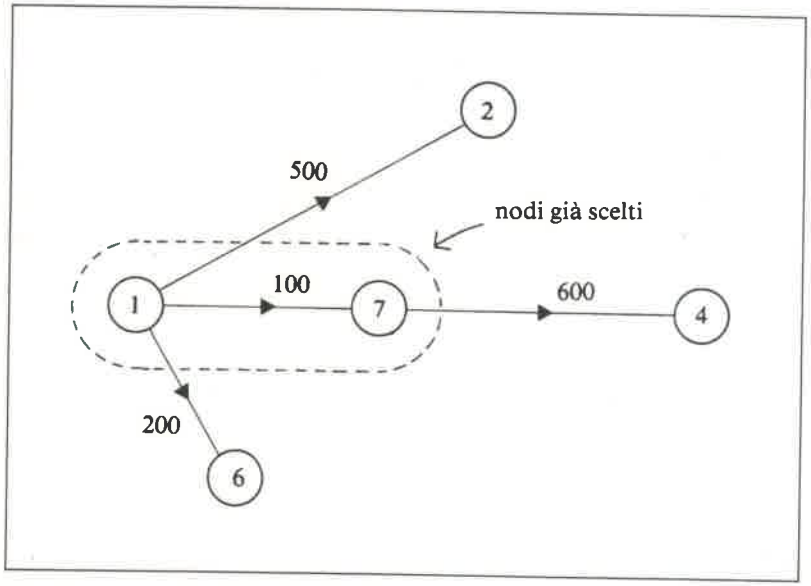


Fig. 76: Cammini coinvolti nella scelta del secondo nodo.

Resta ora il problema di scegliere un nodo in {2, 4, 6}.

Bisogna notare che, al momento attuale, non abbiamo nessuna informazione sulle distanze dei cammini tra 1 e 4. È facile vedere che, ai fini del confronto, basta calcolare la distanza tra 1 e 4 attraverso il cammino che passa dal nodo appena scelto, cioè 7 (vedi fig. 76). E questo in base a un ragionamento analogo al precedente: se c'è un cammino piú breve tra 1 e 4 che può portare alla scelta di 4 dovrà passare per un altro nodo, per il quale avremo già allora dovuto scegliere il cammino piú breve. La situazione dei dati a nostra disposizione per la decisione è la seguente:

nodi candidati	lunghezza del cammino piú breve che passa per uno dei nodi già scelti
2	500
6	200
4	700

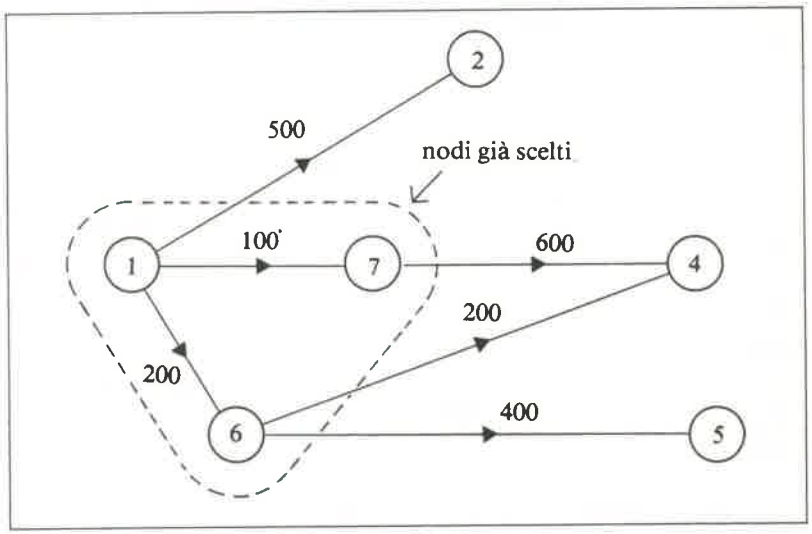


Fig. 77: Cammini coinvolti nella scelta del terzo nodo.

Si noti che 4 ha un cammino piú breve che lo collega a 1 (quello attraverso 6), ma non è necessario che questo sia al momento noto per prendere la decisione sul nuovo nodo da scegliere. A questo punto scegliamo 6, che ha distanza 200.

Ripetiamo il ragionamento. Adesso i nodi candidati sono {2, 4, 5}. Per poter prendere una decisione sul nuovo nodo occorre ricalcolare le distanze minime, perché potrebbe darsi che l'introduzione del nodo 6 nell'insieme dei nodi già scelti abbia modificato qualche percorso minimo (vedi fig. 77).

Per quanto riguarda il nodo 2, l'aver introdotto 6 non modifica la situazione, perché 2 non era collegato con 6.

Il nodo 5 è candidato proprio perché abbiamo introdotto or ora 6 e dunque la sua attuale distanza minima da 1 è la somma della distanza di 1 da 6 e della sua distanza da 6.

Infine, per quanto riguarda 4, la sua distanza minima va ora modificata, perché attraverso 6, nodo appena scelto, ha una distanza da 1 minore che attraverso 7.

nodi candidati	lunghezza del cammino piú breve che passa per uno dei nodi già scelti
2	500
4	400
5	600

A questo punto abbiamo individuato tutte le caratteristiche significative dell'algoritmo che siamo andati via via formulando. Prima però di mostrare l'algoritmo, sarà bene fare due osservazioni che ci permettono di semplificare ulteriormente il metodo:

1. il metodo non cambia sostanzialmente se invece di modificare continuamente l'insieme dei nodi candidati, noi eleggiamo fin dall'inizio tutti i nodi come candidati, assegnando una distanza «molto grande» ai nodi non direttamente collegati con il nodo 1; questo permette di dar luogo a una versione piú semplice dell'algoritmo;
2. allo stesso modo, se consideriamo «molto grande» la distanza tra due nodi qualunque non collegati, ogni volta che dobbiamo ricalcolare le distanze minime, le possiamo calcolare per tutti i nodi non ancora scelti.

L'algoritmo è mostrato nella figura 78.

1. Metti nell'insieme LUNGHEZZE-DEI-CAMMINI-PIÚ-BREVI le distanze tra il nodo 1 e tutti gli altri. Per il nodo 1 metti 0.
2. Metti nell'insieme NODI-NON-ANCORA-SCELTI tutti i nodi del grafo.
3. Aggiungi il nodo 1 all'insieme (inizialmente vuoto) NODI-GIÀ-SCELTI e toglilo dall'insieme NODI-NON-ANCORA-SCELTI.
4. Finché l'insieme NODI-NON-ANCORA-SCELTI non è vuoto esegui:
   - 4.1. Scegli un nodo in tale insieme che ha minimo valore corrispondente in LUNGHEZZE-DEI-CAMMINI-PIÚ-BREVI. Chiama NUOVO tale nodo.
   - 4.2. Aggiungi NUOVO all'insieme NODI-GIÀ-SCELTI.
   - 4.3. Togli NUOVO dall'insieme NODI-NON-ANCORA-SCELTI.
   - 4.4. Per tutti i nodi in NODI-NON-ANCORA-SCELTI esegui:
     - 4.4.1. Calcola la distanza tra il nodo e NUOVO e sommala alla distanza di NUOVO con il nodo 1. Chiama tale valore DISTANZA-TRAMITE-NUOVO.
     - 4.4.2. Se DISTANZA-TRAMITE-NUOVO è minore della distanza attuale in LUNGHEZZE-DEI-CAMMINI-PIÚ-BREVI allora sostituisci a tale valore DISTANZA-TRAMITE-NUOVO.

Fig. 78: Algoritmo di Djikstra per il calcolo delle lunghezze dei cammini minimi.

	1	2	3	4	5
1	1	2	3	4	5
2	2	4	6	8	10
3	3	6	9	12	15
4	4	8	12	16	20
5	5	10	15	20	25

Fig. 79: Tavola pitagorica di ordine 5.

Fino a ora abbiamo affrontato il problema del progetto senza tener conto delle strutture di dati con cui rappresentare i dati del problema. Abbiamo utilizzato nozioni quali grafo e insieme senza porci il problema di come rappresentarli per mezzo delle strutture di dati effettivamente disponibili in un ipotetico linguaggio programmativo a disposizione. Lo stesso atteggiamento abbiamo tenuto per le operazioni, che abbiamo espresso in modo ancora molto generale (aggiungi un nodo, calcola la distanza, ecc.). Affrontiamo ora il problema della rappresentazione, concentrando in particolare l'attenzione sulla struttura «grafo». Un modo tipico con cui i grafi vengono rappresentati in molti linguaggi programmativi è per mezzo di una diversa struttura detta 'matrice'.

Un esempio di matrice è la tavola pitagorica che appare nella figura 57 nel capitolo V, e che riproduciamo in alto riportando per ogni riga e colonna il numero a cui si riferiscono i prodotti (vedi fig. 79).

A ogni numero compreso tra 1 e 5 è associata una riga e una colonna: ogni numero nella matrice è il prodotto dei due numeri che compaiono sulla riga e sulla colonna.

Una matrice è dunque un insieme di dati (nel caso precedente i prodotti tra i primi cinque numeri) ognuno dei quali identificato da due valori, indici di riga e di colonna della matrice (nel nostro caso i due numeri di cui il dato è prodotto). Possiamo utilmente usare una matrice per rappresentare la struttura di una rete stradale?

