Almerico Murli Giulio Giunta Giuliano Laccetti
Mariarosaria Rizzardi

Laboratorio di Programmazione I

Liguori Editore

Tutti i diritti sono riservati. Nessuna parte di questa pubblicazione può essere tradotta, riprodotta, copiata o trasmessa senza l'autorizzazione scritta dell'editore. Fotocopie per uso personale del lettore possono essere effettuate nei limiti del 15% di ciascun volume/fascicolo di periodico dietro pagamento alla SIAE del compenso previsto dall'art. 68, comma 4, della legge 22 aprile 1941 n. 633 ovvero dall'accordo stipulato tra SIAE, AIE, SNS e CNA, CONFARTIGIANATO, CASA, CLAAI, CONFCOMMERCIO, CONFESERCENTI il 18 dicembre 2000. Le riproduzioni ad uso differente da quello personale potranno avvenire, per un numero di pagine non superiori al 15% del presente volume, solo a seguito di specifica autorizzazione rilasciata da AIDRO, via delle Erbe 2, 20121 Milano, telefax 02 809506, e-mail aidro@iol.it.

Prima edizione italiana Settembre 2003
Liguori Editore, Srl
via Posillipo 394
I 80123 Napoli

http://www.liguori.it

Copyright © Liguori Editore, S.r.l. 2003

Murli, Almerico :
Laboratorio di Programmazione I/Almerico Murli, Giulio Giunta, Giuliano Laccetti, Mariarosaria Rizzardi
Napoli : Liguori, 2003
ISBN 88 - 207 - 3372 - 2

1. Progetto di algoritmi 2. Matlab I. Titolo

Ristampe:

9 8 7 6 5 4 3 2 1 0 2011 2010 2009 2008 2007 2006 2005 2004 2003

Questo volume è stato stampato in Italia dalle Officine Grafiche Liguori - Napoli su carta inalterabile, priva di acidi, a PH neutro, conforme alle norme Iso 9706 ∞.

Indice

Prologo 1

1. Introduzione 7
1.1 Il concetto di algoritmo 7

2. Struttura del calcolatore 23
2.1 La macchina fisica 24
2.1.1 La memoria 28
2.1.2 Il processore 31
2.1.3 Sottosistemi di I/O 35
2.2 Rappresentazione delle informazioni 38
2.2.1 Rappresentazione dei dati 38
2.2.2 Rappresentazione delle istruzioni 68
2.3 Dalla macchina fisica alla macchina virtuale 74
2.3.1 Sistema Operativo 77
2.3.2 Traduttori di linguaggi di programmazione 80
2.4 Approfondimenti 85
2.4.1 Cenni storici 85
2.4.2 Approfondimenti di architettura 88
2.4.2.1 Canali di comunicazione 88
2.4.2.2 Sottosistemi di comunicazione in rete 92
2.4.2.3 Valutazione delle prestazioni 94
2.4.3 Approfondimenti di software di base 101

3. Costruzione degli algoritmi 111
3.1 Linguaggio e criteri di sviluppo 111
3.1.1 Variabili 111
3.1.2 Assegnazione 114
3.1.3 Istruzioni di I/O 118
3.1.4 Function e Procedure 120
3.1.5 Costrutti di controllo 129
3.1.6 Prime indicazioni per sviluppare algoritmi 141
3.2 Il tipo strutturato array 163
3.2.1 Strutture dati e variabili strutturate 163
3.2.2 Proprietà di base del tipo array 167
3.2.3 Algoritmi elementari con array 171

3.3 Efficienza di un algoritmo 193
3.3.1 Complessità di tempo di un algoritmo 194
3.3.2 Complessità di spazio di un algoritmo 197
3.3.3 Complessità asintotica di un algoritmo 198
3.3.4 Complessità intrinseca di un problema 206
3.4 Algoritmi per problemi elementari con matrici 209
3.4.1 Definizioni generali e partizionamenti 209
3.4.2 Caratteristiche strutturali 214
3.4.3 Criteri per la rappresentazione 218
3.4.4 Nuclei computazionali di base 227

4. MATLAB 243
4.1 Organizzazione di una computazione 244
4.2 Tipi primitivi, variabili e assegnazione 245
4.3 Operazioni strutturali su matrici 254
4.4 Operazioni aritmetiche matriciali 258
4.5 Operazioni componente per componente 262
4.6 Confronto e operazioni logiche con matrici 264
4.7 Function e script 265
4.7.1 Script 266
4.7.2 Function non primitive 267
4.7.3 Alcune function primitive 268
4.8 Costrutti di controllo 284
4.9 Grafica 287
4.10 Grafica avanzata 295
4.11 Input e Output 298
4.12 Algoritmi elementari in Matlab 301

Bibliografia per approfondimenti 307

Prologo

Io stimo più il trovare un vero, benché di cosa leggera, ch'l disputar lungamente delle massime questioni, senza conseguir verità nissuna.

Galileo Galilei (1564 - 1642)

Il calcolatore è un potente e flessibile strumento per risolvere problemi.

Negli ultimi anni il calcolatore ha pervaso quasi tutti gli aspetti delle società evolute: la casa, l'ufficio, la scuola, i centri di ricerca, i centri di servizi, le aziende manifatturiere, il terziario, la finanza, l'amministrazione pubblica, le telecomunicazioni, l'intrattenimento, i trasporti, la difesa militare, la protezione civile, ...

In tutti questi settori i problemi che vengono risolti con il calcolatore sono di natura diversa.

Vi sono problemi la cui soluzione richiede la ripetizione di lunghe sequenze di operazioni; problemi caratterizzati dalla gestione di grandi quantità di informazioni (testi, dati scientifici, suoni, immagini, filmati video); problemi che richiedono il controllo dell'evoluzione di un processo di lunga durata (telecomunicazioni, processi industriali, controllo satelliti, controllo dei consumi di un'automobile). Si tratta di problemi alla cui risoluzione diretta l'uomo è inadeguato.

Vi sono inoltre problemi, come la visione, la comprensione del linguaggio naturale, la traduzione tra linguagggi naturali, la deduzione logica, ..., in cui l'uomo è abilissimo e il calcolatore viene utilizzato per *mimare*, cioè per riprodurre, alcune funzionalità, spesso in modo non ancora soddisfacente.

Infine vi sono problemi in cui il calcolatore sostituisce o complementa l'attività manuale dell'uomo (industria manifatturiera, grafica, editoria, chirurgia); generalmente in queste applicazioni il calcolatore controlla anche un insieme di apparecchiature meccaniche specifiche.

La diffusione del calcolatore, già così estesa, è destinata a crescere sia in quantità sia in qualità, per divenire una presenza profonda nella cultura e nelle abitudini della società.

Le basi tecnologiche di tale pervasione sono la miniaturizzazione dei componenti elettronici e l'interconnessione dei calcolatori in rete.

La miniaturizzazione ha ottenuto risultati prodigiosi. C'è più elettronica in una attuale automobile del segmento medio di quanta ce ne fosse a bordo dell'Apollo 11 in

missione lunare nel 1969. Una odierna piattaforma per videogiochi, dal costo di un paio di centinaia di Euro, ha una potenza di calcolo maggiore di quella del più potente calcolatore di 15 anni fa, che aveva un volume cento volte maggiore ed era diecimila volte più costoso.

Le reti di calcolatori collegano decine di milioni di calcolatori in tutto il mondo e consentono lo scambio di informazioni, l'accesso ai dati e ai servizi più disparati; il tempo necessario per trasferire informazioni tra due calcolatori è praticamente indipendente dalla distanza fisica tra gli stessi.

Il calcolatore può essere usato a vari livelli di *coscienza*: dall'uso passivo e inconsapevole (es. un calcolatore che gestisce centrali telefoniche), all'uso di routine (es. scrittura testi, posta elettronica, *navigazione* in rete), fino alle attività progettuali.

Lo scopo di questo testo è fornire le competenze di base necessarie per risolvere problemi con il calcolatore, cioè per progettare procedimenti di risoluzione di problemi.

L'insieme di tali competenze, che sono di natura operativa, tecnica, metodologica e teorica, costituisce una disciplina quantitativa che chiamiamo *disciplina del calcolo*[1] o *computazionale* oppure semplicemente *programmazione*.

Si noti che i termini *calcolo* e *computazione* sono da intendere nell'accezione più ampia di quella puramente aritmetica, cioè con il significato (proprio della parola latina *computatio, onis*) di procedimento o serie di operazioni per determinare quantitativamente una grandezza o una entità esprimibile simbolicamente.

Useremo la parola *algoritmo* per indicare la descrizione di una attività di calcolo che risolve un problema.

L'algoritmo è l'attore principale di questo testo. Spesso useremo i termini *algoritmo* e *programma* come sinonimi, anche se preferiamo utilizzare il termine programma per indicare il prodotto finale, usabile, dell'attività di creazione di un procedimento risolutivo effettivo di un problema.

Il termine *software* sarà sempre usato solo con quest'ultimo significato.

La capacità di progettare algoritmi è la discriminante tra gli esperti di calcolatori e i meri utilizzatori.

Vedremo che è possibile progettare algoritmi per problemi che intuitivamente appaiono molto complessi (tra l'altro quantificheremo il concetto di *complesso*) e noteremo che, malgrado alcuni procedimenti possano apparire complicati, la loro descrizione può essere sorprendentemente semplice.

Esistono delle metodologie per dominare la complessità dei problemi; l'idea alla base di questo testo è che è possibile apprendere tali metodologie attraverso la risoluzione effettiva di poche decine di problemi campione.

Non c'è dubbio che la programmazione sia una attività intellettuale che rientra nell'alveo della matematica. Essa può essere presentata come una branca formale della

[1] In inglese *computing*.

matematica. Noi non abbiamo seguito questa strada e abbiamo privilegiato soprattutto il punto di vista della programmazione come risoluzione di problemi; a nostro avviso, questo approccio non riduce il rigore della trattazione e la solidità delle competenze trasmesse.

I problemi e la loro risoluzione computazionale sono presentati cercando di sottolineare le seguenti azioni, alcune delle quali sono fasi essenziali del processo di risoluzione:

definire il problema;
individuare la struttura (lo scheletro) di controllo generale dell'algoritmo;
descrivere precisamente le operazioni all'interno dello scheletro di controllo;
verificare la correttezza dell'algoritmo e valutarne il costo;
verificare il comportamento dell'algoritmo al variare del numero e del valore dei dati;
sviluppare diverse versioni dell'algoritmo per problemi analoghi;
mostrare che problemi diversi tra loro possono essere risolti da algoritmi con la stessa struttura.

La scelta dei problemi campione vuole anche aiutare a sviluppare negli allievi quelli che sono stati[2] scherzosamente ma ingegnosamente chiamati i *cinque sensi computazionali* e cioè il *tatto* per il finito, il discreto e l'accuratezza di una appossimazione, il *naso* per la complessità indotta dall'aumentare delle dimensioni del problema, l'*orecchio* per *sentire* l'efficienza di un algoritmo e quindi determinare la sua effettiva utilizzabilità per risolvere un problema, l'*occhio* per visualizzare geometricamente o graficamente un problema e la sua soluzione e, infine, ma forse il più importante, il *gusto* per la sperimentazione.

La sperimentazione è un aspetto centrale; una competenza fondamentale in campo computazionale è la capacità di interpretare il risultato generato dall'algoritmo e di valutare la sua affidabilità. Essa si acquisisce attraverso una intensa attività individuale di laboratorio che consenta la sperimentazione diretta dei problemi da risolvere, interagendo con il calcolatore come con uno strumento per la comprensione di fenomeni. Si tratta dello stesso tipo di competenze che si acquisisce, per esempio, in un laboratorio di chimica o di fisica realizzando esperimenti.

Parte integrante dell'attività di progetto di un algoritmo[3] è la capacità di analizzarne la *bontà*. L'analisi di un algoritmo viene effettuata in modo quantitativo e riguarda l'efficienza ma anche l'accuratezza dell'algoritmo. Il testo esamina l'analisi dell'efficienza, fornendo solo alcuni esempi relativi al problema dell'accuratezza, la cui trattazione esauriente è rimandata a testi di Calcolo Numerico.

[2] Charles Van Loan, SIAM News, Agosto/Settembre 1995.

[3] "Fino a poco tempo fa, i matematici consideravano risolto un problema se esisteva un qualunque algoritmo per risolverlo; il processo di esecuzione dell'algoritmo era di importanza secondaria. Probabilmente il motivo di questo atteggiamento di indifferenza era dovuto al fatto che, dovendo fare i calcoli a mano, o questi erano assai facili da fare oppure erano totalmente impraticabili. Il calcolatore ha radicalmente cambiato questo quadro. Adesso è importantissimo trovare metodi di risoluzione che siano efficienti per il calcolo". Questa riflessione di E. Bombieri, uno dei più rilevanti matematici italiani di questi anni, sottolinea l'importanza dell'efficienza degli algoritmi.

L'organizzazione complessiva del testo rispecchia la convinzione che apprendere la programmazione significa sviluppare l'*intuito computazionale*.

Questa capacità di *vedere* un problema del mondo reale come un problema computazionale, di progettare un algoritmo per la sua risoluzione e di valutarne l'affidabilità e quindi l'effettiva utilizzabilità, è una competenza di tipo generale, utilizzabile per problemi di natura diversa. Ciò sottolinea l'inutilità di operare distinzioni a priori tra varie classi di problemi[4] e suggerisce una trattazione il più possibile unitaria.

È evidente che uno specialista in un settore farà uso di strumenti e tecniche particolari in applicazioni specifiche, talvolta ignote allo specialista in un altro settore; tuttavia, tale livello di competenze specifiche si pone al di sopra dello strato delle indispensabili competenze di base, che sono l'oggetto di questo testo.

Al crescere della dimensione del problema da risolvere, intesa come numero di sottoproblemi in cui esso è articolato, l'attività di progetto di un programma diventa sempre più una attività di integrazione e assemblaggio di programmi per i vari sottoproblemi. Spesso tali programmi non sono sviluppati da un unico programmatore ma sono programmi già costruiti e disponibili come prodotti software, sotto la forma di librerie, package, sistemi. Paradossalmente, un buon programmatore nel progettare un software fa uso il più possibile di programmi scritti da altri; tali moduli sono sviluppati da specialisti nei vari settori e quindi sono prodotti di elevata qualità che garantiscono un alto grado di affidabilità ed efficacia del prodotto finale complessivo. Questa strategia di sviluppo di un programma richiede la capacità di saper selezionare l'algoritmo più adatto per risolvere efficacemente un particolare problema. Tale capacità va sviluppata e vuole essere una delle ricadute di questo testo.

L'uso della rete mondiale Internet e dei servizi ad essa collegati, come il World Wide Web (www), può risultare molto utile per l'aggiornamento in questo campo; il www consente di vedere i calcolatori connessi a Internet come un unico contenitore di informazioni, cioè di dati e di programmi e, più in generale, di *servizi*, in cui è possibile ricercare soluzioni ai propri problemi. Il testo contiene una breve introduzione a questi aspetti, alcune indicazioni per l'uso e sollecitazioni per approfondimenti individuali.

Il punto di vista che abbiamo scelto per introdurre la disciplina del calcolo pone al centro l'algoritmo e in posizione sussidiaria i linguaggi di programmazione e l'architettura del calcolatore.

L'uso di un particolare linguaggio non restringe l'insieme dei problemi che si possono risolvere ma solo la notazione per la descrizione dei programmi. I linguaggi a cui si fa riferimento nel testo, il linguaggio informale P-like e il Matlab, sono presentati dal punto di vista dell'utilizzatore e il loro approfondimento non è uniforme, ma riguarda solo gli aspetti utili alla risoluzione dei problemi proposti.

L'uso di un linguaggio informale (simile al linguaggio Pascal, spesso referenziato come Pascal-like, ma che noi chiameremo, per evitare confusioni, *P-like*) vuole trasmet-

[4] Per esempio, tra problemi tecnico-scientifici, problemi di tipo gestionale, problemi non numerici, problemi di grafica computazionale, ecc...

tere l'idea che un linguaggio è solo un aspetto, per di più strumentale, del processo di risoluzione computazionale di un problema. Il P-like sintetizza molte delle idee di base dei linguaggi di programmazione e consente di non approfondire dettagli inessenziali. *Matlab* è un *ambiente di risoluzione di problemi* (*problem solving environment*), più che un semplice linguaggio, e sta diventando una *lingua franca* del calcolo scientifico.

Il testo è pensato per una lettura *non lineare*, nel senso che il capitolo sul Matlab deve essere studiato *in contemporanea* rispetto al capitolo sulle metodologie di costruzione degli algoritmi: un programma in P-like deve essere eseguito *a mano*, mentre la sua versione in Matlab deve essere eseguita dal calcolatore.

L'organizzazione interna (hardware e software) di un calcolatore è descritta a un livello elementare con l'obiettivo di sottolineare le caratteristiche macroscopiche fondamentali dell'esecutore degli algoritmi (che costituiscono la definizione del cosiddetto *modello di calcolo*) e quindi evidenziare l'influenza che la struttura del calcolatore esercita su alcune proprietà degli algoritmi. La programmazione è una attività intellettuale profondamente legata alla tecnologia; quindi una analisi, anche se breve, di ciò che si intende oggi per calcolatore (dai PC fino alle reti) disegna il contesto tecnologico di riferimento. L'approfondimento di questi temi è rinviata all'ampia letteratura esistente e ai corsi universitari specifici.

Il testo è una introduzione elementare che non prevede alcuna conoscenza propedeutica dell'argomento; esso è principalmente indirizzato a studenti universitari del primo anno di Corsi di Laurea di carattere scientifico o tecnico, affinchè la cultura computazionale complementi quella sperimentale e quella teorica nella loro formazione professionale complessiva.

Il testo può essere anche utilizzato da studenti di Facoltà non scientifiche e da studenti della scuola superiore che vogliano diventare utilizzatori critici dei calcolatori, e anche da chiunque sia interessato a sviluppare o consolidare l'attitudine alla *visione algoritmica* dei problemi.

Capitolo primo

Introduzione

I processi computazionali sono entità astratte che dimorano nei calcolatori. Durante la loro evoluzione, i processi manipolano altre entità astratte, chiamate dati. L'evoluzione di un processo è diretta da uno schema di regole, chiamato programma. L'uomo crea i programmi che dirigono i processi.Un processo computazionale non può essere visto o toccato. Esso non è composto di materia. Ma è un qualcosa di assolutamente reale! Può effettuare lavoro intellettuale.

H. Abelson, G. Sussman, J. Sussman - *Structure and interpretation of computer programs*, 1985.

1.1 Il concetto di algoritmo

Il concetto centrale nella risoluzione di un problema è la nozione di algoritmo. Nella vita quotidiana, ognuno di noi si imbatte continuamente in algoritmi: istruzioni di uso di un elettrodomestico, ricette di cucina, determinazione del numero di codice fiscale, iter di una pratica in un ufficio, ...

Una prima definizione informale di algoritmo è la seguente (vedi Vocabolario della lingua italiana, Treccani): un procedimento per risolvere un problema.

La parola algoritmo deriva dal nome del matematico arabo Mohammed Ibn Musa Al-Khowaritzmi (Khowaritzmi, l'attuale Khiva in Uzbekistan, è il luogo di nascita), vissuto a Bagdad nel IX secolo. È noto per aver sviluppato, nel suo libro di Algebra notissimo a quel tempo, metodi (cioè algoritmi) per la risoluzione di alcuni problemi di aritmetica e algebra e per averli descritti come una lista di istruzioni, molto simile a una ricetta.

In particolare, spesso ci troviamo a eseguire personalmente un algoritmo (per es. la moltiplicazione di due numeri a più cifre), oppure individuiamo un nuovo procedimento (per es. una nuova ricetta di cucina).

Legati al concetto di algoritmo sono l'esecutore e il progettista dell'algoritmo. La necessità di una comunicazione tra progettista ed esecutore implica che la descrizione dell'algoritmo debba essere fatta in un linguaggio comprensibile dall'esecutore.

Un algoritmo viene progettato (e descritto) per essere eseguito (da un esecutore); la sua esecuzione consente di risolvere un problema.

È importante sottolineare il legame logico tra algoritmo ed esecutore.

L'esecutore deve avere la capacità di comprendere le *azioni* da compiere, che sono indicate dall'algoritmo; tali azioni devono essere descritte sotto forma di *operazioni elementari* eseguibili dall'esecutore. Per esempio, una ricetta di cucina viene eseguita da una persona che è in grado di comprendere il linguaggio in cui è scritto il testo della ricetta e di compiere le azioni ivi richieste (accendere il fuoco, sbattere le uova, controllare il tempo di cottura, etc.). L'esecuzione di una operazione elementare richiede una certa quantità di tempo e produce un effetto ben specificato.

Quando si progetta e si descrive un algoritmo, devono essere note le capacità dell'esecutore.

In definitiva, un algoritmo è un modo per indicare all'esecutore che deve essere effettuata una certa sequenza di operazioni ad esso note, che perciò possiamo definire elementari.

Come primo esempio, si consideri il seguente algoritmo, che risolve il problema del cambio di una ruota bucata di un'auto. L'algoritmo è progettato per un esecutore per cui sono elementari le seguenti operazioni: avvitare/svitare i bulloni di una ruota, individuare la ruota bucata/di scorta, mettere/togliere una ruota, alzare/abbassare un'auto.

```
algoritmo 1.
svitare i bulloni della ruota bucata
sollevare l'auto
togliere la ruota bucata
mettere la ruota di scorta
abbassare l'auto
avvitare i bulloni della ruota di scorta
fine algoritmo 1.
```

L'algoritmo denota esplicitamente la sequenza delle operazioni che l'esecutore deve effettuare, indicando l'ordine temporale secondo cui devono essere eseguite: prima svitare i bulloni, poi sollevare l'auto, poi togliere la ruota, ...

L'algoritmo è descritto da un testo, organizzato come una sequenza di *istruzioni*, ognuna delle quali indica una operazione elementare. L'algoritmo *non è ambiguo*, nel senso che tutte le operazioni coinvolte sono

eseguibili dall'esecutore. Inoltre, l'algoritmo denota una sequenza *finita* di operazioni; ciò implica che il processo di esecuzione dell'algoritmo ha termine in un tempo finito. Infine, si osservi che ogni operazione agisce su un oggetto e ne modifica il suo stato: per es. `sollevare l'auto` agisce sull'auto e ne modifica lo stato (dallo stato *auto abbassata* allo stato *auto sollevata*).

Il risultato dell'esecuzione dell'algoritmo è l'auto con la ruota cambiata, che rappresenta la risoluzione del problema.

Talvolta può risultare conveniente utilizzare una convenzione grafica per la descrizione di un algoritmo, detta *flowchart*[1] (*diagramma di flusso*), come in Fig. 1.1.

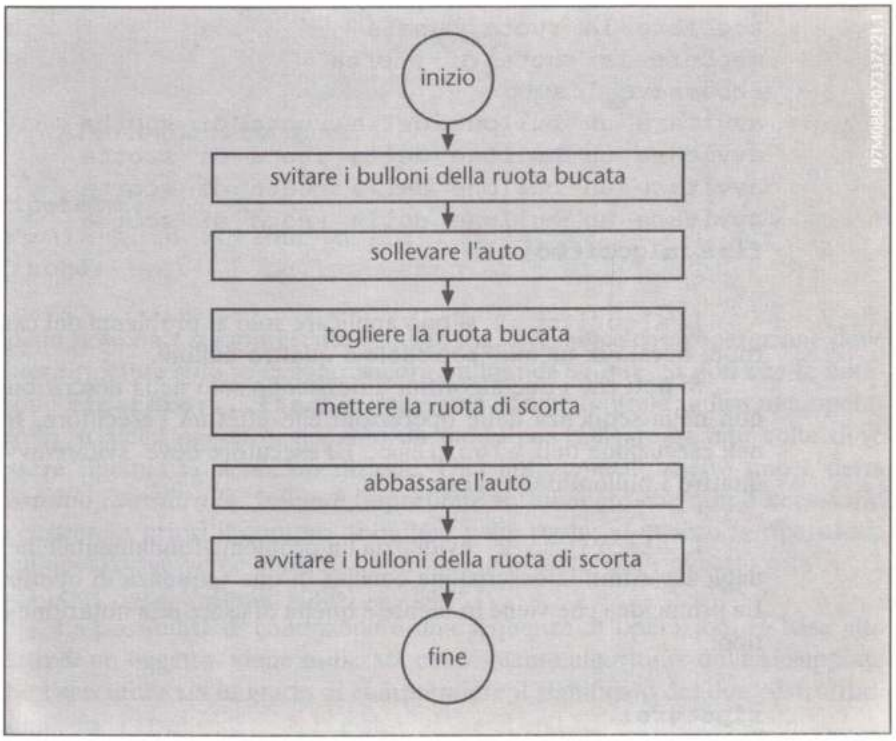


Fig. 1.1: flowchart di algoritmo 1

La caratteristica di non ambiguità di un algoritmo dipende dalle capacità dell'esecutore. Si supponga che l'`algoritmo 1` debba essere eseguito da un esecutore per il quale l'operazione `svitare/avvitare i bulloni` non

[1] La notazione del flowchart fu introdotta dal matematico John von Neumann nel 1946.

sia elementare, mentre l'operazione `svitare/avvitare un bullone` sia elementare.

Ciò significa che l'algoritmo deve descrivere l'azione di `svitare/avvitare` i bulloni in termini di una sequenza di operazioni elementari del tipo `svitare/avvitare un bullone`.

```
algoritmo 2.
svitare un bullone della ruota bucata
svitare un bullone della ruota bucata
svitare un bullone della ruota bucata
svitare un bullone della ruota bucata
sollevare l'auto
togliere la ruota bucata
mettere la ruota di scorta
abbassare l'auto
avvitare un bullone della ruota di scorta
avvitare un bullone della ruota di scorta
avvitare un bullone della ruota di scorta
avvitare un bullone della ruota di scorta
fine algoritmo 2.
```

L' `algoritmo 2` si può applicare solo al problema del cambio di una ruota bucata di un'auto con ruote a quattro bulloni.

Si noti che i due algoritmi differiscono solo nella descrizione formale e non nella sequenza delle operazioni che effettua l'esecutore. Infatti anche nell'esecuzione dell' `algoritmo 1` l'esecutore deve svitare/avvitare tutti e quattro i bulloni della ruota.

L' `algoritmo 2` evidenzia un problema fondamentale nello sviluppo degli algoritmi: la descrizione concisa di una sequenza di operazioni uguali. La prima idea che viene in mente è quella di usare una notazione intuitiva del tipo:

```
ripetere:
svitare un bullone della ruota bucata
```

che in flowchart si rappresenta come in Fig. 1.2.

In questo modo, però, si indicherebbe una *sequenza infinita* (*loop*, letteralmente: *circolo*) di operazioni.

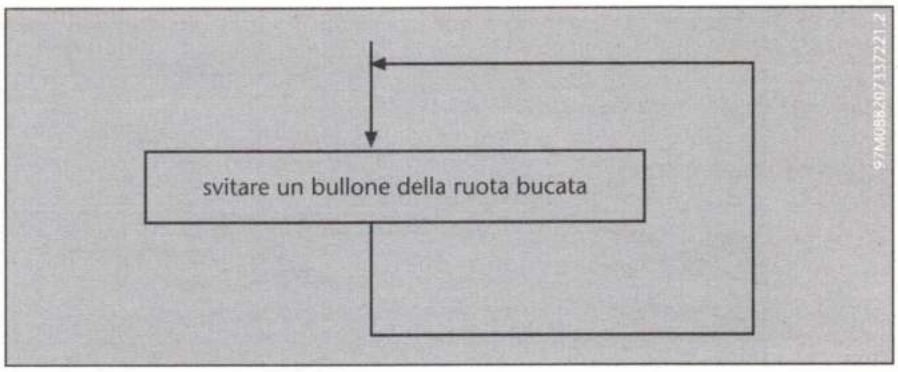


Fig.1.2: loop

Al contrario la notazione:

```
ripetere[2]:
 svitare un bullone della ruota bucata
finché: tutti i bulloni sono svitati
```

che in flowchart si rappresenta come in Fig. 1.3, indica che l'operazione deve essere ripetuta solo se ci sono ancora bulloni da svitare. Si noti che la notazione **`ripetere:...finché`**, non è una istruzione (non indica una operazione su alcun oggetto), ma solo un modo per indicare quante volte deve essere ripetuta la stessa operazione. Una notazione di questo tipo è detta *costrutto di controllo*. Infine è importante sottolineare che non è necessario conoscere a priori il numero di bulloni nella ruota, in quanto la ripetizione dell'operazione viene controllata in base allo stato degli oggetti: `tutti i bulloni sono` (nello stato) `svitati`.

La possibilità di condizionare una sequenza di operazioni in base allo stato di un oggetto viene utilizzata nel seguente algoritmo, dove si suppone che l'esecutore sia in grado di comprendere il significato dei due costrutti di controllo `ripetere:...finché` e `se...allora...altrimenti`[3].

[2] Nei capitoli successivi viene utilizzata la terminologia inglese standard `repeat ... until` invece di `ripetere ... finchè`.

[3] Nei capitoli successivi viene utilizzata la terminologia inglese standard `if ... then ... else ... endif` invece di `se ... allora ... altrimenti`.

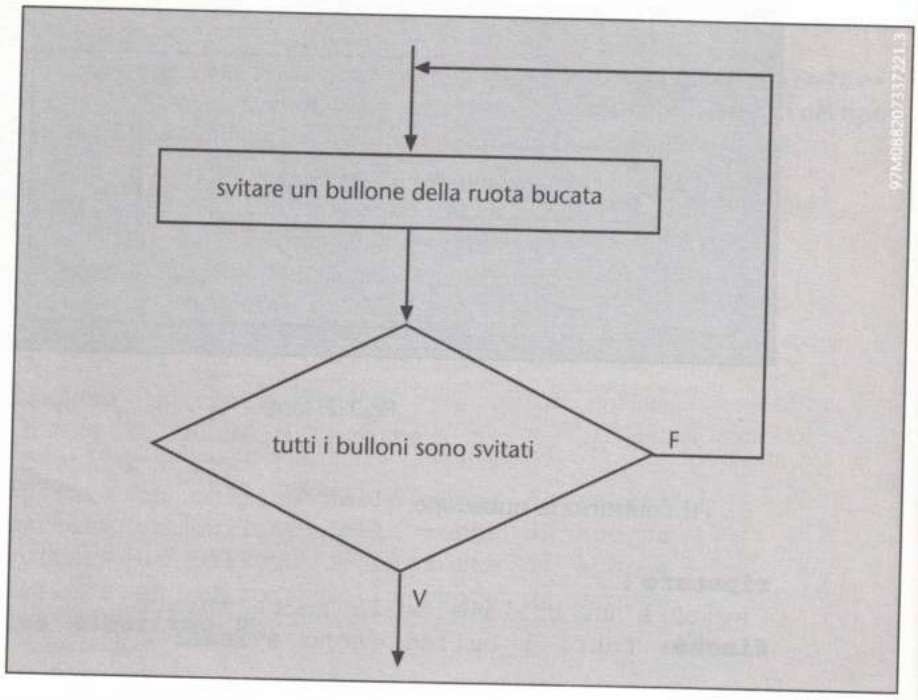


Fig. 1.3: costrutto di controllo di ripetizione

```
algoritmo 3.
se: c'è la ruota di scorta allora:
   ripetere:
    svitare un bullone della ruota bucata
   finché: tutti i bulloni sono svitati
   sollevare l'auto
   togliere la ruota bucata
   mettere la ruota di scorta
   abbassare l'auto
   ripetere:
    avvitare un bullone della ruota di scorta
   finché: tutti i bulloni sono avvitati
altrimenti:
   chiamare il meccanico
fine algoritmo 3.
```

La versione grafica dell'algoritmo 3 è riportata nel flowchart di Fig. 1.4.

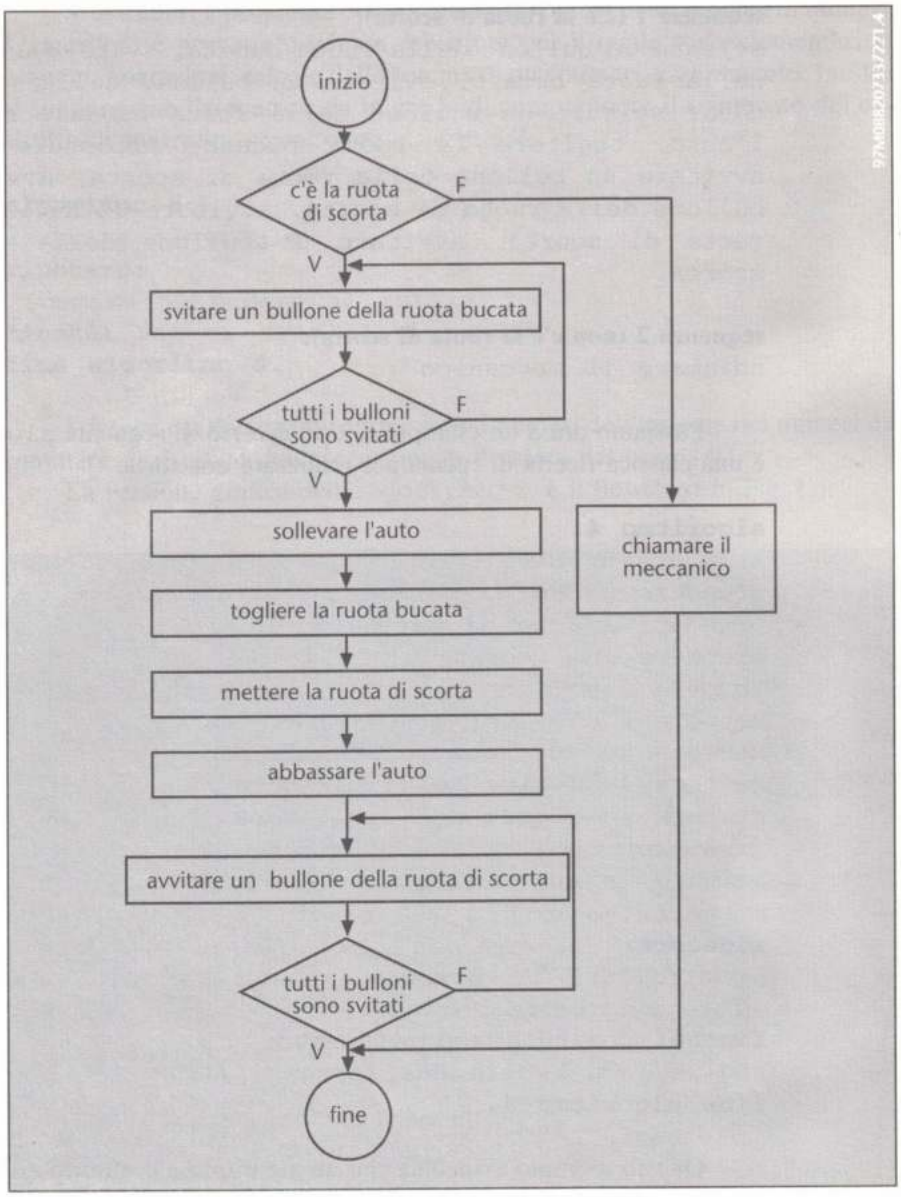


Fig. 1.4: flowchart di algoritmo 3.

L'esecuzione dell'`algoritmo 3` può dar luogo a diverse sequenze di operazioni, in dipendenza del numero di bulloni di una ruota e della presenza o meno della ruota di scorta. Nel caso di ruote a quattro bulloni le due sequenze possibili sono:

sequenza 1 (c'è la ruota di scorta):
svitare un bullone della ruota bucata, svitare un bullone della ruota bucata, svitare un bullone della ruota bucata, svitare un bullone della ruota bucata, sollevare l'auto, togliere la ruota bucata, abbassare l'auto, avvitare un bullone della ruota di scorta, avvitare un bullone della ruota di scorta, avvitare un bullone della ruota di scorta, avvitare un bullone della ruota di scorta.

sequenza 2 (non c'è la ruota di scorta):
chiamare il meccanico

Passiamo ora a un esempio di tipo diverso. Il seguente algoritmo 4 è una classica ricetta di cucina per realizzare una torta.

```
algoritmo 4.
prendere un uovo
prendere un Kg di farina
prendere 1/2 kg di zucchero
prendere uno yogurt
rompere l'uovo
mettere l'uovo nel frullatore
mettere lo zucchero nel frullatore
mettere la farina nel frullatore
mettere lo yogurt nel frullatore
frullare
versare in una teglia
mettere in forno a 200 gradi
ripetere:
 controllare la doratura
 fare altro per 5 minuti
finché: doratura soddisfacente
togliere la teglia dal forno
fine algoritmo 4.
```

Questo esempio evidenzia che un algoritmo è costituito non solo dalle istruzioni, che indicano la sequenza di azioni da effettuare, ma anche dall'insieme degli oggetti su cui operano tali azioni, cioè i dati.

I dati sono parte integrante di un algoritmo[4] e, in particolare, le relazioni che esistono tra i dati giocano un ruolo centrale nel progetto dell'algoritmo.

[4] Algoritmo= istruzioni + dati.

Si consideri il problema del calcolo della somma di un insieme di numeri. L'algoritmo è progettato per un esecutore per il quale sono elementari le seguenti operazioni: calcolo della somma di due numeri, azzeramento. Inoltre si suppone che l'esecutore sia in grado di comprendere il significato del costrutto di controllo `ripetere:...finché`.

```
algoritmo 5.
azzerare il totale[5]
ripetere:
 sommare un numero al totale
finché: non ci sono altri numeri
fine algoritmo 5.
```

I dati su cui l'algoritmo agisce sono i numeri dell'insieme dei numeri da sommare. Il risultato dell'esecuzione è il valore del `totale`.

La versione grafica dell'`algoritmo 5` è il flowchart in Fig. 1.5.

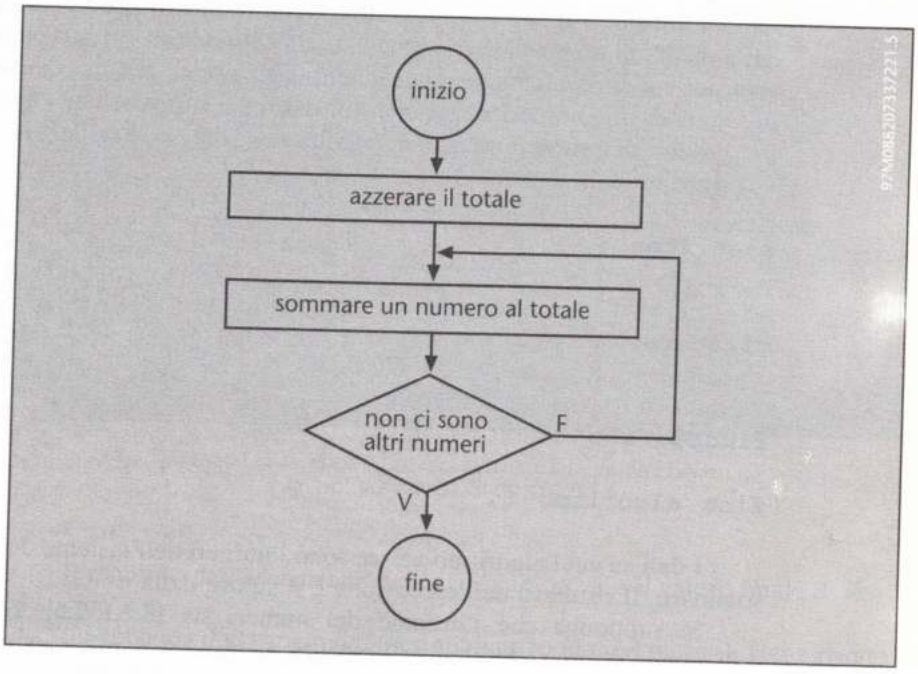


Fig. 1.5: flowchart di algoritmo 5.

[5] Prima di cominciare a sommare, ovviamente il totale, cioè la somma, è uguale a zero.

Si supponga che l'insieme dei numeri da sommare sia {5,7,10,-2}. L'esecuzione dell'`algoritmo 5` dà luogo all'esecuzione della seguente sequenza di operazioni elementari:

```
totale è 0, totale è 0+5, totale è 5+7, totale è 12+10,
totale è 22-2.
```

Il risultato dell'esecuzione dell'`algoritmo 5` è il valore finale di `totale`, cioè 20.

L'`algoritmo 5` risolve il problema della somma di un insieme costituito da un numero qualunque di numeri.

È doveroso sottolineare che molti aspetti di tale algoritmo sono lasciati volutamente a livello intuitivo (per esempio, si è implicitamente supposto che l'esecutore sia in grado di prelevare uno dopo l'altro i numeri dell'insieme). Una trattazione più approfondita è presentata nel capitolo 3.

Si consideri il seguente algoritmo, che risolve il problema del calcolo della media di un insieme di numeri. L'algoritmo è progettato per un esecutore per cui sono elementari le seguenti operazioni: somma di due numeri, divisione di due numeri, azzeramento. Inoltre si suppone che l'esecutore sia in grado di comprendere il significato del costrutto di controllo `ripetere:...finché`.

```
algoritmo 6.
azzerare il totale
azzerare il contatore
ripetere:
 sommare un numero al totale
 sommare 1 al contatore
finché: non ci sono altri numeri
la media è il totale diviso il contatore
fine algoritmo 6.
```

I dati su cui l'algoritmo agisce sono i numeri dell'insieme dei numeri da sommare. Il risultato dell'esecuzione è il valore della `media`.

Si supponga che l'insieme dei numeri sia {8,2,1,-2,6}. L'esecuzione dell'`algoritmo 6` dà luogo all'esecuzione della seguente sequenza di operazioni elementari:

```
totale è 0, contatore è 0, totale è 0+8, contatore è 1,
totale è 8+2, contatore è 2, totale è 10+1, contatore è
3, totale è 11-2, contatore è 4, totale è 9+6, contatore
è 5, media è 15/5.
```

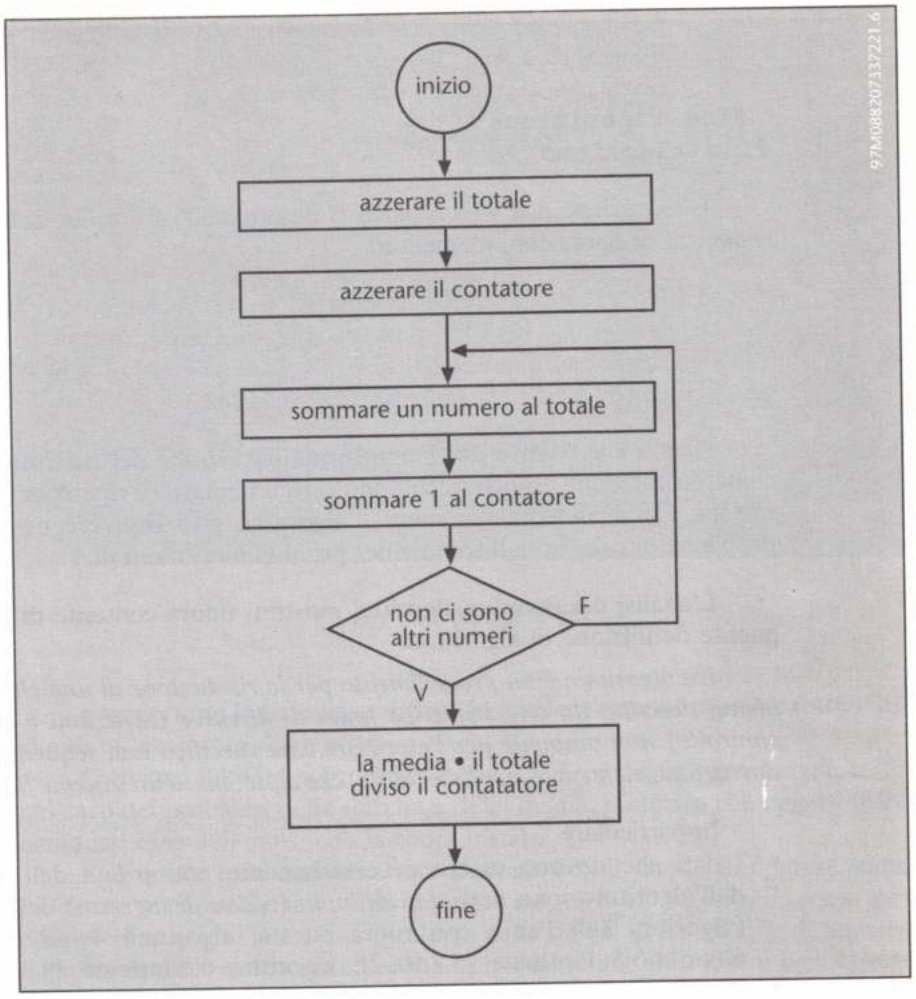


Fig 1.6: flowchart di algoritmo 6.

Il risultato dell'esecuzione dell'`algoritmo 6` è il valore finale di `media`, cioè 3.

Il problema della somma dei primi quattro numeri naturali viene risolto dal seguente `algoritmo 7`:

```
algoritmo 7.
azzerare il totale
il numero da sommare è 0
ripetere 4 volte:
```

```
        numero da sommare è numero da sommare precedente
        aumentato di 1
        sommare  numero da sommare al totale
  fine ripetizione
fine algoritmo 7.
```

L'esecuzione dell'`algoritmo 7` determina l'esecuzione della seguente sequenza di operazioni elementari:

`totale è 0; numero da sommare è 0; numero da sommare è 1, totale è 1 (0+1); numero da sommare è 2, totale è 3 (1+2); numero da sommare è 3, totale è 6 (3+3); numero da sommare è 4, totale è 10 (6+4).`

Si noti che è stato usato una forma particolare del costrutto di ripetizione, in cui viene esplicitamente indicato il numero di ripetizioni delle operazioni. Cambiando tale numero, l'algoritmo può risolvere un qualunque problema di calcolo dell somma dei primi numeri naturali.

L'analisi dei semplici algoritmi mostrati finora consente di dare la seguente definizione di algoritmo:

Un algoritmo è un procedimento per la risoluzione di una classe di problemi, descritto da una sequenza finita di direttive (istruzioni e costrutti di controllo) non ambigue per l'esecutore, che specifica una sequenza finita di operazioni, elementari per l'esecutore, che agiscono su un insieme finito di dati.