Guardando con attenzione l'informazione che ci dà il grafo della figura 72, ci accorgiamo che sono possibili varie scelte.

Rappresentazione mediante matrice di adiacenza. In questa rappresentazione, a ogni riga e a ogni colonna della matrice è associato un nodo del grafo: nel nostro caso la matrice avrà perciò 8 righe e 8 colonne. Nella generica riga della matrice sono rappresentate le distanze che il nodo associato a quella riga ha con tutti i nodi (vedi fig. 80). Per esempio, nella riga 8 sono riportate le distanze

	1	2	3	4	5	6	7	8
8	—	—	600	600	700	—	—	—

che indicano che il nodo 8 non è collegato con i nodi 1 e 2, ha distanza 600 metri dal nodo 3, e così via.

	1	2	3	4	5	6	7	8
1	—	500	—	—	—	200	100	—
2	500	—	400	—	—	—	—	—
3	—	400	—	—	—	—	—	600
4	—	—	—	—	—	200	600	600
5	—	—	—	—	—	400	—	700
6	200	—	—	200	400	—	—	—
7	100	—	—	600	—	—	—	—
8	—	—	600	600	700	—	—	—

Fig. 80: Rappresentazione mediante matrice di adiacenza.

Rappresentazione mediante elenco di coppie adiacenti. In questa rappresentazione viene utilizzata una matrice, in ogni riga della quale si rappresentano le informazioni relative al generico ramo, e cioè:

1. coppia di nodi collegati dal ramo e
2. lunghezza del ramo.

Vedi nella figura 81 la matrice delle 'coppie adiacenti' per la nostra rete stradale.

1	2	500
1	6	200
1	7	100
2	3	400
3	8	600
4	6	200
4	7	600
4	8	600
5	6	400
5	8	700

Fig. 81: Rappresentazione mediante elenco di coppie adiacenti.

Si noti che nella rappresentazione mediante elenco di coppie ogni ramo è stato riportato una sola volta: accade cosí che per esempio compaia la riga

4	8	600

ma non compaia la riga

8	4	600

Confrontiamo le due rappresentazioni dal punto di vista della occupazione di memoria. Nella prima rappresentazione viene usata una matrice 8×8 mentre nella seconda viene usata una matrice 10×3. Se generalizziamo il discorso, e chiamiamo: n il numero dei nodi del grafo, k il numero dei rami, possiamo affermare che la seconda rappresentazione è conveniente, dal punto di vista della occupazione di memoria, ogni volta che

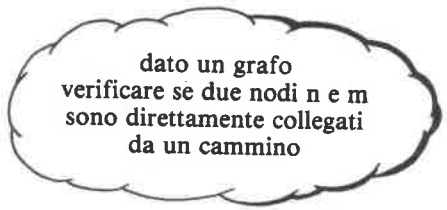
$$n \times n > 3 \times k.$$

Se pensiamo all'applicazione dei grafi in questo caso specifico, e cioè a una rete stradale, ci accorgiamo che il numero dei rami sarà in generale molto piccolo rispetto al numero dei possibili rami tra tutte le coppie dei nodi. Tale numero è dato da:

$$\frac{n \times (n-1)}{2}$$

e dunque possiamo ben dire che la condizione sopra enunciata è soddisfatta. In effetti, per poter avere una valutazione completa sulla convenienza di una scelta rispetto a un'altra, non basta confrontare le due strutture solo dal punto di vista della occupazione di memoria. È necessario anche confrontarle dal punto di vista della efficienza degli algoritmi che operano su di esse.

Scegliamo per esempio il seguente semplice problema (prescindendo dunque dall'algoritmo di Dijkstra, che richiederebbe considerazioni piú approfondite):



Supponiamo che nel linguaggio di programmazione che adottiamo per il confronto sia disponibile una operazione di lettura in memoria del valore di un elemento di una matrice, nota la sua posizione di riga e di colonna.

È chiaro che, avendo a disposizione la prima rappresentazione, possiamo direttamente accedere all'elemento che si trova in riga n e in colonna m, che ci fornisce immediatamente l'informazione desiderata.

Nel caso della seconda soluzione dobbiamo invece scandire la rappresentazione per cercare una riga che abbia in prima e seconda colonna i valori m ed n, in qualunque ordine, e questa operazione è certamente piú costosa.

In base alle osservazioni fatte è chiaro che sarebbe utile avere a disposizione una rappresentazione che unisse i vantaggi della prima (efficienza e naturalezza degli algoritmi) e della seconda (compattezza della rappresentazione). Una tale rappresentazione è mostrata nella figura 82.

La rappresentazione è composta di due matrici:

1. la matrice RAMI fornisce l'elenco ordinato di tutti i successori del nodo 1, del nodo 2, ecc. con le relative distanze;
2. la matrice (a una sola colonna, chiamata nella letteratura e nel seguito 'vettore') POSIZIONE-IN-RAMI descrive per ogni nodo il numero di riga in RAMI a partire dalla quale comincia l'elenco dei successori del nodo. Cosí per esempio in riga 7 del vettore POSIZIONE-IN-RAMI c'è scritto 16 perché l'elenco dei successori di 7 comincia in RAMI dalla riga 16.

In questo caso l'occupazione di memoria è

$$n + 4 \times k$$

poiché il vettore POSIZIONE- IN- RAMI ha n elementi e nella matrice RAMI ogni ramo è di fatto rappresentato due volte.

Con questo abbiamo concluso l'analisi delle possibili realizzazioni di grafi tramite matrici. Non ci interessa procedere oltre nel progetto dell'algoritmo di Dijkstra. Vogliamo notare, anche se non lo proviamo, che la sua complessità in termini di tempo, assumendo come dimensione dell'ingresso il numero di nodi n, è n al quadrato, risultato questo molto interessante nell'ambito degli algoritmi per grafi.

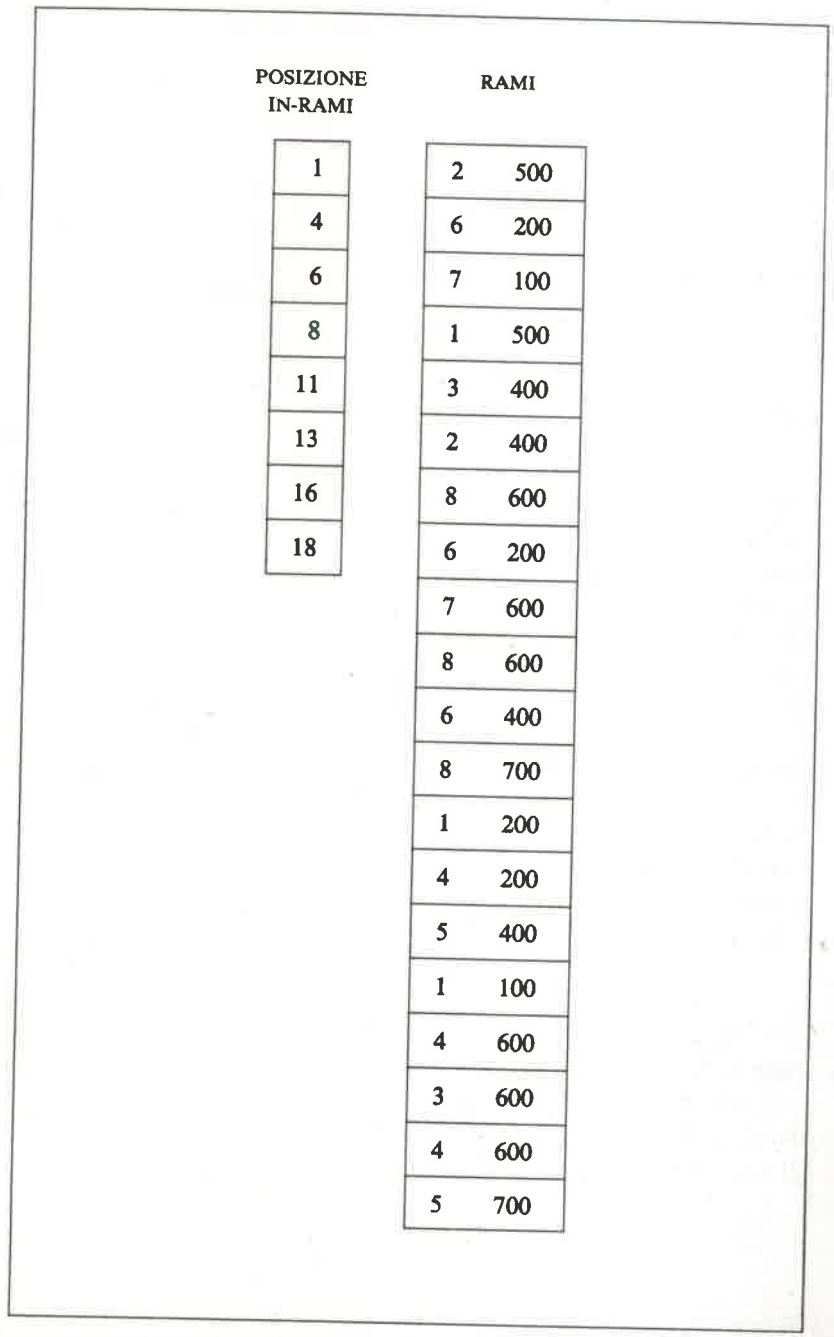


Fig. 82: Una nuova rappresentazione per il grafo.