In particolare

- i dati che devono essere necessariamente noti *prima* dell'esecuzione dell'algoritmo sono detti *dati di input* (o *dati di ingresso*) dell'algoritmo (algoritmi 1-3: l'auto con ruota bucata; algoritmo 4: gli ingredienti; algoritmo 5: l'insieme {5,7,10,-2}; algoritmo 6: l'insieme {8,2,1,-2,6})
- il risultato dell'esecuzione dell'algoritmo è detto *dato* (o *dati*) *di output* (o *dati di uscita*) dell'algoritmo (algoritmi 1-3: l'auto con ruota cambiata; algoritmo 4: la torta; algoritmo 5: il valore 20; algoritmo 6: il numero 3)
- i dati che non sono né di input né di output sono detti *dati locali (o dati intermedi)* dell'algoritmo (`algoritmo 6`: tutti i valori parziali assunti dal `totale` e dal `contatore`).

Il progetto di un algoritmo richiede una fase di analisi del problema da risolvere, nella quale si devono individuare sia le entità note del problema (i dati di input) sia le entità incognite (i dati di output), che sono la soluzione del problema.

L'esecuzione di un algoritmo è un processo che a partire dai dati di input genera i dati di output, attraverso una sequenza di operazioni che coinvolgono i dati di input e i dati locali.

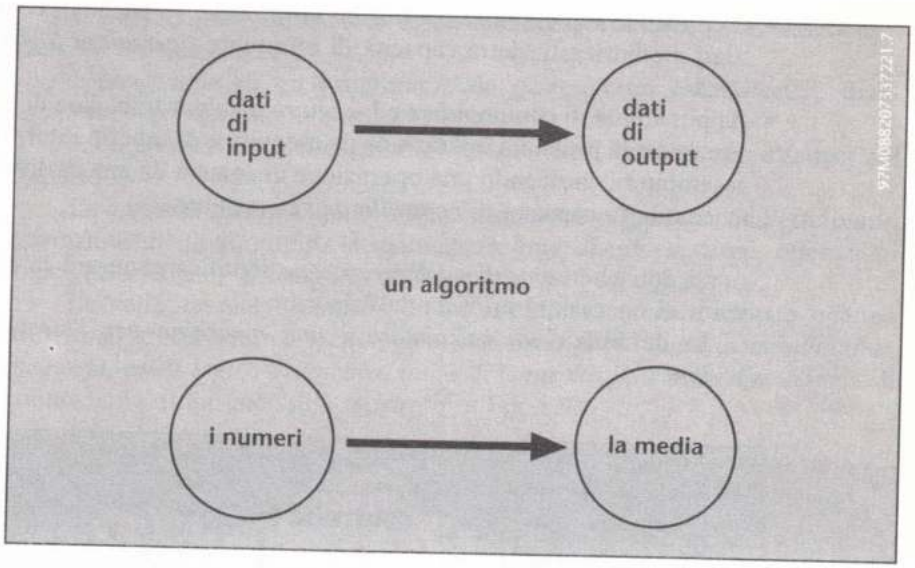


Fig. 1.7: algoritmo 5

È opportuno individuare, almeno in maniera semplificata, le caratteristiche di un esecutore di algoritmi.

Conviene limitarsi a problemi in cui *i dati sono numeri o simboli*, come nel caso dei problemi della somma e della media, piuttosto che oggetti fisici, come nel caso dell'auto con la ruota bucata.

Può essere utile suggerire una analogia con il modo di operare di un esecutore umano. Se un uomo dovesse eseguire un algoritmo, come per esempio l'`algoritmo 5` per il calcolo della media di un insieme di numeri, il suo comportamento consisterebbe tipicamente nel prelevare e annotare le informazioni, operare le trasformazioni sui dati previste dall'algoritmo (per esempio utilizzando una calcolatrice per effettuare le operazioni aritmetiche), riscrivere i dati intermedi sugli appunti e infine comunicare all'esterno i dati di output, come sintetizzato in Fig. 1.8.

In generale, un esecutore di algoritmi deve avere

- la possibilità di comunicare con l'esterno, detta capacità di effettuare *operazioni di input/output*;[6]
- la possibilità di conservare informazioni, detta capacità di effettuare *operazioni di memorizzazione di informazioni (dati e direttive)*;

[6] Spesso input/output si indica con le iniziali maiuscole delle due parole: I/O.

- la possibilità di eseguire operazioni aritmetiche (o anche di confronto) su dati memorizzati, detta capacità di effettuare *operazioni logico-aritmetiche*;
- la possibilità di *comprendere* ed *eseguire* un algoritmo, cioè di individuare quale sia la prossima operazione da eseguire e di saperla interpretare (per esempio riconoscendo una operazione di somma da una di divisione), che viene detta capacità di *controllo dell'esecuzione*.

Si noti che *controllo dell'esecuzione* significa generare la corretta sequenza di operazioni prevista dall'algoritmo.

In definitiva, *un calcolatore è una macchina per l'esecuzione degli algoritmi.*

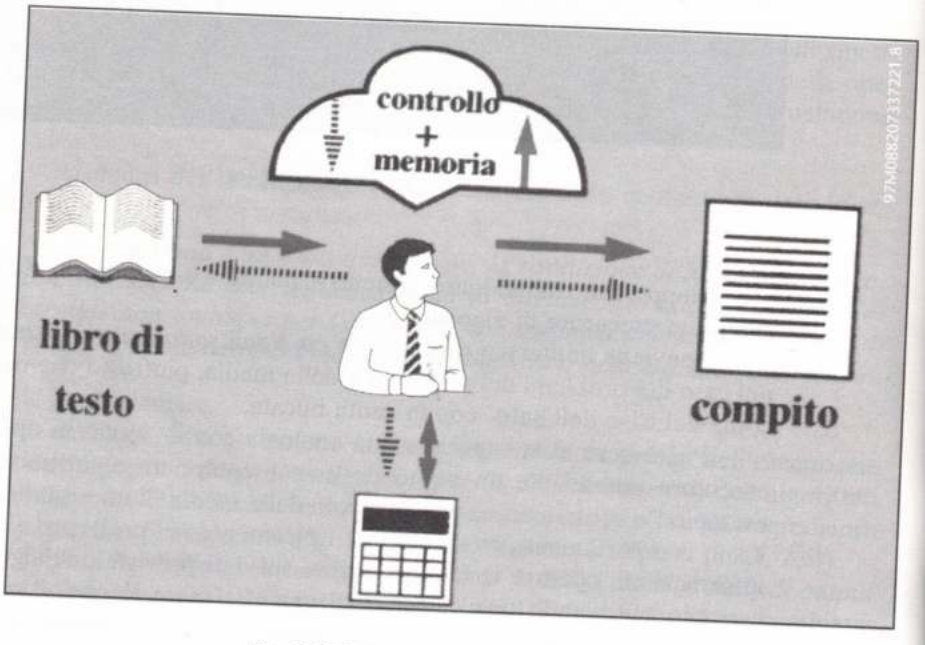


Fig. 1.8: Schema calcolatore uomo

La risoluzione *computazionale* di un problema consiste nel progetto di un algoritmo, in genere da parte di un esperto (intervento umano), e nell'esecuzione dell'algoritmo da parte del calcolatore.

Se un problema è riconducibile a un insieme di dati (dati di input), se la soluzione del problema è un insieme di dati ed è calcolabile a partire dai dati di input eseguendo un algoritmo, allora il problema è risolvibile mediante un calcolatore.

È consuetudine chiamare *programma* un algoritmo che deve essere eseguito da un calcolatore[7].

L'esecuzione di un programma da parte di un calcolatore è detta *computazione* o *processo computazionale*.

L'organizzazione dei calcolatori attuali viene brevemente discussa nel capitolo 2.

Le capacità macroscopiche di un calcolatore consistono nell'effettuare: operazioni di input/output, la memorizzazione di informazioni, operazioni logico-aritmetiche, il controllo dell'esecuzione di un algoritmo.

Le capacità macroscopiche di un calcolatore sono realizzate ognuna attraverso un dispositivo tecnologico; essi sono detti *unità di input/output*, *memoria*, *unità logico-aritmetica*, *unità di controllo*. Un possibile schema di connessione di tali moduli è mostrato in Fig. 1.9.

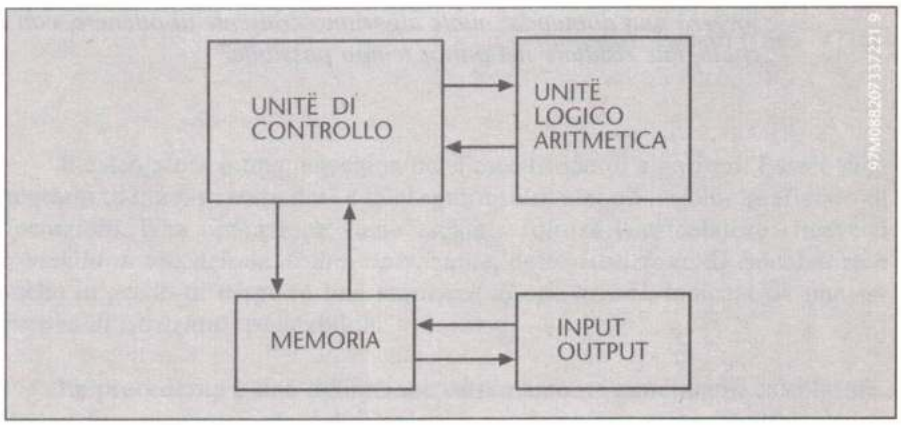


Fig. 1.9: interconnessione dei moduli, a immagine del "calcolatore uomo"

L'individuazione di procedimenti per la risoluzione di problemi computazionali è stata una delle attività principali della Matematica fin dall'antichità e ha costituito il fondamento del pensiero logico/quantitativo. Non a caso, risalgono al periodo greco classico alcuni algoritmi fondamentali come l'algoritmo di Euclide per il calcolo del massimo comun divisore di due numeri naturali (vedi par. 3.1.6) e l'algoritmo di Archimede per il calcolo di una approssimazione di π.

Tuttavia, solo negli ultimi cinquanta anni l'insieme delle attività legate allo sviluppo e all'analisi degli algoritmi si è andato configurando come una disciplina scientifica. Ciò ha coinciso con l'avvento del calcolatore. Gli attuali

[7] Nell'uso corrente, si tende a sottolineare che il programma è una particolare descrizione dell'algoritmo secondo una notazione standard detta linguaggio di programmazione.

(2002) calcolatori sono in grado di eseguire almeno centinaia di milioni di operazioni aritmetiche (o logiche) al secondo (per esempio i diffusissimi *personal computer*) e raggiungere, nel caso dei calcolatori più potenti e costosi (i cosiddetti *supercalcolatori*), velocità di centinaia di miliardi di operazioni al secondo; quindi è possibile risolvere problemi di grande complessità in tempi accettabili per le esigenze applicative concrete. L'efficacia del binomio *calcolatore-algoritmo* nella risoluzione dei problemi più disparati ha creato la *disciplina computazionale* (*computing*) che oggi pervade quasi tutti i campi del lavoro intellettuale.

È istruttivo ricordare a tale proposito ciò che Charles Babbage[8] scriveva nel 1864, riguardo alla sua Analytical Engine, considerata il primo calcolatore programmabile.

Non appena la Analytical Engine esisterà, essa necessariamente guiderà il corso futuro della scienza. Ogni qualvolta si otterrà un risultato con il suo aiuto, sorgerà una domanda: quale algoritmo consente di ottenere, con questa macchina, tale risultato nel minor tempo possibile?

[8] Charles Babbage (Londra 1791-1871) è stato professore di matematica della cattedra Lucasiana (quella che fu già di Newton) a Cambridge.

Capitolo secondo

Struttura del calcolatore

Non è degno di uomini eccellenti perdere ore come schiavi nell'attività manuale di calcolare, che potrebbe essere sicuramente demandata a una macchina

Gottfried W. Leibniz (1646 - 1716)

Il calcolatore è una macchina per l'esecuzione di algoritmi. Esso è cioè in grado di memorizzare dati e di eseguire su di essi un insieme prefissato di *operazioni*. Una operazione viene eseguita solo se il calcolatore riceve il *comando di esecuzione di una operazione*, detto *istruzione*. Il calcolatore è anche in grado di eseguire una *sequenza di operazioni*, indicata da una *sequenza di istruzioni* prestabilita.

La precedente è una definizione estremamente generale di calcolatore. Tuttavia un uso cosciente del calcolatore rende necessario possedere alcune informazioni sulla sua *struttura*. Come è usuale nel caso di macchine, o in generale di sistemi complessi, l'analisi procede attraverso una decomposizione in componenti via via più semplici. Il modo più conveniente per analizzare la struttura del calcolatore, che è la macchina più complessa finora costruita dall'uomo, consiste nel vederlo come una *gerarchia di livelli*. Ogni livello, che può essere considerato come una macchina, è in grado di eseguire compiti (cioè istruzioni-operazioni) specifici.

I due livelli principali in cui è naturale decomporre un calcolatore sono:

- il livello *hardware*;
- il livello *software*.

Entrambi i livelli, come vedremo, sono a loro volta decomponibili in altri livelli.

L'*hardware* è costituito da componenti elettroniche, meccaniche e ottiche. Il *software* è costituito da un insieme di programmi; come vedremo, in genere, hardware + software costituiscono la *macchina virtuale*.

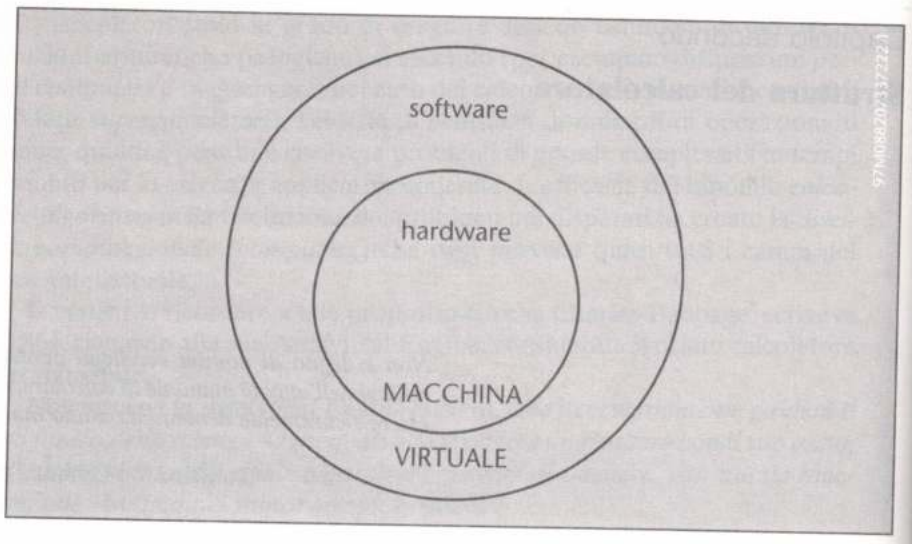


Fig. 2.1: il calcolatore: visione per livelli fisico/virtuale

L'utilizzatore di un calcolatore interagisce con esso, normalmente, attraverso il livello più alto della gerarchia, cioè il software. Si noti che hardware e software sono da considerare logicamente equivalenti, dal punto di vista dell'utilizzatore del calcolatore. Infatti in generale l'utilizzatore di un calcolatore non sa quali operazioni sono effettuate dall'hardware e quali dal software; il risultato di una operazione effettuabile dall'hardware può essere ottenuto via software (cioè l'operazione può essere simulata da un programma) e viceversa il risultato di un programma può essere ottenuto come operazione effettuata da un apposito hardware.

2.1 La macchina fisica

L'hardware è l'insieme delle componenti elettroniche, meccaniche e ottiche del calcolatore. La parte preponderante dell'hardware è costituita da componenti elettroniche, mentre componenti meccaniche e ottiche sono attualmente presenti nei cosiddetti dispositivi di ingresso/uscita (I/O). Le componenti elettroniche sono basate sui principi dell'*elettronica digitale*.[1]

Nel descrivere l'organizzazione dell'hardware è usuale utilizzare i tre termini: architettura, organizzazione, impiantistica.

[1] L'*elettronica digitale* ha come obiettivo il progetto di sistemi che funzionano in *modo binario*. Un sistema digitale è formato da componenti che possono assumere solo due *stati*. Questi due stati, per così dire *quantizzati*, sono chiamati in vari modi: vero, falso; 0,1; alto, basso; impulso, mancanza di impulso; chiuso, aperto. I sistemi digitali, come i calcolatori, i sistemi di commutazione, i sistemi di controllo, etc., sono ottenuti connettendo semplici

L'architettura si riferisce alle caratteristiche funzionali dei sottosistemi principali che costituiscono l'hardware e delle loro interconnessioni.

L'organizzazione si riferisce ai criteri con cui viene realizzato il progetto architetturale, e cioè l'insieme delle scelte pratiche di implementazione.

L'impiantistica copre gli aspetti più legati all'elettronica e alla meccanica dei singoli sottosistemi, per esempio tipi di circuiti elettronici e di cavi di connessione da utilizzare, sistema di raffreddamento, requisiti elettrici, etc.

Nel seguito discuteremo alcuni aspetti dell'architettura del calcolatore, con brevi cenni all'organizzazione.

I sottosistemi fondamentali dell'hardware sono la *memoria*, il *processore* e i *dispositivi di I/O*.

La memoria è una componente passiva dove vengono immagazzinate le informazioni (dati e programmi), che poi vengono utilizzate dagli altri sottosistemi.

Il processore è la componente principale dell'hardware. Esso è in grado di eseguire un programma, interpretandone le istruzioni ed eseguendo le corrispondenti operazioni. Il processore è l'unica componente attiva, nel senso che genera nuove informazioni a partire dalle informazioni presenti in memoria. L'insieme delle istruzioni (da questo momento useremo i termini istruzione e operazione come sinonimi) eseguibili dal processore costituisce il *linguaggio macchina*.

I dispositivi di I/O consentono l'interazione con l'esterno, cioè il trasferimento di informazioni dall'esterno verso l'hardware (input) e dall'hardware verso l'esterno (output); vengono considerati dispositivi di I/O anche le cosiddette memorie di massa (dischi, dischetti, CD-rom, nastri,...).

L'interconnessione fisica tra i vari sottosistemi è realizzata mediante i *canali di comunicazione*.

Lo schema in Fig. 2.2 riassume i sottosistemi principali e mostra la loro interconnessione.

Questo schema di interconnessione, insieme con le funzionalità proprie di ciascun sottosistema è stato suggerito per la prima volta dal matematico John von Neumann[2] nel 1946[3]. L'architettura della maggior parte dei calcola-

circuiti chiamati *porte logiche*. Le porte più comuni hanno due (fili di) ingressi e una (filo di) uscita e realizzano semplici funzioni in cui lo stato in uscita dipende dagli stati in ingresso. Tali funzioni sono dette *funzioni logiche*. I criteri con cui vengono progettati i sistemi digitali sono puramente matematici e vanno sotto il nome di *algebra logica*.

[2] John Louis von Neumann (Budapest 1903, Washington 1957) è considerato il più grande matematico del XX secolo; scienziato geniale, dotato di una erudizione enciclopedica e di una memoria prodigiosa, ha spaziato nei campi più disparati, dall'algebra alla meccanica quantistica, dalla fluidodinamica all'economia, dalla logica all'ingegneria chimica, dalla teoria degli automi al calcolo numerico.

Può essere considerato il padre dei calcolatori contemporanei, in quanto individuò le basi metodologiche sia per il progetto del calcolatore sia per il suo utilizzo.

[3] Nello schema originale, i moduli di ingresso/uscita erano connessi a voci speciali,

tori oggi in uso è riconducibile a questa idea di base, che viene detta *architettura* (o *macchina*) *di von Neumann.* La caratteristica peculiare è che il *calcolatore esegue un programma (istruzioni + dati) che è immagazzinato nella memoria.*

La prima idea di costruire una macchina generale per eseguire algoritmi è attribuita al matematico inglese Charles Babbage. Essa trae origine sia dalle esperienze legate alle macchine calcolatrici (cioè in grado di effettuare le operazioni aritmetiche), che hanno una lunga storia concretizzatasi nei progetti di Pascal (addizioni e sottrazioni), di Leibnitz (addizioni, sottrazioni, moltiplicazioni e divisioni) e dello stesso Babbage (*Difference Engine*), sia da quelle legate allo sviluppo dei telai semiautomatici (John Kay, Joseph M. Jacquard[4],..), che hanno caratterizzato l'inizio della rivoluzione industriale.

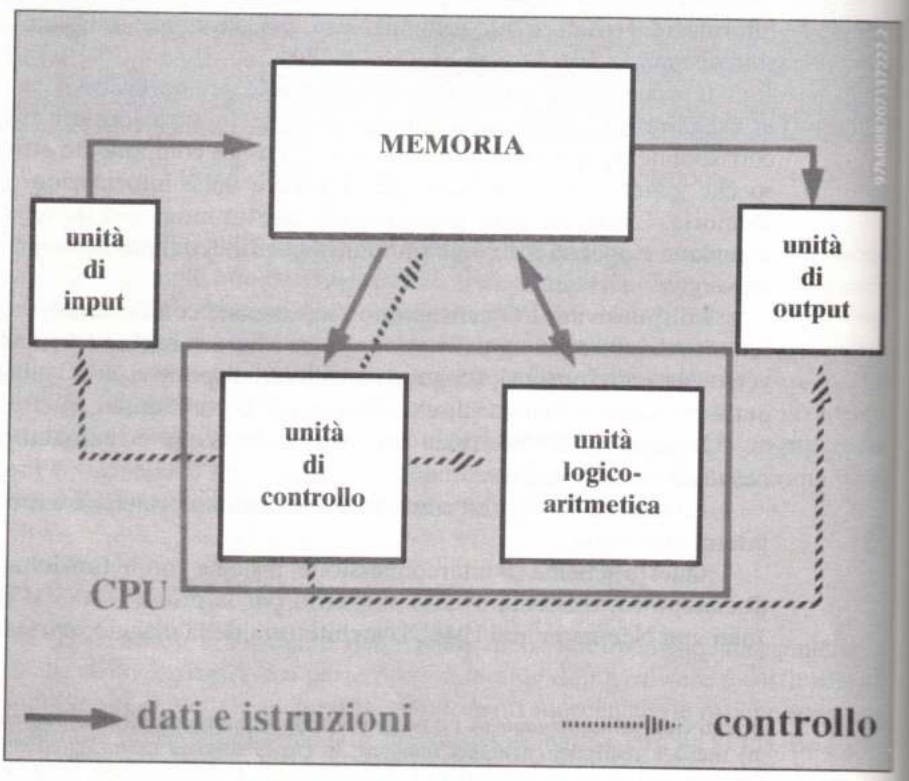


Fig. 2.2: schema di von Neumann

dette accumulatori, dell'unità logico/aritmetica. Per esempio, un dato di input veniva trasferito dall'unità di ingresso all'accumulatore e da qui nella memoria.

[4] Joseph Marie Jacquard (1752-1834) è considerato l'inventore dei telai semiautomatici: un dispositivo, attaccato al telaio, a partire da una serie di schede perforate di legno o di cartone, era in grado di guidare la tessitura, regolando i movimenti degli aghi e l'alternanza dei fili colorati.

Dall'idea originale di Babbage che progettò, senza però riuscire a realizzare, il primo calcolatore che egli chiamò *Analytical Engine*, si assiste a una evoluzione dell'architettura che ha il suo punto di arrivo nella macchina di Von Neumann.

Le innovazioni più significative che hanno caratterizzato l'evoluzione del calcolatore dopo von Neumann sono:

1947: primo esperimento che porta all'invenzione del transistor (Bell Labs.)
1952: primo calcolatore commerciale (Univac)
1956: primo linguaggio di programmazione, il Fortran (IBM)
1968: primo calcolatore con circuiti integrati (Burroughs)
1969: prima rete di calcolatori, Arpanet (Dipartimento della Difesa americano)
1972: primo videogioco (Pong)
1973: 10.000 componenti su un chip di 1 cm^2
1976: primo supercalcolatore (Cray-1)
1977: primo personal computer (PC) con schermo a colori (Apple)
1984: primi calcolatori con più processori
1990: creazione del World Wide Web (CERN)
2000: prima infrastruttura di grid computing.

L'evoluzione dell'organizzazione e dell'impiantistica del calcolatore è ovviamente legata sia a quella dell'architettura sia a quella della tecnologia elettronica, meccanica e ottica.

La classificazione in *generazioni* dell'evoluzione dei calcolatori[5] rispecchia appunto l'evoluzione della tecnologia di base:

- gli antenati: tecnologia meccanica ed elettromeccanica (relè) (-1944)
- I generazione: tecnologia dei tubi a vuoto (1945-1955)
- II generazione: tecnologia dei transistor (1955-1967)
- III generazione: tecnologia dei circuiti integrati (1967-1980)
- IV generazione: tecnologia dei circuiti ad alta integrazione (VLSI) (1980-oggi).

In questo momento si assiste all'avvento di una nuova generazione di calcolatori, in cui la novità non è la tecnologia di base, che è sempre la VLSI, ma è piuttosto di tipo architetturale: si tratta dei calcolatori a più processori (*calcolatori paralleli*).

[5] Una discussione più dettagliata su quelle che abbiamo chiamato *innovazioni* e sulla classificazione in generazioni si può trovare ad esempio in:
CSE Project `http://csep1.phy.ornl.gov/ov/node8.html` (A Brief History of Computer Technology).

2.1.1 La memoria

La memoria è il sottosistema di immagazzinamento delle informazioni (dati e istruzioni) che vengono utilizzate dal processore. Dal punto di vista architetturale, la memoria è un insieme di componenti elementari, dette locazioni, ognuna individuata univocamente da un *indirizzo*. Tale insieme può essere pensato come una tabella lineare, cioè una sequenza di locazioni consecutive. L'indirizzo di una locazione è quindi un numero intero, che individua la locazione nella tabella; due numeri interi consecutivi individuano due locazioni adiacenti.

Una locazione contiene un dato o una istruzione (o una parte di essi). In tal modo le informazioni sono individuabili in modo univoco all'interno della memoria. La localizzazione di una informazione in memoria avviene attraverso l'indirizzo della locazione dove è stata memorizzata.

L'organizzazione di una locazione è attualmente basata ancora su un'idea di von Neumann: una locazione è una tabella lineare di bit, cioè una sequenza di caselle ognuna delle quali può contenere una delle due cifre binarie 0, 1 (di qui il nome *bit*: *bi*nary digi*t*).

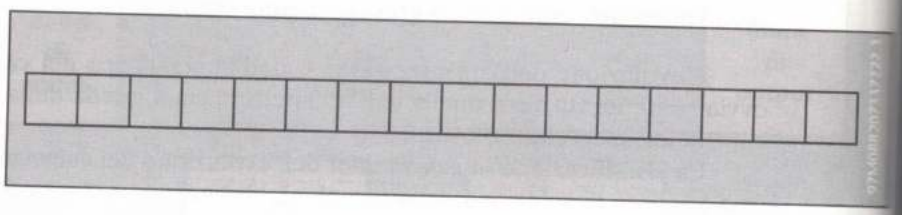


Fig. 2.3: schema logico di locazione a 16 bit

Il numero di bit che costituisce una locazione è uguale per tutte le locazioni e costituisce una scelta organizzativa importante per le prestazioni dell'hardware.

Più precisamente, infine, la *locazione è la minima quantità di bit indirizzabile*. In anni recenti, molti costruttori hanno standardizzato locazioni a 8 bit, chiamate *byte* (1 byte=8 bit). In genere le locazioni sono raggruppate in voci (o parole), per cui si parla di voci a 16 bit (2 byte), a 32 bit (4 byte), ecc.... facendo in modo che istruzioni e dati possano venire memorizzati in più byte.

Si può dire allora che la *voce è il minimo numero di bit necessario per memorizzare un dato* o una istruzione. Nel seguito useremo i termini voce e locazione come sinonimi.

La Fig. 2.4 mostra una locazione di 16 bit, insieme con i valori assunti da ogni bit.

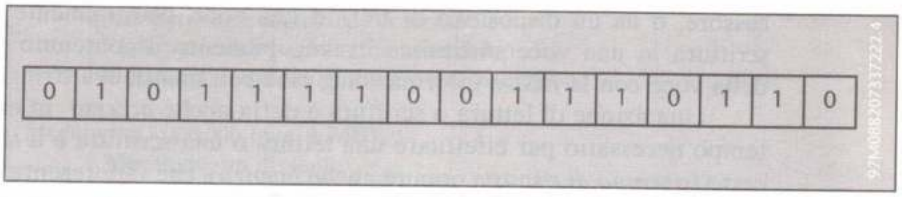


Fig. 2.4: contenuto di una locazione di 16 bit

La sequenza dei valori dei singoli bit della locazione, che diremo più semplicemente sequenza di bit (o *stringa* di bit), è l'*informazione immagazzinata* nella locazione (o *contenuto* della locazione).

Lo schema architetturale della memoria è sintetizzato in Fig. 2.5.

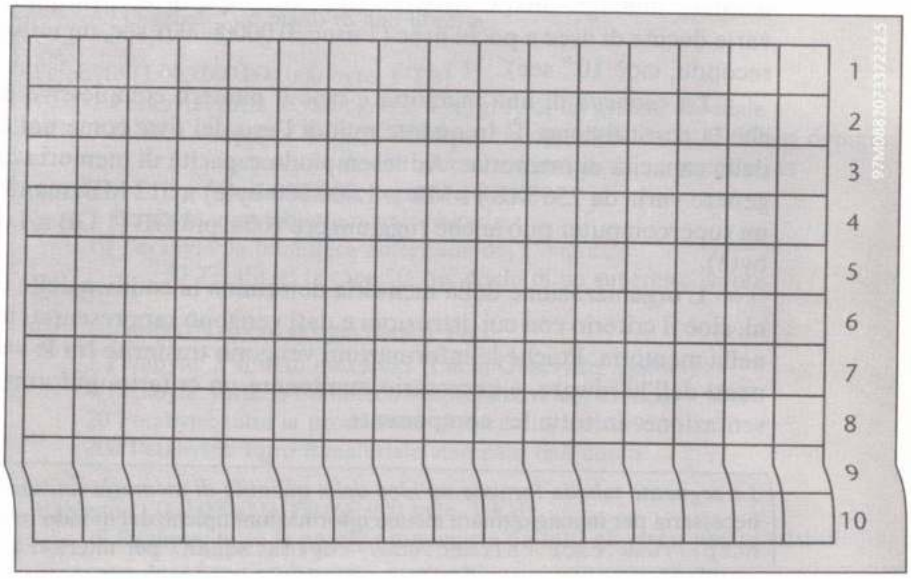


Fig. 2.5: schema di memoria con 10 locazioni di 16 bit

Le azioni elementari effettuabili sulla memoria sono la *lettura* e la *scrittura*.

La lettura è l'azione di trasferimento di una informazione immagazzinata in memoria (cioè il contenuto di una voce) verso il processore o un dispositivo di I/O. La lettura lascia inalterato il contenuto della voce di memoria su cui agisce, nel senso che l'oggetto del trasferimento è una copia della stringa di bit memorizzata nella voce.

La scrittura è l'azione di trasferimento di una informazione dal pro-

cessore, o da un dispositivo di I/O, in una voce. Naturalmente l'azione di scrittura in una voce sostituisce irreversibilmente il contenuto precedente della voce con la nuova informazione, cioè con una nuova stringa di bit.

Una azione di lettura o scrittura è detta anche *accesso* in memoria. Il tempo necessario per effettuare una lettura o una scrittura è il *tempo di accesso* (o *tempo di risposta* oppure anche *latenza*), che rappresenta una importante caratteristica macroscopica dell'organizzazione della memoria.

Si noti che la scelta architetturale della memoria come tabella di voci, ciascuna individuata da un indirizzo, e l'organizzazione effettiva consentita dalla tecnologia elettronica, permettono di accedere a qualunque voce di memoria con il medesimo tempo di accesso. Per questo motivo, la memoria è talvolta denotata con il termine RAM (Random Access Memory), a sottolineare il fatto che si può accedere in qualunque ordine e con lo stesso costo a ogni locazione (o gruppi di locazioni), cioè a qualunque informazione elementare immagazzinata. Attualmente[6], il tempo di accesso può variare da varie decine di nsec a pochi nsec (1 nsec=0.000000001 sec, un miliardesimo di secondo, cioè 10^{-9} sec).

La *capacità* di una memoria è cioè il numero complessivo di locazioni che la costituiscono. È frequente quindi l'uso del *byte* come unità di misura della capacità di memoria. Ad esempio la capacità di memoria di un PC in genere varia da 256 MB (1 MB ≅ 1.000.000 byte) a 512 MB, mentre quella di un supercomputer può anche raggiungere 100 o più GB (1 GB ≅ 1.000.000.000 byte).

L'organizzazione della memoria determina la codifica delle informazioni, cioè il criterio con cui istruzioni e dati vengono rappresentati logicamente nella memoria. Poiché le informazioni vengono trasferite tra le varie componenti dell'hardware, è necessario mantenere un criterio uniforme di rappresentazione in tutte le componenti.

La seguente tabella fornisce un'idea della quantità di memoria (misurata in byte) necessaria per immagazzinare alcune informazioni tipiche del mondo reale (vedere `http://www.cacr.caltech.edu/~roy/dataquan/` per ulteriori approfondimenti).

Byte(8 bit)
- 1 byte: un carattere dell'alfabeto
- 10 byte: una parola
- 100 byte: un telegramma

Kilobyte (1000 byte, 1 KB)
- 1 Kilobyte: un brevissimo racconto

[6] Una legge empirica ma ampiamente accettata indica che il tempo di accesso alle memorie in commercio diminuisce all'incirca del 7% all'anno. I dati riportati si riferiscono al 2000.

2 Kilobyte: una pagina dattiloscritta
10 Kilobyte: una pagina di una enciclopedia
100 Kilobyte: una fotografia a bassa risoluzione

Megabyte (1.000.000 byte, 1 MB)
1 Megabyte: un dischetto
2 Megabyte: una fotografia ad alta risoluzione
5 Megabyte: l'opera omnia di Shakespeare
10 Megabyte: un minuto di suono alta fedeltà
100 Megabyte: una enciclopedia in due volumi
500 Megabyte: un CD-ROM

Gigabyte (1.000.000.000 byte, 1 GB)
1 Gigabyte: un camioncino pieno di fogli dattiloscritti oppure una sinfonia in alta fedeltà
20 Gigabyte: l'opera omnia di Beethoven in alta fedeltà
50 Gigabyte: un piano di una libreria

Terabyte (1.000.000.000.000 byte, 1 TB)
1 Terabyte: tutte le radiografie a raggi X di un grande ospedale oppure la carta dattiloscritta ottenibile da 50000 alberi oppure la quantità di dati giornalieri inviati da un satellite per l'osservazione terrestre (EOS)
2 Terabyte: una biblioteca universitaria
10 Terabyte: la biblioteca americana del Congresso
50 Terabyte: la capacità dei dischi di un supercalcolatore

Petabyte (1.000.000.000.000.000 byte, 1 PB)
1 Petabyte: 3 anni di dati EOS (Earth Observing System)
4 Petabyte: tutte le biblioteche universitarie del mondo
20 Petabyte: tutta la produzione di hard-disk nel 1995
200 Petabytes: tutto il materiale stampato dall'uomo

Exabyte (1.000.000.000.000.000.000 byte, 1 EB)
5 Exabyte: tutte le parole pronunciate da tutti gli esseri umani esistiti!

2.1.2 Il processore

Il processore è l'esecutore delle istruzioni. L'esecuzione di una istruzione consiste nel:

- prelevare l'istruzione dalla memoria (*fetch*)
- individuare l'operazione da eseguire ed eventualmente i dati su cui essa agisce (*decode*)
- eseguire l'operazione, prelevando eventualmente i dati dalla memoria (*execute*)

L'esecuzione di un programma richiede la ripetizione delle tre azioni precedenti (detto *ciclo fetch-decode-execute*) per ognuna delle istruzioni che compongono il programma.

Le istruzioni eseguibili dai processori attualmente prodotti sono sorprendentemente elementari: lo spostamento di informazioni tra memoria e altri sottosistemi (istruzioni di trasferimento di dati), le quattro operazioni aritmetiche su due numeri e alcune operazioni logiche, come ad esempio determinare se due numeri sono uguali (istruzioni logico/aritmetiche), la scelta selettiva della successiva istruzione da eseguire (istruzioni di controllo). Questa scelta organizzativa è rimasta sostanzialmente inalterata in tutte le generazioni di calcolatori.

L'azione di *fetch* richiede che il processore conosca qual è l'istruzione da eseguire, cioè il processore deve conoscere l'indirizzo di memoria dell'istruzione da eseguire. L'istruzione viene fisicamente prelevata dalla memoria, trasferita (attraverso un canale di comunicazione) nel processore e ivi immagazzinata.

L'azione di *decode* consiste nel determinare la particolare operazione denotata dall'istruzione immagazzinata nel processore.

L'azione di *execute* effettua praticamente l'operazione. Esaminiamo due tipiche categorie di operazioni.

Le istruzioni di trasferimento di dati coinvolgono i sottosistemi tra cui avviene il trasferimento (processore-memoria, memoria-processore, memoria-memoria, memoria-I/O, I/O-memoria). Esse quindi richiedono l'indirizzo di memoria del dato da spostare ed eventualmente quello della destinazione. L'azione di execute in tal caso può effettuare un prelievo dalla memoria, da un dispositivo di I/O o dal processore, l'uso di un canale di comunicazione e l'immagazzinamento di un dato in memoria, in un dispositivo di I/O o nel processore.

Le istruzioni logico/aritmetiche agiscono su uno o due dati (operandi) e generano un dato (il risultato). Gli operandi sono dati immagazzinati in memoria, che vengono prelevati e trasferiti nel processore. La generazione del risultato è effettuata da una componente del processore detta *unità logico/aritmetica* (ALU). Il dato generato viene trasferito dal processore e immagazzinato in memoria.

L'architettura del processore può essere sintetizzata nello schema in Fig. 2.6.

Il *registro prossima istruzione* (rpi) contiene l'indirizzo di memoria dell'istruzione da eseguire (viene utilizzato nell'azione di fetch). Il *registro istruzione* contiene l'istruzione da eseguire, immagazzinata dall'azione di fetch e utilizzata dall'azione di decode. I *registri dati* contengono gli operandi, immagazzinati dall'azione di execute; il *registro risultato* contiene il risultato generato dall'ALU, sempre durante l'azione di execute.

Il registro prossima istruzione viene normalmente gestito dal controllore; al termine dell'azione di fetch, viene immagazzinato nel registro l'indirizzo

di memoria della successiva istruzione da eseguire. Le istruzioni di controllo non generano dati ma possono modificare l'indirizzo immagazzinato nel registro prossima istruzione; ciò implica una variazione dell'ordine temporale con cui vengono eseguite le istruzioni.

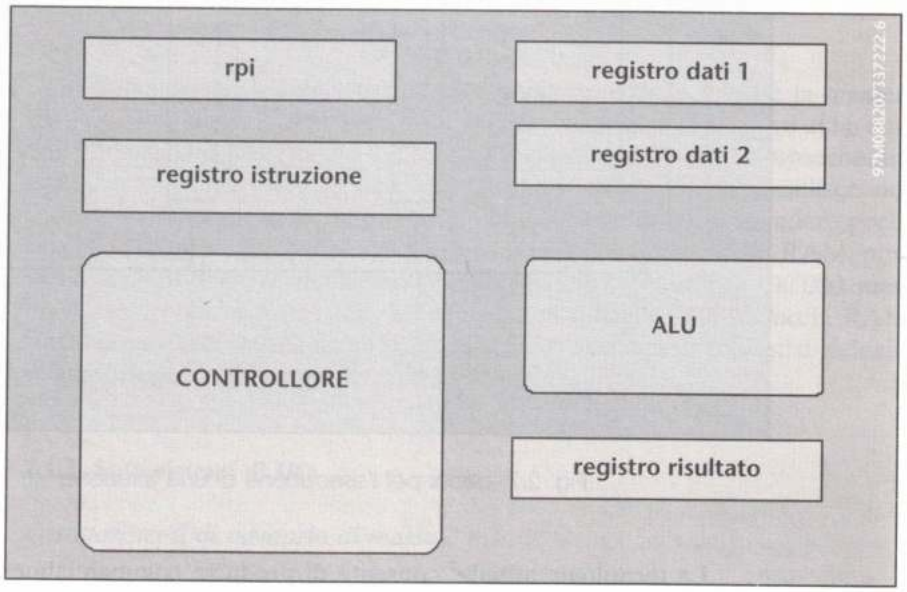


Fig. 2.6: il processore

Il numero e la struttura dei registri, e l'insieme delle istruzioni eseguibili sono scelte che riguardano l'organizzazione di un particolare processore. Queste scelte determinano la principale caratteristica macroscopica di un processore: la velocità operativa, cioè il tempo per l'esecuzione di una istruzione. In generale, istruzioni diverse richiedono differenti tempi di esecuzione. Tale tempo dipende dal tempo necessario per compiere ciascuna delle tre azioni (fetch-decode-execute) in cui viene decomposta nel processore l'esecuzione di una istruzione. Il tempo di esecuzione di una azione è un multiplo del tempo di *ciclo di clock* (o *periodo di clock*). Il motivo di ciò discende da vincoli di tecnologia elettronica digitale. In breve, in un circuito digitale complesso l'ordine temporale in cui gli eventi si succedono viene sincronizzato attraverso un circuito di controllo, detto *cadenzatore* o *clock*, che genera una sequenza di impulsi; il tempo (fisso) tra due impulsi consecutivi è il ciclo di clock.

Poiché il segnale generato dal cadenzatore è periodico, spesso viene indicata la *frequenza* di clock, cioè l'inverso del ciclo di clock.

La Fig. 2.7 mostra un esempio di organizzazione del processore in cui ogni azione del processo di esecuzione di una istruzione richiede un ciclo di clock.

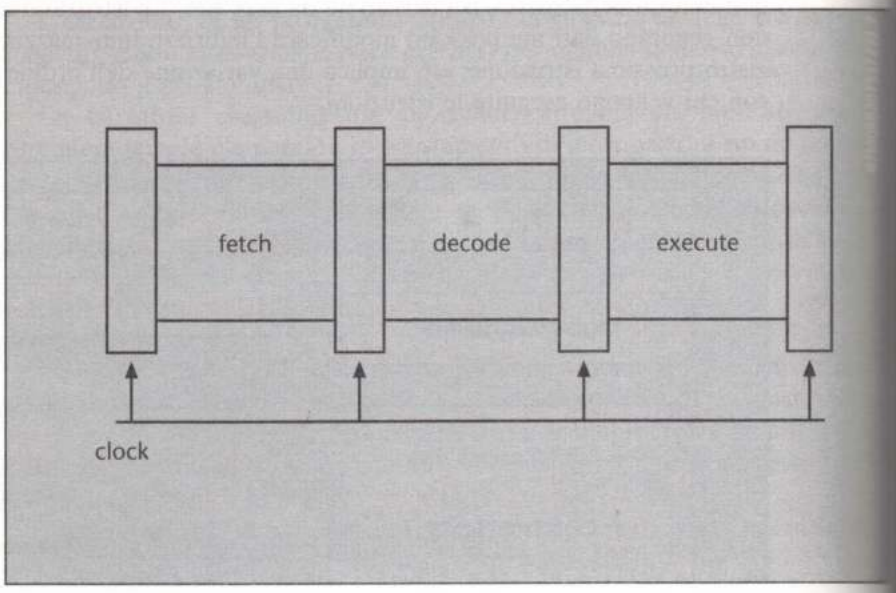


Fig. 2.7: azioni per l'esecuzione di una istruzione

La tecnologia attuale[7] consente di produrre commercialmente a basso costo processori con ciclo di clock tra 10 nsec e 1 nsec, cioè con frequenze tra 100 MHz (1 MHz = 1 megahertz = 10^6 impulsi al secondo) e 1 GHz (1 GHz = 1 gigahertz = 10^9 impusi al secondo); i personal computer sono dotati di processori con queste caratteristiche.

Per dare un'idea di questi tempi, si pensi che un clock di 100 Mhz genera ogni secondo un numero di impulsi maggiore del numero di minuti della nostra vita e che in 10 nsec la luce percorre circa 3 metri.

> La velocità della luce nel vuoto è circa 30 cm/nanosecondo, cioè un miliardo di Km/h, mentre un segnale elettrico in un filo di rame si muove alla velocità di circa 20 cm/nanosecondo.

Prendendo ad esempio le varie generazioni di processori Intel, è chiaro come ciascuna generazione abbia consentito un miglioramento in termini di prestazioni. Se ad esempio si deve sommare un vettore di numeri, si ha che:

[7] Una legge empirica ma ampiamente accettata indica che la velocità dei processori in commercio raddoppia ogni anno e mezzo. I dati riportati si riferiscono al 2000.

CPU 386	6 cicli di clock per ogni addizione (2 numeri)
CPU 486	2 cicli di clock per ogni addizione "
CPU Pentium	1 ciclo di clock per ogni addizione "
CPU Pentium II, III, IV	1 ciclo di clock per ogni addizione ma con sempre maggiori possibilità di "scegliere" la successiva istruzione[8].

Non sempre, infine, migliorare le prestazioni della CPU è la maniera migliore per migliorare le prestazioni dell'intero sistema; il prelievo di un dato dalla memoria RAM (tempo di accesso) è, supponiamo, di 60 nanosecondi; il tempo di accesso al disco rigido HD (vedi par. successivo) è di 10 millisecondi: il disco è circa 100.000 volte più lento della RAM! Volendo spendere poche centinaia di euro, potrebbe essere più conveniente potenziare la RAM, piuttosto che il processore, migliorando le prestazioni di un fattore 100'000 quando bisogna prelevare un dato che, in mancanza di spazio in memoria RAM, starebbe sul disco, piuttosto che migliorare le prestazioni in termini di velocità di esecuzione del processore.

2.1.3 Sottosistemi di I/O

a) *sottosistemi di memoria di massa*

Una memoria di massa è un supporto per immagazzinare informazioni in modo praticamente permanente. Sono memorie di massa: i dischi, i dischetti, i cd-rom, i nastri.

È fondamentale osservare che le informazioni (istruzioni e dati) che devono essere elaborate dal calcolatore (cioè l'esecuzione di un programma che agisce su certi dati) devono essere trasferite nella memoria. Infatti il processore, che è l'unica componente attiva, esegue un programma che deve risiedere in memoria per agire su dati anch'essi immagazzinati in memoria.