4. Conclusioni. Come detto nell'introduzione, in questo capitolo abbiamo condotto il discorso mediante un'analisi di casi. Proviamo ora a riassumere i principali concetti che abbiamo incontrato.

Progetto dell'algoritmo e progetto delle strutture di dati sono dunque strettamente legati, come i precedenti esempi hanno chiaramente mostrato. Si è anche visto che nei casi complessi (la quasi totalità!) il progetto dell'algoritmo procede per approssimazioni successive e ogni versione è ottenuta dalla precedente raffinando qualche passo dell'algoritmo. Nel raffinamento del generico passo possiamo scoprire nuove strutture o variabili che non erano «visibili» ai livelli precedenti. In tal modo a ogni passo abbiamo solo a che fare con i dettagli che sono significativi a quel passo, ottenendo cosí una semplificazione del progetto.

Questa tecnica di progettazione, detta usualmente tecnica *top-down* (cioè dall'alto in basso) viene istintivamente usata da ciascuno di noi nella vita quotidiana quando, dovendo risolvere un problema complesso, lo dividiamo in problemi via via piú semplici finché ogni problema rimasto risulta di facile soluzione.

Naturalmente può accadere che a un dato passo del procedimento ci accorgiamo che la soluzione trovata in termini di strutture e di operazioni su tali strutture non ci soddisfi per ragioni di efficienza o anche di complicazione strutturale dell'algoritmo (quanto piú complicata è la struttura di un algoritmo tanto piú alto è il rischio di errore e difficile è la futura manutenzione). Possiamo allora rivedere la precedente scelta, correggendo il passo che abbiamo dettagliato (e lasciando inalterati gli altri) fin quando il procedimento di progetto non raggiunge lo scopo desiderato e tutti i passaggi hanno raggiunto il massimo livello di dettaglio.

Questo processo dà luogo a molte altre questioni che ricadono nella disciplina delle metodologie di programmazioni, di cui qui non possiamo occuparci.

VII. INFORMATICA E MODELLI

Modelli come approssimazione della realtà – Modelli di progetto e modelli di utilizzo – I rischi dei modelli

Nel progetto degli algoritmi esaminati nel capitolo precedente, in particolare per il problema del percorso minimo, un passo importante è consistito nella scelta della struttura dei dati del problema.

Tale scelta è avvenuta in due passi (vedi fig. 83):

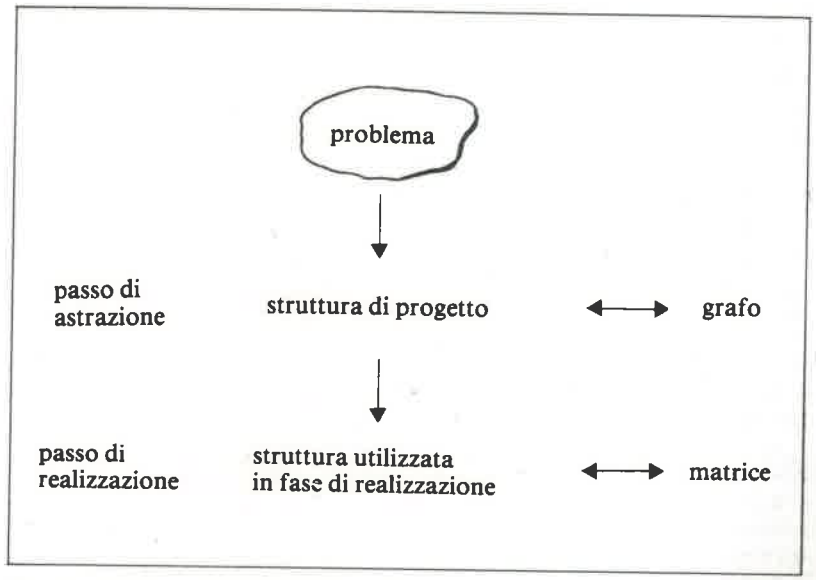


Fig. 83: Rappresentazioni utilizzate nell'esempio del capitolo V.

1. un primo passo di astrazione, in cui abbiamo individuato e schematizzato le informazioni sul mondo reale che risultano interessanti ai fini della risoluzione del problema; tali informazioni sono state rappresentate inizialmente in modo astratto per mezzo della struttura matematica grafo;
2. un secondo passo di realizzazione, in cui abbiamo indagato sulle possibili rappresentazioni dei grafi per mezzo di strutture di dati esprimibili nei linguaggi programmativi.

Le rappresentazioni che compaiono nella figura 83 sono esempi di modelli. Nel contesto della informatica, per 'modello' si intende una rappresentazione della realtà, in cui sono descritti in modo formale gli aspetti rilevanti ai fini del problema di cui vogliamo automatizzare la soluzione.

Lo scopo di questo capitolo è analizzare il fondamentale ruolo che i modelli hanno nell'informatica e, di conseguenza, nella vita di ciascuno di noi.

1. Modelli come approssimazione della realtà. L'uso delle piante topografiche per orientarsi nella visita di una città è relativamente recente, e risale alla fine del secolo scorso.

Nel numero dell'agosto 1913 della rivista mensile del Touring club italiano si affronta il problema delle regole grafiche da usare nel disegno delle piante di città. E si dice:

«Una buona pianta deve - insieme a tante altre cose - tener conto di due accorgimenti principali. Uno consiste in una sagace schematizzazione. Certe viuzze, certe strade cieche, tutti i cortili, sono dettagli da sopprimere perché rendono inutilmente trito l'insieme. Le grandi arterie di circolazione vanno invece un poco allargate, rispetto alle altre vie, in maniera che risaltino di piú. È in queste che d'ordinario il turista deve muoversi; è in queste che si innescano le vie d'accesso ai luoghi che deve visitare. Il suo occhio resta, da un moderato allargamento, aiutato ad orientarsi; nelle arterie cosí allargate si possono meglio inserire i trams ed i nomi sono piú leggibili.

«Il secondo accorgimento consiste nell'incidere i nomi delle vie su una diversa pietra di quella che porta la planimetria dei fabbricati e nello stampare questi in tinta piú o meno trasparente relativamente chiara (per esempio un bistro tenue o in un rossiccio) e quelli in un nero duro e schietto.

«Non soltanto i caratteri dello scritto sono cosí meno disturbati dal fondo e perciò piú leggibili, ma - ciò che ha grande importanza - cosí si possono tenere un pochino piú grandi di quanto lo permetterebbero a rigore le due linee parallele, appunto perché queste sono di diverso e piú leggero colore. Ma si ha cosí il vantaggio grande di leggere meglio i nomi delle strade ed i numeri di richiamo dei monumenti. A rendere piú leggibili le piante, giova altresí introdurre qualche colore che di spesso si tralascia per economia: il verde, per i passeggi ed i giardini pubblici, il celeste per le acque, ecc.».

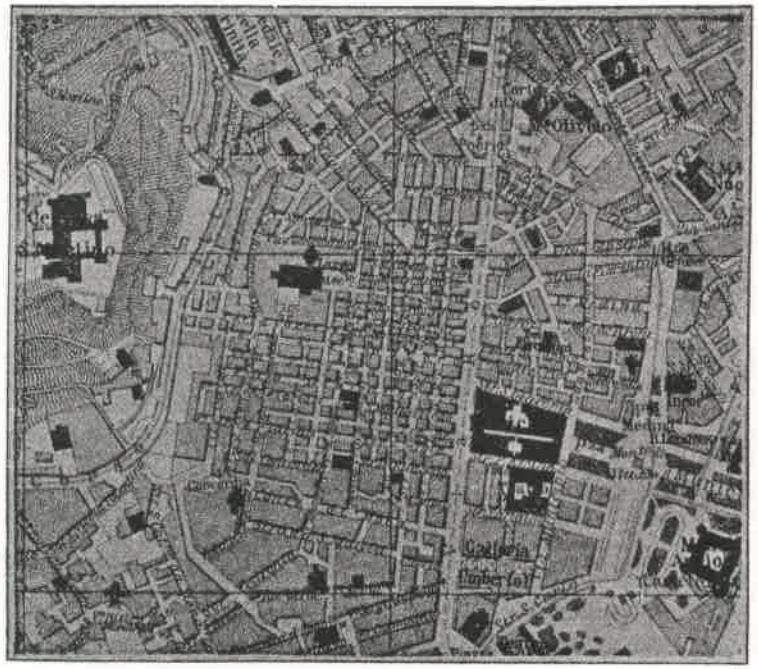
Accanto a queste considerazioni, nell'articolo vengono mostrate due piante di un quartiere di Napoli (vedi figg. 84,85) e una fotografia di scorcio del quartiere (vedi fig. 86, con l'immancabile Vesuvio sullo sfondo). Abbiamo riportato anche le didascalie originarie.

Le piante delle figure 84 e 85 sono modelli della fotografia della figura 86 (che a sua volta può essere considerata un modello del «vero» quartiere). Entrambe fanno astrazione da dettagli non interessanti; per esempio i cortili a cui si fa riferimento nel testo. La prima però riproduce le strade in maniera molto piú schematica, fino al punto da far credere a chi acquista la mappa senza essere mai stato a Napoli che i bassi napoletani siano un ameno quartiere residenziale di villette.