Una memoria di massa, rispetto alla memoria e a parità di prezzo, ha un tempo di accesso superiore e una maggiore capacità.

Dal punto di vista dell'utilizzatore una memoria di massa può essere considerata ancora come una tabella lineare di voci, o di byte. L'organizza-

[8] La tecnologia e il disegno architetturale dei processori sono decisamente migliorati negli ultimi anni. Per quanto riguarda il Pentium IV, ad esempio, rispetto al processore Pentium III, è stata raddoppiata la profondità della pipeline, è stata migliorata la "precisione" sui branch, è stata ampliata la scelta della prossima istruzione da eseguire in caso di attesa per dipendenza da istruzioni in corso, le ALU possono effettuare alcune operazioni in 1/2 periodo di clock, è stato aumentato il il numero di istruzioni SIMD a 128 bit (intere e floating point), è circa triplicata la velocità di trasferimento tra processore e memoria cache (da 16 a 44 GB/sec). Tutto ciò si ripercuote in un aumento di prestazioni generale, maggiormente evidente in alcune specifiche applicazioni.

zione pratica dipende dal tipo di memoria di massa e dalla tecnologia utilizzata nella costruzione.

Il disco (HD, cioè hard disk) è un supporto circolare, sulla cui superficie sono ricavate piste magnetizzabili dette *tracce*, organizzate secondo circonferenze concentriche. Generalmente il numero totale di byte è uguale per tutte le tracce. Un dispositivo elettromeccanico consente di rilevare la stringa di bit memorizzata in un byte, cioè di effettuare una operazione di lettura, oppure di immagazzinare una stringa di bit in una determinata posizione di una traccia, cioè di effettuare una operazione di scrittura. Il numero della traccia e la posizione all'interno della traccia costituiscono l'indirizzo di disco dell'informazione. Il supporto di memorizzazione e il meccanismo di lettura/scrittura sono indicati complessivamente con il termine *disk-drive* e costituisce la cosiddetta *memoria secondaria*. Un HD di un PC ha in genere una capacità di decine di GB. I supercalcolatori hanno sistemi HD fino all'ordine delle decine di TB.

Il dischetto ha una organizzazione analoga al disco, ed è caratterizzato da un maggiore tempo di accesso e da una minore capacità (da 1.44, MB fino a 100MB, o addirittura fino ai più recenti da 2GB).

Il cd-rom ha una architettura analoga al disco, ma la sua organizzazione è basata sull'uso della tecnologia ottica (laser), invece di quella magnetica. Il supporto è circolare e costituito in materiale plastico e in alluminio. Vi sono cd in cui la memorizzazione di una informazione è fisicamente irreversibile (è un insieme di buchi sul supporto), e quindi utilizzabili solo per operazioni di lettura; di qui il termine rom, cioè read only memory, memoria a sola lettura. Negli ultimi tempi sono stati realizzati cd in cui è possibile effettuare più volte la memorizzazione dei dati. L'azione di scrittura di un insieme di informazioni su un cd-rom è detta *masterizzazione*.

Il nastro, nelle sue varie evoluzioni di cassetta e *streamer*, è il più antico dispositivo di memoria di massa. Il supporto è un nastro magnetico su cui i byte vengono memorizzati in sequenza. Sono effettuabili sia l'azione di lettura sia l'azione di scrittura. La natura fisica del supporto implica che per accedere all'informazione memorizzata in una certa posizione è necessario svolgere il nastro fino a quella posizione; il tempo per accedere a una informazione dipende quindi dalla posizione dove è memorizzata l'informazione. Ciò viene sintetizzato dicendo che il nastro consente l'accesso *sequenziale* alle informazioni memorizzate, in contrasto con l'accesso *uniforme* (o *random*) delle altre memorie di massa e della memoria.

b) *sottosistemi di interazione uomo/calcolatore*

Attualmente il dispositivo di I/O per l'interazione uomo/macchina più noto, diffuso e utilizzato è il *terminale*, costituito da uno *schermo* (o *video*, o *monitor*), una *tastiera* e generalmente fornito di un *mouse*.

La tastiera può essere vista come un tipico dispositivo di input, nel senso

che premendo i tasti si forniscono informazioni al calcolatore. Dal punto di vista fisico, si può ricordare che nelle tastiere economiche ogni tasto è un semplice interruttore, che genera un contatto elettrico se premuto. Tastiere più sofisticate hanno un magnete sotto ogni tasto che, se il tasto è premuto, si posiziona all'interno di una bobina, producendo una corrente che viene rilevata.

Il video è un tipico dispositivo di output, nel senso che attraverso di esso vengono visualizzati i risultati dell'esecuzione dei programmi. La componente principale è, in genere, un tubo a raggi catodici (CRT, Cathod Ray Tube). Un CRT contiene un dispositivo (*cannone*) che emette un raggio di elettroni verso uno schermo fosforescente. I video a colori hanno generalmente tre cannoni di elettroni, per il rosso, il verde e il blu, rispettivamente (RGB, da *red-green-blue*). Durante la scansione orizzontale, il raggio percorre lo schermo (per esempio da sinistra verso destra) in circa 50 microsecondi, tracciando una linea leggermente obliqua; poi esegue un ritorno orizzontale per posizionarsi sul margine sinistro e cominciare la successiva scansione. Un dispositivo che produce così una immagine linea per linea si chiama dispositivo a *raster scan*. In genere, in un secondo un intero schermo è ridisegnato dalle 30 alle 60 volte. Lo schermo è considerato come una tabella di *punti indirizzabili*, i *pixel* (contrazione di *picture elements*). Ogni pixel può essere *acceso* o *spento* e rappresenta una informazione grafica elementare. Il numero di pixel complessivo è uno dei principali parametri di qualità di uno schermo. Per esempio, vi sono schermi che contengono 640×480 o 800×600 pixel, mentre schermi più avanzati hanno 1024×768 (standard SVGA) fino a 4096×4096 pixel. Le informazioni da visualizzare sono memorizzate in una memoria specifica associata allo schermo (si parla di video RAM) che viene vista come una tabella di bit. Questo tipo di schermi consente l'uso efficiente delle *finestre*, che in diverse porzioni dello schermo consentono di visualizzare contemporaneamente i risultati di diversi programmi.

Il *mouse* è un dispositivo di input collegato a un terminale video. L'uso del mouse complementa l'uso della tastiera. Il mouse non è altro che una scatoletta di plastica che è possibile spostare sul piano su cui si trova poggiato, mediante un semplice movimento della mano. Questo movimento provoca un corrispondente movimento sullo schermo di un puntatore, normalmente una freccia, che è possibile quindi posizionare su una particolare zona dello schermo stesso, quella in cui per esempio compare un *menu*. La pressione di un pulsante posto sulla parte superiore del mouse provoca poi l'attivazione del programma o del comando richiesti. Esistono mouse con uno, due o tre pulsanti. La rilevazione dello spostamento avviene mediante metodi meccanici (palline o rotelle che fuoriescono dalla parte inferiore del mouse e poggiano quindi sul piano di lavoro) o ottici (diodi LED e fotorilevatori nella parte inferiore, che misurano gli spostamenti).

2.2 Rappresentazione delle informazioni

L'esecuzione di un programma da parte della macchina fisica (il livello hardware di un calcolatore) richiede la memorizzazione delle informazioni che costituiscono un programma: istruzioni e dati. La *rappresentazione delle informazioni elementari* nella macchina fisica si riferisce ai criteri che sono utilizzati per rappresentare in memoria (rappresentazione *interna*) i dati e le istruzioni e alla descrizione delle principali operazioni sui dati.

Dati e istruzioni sono rappresentati in memoria mediante stringhe di bit che costituiscono il contenuto di una locazione, o talvolta di due o più locazioni. I criteri di rappresentazione sono quindi convenzioni per descrivere una informazione elementare mediante una stringa di bit di lunghezza prefissata (per es. i 32 bit di una locazione di un PC).

È importante sottolineare che la rappresentazione interna è utilizzata non solo per la memorizzazione di dati e istruzioni in memoria ma anche per la loro memorizzazione nei vari dispositivi di memoria di massa, per la loro trasmissione nei canali di comunicazione interni e per la loro trasmissione in rete.

Si noti che non c'è alcun *significato inerente* in una stringa di bit. Il *significato* di una stringa di bit dipende solo da come essa è utilizzata.

2.2.1 Rappresentazione dei dati

È usuale classificare i dati in insiemi, detti *tipi di dati*, caratterizzati da un comune criterio di rappresentazione e dalle operazioni che sono eseguibili su di essi. I principali tipi di dati elementari sono il *tipo testo* (o *carattere* o *alfanumerico*), il *tipo logico*, il *tipo intero* e il *tipo reale*.

L'obiettivo dei tipi di dati elementari è quello *riprodurre il più fedelmente possibile* i dati che si presentano frequentemente nei problemi concreti. In particolare il tipo carattere serve per riprodurre testi, il tipo intero i numeri interi, il tipo reale i numeri reali, mentre il tipo logico consente di esprimere il risultato di un confronto o più in generale il valore di verità di una affermazione.

Il tipo testo

L'insieme dei dati di tipo testo è formato dalle 26 lettere minuscole e dalle 26 lettere maiuscole dell'alfabeto inglese, dalle 10 cifre decimali, dallo spazio di separazione (detto *blank*), dai segni di punteggiatura e da altri simboli come

#,@,&,",',(,ç,è,),£,à,é,-,=,+,{,[,],}~,*,^,/,%,$,ù,§,°,<,>, etc...

Una sequenza (o *stringa*) di caratteri costituisce un testo. Il rigo (e quindi anche tutto questo libro) che state leggendo è una stringa di dati di tipo testo.

La rappresentazione di un (dato di tipo) carattere è basata su un *codice* ed è perciò detta anche *codifica*. Un codice è una applicazione che associa a ciascun carattere una sequenza di bit di lunghezza prefissata. Il codice normalmente usato nei calcolatori attuali è il codice ASCII (acronimo per *American Standard Code for Information Interchange*) che rappresenta un carattere con un byte[9].

La rappresentazione in memoria di un testo consiste semplicemente nella rappresentazione di ogni singolo carattere che lo costituisce mediante la corrispondente stringa di bit prevista dal codice e nella memorizzazione di tali stringhe in byte consecutivi della memoria[10].

blank	0010000	A	01000001	M	01001101	a	01100001	m	01101101
0	00110000	B	01000010	N	01001110	b	01100010	n	01101110
1	00110001	C	01000011	O	01001111	c	01100011	o	01101111
2	00110010	D	01000100	P	01010000	d	01100100	p	01110000
3	00110011	E	01000101	Q	01010001	e	01100101	q	01110001
4	00110100	F	01000110	R	01010010	f	01100110	r	01110010
5	00110101	G	01000111	S	01010011	g	01100111	s	01110011
6	00110110	H	01001000	T	01010100	h	01101000	t	01110100
7	00110111	I	01001001	U	01010101	i	01101001	u	01110101
8	00111000	J	01001010	V	01010110	j	01101010	v	01110110
9	00111001	L	01001100	Z	01011010	l	01101100	z	01111010

97M0882073172228

Fig. 2.8: alcuni caratteri del codice ASCII

Ad esempio, per la rappresentazione della parola `Italia` in una memoria organizzata con voci di 32 bit, utilizzando il codice ASCII, sono necessari 6 byte, la cui memorizzazione richiede due voci (Fg. 2.9).

[9] In un byte è possibile rappresentare al più 2^8=256 caratteri diversi, che è il numero delle combinazioni con ripetizione di 2 oggetti (le due cifre binarie 0 e 1) su 8 posti (un byte). Il codice ASCII usa solo 7 bit di un byte ed è quindi in grado di rappresentare 128 caratteri.

[10] Si noti che la capacità di una memoria espressa in byte rappresenta il numero di caratteri memorizzabili in tale memoria.

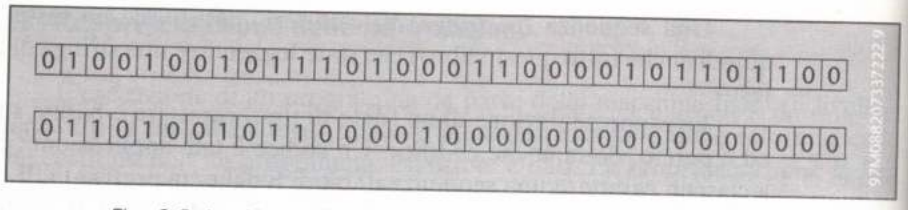


Fig. 2.9: la stringa di caratteri `Italia` in due voci da 32 bit

Le principali operazioni elementari effettuabili su dati di tipo testo sono

- il *confronto* tra due caratteri, in particolare per determinare se due caratteri sono uguali o se uno precede l'altro nell'ordine alfabetico;
- l'*estrazione* di un carattere (o di una *sotto*stringa) da una stringa di caratteri;
- il *concatenamento* di due caratteri (o di due stringhe) in un'unica stringa.
- il *trasferimento di I/O*: la *lettura esterna*, che determina la memorizzazione di una stringa di caratteri immessa da un dispositivo di input (per es. digitata sulla tastiera del calcolatore); la *scrittura esterna*, cioè l'invio di una stringa di caratteri dalla memoria a un dispositivo di output (per es. la visualizzazione di un testo).

Si noti che l'operazione di estrazione richiede la lettura (in memoria) di uno (o più) byte; il concatenamento richieda la scrittura di due (o più) byte in una (o più) voci.

Per esempio, il concatenamento della stringa `mare` e della stringa `moto` genera la nuova stringa `maremoto`; che viene memorizzata come in Fig. 2.10.

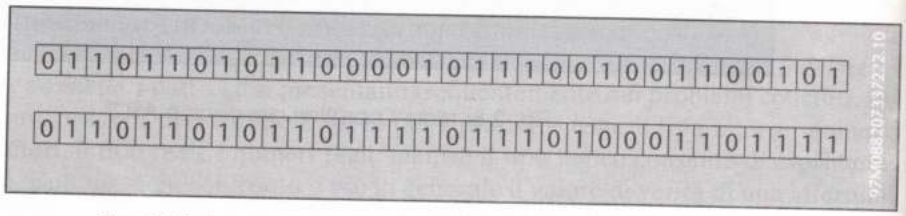


Fig. 2.10: la stringa di caratteri `maremoto` in due voci da 32 bit

L'esempio sottolinea che lo spazio di separazione è un carattere a tutti gli effetti e deve essere concatenato quando necessario; per ottenere la stringa `bella Italia` è necessario concatenare le stringhe `bella`, il blank, `Italia`.

Una osservazione importante è che non sono definite operazioni aritmetiche su stringhe di cifre. Per esempio la stringa `1423` rappresenta un testo e non un numero coinvolgibile in operazioni aritmetiche.

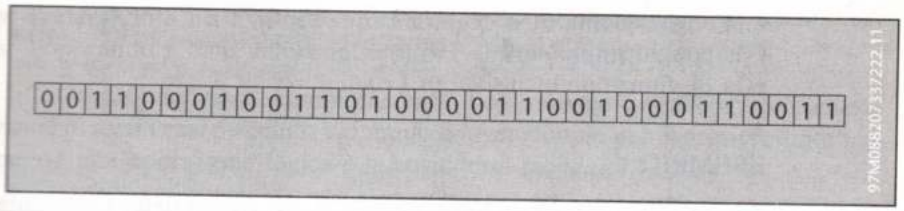


Fig. 2.11: la stringa di caratteri `1423` in una voce da 32 bit

Un classico esempio è costituito dai numeri di telefono, dai numeri civici di abitazioni, dai codici fiscali, dai codici postali, ecc.. Infatti, su nessuno di questi è usualmente richiesto di effettuare operazioni aritmetiche, ma al più operazioni di confronto, di estrazione di sottostringa e di concatenamento. Un numero su cui si vogliono effettuare operazioni aritmetiche deve essere rappresentato come dato di tipo intero o di tipo reale.

In definitiva, il tipo carattere consente di riprodurre esattamente in memoria un qualunque testo, senza alcuna perdita di informazione, cioè senza alcuna approssimazione.

Il tipo logico

Il tipo logico (detto anche *booleano*[11]) è l'insieme costituito dai due valori *vero* e *falso*.

Poiché il tipo logico ha solo due elementi, è evidente che ogni elemento può essere rappresentato con un unico bit, per esempio adottando la convenzione per cui 1 rappresenta il valore *vero* e 0 rappresenta il valore *falso*[12].

L'obiettivo del tipo logico è di riprodurre alcune operazioni della logica elementare e del cosiddetto calcolo dei predicati.

In logica, una *sentenza* è una affermazione che può essere vera (V) o falsa (F). Per esempio:

sentenza A: Napoli è il capoluogo della Campania
sentenza B : la Terra è più grande di Urano
sentenza C : 5 > 3

In particolare, A è vera, B è falsa, C è vera.

Un *predicato* è una struttura logica che si ottiene connettendo sentenze mediante i connettivi logici elementari che sono

[11] Dal nome del matematico inglese George Boole (1815-1864) considerato il fondatore della logica formale: egli stabilì gli assiomi della logica (come fece Euclide per la geometria) e mostrò che l'intero campo della logica poteva essere soggetto a una trattazione algebrica.

[12] In pratica la memorizzazione di un dato logico può richiedere un intero byte, se la macchina fisica non consente operazioni di lettura/scrittura a livello di singolo bit.

- la negazione, not ($\neg$)
- la congiunzione, and ($\wedge$)
- la disgiunzione inclusiva, or ($\vee$).

ESEMPIO:

il predicato $\neg$ A (cioè Napoli *non* è il capoluogo della Campania) è falso
il predicato $\neg$ B (cioè la Terra *non* è più grande di Urano) è vero
il predicato $\neg$ C (cioè $5 \leq 3$) è falso
il predicato $B \wedge C$ (cioè la Terra è più grande di Urano *e* $5 > 3$) è falso
il predicato $A \wedge C$ (cioè Napoli è il capoluogo della Campania *e* $5 > 3$) è vero
il predicato $A \vee B$ (cioè Napoli è il capoluogo della Campania *oppure* la Terra è più grande di Urano) è vero
il predicato $A \vee C$ (cioè Napoli è il capoluogo della Campania *oppure* $5 > 3$) è vero.

I predicati possono essere a loro volta connessi per dar luogo a predicati più complicati.

Per esempio il predicato $\neg((A \wedge \neg C) \vee B)$ è vero.

ESEMPIO:

si consideri il problema di determinare se un numero reale x appartiene ad un intervallo $[a,b]$.

Il seguente predicato è vero solo se x appartiene all'intervallo:

$$(x \geq a) \wedge (x \leq b)$$

Un modo alternativo è costituito dal predicato

$$\neg\ ((x < a) \vee (x > b))$$

che esprime l'appartenza di x all'intervallo $[a,b]$ attraverso il fatto che x non deve essere minore di a o maggiore di b; è semplice convincersi che i due predicati sono equivalenti.

I connettivi logici elementari denotano operazioni logiche, cioè operazioni in cui operandi e risultato appartengono all'insieme {vero, falso}.

Esaminiamo in dettaglio le operazioni logiche elementari.

- Negazione

La negazione è una operazione che coinvolge un solo operando. In particolare

se l'operando p ha il valore vero allora $\neg p$ ha il valore falso
se l'operando p ha il valore falso allora $\neg p$ ha il valore vero.

Questo definisce completamente l'operazione attraverso l'enumerazione di tutti i casi possibili. Un modo convenzionale per esprimere sinteticamente una operazione logica è la cosiddetta *tabella di verità*.

p	$\neg p$
vero	falso
falso	vero

97M088207337222.12

Fig. 2.12: tabella di verità della negazione

- Congiunzione

La congiunzione è una operazione che coinvolge due operandi. In particolare se gli operandi p e q hanno entrambi il valore vero allora $p \wedge q$ ha il valore vero altrimenti $p \wedge q$ ha il valore falso.

In altri termini la congiunzione di due sentenze è vera solo se entrambe le sentenze che la costituiscono sono vere.

p	q	$p \wedge q$
vero	vero	vero
vero	falso	falso
falso	vero	falso
falso	falso	falso

97M088207337222.13

Fig. 2.13: tabella di verità della congiunzione

- Disgiunzione

La disgiunzione (inclusiva) è una operazione che coinvolge due operandi. In particolare

se gli operandi p e q hanno entrambi il valore falso allora $p \vee q$ ha il valore falso altrimenti $p \vee q$ ha il valore vero.

In altri termini la disgiunzione di due sentenze è vera se almeno una delle sentenze che la costituiscono è vera.

p	q	$p \wedge q$
vero	vero	vero
vero	falso	vero
falso	vero	vero
falso	falso	falso

Fig. 2.14: tabella di verità della disgiunzione

A partire dalle operazioni elementari è possibile definire nuove operazioni logiche, come quelle riportate in Fig. 2.15 mediante le loro tabelle di verità.

p	q	*nor*	*nand*	*xor*
vero	vero	falso	falso	falso
vero	falso	falso	vero	vero
falso	vero	falso	vero	vero
falso	falso	vero	vero	falso

Fig. 2.15: tabella di verità di nor, nand, xor

ESEMPIO:

utilizzando solo l'operazione nand costruire l'operazione logica con la seguente tabella della verità

p	q	$p \wedge q$
vero	vero	vero
vero	falso	falso
falso	vero	vero
falso	falso	vero

97M0882073372221 56

Come può essere facilmente verificato, l'operazione è p nand (p nand q).

ESEMPIO:

i connettivi and e or sono legati dalla relazione detta di De Morgan[13]

$$p \wedge q = \neg (\neg p \vee \neg q)$$

o in modo del tutto equivalente

$$\neg(p \wedge q) = \neg p \vee \neg q$$

ESEMPIO:

un predicato che è sempre falso è detto *contraddizione*, come il seguente

$$p \wedge \neg p.$$

ESEMPIO:

un predicato che è sempre vero è detto *tautologia*, come il seguente

$$p \vee \neg p.$$

Le operazioni logiche elementari di negazione, disgiunzione e congiunzione sono le operazioni definite sul tipo logico.

Il tipo logico consente di riprodurre esattamente in memoria i dati logici e di effettuare su di essi le operazioni logiche elementari senza alcuna approssimazione. In definitiva si ha che è possibile rappresentare in modo esatto predicati complicati quanto si voglia.

[13] Augustus De Morgan (1806-1871) matematico inglese, considerato con Boole e Hamilton tra i fondatori della logica matematica moderna.

Si ricorda che l'elettronica digitale opera su una informazione binaria, perciò l'ingresso e l'uscita, per esempio di una porta AND, sono interpretabili come gli operandi e il risultato dell'operazione logica *and*. Le principali porte logiche sono la porta NOT, la porta AND, la porta OR, la porta NAND, e la porta NOR.

Dio ha creato gli interi, tutto il resto è opera dell'uomo.

Leopold Kronecker (1823-1891)

Il tipo intero

L'insieme infinito degli interi, usualmente indicato con Z, comprende i numeri interi (positivi e negativi).

L'obiettivo del tipo intero è quello *riprodurre il più fedelmente possibile* l'insieme Z e le operazioni aritmetiche definite su di esso.

La difficoltà risiede nel fatto che Z è un insieme infinito, mentre il calcolatore, e la sua memoria, è costituito da un numero finito di componenti.

L'insieme dei dati di tipo intero è un sottoinsieme di Z costituito da un numero finito di numeri interi. In particolare il tipo intero è l'insieme:

$$\{-I_{\max}, -I_{\max}+1, \ldots, -2, -1, 0, 1, 2, \ldots, I_{\max}-1, I_{\max}\}$$

$I_{\max}$ è detto il *massimo intero rappresentabile.*

Per esempio se $I_{\max} = 9$ l'insieme è

$$\{-9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

La rappresentazione di un dato del tipo intero è basato sul criterio *posizionale in base 2, utilizzando un numero finito di cifre.*

La notazione che sin da bambini usiamo per scrivere un numero intero è quella posizionale in base 10. Ciò significa che si usano dieci cifre (0,1,2,3,4,5,6,7,8,9) e il loro valore dipende dalla posizione che la cifra ha nella sequenza di cifre che denota il numero. Per esempio la notazione 4515 indica il numero

4 migliaia +5 centinaia +1 decina +5 unità

ovvero:

$$4515 = 4 \cdot 10^3 + 5 \cdot 10^2 + 1 \cdot 10^1 + 5 \cdot 10^0 = 4 \cdot 10^3 + 5 \cdot 10^2 + 1 \cdot 10 + 5 =$$
$$= 4000 + 500 + 10 + 5$$

Si noti il diverso valore indicato dalla cifra 5 che appare due volte nella sequenza.

Il criterio posizionale non dipende dalla scelta della base, che può essere un qualunque intero maggiore di uno. È una interessante eredità storica il fatto che attualmente in tutte le culture avanzate la base sia dieci[14].

Formalmente il criterio di rappresentazione posizionale in base b si può riassumere nel seguente modo:

sia $b(>1)$ un numero intero detto *base*, allora $\forall k \in Z$, che supponiamo per il momento non negativo, si ha

$$k = (c_n c_{n-1} \ldots\ldots c_0)_b = c_n \cdot b^n + c_{n-1} \cdot b^{n-1} + \ldots. + c_0 \cdot b^0$$

con $0 \le c_i < b$. Si noti che c_i è una cifra nella base b e $(c_n c_{n-1} \ldots\ldots c_0)_b$ denota una stringa di n+1 cifre nella base b.

ESEMPIO:

considerato il numero decimale k = 15 sia si ha

se $b = 10$ allora

$k = (15)_{10} = 1 \cdot 10^1 + 5 \cdot 10^0 = 10 + 5 = 15$;

se $b = 8$ allora

$k = (17)_8 = 1 \cdot 8^1 + 7 \cdot 8^0 = 8 + 7 = 15$;

se $b = 2$ allora

$k = (1111)_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 2 + 1 = 15$

se $b = 12$ allora

$k = (13)_{12} = 1 \cdot 12^1 + 3 \cdot 12^0 = 12 + 3 = 15$.

È importante sottolineare che le cifre in base 10 sono {0,1,2,3,4,5,6,7,8,9}, le cifre in base 8 sono {0,1,2,3,4,5,6,7} e le cifre in base 2 sono {0,1}.

[14] I Babilonesi utilizzavano un criterio posizionale con base 60, di cui è rimasta traccia tangibile nella civiltà occidentale, come testimoniato dalle attuali unità sessagesimali di ore, minuti, secondi e di gradi, primi, secondi. Al contrario, i Romani utilizzavano un criterio non posizionale, bensì additivo.

Con basi maggiori di 10 sorge la necessità di denotare con un simbolo l'undicesima cifra, la dodicesima cifra, etc. Per esempio, se $b = 12$, denotando rispettivamente con A e con B l'undicesima e la dodicesima cifra, si ha

$$k = (1A\ 0B)_{12} = 1 \cdot 12^3 + 10 \cdot 12^2 + 0 \cdot 12^1 + 11 \cdot 12^0 =$$
$$= 1 \cdot 1728 + 10 \cdot 144 + 0 \cdot 12 + 11 = 3179$$

```
1,
10, 11,
100, 101, 110, 111,
1000, 1001, 1010, 1011, 1100,1101,1110, 1111,
10000
```

Fig. 2.16: i numeri 1,2,3,...,16 espressi in binario

Abbiamo detto che un dato di tipo intero è rappresentato secondo il criterio posizionale in base 2 *finito*[15]. L'aggettivo finito indica semplicemente che il numero massimo di cifre (in base 2) è prestabilito e non è infinito; ciò è imposto dalla lunghezza finita della locazione di memoria.

Un qualunque numero intero che sia minore di $-I_{max}$ oppure maggiore di I_{max} non fa parte del tipo intero.

La quantità I_{max} dipende dalla organizzazione delle voci di memoria del calcolatore.

Si consideri una voce costituita da s bit; la soluzione più ovvia consiste nell'utilizzare un bit per denotare il segno del numero e i restanti s-1 per rappresentare il suo valore assoluto, secondo lo schema in Fig. 2.17.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	0	1	1	1	1

Fig. 2.17: rappresentazione di 1423

Poiché il massimo numero rappresentabile con il criterio posizionale in base 2, avendo a disposizione s–1 cifre binarie, è $2^{s-1}-1$, immediatamente si ha che $I_{max} = 2^{s-1} - 1$.

[15] La scelta della rappresentazione binaria dei numeri in un calcolatore fu suggerita da von Neumann.

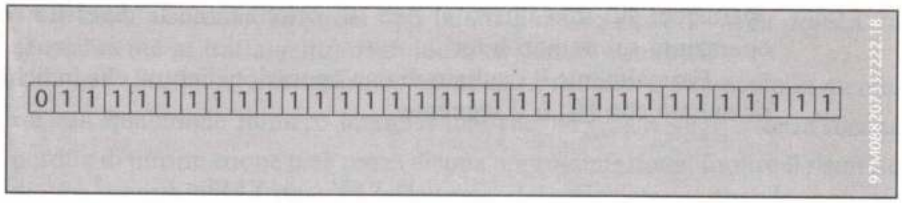


Fig. 2.18: massimo intero rappresentabile

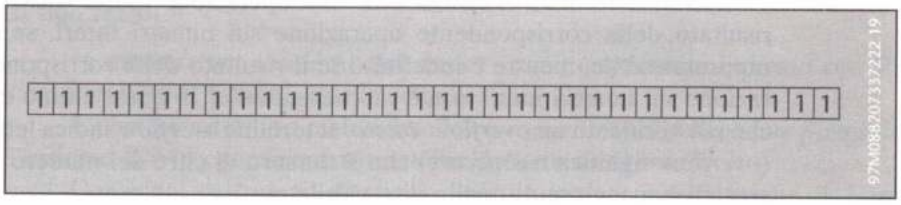


Fig. 2.19: minimo intero rappresentabile

con questo criterio due rappresentazioni dello 0: +0 e –0.

Un numero intero n si dice rappresentabile se $-I_{\max} \le n \le I_{\max}$.

Si noti che, se la memoria è organizzata a locazioni di 32 bit, si ha

$$I_{\max} = 2^{31} - 1 = 2147483647 \ .$$

Analogamente, si ha che nel caso di locazioni a 64 bit

$$I_{\max} = 2^{63} - 1 = 9223372036854775807 \quad (\approx 0.9 \times 10^{19}) .$$

Un criterio alternativo (utile nel caso di base 2) alla scelta di rappresentare un numero nella forma segno e valore assoluto, consiste nella cosiddetta rappresentazione traslata che esprime un qualunque numero n (rappresentabile) come $n + I_{\max}$ su s bit. In questo modo il più piccolo intero rappresentabile viene espresso dalla stringa nulla e il più grande dalla stringa composta tutta dalla cifra 1. Questa tecnica evita di rappresentare esplicitamente il segno e consente di utilizzare tutti i bit della voce per rappresentare cifre. La costante che deve essere sottratta alla stringa memorizzata è detta costante di traslazione (bias).

Sul tipo intero sono definite le operazioni di *addizione intera*, *sottrazione intera*, *moltiplicazione intera*, *divisione intera*.

Tali operazioni hanno, come le corrispondenti operazioni sui numeri interi, due operandi e un risultato.

Solo in questo paragrafo indicheremo con i simboli $\boxplus$, $\boxminus$, $\boxast$, $\boxslash$ le

operazioni sul tipo intero al fine di sottolinearne la diversità rispetto alle operazioni sui numeri interi.

Formalmente il risultato di una operazione intera, che indichiamo genericamente con $\boxed{\#}$, è il seguente

$$X \boxed{\#} Y = \begin{cases} X\#Y & \text{se} |X\#Y| \le I_{\max} \\ \textit{indefinito} & \text{se} |X\#Y| > I_{\max} \end{cases}$$

In altri termini, il risultato di una operazione intera è esattamente il risultato della corrispondente operazione sui numeri interi, se questo è rappresentabile, mentre è indefinito se il risultato della corrispondente operazione sui numeri interi non è rappresentabile. In quest'ultimo caso si dice che si è verificato un *overflow intero*. Il termine *overflow* indica letteralmente (*overflow* significa *traboccare*) che il numero di cifre del numero da rappresentare è maggiore di quello disponibile.

ESEMPIO:

Se $I_{\max} = 31$, allora

$12 \boxed{+} 15 = 27$; $5 \boxed{*} 6 = 30$; $12 \boxed{/} 5 = 2$; $-2 \boxed{-} 30 =$ indefinito;
$7 \boxed{*} 7 =$ indefinito.

Si noti che il risultato della divisione intera è la parte intera del quoziente.

È importante sottolineare che alcune proprietà dei numeri interi non sono valide per il tipo intero. Per esempio, la proprietà associativa non vale per il tipo intero, poiché l'ordine con cui si effettuano le operazioni può causare un overflow.

Si consideri infatti la valutazione dell'espressione $15 \boxed{+} 20 \boxed{-} 8$, il cui risultato è 27, che è rappresentabile (consideriamo ancora $I_{\max} = 31$). Se l'ordine di valutazione è $(15 \boxed{+} 20) \boxed{-} 8$ il risultato finale è indefinito, poiché l'addizione $15 \boxed{+} 20$ provoca overflow; al contrario se l'ordine di valutazione è $15 \boxed{+} (20 \boxed{-} 8)$ il risultato è esatto.

Nelle applicazioni pratiche l'overflow intero può creare problemi quando si valutano funzioni intere rapidamente crescenti come la funzione fattoriale.

Infatti si ha che

$$13! = 13 \cdot 12 \cdot 11 \cdot 10 \cdot \ldots \cdot 2 \cdot 1 = 6227020800$$

non è rappresentabile come dato di tipo intero in un comune PC con voce a 32 bit ($6227020800 > 2147483647$).

Come vedremo, l'uso del tipo intero è spesso ristretto nella pratica essenzialmente al trattamento degli indici di vettori e matrici.

In definitiva, il tipo intero consente di riprodurre esattamente in memoria un qualunque numero intero n tale che $-I_{\max} \leq n \leq I_{\max}$, senza alcuna perdita di informazione cioè senza alcuna approssimazione. Inoltre il risultato di una operazione aritmetica effettuata dal calcolatore, quando non sia overflow, è esatto, cioè è uguale al risultato della corrispondente operazione sui numeri interi.

Il tipo reale

In matematica, l'insieme dei numeri reali, usualmente indicato con R, comprende tutti i numeri della retta reale.

L'obiettivo del tipo reale è quello *riprodurre il più fedelmente possibile* l'insieme R e le operazioni aritmetiche definite su di esso.

Come nel caso degli interi, la difficoltà risiede nel fatto che R è un insieme infinito e il calcolatore ha a disposizione solo un numero finito di bit per la rappresentazione.

In verità, la *riproduzione* dell'insieme dei reali è *più difficile* della *riproduzione* degli interi. Infatti, Z è costituito da un'infinità numerabile, mentre R da un'infinità continua di numeri[16].

L'insieme dei dati di tipo reale è un sottoinsieme di R costituito da un numero finito di numeri reali.

La rappresentazione di un dato del tipo reale è basato sul criterio *floating-point* (letteralmente: *virgola mobile*) *normalizzato in base 2 a precisione finita.*

La rappresentazione floating-point, detta anche notazione scientifica[17], consiste nel denotare un numero nella forma *mantissa-esponente*; per esempio il numero –3.14159 si può esprimere come in Fig. 2.20

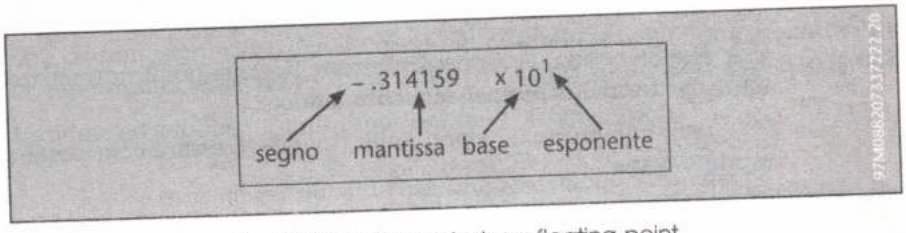


Fig. 2.20: rappresentazione floating-point

[16] Se x e y sono due qualunque numeri reali con x<y, vi è una infinità non numerabile di numeri reali compresi tra x e y.

[17] Poiché è la notazione comunemente usata nei libri scientifici: per esempio la velocità della luce viene indicata come $3 \cdot 10^2$ m/sec, il raggio atomico dell'idrogeno come $25 \cdot 10^{-12}$ m.

L'esponente è un numero intero. Sia la mantissa sia l'esponente sono rappresentati secondo il criterio posizionale[18].

In particolare, se la mantissa è minore di 1 e la prima cifra dopo il punto frazionario è divera da 0, come ad esempio in –.314159, si parla di rappresentazione floating-point *normalizzata*.

Esempi di rappresentazione floating-point non normalizzata dello stesso numero sono i seguenti:

$$-.0314159\times10^{2}\,,\quad -.00314159\times10^{3}\,,\quad -.000314159\times10^{4}$$

La normalizzazione ha il vantaggio di

- rendere unica la rappresentazione di un numero (esiste un'unica mantissa normalizzata)
- rappresentare solo le *cifre significative* di un numero, cioè le cifre della mantissa con l'esclusione degli zeri che vengono subito dopo il punto frazionario.

numero	rappresentazione fp normalizzata
0.00000000904	0.904×10^{-9}
76554.002	0.76554002×10^{-5}
-1	0.1×10^{-1}
30000000000	0.3×10^{-11}

9788820337221

Fig. 2.21: esempi di rappresentazione floating-point normalizzata

Riassumendo, la rappresentazione floating-point normalizzata può sintetizzarsi formalmente nel seguente modo:

ogni numero reale x non nullo (lo 0 deve essere considerato a parte) si esprime come

$$x = m\times b^{e}$$

con $m\in R:\frac{1}{b}\le|m|<1\,,\quad b\in N, b>1\,,\ e\in Z\,.$

[18] L'estensione del criterio posizionale ai numeri frazionari è immediato: le cifre della parte frazionaria sono i coefficienti di potenze negative della base. Per esempio la notazione 0.376 esprime il numero $3*10^{-1} + 7*10^{-2} + 6*10^{-3}$.

Se si adotta tale criterio di rappresentazione, allora la memorizzazione di un numero in una voce di memoria consiste nella memorizzazione delle seguenti informazioni: il segno del numero (cioè il segno della mantissa), la parte frazionata della mantissa, il segno dell'esponente, il valore assoluto dell'esponente.

La lunghezza finita della voce di memoria impone che sia il numero di cifre a disposizione per rappresentare la mantissa (detto *precisione*) sia il numero di cifre per l'esponente devono essere finiti; inoltre in un calcolatore essi sono prefissati e costituiscono, insieme alla base, i parametri fondamentali dell'organizzazione della parte dell'hardware detta *sistema aritmetico reale o sistema aritmetico floating point.*

Vediamo quali conseguenze comporta la precisione finita e la finitezza dell'esponente sulla rappresentabilità dei numeri reali.

La finitezza dell'esponente implica l'esistenza di un esponente massimo e di un esponente minimo e quindi che esistono un massimo e un minimo (in valore assoluto) numero reale rappresentabile come dato di tipo reale. Questo fatto si sintetizza dicendo che il *range è limitato.*

Il più piccolo numero reale positivo rappresentabile in un sistema aritmetico reale è detto *soglia di underflow* ($r_{\min}$), il più grande è detto *soglia di overflow* ($r_{\max}$).

ESEMPIO:

se la base è 10 e il numero di cifre per l'esponente è 2, si ha che il massimo numero reale rappresentabile è $0.999\ldots\times10^{99}$ e che il minimo numero reale rappresentabile è $0.100\ldots\times10^{-99}$.

Analogo discorso vale per i numeri reali negativi.

Si noti, per inciso, che il criterio di normalizzazione non consentirebbe la rappresentazione dello zero. Lo zero viene comunque espresso come mantissa ed esponente nullo (per esempio $0.000\ldots0\times10^{00}$).

Si dice *insieme dei numeri reali rappresentabili* l'insieme

$$[-r_{\max},-r_{\min}]\cup\{0\}\cup[r_{\min},r_{\max}] \quad .$$

La finitezza della mantissa (cioè la *precisione finita*) implica che solo una parte (un sottoinsieme finito) dei numeri reali rappresentabili possono essere rappresentati esattamente; si tratta dei numeri che hanno un numero di cifre significative non maggiore della precisione.

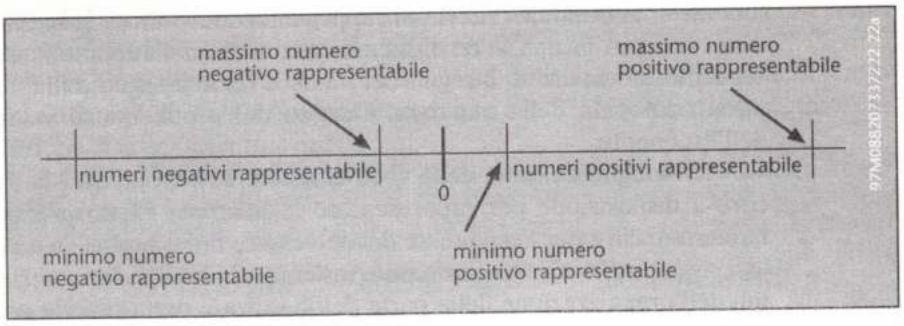


Fig. 2.22a: i numeri reali rappresentabili

Tali numeri sono detti *numeri macchina. Il tipo reale è l'insieme dei numeri macchina.*

Ciò significa che esiste una spaziatura (*gap*) tra due numeri macchina consecutivi.

Per esempio, il numero 3.14159 è un numero macchina solo se (nel caso di base 10) si hanno a disposizione almeno 6 cifre per la mantissa, cioè se la precisione è almeno 6.

I numeri reali rappresentabili che hanno un numero di cifre significative maggiore della precisione (cioè che non sono numeri macchina) vengono rappresentati in modo approssimato, mediante il numero macchina più vicino (arrotondamento).

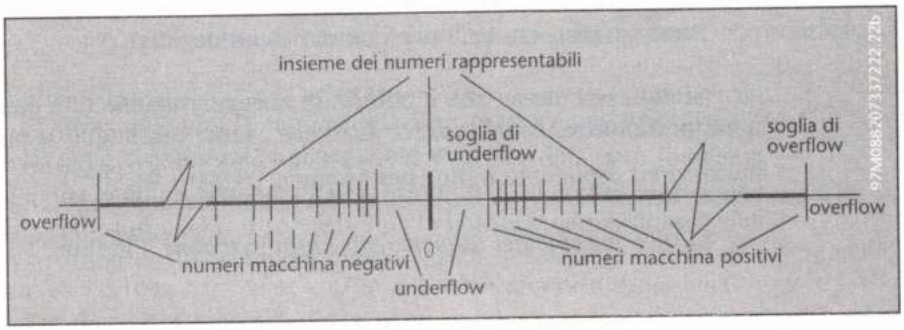


Fig. 2.22b: i numeri macchina

I numeri macchina sono indicati in Fig. 2.22b da barrette verticali sull'asse reale; l'insieme dei numeri reali rappresentabili è costituito dai numeri reali compresi tra la soglia di underflow e la soglia di overflow, dai numeri reali compresi tra meno

la soglia di overflow e meno la soglia di underflow e dallo zero. La distanza tra due barrette consecutive è costante quando la potenza della base è la stessa mentre si raddoppia, se la base è 2, quando l'esponente aumenta di uno: ciò si suole sintetizzare dicendo che la spaziatura dei numeri macchina non è uniforme, ma la loro spaziatura relativa è uniforme. La figura è simmetrica rispetto all'origine.

Il tipo reale dipende da quattro parametri: la base (b), la precisione (che indichiamo con t), il massimo esponente ($e_{\max}$), il minimo esponente ($e_{\min}$). Inoltre, fissato un qualunque numero reale x, si può verificare solo una delle seguenti situazioni:

- x appartiene al tipo reale (x è un numero macchina)
- x non appartiene al tipo reale, ma appartiene all'insieme dei numeri rappresentabili
- x non appartiene all'insieme dei numeri rappresentabili (è troppo piccolo oppure troppo grande in valore assoluto)

cioè

- x è rappresentabile esattamente
- x è rappresentabile in modo approssimato
- x non è rappresentabile.

La rappresentazione approssimata di x comporta l'eliminazione delle cifre significative non memorizzabili (dalla $t+1$-sima in poi) e *l'arrotondamento* della t-sima cifra.[19]

Indicheremo con $fl(x)$ il numero macchina che rappresenta x.

ESEMPIO:

Sia $b = 10$, $t = 2$, $e_{\max} = 2$, $e_{\min} = -2$, allora il numero $x = 1.26 = 0.126 \times 10^1$ appartiene all'insieme dei numeri rappresentabili ma non è un numero macchina, poiché ha tre cifre significative a dispetto della precisione due. Il numero macchina che lo precede è $a = 1.2 = 0.12 \times 10^1$ e quello che lo segue è $b = 1.3 = 0.13 \times 10^1$. L'arrotondamento consente di rappresentare x mediante il numero macchina più vicino, in questo caso $b = 1.3 = 0.13 \times 10^1$; si noti che si tratta di una rappresentazione approssimata che introduce un errore:

$$|x - fl(x)| = |x - b| = |1.3 - 1.26| = 0.04 .$$

[19] Un altro modo per approssimare un numero con più di t cifre significative è utilizzare solo le prime t, eliminando semplicemente quelle dalla t+1-sima in poi. Questo è il *troncamento*.