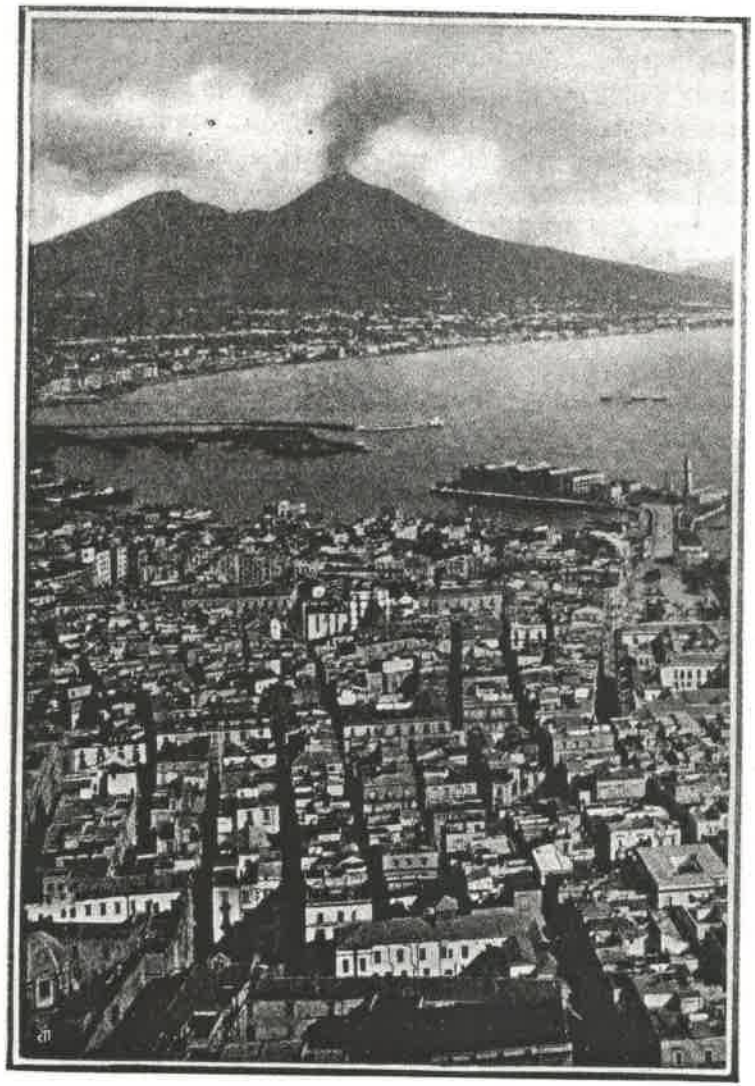
Anche l'informatica fa un diffuso uso di modelli, e, come vedremo tra poco, con problemi simili a quelli che abbiamo ora esaminato. Nel progetto di applicazioni informatiche vengono tipicamente usati due tipi di modelli, che chiameremo nel seguito modelli di progetto e modelli di utilizzo.

I 'modelli di progetto' vengono usati per rappresentare formalmente l'algoritmo e le informazioni su cui opera, in maniera in genere ancora indipendente dal linguaggio finale, cioè senza fare riferimento alle strutture di dati e alle operazioni a disposizione nel linguaggio. Abbiamo visto nel capitolo precedente una esemplificazione di questo aspetto metodologico. L'esigenza di un passo di progetto che dia luogo a una (o piú) rappresentazione intermedia è legata alla grande complessità delle scelte che devono essere in genere effettuate in termini di organizzazione del calcolo e di rappresentazione delle informazioni. Tale procedimento risponde alla strategia tipica nel progetto di sistemi complessi di andare avanti per approssimazioni successive, trascurando a un certo passo dettagli che non sono significativi per le scelte da compiere in quel momento.

QUARTIERE DI NAPOLI TRA VIA ROMA E S. MARTINO
La schematizzazione eccessiva fa credere che via Roma (Toledo) sia larghissima e il quartiere sia a piccoli fabbricati ariosi e salubri.
Vedasi invece la fotografia seguente.



LO STESSO QUARTIERE PIÙ BENE SCHEMATIZZATO.
Via Roma appare ancora come un'arteria, il quartiere a isolati è ancora forzatamente più arioso del vero ma dà già un'idea più prossima a verità che il disegno a pag. 410.



QUARTIERE TRA VIA ROMA E S. MARTINO VISTO DALL'ALTO
Corrisponde al punto male schematizzato sul disegno precedente; si vede qui che invece di piccoli fabbricati ariosi e salubri uso villette si tratta di case a molti piani divise da viuzze anguste.

Nelle figure 84, 85, 86 due piante di un quartiere di Napoli e una fotografia di scorcio del quartiere.

Mediante l'utilizzo di linguaggi programmativi il modello di progetto viene poi tradotto in programma e dati, cui corrisponde un modello di utilizzo. Intendiamo per 'modello di utilizzo' l'insieme di modalità operative che l'utente del calcolatore deve seguire nella sua interazione con il calcolatore. Una particolare ma importante classe di modelli di utilizzo sono gli stessi linguaggi di programmazione e corrispondenti traduttori, che noi adoperiamo per poter produrre quei particolari dati di uscita che sono i programmi eseguibili.

Consideriamo adesso piú in dettaglio le due classi di modelli.

2. Modelli di progetto e modelli di utilizzo. Consideriamo anzitutto i modelli di progetto. Un modello di progetto deve essere in grado di rappresentare efficacemente e allo stesso tempo formalmente algoritmo e informazioni su cui opera. A seconda della natura del problema può cambiare di molto la complessità relativa delle descrizioni formali dell'algoritmo e delle informazioni, e di conseguenza delle attività che le producono. Consideriamo qui una particolare, ma molto significativa, classe di applicazioni informatiche, in cui in genere la descrizione dell'algoritmo e la descrizione delle informazioni hanno una notevole complessità. Ci riferiamo al progetto di sistemi informativi. L'espressione *sistema informativo* è un po' abusata in informatica, ed è diventata una di quelle parole dai mille significati che vogliono dire tutto e niente.

Possiamo definire un 'sistema informativo' come un insieme di:

1. risorse umane;
2. strumenti automatici e manuali di memorizzazione, scambio, elaborazione e acquisizione di dati;
3. procedure manuali e automatizzate e dati su cui tali procedure operano.

Riprendendo l'esempio dell'agenda e dell'elenco telefonico, il sistema informativo per la ricerca dei numeri di telefono di una persona che abbia molte relazioni sociali, utilizza le seguenti risorse di dati:

— un insieme di numeri (in genere molto ridotto, sotto la decina) tenuti a mente;
— un'agenda, le cui dimensioni e organizzazione variano a seconda della quantità di relazioni sociali della persona;
— l'elenco telefonico della città in cui la persona abita (e in caso gli elenchi telefonici di altre città rilevanti per la persona);
— alcuni documenti informali, in genere fogli o foglietti di carta, su cui vengono temporaneamente riportati uno o piú numeri che ci servono temporaneamente o saltuariamente e che non si vuole o non si può riportare sull'agenda. Può capitare, per esempio, che un certo numero sia riportato soltanto sul frontespizio di una lettera che è arrivata ed è stata archiviata in un opportuno classificatore.

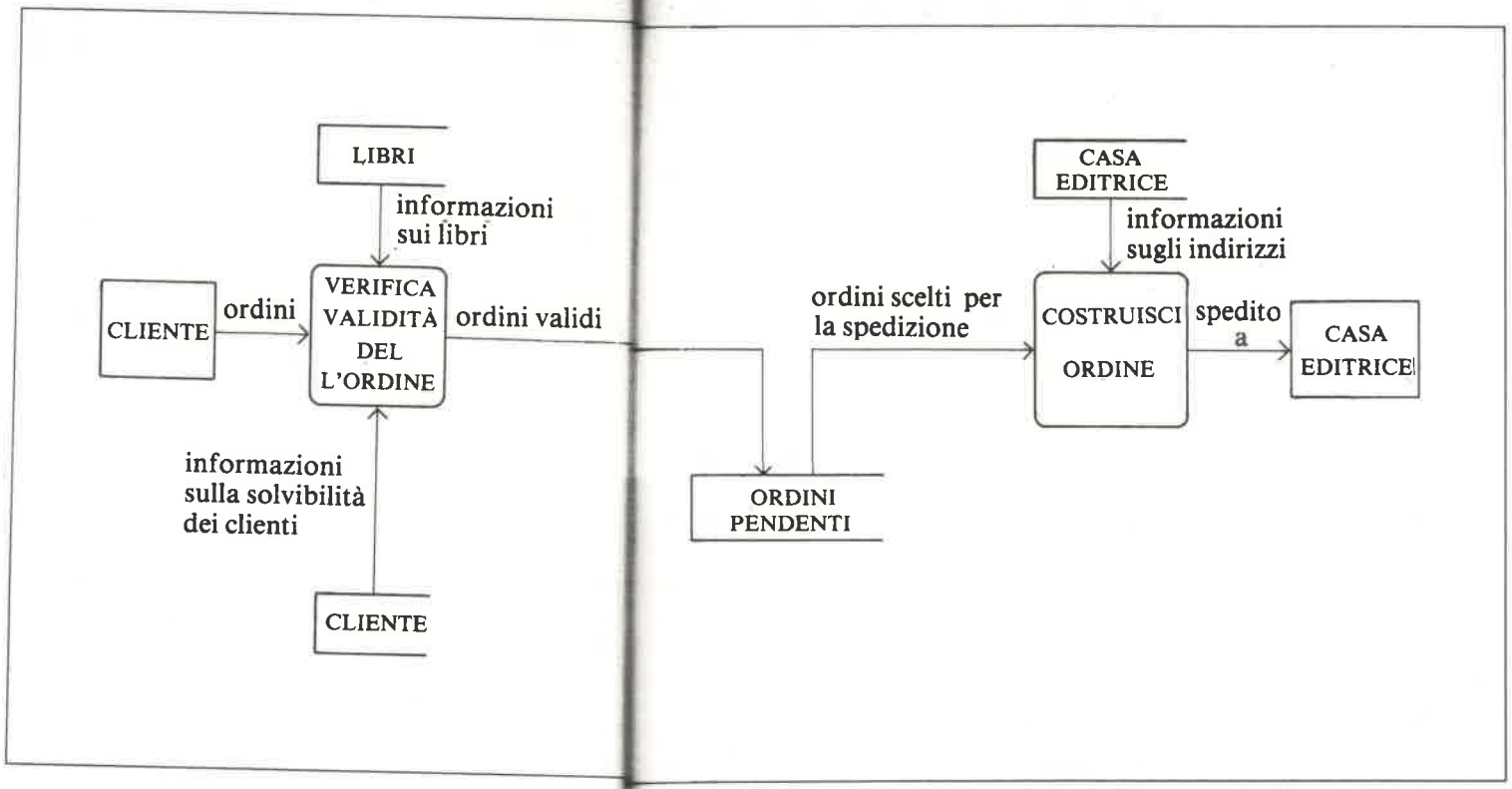
Questo piccolo sistema informativo può essere esteso in varie direzioni, costituite da un certo numero di altri sistemi informativi che vengono consultati piú raramente, e cioè:

— le agende di altre persone, cui possiamo saltuariamente chiedere un numero di telefono che non siamo riusciti a trovare consultando il nostro sistema informativo;
— l'azienda dei telefoni, cui ci rivolgiamo per trovare il numero di telefono di una persona che ancora non compare, nel caso di una recente installazione, nell'elenco attuale.

Il sistema informativo che abbiamo descritto utilizza strumenti di memorizzazione prevalentemente fatti di carta e consultati manualmente. Le procedure vengono svolte a mano e stanno nella testa della persona. Tuttavia sono già in vendita agende realizzate mediante piccoli calcolatori tascabili e in un futuro ormai vicinissimo potremo richiedere i numeri di telefono mediante una tastiera collegata al televisore di casa nostra.

Sistemi informativi automatizzati ormai in modo molto spinto sono i sistemi di prenotazione dei posti aerei o dei treni, le anagrafi dei comuni, ecc.

Fig. 87.



Quando si progetta un sistema informativo presso un'azienda, il progettista deve acquisire dagli utenti futuri del sistema tutte le notizie utili ai fini del progetto del sistema. Il progettista deve sapere bene quali sono le informazioni da rappresentare e quali operazioni devono essere fatte su tali informazioni. Gli utenti del sistema hanno in generale ruoli e poteri molto diversi nell'ambito dell'azienda, e quindi visioni diverse dei dati e delle applicazioni da automatizzare. Inoltre, non hanno in genere una cultura informatica e tendono a esprimersi in maniera piuttosto informale, senza preoccuparsi troppo delle ambiguità e imprecisioni inevitabilmente connesse al linguaggio verbale. Tali descrizioni sono poi influenzate dal ruolo che l'utente ha nella azienda, dalla sua cultura ed esperienza.

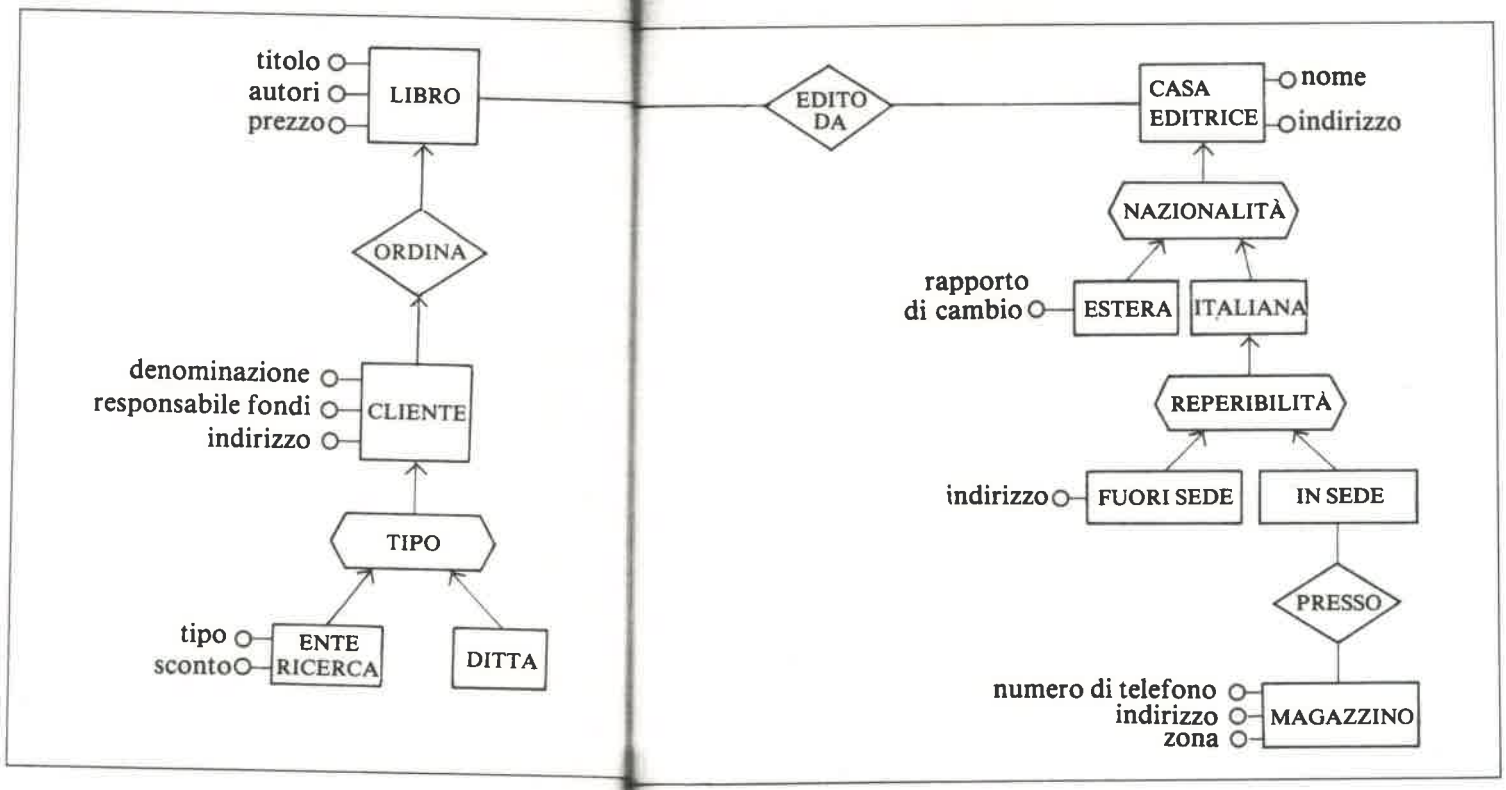
Naturalmente, al di là dei problemi linguistici e delle diverse visioni che hanno gli utenti delle informazioni e delle funzioni che su di essi operano, è chiaro che le decisioni sul progetto e sulla realizzazione sono profondamente influenzate dal potere che i diversi gruppi di utenti hanno nella struttura aziendale.

A fronte di questi problemi, i modelli usati nel progetto di sistemi informativi sono rivolti a descrivere formalmente:

1. i flussi informativi tra settori della organizzazione;
2. le attività di manipolazione e ritrovamento della informazione;
3. le informazioni da rappresentare al fine di eseguire tali attività.

Nella figura 87 è riportata una rappresentazione sintetica delle attività che interessano una ditta che si occupa di ordinare a case editrici libri per clienti che non lo vogliano o possano fare per conto proprio.

Fig. 88.



Tale rappresentazione usa un modello di progetto in cui sono definite queste categorie:

1. i flussi informativi, rappresentati da frecce
2. le organizzazioni o gruppi di persone che sono coinvolti nel sistema informativo, rappresentati col simbolo

3. le attività di ritrovamento e manipolazione delle informazioni, rappresentate col simbolo

4. le classi di informazioni coinvolte dalle attività, e di cui è necessario mantenere memoria nel sistema informativo, rappresentate col simbolo

È facile convincersi che il modello descrive flussi, attività, informazioni solo a un livello molto generale. In particolare, per quanto riguarda le informazioni, esse sono descritte soltanto attraverso un nome.

Una rappresentazione che descrive in modo piú ricco le informazioni coinvolte dalla applicazione è mostrata nella figura 88. Le classi di informazioni coinvolte dalla applicazione sono tre: clienti, libri, case editrici.

Dei clienti ci interessa conoscere nome e cognome, sigla, il nome del responsabile dei fondi e l'indirizzo a cui mandare le comunicazioni; ci interessa inoltre distinguere quali clienti siano ditte e quali enti di ricerca, perché per questi ultimi, a seconda del tipo, sono previsti opportuni sconti sugli ordini; infine, ci interessa evidentemente sapere quali libri i clienti abbiano ordinato.