ESEMPIO:

Sia $b = 10$, $t = 4$, $e_{max} = 9$, $e_{min} = -9$, allora il numero $x = 3.14159 = .314159 \times 10^1$ appartiene all'insieme dei numeri rappresentabili ma non è un numero macchina, poiché ha sei cifre significative a dispetto della precisione quattro. Il numero macchina che lo precede è $a = 3.141 = 0.3141 \times 10^1$ e quello che lo segue è $b = 3.142 = 0.3142 \times 10^1$. L'arrotondamento consente di rappresentare x mediante il numero macchina più vicino, in questo caso $b = 0.3142 \times 10^1$; si noti che si tratta di una rappresentazione approssimata che introduce un errore

$$|x - fl(x)| = |x - b| = |3.14159 - 3.142| = 0.00041 .$$

ESEMPIO:

Sia $b = 2$, $t = 6$, $e_{max} = 7$, $e_{min} = -7$, allora il numero $x = 0.10110001 \times 2^4 = 11.0625$ appartiene all'insieme dei numeri rappresentabili ma non è un numero macchina, poiché ha otto cifre significative, a dispetto della precisione sei. Il numero macchina che lo precede è $a = 0.101100 \times 2^4 = 11.00$ e quello che lo segue è $b = 0.101101 \times 2^4 = 11.25$. L'arrotondamento consente di rappresentare x mediante il numero macchina più vicino, in questo caso $a = 0.101100 \times 2^4$; l'errore introdotto è

$$|x - fl(x)| = |x - a| = |11 - 11.0625| = 0.0625$$

ESEMPIO:

Sia $b = 10$, $t = 4$, $e_{max} = 99$, $e_{min} = -99$, allora il numero $x = 16765000000 = 0.16765 \times 10^{11}$ appartiene all'insieme dei numeri rappresentabili ma non è un numero macchina, poiché ha cinque cifre significative, a dispetto della precisione quattro. Il numero macchina che lo precede è $a = 0.1676 \times 10^{11}$ e quello che lo segue è $b = 0.1677 \times 10^{11}$. Si noti che in questo caso x è equidistante da a e b; l'arrotondamento risolve il conflitto rappresentando x mediante il numero macchina più vicino *pari*,[20] in questo caso $a = 0.1676 \times 10^{11}$. L'errore introdotto è

$$|0.16765 \times 10^{11} - 0.1676 \times 10^{11}| = 0.5 \times 10^7 .$$

L'errore che si commette rappresentando un numero reale mediante un numero macchina è detto errore di roundoff.

La presenza dell'errore di roundoff è la conseguenza più importante della rappresentazione floating-point a precisione finita e quindi dell'uso del tipo reale.

[20] Questa è la strategia utilizzata nello standard IEEE.

Errore assoluto ed errore relativo.
Siano x e $\tilde{x}$ due numeri e consideriamo $\tilde{x}$ una approssimazione di x.
L'errore assoluto E_a e l'errore relativo E_r dell'approssimazione di x con $\tilde{x}$ sono definiti da

$$E_a = |x - \tilde{x}|$$

$$E_r = \frac{|x - \tilde{x}|}{|x|} = \frac{E_a}{|x|}$$

L'errore assoluto è legato al numero delle cifre della parte frazionaria corrette, cioè al numero di cifre dopo il punto frazionario che sono uguali in x e $\tilde{x}$.
L'errore relativo è legato al numero delle cifre significative corrette, cioè al numero di cifre significative che sono uguali in x e $\tilde{x}$.

Per esempio,

$$x = 8467915.78755, \tilde{x} = 8467915.78654, E_a = 0.10 \cdot 10^{-2}, E_r = 0.11 \cdot 10^{-9}$$

$$x = 741.329642, \tilde{x} = 741.320531, E_a = 0.91 \cdot 10^{-2}, E_r = 0.12 \cdot 10^{-4}$$

$$x = 0.00741329642, \tilde{x} = 0.00741320531, E_a = 0.91 \cdot 10^{-7}, E_r = 0.12 \cdot 10^{-4}$$

$$x = 0.0000326, \tilde{x} = 0.0000215, E_a = 0.11 \cdot 10^{-4}, E_r = 0.3$$

L'errore relativo è spesso espresso in percentuale. Per gli esempi precedenti si ha: 0.000000011%, 0.0012%, 30%.

Proprietà dell'errore assoluto:
Se $\tilde{x}$ è una approssimazione di x corretta a p cifre frazionarie, allora

$$E_a = |x - \tilde{x}| < b^{-p}.$$

Se la base è 10 $\quad E_a = |x - \tilde{x}| < 10^{-p}$.

Proprietà dell'errore relativo:
Se $\tilde{x}$ è una approssimazione di x corretta a p cifre significative, allora

$$E_r = \frac{|x - \tilde{x}|}{|x|} < b^{-p+1}.$$

Se la base è 10 $E_r = \frac{|x - \tilde{x}|}{|x|} < 10^{-p+1}$.

Da tali proprietà discende che

- se E_a è un numero piccolo rispetto a 1, nulla può dirsi a proposito della bontà dell'approssimazione di x con $\tilde{x}$; solo se E_a è piccolo rispetto a x, allora si può concludere che $\tilde{x}$ è una buona approssimazione di x
- se E_r è un numero piccolo rispetto a 1, allora si può concludere che $\tilde{x}$ è una buona approssimazione di x, indipendentemente dall'ordine di grandezza di x.

Attualmente tutti i PC e le workstation adottano i medesimi valori della base, della precisione e del numero di cifre dell'esponente. Esiste infatti un insieme di convenzioni che viene utilizzato come standard di riferimento dai costruttori di calcolatori, detto *sistema aritmetico standard 754 dell'IEEE*[21]. Il sistema è stato progettato dal matematico americano W. Kahan nel 1987.

Tale sistema prevede che la base sia 2 e che nel caso di locazioni a 32 bit (singola precisione) il numero di bit per la mantissa sia 23 bit e il numero di bit per l'esponente sia 8. L'esponente, che è un intero, è espresso nella forma traslata, con costante di traslazione uguale a 127 (vedi nota[22]).

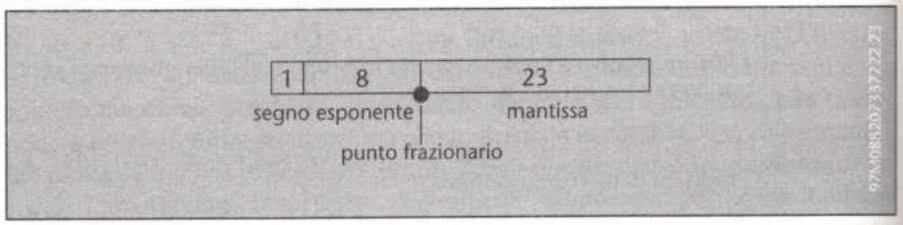


Fig. 2.23: IEEE precisione singola

Nel caso di locazioni a 64 bit[23] (doppia precisione) il numero di bit per la mantissa è 52 bit e il numero di bit per l'esponente è 11.

[21] Institute of Electrical and Electronics Engineers (`http://www.IEEE.org/`).

[22] In tal modo, detto p l'esponente traslato, si ha che $0 \le p \le 255$; $p = 0$ e $p = 255$ sono utilizzati per rappresentare lo zero e per le situazioni eccezionali, per cui l'esponente varia tra -126 e +127. In doppia precisione la costante di traslazione è 1023.

[23] In tutti i calcolatori con locazioni a 32 esiste la possibilità di utilizzare due locazioni consecutive come un'unica locuzione virtuale a 64 bit (*doppia precisione*).

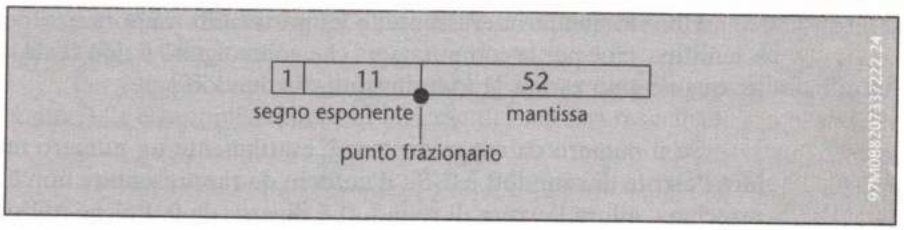


Fig. 2.24: IEEE precisione doppia

Quando la base è due il criterio di normalizzazione implica che la prima cifra (binaria) della mantissa debba essere necessariamente uno (è l'unica cifra in base due diversa da zero!). In considerazione di ciò, il primo bit della mantissa non viene esplicitamente memorizzato nella voce (*rappresentazione col primo bit implicito*); in tal modo i 23 (o 52) bit disponibili per la memorizzazione della mantissa consentono di avere una precisione effettiva di 24 (o 53) bit. Formalmente, il bit implicito viene considerato prima del punto frazionario per cui la mantissa effettiva è $1.0 + f$, dove f è la mantissa memorizzata.

ESEMPIO:

se il contenuto della voce è

0|0000 1110| 1010 000 0000 0000 0000 0000

allora la mantissa effettiva è

1.1010 000 0000 0000 0000 0000 = 1.625

e l'esponente è

00001110 – 01111111 = –01110001 = –113,

il numero è

$$.10100000000000000000000 \times 2^{-01110001} = 1.625 \times 2^{-113}$$

	32 bit singola precisione	64 bit doppia precisione
esponente minimo	-126	-1022
esponente massimo	127	1023
soglia di overflow	$(2-2^{-23})\times 2^{127} \approx 3.4\times 10.^{38}$	$0.179\times 10.^{309}$
soglia di underflow	$2^{-126} \approx 1.2\times 10^{-38}$	0.223×10^{-307}

97M0882073372225

Fig. 2.25: sistema aritmetico standard IEEE

Una domanda interessante, e di particolare importanza per il calcolo scientifico, cioè per le computazioni che coinvolgono il tipo reale, è la seguente: quanto può essere grande l'errore di roundoff?

Se il numero da rappresentare è esattamente un numero macchina, allora l'errore di roundoff è 0. Se il numero da rappresentare non è un numero macchina, allora l'errore di roundoff è diverso da 0. Poiché nel sistema reale il numero di cifre significative per la rappresentazione è fissato, è conveniente utilizzare l'errore relativo per quantificare il roundoff (in tal caso si parla di errore relativo di roundoff). Si ha che:

esiste un *massimo errore relativo di roundoff*, che indichiamo con u, e inoltre

$$u = \frac{1}{2} b^{1-t}.$$

Si noti che nel caso $b = 2$, $u = 2^{-t}$.

Il risultato discende immediatamente dalla proprietà dell'errore relativo.

In conclusione, la risposta alla domanda precedente è:

L'errore relativo di roundoff nella rappresentazione di un qualunque numero reale appartenente all'insieme dei numeri rappresentabili è maggiore (o uguale) di zero e minore (o uguale) di u.

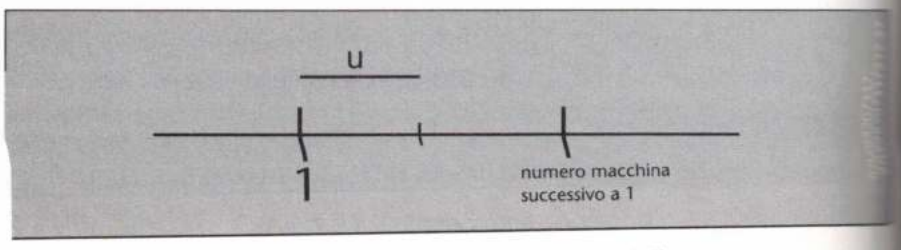


Fig. 2.26: massimo errore relativo di roundoff

In termini più formali, fissato un qualunque numero reale x appartenente all'insieme dei numeri rappresentabili, esiste un numero macchina $fl(x)$ tale che $|x - fl(x)| \le u|x|$ ovvero

$$fl(x) = x(1+r), \text{ con } |r| \le u.$$

È utile osservare che u ha una semplice interpretazione geometrica

è la metà della distanza tra il numero 1 e il numero macchina successivo a 1. (Vedi Fig. 2.26 e nota[24])

Per esempio, nel sistema standard IEEE, in singola precisione, l'intervallo [1,2] è rappresentato dall'insieme dei numeri macchina

$$1,\ 1+2^{-23},\ 1+2\times 2^{-23},\ 1+3\times 2^{-23},\ \ldots,\ 2 \qquad (1)$$

L'intervallo [2.4] è rappresentato dagli stessi numeri moltiplicati per 2

$$2,\ 2+2^{-22},\ 2+2\times 2^{-22},\ 2+3\times 2^{-22},\ \ldots,\ 4$$

In generale, l'intervallo $\left[2^k, 2^{k+1}\right]$ è rappresentato dai numeri in (1) moltiplicati per 2^j. Dunque la spaziatura tra due numeri macchina consecutivi non è mai, in termini relativi, maggiore di $2^{-23} \approx 1.192 \times 10^{-7}$.

Il massimo errore relativo di roundoff nel sistema standard IEEE è $u = 2^{-24} = 0.596\times 10^{-7}$, in singola precisione, e $u = 2^{-53} = 0.111\times 10^{-15}$, in doppia precisione.

È importante sottolineare che ciò significa che non è possibile rappresentare in un qualunque calcolatore oggi sul mercato, numeri con più di sette cifre significative (in base dieci) e che di conseguenza non è possibile ottenere dall'esecuzione di un qualsiasi programma un risultato con più di sette cifre significative accurate, utilizzando la singola precisione. Analogamente, utilizzando la doppia precisione, non è possibile rappresentare numeri con più di quindici cifre significative (in base dieci) e la massima accuratezza richiedibile a una computazione scientifica è di quindici cifre significative.

Sul tipo reale sono definite le operazioni di *addizione floating-point*, *sottrazione floating-point*, *moltiplicazione floating-point*, *divisione floating-point*.

Tali operazioni hanno, come le corrispondenti operazioni sui numeri reali, due operandi e un risultato.

Solo in questo paragrafo indicheremo con i simboli $\oplus$, $\ominus$, $\otimes$, $\varnothing$ le operazioni sul tipo reale al fine di sottolinearne la diversità rispetto alle operazioni sui numeri reali.

[24] Per semplicità la dimostrazione si riferisce al caso $b = 2$. Poiché $1 = .100\cdots 00\times 2^1$, il successivo numero macchina è $.100\cdots 01\times 2^1 = .100\cdots 0\times 2^1 + .000\cdots 01\times 2^1 = 1+2^{-t}\cdot 2$, da cui segue che $\frac{1}{2}\left(1+2^{-t+1}-1\right) = 2^{-t} = u$.

Il risultato di una operazione floating-point tra due numeri macchina x e y, che indichiamo genericamente con $\oplus$, è il seguente

$$x \oplus y = \begin{cases} fl(x\#y) & \text{se } x\#y \text{ è rappresentabile} \\ indefinito & \text{se } x\#y \text{ non è rappresentabile} \end{cases}$$

Questo significa che il risultato di una operazione floating-point (quando non è indefinito[25]) è il numero macchina dato dal risultato della corrispondente operazione sui numeri reali arrotondato alla precisione del sistema aritmetico reale.

In termini formali, si ha che:

$$x \oplus y = (x\#y)(1+r) \qquad \text{con} \quad |r| \le u$$

Tale proprietà si sintetizza dicendo che il *sistema aritmetico garantisce la massima accuratezza*.

Si noti che in generale se x e y sono due numeri macchina il risultato di una operazione aritmetica (la vera, non quella floating-point!) non è necessariamente un numero macchina. Si pensi ad esempio che il prodotto di due numeri di t cifre può avere 2t cifre.

L'errore che si introduce rappresentando il risultato dell'operazione $x \# y$ con quello dell'operazione $x \oplus y$, cioè con $fl(x \# y)$ è detto, ancora, *errore di roundoff*.

Anche a proposito delle operazioni floating-point è naturale porsi la domanda: quanto può essere grande l'errore di roundoff del risultato di una operazione aritmetica?

La risposta è che, nel caso di sistema aritmetico che garantisce la massima accuratezza, l'errore relativo di roundoff è maggiore (o uguale) di zero e minore (o uguale) di u.

È importante sottolineare che per ottenere un sistema aritmetico che garantisce la massima accuratezza è necessario che l'hardware del processore sia progettato in modo opportuno. In particolare, il requisito di massima accuratezza si traduce in vincoli sulla struttura dei registri interni su cui opera l'unità logico-aritmetica.

Per rendersi conto di ciò, consideriamo in dettaglio come opera l'unità aritmetica per effettuare una operazione di addizione floating-point; supporremo dapprima che i registri interni abbiano la stessa struttura delle voci di memoria (4 cifre per la mantissa, in base 10, nell'esempio).

[25] In particolare, si ha che il risultato è indefinito per underflow se $|x\#y| < r_{\min}$, o per overflow se $|x\#y| > r_{\max}$

ESEMPIO:

l'operazione $0.9983 \cdot 10^2 \oplus 0.4652 \cdot 10^{-1}$ in un sistema aritmetico con registri a precisione uguale alla precisione di memoria viene effettuata nel seguente modo:[26]

passo 1. operandi nei registri

9	9	8	3		+	2
mantissa					esponente	

4	6	5	2		-	1
mantissa					esponente	

passo 2. incolonnamento: la mantissa dell'operando con esponente minore viene traslata a destra un numero di volte pari alla differenza tra gli esponenti degli operandi

9	9	8	3		+	2
mantissa					esponente	

0	0	0	4		+	2
mantissa					esponente	

passo 3. somma delle mantisse ed eventuale normalizzazione del risultato

9	9	8	3		+	2	
0	0	0	4		+	2	
9	9	8	7		+	2	risultato
mantissa					esponente		

Il risultato calcolato è $0.9987 \cdot 10^2$; il risultato dell'operazione sui reali, arrotondato alla precisione 4, è $0.9988 \cdot 10^2$. Il sistema aritmetico presentato in questo esempio non fornisce la massima accuratezza.

[26] Si ricorda che mantissa 9983 significa 0.9983; in pratica, della mantissa non viene memorizzata la parte intera.

Al contrario, se i registri interni hanno precisione doppia rispetto alle voci di memoria si avrebbe massima accuratezza.

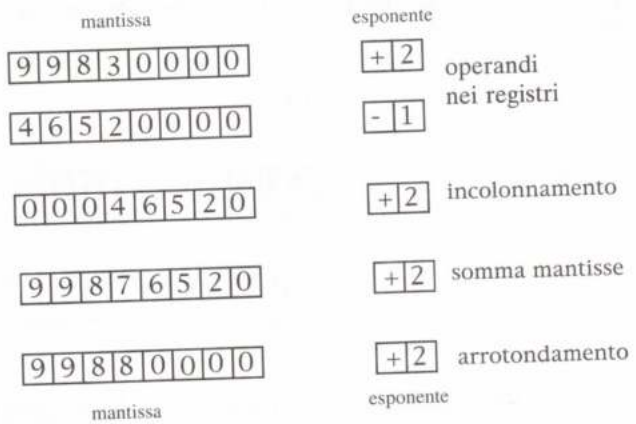


Fig. 2.27

Il risultato calcolato è $0.9988 \cdot 10^2$; questo sistema aritmetico è di massima accuratezza.

Un'altra conseguenza importante della precisione finita è la seguente: può accadere che sommando (somma floating-point) a un numero macchina non nullo x un altro numero macchina non nullo y il risultato sia uguale al primo addendo, cioè $x \oplus y = x$.

Questo significa che, oltre allo zero, esistono numeri macchina che si comportano come lo zero nella somma floating-point.

Se si considera la somma floating-point $1 \oplus y$, allora esistono vari numeri macchina y non nulli tali che

$$1 \oplus y = 1.$$

Si dice *epsilon macchina*, e viene denotato con ε_M, il più piccolo numero macchina tale che:

$$1 \oplus \varepsilon_M > 1.$$

L'esistenza dell'epsilon macchina può essere confermata sperimentalmente eseguendo su un qualsiasi calcolatore un algoritmo del tipo:

```
dare il valore 1 all'addendo
ripetere
   dare il valore addendo/2   all'addendo
```

```
finché 1+addendo è uguale a 1
stampare il valore di addendo*2
```

In singola precisione si ha

$$\varepsilon_M = 1.192092 \times 10^{-7}$$

In doppia precisione si ha

$$\varepsilon_M = 2.220446 \times 10^{-16}$$

Si noti che $\varepsilon_M = 2\,u$.

L'epsilon macchina individua il sottoinsieme dei numeri macchina che si comportano come lo zero nella somma floating-point con 1: si tratta dei numeri macchina appartenenti all'intervallo $[0, \varepsilon_M)$.

Si noti che l'epsilon macchina *non è* il più piccolo numero reale rappresentabile, ma anzi ε_M è di molti ordini di grandezza più grande di $r_{\min}$.

Se il primo addendo x è un numero macchina (che supponiamo positivo per semplicità), vale la seguente generalizzazione dell'epsilon macchina:

Se si considera la somma floating-point $x \oplus y$, allora esistono vari numeri macchina y non nulli tali che

$$x \oplus y = x \ .$$

Il più piccolo numero macchina z tale che:

$$x \oplus z > x \ ,$$

è detto *epsilon relativo* (al numero x), e si ha che $z = x \cdot \varepsilon_M$. L'epsilon relativo individua i numeri macchina che si comportano come lo zero nella somma floating-point con x.

Una importante conseguenza di questa proprietà è che nel progetto di un algoritmo da eseguire in un sistema a precisione finita bisogna prestare attenzione al caso in cui si sommano numeri che differiscono tra loro per molti ordini di grandezza[27].

[27] In generale, è necessario prestare attenzione alla cosiddetta scala dei dati su cui si opera; spesso è opportuno, quando si formula il problema da risolvere, agire sulle unità di misura dei dati in modo da avere una scala uniforme.

Alla luce delle caratteristiche del tipo reale può risultare naturale porsi la seguente domanda:

quanto è affidabile un sistema aritmetico a precisione finita?

Ovvero:

qual è l'accuratezza complessiva ottenibile dagli algoritmi che operano su dati di tipo reale, visto che ogni dato di input e il risultato di ogni operazione aritmetica floating-point sono affetti da un errore (l'errore di roundoff)?

La risposta esauriente a questa domanda pervade l'intera disciplina del calcolo. In breve, si può dire che si possono verificare due scenari:

- il primo è lo scenario positivo. I singoli errori di roundoff si compongono in modo da determinare un errore complessivo trascurabile, sul risultato.
- il secondo è lo scenario negativo. I singoli errori di roundoff si compongono in modo da determinare un errore complessivo non trascurabile, a volte tale da inquinare completamente il risultato.

Si noti che è l'algoritmo, cioè la sequenza delle operazioni, che determina lo scenario.

Nel primo caso, si parla di un algoritmo *stabile*, cioè di un algoritmo che non è sensibile alla precisione finita.

Nel secondo caso, l'algoritmo è detto *instabile*, cioè l'algoritmo risente in modo inaccettabile delle conseguenze della precisione finita.

La proprietà di stabilità è un requisito fondamentale nel progetto di un algoritmo che opera su dati di tipo reale.

Il problema della stabilità non si pone per algoritmi che operano su dati di tipo intero o di tipo carattere.

Quando il risultato di una operazione floating-point è indefinito, cioè il risultato della corrispondente operazione sui numeri reali è un numero che non appartiene all'insieme dei numeri rappresentabili, si dice che si è verificata una *situazione eccezionale* (*exception*).

Prima dell'introduzione del sistema aritmetico standard IEEE la gestione delle situazioni eccezionali differiva da calcolatore a calcolatore. Lo stesso programma eseguito da diversi calcolatori forniva un diverso risultato. In genere, quando si verificava una situazione eccezionale l'esecuzione del programma terminava.

Il sistema aritmetico IEEE prevede una gestione completa delle situazioni eccezionali. L'idea è che l'esecuzione del programma continui anche dopo il verificarsi di una eccezione. In particolare il risultato di una operazione che provoca una eccezione è determinato in modo convenzionale come riportato nella seguente tabella, in cui sono classificate le quattro categorie di situazioni eccezionali previste.

Eccezione	Causa	Risultato
Overflow		±Infinito
Underflow		0 oppure numero denormalizzato[28]
Divisione per zero con x finito e non nullo	x/0	±Infinito
Operazione invalida	Infinito-Infinito, 0*Infinito	NaN
	Infinito/Infinito, 0/0	NaN
	confronto con NaN	NaN

Fig. 2.28: gestione delle eccezioni nel sistema IEEE

Il sistema aritmetico IEEE consente di rappresentare anche l'infinito con segno e il NaN (acronimo per Not a Number, cioè *non numero)*, cioè il risultato di una operazione non definita sui numeri reali (operazione invalida). Il sistema inoltre è in grado di determinare anche il risultato di operazioni floating-point che coinvolgono come operandi queste due entità convenzionali.

Il problema della gestione della "finitezza" così come delle eccezioni può sembrare a prima vista frutto di una eccessiva attenzione ai dettagli. Tuttavia non sono rari i casi in cui si sono avute conseguenze eclatanti da una gestione impropria. Il caso del fallimento del lancio del vettore europeo Ariane 5 del giugno 1996 e quello di alcuni missili-antimissile Patriot durante la guerra del Golfo del 1991 sono esemplari.
Ariane è esploso in volo a causa di un errato angolo di attacco.
(Il rapporto dettagliato della commissione scientifica di inchiesta è consultabile al sito: `http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html`).
Il calcolo di tale angolo è risultato essere sbagliato a causa di un overflow intero verificatosi nel passare da un numero del tipo reale in doppia precisione al corrispondente numero di tipo intero (voce a 16 bit).
Vari missili Patriot durante la guerra del Golfo fallirono l'intercettazione di un missile nemico a causa di un errore di rotta, causato dal fatto che il calcolatore di bordo non arrotondava correttamente alla precisione finita il valore dell'ora fornito dall'orologio; invece dell'arrotondamento veniva effettuato un troncamento, cioè venivano semplicemente eliminate le cifre oltre la precisione del calcolatore (Ciò comporta che un numero reale viene rappresentato non dal numero macchina più vicino, ma da quello che lo precede).

[28] I numeri denormalizzati sono numeri con mantissa non normalizzata ed esponente minimo. Il loro uso è previsto dal sistema aritmetico IEEE.

Un ulteriore esempio dell'importanza del funzionamento del sistema aritmetico è fornito dal caso della prima versione del processore Intel Pentium, la cui unità logico-aritmetica conteneva un errore progettuale. In particolare, si aveva che, per la divisione floating-point in doppia precisione, il sistema non garantiva la massima accuratezza, ma restituiva un risultato con una accuratezza di 14, invece di 15, cifre significative (in base dieci). Ciò comportava che, per esempio, il calcolo della seguente espressione, coinvolgente numeri e operazioni floating-point in doppia precisione,

4195835- (4195835/3145727)*3145727

forniva, invece del risultato corretto 0, il risultato errato 256. La stessa computazione effettuata con un processore Intel 286, 386 e 486 fornisce correttamente 0. La causa di tale malfunzionamento fu in breve individuata e rimossa.

In conclusione, il tipo reale consente di riprodurre in memoria in modo approssimato un qualunque numero reale x tale che $r_{\min} \leq |x| \leq r_{\max}$, con un errore, detto di roundoff, che in termini relativi è minore (o uguale) di u. Analogamente, il risultato di una operazione aritmetica effettuata dal calcolatore (operazioni floating-point), quando non sia under/overflow, è una approssimazione del risultato della corrispondente operazione sui numeri reali, con un errore relativo di roundoff minore (o uguale) di u.

Vogliamo formulare istruzioni macchina, ... e vedere quanta capacità di controllo esse racchiudono e quanto tempo richiedono per essere eseguite.

John Von Neumann, 1944

2.2.2 Rappresentazione delle istruzioni

Nel paragrafo precedente sono stati esaminati i criteri di rappresentazione dei dati elementari e le operazioni elementari che è possibile effettuare su di essi. La rappresentazione delle istruzioni si riferisce al criterio con cui sono denotate all'interno della memoria tali operazioni. L'insieme delle istruzioni elementari (insieme con i tipi di dati elementari) costituisce il *linguaggio macchina*, cioè il linguaggio con cui devono essere espressi gli algoritmi per poter essere eseguiti dall'hardware del calcolatore.

Calcolatori diversi hanno in genere linguaggi macchina diversi, con differenze che possono essere molto accentuate. Non è insolito un calcolatore che abbia un linguaggio macchina costituito da più di cento istruzioni.

Nel seguito vengono accennati solo i tratti salienti di un linguaggio macchina.

Le principali istruzioni che denotano una azione sui dati sono: le istruzioni (per le operazioni) di confronto tra due dati dello stesso tipo, le istruzioni di concatenamento di byte, le istruzioni aritmetiche sul tipo intero, le istruzioni aritmetiche sul tipo reale, le istruzioni logiche.

I dati su cui operano tali istruzioni possono essere memorizzati in memoria oppure nei registri interni del processore. Vi sono istruzioni che consentono di spostare un dato da una voce di memoria a uno dei registri[29] e di spostare un dato da un registro alla memoria.

Inoltre vi sono istruzioni che consentono di esprimere i costrutti di controllo: le istruzioni di *salto* (che indicano, cioè, l'indirizzo della prossima istruzione da eseguire) *non condizionato* e *condizionato* (una forma elementare del costrutto di selezione) e una istruzione che realizza un costrutto ripetitivo.

Le informazioni che costituiscono una istruzione sono l'azione da effettuare e uno o più indirizzi, che indicano, a secondo dell'operazione, le voci di memoria di operandi e risultato, oppure la prossima istruzione, i registri del processore, ecc.

L'organizzazione più semplice consiste nel rappresentare una istruzione del linguaggio macchina con un formato fisso, costituito da un *codice operativo*, che specifica l'operazione da eseguire, e uno o due indirizzi.

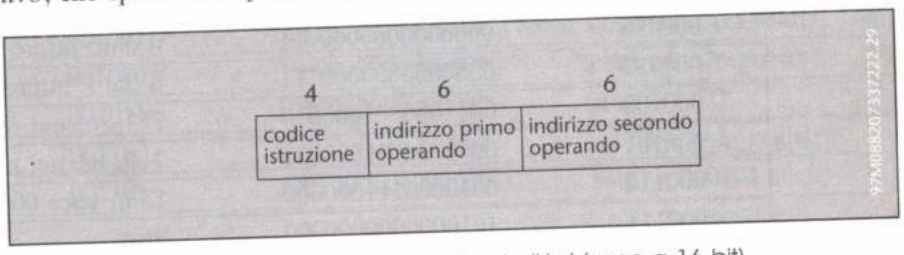


Fig. 2.29: istruzione a due indirizzi (voce a 16 bit)

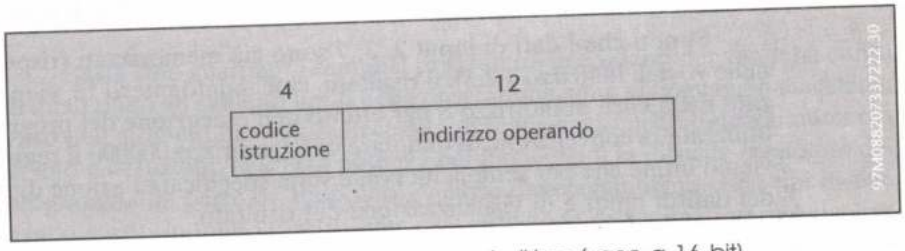


Fig. 2.30: istruzione a un indirizzo (voce a 16 bit)

[29] Di solito sono presenti da 8 a più di 100 registri, a seconda del tipo di processore.

Di solito, gli indirizzi di una istruzione a due indirizzi individuano i due operandi; il risultato dell'operazione viene memorizzato in un registro. Negli esempi che seguono consideriamo il caso semplificato di un unico registro del processore.

Un esempio di istruzioni elementari per un calcolatore con voci a 16 bit è il seguente:

0001 somma intera; indirizzi dei due operandi; risultato nel registro.
0011 moltiplicazione intera; indirizzi dei due operandi; risultato nel registro
0101 divisione intera; indirizzi dei due operandi; risultato nel registro.
0010 copia il contenuto del registro; indirizzo della voce in cui trasferire.
1010 stop.

ESEMPIO:

il calcolo del valore dell'espressione 2*3+7 richiede l'esecuzione di due operazioni: 2*3, e poi 7+ il risultato dell'operazione precedente.

In tabella si riporta il programma in linguaggio macchina che genera tale sequenza di operazioni.

indirizzo	contenuto (binario)	legenda
000001	0000000000000010	il dato intero 2
000010	0000000000000011	il dato intero 3
000011	0000000000000111	il dato intero 7
000100	0011000001000010	2*3, ris. nel registro
000101	0001000011000000	7+6, ris. nel registro
000110	0010000111000000	13 in voce 000111
000111	1010000000000000	Stop
001000	0000000000001101	il dato intero 13

Si noti che i dati di input 2, 3, 7 sono già memorizzati (rispettivamente nelle voci di indirizzo 1,2,3); il risultato, cioè il dato intero 13, viene memorizzato nella voce di indirizzo 8 per effetto dell'esecuzione del programma. Si è utilizzata la convenzione di denotare con l'indirizzo 000000 il registro. Osserviamo infine che per semplicità non è stata specificata l'azione di immissione dei dati di input e di visualizzazione del risultato.

Un ulteriore esempio di istruzioni elementari per un calcolatore con voci a 16 bit è il seguente:

1001 salto; indirizzo della prossima istruzione.
1011 azzeramento del registro.

1101 incremento di 1 del contenuto del registro.
0110 copia nel registro; indirizzo della voce da cui trasferire.
0100 se il registro contiene 'falso' salta l'istruzione successiva.
0111 confronto (uguaglianza) tra interi; indirizzi dei due operandi; risultato (vero, falso) nel registro.

ESEMPIO:

il seguente programma in linguaggio macchina descrive un algoritmo per il calcolo della somma dei primi 8 numeri naturali.

indirizzo	contenuto (binario)	legenda
000001	0000000000001001	il dato intero 9
000010	0000000000000000	dato locale
000011	0000000000000000	risultato
000100	1011000000000000	azzeramento registro
000101	1101000000000000	incremento registro
000110	0010000010000000	da registro in 000010
000111	0001000000000011	somma (reg. e 000011)
001000	0010000011000000	da registro in 000011
001001	0111000010000001	confronto tra dati
001010	0100000000000000	confronto registro
001011	1010000000000000	stop
001100	0110000010000000	da 000010 in registro
001101	1001000101000000	salto a 000101

Il risultato al termine dell'esecuzione, è il contenuto della voce di indirizzo 000011.

Basta solo guardare i semplici programmi precedenti per rendersi conto di quali siano gli evidenti limiti della programmazione in linguaggio macchina: l'uso dell'alfabeto binario; la necessità di tener conto degli indirizzi numerici di dati e istruzioni; la scarsa potenza del linguaggio; la difficile *leggibilità* del programma da parte di un uomo. A ciò si deve aggiungere che i tipi di dati vanno gestiti in modo diretto (per es. l'istruzione di somma intera è diversa dall'istruzione di somma reale) e che il programma non è *portabile*, cioè è eseguibile solo su calcolatori dotati dello stesso processore.

Un modo per semplificare la notazione con cui descrivere un programma in linguaggio macchina (e per migliorare la sua leggibilità) consiste nell'utilizzare nomi convenzionali al posto dei codici operativi delle istruzioni e

nomi simbolici per indicare i dati in luogo di indicare l'indirizzo dove ogni dato è memorizzato. Un tale linguaggio è detto *linguaggio assembler*.

ESEMPIO:

se si pone

MPYI	in luogo di 0011, codice operativo della moltiplicazione intera
ADDI	in luogo di 0001, codice operativo della somma intera
STORE	in luogo di 0100, codice operativo del trasferimento da registro a voce di memoria
STOP	in luogo di 1010
R	in luogo del registro
A	in luogo dell'indirizzo 000010 del dato 2
B	in luogo dell'indirizzo 000011 del dato 3
C	in luogo dell'indirizzo 000111 del dato 7
RIS	in luogo dell'indirizzo 000111 del risultato

allora la versione in linguaggio assembler del programma in linguaggio macchina dell'esempio 2*3+7 è :

immissione dei dati
MPYI A,B
ADDI C, R
STORE RIS
visualizzazione risultato
STOP

ESEMPIO:

se si pone

GOTO	in luogo di 1001, codice operativo dell'istruzione di salto
ZEROREG	in luogo di 1011, azzeramento del registro
INCRREG	in luogo di 1101, incremento del registro.
LOAD	in luogo di 0110, copia nel registro
IFREG	in luogo di 0100, costrutto di selezione basato sul valore del registro
IFEGUAL	in luogo di 0111, confronto (uguaglianza) tra dati
R	in luogo del registro
A	in luogo dell'indirizzo 000001 del dato 9
B	in luogo dell'indirizzo 000010 del dato locale
RIS	in luogo dell'indirizzo 000011 del risultato

allora la versione in linguaggio assembler del programma in linguaggio macchina dell'esempio "somma dei primi 8 numeri interi" è:

```
  immissione dei dati
  ZEROREG
1 INCRREG
  STORE B
  ADDI R,RIS
  STORE RIS
  IFEGUAL A,B
  IFREG
  STOP
  LOAD B
  GOTO 1
```

I due esempi precedenti si riferiscono a un linguaggio macchina e un linguaggio assembler fittizi, utilizzati esclusivamente per mostrare alcune caratteristiche di base di questi linguaggi. In realtà i linguaggi macchina e assembler di processori realmente esistenti sono ben più completi. Per esempio, già nel caso del vecchio e *semplice* processore Intel 8088 (lontano antenato degli attuali Pentium) esistono 14 registri indirizzabili più un registro aggiuntivo *senza nome* e centinaia di istruzioni macchina.

L'esempio Algoritmo 7. del par. 1.1 (somma dei primi 4 numeri naturali) potrebbe essere, nell'assembler dell'8088

```
immissione dei dati
MOV AX,0
MOV RIS,AX
MOV CX,4
RIP
  INC AX
  ADD RIS,AX
  LOOP RIP
```

dove AX e CX sono due registri, RIS contiene la somma e l'istruzione LOOP indica un costrutto di controllo di ripetizione della sequenza delle istruzioni tra l'etichetta RIP e l'istruzione LOOP stessa; il contatore è il registro CX.

La versione assembler dei programmi degli esempi è più *leggibile* rispetto a quella in linguaggio macchina e consente di ovviare ad alcune limitazioni di questo.

Un programma in linguaggio assembler non è eseguibile dall'hardware.

Un programma in linguaggio assembler deve essere prima trasformato nel corrispondente programma in linguaggio macchina e poi eseguito dall'hardware.

Una simile trasformazione è detta *traduzione*.

L'osservazione fondamentale è che la traduzione può essere effettuata da un programma.

Un programma che trasforma (il testo di) un programma in linguaggio assembler nel (testo del) corrispondente programma in linguaggio macchina è detto *assemblatore*.

Non è necessario avere una infinità di macchine diverse per svolgere compiti diversi. Sarà sufficiente averne una sola. I problemi che sorgono quando si debbono produrre varie macchine per diversi compiti, si trasformeranno in un lavoro a tavolino, quello cioè che consiste nel programmare la macchina universale a svolgere quei compiti.

Alan Turing, 1940

2.3 Dalla macchina fisica alla macchina virtuale

Nei paragrafi precedenti sono state sintetizzate le caratteristiche dell'hardware di un calcolatore.

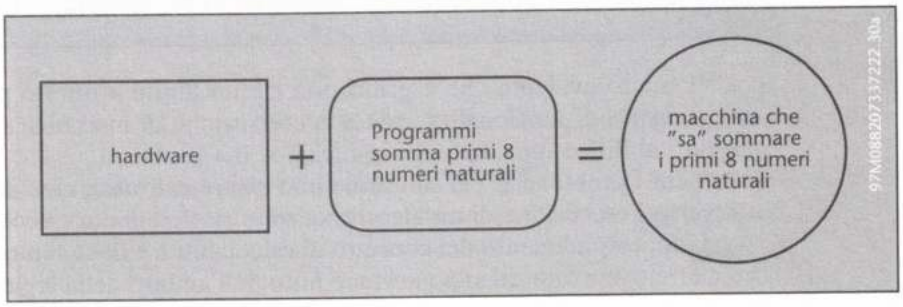
Dal punto di vista di un utilizzatore si può concludere che la macchina fisica:

- è in grado di eseguire programmi costituiti da un numero anche molto grande di istruzioni e di dati;
- è in grado di memorizzare sui dispositivi di memoria secondaria un grande numero di informazioni (istruzioni e dati) e di trasferirli da memoria secondaria a memoria in tempi molto brevi;
- le istruzioni elementari denotano operazioni molto semplici: ad esempio, l'addizione di due numeri interi, la moltiplicazione di due numeri reali, lo spostamento di un dato tra due voci di memoria.
- i tipi di dati elementari sono molto semplici; non esistono istruzioni del linguaggio macchina per costruire in modo esplicito nuovi tipi o per indicare relazioni tra dati;
- l'esecuzione di una istruzione da parte del processore richiede un tempo dell'ordine delle decine di nanosecondi, sicché si possono in genere eseguire tra 100 milioni e un miliardo di istruzioni al secondo.

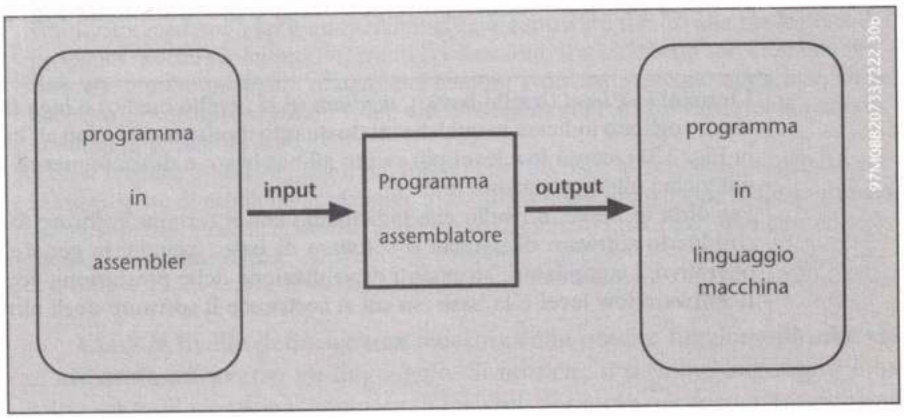
Queste caratteristiche sono sufficienti per realizzare la macchina universale sognata da Babbage.

L'idea non è di costruire un hardware più complesso in grado di espletare più funzionalità di quelle sopra riportate, bensì di realizzare funzionalità più complesse in modo virtuale, cioè attraverso programmi.

Si consideri, ad esempio, il programma in linguaggio macchina per la somma dei primi 8 numeri naturali sviluppato nel par. 2.2.2. L'insieme costituito dall'hardware e da tale programma definisce una nuova macchina che ha tutte le funzionalità dell'hardware e in più è in grado di calcolare la somma dei primi 8 numeri naturali. Questa è una macchina *virtuale*, nel senso che alcune sue funzionalità sono realizzate attraverso programmi. Si osservi che per un utilizzatore di tale macchina l'operazione di somma dei primi 8 numeri naturali è una operazione elementare.

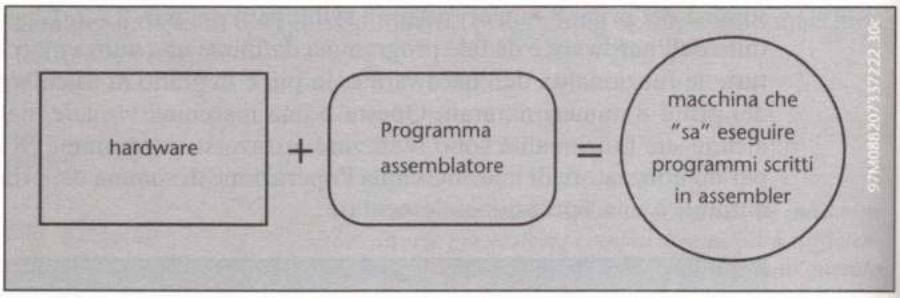


Nel par 2.2.2 è stata introdotta l'idea di usare una notazione più semplice (dal punto di vista dell'uomo) per descrivere un programma in linguaggio macchina, il linguaggio assembler. L'utilizzo effettivo del linguaggio assembler richiede l'esistenza del programma *assemblatore* che effettua la traduzione del testo di programma scritto in assembler nel testo del corrispondente programma scritto in linguaggio macchina.



L'insieme costituito dall'hardware e dall'assemblatore è una nuova macchina che, oltre alle funzionalità dell'hardware, è in grado di eseguire un

qualunque programma scritto in assembler. Si tratta di una macchina virtuale perché alcune funzionalità sono realizzate attraverso programmi. Un utilizzatore di tale macchina vede come elementare il linguaggio assembler.



È subito evidente che a priori non c'è un limite a questo processo di arricchimento di funzionalità, cioè alla costruzione di macchine virtuali con maggiori abilità e funzioni nella risoluzione dei problemi.

Tutti i problemi la cui soluzione può essere *calcolata*, cioè determinata attraverso l'esecuzione di un algoritmo, sono risolvibili dal calcolatore[30].

L'approfondimento del concetto di calcolabilità e delle limitazioni a ciò che è effettivamente calcolabile viene fatto nell'ambito della logica matematica e non è tra gli scopi di questo testo.

Nel seguito descriveremo brevemente l'architettura della parte virtuale di un calcolatore, cioè le principali funzionalità del software che contribuisce alla realizzazione dei calcolatori attuali.

Procedendo in modo analogo alla descrizione dell'architettura hardware, l'architettura della parte virtuale viene presentata individuando livelli e moduli con funzionalità specifiche.

Una possibile suddivisione per livelli è mostrata nella Fig. 2.31.

I termini *low level* (livello basso), *medium level* (livello medio) e *high level* (livello alto), vogliono indicare in qualche modo quanto il software è vicino all'hardware (H in Fig. 2.31), con il low level più vicino all'hardware e di conseguenza l'high level più vicino alle applicazioni reali.
Per dirla in breve, il livello che indichiamo con il termine *software low level* è il cosiddetto software di sistema o software di base: include, in genere, il sistema operativo, i compilatori, strumenti di valutazione delle prestazioni, ecc...
Il software low level è la base su cui si costruisce il software degli altri livelli.

[30] Nel 1936 il matematico inglese Alan Turing (1912 -1954) nel famoso articolo *On computable numbers* dimostrò formalmente che tutto quello che un uomo può calcolare può essere calcolato anche da un macchina.