Dei libri ci interessa il titolo, il prezzo, gli autori, e la casa editrice che li pubblica.

Delle case editrici ci interessa il nome e l'indirizzo; inoltre ci interessa sapere quali sono italiane e quali estere. Infatti, per le case editrici estere viene adottato un cambio di riferimento della moneta, che usualmente è maggiore del cambio ufficiale, per tener conto del ritardo con cui avviene il pagamento e della fluttuazione del cambio. Per le case editrici italiane è utile sapere quali hanno un magazzino in sede e quali no; per queste ultime, ci interessa l'indirizzo di un solo magazzino, per le prime indirizzi, numeri di telefono e zone della città di tutti i magazzini della città.

Il modello utilizzato nella rappresentazione descrive perciò informazioni e loro proprietà per mezzo delle seguenti categorie:

1. classi di oggetti della realtà di interesse con proprietà omogenee ai fini della applicazione, rappresentate col simbolo

Per esempio «magazzino» rappresenta la classe dei magazzini, di tutti i quali ci interessa ricordare numero di telefono, indirizzo, zona e la relazione con le case editrici in sede;

2. relazioni tra classi di oggetti, cioè fatti che le coinvolgono e che sono di interesse per la applicazione, rappresentate col simbolo

Per esempio «edito da» mette in relazione i libri con le case editrici che li pubblicano;

3. proprietà elementari delle classi di oggetti, rappresentate col simbolo

Per esempio titolo, autore e prezzo sono le proprietà elementari della classe dei «libri»;

4. generalizzazioni che sussistono tra le classi di oggetti, rappresentate col simbolo

Per esempio «cliente» è una astrazione delle classi «ditta» ed «ente di ricerca», che costituiscono i possibili tipi di clienti.

I modelli di progetto che abbiamo mostrato hanno essenzialmente lo scopo di descrivere la realtà di interesse in modo formale (cioè non ambiguo) e completo (cioè senza trascurare nessun aspetto utile per la applicazione).

Alla fase di progetto segue la fase di realizzazione della applicazione informatica, il cui risultato finale è un insieme di programmi cui corrisponde un modello di utilizzo. Il modello di utilizzo stabilisce, come abbiamo detto, le modalità di interazione tra utente informatico e strumento informatico, e quindi ha un ruolo determinante nella organizzazione del lavoro connessa alla utilizzazione del calcolatore. Facciamo un esempio concreto.

Un modello di utilizzo la cui influenza sta diventando via via più rilevante è il cosiddetto *text editor*, un programma cioè che aiuta a scrivere testi in linguaggio naturale. Questi programmi sono dotati di un insieme di comandi che permette di eseguire automaticamente certe attività tipiche della scrittura di testi. Questo stesso libro è stato scritto usando un *text editor*.

Molti *text editor* sono dotati dei seguenti comandi:

1. componi il testo in pagine, un certo numero di righe per pagina, distribuendo le parole nelle varie righe e inserendo omogeneamente tra esse spazi bianchi in modo tale che l'ultima parola termini al termine della riga;
2. cerca una parola (ed eventualmente sostituiscila con un'altra ogni volta che la trovi);

3. rinumera le figure del testo in ordine progressivo crescente;
4. lascia un certo numero di righe bianche e cosí via.

In molti *text editor* alcuni comandi vanno inseriti nel testo, e verranno eseguiti soltanto al momento in cui deve essere prodotto il testo scritto completo. Per esempio, col *text editor* con cui è stato scritto questo libro, per produrre il testo della figura 89 occorre scrivere il testo della figura 90.

Algoritmo somma 100 numeri

1. Poni la variabile CONTATORE al valore 1.
2. Poni la variabile SOMMA a 0.
3. Leggi da un foglio un valore e assegnalo alla variabile NUMERO. Butta il foglio.

Fig. 89: Esempio di testo che compare nel libro.

```
Algoritmo somma 100 numeri

.s 1
.lm 5
.rm 53
1. Poni la variabile CONTATORE al valore 1.
   .s 1
2. Poni la variabile SOMMA a 0.
   .s 1
3. Leggi da un foglio un valore e assegnalo alla variabile
NUMERO. Butta il foglio.
```

Fig. 90: Versione fornita al *text editor* perché fornisca il testo di fig. 89.

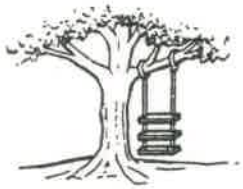
L'uso del *text editor* permette di rendere molto piú rapida la produzione di testi di cui si ha necessità di costruire molte versioni di poco differenti tra loro (per esempio questo libro, i cui capitoli sono stati spesso sottoposti a successivi aggiustamenti). Nello stesso tempo, però, obbliga l'utente a cambiare stile di lavoro poiché richiede che l'interazione col calcolatore avvenga attraverso le modalità previste dal modello di utilizzo. Per esempio mentre è tipico nei testi scritti a penna o matita cancellare la precedente versione mediante un tratto sopra le parole, usando il *text editor* si ha a disposizione un comando che permette di cancellare una parola o una riga per volta. Questo ha i suoi pregi (maggiore pulizia del testo) ma anche i suoi difetti (impossibilità di far convivere con la versione attuale versioni precedenti). Ancora, in un testo scritto a mano posso mettere note a margine, cerchiare parole o frasi significative, usare simbolismi anche inventati per l'occasione per rappresentare certi tipi di informazioni significative sul testo; usando il *text editor* la capacità espressiva deve essere tutta disciplinata nel tipo di simbolismi e interazioni permessi dallo strumento automatico.

3. I rischi dei modelli. A fronte dei tipi di utilizzo di cui abbiamo parlato, l'uso dei modelli che si fa in informatica dà luogo a vari possibili rischi di uso scorretto o dannoso per coloro che sono coinvolti dalla applicazione informatica.

Nella figura 91 è mostrato un esempio di che cosa può succedere nel corso del progetto e della realizzazione di una applicazione. La figura, come le due seguenti, è tratta dal numero di febbraio 1980 della rivista scientifica *The Computer Journal* che si stampa a Londra.

Dalla figura escono male un po' tutti. L'utente voleva un'altalena un po' artigianale, fatta con un copertone di automobile e appesa a un albero. Il finanziatore del progetto ha manifestato esigenze diverse, che con l'altalena hanno poco a che fare: in piú non si è saputo spiegare. Il progettista non solo non ha capito cosa volevano, ma ha prodotto un modello di progetto che corrisponde a un oggetto assolutamente inutilizzabile. Il programmatore, che deve tradurre in procedure funzionanti (cioè in un modello di utilizzo) le indicazioni dell'analista, si accorge che l'oggetto specificato dall'analista non dondola, e, intuendo che dovrebbe farlo, in-

l'applicazione...



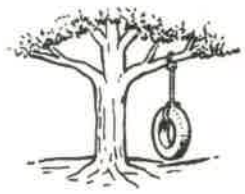
come è stata proposta dal finanziatore del progetto

come è stata descritta

come è stata progettata dal progettista

come è stata prodotta dal programmatore

come è stata installata presso l'utente



cosa voleva veramente l'utente

Fig. 91.

cosa l'utente pensava di volere (all'inizio del progetto)

cosa l'utente ha detto di volere (all'inizio del progetto)

cosa l'utente sostiene (alla fine del progetto) di avere sempre voluto

Fig. 92.

terpreta maldestramente le istruzioni appendendo l'oggetto in alto, ma non permettendogli di dondolare. Alla fine, per fare in modo che, come che sia, qualche risultato venga ottenuto, l'albero viene orrendamente mutilato e sorretto, per fargli passare sotto il seggiolino.

Nella figura 92 è analizzato con maggiore dettaglio il comportamento dell'utente, partendo dall'ipotesi piú semplice che l'utente sia unico.

Quando l'utente, come accade nella grande maggioranza dei sistemi informativi, è piú d'uno, il compito del progettista si complica ancor di piú, poiché ogni utente si esprime in maniera diversa e ha una sua visione del problema. Si veda nella figura 93 cosa può succedere in questo caso. Gli esempi

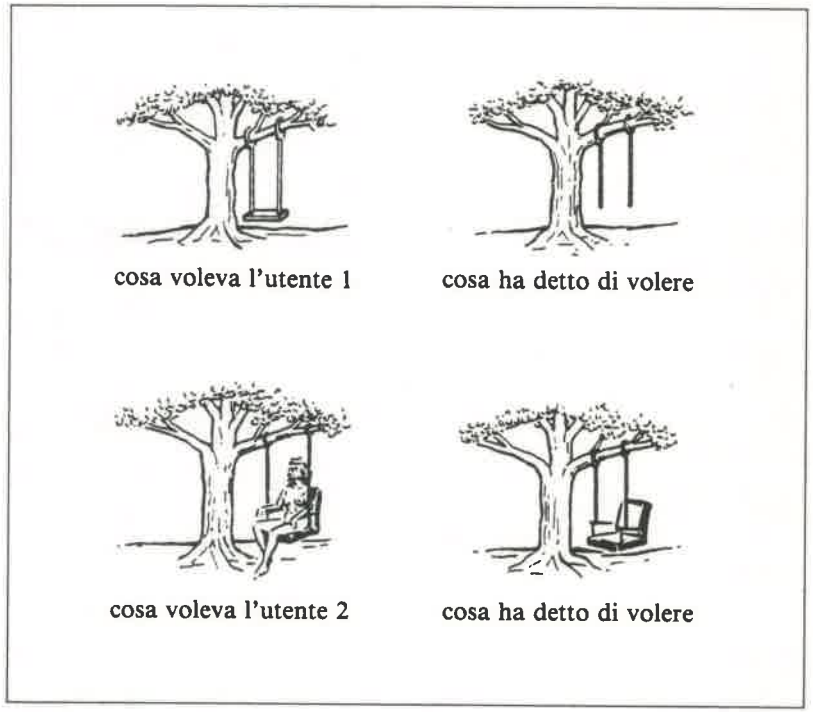


Fig. 93.

precedenti affrontano in modo scherzoso un problema evidentemente molto serio, e cioè l'uso distorto che chi possiede il modello (o in senso intellettuale, o in senso economico-politico) può farne nel corso del progetto e nella realizzazione. I modelli e il modo in cui vengono utilizzati non sono infatti neutrali, ma riflettono sempre le scelte di chi li possiede.