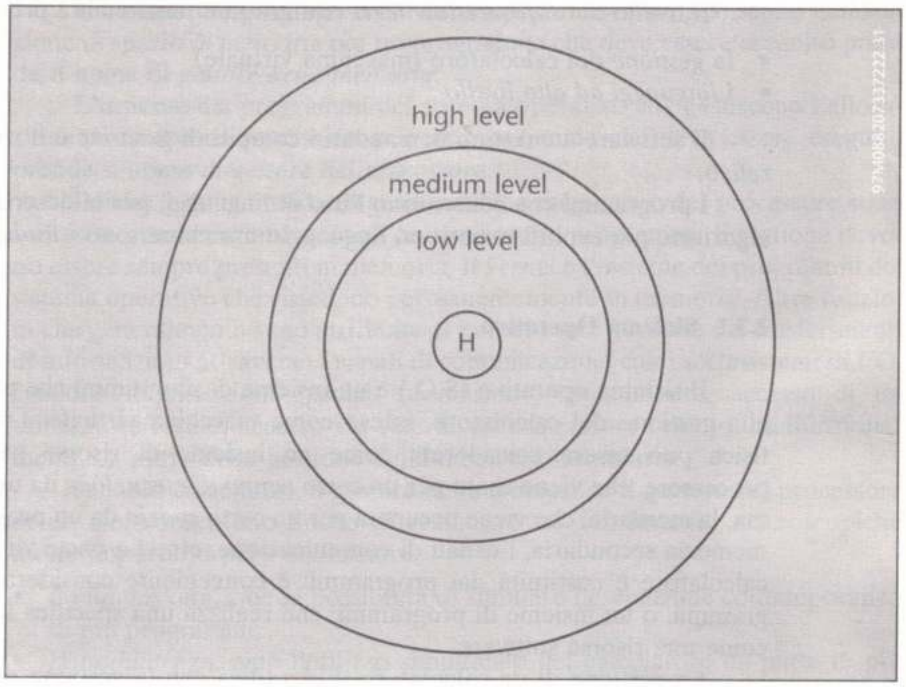


Fig. 2.31: livelli di software

Una caratteristica indispensabile di questo livello di software è la capacità di sfruttare appieno tutte le potenzialità dell'hardware, per cui è necessario un suo continuo e rapido evolversi, all'interno di precisi standard di riferimento, insieme con la tecnologia hardware.
Il *software medium level* è essenzialmente quel software rivolto alla risoluzione dei problemi (nuclei) comuni a differenti applicazioni. Il cosidddetto software matematico, che risolve problemi matematici comuni a diversi problemi applicativi, è un esempio di software medium level. Un altro esempio è il software per la realizzazione di applicazioni web.
Con il termine *software high level*, infine, ci si riferisce al software progettato per la risoluzione di problemi applicativi, quali ad esempio simulazione di flussi attorno ad un velivolo, modelli climatici, andamento di mercati finanziari, applicazioni web, grafica e multimedialità.

Ciascun livello definisce una macchina con precise funzionalità, che viene utilizzata attraverso un linguaggio. Si noti che il termine linguaggio è qui utilizzato nell'accezione generica di insieme di regole per formulare richieste di esecuzione di operazioni. Per esempio, l'interazione con l'interfaccia grafica uomo-sistema (GUI, Graphical User Interface) viene effettuata attraverso il mouse, l'uso di *bottoni virtuali*, di *menu*, etc.

Il livello del *software low level* realizza due funzionalità principali:

- la *gestione* del calcolatore (macchina virtuale)
- i *linguaggi ad alto livello*

Il software a cui sono demandati i compiti di gestione è il *sistema operativo*.

I programmi che consentono l'uso di linguaggi, per la descrizione degli algoritmi, più evoluti rispetto al linguaggio macchina sono i *traduttori*.

2.3.1 Sistema Operativo

Il sistema operativo (S.O.) è un insieme di programmi che provvedono alla gestione del calcolatore, inteso come macchina virtuale. La macchina fisica può essere considerata come un insieme di risorse hardware: il processore, che viene usato per un certo *tempo di esecuzione* da un programma, la memoria, che viene occupata per un certo *spazio* da un programma, la memoria secondaria, i canali di comunicazione, etc.. La parte virtuale di un calcolatore è costituita dai programmi: è conveniente considerare un programma, o un insieme di programmi, che realizza una specifica funzionalità come una risorsa software.

La gestione di un calcolatore si identifica con la gestione delle risorse hardware e software che lo costituiscono.

Dal punto di vista dell'utilizzatore, il livello del sistema operativo è una macchina virtuale che viene utilizzata attraverso un linguaggio convenzionale, detto linguaggio di comando. Tale linguaggio consente di formulare richieste di utilizzo di una risorsa. In molti calcolatori attuali l'uso di tale linguaggio di comando è apparentemente eliminato e sostituito da una interfaccia grafica (realizzata dai programmi GUI) che consente di formulare richieste al sistema operativo attraverso un semplice click del mouse.

La principale funzionalità di un sistema operativo è di consentire l'utilizzo di una risorsa software, cioè di consentire l'esecuzione di un programma (qui inteso come programma scritto in linguaggio macchina e pronto per essere eseguito dal processore).

Il processore esegue un programma solo se questo è immagazzinato nella memoria. Si dice che un programma è *in esecuzione* a un certo istante se il processore in quell'istante sta eseguendo una delle istruzioni del programma.

Un programma che non è in esecuzione viene normalmente immagazzinato in memoria secondaria.

L'esecuzione di un programma richiede quindi due azioni fondamentali, che vengono effettuate dal sistema operativo:

- trasferire il programma in memoria (allocazione di spazio di memoria);
- mettere a disposizione il processore (allocazione di tempo di processore).

L'insieme dei programmi del sistema operativo che gestiscono l'allocazione di spazio di memoria per un programma che deve essere eseguito prende il nome di *gestore della memoria*;

L'insieme dei programmi del sistema operativo che gestiscono l'allocazione di tempo di processore per un programma che deve essere eseguito prende il nome di *gestore del processore*.

Poiché le funzionalità di gestione della memoria e del processore sono utilizzate quasi continuamente, i corrispondenti programmi di gestione devono essere sempre presenti in memoria. Il *kernel* è l'insieme dei programmi del sistema operativo che risiedono permanentemente in memoria. Altre funzioni che generalmente sono attribuite al kernel sono: gestione dei trasferimenti di informazioni attraverso i canali di comunicazione con i sottosistemi di I/O, gestione di situazioni speciali (accensione, spegnimento, accesso di un utilizzatore locale, accesso via rete di un utilizzatore remoto, malfunzionamenti,...), parte della gestione della memoria secondaria.

In molti calcolatori, il gestore della memoria e il gestore del processore di un s.o. consentono anche di realizzare tre funzionalità macroscopiche molto importanti per l'utilizzatore:

- il *multitasking*, cioè la possibilità di simulare l'esecuzione contemporanea di più programmi;
- la *multiutenza*, cioè l'utilizzo simultaneo del calcolatore da parte di più utilizzatori, normalmente connessi al calcolatore attraverso più terminali a esso collegati.
- la *memoria virtuale*, che consente di eseguire programmi in cui il numero complessivo delle istruzioni e dei dati è maggiore della capacità di memoria del calcolatore.

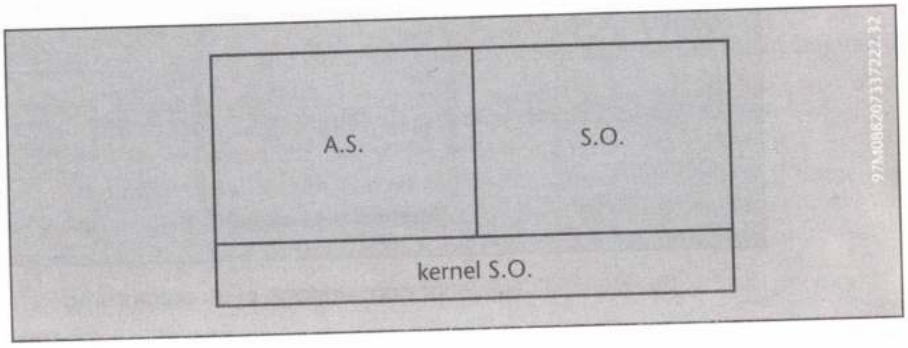


Fig. 2.32: il software di base: S.O. è il sistema operativo; A.S. indica genericamente l'altro software low level.

2.3.2 Traduttori di linguaggi di programmazione

Lo scopo di un linguaggio di programmazione è la descrizione semplice e sintetica di un algoritmo, cioè dei dati e delle sequenze di operazioni su di essi. Un programma (cioè un algoritmo descritto in un linguaggio di programmazione esistente) è un testo che descrive processi computazionali.

Un linguaggio di programmazione è un insieme di convenzioni formali per scrivere un programma. *Un linguaggio di programmazione esiste se esiste un suo programma traduttore.*

Un traduttore del linguaggio L per il calcolatore C è un programma che trasforma (*traduce*) un programma scritto nel linguaggio L, detto programma *sorgente*, in un programma *equivalente* scritto nel linguaggio macchina del calcolatore C, detto *codice oggetto*.

Un programma scritto in un linguaggio di programmazione deve essere tradotto prima di essere eseguito dalla machina fisica[31].

Il processo di traduzione può essere effettuato in due modi diversi che permettono di distinguere due categorie di programmi traduttori: il *compilatore* e l'*interprete*.

a) compilatore

Il compilatore è un programma che ha come dato di input il (testo del) programma sorgente (nel linguaggio di programmazione) e come dato di output il corrispondente programma in linguaggio macchina.

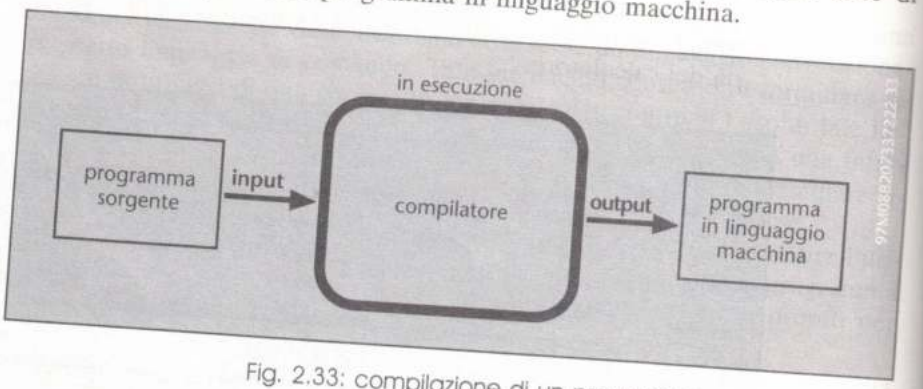


Fig. 2.33: compilazione di un programma

[31] Un linguaggio è costituito da un insieme di simboli (il *lessico* o *vocabolario*) e da un insieme di regole (la *sintassi*) che permettono di combinare sequenze di simboli in *linee* e sequenze di linee in un programma. In un linguaggio naturale (l'italiano, l'inglese, l'arabo,..) i simboli sono le parole e la sintassi indica il modo per creare con le parole una frase. Quindi un programma in un linguaggio di programmazione è analogo a una sequenza di frasi sintatticamente corrette in un linguaggio naturale.

L'esecuzione del compilatore determina quindi la creazione di un programma in linguaggio macchina che è equivalente al programma scritto nel linguaggio di programmazione.

A questo punto il programma in linguaggio macchina può essere eseguito (con i suoi dati di input). Tale programma può essere eseguito quante volte si vuole senza dover più ripetere la traduzione.

Nel caso della compilazione la fase di traduzione e la fase di esecuzione del programma sono distinte.

La compilazione è una trasformazione del programma sorgente che avviene usualmente in quattro fasi.

Nella prima fase, detta analisi lessicale, vengono individuati e classificati i vari simboli che compaiono nel programma (costanti, variabili, operatori, costrutti di controllo, etc.). In un linguaggio naturale questa fase corrisponde a classificare le parole di una frase in articoli, sostantivi, aggettivi, verbi, etc. L'analisi sintattica controlla se i simboli sono correttamente combinati in strutture sintattiche (per esempio espressioni aritmetiche). In un linguaggio naturale questa fase corrisponde a controllare la correttezza di una frase (la cosiddetta sintassi del periodo). Nella fase di generazione del codice per ogni istruzione (in genere una istruzione è scritta in una linea) del programma viene generata una opportuna sequenza di istruzioni del linguaggio macchina (e/o riservato spazio di memoria per i dati) e l'insieme di tali sequenze viene assemblato in una unica sequenza che costituisce il programma in linguaggio macchina. Lo scopo dell'ultima fase, quella di ottimizzazione, è l'eliminazione di eventuali ridondanze nel programma in linguaggio macchina.

Eseguire un programma in un linguaggio di programmazione compilato significa:

- eseguire il programma compilatore per ottenere il programma in linguaggio macchina;
- eseguire il programma in linguaggio macchina.

In genere, al *prodotto* dell'intera fase di compilazione si dà il nome di *programma oggetto.*

A una frase sintatticamente corretta di un linguaggio naturale può o meno essere associato un significato (*semantica*): per esempio la frase sintatticamente corretta in italiano *il gatto beve il quaderno* è semanticamente scorretta.

La semantica di un programma è un concetto alquanto elusivo; in definitiva è quello che il programma calcola quando viene eseguito. In verità, quando si parla genericamente di *programma corretto* si intende se il programma, quando viene eseguito, calcola il risultato che lo sviluppatore si aspetta; in questo senso *il significato di un programma sta nella mente di chi lo ha sviluppato.*

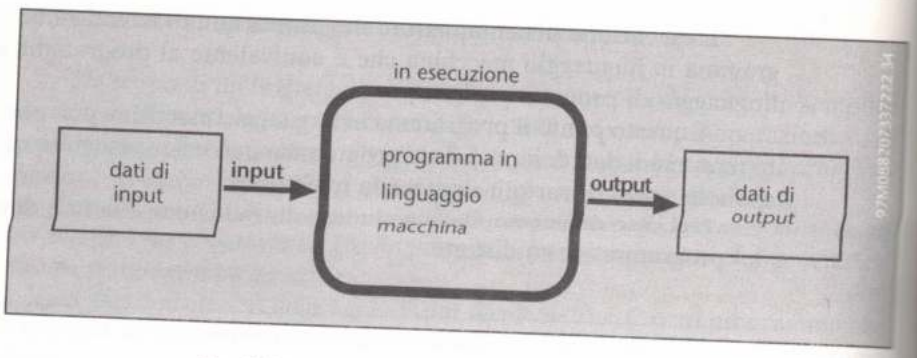


Fig. 2.34: esecuzione di un programma compilato

Come vedremo nel seguito, un programma può essere articolato in più moduli, ognuno dei quali si riferisce a una parte dell'algoritmo complessivo. Inoltre un programma può fare riferimento a moduli già esistenti, che fanno parte delle risorse software. La compilazione di ogni modulo dà luogo a un

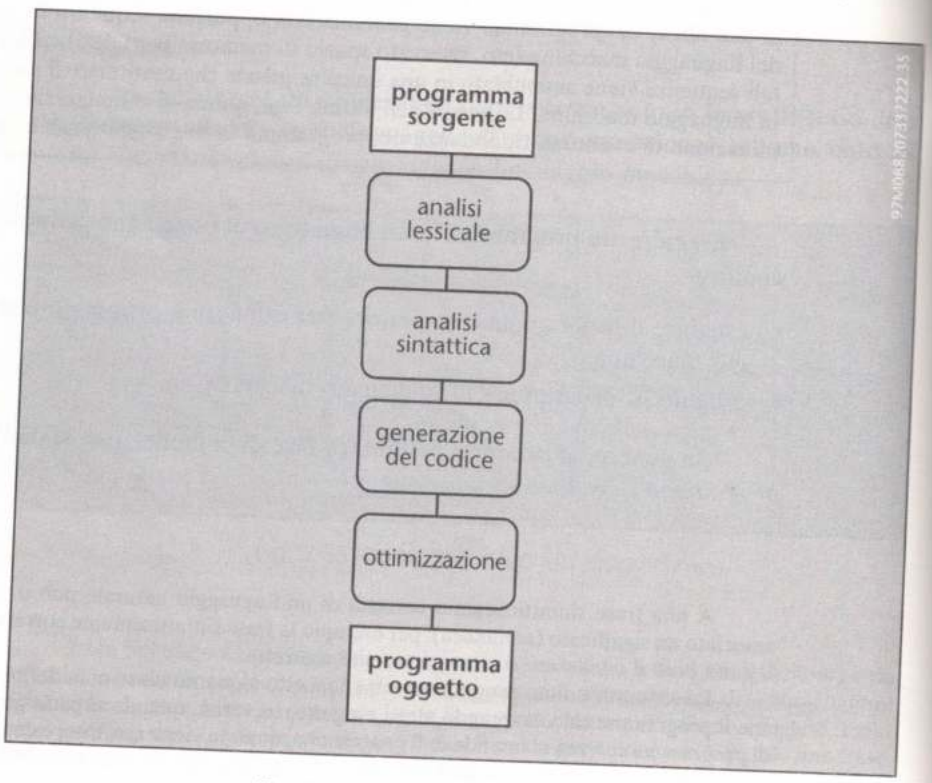


Fig. 2.35: sequenza di compilazione

distinto programma oggetto. L'opportuno assemblaggio, fatto da un programma del software di base detto *linker* (letteralmente *collegatore*), di tali programmi oggetto costruisce il programma in linguaggio macchina, eseguibile dalla macchina fisica (generalmente detto *assoluto* o *eseguibile*).

b) interprete

L'interprete non crea un programma in linguaggio macchina complessivamente equivalente al programma sorgente.

Le fasi di traduzione ed esecuzione non sono distinte, o meglio non sono distinte a livello di programma complessivo, pur se lo sono a livello della singola istruzione del programma sorgente.

Eseguire un programma in un linguaggio di programmazione interpretato significa:

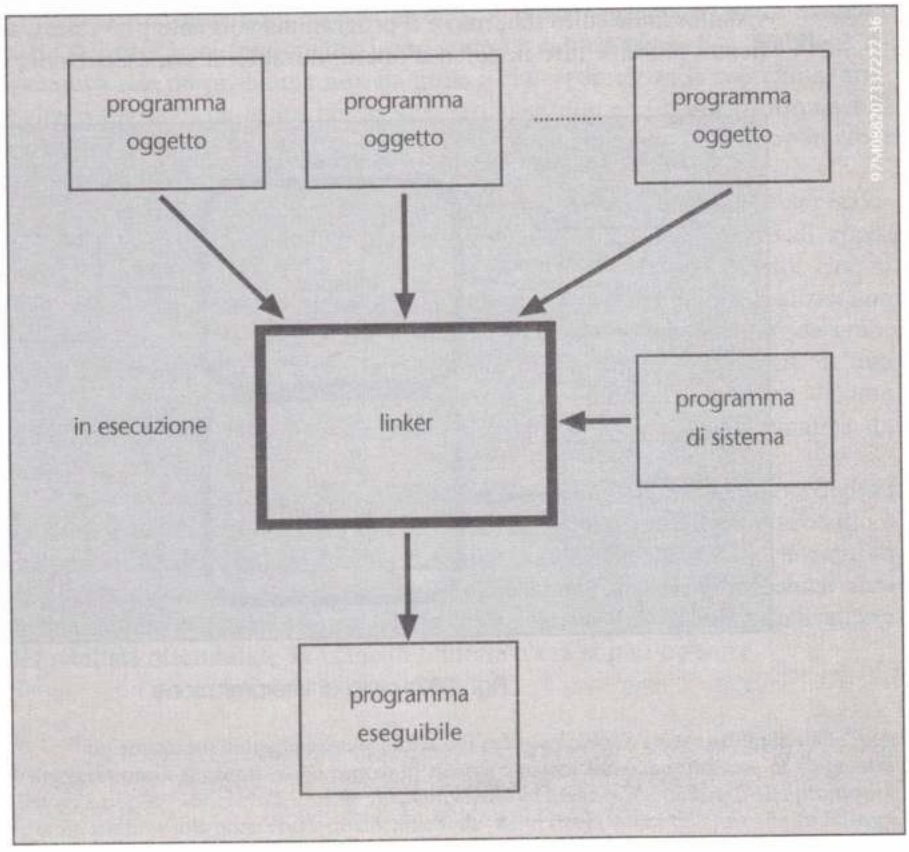


Fig. 2.36: fase di linking

- eseguire il programma interprete, fornendogli come dato di input il (testo del) programma sorgente.

L'interprete ripete le seguenti quattro azioni:

- considera una linea del programma sorgente;
- traduce la linea (analisi lessicale, analisi sintattica e generazione del codice *per la singola linea*) in una opportuna sequenza di istruzioni in linguaggio macchina;
- fa eseguire tale sequenza di istruzioni;
- conserva opportunamente i dati di input e i dati generati durante l'esecuzione.

Se è necessario eseguire più volte lo stesso programma si deve ripetere ogni volta il processo di interpretazione.

Il principale vantaggio dell'interprete è di consentire l'*interattività* tra chi programma e i processi computazionali descritti dal programma, in quanto il risultato dell'esecuzione di una singola istruzione può essere immediatamente visualizzabile sullo schermo e il programma sorgente può essere anche modificato, per così dire in corso d'opera, durante la sua *esecuzione*.

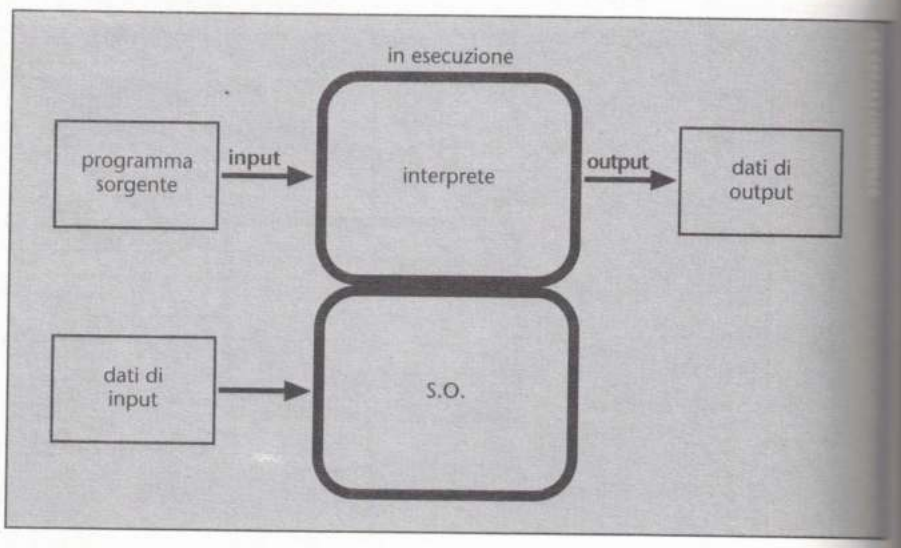


Fig. 2.37: ciclo di interpretazione

2.4 Approfondimenti

2.4.1 Cenni storici[32]

Charles Babbage fu il primo che intuì e cercò di progettare una macchina per eseguire programmi.

I contributi fondamentali di Babbage furono due. Egli comprese che la reale importanza di un calcolatore automatico risiede nella possibilità di eseguire ripetitivamente una data sequenza di istruzioni, fissando a priori il numero delle ripetizioni oppure rendendolo dipendente dai risultati della computazione in atto. Inoltre, contrariamente a quanto affermavano i più illustri scienziati dell'epoca, come Lord Kelvin e J.C. Maxwell, Babbage sostenne che la macchina poteva essere costruita imitando il modo in cui l'uomo rappresenta i numeri, cioè attraverso una sequenza di cifre, ed esegue i calcoli, cioè contando e utilizzando le quattro operazioni aritmetiche (cioè, sempre contando!). Un simile approccio è detto digitale, dall'operazione digitale per eccellenza, cioè contare con le dita. A chi gli ribatteva che la maggior parte dei problemi (matematici) non sono di natura aritmetica, Babbage rispondeva che un problema non ha nulla a che vedere con la macchina, analogica o digitale, da usare per la sua risoluzione, e che basta riformulare il problema in modo da ridurlo in una forma risolvibile con le sole operazioni aritmetiche.

L'approccio alternativo è quello analogico. Esso si basa sull'osservazione che è possibile descrivere matematicamente il funzionamento di apparecchiature complesse; l'idea è allora di invertire questo processo, cioè si fissa una formula matematica (che rappresenta il problema) e si realizza una macchina che è descritta da tale formula. I dati sono rappresentati da grandezze fisiche continue, come la rotazione di un disco, la tensione di una molla o l'intensità di una corrente elettrica, e vengono elaborati da una macchina che ha un comportamento *analogo* alla formula matematica da calcolare.

La storia ha dato ragione a Babbage mostrando che i calcolatori digitali possono eseguire centinaia di milioni di operazioni aritmetiche al secondo e che praticamente tutti i problemi possono essere opportunamente riformulati in modo da essere risolvibili da una tale macchina; invece, le macchine analogiche hanno rilevanti limiti come la scarsa flessibilità, la bassa accuratezza dei risultati ottenibili, e la velocità limitata a cui si può operare.

[32] Informazioni dettagliate sulla storia del calcolo possono essere ottenute dalla rete. Si suggeriscono i seguenti siti: il centro di ricerca Charles Babbage Institute of Computer History `http://www.cbi.umn.edu/`, University of Minnesota, dedicato alla promozione dello studio e alla preservazione documentale della storia del calcolatore; il sito History of Computing `http://ei.cs.vt.edu/~history/index.html`, Virginia Tech, con un museo virtuale del calcolo.

Il Difference Engine (letteralmente *il motore per calcolare le differenze*: il motivo per cui il calcolo delle differenze è importante in moltissime applicazioni scientifiche è accennato nel paragrafo 3.5) fu presentato all'Esposizione Internazionale del 1862; una sua versione realizzata dal tipografo svedese P.G. Scheutz fu commercializzata e venduta a vari centri di ricerca in astronomia e a una compagnia di assicurazioni. L'Analytical Engine fu progettato completamente ma non realizzato da Babbage; recentemente è stata realizzata, ed esposta in un museo, una versione di tale macchina a partire dal progetto originario.

Citando Babbage "*L'Analytical Engine consiste di due parti: il magazzino (store) in cui sono piazzate tutte le variabili su cui si opera, così come tutte quelle quantità create come risultato di tali operazioni; la macina (mill) in cui sono sempre portate le quantità quando si opera*". L'Analytical Engine lavorava su formule contenenti numeri e variabili simboliche ed era in grado di valutare numericamente una formula quando a tali variabili venivano assegnati valori numerici. Essa consentiva di superare i confini dell'aritmetica: l'Analytical Engine non è solo una macchina calcolatrice. Una assistente di Babbage, Lady Ada Byron Lovelace, sviluppò e pubblicò alcuni programmi per l'Analytical Engine; è stata la prima programmatrice della storia.

Le macchine di Babbage venivano azionate a mano o a vapore.

Nel 1890 H. Hollerith realizzò una macchina elettrica per selezionare e contare le schede perforate utilizzate per il censimento della popolazione degli USA (63 milioni di persone/schede); egli fondò la Tabulating Machine Company, che divenne nel 1924 la International Business Machine (IBM).

Nel 1893 lo spagnolo T. Quevedo suggerì di utilizzare la tecnologia elettrica per realizzare le macchine di Babbage.

Nel 1934 il matematico Vannevar Bush, seguendo le idee di Kelvin, realizzò il primo calcolatore analogico di tipo meccanico, l'Analizzatore differenziale.

Nel 1940 G. Stibitz, della Bell Telephone, costruì un calcolatore elettrico che eseguiva programmi scritti su un nastro perforato; i dati venivano introdotti utilizzando schede perforate o attraverso un pannello con interruttori. Questa macchina (il Model I Relay Calculator) era costituita da 3000 relè e da 500 chilometri di fili elettrici.

In Germania dal 1940 al 1943 K. Zuse realizzò vari calcolatori elettromeccanici per scopi militari.

Nel 1942 J.V. Atanasoff realizzò il primo prototipo di calcolatore a tecnologia elettronica.

Nel 1943 il matematico inglese A.J. Turing sviluppò il calcolatore/decifratore Colossus che decifrava i messaggi in codice del comando tedesco durante la seconda guerra mondiale.

Nel 1943 il matematico americano H. Aiken progettò e realizzò con l'aiuto dell' IBM l'Automatic Sequence Controlled Calculator Mark 1, in grado di eseguire una sequenza di operazioni scritta su un nastro perforato.

La macchina era costituita da più di mezzo milione di componenti elettromeccaniche, era dotata di una memoria di 72 numeri e utilizzava la rappresentazione binaria. Era in grado di eseguire tre addizioni al secondo, una moltiplicazione in sei secondi e una divisione in sette.

Nel 1942 partì il progetto ENIAC (Electronic Numerical Integrator and Computer), con J.W. Mauchly, J. P. Eckert e H.H. Goldstine, che nel dicembre 1945 produsse il primo calcolatore elettronico digitale realmente operativo (lo è stato fino ai primi anni 50). Era costituito da 100.000 componenti elettronici (valvole, resistenze, condensatori, interruttori,...) con una memoria contente fino a 20 numeri. Era in grado di eseguire più di duemila addizioni al secondo, circa 500 moltiplicazioni al secondo e 50 divisioni al secondo. La programmazione era fatta a mano mediante connessioni volanti con cavi su pannelli. È curioso ricordare che gli addetti a queste operazioni venivano chiamati "computer", cioè il nome che oggi è dato al calcolatore stesso.

Si noti che in tutte queste macchine, il programma non veniva memorizzato nella memoria, che conteneva solo i dati su cui operare.

Nel 1945 con il progetto EDVAC, von Neumann realizzò la sua idea di calcolatore elettronico digitale in cui anche le istruzioni del programma, oltre ai suoi dati, sono immagazzinate in memoria. Nei due famosi articoli *First* (e *Second*) *Draft of a Report on the EDVAC, Electronic Discrete Variable Automatic Computer* egli descrisse l'architettura, l'organizzazione e l'impiantistica del suo calcolatore.

È interessante leggere la descrizione originale, dal *First Draft*:

"*Primo: poiché la macchina (device) deve fondamentalmente calcolare, essa deve eseguire le operazioni elementari dell'aritmetica. Queste sono addizione, sottrazione, moltiplicazione e divisione. È quindi ragionevole che essa contenga un modulo specializzato solo per queste operazioni. Deve quindi esistere una centrale aritmetica che costituisce il primo modulo specifico: CA.*

Secondo: il controllo logico della macchina, cioè l'opportuna cadenza sequenziale delle sue operazioni, può essere effettuata efficientemente da un modulo centrale di controllo. Poiché la macchina deve essere elastica, cioè il più possibile generale (all purpose), allora si deve fare una distinzione tra istruzioni specifiche, date per un particolare problema, e i moduli di controllo generale che controllano che tali istruzioni siano eseguite. Le prime devono essere memorizzate e le seconde sono rappresentate da ben definite parti operative della macchina. Per controllo centrale si intende solo quest'ultima funzione, e il modulo che la realizza è CC.

Terzo: qualunque macchina che deve eseguire lunghe e complicate sequenze di operazioni deve avere una considerevole memoria. La memoria costituisce il terzo modulo specifico della macchina: M.

I tre moduli specifici CA, CC (che chiameremo insieme C) e M corrispondono ai neuroni associativi del sistema nervoso umano. ... La macchina deve

inoltre avere la capacità di mantenere un contatto di ingresso/uscita con uno strumento esterno per conservare i dati: R.

Quarto: la macchina deve avere componenti per trasferire informazioni da R nei moduli C e M. Queste componenti sono i moduli di input: I. Si vedrà che è più conveniente fare tutti i trasferimenti da R (attraverso I) in M e mai direttamente in C. Quinto: la macchina deve avere una componente per trasferire da C e M in R. Queste componenti sono i moduli di output: O. Si vedrà che è più conveniente fare tutti i trasferimenti da M (attraverso O) in R e mai direttamente da C".

2.4.2 Approfondimenti di architettura

2.4.2.1 *Canali di comunicazione*

Un canale di comunicazione (detto anche *bus*) è un collegamento fisico che rende possibile il trasferimento di informazioni tra due componenti. Invece di connettere ogni coppia di componenti mediante un canale dedicato, si preferisce utilizzare un unico canale (*bus di sistema*) a cui sono connesse permanentemente tutte le componenti: un meccanismo di commutazione (*switch*) rende possibile ad un certo istante di tempo la comunicazione selettiva solo tra una coppia di componenti.

All'interno di ogni componente le comunicazioni sono realizzate sempre mediante bus specializzati (bus interni).

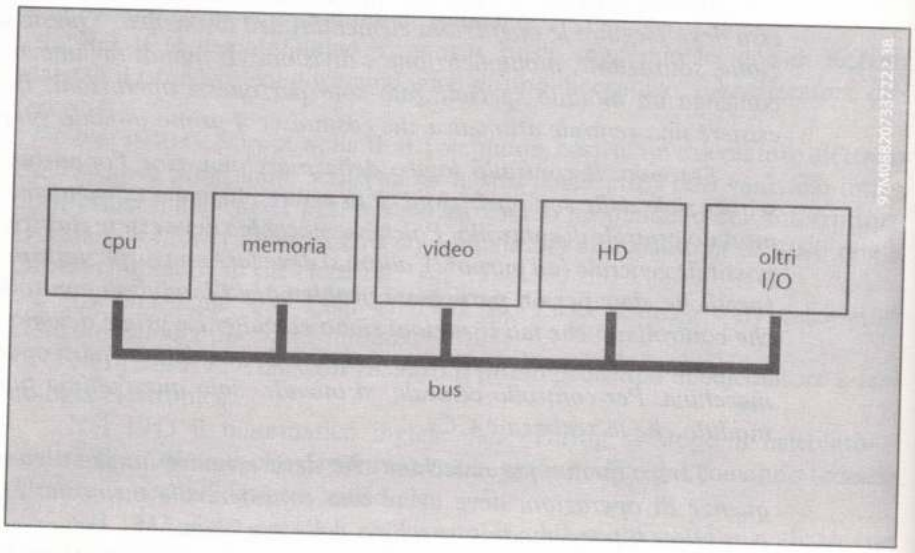


Fig. 2.38: schema di connessione mediante bus

I vari dispositivi di I/O sono connessi al bus di sistema attraverso particolari circuiti, detti *controllori*, che hanno il compito di gestire l'accesso al bus.

Lo schema precedente si riferisce all'architettura di un generico PC.

Da un punto di vista architetturale, la struttura di un PC è analoga a quella di calcolatori più potenti e costosi che vengono usualmente indicati, in ordine crescente, con i termini *workstation*, *mainframe* e *supercomputer*. La Fig. 2.39 mostra l'architettura di un mainframe[33].

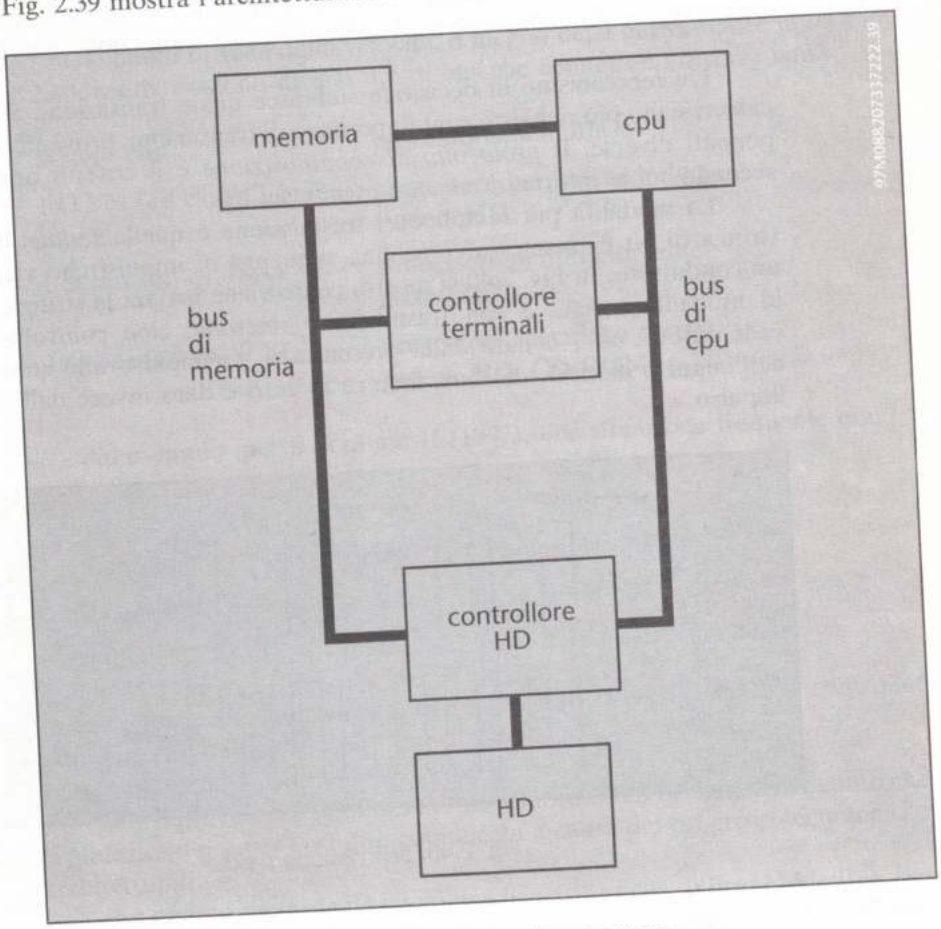


Fig. 2.39: schema di mainframe

[33] Si pensi che a metà degli anni 70 il mainframe Univac 1106 dell'Università di Napoli, che serviva l'intera comunità scientifica dell'Ateneo, aveva una memoria centrale (RAM) di 131K parole di 36 bit, una memoria di massa di 40 MB e una potenza complessiva ben inferiore a quella di un attuale PC.

La differenza principale risiede nella presenza di canali di comunicazione distinti per l'accesso alla memoria e per l'accesso ai sottosistemi di I/O. Il motivo di questa scelta progettuale è quello di ottenere prestazioni migliori.

L'utilizzo del bus per il trasferimento di informazioni tra due componenti prende il nome di *transazione*.

Una transazione prevede:

a) che vi sia una componente che invia e una che riceve l'informazione;
b) che il bus sia dedicato completamente a una transazione e non ad altre.

Un meccanismo di decisione stabilisce quale transazione attivare, nel caso vi siano più richieste contemporanee di transazioni, provenienti da componenti diverse. Il *protocollo di comunicazione* è il criterio organizzativo secondo cui le informazioni sono inviate sul bus.

La modalità più semplice di trasmissione è quella *seriale*, in cui una stringa di bit è trasmessa come una sequenza di impulsi che viaggiano su un conduttore. In Fig. 2.40 si mostra come viene inviata la stringa 10100111 in modalità seriale e con trasmissione *sincrona*, cioè controllata da un cadenzatore; un 1 è dato dalla presenza di un impulso allo stesso istante dell'impulso di clock del bus, mentre lo zero è dato invece dall'assenza di impulso.

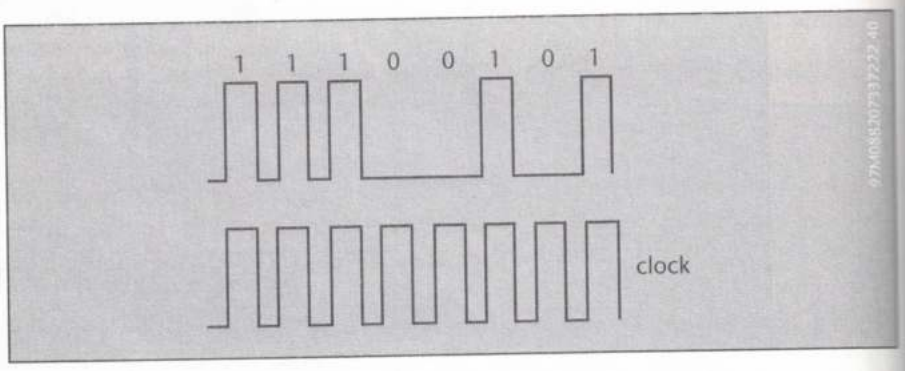


Fig. 2.40: trasmissione seriale

La velocità operativa del canale di comunicazione è detta l'*ampiezza di banda* (bandwidth) di un bus e viene misurata in *bit al secondo* (bps), oppure in byte al secondo (Bps); l'ampiezza di banda indica il numero di informazioni elementari trasferibili nell'unità di tempo.

Ovviamente questa velocità dipende dal clock e dal numero di linee utilizzate.

In un bus in genere le linee di trasmissione si possono classificare in 3

gruppi funzionali: si parla in genere di *data bus*, *address bus* e *control bus*. Il numero di linee dipende da vari fattori; ad esempio, per quanto riguarda il data bus, un processore Pentium ha 64 linee, il che significa che possono essere trasferiti dati di 64 bit alla volta, cioè con un'unica transazione (trasmissione *parallela*); supponendo che una transazione coinvolgente un bit richieda un tempo di 100 nsec, allora l'ampiezza di banda (bandwidth) è 64x10.000.000 bps = 640 Mbps; ancora, per un processore Pentium, l'address bus è di 32 linee, cioè ci sono circa 4 Giga di indirizzi possibili (precisamente 2^{32} = 4.294.967.296).

Dal punto di vista della velocità, o meglio della performance in termini di bandwidth, esistono diversi tipi di bus che connettono diverse unità

- system bus : processore/memoria
- I/O bus (ad es. PCI bus) : processore/periferiche veloci (video, HD, ...)
- I/O bus (ad es. ISA) : processore/periferiche lente (stampanti, ...)
- I/O bus (ad es. SCSI) : processore/specifiche periferiche (ad es. HD, CD-ROM, ...)

Ad esempio per il Pentium II (1997), una situazione frequente era:[34]

	clock	bandwidth
* system bus	(100 MHz)	800 MB/sec
* I/O bus (PCI)	(33 MHz)	132 MB/sec
* I/O bus (ISA)	(8 MHz)	8 MB/sec

anche se già esistevano altri tipi di bus dedicati, diversi protocolli, controller e adapter.

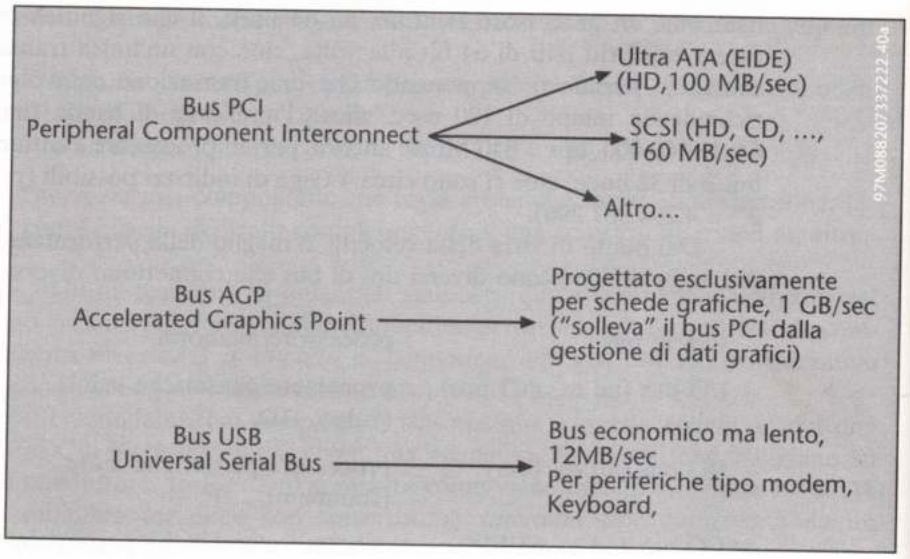
Con il più recente Pentium IV (novembre 2000) bisogna tener conto che la situazione si è per così dire complicata a causa dei progressi tecnologici e architetturali.

Per quanto riguarda il system bus (si parla anche di Front Side Bus, bus FSB) si è arrivati a 400 MHz (a 128 bit), con una bandwidth di 3.2 GB/sec.

[34] Per ulteriori letture e approfondimenti sull'architettura e sull'hardware di PC:

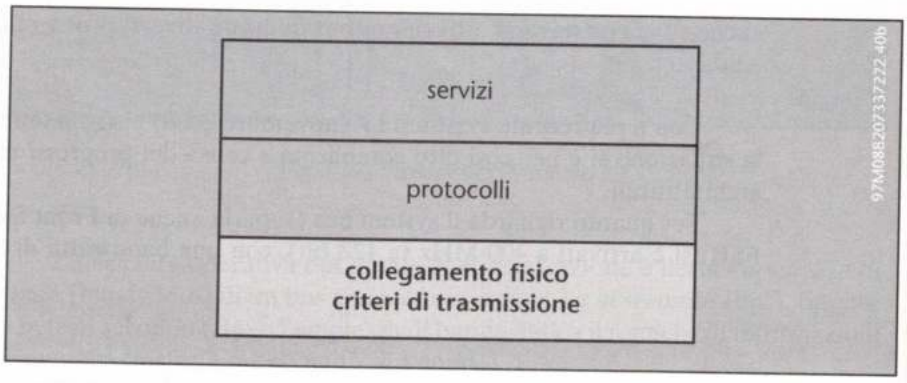
http://www.mkdata.dk/click
http://www.harris-lp.k12.ia.us/help/~jws/comp/PCInfo/PLATYPUS.HTM

Per quanto riguarda gli I/O bus, uno schema potrebbe essere:



2.4.2.2 *Sottosistemi di comunicazione in rete*

Una rete è un insieme di calcolatori indipendenti ma collegati tra loro, cioè in grado di scambiarsi informazioni. Una rete è realizzata da dispositivi per la trasmissione fisica, da dispositivi che collegano un calcolatore a tali mezzi di trasmissione, dal software che controlla il trasferimento e la corretta ricezione delle informazioni tra una coppia di calcolatori collegati e dal software che realizza i cosiddetti *servizi* di rete.



Il collegamento fisico avviene mediante supporti guidati (fili di conduttore, fibra ottica,..) oppure non guidati (microonde, laser,..).

Si noti che, come per il bus interno al calcolatore, anche nel caso delle reti le informazioni da trasferire sono stringhe di bit, normalmente suddivise in pacchetti (centinaia o migliaia di byte a seconda del protocollo di trasmissione) che vengono spediti in sequenza sul mezzo di trasmissione.

Se la trasmissione sul mezzo è analogica (come ancora parte della attuale rete telefonica) allora il calcolatore deve essere collegato al mezzo attraverso un dispositivo che trasforma l'informazione da digitale (come è all'interno del calcolatore) in analogica, quando il segnale è inviato, e viceversa che trasforma l'informazione da analogica in digitale, quando il segnale è ricevuto. Un tale dispositivo è detto *modem*.

Se la trasmissione sul mezzo è digitale il dispositivo realizza solo una connessione passiva.

Le reti di calcolatori si distinguono in *locali* e *geografiche*.

Una rete locale (Local Area Network, LAN) collega fino a varie centinaia di calcolatori che si trovano fisicamente dislocati in una piccola area, come un edificio o un campus universitario. Spesso una Lan è realizzata con un unico cavo (doppino telefonico, cavo coassiale, fibra ottica,...) al quale sono collegati i calcolatori (ma anche stampanti, plotter,..).

Il canale di trasmissione è condiviso tra tutti i calcolatori collegati, anche se spesso la rete è opportunamente suddivisa in sottoreti, mediante dispositivi specifici, al fine di evitare che malfunzionamenti in una parte della rete condizionino il corretto funzionamento dell'intera rete. Un classico esempio è la rete locale Ethernet che ha una ampiezza di banda di 10 Mbps oppure di 100 Mbps (Fast Ethernet).

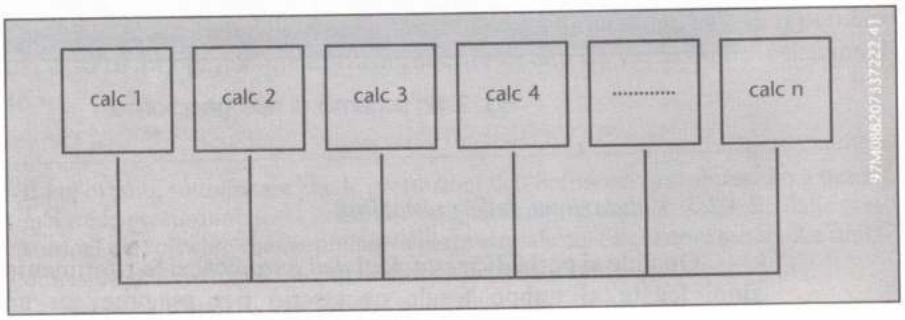


Fig. 2.41: schema di LAN

L'hardware di una rete è responsabile della trasmissione fisica di stringhe di bit, del controllo degli errori di trasmissione, della scomposizione in pacchetti, della regolazione del traffico dei pacchetti, della gestione delle interferenze,...

In genere ogni Lan è collegata ad altre Lan in modo da costituire una rete geografica su scala cittadina; e così via, più reti geografiche su scala cittadina opportunamente connesse danno luogo a una rete geografica su scala regionale o nazionale; la rete costituita da tutte le reti è detta *inter-rete* (*INTERNET*).

Una rete geografica è costituita, oltre che dalle varie Lan o sottoreti (Fig. 2.42), dalle linee di trasmissione (*canali* o *dorsali*) che realizzano la connessione fisica e da un insieme di calcolatori che consentono di collegare tra loro due o più linee di trasmissione (calcolatori di *commutazione* o *router*).

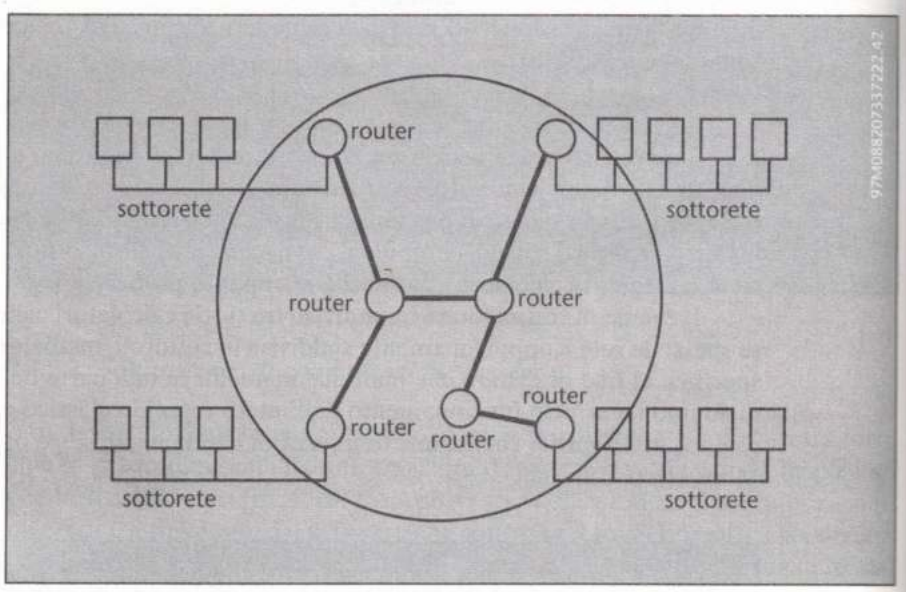


Fig. 2.42: schema di rete geografica

2.4.2.3 *Valutazione delle prestazioni*

Quando si parla di *prestazioni dell'hardware* si fa riferimento a informazioni legate al tempo totale necessario per eseguire, su uno specifico hardware, un generico programma, cioè un programma che sia sufficientemente rappresentativo di una vasta classe di programmi per risolvere problemi concreti.

Il processo di esecuzione di un generico programma coinvolge l'uso di tutte le componenti dell'hardware.

Da questo punto di vista, *l'hardware può essere considerato un insieme di risorse che vengono utilizzate da un programma in esecuzione*. In particolare, l'esecuzione di un programma richiede:

- l'uso per un certo tempo del processore;
- l'uso di una certa quantità di voci di memoria per immagazzinare dati e istruzioni del programma stesso;
- l'uso per un certo tempo del/dei canale/i di comunicazione.

È consuetudine denotare le risorse hardware con *tempo di processore*, *spazio di memoria* e *tempo di comunicazione*.

La struttura del programma determina quanto uso di tale risorse è necessario per l'esecuzione del programma stesso; la minimizzazione dell'uso delle risorse è un aspetto fondamentale del progetto di un algoritmo per la risoluzione di un problema. Se si considerano due programmi equivalenti e si eseguono su due diversi hardware, si ottengono in generale due tempi totali di esecuzione differenti. Tale differenza è dovuta alla diversità sia nell'architettura, sia nell'organizzazione (per es. tempo di accesso alla memoria, ampiezza di banda dei bus,...). Le prestazioni dipendono dalla struttura dell'intero hardware e quindi aumentare la velocità operativa di una singola componente può non avere influenze significative sulle prestazioni complessive. Per esempio, raddoppiare il solo tempo di clock del processore non determina il raddoppio delle prestazioni. D'altra parte, il solo tempo di clock non è una informazione sufficiente a caratterizzare le effettive prestazioni dell'hardware.

Per esempio, l'esecuzione di una operazione logico/aritmetica richiede vari accessi alla memoria (fase di fetch, caricamento dei dati nei registri del processore), e quindi sia il tempo di accesso alla memoria sia l'ampiezza di banda del bus che collega memoria e processore concorrono a determinare il tempo di esecuzione dell'operazione. Una considerazione analoga si può fare nel caso di un'operazione di trasferimento di dati da/verso un sottosistema di I/O.

È importante sottolineare che le prestazioni dell'hardware contribuiscono a determinare le prestazioni del calcolatore, ma queste ultime dipendono anche dalle prestazioni del software che costituisce la parte virtuale del calcolatore stesso. Le unità di misura della velocità operativa di un calcolatore sono:
- MIPS, milioni di operazioni (generiche) al secondo;
- Megaflops, milioni di operazioni floating-point al secondo.

Una informazione importante, che dovrebbe essere sempre associata a quella del tempo di clock, è il numero di periodi clock necessario per eseguire una operazione. In generale, operazioni diverse richiedono un diverso numero di periodi di clock. Se si denota con k il numero medio di periodi di clock per istruzione, con c il periodo di clock e con n il numero totale di operazioni

eseguite da un programma, si ottiene che il tempo totale di esecuzione t è dato da:

$$t = k \cdot c \cdot n \cdot$$

Analizzando i singoli contributi al tempo totale di esecuzione, si vede che

- n dipende dal programma, ma anche dalla natura delle istruzioni eseguibili dal processore;
- c è una caratteristica organizzativa propria del processore (dipende dal tipo di circuiti utilizzati, dalla tecnologia dei semiconduttori,..);
- k è l'informazione che tiene conto delle prestazioni complessive dell'hardware.

Alcune scelte architetturali consentono di diminuire il valore di k e quindi di ottenere hardware con prestazioni più elevate, a parità di n e c.

Esse sono:

- il *pipeline nel processore*, che diminuisce il numero di periodi di clock per istruzione, per tutto l'insieme delle operazioni eseguibili;
- la *struttura gerarchica della memoria*, che ha effetti soprattutto sulle operazioni che accedono alla memoria (ma anche sulla fase di fetch di tutte le istruzioni);
- l'*accesso diretto alla memoria* e, in generale, la *distribuzione della capacità elaborativa*, che ha effetti sulle operazioni di trasferimento di dati tra memoria e sottosistemi di I/O.

a) pipeline nel processore

Il pipeline è una soluzione architetturale che consente di introdurre il parallelismo nell'esecuzione delle fasi di fetch-decode-execute, durante l'esecuzione di un programma. L'idea è molto semplice: si supponga che il processore debba eseguire consecutivamente m istruzioni, che chiameremo istruzione A, B , C, ... La prima azione da effettuare è il fetch dell'istruzione A. La successiva azione è il decode dell'istruzione A; ma in contemporanea, poiché la fase si fetch di A è terminata, si può iniziare la fase di fetch dell'istruzione B. Questo ovviamente richiede che il processore abbia un'architettura modulare (modulo per l'azione di fetch, modulo per l'azione di decode e modulo per l'azione di execute), e che ciascun modulo possa agire simultaneamente (cioè in parallelo) agli altri due.

In definitiva, la situazione può essere schematizzata come in Fig. 2.43: Nell'ipotesi che l'esecuzione di ciascuna fase richieda un periodo di clock, si ha che l'esecuzione di m istruzioni richiede complessivamente m+2 periodi di clock. Quindi il processore è in grado di eseguire una istruzione per ogni periodo di clock, se si esclude il tempo iniziale necessario per *caricare* il pipeline. Un

pipeline di questo tipo è a *tre stadi* (i tre moduli distinti). Si osservi che il parallelismo introdotto dal pipeline non altera la sequenzialità dell'esecuzione delle istruzioni, nel senso che l'esecuzione dell'istruzione A termina prima dell'esecuzione dell'istruzione B, quella di B prima di C e così via. In questo senso il pipeline introduce il parallelismo nell'esecuzione di ogni singola istruzione.

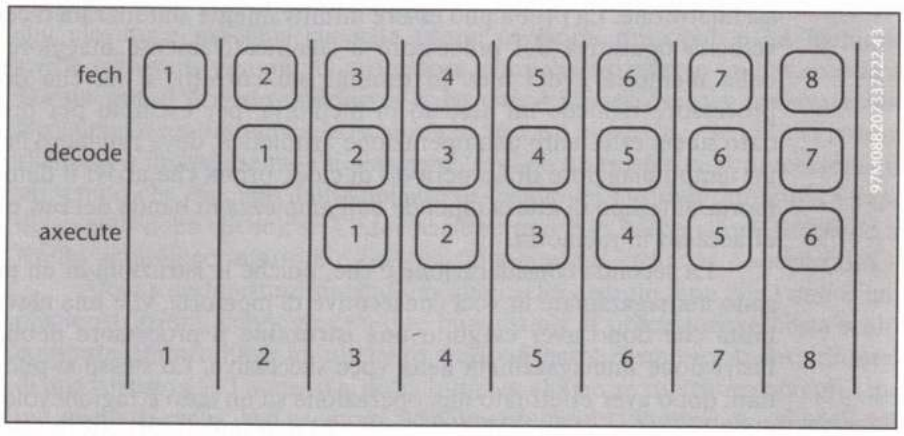


Fig. 2.43: pipeline a 3 stadi

Il numero di stadi può essere maggiore di tre, come è mostrato nella figura seguente.

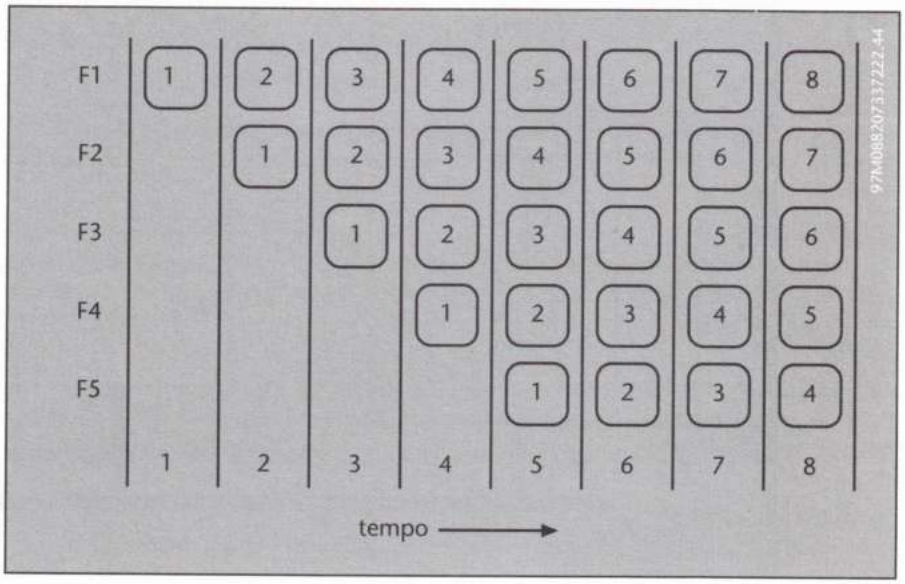


Fig.2.44: pipeline a 5 stadi

L'idea di introdurre questo tipo di simultaneità nel funzionamento del processore risale ancora a Von Neumann.

b) struttura gerarchica della memoria

Vi sono due considerazioni preliminari che giustificano questa soluzione architetturale. La prima può essere intuitivamente sintetizzata dicendo che la velocità operativa del processore è significativamente maggiore di quella della memoria e del bus. In termini più corretti, si ha che dopo che il processore richiede un accesso in memoria, per esempio per prelevare un dato su cui effettuare una operazione aritmetica, deve rimanere in attesa per un tempo maggiore di un periodo di clock prima che arrivi il dato dalla memoria. Il tempo di attesa dipende dall'ampiezza di banda del bus e dal tempo di accesso in memoria.

La seconda considerazione è che, poiché le istruzioni di un programma sono immagazzinate in voci consecutive di memoria, vi è una elevata probabilità che dopo aver eseguito una istruzione il processore debba eseguire l'istruzione immagazzinata nella voce successiva. Lo stesso si può dire per i dati: dopo aver effettuato una operazione su un dato è ragionevole aspettarsi che le successive operazioni agiscano su dati immagazzinati in voci contigue. Questa considerazione è basata solo sul buon senso, ma vale in pratica per quasi tutti i programmi. Essa prende il nome di *principio di località* degli accessi alla memoria nell'esecuzione di un programma.

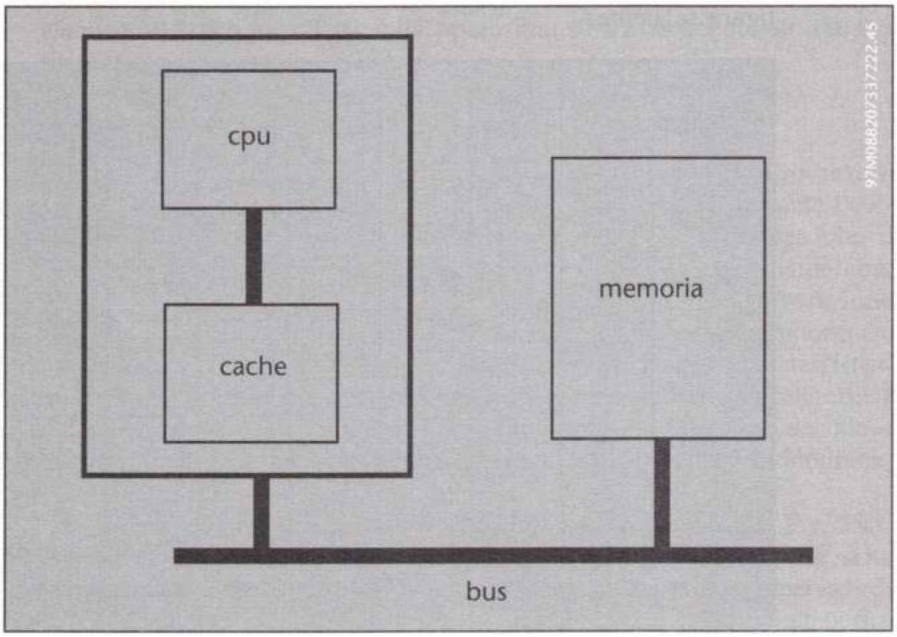


Fig. 2.45: schema di CPU con cache di primo livello

Queste due considerazioni suggeriscono di strutturare la memoria, piuttosto che come una unica componente, come un insieme di due (o più) memorie distinte, con diverse capacità e tempi di accesso. La memoria con tempo di accesso minore, che per ragioni puramente economiche ha generalmente una minore capacità, è detta *cache*, e rappresenta il livello gerarchico massimo; il livello più basso è costituito dalla memoria con tempo di accesso più elevato e massima capacità (detta memoria principale). La memoria cache si trova in genere nel processore, connessa attraverso un bus specifico. Da un punto di vista operativo, si ha che il processore per prelevare una istruzione o un dato invia la richiesta di accesso alla cache. Se l'informazione richiesta è immagazzinata nella cache, essa viene trasferita al processore (in un tempo che dipende dall'ampiezza di banda del bus dedicato e dal tempo di accesso della cache); se l'informazione non è in cache allora si accede al livello gerarchico inferiore e così via fin quando la richiesta è soddisfatta.

Alcune architetture prevedono due cache distinte, una per i dati e una per le istruzioni, connesse con due bus distinti al processore. Questa scelta consente di introdurre un ulteriore grado di parallelismo tra la fase di fetch di una istruzione e l'accesso ai dati. Infine in alcune architetture sono presenti più livelli di cache. Ad esempio, attualmente nei PC vi sono due o tre livelli

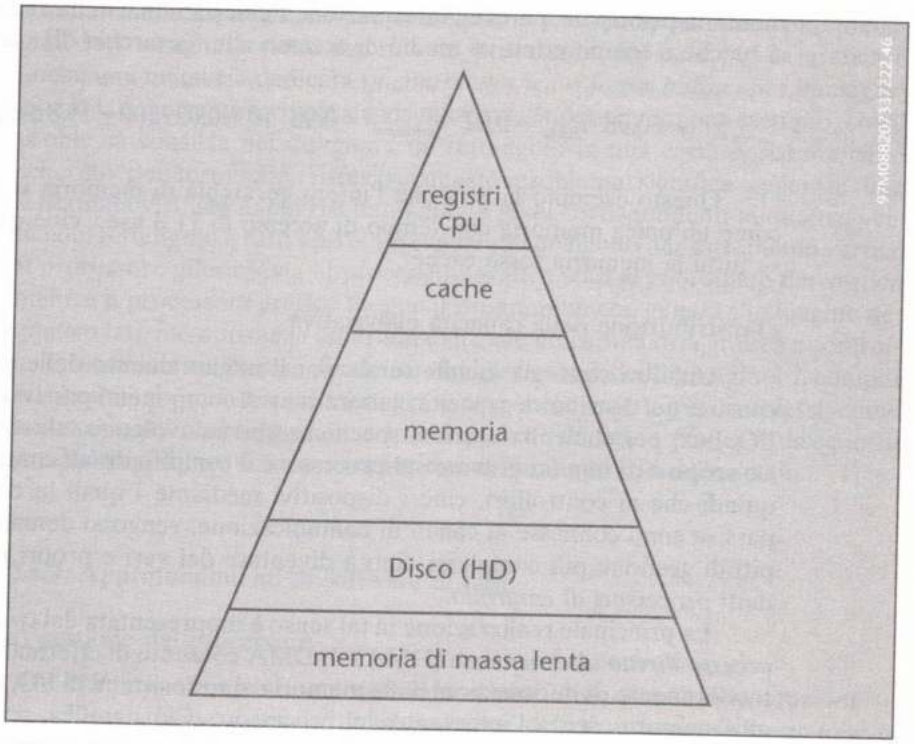


Fig. 2.46: livelli di memoria

di cache: la cache con tempo di accesso minore si trova nel processore, mentre la cache con tempo di accesso maggiore è parte della memoria. La capacità e la velocità dei livelli di cache sono fattori determinanti per la velocità operativa del calcolatore.

In questa visione gerarchica della memoria, è conveniente vedere anche la memoria secondaria (in genere i dischi) come parte logicamente integrante della gerarchia di memoria, ovviamente di livello minimo.

Il miglioramento delle prestazioni che si ottiene con l'introduzione di una gerarchia di memoria giustifica l'hardware aggiuntivo necessario per gestire la gerarchia e soprattutto i trasferimenti di insiemi di informazioni (detti *blocchi*) tra i livelli di memoria, necessari per rendere minima ad ogni istante la probabilità che una informazione non si trovi in cache e debba essere ricercata nei livelli inferiori.

L'esempio seguente fornisce un'idea del miglioramento delle prestazioni introdotto da una semplice strutturazione della memoria in due livelli (cache e memoria principale). Si supponga che il tempo di accesso alla cache sia 10 nsec e quello di accesso alla memoria principale sia 100 nsec. Inoltre si supponga che il numero di richieste di accesso alle informazioni che viene soddisfatto con il solo accesso alla cache sia il 98% del numero totale delle richieste; questo significa che solo nel 2% dei casi è necessario accedere alla memoria principale, perché l'informazione richiesta non è nella cache. Allora si ha che il tempo effettivo medio di accesso alla (gerarchia di) memoria è

$$t = 0.98 \cdot t_{cache} + 0.02 \cdot t_{principale} = 0.98 \cdot 10 + 0.02 \cdot 100 = 11.8\,\text{nsec}$$

Questo esempio mostra che l'intera gerarchia di memoria si comporta come un'unica memoria con tempo di accesso di 11.8 nsec, cioè quasi come se tutta la memoria fosse cache.

c) distribuzione della capacità elaborativa

Un'altra strategia architetturale per il miglioramento delle prestazioni consiste nel distribuire capacità elaborativa nei componenti passivi (memoria, I/O, bus) per controllare azioni specifiche che coinvolgono tali componenti. Lo scopo è di non far gravare sul processore il compito di tali controlli. Si ha quindi che ai controllori, cioè i dispositivi mediante i quali le componenti passive sono connesse ai canali di comunicazione, vengono demandati compiti di gestione più complessi, fino a diventare dei veri e propri processori, detti *processori di controllo*.

La principale realizzazione in tal senso è rappresentata dal *controllore di accesso diretto alla memoria* (DMA). Il DMA consente di effettuare azioni di trasferimento di informazioni dalla memoria ai sottosistemi di I/O, e da questi alla memoria, senza l'intervento del processore. Ciò significa, per esempio, che se durante l'esecuzione di un programma vi è una istruzione di trasferi-

mento di più dati dalla memoria secondaria alla memoria, allora il processore demanda la fase di esecuzione dell'istruzione al controllore DMA; questo localizza i dati sulla memoria secondaria, preleva i dati, li trasferisce attraverso il bus alla memoria e genera la sequenza di azioni di scrittura dei dati nelle voci di memoria. In alcuni casi il controllore DMA può essere in grado di eseguire programmi per svolgere azioni specializzate di controllo del funzionamento dei dispositivi di I/O e di organizzazione del trasferimento delle informazioni.

È importante sottolineare che, poiché il tempo di accesso alla memoria secondaria è molto maggiore sia del periodo di clock del processore sia del tempo di accesso alla memoria, durante il trasferimento anche di pochi dati il processore è in grado di eseguire un numero molto elevato (talvolta decine di migliaia) di istruzioni logico/aritmetiche. Il DMA consente quindi l'esecuzione in parallelo di istruzioni di trasferimento di I/O e di istruzioni logico/aritmetiche. La realizzazione di questo parallelismo rende necessaria l'introduzione di una sincronizzazione al fine di evitare, per esempio, l'esecuzione di istruzioni logico/aritmetiche su dati il cui trasferimento in memoria non sia stato ancora completato.

Un'altra importante realizzazione dell'idea di attribuire capacità di elaborazione anche ai sottosistemi di I/O, consiste nel dotare i terminali grafici con hardware specializzato per la creazione e la manipolazione delle immagini. Nel terminale viene inserito un processore dedicato per la grafica e anche una memoria dedicata (*memoria grafica* o *frame buffer*) per immagazzinare la schermata virtuale da visualizzare. Supponiamo, per esempio, che il problema consista nel disegnare un rettangolo in una certa posizione dello schermo del terminale; risolvere questo problema significa generare una schermata virtuale in cui risultano *attivi* i pixel corrispondenti ai quattro vertici del rettangolo e tutti i pixel sui segmenti che hanno per estremi tali vertici. Il processore allora invia al processore grafico solo le coordinate dei vertici, mentre il processore grafico esegue il programma che genera l'immagine dei quattro lati, memorizza la schermata virtuale nella memoria grafica e controlla il terminale nella fase finale di visualizzazione dell'immagine. I compiti attribuibili al processore dedicato possono anche essere più complessi, come per esempio l'ombreggiatura e la colorazione di figure piane, il disegno di parallelepipedi, rotazioni e traslazioni di immagini tridimensionali,...

2.4.3 Approfondimenti di software di base

a) gestione del processore

Un programma è una descrizione *statica* di sequenze di operazioni.

L'esecuzione di un programma viene detto *processo di esecuzione* o semplicemente *processo* (o *task*). Questa distinzione ha il pregio di separare

la descrizione di una sequenza di operazioni dalla fase dinamica della sua esecuzione. Si noti che un programma può dare luogo a più processi. Un processo termina dopo l'esecuzione dell'ultima istruzione del programma associato. Al contrario, se l'esecuzione di un programma viene temporaneamente sospesa, allora si dice che il processo associato è nello stato di *attesa*. Quando l'esecuzione del programma sospeso viene ripresa, allora si dice che il processo associato è nello stato *attivo*.

Su un unico processore l'esecuzione contemporanea di più programmi può essere solo simulata: nella terminologia appena introdotta ciò significa che vi può essere al più un processo attivo.

Il meccanismo di simulazione è il seguente. Si supponga di voler simulare l'esecuzione simultanea di tre programmi. A ciascun programma viene associato un processo. Tutti i processi sono posti nello stato di attesa; a questo punto il primo processo viene attivato per un intervallo di tempo prefissato (detto *fetta di tempo*). Al termine di tale intervallo di tempo (in genere dell'ordine del millisecondo), il processo viene posto in stato di attesa e viene attivato il secondo processo, che rimane attivo per una fetta di tempo; al termine, il secondo processo è posto in attesa e il terzo processo viene attivato per una fetta di tempo. L'alternanza di uso del processore da parte dei processi continua fino alla fine dei processi stessi.

Questa tecnica di gestione del (tempo di) processore è detta tecnica del *tempo ripartito* (*time sharing*).

Dal punto di vista dell'utilizzatore, l'effetto di tale strategia è che i programmi vengono eseguiti *simultaneamente* anche se il tempo di esecuzione di ciascun programma è in questo caso maggiore di quello che si avrebbe se il programma fosse eseguito *normalmente*. Il tempo totale richiesto per terminare un programma è detto *tempo totale*, mentre il tempo di effettivo uso del processore è detto *tempo di elaborazione*. Si noti che il tempo totale dipende anche dal numero di processi presenti; al contrario il tempo di elaborazione dipende solo dalla struttura del programma e dalle prestazioni dell'hardware.

La tecnica del time sharing risulta ancora più vantaggiosa quando un programma richiede l'esecuzione di operazioni di I/O; infatti una tale operazione richiede un tempo di esecuzione molto maggiore (almeno mille volte) rispetto all'esecuzione di una istruzione sulla memoria. Il processo viene bloccato per tutta la durata delle operazioni di I/O e non partecipa all'allocazione di tempo di processore. Quando le operazioni di I/O terminano, il processo viene posto in stato di attesa della sua fetta di tempo.

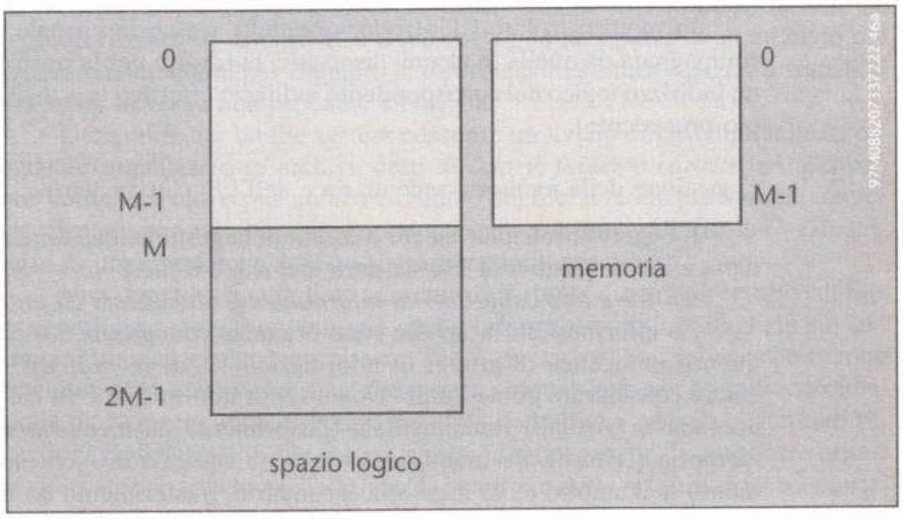
Questa situazione viene sintetizzata dicendo che i processi sono *concorrenti* perché competono tra loro per ottenere la risorsa tempo di processore. In generale l'allocazione del tempo di processore viene fatta con un algoritmo (detto algoritmo di *scheduling*) più complicato rispetto alla semplice attivazione a rotazione dei processi in attesa.

La strategia del time sharing viene realizzata attraverso programmi (processi) del s.o. (si noti che anch'essi competono per una fetta di tempo).

b) gestione della memoria

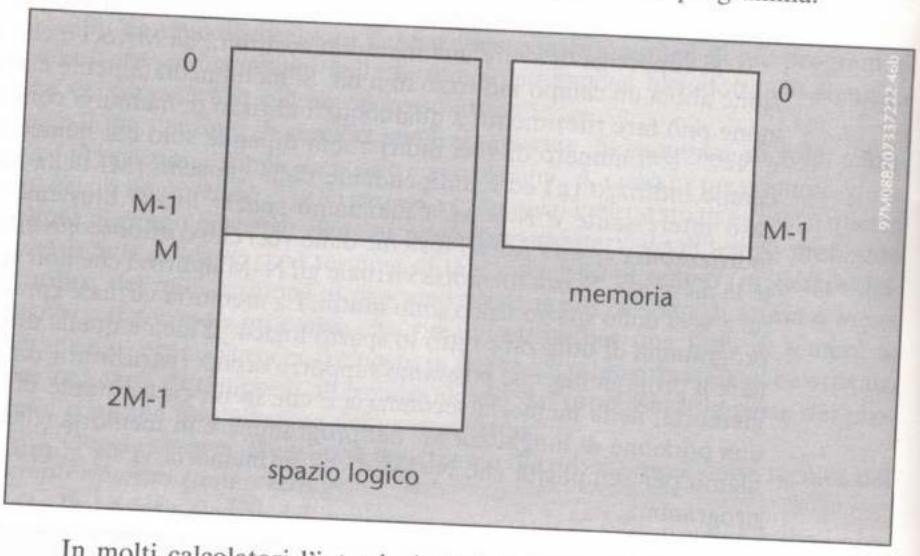
Per un utilizzatore, una gestione efficiente della memoria da parte della macchina virtuale significa la possibilità di avere programmi più grandi della memoria. In tal caso, il s.o. realizza una *memoria virtuale*, simulata cioè con tecniche software, la cui capacità non dipende dalla capacità della memoria (fisica). Quello che segue è solo un cenno all'idea di base della memoria virtuale, la cui implementazione effettiva è in genere molto più sofisticata.

Si supponga di avere una memoria costituita da M voci e che una istruzione abbia un campo indirizzo di n bit. Si ha immediatamente che una istruzione può fare riferimento a qualunque indirizzo di memoria compreso tra 0 e $N=2^n-1$; il numero di voci indirizzabili dipende solo dal numero di bit del campo indirizzo (n) ed è indipendente dalla capacità (M) della memoria. Il caso interessante è N > M. Chiamiamo *spazio logico* l'insieme delle voci indirizzabili e *spazio fisico* l'insieme delle voci che costituiscono effettivamente la memoria. Senza memoria virtuale gli N–M indirizzi che non appartengono a voci dello spazio fisico sono inutili. La memoria virtuale consente in un programma di utilizzare tutto lo spazio logico. L'idea è quella di immagazzinare il programma, che possiamo supporre occupi (istruzioni + dati) N voci di memoria, nella memoria secondaria e che in un certo istante di tempo solo una porzione di lunghezza M del programma è in memoria (fisica). Supponiamo per semplicità che N=2M e che in memoria vi sia la prima metà del programma.



Nell'esecuzione della prima porzione del programma un riferimento a un qualunque indirizzo logico x compreso tra 0 e M–1 non crea alcun problema; al contrario, se una istruzione coinvolge un indirizzo logico x compreso tra M e 2M–1, allora

- il contenuto della memoria è immagazzinato su disco (al posto della prima metà del programma);
- la seconda metà del programma è trasferita in memoria;
- viene calcolato l'indirizzo fisico y corrispondente all'indirizzo logico x; in questo caso y = x – M;
- a questo punto continua la normale esecuzione del programma.



In molti calcolatori l'introduzione della memoria virtuale è anche accompagnata da quella di alcuni dispositivi hardware per la trasformazione di un indirizzo logico nel corrispondente indirizzo effettivo (y = x – M nell'esempio precedente).

c) gestione della memoria secondaria e dell'I/O (file system)

L'oggetto principale su cui è costruita la gestione della memoria secondaria e dei dispositivi di I/O da parte del s.o. è il *file*.

Un file è una collezione di informazioni, identificata da un nome.

Le informazioni in un file sono organizzate logicamente come una sequenza omogenea di gruppi di informazioni, detti *record*. Un *record deve essere considerato come l'unità (minima) di informazione* su cui agiscono le operazioni (virtuali) fondamentali: trasferimento da memoria secondaria a memoria (lettura), da memoria a memoria secondaria (scrittura), trasferimento nell'ambito della memoria secondaria, trasferimento da memoria secondaria a un dispositivo di I/O e viceversa.

Oltre a coinvolgere un singolo record, le operazioni precedenti possono agire su più record o anche su tutto il file. Ognuna di queste operazioni è elementare per la macchina virtuale, nel senso che viene richiesta dall'utilizzatore al s.o. senza che l'utilizzatore conosca né i dettagli del funzionamen-

to dei vari dispositivi di memoria secondaria e di I/O, né il modo in cui le informazioni sono fisicamente immagazzinate in tali unità.

Un record può essere un oggetto semplice come un singolo carattere, un numero, una istruzione in linguaggio macchina, oppure può essere costituito da più dati elementari, per esempio tre sequenze di 10 caratteri ciascuna ('cognome', 'nome', 'indirizzo') e due numeri interi ('numero civico', 'numero di telefono').

Una operazione su un record di un file viene genericamente indicata con il termine *accesso* al file. Si può accedere a (i record di) un file in

- modo sequenziale, cioè accedendo successivamente a tutti i record;
- direttamente a un record, attraverso il suo indirizzo logico, cioè il numero d'ordine del record nella sequenza;
- attraverso una chiave, cioè il valore di una particolare informazione nel record.

L'insieme dei programmi del s.o. che hanno la funzione di gestire i file prende il nome di *file system*.

Il file system è una delle risorse virtuali del calcolatore più visibili all'utilizzatore e una delle più utilizzate.

L'utilizzatore vede l'insieme dei file come una organizzazione gerarchica, analoga a quella che usualmente si adopera in un ufficio per catalogare le informazioni in un archivio. In un ufficio un insieme di informazioni da considerarsi come una unità, per esempio le informazioni riguardanti un paziente di un ospedale, fa parte di un archivio che comprende le informazioni sui pazienti ricoverati; tale archivio a sua volta può fare parte di un archivio più generale contenente per esempio le informazioni su tutti i pazienti transitati nell'anno in corso nell'ospedale e così via.

Generalmente un file system consente un livello di visibilità delle informazioni organizzato in archivi, detti *directory*, *folder* o *cartelle*; un archivio può contenere uno o più archivi e/o uno o più file. Il modo più immediato per visualizzare una tale organizzazione gerarchica è quello di Fig. 2.47; tale criterio di organizzazione è detto (organizzazione ad) *albero*[35].

Una directory è quindi un contenitore virtuale di altre directory e di file; in genere, una directory contiene anche informazioni più dettagliate sul suo contenuto, cioè riguardanti ciascun file o directory. Tali informazioni sono ottenibili attraverso richieste al file system e riguardano per esempio, occupazione di memoria, data di creazione, data dell'ultimo accesso, eventuali restrizioni sugli accessi. Il file system consente infatti anche di introdurre criteri di riservatezza spesso necessari per la sicurezza delle informazioni memorizzate; tale aspetto è molto importante soprattutto nel caso dei grandi calcolatori e diventa critico nel caso di connessione in rete.

[35] L'introduzione di uno schema ad albero per classificare informazioni risale al filosofo francese Pierre de la Ramée (1515-1572).

Il criterio di disposizione logica dei file nel file system consente all'utilizzatore di organizzare le proprie informazioni, memorizzate nella memoria secondaria, nel modo che egli ritiene più appropriato e senza occuparsi dei dettagli della gestione sia della memorizzazione, sia degli accessi, sia dei trasferimenti dei file all'interno dei vari dispositivi di memoria secondaria.

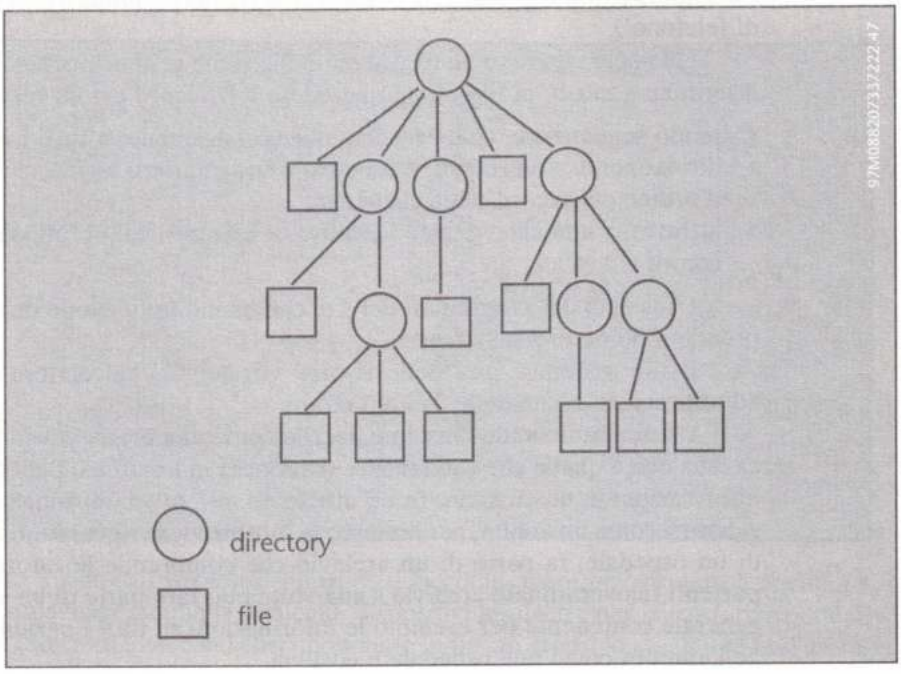


Fig.2.47: organizzazione gerarchica dei file nel file system

In questo contesto, anche i trasferimenti di informazioni tra memoria secondaria e i dispositivi di I/O (e viceversa) sono trattati dal file system in modo analogo ai trasferimenti all'interno della memoria secondaria. Un dispositivo di I/O è visto come una directory e allora, per esempio, la stampa di un file viene considerata semplicemente come il trasferimento del file nella directory corrispondente al dispositivo di stampa. Si noti che in tal modo tutto l'insieme dei dispositivi di I/O, e anche le reti a cui il calcolatore è eventualmente connesso, viene visto operativamente dall'utilizzatore come una estensione della memoria gerarchica del calcolatore.

È utile accennare alla rappresentazione fisica delle informazioni che costituiscono un file. Nell'ottica della visione per livelli della macchina virtuale, anche il file system ha una architettura per livelli; il livello superiore è quello visibile

all'utilizzatore, di cui si sono brevemente illustrate le funzionalità; il livello più basso è quello dei programmi che effettuano gli accessi fisici alle informazioni memorizzate. Questi programmi del file system trattano un file come una sequenza di byte, o di voci, indipendentemente dalla organizzazione logica (in record) del file. In particolare, allo scopo di avere uno sfruttamento ottimale dello spazio di memoria secondaria, la sequenza di byte che costituisce un file può non essere memorizzata fisicamente in modo continuo; in tal caso il file system deve essere in grado di ricostruire la corretta sequenza accedendo ai vari singoli tratti in cui la sequenza è stata suddivisa.
La trasmissione fisica delle informazioni di un file tra i vari componenti dell'hardware viene effettuata da alcuni programmi del file system; tali programmi usano un insieme di convenzioni per gestire le varie fasi della trasmissione fisica, detto protocollo. Generalmente, la sequenza di byte che costituisce il file da trasferire viene decomposta in blocchi, ognuno costituito da un numero prefissato di byte; un blocco da trasmettere viene memorizzato in un insieme di voci di memoria (detto buffer o memoria tampone) nel controllore del dispositivo, e quindi trasmesso attraverso un bus nel buffer del controllore del dispositivo ricevente, che provvede poi alla memorizzazione effettiva nel dispositivo. Il processo prosegue, blocco dopo blocco, fino all'esaurimento dei byte da trasferire. Normalmente l'utilizzatore ignora questi aspetti della memorizzazione e della trasmissione fisica.

d) interfaccia utente

L'interazione dell'utilizzatore con le funzionalità della macchina virtuale realizzata dal s.o. avviene attraverso la formulazione di *richieste* dell'utilizzatore, espresse in uno specifico linguaggio detto *linguaggio di comando* o linguaggio di *shell*. Il nome *shell* indica di solito il programma del s.o. che interpreta le richieste, cioè analizza una frase del linguaggio di comando, riconosce la funzionalità richiesta e trasmette la richiesta allo specifico programma del s.o. a cui è demandata la realizzazione di quella funzionalità.

Il linguaggio di comando è pensato per un utilizzatore-uomo che usa interattivamente il calcolatore attraverso un terminale (tastiera, mouse e video). È opportuno sottolineare che utilizzatori della macchina virtuale realizzata dal s.o. sono anche i programmi applicativi (software high e medium level); in tal caso l'interazione con il s.o. avviene durante l'esecuzione del programma ed è codificata attraverso richieste specifiche, dette *chiamate al s.o.*, oppure indirettamente attraverso il linguaggio di programmazione.

Ci occuperemo solo dell'interazione tra utilizzatore-uomo e s.o.

I comandi principali riguardano la richiesta di:

- esecuzione di un programma applicativo (in l.m.);
- utilizzo di un traduttore (più in generale di un ambiente di sviluppo di programmi); si noti che anche in questo caso si tratta di una richiesta di esecuzione di un programma (il traduttore o il programma di livello più elevato dell'ambiente di sviluppo);

- utilizzo del file system:
  - visualizzazione di un sottoinsieme dell'insieme dei file;
  - visualizzazione del contenuto di una directory o di un file;
  - azioni specifiche su un file o su una directory: creazione, modifica, eliminazione, trasferimenti;
  - azioni specifiche su un insieme di file.

In molti calcolatori l'interazione è realizzata a un livello superiore rispetto a quello del linguaggio di comando, cioè attraverso l'*interfaccia grafica per l'utilizzatore* (GUI). Le funzionalità della macchina virtuale realizzata a questo livello sono visualizzabili in modo molto immediato attraverso *finestre* sullo schermo, con cui l'utilizzatore interagisce utilizzando il mouse. I dettagli riguardanti l'aspetto grafico e le funzionalità in una finestra dipendono dalla particolare GUI.

In genere una finestra contiene:

- un insieme di *menu*, ognuno costituito da un gruppo di comandi che denotano funzionalità affini;
- un insieme di *bottoni* (virtuali), che quando sono selezionati attivano una specifica funzionalità;
- un insieme di *icone*, cioè la rappresentazione grafica di un programma applicativo, di un file, di una directory, di un dispositivo di memoria secondaria o di una finestra.

Ogni finestra può essere considerata come una particolare macchina virtuale a disposizione dell'utilizzatore; inoltre se il s.o. è multitasking queste varie macchine virtuali operano *simultaneamente* (cioè in modo concorrente).

In definitiva, il modo migliore per comprendere le convenzioni per l'utilizzo di una GUI è quello di interagire operativamente con essa lasciandosi guidare dal buon senso; le modalità d'uso di una GUI sono state progettate per rendere l'interazione la più semplice e immediata possibile e infatti esse risultano essere naturali anche per un bambino.

Tra i più noti e diffusi s.o. vi è Unix (o meglio varie versioni di Unix), utilizzato in special modo su workstation e calcolatori di elevate prestazioni. Per varie versioni si intendono particolari realizzazioni di Unix, che nacque come sistema operativo non commerciale in ambiente di ricerca (Bell Labs), tagliate per specifiche macchine (per esempio, HP-UX per ws HP, Aix per ws IBM, Ultrix per sistemi Digital, Solaris per ws Sun, Irix per ws SGI, ...). Più recentemente si sta diffondendo Linux, una versione di Unix adatta ai PC. Tutte queste versioni, in genere, forniscono una potente e flessibile GUI, non presente nelle prime realizzazioni di Unix, che risalgono al 1978.

Il sistema operativo *storico* per i PC è stato sicuramente MsDos di Microsoft, di cui sono esistite varie versioni successive. Una evoluzione dei sistemi operativi per PC

è Windows, giunto alla versione Windows XP ma affermatasi con la versione Windows 95, la cui diffusione a livello mondiale ha costituito un importante passo nell'affermazione dei calcolatori come strumento di consumo di massa.
Notevole è stata la versione denominata Windows NT (NT sta per new technology), utilizzata come sistema operativo anche per calcolatori in grado di gestire reti locali complesse.
Una citazione infine al Mac Operating System, per calcolatori Apple, che per primo ha introdotto negli anni '80 l'uso massiccio di GUI nell'interazione uomo-sistema.

Il Calcolatore è la Rete
detto popolare, 1994

e) servizi di rete

Internet (la rete delle reti) è un insieme di reti interconnesse e cooperative che usano le stesse convenzioni per la trasmissione delle informazioni. Dal punto di vista dell'utilizzatore Internet agisce come una unica immensa rete che collega decine di milioni di calcolatori in tutti i continenti[36].