Molti ricorderanno il dibattito e le polemiche che si sono sviluppate a seguito dell'uscita del rapporto del Club di Roma, *I limiti dello sviluppo*, Mondadori, 1972. L'obiettivo del gruppo era quello di

«simulare in un modello matematico globale, con l'impiego di elaboratori elettronici, le tendenze e le interazioni di un certo numero di fattori dai quali dipende la sorte della società nel suo insieme: l'aumento della popolazione, la disponibilità di cibo, le riserve e i consumi di materie prime, lo sviluppo industriale, l'inquinamento».

Si noti che nella citazione la parola *modello* è utilizzata in

un senso un po' diverso dal nostro: modello è in questo caso la vera e propria descrizione della realtà di interesse, mentre in questo libro per modello abbiamo inteso l'insieme di categorie concettuali che potevano essere usate nella descrizione. Il modello adottato, accanto ad aspetti tipici della informatica quali la descrizione delle informazioni di interesse, usa formalismi tipici della dinamica dei sistemi, la disciplina che si occupa delle interazioni tra sistemi complessi. Le conclusioni del gruppo di esperti furono che soltanto un cambiamento di alcuni dei valori fondamentali della società umana, che causasse una riduzione della tendenza del sistema verso lo sviluppo, poteva garantire una evoluzione non catastrofica dei fattori sopra citati; da quel momento non erano possibili cambiamenti significativi nella distribuzione degli investimenti a livello mondiale ed era necessario un congelamento proporzionale dei prodotti nazionali lordi.

Gli stessi estensori del rapporto riferiscono varie critiche formulate al loro lavoro da persone da loro interpellate:

«Qualsiasi modello non può comprendere che un numero limitato di variabili, e quindi le interazioni risultano solo parzialmente studiate; venne altresí fatto notare che in un modello di tipo globale come questo il grado di aggregazione è necessariamente elevato [...]. Il modello apparve ad alcuni viziato da eccessivo 'tecnocraticismo', nel senso che in esso non vengono presi in considerazione certi fattori sociali di estrema importanza (per esempio il subentrare di sistemi di valori totalmente diversi)».

Accanto a tali critiche, occorre osservare che proprio l'uso dello strumento informatico permise di adottare una descrizione cosí complessa, altrimenti impossibile da usare facendo i calcoli a mano. A sua volta, proprio la complessità della descrizione rese il modello non «verificabile» in modo formale, non suscettibile cioè di uno studio sulla validità di ipotesi fatte empiricamente. In questo caso, cioè, l'uso di strumenti informatici permette solo di modellare la propria ignoranza.

La non neutralità tipica dell'uso di modelli è tanto piú pericolosa nel caso dell'informatica perché il calcolatore piú di altre tecnologie porta nascoste al suo interno le scelte progettuali; questo può rendere nel tempo piú difficile un controllo da parte di tutti i soggetti coinvolti dalla applicazione informatica sulle scelte via via effettuate.

Si assiste perciò nelle applicazioni informatiche a quella che potremmo chiamare «la inversione dei ruoli» tra realtà ed effettiva descrizione fornita dal modello; quest'ultima diventa cioè la «vera» realtà ed è il calcolatore che stabilisce cosa sia vero e cosa falso in assoluto ed è in base alle risposte che il calcolatore fornisce che si interpreta la realtà.

Il seguente esempio è riferito in J. Weizembaum, *Computer power and human reason*, 1976, S. Francisco, W.H. Freeman and Company.

«Quando all'epoca della guerra del Vietnam il presidente degli Stati Uniti decise di bombardare la Cambogia e di tenere all'oscuro della decisione il Congresso americano, i calcolatori del Pentagono furono opportunamente istruiti per trasformare i veri rapporti di guerra sui bombardamenti nei falsi rapporti a cui veniva dato accesso ai deputati del Congresso. Era stato finalmente automatizzato il ministero della Verità citato da George Orwell. La storia non veniva distrutta, veniva ricreata. E gli importanti deputati che si sentivano privilegiati per aver accesso alla lettura dei rapporti segreti forniti dai calcolatori del Pentagono credevano naturalmente che fossero veri. Dopo tutto, l'aveva detto il calcolatore».

L'asettica oggettività della macchina è dunque utilizzata per soggettivi interessi di guerra.

dati-studenti

cognome studente	data di nascita
ROSSI	3.5.1966
VERDI	30.12.1965
NERI	23.4.1964
BINI	12.12.1965
RISI	4.7.1967

dati-esami

cognome studente	materia	voto
ROSSI	INFORMATICA	23
ROSSI	FISICA	30
VERDI	INFORMATICA	30
BINI	INFORMATICA	30
RISI	CHIMICA	28
RISI	INFORMATICA	25
NERI	ELETTRONICA	26

Fig. 94.

Un ulteriore rischio connesso all'uso dei modelli nasce dalla degradazione che il progettista deve effettuare sulle scelte iniziali quando il modello di utilizzo non abbia sufficiente ricchezza linguistica da coglierne tutti gli aspetti rilevanti. Mostriamo, a questo proposito, quali problemi comporti l'uso di un linguaggio di interrogazione, un linguaggio, cioè, espressamente concepito per esprimere interrogazioni su insiemi di dati in modo semplice e naturale.

Consideriamo dunque le due tabelle della figura 94, che rappresentano informazioni relative a studenti universitari. La tabella «dati-studenti» contiene dati relativi ai cognomi degli studenti e alle loro date di nascita; la tabella «dati-esami» contiene dati relativi ai cognomi degli studenti, gli esami sostenuti e i voti ottenuti.

Supponiamo ora di voler sapere i nomi degli studenti nati a partire dal 1966 che hanno avuto piú di 27 all'esame di informatica.

In un linguaggio di interrogazione la richiesta può essere espressa nel seguente modo:

```
dati-studenti-esami     ← JOIN (dati-studenti, dati-esami)
dati-studenti-bravi     ← SELECT dati-studenti-esami
                          WHERE materia = 'INFORMATICA'
                          AND voto > 27
cognomi-studenti -bravi ← PROJECT dati-studenti-bravi
                          ON cognome studente
```

La prima istruzione costruisce una tabella in cui compaiono tutti i campi della prima e della seconda e in cui vengono messe insieme le informazioni contenute nelle due tabelle.

La seconda istruzione costruisce una tabella ottenuta dalla precedente selezionando le righe in cui la materia è INFORMATICA e il voto è maggiore di 27.

La terza istruzione elimina dalla precedente tabella tutti i campi eccetto «cognome studente».

È chiaro che il risultato della esecuzione del precedente programma produce una tabella vuota.

Se avessimo chiesto di fare una analoga statistica a un esecutore umano, probabilmente ci avrebbe risposto: «Non c'è nessuno che risponde esattamente alle tue richieste ma ci sono due studenti che vanno quasi bene, perché sono nati nel dicembre del 1965 e hanno preso 30».

Qual è la differenza tra il calcolatore e l'operatore umano, in questo caso? È la capacità dell'operatore umano di approssimare, di assumere un concetto di «distanza» tra informazioni che manca completamente agli attuali linguaggi di interrogazione.

Simili limitazioni hanno i linguaggi riguardo al trattamento della informazione cosiddetta incompleta, cioè che non può essere espressa mediante un dato preciso. È pratica di ogni giorno, per noi, trattare informazione incompleta; «Luigi avrà una quarantina d'anni» esprime una valutazione approssimata sulla età di una persona, e dà una informazione ben diversa dalla frase «Luigi ha quarant'anni».

Come affrontare i rischi che abbiamo descritto? Cercando di creare una cultura modellistica che dia a tutti coloro che sono coinvolti dalla applicazione informatica gli strumenti per comprendere le scelte fatte, e le degradazioni, volute od obbligate, rispetto alle esigenze iniziali. I modelli di progetto, per esempio, possono diventare il linguaggio di comunicazione tra utente e progettista e quindi uno strumento importante con cui l'utente controlla che l'applicazione sia realizzata secondo le specifiche iniziali. A fronte di qualunque cambiamento da effettuare, l'utente può controllare le conseguenze della modifica esprimendo tale modifica in termini del modello di progetto con cui ha familiarità. I modelli possono dunque diventare uno strumento culturale importante per rendere trasparente l'uso dello strumento informatico.