Utilizzare Internet significa usare uno dei *servizi di rete*, e cioè:

- la posta elettronica (*e-mail*)
- l'accesso remoto a un calcolatore (*telnet*)
- lo scambio di file (*ftp*)
- l'accesso a documenti multimediali, cioè ipertesti (*world wide web*)

Il world wide web (www o web) è stato sviluppato originariamente presso i laboratori europei del CERN a Ginevra per consentire lo scambio di informazioni multimediali tra i gruppi di scienziati impegnati in progetti comuni. In breve è diventato il principale servizio di rete, trasformandosi in un vero fenomeno sociale che ha cambiato le modalità con cui le persone, le aziende e le istituzioni acquisiscono, producono e scambiano le informazioni nelle società avanzate.

Il www è basato sul servizio di scambio di file e sui *browser*; browser è il nome che viene dato a un programma che consente di visualizzare un *ipertesto* (cioè un testo multimediale) e di interagire con esso, cioè per esempio di instaurare un collegamento con un'altra parte dell'ipertesto stesso o con un nuovo ipertesto, attraverso un semplice click del mouse.

In pratica, le informazioni risiedono su calcolatori detti *server www* (o *server web*). Attraverso un browser un qualunque calcolatore connesso a Internet può accedere a un server web e visualizzare i suoi ipertesti. La facilità

[36] Per esempio, si possono ottenere informazioni interessanti sulla rete dell'Università e della Ricerca Scientifica Italiana (Garr) al sito `http://www.garr.net/`.

con cui si stabilisce la connessione con un server, si esaminano i suoi documenti multimediali e si passa a collegarsi in sequenza con altri server, da cui creato il termine di uso comune *navigare in rete* (*websurfing*).

Un qualunque documento accessibile via web è individuato univocamente da un nome, detto URL (*uniform resource locator*). Tale nome identifica il server web in cui risiede fisicamente la risorsa, il nome del file che contiene il documento e il tipo di informazione (ovvero il protocollo per la trasmissione dell'informazione):

```
<tipo informazione>://<indirizzo server web>/<nome file>
```

Nella maggior parte dei casi si ha a che fare con documenti multimediali, individuati da `http` come `<tipo informazione>`, cioè il protocollo di trasmissione di ipertesti. L'`<indirizzo server web>` è il nome del cosiddetto *dominio Internet* di un calcolatore server web. Spesso un URL viene gergalmente detto *sito* o *indirizzo* Internet.

Di particolare utilità sono alcuni server web che consentono di ricercare i siti che contengono specifiche informazioni; la ricerca viene fatta indicando una o più parole chiave a cui si è interessati. Tali server sono detti *motori di ricerca* (*search engine*)[37].

Un utile esercizio all'uso del web consiste nel ricercare informazioni su Internet e sui suoi servizi utilizzando il web stesso!

In conclusione, il sistema operativo realizza una o più macchine virtuali che operano *simultaneamente*, o meglio in una situazione di ripartizione di risorse che avviene sequenzialmente nel tempo, ognuna con una memoria (virtuale) con capacità maggiore di quella fisica, con una gestione delle informazioni in memoria secondaria che non richiede la conoscenza di dettagli sul funzionamento dei dischi, dei nastri e degli altri dispositivi, con un interfacciamento semplice e omogeneo per il trasferimento di informazioni su/da dispositivi di I/O, con un accesso immediato alle reti e ai loro servizi.

[37] Il più diffuso motore di ricerca è: `http://www.google.com`

Capitolo terzo

Costruzione di algoritmi

[Progettando algoritmi] ... noi evochiamo gli spiriti del calcolatore con i nostri incantesimi

Harold Abelson, 1985

3.1 Un linguaggio di descrizione e criteri di sviluppo

In questo capitolo sono presentate le caratteristiche fondamentali di un linguaggio per la descrizione degli algoritmi e le metodologie che consentono di progettare un algoritmo.

Il linguaggio viene costruito per passi, a partire dall'esigenza di risolvere computazionalmente un problema. Si tratta di un linguaggio informale, chiamato *P-like*, per il quale non esiste un traduttore e che è utilizzato solo per evidenziare le caratteristiche salienti del progetto di un algoritmo, cioè la descrizione sia dell'organizzazione dei dati sia delle operazioni su di essi. L'enfasi è sugli algoritmi, non sulla notazione.

Le caratteristiche formali e le proprietà specifiche di un linguaggio di programmazione sono illustrate nel capitolo 4, dove si introduce un *vero* linguaggio di programmazione, il Matlab.

3.1.1 Variabili

La caratteristica primaria di un linguaggio di descrizione degli algoritmi è la possibilità di utilizzare *nomi* per riferirsi sia alle azioni sia agli oggetti su cui queste operano.

Una fondamentale limitazione del linguaggio macchina è rappresentata dal fatto che un dato è individuato solo attraverso l'indirizzo della voce di memoria dove è memorizzato. Una semplice astrazione consente di denotare un dato con un nome, evitando di considerare i dettagli della sua memorizzazione. Essa è basata sui due concetti fondamentali di *variabile* e *tipo*.

Una variabile è un nome a cui si può associare un dato, che viene detto il valore della variabile.

Il tipo di una variabile indica l'insieme dei valori che è possibile associare a una variabile.

L'operazione che consente di specificare in modo univoco il tipo di una variabile è detta *dichiarazione* di una variabile. Ogni variabile che si utilizza in un algoritmo è esplicitamente introdotta attraverso una dichiarazione di tipo; una variabile è detta *indeterminata* se non è stata dichiarata. Il tipo di una variabile determina l'insieme dei suoi possibili valori e le operazioni che possono essere effettuate su di essa.

In P-like l'operazione di dichiarazione si indica con

```
var <variabili>: <tipo>
```

Per esempio, le dichiarazioni

```
var lato: real
var raggio, circonferenza: real
var nome_proprio, cognome: character
var primo_indice: integer
var p: logical
```

specificano che `lato, raggio, circonferenza` sono variabili di tipo reale, cioè a cui è possibile associare dati del tipo reale, che `nome_proprio, cognome` sono variabili di tipo testo, cioè a cui è possibile associare dati del tipo testo, che `primo_indice` è una variabile di tipo intero e che `p` è una variabile di tipo logico.

In questo capitolo vengono trattate variabili a cui è possibile associare un solo valore; queste variabili sono dette *variabili scalari*. I tipi fondamentali sono il tipo intero, il tipo reale, il tipo carattere e il tipo logico, di cui si è già vista la rappresentazione nella macchina fisica e le operazioni effettuabili. Questi tipi sono detti *tipi (scalari) primitivi* del P-like.

> Un tipo consiste di *un insieme di valori*, di *un criterio di rappresentazione* di tali valori in memoria (rappresentazione interna) e di *un insieme di operazioni* consentite sui valori. Per esempio il tipo intero è costituito dall'insieme degli interi rappresentabili, dal criterio di rappresentazione posizionale in binario puro a precisione finita e dalle cinque operazioni di addizione, sottrazione, moltiplicazione, divisione intera, potenza. Le operazioni sono denotate mediante gli operatori (sul tipo): `+`, `-`, `*`, `/`, `^`.

Le variabili devono la loro ragion d'essere al fatto che si deve associare a esse un valore. Una variabile è detta *definita* quando a essa è associato un

valore, che deve essere tra quelli appartenenti al suo tipo, mentre è *indefinita* in caso contrario, cioè se non è associato ad essa alcun valore.

Si può immaginare una variabile come una scatola dotata di una etichetta (il nome): il contenuto della scatola è il valore della variabile; se la scatola è vuota, la variabile è indefinita. Questa interpretazione intuitiva di una variabile, pur se imprecisa, dà un'idea del meccanismo di astrazione su cui è basata, in quanto una variabile non è nient'altro che un modo per denotare mediante un nome una (o più) voci di memoria.

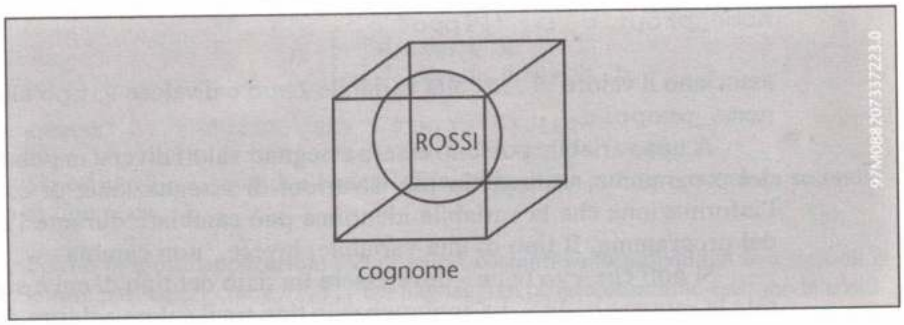


Fig. 3.0: cognome è (il nome del)la variabile e ROSSI è il valore associato

La dichiarazione di tipo stabilisce la corrispondenza tra (il nome del)la variabile e la voce di memoria a cui essa fa riferimento. Il contenuto della voce è il valore della variabile. È stato già sottolineato che il contenuto di una voce di memoria deve essere interpretato, al fine di risalire al valore corretto che è memorizzato; la dichiarazione di tipo fornisce il criterio per risalire al valore (a partire dalla rappresentazione in memoria).

La corrispondenza tra variabile e voce (voci) di memoria è detta *allocazione di memoria* alla variabile. Se l'allocazione è fatta preventivamente (e allo stesso tempo per tutte le variabili) e non è modificabile durante l'esecuzione del programma, si dice che l'allocazione è statica. Al contrario si parla di allocazione dinamica quando è possibile allocare e/o deallocare (cioè eliminare la corrispondenza tra variabile e memoria) memoria durante l'esecuzione.

L'operazione che consente di associare un valore a una variabile è l'operazione fondamentale di un algoritmo. Tale operazione può essere effettuata utilizzando due diverse istruzioni: l'istruzione di *assegnazione* e l'istruzione di *lettura da dispositivo esterno di input*.

3.1.2 Assegnazione

L'istruzione di assegnazione in P-like, nella sua forma più semplice, è

```
<variabile> := <valore>
```

Per esempio le istruzioni di assegnazione

```
lato := 3.12
nome_proprio := 'Pippo'
```

associano il valore `3.12` alla variabile `lato` e il valore `Pippo` alla variabile `nome_proprio`.

A una variabile possono essere assegnati valori diversi in punti differenti del programma, utilizzando più istruzioni di assegnazione; di conseguenza l'informazione che la variabile identifica può cambiare durante l'esecuzione del programma. Il tipo di una variabile, invece, non cambia.

Si noti che `<valore>` deve essere un dato del tipo di cui è stata dichiarata la `<variabile>`. La congruenza in tipo tra il valore a destra del simbolo di assegnazione := e la variabile a sinistra deve essere sempre garantita.

Negli esempi precedenti sono stati introdotti due oggetti (`3.12`, `'Pippo'`) sui quali è necessario fare alcune considerazioni. In molti algoritmi è utile, come si vedrà spesso nel seguito, avere la possibilità di denotare direttamente un dato, cioè senza accedervi tramite una variabile; ciò porta a introdurre le *costanti*.

Una costante denota esplicitamente un dato appartenente a un certo tipo.

Per esempio `5` è una costante di tipo intero, `3.141592` è una costante di tipo reale, `'Giulio'` è una costante di tipo testo.

Il tipo di una costante viene indicato direttamente dalla notazione che esprime la costante.

Le costanti che appartengono ai quattro tipi primitivi sono indicate in P-like nel seguente modo:

`costante testo:` una sequenza di caratteri racchiusa tra apici:

$$\text{'}a_1 \ldots\ldots a_s\text{'} \quad \text{con} \quad a_i \in \{a,b,..,A,B,...,1,2,...,+,-,...,\%,/,\ ,...\}$$

esempi:

```
'il sole', 'a13?4%', 'il risultato è', 'Rossi'
```

Si noti che l'uso degli apici è necessario perchè Rossi (senza apici) denota una variabile.

`costante logica:` true (cioè, vero), false (cioè, falso)

`costante intera:` una sequenza di cifre decimali precedute eventualmente dal segno [+-] $d_1...d_n$ con $d_i \in \{0,1,..,9\}$

esempi: `-99, +3, 3, 32167`

`costante reale:` una sequenza di cifre decimali precedute eventualmente dal segno, e contenente il punto decimale (.) oppure in notazione esponenziale contenente esplicitamente la E:

[+-] $d_1....d_{k-1}.d_k...d_n$
[+-] $d_1....d_{k-1}.d_k...d_nE$[+-]$b_1...b_n$

esempi: `3., -0.417, 5E9, +2E-1, -3.4E2`

Si noti che 3 e 3. sono due costanti diverse; la prima è di tipo intero, la seconda di tipo reale.

Poichè in molte applicazioni alcuni valori costanti sono individuati con simboli o lettere particolari ($\pi, e, ..$), nei linguaggi di programmazione esistono le cosiddette *costanti simboliche.* Si tratta di nomi a cui viene associato un valore che non può essere modificato nel corso dell'algoritmo. Tale associazione viene fatta al momento della dichiarazione della costante simbolica:

```
const pi_greco=3.1415926
const e_neper=2.7182818
```

La forma più generale dell'istruzione di assegnazione è la seguente:

```
<variabile> := <espressione>
```

Una espressione è una combinazione di variabili e costanti connesse da operatori. Una espressione deve essere corretta, cioè deve coinvolgere variabili, costanti e operatori in modo che siano rispettatate le congruenze di tipo.

Una espressione ha un unico valore, che si ottiene valutando l'espressione.

La *valutazione* di una espressione utilizza i valori delle variabili e delle costanti e quindi non deve contenere variabili indefinite o indeterminate.

ESEMPIO:

```
var alfa,beta,gamma: integer
begin
      alfa := 4
      beta := 9
      gamma := 3*(alfa+5)-beta
end
```

Dopo la prima assegnazione il valore associato ad `alfa` è 4, dopo la seconda quello associato a `beta` è 9, dopo la terza quello associato a `gamma` è 18. Si sottolinea che è consentita la usuale notazione con le parentesi per indicare esplicitamente l'ordine di valutazione all'interno dell'espressione.

ESEMPIO:

```
var i: integer
begin
      i := 1
      i := i+1
end
```

Dopo la prima assegnazione ad `i` è associato il valore 1, dopo la seconda ad `i` è associato il valore 2. Si noti che la seconda assegnazione è corretta, in quanto prima viene valutata l'espressione a destra (il cui valore è 2) e poi tale valore viene associato alla variabile `i`. Una variabile può comparire contemporaneamente a sinistra e a destra del simbolo di assegnazione.

ESEMPIO:

```
var nome_proprio,cognome,persona: character
begin
      nome_proprio := 'Carlo'
      cognome := 'Rossi'
      persona := nome_proprio//' '//cognome
end
```

Dopo la prima assegnazione a `nome_proprio` è associato il valore `Carlo`, dopo la seconda a `cognome` è associato il valore `Rossi`, dopo la terza a `persona` è associato il valore `Carlo Rossi`. Si noti l'uso dell'operatore di connessione di stringhe `//` e della costante di tipo testo `' '`.

ESEMPIO:

```
var alfa,beta: real
var delta: logical
begin
      alfa := 4.
      beta := 9.1
      delta := alfa < beta
end
```

Dopo la prima assegnazione il valore associato a `alfa` è `4.`, dopo la seconda quello associato a `beta` è `9.1`, dopo la terza quello associato a `delta` è `true`: infatti l'espressione a destra ha un valore di tipo logico (vero) e tale valore può essere associato alla variabile logica `delta`.

ESEMPIO:

```
var alfa,beta,gamma: integer
begin
      alfa := 4
      beta := 9
      beta := 3*(alfa+5)-gamma
end
```

Dopo la prima assegnazione il valore associato a `alfa` è `4`, dopo la seconda quello associato a `beta` è `9`, dopo la terza assegnazione `beta` è indefinita, poichè `gamma` è indefinita. La terza assegnazione è errata.

L'istruzione di assegnazione è errata, cioè rende indefinita la variabile a sinistra, se

- l'espressione non può essere valutata, in quanto vi compaiono variabili indefinite o indeterminate oppure le variabili e le costanti non sono connesse con operatori definiti sui loro tipi;
- il valore dell'espressione è di tipo diverso rispetto a quello della variabile a sinistra.

ESEMPIO:

```
var nome_proprio: character
var zeta:real
begin
      nome_proprio := 'Giulio'
      zeta := nome_proprio
end
```

La seconda assegnazione è errata, poichè la variabile a sinistra `(zeta)` e il valore dell'espressione a destra non sono dello stesso tipo.

ESEMPIO:

```
var alfa,beta,gamma: integer
begin
      alfa := 4
      beta := 9
      gamma := alfa//beta
end
```

La terza assegnazione è errata, poichè nell'espressione a destra l'operatore `//` è applicato a variabili di tipo intero, a cui tale operatore non è applicabile.

3.1.3 Istruzioni di I/O

L'istruzione di *lettura da dispositivo esterno di input* associa a una variabile *un valore fornito dall'esterno durante l'esecuzione del programma*, attraverso uno dei dispositivi di input del calcolatore che viene utilizzato. In un suo linguaggio di programmazione è possibile indicare il particolare dispositivo di input. In P-like assumiamo che si tratti sempre di una generica tastiera. L'istruzione di lettura dall'esterno in P-like è

```
read <variabili>
```

ESEMPIO:

```
read lato
read raggio, circonferenza
```

che associa a `lato` il dato che viene trasmesso dal dispositivo esterno[1] e che associa a `raggio` e `circonferenza` i due dati trasmessi dal dispositivo esterno.

In pratica, l'esecuzione dell'istruzione `read` sospende l'esecuzione del programma che rimane in attesa che i dati siano prelevati dall'esterno; quando il dispositivo di input ha fornito i dati, questi sono associati alle variabili che compaiono nell'istruzione di lettura, secondo l'ordine che le variabili hanno nella lista `<variabili>`. Deve essere sempre garantita la congruenza tra il tipo della variabile e quello del dato che le viene associato.

L'istruzione di *scrittura su dispositivo esterno di output*, denotata con

```
print <variabili>
```

determina l'invio sul dispositivo standard di uscita, per esempio lo schermo, del valore associato alle variabili che compaiono nella lista `<variabili>`.

Siamo in grado di scrivere un algoritmo per risolvere un semplice problema: calcolare la circonferenza c di un cerchio, di cui è noto il raggio r.

L'algoritmo ha un dato di input (il valore del raggio) e un dato di output (il valore della circonferenza). Sviluppiamo una prima versione dell'algoritmo basandoci sulla nota formula $c=2\pi r$ e in cui l'input viene fornito dall'esterno e l'output inviato all'esterno.

ALGORITMO 3.1

```
var raggio, circonferenza: real
const pi_greco = 3.1415926
```

[1] In un vero linguaggio di programmazione il dato deve essere digitato sulla tastiera e seguito da enter.

```
begin
     read raggio
     circonferenza := 2.*pi_greco*raggio
     print circonferenza
end
```

Si noti che la scelta dei nomi delle variabili e della costante simbolica è basata su ovvi motivi di leggibilità del programma; ma si ricordi che la scelta dei nomi è solo un fatto formale e la correttezza dell'algoritmo dipende esclusivamente dalla correttezza della sequenza delle operazioni. È importante sottolineare che la costante simbolica `pi_greco` è definita dal valore `3.1415926`, cioè dal numero costituito dalle prime otto cifre significative del numero π. Nell'ipotesi che il programma venga eseguito da un calcolatore con il sistema aritmetico standard in singola precisione, questo è il massimo numero di cifre significative memorizzabili per un dato reale. Se si definisse la costante `pi_greco` con un numero di cifre significative maggiore di otto, le cifre dopo l'ottava non sarebbero memorizzate nella locazione di memoria. Se si fornisse un numero di cifre significative minore di otto, il valore calcolato di circonferenza avrebbe meno di otto cifre significative accurate.

Un analogo discorso vale per l'immissione dall'esterno del dato da associare a `raggio`. Si noti che, immettendo dall'esterno per esempio il valore 1.34 per il raggio, il valore associato alla variabile `raggio` è 1.3400000; in tal caso l'ALGORITMO 3.1 calcola la circonferenza, con otto cifre significative corrette, del cerchio di raggio 1.3400000.

All'ALGORITMO 3.1 può essere associato un nome; per indicare il nome di un algoritmo in P-like si usa il comando

```
program <nome algoritmo>
```

Quindi ALGORITMO 3.1 diventa

```
program calcolo_circonferenza
var raggio, circonferenza: real
const pi_greco = 3.1415926
begin
     read raggio
     circonferenza := 2.*pi_greco*raggio
     print circonferenza
end
```

In tal modo l'algoritmo è formalmente organizzato come una entità individuata da un nome. Esso è costituito da due parti: la prima, che diremo *dichiarativa*, contiene le dichiarazioni delle variabili e delle costanti simboliche; la seconda, che diremo *esecutiva* (racchiusa dalle cosiddette parentesi `begin-end`), contiene le istruzioni. In particolare il programma contiene

istruzioni di I/O e di assegnazione, che appaiono scritte secondo un ordine che indica la sequenza temporale in cui devono essere eseguite.

Il programma `calcolo_circonferenza` è in grado di associare a qualunque valore possibile (di tipo reale) del raggio il corrispondente valore della circonferernza[2]. Se si vuole ottenere il valore della circonferenza di due (o più) cerchi di raggio diverso, è necessario eseguire due (o più) volte il programma, fornendo ogni volta in input il relativo valore del raggio.

3.1.4 Function e Procedure

L'organizzazione di un algoritmo come una entità della forma del programma `calcolo_circonferenza` ha una limitazione: il programma non può essere utilizzato da altri programmi.

La possibilità di costruire un programma mettendo insieme altri programmi è un aspetto fondamentale del progetto, costituisce una tecnica essenziale per sviluppare algoritmi per problemi complessi e, in generale, è una metodologia per dominare la complessità dei problemi e per ottenere programmi affidabili.

L'organizzazione dell'ALGORITMO 3.1 come programma `calcolo_circonferenza` consente di identificare l'algoritmo con un nome.

Il passo successivo è di far sì che un algoritmo possa utilizzare un altro algoritmo, facendo riferimento al suo nome, e avendo la possibilità di scambiare dati con esso. Ciò si ottiene introducendo nel linguaggio di programmazione due oggetti computazionali: la *function* e la *procedura*. I due concetti sono simili e ne mostreremo le proprietà analizzando prima la function.

In P-like per indicare che un algoritmo è organizzato come function si usa il comando

```
function <nome algoritmo>(<variabili>):<tipo>
```

detto *intestazione* della function o anche *testata* della function.

L' ALGORITMO 3.1. organizzato come function è

```
function circonferenza(raggio): real
var raggio: real
const pi_greco = 3.1415926
begin
     circonferenza := 2.*pi_greco*raggio
end
```

[2] A meno di overflow/underflow nella valutazione dell'espressione che definisce il valore di `circonferenza`.

Si è soliti chiamare *corpo* della function l'insieme dei comandi che compaiono dopo l'intestazione.

La lista delle variabili che appaiono nell'intestazione è la lista dei *parametri* (o *argomenti*) *formali* della function e indica le variabili che identificano i dati di input dell'algoritmo.

Nell'esempio la `function circonferenza` ha come unico parametro formale la variabile `raggio`. Si noti l'assenza, rispetto all' ALGORITMO 3.1 , di istruzioni di I/O nella function. La variabile `raggio` è lasciata indefinita (senza valore associato) nel corpo della function.

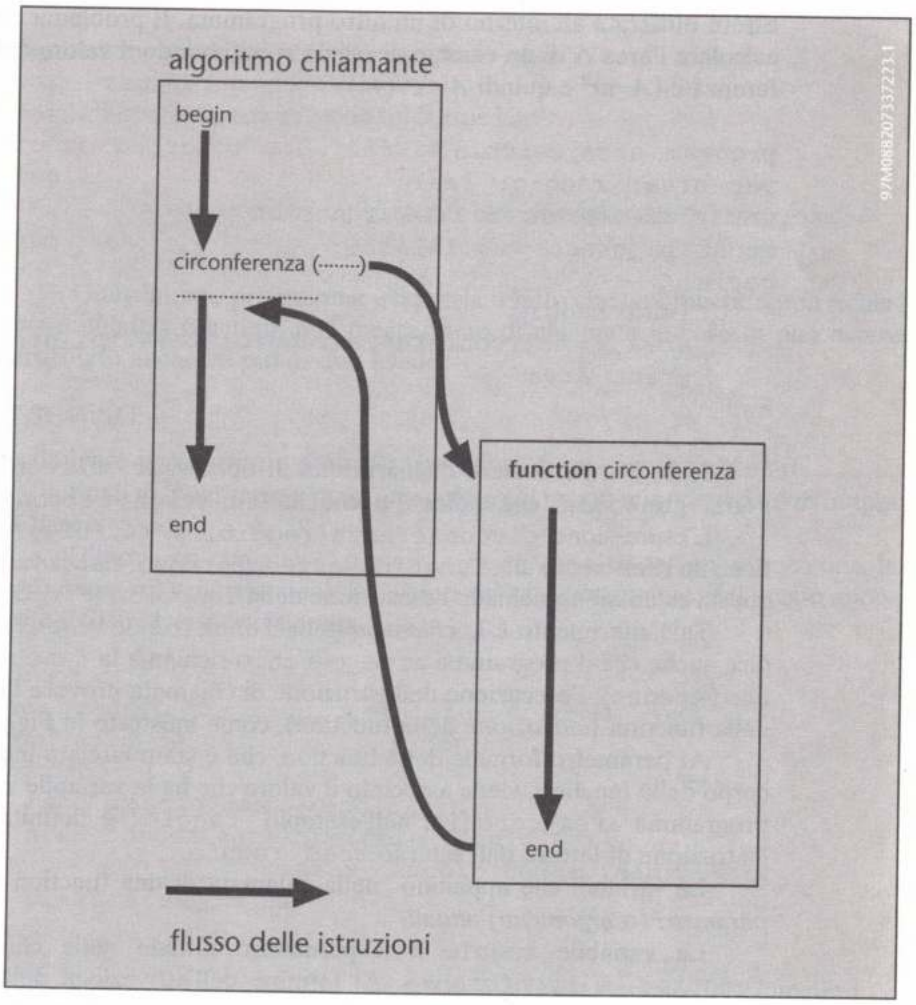


Fig. 3.1: attivazione di una function

In P-like il nome della function, `circonferenza` nell'esempio, *deve essere il nome della variabile a cui è associato l'unico dato di output dell'algoritmo.*

Il tipo del dato di output della function è specificato da `<tipo>` nell'intestazione. Tale dato è detto valore restituito dalla function.

Una function deve necessariamente contenere una istruzione di assegnazione che definisca il valore restituito dalla function (l'istruzione `circonferenza:=2.*pi_greco*raggio`, nell'esempio).

Il seguente esempio mostra come la `function circonferenza` può essere utilizzata all'interno di un altro programma. Il problema consiste nel calcolare l'area A di un cerchio di raggio r, utilizzando il valore della circonferenza c ($A=\pi r^2$ e quindi $A= c^2/(4\pi)$)

```
program area_cerchio
var area, raggio: real
var circonferenza: real function
const pi_greco = 3.1415926
begin
     read raggio
     area := circonferenza(raggio)^2/(4.*pi_greco)
     print area
end
```

Si noti la particolare dichiarazione di tipo `var circonferenza: real function` che indica il tipo della `function circonferenza`.

L'espressione `circonferenza(raggio)^2/(4.*pi_greco)` contiene un riferimento alla `function circonferenza`, cioè la valutazione di questa espressione richiede l'esecuzione della `function circonferenza`.

Tale riferimento è la *chiamata* della `function circonferenza` (si dice anche che il programma `area_cerchio` richiama la `function circonferenza`); l'esecuzione dell'istruzione di chiamata provoca l'esecuzione della function (*attivazione* della function), come mostrato in Fig. 3.1.

Al parametro formale della function, che è stato lasciato indefinito nel corpo della function, viene associato il valore che ha la variabile `raggio` nel programma `area_cerchio`; nell'esempio `raggio` è definita mediante l'istruzione di lettura dall'esterno `read raggio`.

Le variabili che appaiono nella chiamata di una function sono dette *parametri* (o *argomenti*) *attuali*.

La variabile `raggio` è il parametro attuale nella chiamata alla `function circonferenza`. Al termine dell'attivazione della function, cioè dopo l'esecuzione della sua ultima istruzione, il valore associato alla variabile `circonferenza` e restituito dalla function può essere utilizzato per

valutare l'espressione. Ovviamente l'area A di un cerchio di raggio r è più semplicemente calcolabile con $A=\pi r^2$ come nel seguente algoritmo, organizzato come function

```
function area_cerchio(raggio): real
var  raggio: real
const pi_greco = 3.1415926
begin
        area_cerchio := pi_greco*raggio^2
end
```

invece di

```
function area_cerchio(raggio): real
var  raggio: real
var circonferenza: real function
const pi_greco = 3.1415926
begin
       area_cerchio := circonferenza(raggio)^2/(4.*pi_greco)
end
```

Quest'ultimo programma evidenzia il fatto che una function può richiamare un'altra function; tale meccanismo di chiamata in cascata può essere articolato anche su più di due livelli.

ESEMPIO:

sviluppare un algoritmo per calcolare l'area di una corona circolare.

I dati di input sono il raggio maggiore e il raggio minore; il dato di output è l'area.

Denotiamo con `raggio_max`, `raggio_min` e `area_corona` le variabili a cui vengono associati i valori del raggio maggiore, del raggio minore e dell'area, rispettivamente.

```
program corona_circolare
var raggio_max,raggio_min,area_corona: real
var area_cerchio: real function
begin
 read raggio_max, raggio_min
 area_corona:=area_cerchio(raggio_max)-
                area_cerchio(raggio_min)
 print area_corona
end
```

Questo esempio mostra una proprietà fondamentale delle function (e delle procedure), cioè la possibilità di essere chiamate più volte con differenti parametri attuali.

La prima chiamata (`area_cerchio(raggio_max)`) provoca l'attivazione della `function area_cerchio`, in cui al parametro formale `raggio` risulta associato il valore del parametro attuale `raggio_max`. Al termine dell'esecuzione della function si ottiene il valore del primo dei due addendi dell'espressione

`area_cerchio(raggio_max)-area_cerchio(raggio_min)`.

La seconda chiamata (`area_cerchio(raggio_min)`) provoca l'esecuzione della `function area_cerchio`, in cui al parametro formale `raggio` risulta associato il valore del parametro attuale `raggio_min`. Al termine della seconda attivazione della `function area_cerchio`, anche al secondo addendo è associato un valore; quindi l'espressione viene valutata e il suo valore viene associato alla variabile `area_corona`.

La Fig. 3.2 mostra in modo intuitivo come opera il meccanismo delle chiamate di una function.

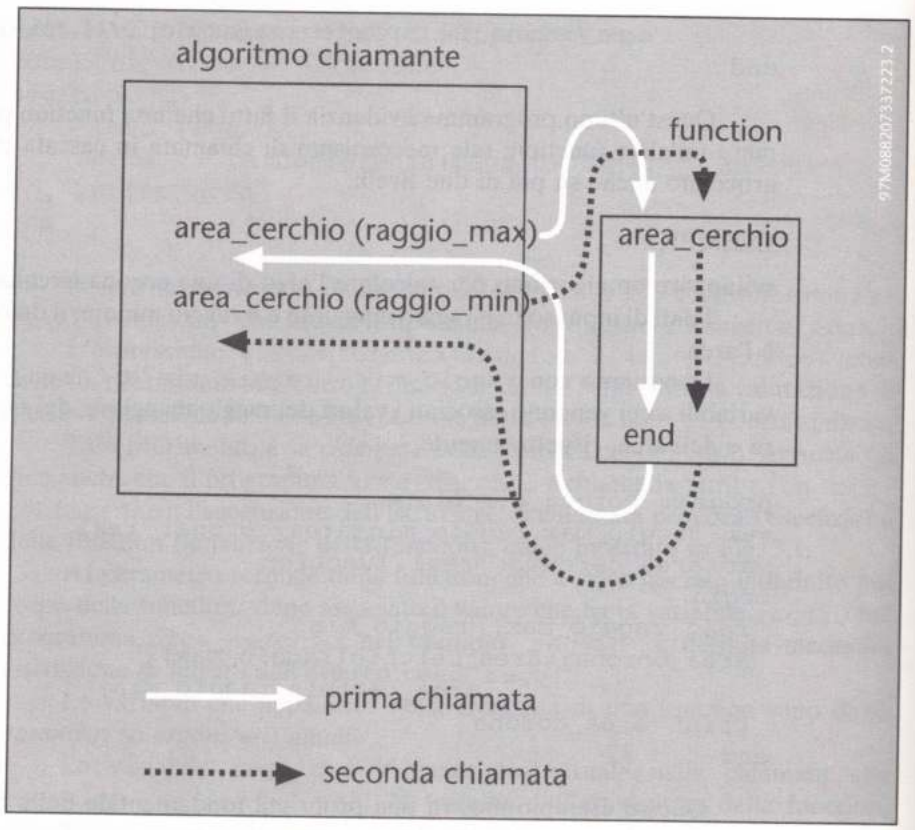


Fig. 3.2: attivazioni successive di una function

La corrispondenza che si determina tra parametri formali e parametri attuali, che rimane attiva per tutto e solo il processo di esecuzione della function chiamata, è detta *passaggio dei parametri*.

Il passaggio dei parametri è un aspetto centrale dell'uso delle function (e delle procedure); esso costituisce l'unico canale di comunicazione tra il programma chiamante e il programma chiamato. In P-like il passaggio dei parametri è realizzato attraverso il *meccanismo di sostituzione del valore*, cioè avviene copiando il valore del parametro attuale (di input) nella voce di memoria associata al parametro formale.

Le variabili che non sono parametri attuali (e non sono il nome della function), nel programma chiamante, sono dette variabili *locali* e non sono utilizzabili dal programma chiamato.

Allo stesso modo nel programma chiamato le variabili che non sono parametri formali (e non sono il nome della function) sono locali e il loro valore è *nascosto* al programma chiamante.

Nei linguaggi di programmazione esiste un insieme di function predefinite, che è possibile richiamare direttamente; tali function sono dette *primitive* (o anche *built-in functions*) Di seguito si enumerano alcune function primitive che verranno utilizzate negli esempi. I dettagli sui parametri di input e di output di queste function, quando non siano banali, sono precisate nell'esempio dove sono utilizzate per la prima volta:
`abs` (calcola il valore assoluto), `sqrt` (calcola la radice quadrata), `sin`, `cos`, `tan`, `cotan`, `asin` (arcseno), `acos`, `atan`, `acotan`,..., `log`, `log10`, `exp` (esponenziale), ..., `mod`,...,`int`, `float`,.....

Nel seguito si osserva la convenzione di non dichiarare esplicitamente il tipo delle function primitive nel programma che le richiama.

Il seguente esempio mostra come utilizzare il passaggio dei parametri nel caso di una function con due parametri formali.

ESEMPIO:

sviluppare una function per il calcolo della distanza d dall'origine di un punto di coordinate (x, y). È ben noto che la soluzione è data da $d = \sqrt{x^2 + y^2}$; l'algoritmo risolutore, organizzato come function, è il seguente:

```
function distanza_origine(x,y): real
var x,y: real
begin
      distanza_origine := sqrt(x^2+y^2)
end
```

Si noti che l'assegnazione `distanza_origine := sqrt(x^2+y^2)` richiede l'attivazione della function primitiva `sqrt(z)` per il calcolo della radice quadrata dell'argomento. È importante sottolineare che nella chiamata `sqrt(x^2+y^2)` non vi è il parametro attuale, ma una espressione: un parametro attuale può essere sostituito da una espressione, il cui valore viene associato al corrispondente parametro formale al momento dell'attivazione.

Il seguente programma calcola la distanza dall'origine di due punti, le cui coordinate sono fornite in input:

```
program usa_distanza
var primo_punto, secondo_punto,primo_x, .....
        primo_y,secondo_x,secondo_y: real
var distanza_origine: real function
begin
      read primo_x,primo_y,secondo_x,secondo_y
      primo_punto := distanza_origine(primo_x,primo_y)
      secondo_punto := distanza_origine .....
                              (secondo_x,secondo_y)
      print primo_punto, secondo_punto
end
```

La chiamata a una function può anche avvenire in espressioni che compaiono in istruzioni diverse dall'assegnazione, come per esempio:

```
print distanza_origine(primo_x,primo_y), .....
       distanza_origine(secondo_x,secondo_y)
```

La *corrispondenza tra parametri attuali e formali* è stabilita dalla *posizione* dei parametri nella *lista dei parametri formali* e *nella lista dei parametri attuali.* L'argomento attuale che appare al primo posto corrisponderà, all'atto della chiamata, all'argomento formale che appare al primo posto; l'argomento attuale che appare al secondo posto corrisponderà al secondo argomento formale.

Nella prima chiamata dell'esempio la lista dei parametri attuali è `primo_x,primo_y` e quella dei parametri formali è `x,y`; quindi la corrispondenza è tra `primo_x` e `x`, tra `primo_y` e `y`. Analogamente per la seconda chiamata la corrispondenza è tra `secondo_x` e `x`, tra `secondo_y` e `y`.

Tale corrispondenza temporanea è basata solo sull'ordine in cui le variabili compaiono nella lista e non su analogie tra i nomi.

Il discorso può essere ovviamente generalizzato a liste con più di due variabili.

In ogni caso deve essere rispettata l'uguaglianza di tipo tra ogni parametro attuale e il corrispondente parametro formale.

In P-like la function è un criterio di organizzazione di un algoritmo che

ha uno o più dati di input e un unico dato di output. Il fatto che il dato di output di una function sia il valore associato al nome della function è una caratteristica che spesso risulta di grande utilità, in quanto rende possibile denotare semplicemente la chiamata in cascata di più function.

Si consideri per esempio l'espressione

```
sqrt(distanza_origine(abs(cos(teta)),abs(sin(teta))))
```

che mostra come sia possibile avere come parametro attuale una espressione complessa che contiene chiamate ad altre function. In tal caso la sequenza temporale delle chiamate è la seguente: function `cos`, function `sin`, function `abs`, function `abs`, function `distanza_origine`, function `sqrt`.

Questa modalità di chiamate in cascata è detta *innestamento di function*.

Nel caso un algoritmo abbia più di un dato di output, oppure non si sia interessati all'innestamento, è possibile organizzare il programma come *procedura*.

In P-like l'intestazione di una procedura è

```
procedure <nomealgoritmo>.....
      (in:<variabili>;inout:<variabili>;out:<variabili>)
```

dove le variabili che appaiono dopo `in:` sono i parametri formali di input, quelli che appaiono dopo `out:` sono i parametri formali di output e quelli dopo `inout:` sono parametri formali sia di input sia di output.

L'ALGORITMO 3.1 organizzato come procedura è

```
procedure circ(in: raggio;out: circonferenza)
var raggio,circonferenza: real
const pi_greco = 3.1415926
begin
     circonferenza := 2.*pi_greco*raggio
end
```

Si noti che il nome della procedura denota solo l'algoritmo e a esso non è associato alcun dato[3].

[3] In P-like il passaggio dei parametri è realizzato mediante la sostituzione del valore: il valore di un parametro attuale di input viene copiato nella voce di memoria associata al corrispondente parametro formale; il valore di un parametro formale di output viene copiato, al termine dell'esecuzione della procedura, nella voce di memoria associata al corrispondente parametro attuale; nel caso dei parametri di input e output (inout) convivono i due precedenti approcci.

Nel testo di un programma la chiamata di una procedura viene denotata semplicemente mediante il nome della procedura, come nell'esempio

```
program usa_circ
var raggio, circonferenza: real
begin
      read raggio
      circ(raggio, circonferenza)
      print circonferenza
end
```

La seguente procedura realizza lo scambio del valore di due variabili di tipo intero[4]:

```
procedure scambiare_i(inout:variabile_1,variabile_2)
var variabile_1,variabile_2, .....
     variabile_di_appoggio: integer
begin
     variabile_di_appoggio := variabile_1
     variabile_1 := variabile_2
     variabile_2 := variabile_di_appoggio
end
```

L'esecuzione del programma:

```
program usa_scambiare_i
var alfa, beta: integer
begin
      read alfa, beta
      scambiare_i(alfa, beta)
      print alfa, beta
end
```

fornendo in input, per esempio, i dati `7, 33`, visualizza in output `33, 7`.

All'interno di una procedura o di una function è possibile chiamare altre procedure o altre function, senza limitazioni in livelli. La strategia di organizzazione modulare di un programma è spesso indispensabile per la risoluzione di problemi complessi.

Una function o procedura può essere vista e utilizzata come una scatola chiusa, una volta che si sappia come chiamarla all'interno di un altro program-

[4] Nel seguito si fa ampio uso della procedura `scambiare_i` e anche di analoghe procedure, denotate con `scambiare_r` e `scambiare_c`, che realizzano lo scambio di valori di coppie di variabili di tipo reale e di tipo carattere, rispettivamente.

ma. Le informazioni necessarie per la chiamata sono: il nome della function o della procedura e la lista dei parametri formali (insieme al loro tipo); spesso è necessario conoscere il sottoinsieme dei valori che è possibile associare ai dati di input affinchè la procedura possa essere eseguita correttamente (si pensi ad esempio a dati di input che provocano *overflow* nell'esecuzione della function o della procedura). Tali informazioni sono dette *specifiche*.

Per usare una function o una procedura a scatola chiusa non è necessario conoscere alcun dettaglio sull'algoritmo (*implementato* dalla procedura, come si suole dire), ma solo quali sono i parametri di input e di output. Spesso è importante sapere solo *che cosa* calcola una procedura, non *come* calcola. Infatti nell'esempio della `function distanza` è stata usata la `function sqrt`, senza che sia stato precisato quale sia l'algoritmo per il calcolo della radice quadrata di un numero; ancora, nell'esempio del programma `corona_circolare` si può richiamare una qualunque delle due function per il calcolo dell'area del cerchio che abbiamo sviluppato. Per questo motivo si dice che function e procedure consentono di realizzare l'*astrazione funzionale*, cioè di organizzare il processo di risoluzione per livelli di visibilità, ignorando i dettagli che non si ritengono essenziali ad un certo livello. Chiariremo questo concetto chiave con molti esempi nel seguito, anche se gli esempi della `function distanza` e del programma `corona_circolare` sono sufficientemente esplicativi dell'idea. Al livello della `function distanza` non interessano i dettagli del calcolo della radice quadrata; questi dettagli sono nascosti utilizzando la `function sqrt`, che dunque astrae dal particolare algoritmo per calcolare la radice e indica che bisogna calcolare, in qualche modo, la radice quadrata. Analogamente, al livello del programma `corona_circolare`, l'idea principale è che l'area della corona circolare deve essere calcolata come differenza tra l'area del cerchio maggiore e quella del cerchio minore; i dettagli sul calcolo dell'area del cerchio non sono rilevanti per sviluppare questa idea e sono nascosti mediante l'uso della `function area_cerchio`. In altre parole, cambiare il corpo della `function area_cerchio`, cioè l'algoritmo per il calcolo dell'area del cerchio, non comporta nessun cambiamento nel programma chiamante `corona_circolare`.

3.1.5 Costrutti di controllo

È stato più volte detto che un algoritmo descrive un *processo di calcolo*, cioè sequenze di operazioni dette *sequenze computazionali*.

Analizzando i semplici algoritmi sviluppati finora, si nota che una operazione della sequenza computazionale è una istruzione di assegnazione oppure una istruzione di I/O. Una chiamata di procedura (o di function) indica che la sequenza computazionale continua con quella descritta dall'algoritmo implementato dalla procedura.

Si noti anche che la sequenza computazionale è stata finora indicata in modo esplicito, cioè scrivendo una dopo l'altra le varie istruzioni. Tuttavia in un procedimento risolutivo di un problema, spesso è necessario operare delle scelte, cioè scegliere una tra varie possibili sequenze computazionali, in dipendenza del valore di qualche variabile.

Si consideri ad esempio il problema del calcolo del valore assoluto $|x|$ di un numero reale. L'algoritmo per tale problema deve operare la scelta tra l'istruzione `a:=`x oppure `a:=-`x per calcolare la soluzione `a`, in dipendenza del fatto che sia $x \geq 0$ oppure $x < 0$ (in questo semplice caso vi sono solo due sequenze possibili e ognuna è costituita da un'unica istruzione).

Il comando che denota la scelta tra due sequenze computazionali è il *costrutto di selezione*, detto anche *costrutto condizionale*,

```
if <predicato>
  then
      <sequenza 1>
else
      <sequenza 2>
endif
```

dove `<predicato>` è una espressione logica, cioè una espressione che può assumere come valore *vero* oppure *falso*; `<sequenza 1>` è eseguita nel caso `<predicato>` sia vero, `<sequenza 2>` è eseguita nel caso `<predicato>` sia falso.

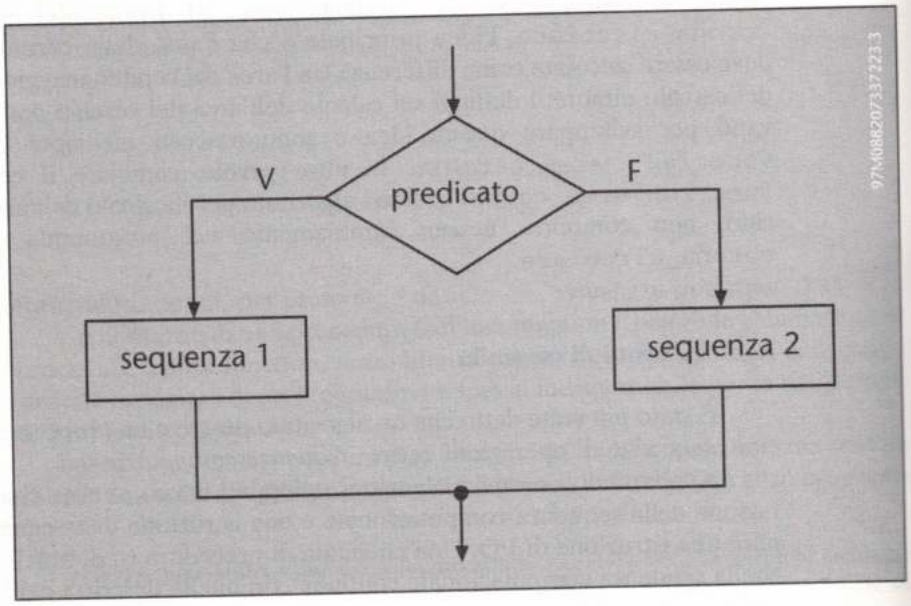


Fig. 3.3: costrutto `if then else`

Il costrutto di selezione nella notazione grafica del flowchart è riportato in Fig. 3.3 che sottolinea visivamente che al termine dell'esecuzione della `<sequenza 1>` (o della `<sequenza 2>`) la sequenza computazionale continua con l'istruzione dopo `endif`.