Su questo aspetto si innesca una importante questione, con cui vorremmo concludere il libro: quale cultura informatica serve all'uomo d'oggi? Forse mai come per altre discipline c'è il rischio che l'informatica venga culturalmente ridotta a un fatto consumistico, e venga insegnata, consapevolmente o no, per creare nuova domanda di nuovi consumi. In questo libro ho cercato di tracciare una via diversa, provando a mostrare alcuni concetti che l'informatica può fornire a tutti noi, concetti che possono già diventare, in varie forme, cultura di tutti.

Nello scrivere questo libro ho avuto un grande aiuto da varie persone che hanno fatto da cavie o da consiglieri fornendo osservazioni che sono state molto importanti ai fini della stesura definitiva. Esse sono: Giorgio Ausiello, Curzio Batini, Giuseppe Batini, Roberto Battistoni, Paola Bertolazzi, Luigia Carlucci Aiello, Giacomo Cioffi, Giulio De Petra, Luigi Grippo, Maurizio Lenzerini, Claudio Leporelli, Mario Lucertini, Alberto Marchetti Spaccamela, Gianni Nardi, Nando Pompei, Marco Protasi, Aldo Roveri, Maurizio Talamo e Loredana, mia moglie.

Naturalmente, come sempre, la responsabilità delle cose scritte è solo mia.

INDICE DEI TERMINI DEFINITI E DEGLI ARGOMENTI

alfabeto binario (o codice), 33
algoritmo, 27
asserzione, 58
automatizzare, 8

calcolabilità, 70 sgg.
calcolatore elettronico, 20
cella, 34
ciclo, 67 sgg.
codifica, 44
complessità, 105 sgg.
comportamento asintotico, 101
- nel transitorio, 101
computer, 22
coppie adiacenti, 134
correttezza, 43 sgg.
correzione, 44
costante, 36

dati
- di ingresso, 27
- di test, 27 sgg.
- di uscita, 28
deduzione, 59
Dijkstra, algoritmo di, 127
documentazione, 44

efficienza, 88 sgg.
espressione aritmetica, 36
- logica, 36
Euclide, algoritmo di, 94

forma sintattica, 77

grafo di flusso, 36
induzione, 62
informatica, 7
informazione, 26
interprete, 36
istruzione, 29

legge generativa, 17
lettore di schede, 32
linguaggio ad alto livello, 35
- dei grafi di flusso, 36
- macchina, 34
- naturale, 25
- programmativo, 29
- programmativo di Pascal, 29
logica, 59

macchina a programma, 76
- di Turing, 70
massimo comun divisore, 91
matrice, 132
- di adiacenza, 133
memoria, 33
metodi formali, 57 sgg.
modello, 140
- di progetto, 141
- di utilizzo, 144

numero primo, 95

organo di ingresso, 32
- di governo, 33
- di uscita, 32

personal computer, 22
problema, 25 sgg.
progetto di programmi, 44 sgg.
- logico, 44
programma, 28, 42

ridondante, 16

scheda perforata, 32
simbolo di sommatoria, 58
sintassi 35, 77
sistema informativo, 144
stampante, 33
stato, 71
struttura di dati, 113 sgg.

tabella di transizione, 72
tastiera, 32
tecnica di deduzione, 59
terminazione, 64
tesi di Church, 82
traduttore, 35

unità logico-aritmetica, 33

variabile, 29
vettore, 136
video, 33

LETTURE DI ALTRI LIBRI DI BASE

Questo libro di base può essere messo in relazione con uno o piú volumi delle otto sezioni di cui la collana si compone. In questo modo si costituiscono percorsi di lettura per temi e problemi fino a formare una enciclopedia scomponibile. Ecco il percorso di lettura che vi proponiamo in relazione a questo volume, *Le basi dell'informatica* (*sez. 7, n° 69*):

dalla sezione 4. Arti e comunicazioni: linguaggi e tecniche espressive

De Mauro, Guida all'uso delle parole (*Ldb 3*)
Lenzi, Il giornale (*Ldb 22*)

dalla sezione 5. Economia e lavoro: organizzazione e tecnologie

Frova, La rivoluzione elettronica (*Ldb 18*).

dalla sezione 7. Il sapere: scienze e campi di ricerca

Lombardo Radice, L'Infinito (*Ldb 26*).

ALTRE LETTURE

Vorrei anzitutto citare un testo il cui scopo è in qualche modo analogo a quello di questo libro, mostrare cioè alcuni contributi metodologici della informatica. Piú avanti compaiono citazioni specifiche ai vari capitoli.

Il libro già citato di G. Lariccia, *Le radici della informatica*, Firenze, Sansoni, 1981, L. 14.000 è una profonda riflessione sulla «Informatica senza calcolatore» in cui (citazione dal testo) «si dimostra come l'informatica, le cui radici sono nell'uomo, venga da lontano, e debba andare piú lontano delle macchine a cui pure è naturalmente legata». Il lettore che sia rimasto interessato dal presente libro e voglia continuare l'itinerario che qui si è tentato di tracciare, troverà nel libro di Lariccia una interessante «seconda puntata»

Capitolo II

Per quanto riguarda l'evoluzione dei calcolatori elettronici, si veda G. Friedrichs, A. Schaff, *Rivoluzione microelettronica*, Milano, Est Mondadori, 1982, L. 20.000. Piú in generale, esistono ormai vari testi tradotti in italiano o di autori italiani che forniscono monografie complete su specifici aspetti tecnologici dell'informatica quali l'architettura dei sistemi di elaborazione, i linguaggi di programmazione, i traduttori, i sistemi di gestione di archivi e basi di dati. Una collana completa negli argomenti e accurata e aggiornata nei contenuti è la serie di informatica 1 della collana «Testi scientifici modulari» della Isedi. Altre collane di testi di informatica sono edite da Franco Angeli e da Boringhieri (quest'ultima è citata nel seguito per specifici testi).

Capitolo III

La problematica dei metodi formali può essere approfondita nel libro di Z. Manna, *Teoria matematica della computazione*, Torino, Boringhieri, che però richiede una certa conoscenza di argomenti di logica come il calcolo delle proposizioni e dei predicati. Una introduzione a questi argomenti può trovarsi in C. Mangione, *Elementi di logica matematica*, Torino, Boringhieri.

Capitolo IV

Un ottimo libro sui metodi per valutare l'efficienza di algoritmi è F. Luccio, *La struttura degli algoritmi*, Torino, Boringhieri, 1982, L. 20.000.

Capitolo V

La citazione di Turing è tratta dalla traduzione italiana apparsa nel libro a cura di V. Somenzi, *La filosofia degli automi*, Torino, Boringhieri, che raccoglie scritti di diversi autori sugli aspetti filosofici del calcolatore. Tutto il libro è molto interessante.

Sul problema della calcolabilità si veda E. Casari, *Computabilità e ricorsività*, Varese, Quaderni della Scuola di studi superiori sugli idrocarburi dell'ENI, 1959 e inoltre la voce «Calcolo», della Enciclopedia Einaudi, Torino, Einaudi.

Una trattazione piuttosto approfondita sulle macchine di Turing viste come modello di calcolo appare in M.G. Roccatelli Micci, G. Veredice, *La formalizzazione del pensiero*, ESA Editrice, 1983, L. 10.000 in cui è anche interessante la prefazione sul concetto di «pensiero algoritmico» scritta da L. Lombardo Radice.

Capitolo VI

Sul progetto di algoritmi e programmi sono stati scritti molti libri. Tutti in pratica richiedono come propedeutica la conoscenza di almeno un linguaggio programmativo. Al lettore interessato a imparare un linguaggio consiglierei anzitutto di trovare da qualche parte un *personal computer* (presso una ditta, un amico, ecc.). Questi calcolatori di relativamente basso costo (circa 3-8 milioni a seconda della configurazione) hanno però ormai traduttori per linguaggi anche sofisticati come il Pascal. In mancanza di un *personal* può cercare di procurarsi un calcolatore da tavolo, da collegare al televisore (spesa circa 400.000-800.000 lire) su cui spesso funzionano il linguaggio Basic e versioni semplificate dello stesso Pascal. I limiti di queste macchine rendono però piú fastidioso e meno produttivo il processo di apprendimento.

Un classico libro sull'argomento, semplice nello stile ma rigoroso nei concetti è N. Wirth, *Principi di programmazione*, Milano, Isedi.

Il libro di F. Luccio, *Strutture, Linguaggi, Sintassi*, Torino, Boringhieri, 1972, nonostante abbia qualche anno, rimane a mio parere la piú accessibile e piacevole introduzione alle strutture di dati.

Altro testo utile per approfondire il problema dei metodi di progettazione strutturata è G.A. Lanzarone, M. Maiocchi, R. Polillo, *Introduzione alla programmazione strutturata*, Milano, Franco Angeli, 1977.

Capitolo VII

Una riflessione sul ruolo dei modelli in informatica compare (con molte idee importanti e qualche ingenuità) in P. Manacorda, *Il calcolatore del capitale*, Milano, Feltrinelli, 1976 e in successivi scritti della stessa Manacorda sul quotidiano *Il Manifesto*. Un accenno al problema si trova in B. Morandi, *La merce che discute*, Milano, Feltrinelli, 1978.

Piú in generale riguardo al ruolo ideologico, politico ed economico dell'informatica si veda ancora il libro della Manacorda, S. Nora, A. Mine, *Convivere con il calcolatore*, Milano, Bompiani.