Il costrutto condizionale non modifica il valore di nessuna variabile, ma indica solo come è organizzata la sequenza computazionale.

Vi sono vari costrutti che operano un controllo sulla sequenza delle operazioni in un algoritmo e che sono appunto detti *costrutti di controllo*.

La selezione è dinamica, in quanto dipende dal valore del predicato, cioè dai valori delle variabili che compaiono nel predicato; tali valori possono essere modificati durante l'esecuzione dell'algoritmo.

La seguente `function valore_assoluto(x)` calcola |x|:

```
function valore_assoluto(x): real
var x: real
begin
      if x >= 0.0
          then
              valore_assoluto := x
          else
              valore_assoluto := -x
      endif
end
```

Si noti l'uso in P-like del simbolo `>=` per indicare $\geq$ (analogamente, si usa `<=` per $\leq$, e `~=` per $\neq$).

Viene proposta anche la versione alternativa

```
function valore_assoluto(x): real
var x: real
begin
      valore_assoluto := x
      if x < 0.0
          then
              valore_assoluto := -x
      endif
end
```

che utilizza la forma abbreviata del costrutto di selezione

```
if <predicato>
   then
       <sequenza>
endif
```

cioè, in flowchart:

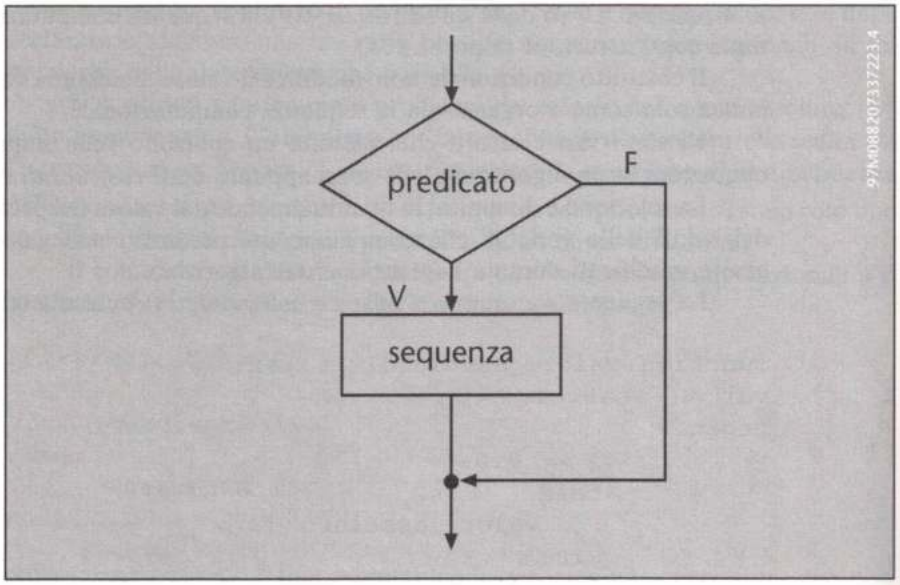


Fig. 3.4: costrutto di selezione semplificato `if then`

La seguente `function appartiene(x,a,b)` determina se `x` appartiene all'intervallo `[a,b]`:

```
function appartiene(x,a,b): logical
var x,a,b: real
begin
      if x >= a and x <= b
          then
              appartiene := true
          else
              appartiene := false
      endif
end
```

Il costrutto di selezione è basato sul *principio del terzo escluso* (*tertium non datur*). Tale principio, che risale alla logica aristotelica, afferma che nell'alternativa tra due giudizi contraddittori (predicato vero, predicato falso) è esclusa ogni altra possibilità.

Nel caso un algoritmo debba prevedere la scelta tra tre o più sequenze, in dipendenza del valore di verità di più predicati, è necessario connettere in cascata due o più costrutti di selezione.

Nell'esempio seguente, il problema è il calcolo di:

$$a = \begin{cases} 1 & \text{se} \quad x > 0 \\ 0 & \text{se} \quad x = 0 \\ -1 & \text{se} \quad x < 0 \end{cases}$$

L'algoritmo è basato sulle selezioni in cascata:

```
........
if x > 0.0
   then
       a := 1.0
    else if x = 0.0
        then
            a := 0.0
         else
             a :=-1.0
    endif
endif
........
```

La generalizzazione al caso di tre o più costrutti di selezione in cascata è ovvia.

Si noti che un algoritmo che comprende un costrutto di selezione indica più di una sequenza computazionale; la particolare sequenza che viene eseguita dipende dal valore dei dati di input.

Nella quasi totalità delle applicazioni concrete, gli algoritmi richiedono la ripetizione dell'esecuzione di sequenze di operazioni; ovvero, la sequenza computazionale descritta dall'algoritmo è costituita da sottosequenze che si ripetono.

È fondamentale poter descrivere con una notazione semplice e concisa la ripetizione di una sequenza di operazioni.

Si supponga di dover eseguire mille volte l'istruzione `a:=a/2.0` con `a` variabile reale.

La sequenza computazionale è formata da mille operazioni. Il linguaggio di programmazione deve poter descrivere in modo convenzionale un tale tipo di sequenza; la soluzione di scrivere mille volte di seguito l'istruzione `a:=a/2.0` non è una buona idea!

Il *costrutto di ripetizione* `for` (o *ciclo* `for`) consente di denotare semplicemente la sequenza computazionale formata dalla ripetizione N volte di una sequenza di operazioni:

```
for i=1,N do
     <sequenza>
endfor
```

Il costrutto di ripetizione `for` nella notazione grafica del flowchart è mostrato in Fig. 3.5.

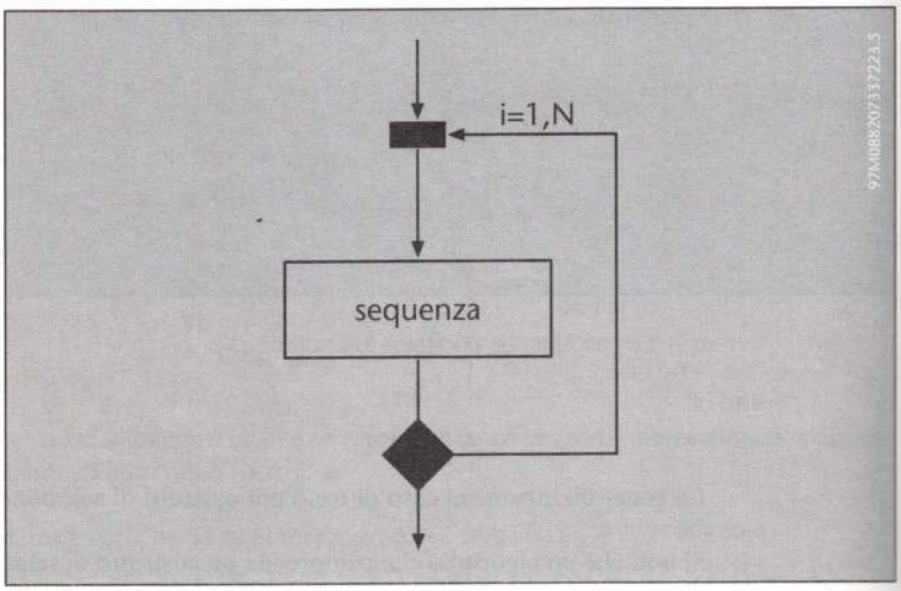


Fig. 3.5: costrutto di ripetizione `for`

L'esecuzione per mille volte dell'istruzione `a:=a/2.0` viene denotata nel seguente modo :

```
......
a:=1.0
......
for i=1,1000   do
     a:=a/2.0
endfor
print a
......
```

La forma generale del costrutto `for` è:

```
for <variabile> = <espr 1>, <espr 2>,step= <espr 3> do
      <sequenza>
endfor
```

dove `<variabile>`, detta *contatore del ciclo*, varia da `<espr 1>` a `<espr 2>` e viene incrementata di `<espr 3>` ogni volta al termine dell'esecuzione delle operazioni che costituiscono la `<sequenza>`.

Una esecuzione di tutte le operazioni della `<sequenza>` è detta *passaggio* o *passo* del ciclo. Se `step=<espr 3>` è omessa, l'incremento del contatore del ciclo è unitario. Per semplicità, in P-like i contatori di ciclo non devono essere né dichiarati né definiti esplicitamente nell'algoritmo; inoltre, l'incremento del contatore è automatico. All'interno del ciclo, cioè nelle istruzioni che costituiscono la `<sequenza>`, è possibile utilizzare il contatore nelle espressioni, ma non è possibile modificare il suo valore né il valore delle altre variabili che compaiono nelle tre `<espr>`.

ESEMPIO:

sviluppare un algoritmo per visualizzare in output i numeri interi pari tra -8 e 10.

L'esecuzione dell'algoritmo:

```
program pari
begin
      for i=-8,10, step=2   do
            print i
      endfor
end
```

produce in output: `-8  -6 ..... 8  10;`

mentre l'esecuzione di

```
program pari
begin
      for i=10,-8, step=-2   do
            print i
      endfor
end
```

produce: `10  8 .....-6  -8.`

ESEMPIO:

sviluppare un algoritmo che visualizzi in output la sequenza dei numeri dispari compresi tra i due numeri dispari `a` e `b`.

```
procedure stampa_dispari(in: a,b)
var a,b: integer
var inizio,fine: integer
var dispari: logical function
begin
      if dispari(a) and dispari(b)
         then
            if a < b
                then
                   inizio := a
                   fine := b
                 else
                   inizio := b
                   fine := a
            endif
            for i=inizio,fine, step=2   do
                print i
            endfor
         else
            print 'errore nei dati di input'
      endif
end
```

La `function dispari(x)` di tipo logico assume il valore vero se il valore di x è un numero dispari; il predicato `dispari(a) and dispari(b)` è vero solo se i valori di `a` e `b` sono entrambi numeri dispari. Si noti che la presenza del costrutto di selezione ha lo scopo di controllare se i valori dei parametri attuali di chiamata sono corretti, alla luce delle specifiche della procedura `stampa_dispari` (cioè se sono entrambi numeri dispari).

Un algoritmo che determina se il valore di una variabile intera `x` è dispari è il seguente

```
function dispari(x): logical
var x: integer
begin
      if x-2*(x/2) ~= 0
         then
            dispari := true
          else
            dispari := false
```

```
        endif
end
```

Si ricorda che `x/2` è la divisione intera. Una versione alternativa è:

```
function dispari(x): logical
var x: integer
begin
      if mod(x,2) = 1
         then
            dispari := true
          else
            dispari := false
      endif
end
```

dove si fa uso della function primitiva `mod(r,s)` che calcola `r` modulo `s`, cioè il resto della divisione intera tra `r` e `s`.

Il costrutto di ripetizione `for` può essere usato solo quando è noto il numero dei passi di ripetizione, cioè quante volte il ciclo deve essere ripetuto.

In molti problemi si ha la necessità di denotare la ripetizione di operazioni senza conoscere a priori il numero delle ripetizioni. In tal caso il costrutto `for` non può essere utilizzato.

Il costrutto di *ripetizione* (o *ciclo*) `repeat until` denota la ripetizione di una sequenza computazionale in dipendenza del valore di un predicato:

```
repeat
    <sequenza>
until <predicato>
```

La notazione flowchart per il costrutto di ripetizione `repeat until` è in Fig. 3.6.

La `<sequenza>` di operazioni viene eseguita almeno una volta e l'esecuzione viene ripetuta se il valore del `<predicato>` è falso; quando il `<predicato>` è vero la ripetizione termina.

ESEMPIO: il seguente algoritmo

```
program es_repeat
var i: integer
begin
    i := 0
    repeat
         i := i+1
```

```
        print i
    until i=10
end
```

visualizza in output `1 2 3 ... 10`.

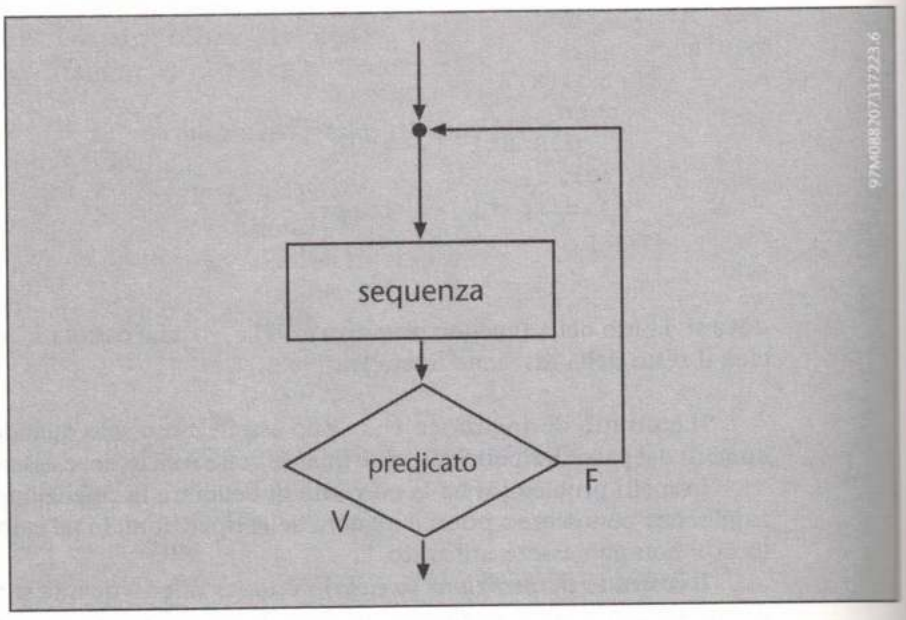


Fig. 3.6: costrutto di ripetizione `repeat until`

Si osservi che il valore di almeno una delle variabili che compaiono nel predicato deve essere modificato dalle istruzioni che fanno parte della sequenza di operazioni ripetuta, altrimenti il predicato non cambia il suo valore; in tal caso, se tale valore è falso, la ripetizione non termina provocando un classico errore di programmazione (*ciclo infinito*).

Inoltre, per convenzione, si stabilisce che il costrutto `repeat`, al contrario del `for`, non gestisce automaticamente il valore del contatore di ciclo.

ESEMPIO:
la procedura `epsmach` calcola una stima dell'epsilon macchina di un calcolatore:

```
procedure epsmach(out:epsilon)
var epsilon, epspiu1  : real
begin
```

```
        epsilon := 1.0
        repeat
            epsilon := epsilon/2.0
            epspiu1 := epsilon+1
        until epspiu1 = 1.0
        epsilon = 2 * epsilon
end
```

Un altro costrutto di ripetizione analogo al `repeat until` è il costrutto di *ripetizione* (o *ciclo*) `while`:

```
while <predicato>
    <sequenza>
endwhile
```

La notazione flowchart per il costrutto di ripetizione `while` è la mostrata in Fig. 3.7.

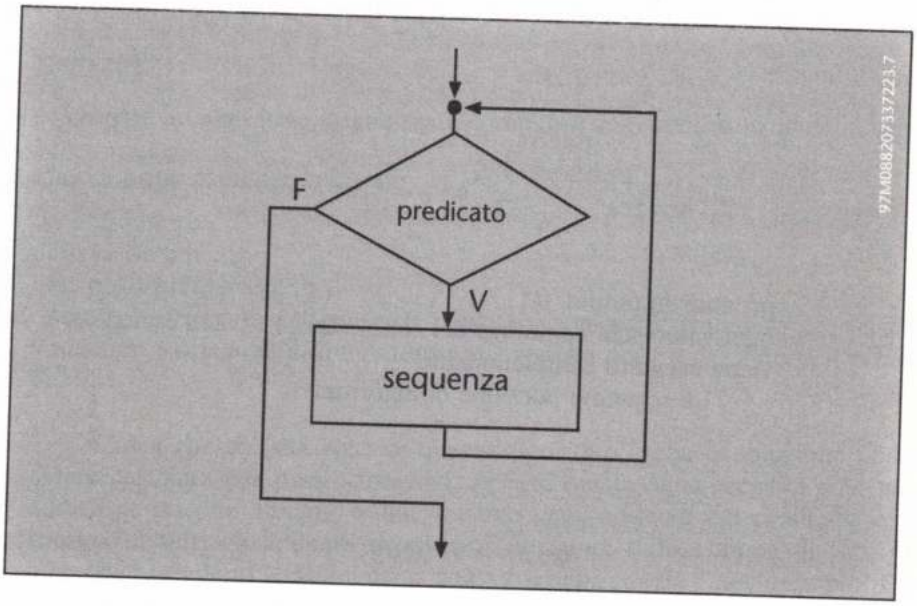


Fig. 3.7: costrutto di ripetizione `while`

La `<sequenza>` di operazioni viene eseguita se il valore del `<predicato>` è vero; quando il `<predicato>` è falso la ripetizione termina.

ESEMPIO:
il seguente algoritmo

```
program es_while
var i: integer
begin
    i := 0
    while i < 10
         i := i+1
         print i
    endwhile
end
```

produce in output `1  2  3 ... 10`.

Un aspetto molto importante dell'uso dei costrutti di ripetizione è la possibilità di *innestare* i cicli, cioè inserire un ciclo all'interno di un altro ciclo.

La seguente porzione di algoritmo:

```
.......
    for i=1,2 do
        for j=1,3 do
            print '(',i,j,')'
         endfor
     endfor
.......
```

produce in output `(1 1) (1 2) (1 3) (2 1) (2 2) (2 3)`; per ogni valore del contatore del ciclo esterno (`i` nell'esempio), il ciclo interno viene eseguito completamente

La seguente porzione di algoritmo

```
.......
    for j=1,3
        for i=1,2
            print '(',i,j,')'
         endfor
     endfor
.......
```

produce in output `(1 1) (2 1) (1 2) (2 2) (1 3) (2 3)`.

Si osservi che in entrambi i casi l'istruzione `print` viene eseguita 2x3=3x2=6 volte.

3.1.6 Prime indicazioni per sviluppare algoritmi

In questo paragrafo viene utilizzato il sottoinsieme del linguaggio P-like, finora illustrato, per progettare algoritmi per la risoluzione di semplici problemi.

Il progetto viene affrontato attenendosi a una semplice disciplina di sviluppo che prevede l'individuazione di:

dati di input del problema
dati di output del problema
strutture ripetitive nel procedimento risolutivo
  costrutti ripetitivi più adatti
  operazioni che vengono ripetute
  eventuali predicati di uscita dai cicli

Queste direttrici da seguire nel progetto non sono rigorose e non danno la garanzia di uno sviluppo, per così dire, automatico dell'algoritmo; tuttavia possono risultare utili a chi si appresta per la prima volta a sviluppare semplici algoritmi.

ESEMPIO:

sviluppare un algoritmo, organizzato come function, per il calcolo della somma dei primi n numeri naturali: $s_{nat} = \sum_{i=1}^{n} i = 1+2+3+\ldots+n$
dato di input: n
dato di output: s_{nat}
costrutto ripetitivo: `for`
operazione ripetuta (al generico passo `i`): sommare il numero `i` (che è l'addendo `i`-simo della sommatoria) alla somma degli `(i-1)` numeri precedenti.

L'idea che sta alla base di questo algoritmo è che la soluzione possa essere calcolata per passi successivi. A ogni passo viene eseguita solo una addizione tra due numeri, e cioè tra uno degli addendi del problema e la somma di tutti gli addendi precedenti. Il valore della somma di tutti gli addendi precedenti è associato a una variabile, `somma` nell'esempio. Più precisamente, al generico passo `i` del ciclo, il nuovo valore da assegnare a `somma` è ottenuto sommando un nuovo addendo (si noti che il nuovo addendo è proprio `i`) al valore assegnato a `somma` al passo precedente. È interessante osservare che al termine del passo `i`, il valore di `somma` è la soluzione del problema della somma dei primi `i` numeri naturali. Poichè la soluzione è costruita attraverso incrementi successivi, questa idea è detta *approccio incrementale* ed è di portata generale.

```
function somma(n): integer
var n: integer
begin
    somma := 0
    for i=1,n do
        somma := somma + i
    endfor
end
```

La struttura di questo algoritmo è estremamente significativa: la prima assegnazione definisce il valore della variabile `somma` (`somma:=0.0`) ed è detta *inizializzazione*; il contatore `i` del ciclo `for` assume in sequenza il valore di tutti gli addendi (il suo range sono i numeri naturali da `1` a `n`); alla variabile `somma` è associato ad ogni passo `i` il valore della somma (parziale) dei primi `i` numeri naturali; al termine della ripetizione (`i=n`) il valore associato a `somma` è la soluzione del problema.

Riportiamo in dettaglio la sequenza computazionale descritta dalla `function somma` per `n=5`:

`somma:=0+1;` `somma:=1+2;` `somma:=3+3;` `somma:=6+4;` `somma:=10+5.`

Il progetto di un algoritmo risolutore di un problema si basa sulle conoscenze che si hanno a proposito del problema stesso. Nel caso del problema della somma dei primi n numeri naturali esiste una formula che consente di calcolare direttamente la soluzione (l'aneddotica la fa risalire al grande matematico F. Gauss, che pare la ricavò a sette anni, per sfuggire al tedioso calcolo manuale delle somme impostogli come punizione a scuola):

$$s_{nat} = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

La dimostrazione è semplice:

$$\begin{aligned} s_{nat} &= 1 + 2 + 3 + \ldots + (n-1) + n \quad + \\ s_{nat} &= n + (n-1) + (n-2) + \ldots + 2 + 1 \quad = \\ \hline 2s_{nat} &= n \times (n+1) \end{aligned}$$

da cui segue immediatamente che

$$s_{nat} = \frac{n(n+1)}{2}$$

L'algoritmo che risolve il problema del calcolo della somma dei primi n numeri naturali utilizzando la formula di Gauss è:

```
function snat_gauss(n):integer
var n:integer
begin
    snat_gauss := n*(n+1)/2
end
```

L'algoritmo nella function somma costruisce la soluzione riproducendo la struttura del problema; l'algoritmo in snat_gauss nasce da un'analisi teorica del problema, che consente di esprimere direttamente la sua soluzione. Si noti che il primo algoritmo effettua un numero di addizioni pari a $n - 1$, mentre il secondo effettua sempre due moltiplicazioni e una addizione, indipendentemente dal valore di n.
La somma dei primi n numeri naturali, oltre a essere un semplice problema di teoria dei numeri, consente di risolvere problemi concreti; in questo senso esso rappresenta un modello matematico di base.
Si consideri per esempio il seguente problema.
In una classe, formata da n studenti, c'è l'abitudine tra gli studenti di stringersi la mano ogni mattina: qual è il numero complessivo di strette di mano quotidiane?
Il problema può essere risolto facendo un semplice ragionamento induttivo:
il primo studente ad arrivare non saluta nessuno e quindi se $n = 1$ la soluzione è 0; il secondo saluta il primo, e quindi se $n = 2$ la soluzione è 1; il terzo saluta il primo e il secondo, e quindi se $n = 3$ la soluzione è $1 + 2 = 3$; il quarto saluta il primo, il secondo e il terzo, e quindi se $n = 4$ la soluzione è $1 + 2 + 3 = 6$; in generale, il k-simo studente saluta i k-1 studenti già arrivati e queste strette di mano vanno ad aggiungersi a tutte le precedenti, dando luogo a $1 + 2 + 3 + ... + k - 1$ strette di mano.
La soluzione del problema è quindi la somma dei primi n –1 numeri naturali.

Il problema del calcolo della somma dei numeri naturali pari non maggiori di n `(>2)` è risolto da una semplice variante dell'algoritmo implementato dalla `function somma`:

```
function somma_pari(n): integer
var n: integer
begin
    somma_pari := 0
    for i=2,n,step=2 do
        somma_pari := somma_pari + i
    endfor
end
```

La struttura dei due algoritmi precedenti è utilizzabile per sviluppare algoritmi per la risoluzione di problemi che coinvolgono il calcolo di sommatorie.

ESEMPIO:
sviluppare un algoritmo, organizzato come function, per il calcolo della somma dei quadrati dei primi n numeri naturali:

$$s_{qua} = \sum_{i=1}^{n} i^2 = 1 + 4 + 9 + \ldots + n^2$$

dato di input: n
dato di output: s_{qua}
costrutto ripetitivo: `for`
operazione ripetuta (al generico passo `i`): sommare il valore `i^2` (l'addendo `i`-simo) alla somma degli (`i-1`) quadrati precedenti.

```
function somma_quadrati(n): integer
var n: integer
begin
    somma_quadrati := 0
    for i=1,n do
        somma_quadrati := somma_quadrati + i^2
    endfor
end
```

Anche in questo caso il problema ha una semplice soluzione analitica, data dalla formula

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

L'algoritmo che calcola la soluzione per via analitica è:

```
function somma_quad_dir(n): integer
var n: integer
begin
    somma_quad_dir := n*(n+1)*(2*n+1)/6
end
```

Il numero di operazioni effettuate da questo algoritmo non dipende da n.

ESEMPIO:

sviluppare un algoritmo, organizzato come function, per il calcolo della sommatoria (detta *armonica*):

$$s_{arm} = \sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$$

dato di input: n
dato di output: s_{arm}
costrutto ripetitivo: `for`
operazione ripetuta (al generico passo `i`): sommare l'addendo `1/i` alla somma degli (`i-1`) precedenti.

```
function somma_arm(n): real
var n: integer
begin
    somma_arm := 0.0
    for i=1,n do
        somma_arm := somma_arm + 1.0/float(i)
    endfor
end
```

Si noti l'uso della function primitiva `float(x)` che assume come valore di tipo reale il valore del dato associato alla variabilbile intera `x`. In pratica la `float(x)` consente di cambiare la rappresentazione interna di un dato: se `x` è di tipo intero, `float(x)` è di tipo reale.

È interessante osservare che non esiste una semplice formula coinvolgente solo operazioni aritmetiche (del tipo di quelle degli esempi precedenti) che consenta di esprimere s_{arm} in dipendenza di n.
Ciò significa che l'unica strada pratica per calcolare la sommatoria armonica è quella di utilizzare l'algoritmo in `somma_arm`.

ESEMPIO:
sviluppare un algoritmo, organizzato come function, per il calcolo della sommatoria:

$$s_{alt} = \sum_{i=0}^{n} \frac{(-1)^i}{2i+1} = 1 - \frac{1}{3} + \frac{1}{5} - \ldots$$

dato di input: n
dato di output: s_{alt}
operazione ripetuta (al generico passo `i`): sommare l'addendo `i`-simo alla somma degli (`i-1`) precedenti.

```
function somma_alt(n): real
var n: integer
var addendo, segno: real
```

```
begin
    somma_alt := 1.0
    segno = -1.0
    for i=1,n do
        addendo := 1.0/(2.0*float(i)+1.0)
        somma_alt := somma_alt + segno*addendo
        segno := - segno
    endfor
end
```

Si osservi che il termine $(-1)^i$ indica solo che la sommatoria è a segni alterni. Nell'algoritmo il cambio di segno è ottenuto mediante l'uso opportuno della variabile `segno`.

Il seguente esempio chiarifica l'uso di tale variabile per ottenere il cambiamento del segno ripetuto a ogni passo:

ESEMPIO:

produrre in output la sequenza di n numeri `a -a a -a ....`
dati di input: `n, a`
dato di output: nessuno
costrutto ripetitivo: `for`
operazione ripetuta (al generico passo `i`): cambio del segno, stampa.

```
procedure stampa(in: n,a)
var n, a, segno: integer
begin
    segno := 1
    for i=1,n do
        print segno*a
        segno := -segno
    endfor
end
```

ESEMPIO:

sviluppare un algoritmo, organizzato come function, per il calcolo della media aritmetica m_a di n numeri (forniti sul dispositivo esterno di input).

dati di input: n, gli n numeri
dato di output: m_a
costrutto ripetitivo: `for`
operazione ripetuta (al generico passo `i`): sommare l'`i`-simo numero alla somma degli (`i-1`) precedenti.

```
function media_aritm(n): real
var n: integer
var numero, somma: real
begin
    somma := 0.0
    for i=1,n do
        read numero
        somma := somma + numero
    endfor
    media_aritm := somma/float(n)
end
```

In questo problema i numeri che devono essere sommati sono forniti attraverso il dispositivo esterno di input: al primo passo l'istruzione `read` permette di ottenere il primo numero, al secondo passo di ottenere il secondo numero e così via.

ESEMPIO:

sviluppare un algoritmo, organizzato come function, per il calcolo della sommatoria di potenze (detta *geometrica*):

$$s_{geo} = \sum_{i=0}^{n} x^i = 1 + x + x^2 + \ldots + x^n$$

dati di input: x, n
dato di output: s_{geo}
costrutto ripetitivo: `for`

operazione ripetuta (al generico passo `i`): sommare l'addendo `i`-simo x^i alla somma degli (`i-1`) precedenti.

```
function somma_potenze(x,n): real
var n: integer
var x: real
begin
    somma_potenze := 1.0
    for i=1,n do
        somma_potenze := somma_potenze + x^i
    endfor
end
```

Si osservi che l'istruzione di assegnazione all'interno del ciclo `for` viene eseguita `n` volte. Il calcolo di x^i richiede `i-1` moltiplicazioni di x per sè stesso. Se si tiene conto di queste moltiplicazioni, presenti implicitamente nel testo

del programma, si ha che il numero complessivo di moltiplicazioni è `0+1+2+3 +...+n-1=n*(n-1)/2`.

È possibile sviluppare un algoritmo che richiede un numero inferiore di moltiplicazioni.

L'idea nasce dall'osservazione che al generico passo `i`, quando si deve cioè calcolare x^i, è stata già calcolata, al passo precedente, la quantità x^{i-1}; quindi x^i può essere calcolata con una sola moltiplicazione, poichè $x^i = x \cdot x^{i-1}$. È necessario solo memorizzare il valore della potenza di x al passo precedente; a tale scopo, nell'algoritmo viene utilizzata la variabile `potenza`.

```
function somma_potenze(x,n): real
var n: integer
var x,potenza: real
begin
    somma_potenze := 1.0
    potenza := 1.0
    for i=1,n do
        potenza := potenza*x
        somma_potenze := somma_potenze + potenza
    endfor
end
```

All'inizio del passo `i`, il valore associato a `potenza` è x^{i-1}; tale valore viene modificato dall'istruzione `potenza := potenza*x` che associa a `potenza` la quantità x^i.

Il problema della somma geometrica ha una semplice soluzione analitica, data dalla formula:

$$\sum_{i=0}^{n} x^i = \frac{1-x^{n+1}}{1-x} \quad , \quad x \neq 1$$

La struttura degli algoritmi per il calcolo di sommatorie è utilizzabile anche per sviluppare un algoritmo per la risoluzione di un problema di calcolo di prodotti.

ESEMPIO:

sviluppare un algoritmo, organizzato come function, per il calcolo del fattoriale di un numero n. La definizione di fattoriale è la seguente: $n! = n(n-1)(n-2)\cdots 1$.

dato di input: n
dato di output: $n!$
costrutto ripetitivo: `for`
operazione ripetuta (al generico passo `i`): moltiplicare `i` per il prodotto degli `(i-1)` numeri precedenti.

```
function fattoriale(n): integer
var n: integer
begin
    fattoriale := 1
    for i=1,n do
        fattoriale := fattoriale * i
    endfor
end
```

Si riporta in dettaglio la sequenza computazionale descritta dalla `function fattoriale` per `n=5`:

```
fattoriale:=1*1;   fattoriale:=1*2;   fattoriale:=2*3;
fattoriale:=6*4;   fattoriale:=24*5.
```

La seguente versione ha specifiche completamente congruenti con la definizione di fattoriale di un intero ($0! = 1! = 1$); inoltre la function è di tipo reale, al fine di evitare problemi di overflow sul tipo intero.

```
function fattoriale(n): real
var n: integer
begin
    fattoriale := 1.0
    if n > 1
     then
        for i=2,n do
           fattoriale := fattoriale * float(i)
        endfor
    endif
end
```

L'algoritmo per il calcolo del fattoriale è basato anch'esso sull'idea incrementale. Infatti la costruzione della soluzione avviene per passi successivi e, al termine del generico passo `i` del processo ripetitivo, il valore associato a `fattoriale` è la soluzione del problema del calcolo del fattoriale di `i`.

ESEMPIO:

sviluppare un algoritmo che determina l'appartenenza di un dato a un certo insieme assegnato di *n* dati; questo problema è molto comune in varie applicazioni ed è detto *problema della ricerca* (*search*). Nell'esempio si suppone che l'insieme dei dati sia l'archivio degli *n* nomi dei clienti di una certa azienda e che il dato da ricercare sia un certo nome (detto *chiave* della ricerca). L'archivio viene fornito sul dispositivo esterno di input.

L'idea che è alla base dell'algoritmo consiste nel fare un confronto della chiave in sequenza con ognuno degli elementi dell'insieme, finchè l'esito del confronto è positivo oppure tutti gli elementi dell'insieme sono stati presi in considerazione. Questa strategia di ricerca è una applicazione dell'approccio incrementale ed è detta *ricerca sequenziale*.

dati di input: `nome`, *n*, gli *n* nomi dei clienti (l'archivio)
dato di output: `true`, `false`
costrutto ripetitivo: `repeat`
operazione ripetuta (al generico passo `i`): leggere l'`i`-simo nome di cliente, confrontare l'`i`-simo nome di cliente con `nome`.
predicato di uscita: successo nel confronto oppure completamento dell'esame dell'archivio.

```
function ricerca_seq(nome,n): logical
var nome,nome_cliente: character
var n,i: integer
begin
      i := 0
      repeat
          read nome_cliente
          i := i+1
      until nome=nome_cliente   or i=n
      if nome=nome_cliente
          then
              ricerca_seq := true
          else
              ricerca_seq := false
      endif
end
```

Il predicato di uscita dal ciclo `repeat` è composto da due sentenze semplici (`nome=nome_cliente`, `i=n`) e tiene conto sia del confronto tra chiave ed elemento dell'archivio sia del numero complessivo di elementi dell'insieme che sono stati esaminati.

ESEMPIO:

sviluppare un algoritmo che calcola il numero dei dati, di un insieme assegnato, che sono minori di un certo valore prefissato (detto *soglia*). Nell'esempio si suppone che l'insieme dei dati sia un insieme di n numeri reali fornito sul dispositivo esterno di input.

Anche questo algoritmo è basato sull'idea incrementale.

dati di input: `soglia`, n, gli n numeri
dato di output: `contatore`
costrutto ripetitivo: `for`
operazione ripetuta (al generico passo `i`): leggere l'`i`-simo numero, confrontare l'`i`-simo numero con `soglia`; eventuale incremento di `contatore`.

```
function contatore(soglia,n): integer
var n: integer
var soglia,numero: real
begin
      contatore := 0
      for i=1,n
          read numero
          if numero < soglia
             then
                contatore := contatore+1
          endif
      endfor
end
```

ESEMPIO:

sviluppare un algoritmo che determina l'elemento massimo di un certo insieme assegnato di dati. Si considera un insieme di n numeri reali che viene fornito sul supporto esterno di input.

dati di input: n, gli n numeri
dato di output: `massimo`
costrutto ripetitivo: `for`
operazione ripetuta (al generico passo `i`): leggere l'`i`-simo numero, confrontare l'`i`-simo numero con massimo, se necessario aggiornare il valore di massimo.

L'algoritmo è basato sull'approccio incrementale. La soluzione è costruita attraverso aggiornamenti successivi. Al generico passo `i` del ciclo, il nuovo valore della soluzione è ottenuto confrontando il valore della soluzione al passo precedente con un nuovo numero. Al termine del passo `i` si ha la soluzione del problema del massimo dei primi `i` numeri.

```
function massimo(n): real
var numero: real
var n: integer
begin
      read numero
      massimo := numero
      for i=2,n do
          read numero
          if numero > massimo
             then
                massimo := numero
          endif
      endfor
end
```

Si noti la particolare inizializzazione (`massimo:=numero`); in pieno spirito incrementale, alla variabile `massimo` è associato all'inizio l'unico (il primo) numero considerato; a ogni passo `i` a `massimo` è associato il valore del massimo dei primi `i` numeri; al termine della ripetizione (`i=n`) il valore associato a `massimo` è la soluzione del problema.

L'algoritmo può immediatamente essere modificato per calcolare l'elemento minimo di un certo insieme assegnato di dati. Inoltre è semplice scrivere una versione dell'algoritmo che determini il primo o l'ultimo nome, nell'ordine alfabetico (si tratta dell'elemento minimo e dell'elemento massimo, rispettivamente), di un certo archivio di n nomi.

Un aspetto importante del progetto di un algoritmo è l'individuazione delle specifiche dell'algoritmo stesso. Un algoritmo è un procedimento di risoluzione di un problema: è utile che sia accompagnato da precise informazioni sull'insieme dei dati di input a cui è applicabile correttamente.

Per gli algoritmi sviluppati finora, questo può essere fatto semplicemente. Per esempio la function `massimo` richiede $n \geq 1$, così come la function per la media.

È possibile rendere più *robusto* un programma introducendo un controllo della congruenza tra i dati forniti effettivamente in input e le specifiche, come mostrato nella seguente:

```
function minimo(n): real
var numero: real
var n: integer
begin
      if n > 0
         then
            read numero
            minimo := numero
```

```
            for i=2,n do
              read numero
              if numero < minimo
                then
                  minimo := numero
              endif
            endfor
          else
              print 'errore in input'
      endif
end
```

ESEMPIO:

sviluppare un algoritmo, organizzato come function, per la valutazione in un punto x della funzione:

$$f(x)=\begin{cases} 0 & ,x\leq 1 \\ x-1 & ,1<x\leq 2 \\ 1 & ,2<x \end{cases}$$

dato di input: x
dato di output: $f(x)$
costrutto ripetitivo: *nessuno*

```
function mia_f(x):real
var x:real
begin
 if x <= 1.0
  then
    mia_f := 0.0
  elseif x <= 2.0
    then
     mia_f := x-1.0
    else
     mia_f := 1.0
   endif
 endif
end
```

ESEMPIO:

sviluppare un algoritmo, organizzato come procedura, per la visualizzazione dei primi n termini della successione:

$$\left\{\frac{x^k}{k!}\right\}_{k=0}^{n} = 1, x, \frac{x^2}{2}, \frac{x^3}{6}, \ldots, \frac{x^n}{n!}$$

dati di input: x, n
dato di output: visualizzazione di tutti gli n termini
costrutto ripetitivo: `for`
operazione ripetuta (al generico passo `k`): calcolare il `k`-simo termine della successione e visualizzarlo

```
procedure successione_1(in: x,n)
var x: real
var n: integer
var fattoriale: real function
begin
 for k=0,n do
    print x^k/fattoriale(k)
 endfor
end
```

Poichè sia la valutazione dell'espressione `x^k` sia quella di `fattoriale(k)` richiedono ognuna `k`-1 moltiplicazioni, il costo complessivo dell'esecuzione della procedura è di $n\,(n-1)$ moltiplicazioni.

La seguente procedura genera il medesimo risultato con un numero inferiore di moltiplicazioni:

```
procedure successione_2(in: x,n)
var x: real
var n: integer
var elem: real
begin
 elem := 1.0
 print elem
 for k=1,n do
    elem := elem*x/float(k)
    print elem
 endfor
end
```

L'algoritmo si basa sull'osservazione che

$$x^k = x \cdot x^{k-1} \qquad \text{e} \qquad k! = k\,(k-1)!$$

e quindi

$$\frac{x^k}{k!} = \frac{x}{k} \cdot \frac{x^{k-1}}{(k-1)!}$$

Si noti inoltre che al passo k la quantità

$$\frac{x^{k-1}}{(k-1)!}$$

è già stata calcolata (al passo precedente). Il numero di moltiplicazioni della procedura `successione_2` è $2n$.

ESEMPIO:

sviluppare un algoritmo, organizzato come function, per calcolare il valore approssimato della funzione e^x mediante il seguente sviluppo in serie (detto *di McLaurin*) arrestato al termine n-simo

$$e^x \approx S_n(x) = \sum_{k=0}^{n} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \ldots + \frac{x^n}{n!}$$

dati di input: x, n
dato di output: S_n
costrutto ripetitivo: `for`
operazione ripetuta (al generico passo `k`): calcolare il `k`-simo elemento della serie e sommarlo al valore precedente della somma.

```
function mia_exp(x,n): real
var x: real
var n: integer
var elem: real
begin
 mia_exp := 1.0
 elem := 1.0
 for k=1,n do
    elem := elem*x/float(k)
    mia_exp := mia_exp+elem
 endfor
end
```

L'errore relativo di approssimazione è:

$$\frac{\left|e^x - S_n(x)\right|}{e^x}$$

e dipende da n, come mostrato dai grafici in Fig. 3.8.

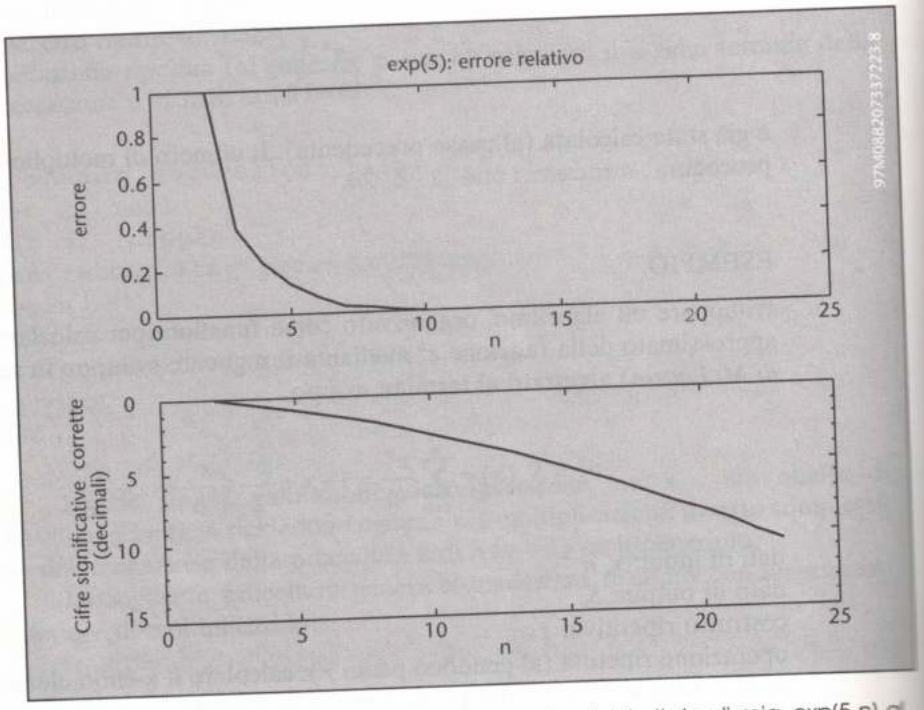


Fig. 3.8: errore relativo e numero di cifre corrette del risultato di mia_exp(5,n) al variare di n

Se si effettuasse una chiamata alla function con un valore negativo del parametro x, per esempio $x = -14$, n=35, si otterrebbe come risultato (doppia precisione) il valore -0.3547, mentre $e^{-14} = 0.83153 \times 10^{-6}$. In tal caso, l'errore di approssimazione non diminuisce al crescere di n.

Il motivo per cui viene generato un risultato errato sta nel fatto che l'algoritmo in `mia_exp` è *instabile* per valore negativi di x. In breve, l'instabilità deriva dalla sequenza computazionale descritta dall'algoritmo. Si tratta di una sequenza alternata di addizioni e sottrazioni che causa il fenomeno di *cancellazione* di cifre significative. Per l'approfondimento dell'analisi di stabilità di un algoritmo si rinvia a un testo di Calcolo Numerico.

Il seguente algoritmo è *stabile*:

```
function mia_exp(x,n): real
var x: real
var n: integer
var elem,modulo_x: real
begin
 modulo_x := abs(x)
 mia_exp := 1.0
 elem := 1.0
 for k=1,n do
    elem := elem*modulo_x/float(k)
    mia_exp := mia_exp+elem
 endfor
 if x < 0.0
  then
     mia_exp := 1.0/mia_exp
 endif
end
```

Tale algoritmo si basa sull'osservazione che:

$$e^{-x} = \frac{1}{e^x}$$

e la stabilità discende dal fatto che non si effettuano sottrazioni se x è positivo.

ESEMPIO:

sviluppare un algoritmo, organizzato come function, per calcolare il valore approssimato della funzione e^x mediante lo sviluppo in serie:

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$

arrestato in modo tale da ottenere la massima accuratezza possibile nel sistema aritmetico che si utilizza.

In questo esempio non viene fornito in input il numero n dei termini della serie da sommare. Il numero dei termini da sommare deve essere determinato dall'algoritmo e deve garantire la massima accuratezza. La difficoltà è evidente. Se si considera un numero troppo piccolo di termini, non si avrà la massima accuratezza; se si considera un numero troppo grande di termini si avrà un algoritmo che effettua troppe operazioni. Il problema centrale consiste nell'utilizzare un *criterio di arresto* per un algoritmo iterativo che approssima un processo infinito (come è quello che definisce la somma di una

serie). Tale problema ha una elegante soluzione. Infatti, poichè l'algoritmo viene eseguito nel sistema aritmetico a precisione finita di un calcolatore e osservando che gli elementi della serie al crescere dell'indice i sono sempre più piccoli in modulo, si può concludere che esiste certamente un valore dell'indice, sia N, a partire dal quale la sequenza delle operazioni dell'algoritmo non modifica più il valore della variabile a cui è associato il risultato. Questo fatto, a prima vista sorprendente, ha una semplice spiegazione in termini di epsilon macchina.

Sia:

$$S_N = \sum_{k=0}^{N} \frac{x^k}{k!} \; ;$$

se:

$$\frac{x^{N+1}}{(N+1)!} < \varepsilon_M \cdot S_N$$

allora si ha che:

$$S_N \oplus \frac{x^{N+1}}{(N+1)!} \equiv S_N \; .$$

dati di input: x
dato di output: S_N
costrutto ripetitivo: `repeat`
operazione ripetuta (al generico passo `k`): calcolare il `k`-simo elemento della serie e sommarlo al valore precedente della somma.
predicato di uscita: l'addendo è minore dell'epsilon macchina moltiplicato per il valore precedente della somma.

```
function esponenziale(x): real
var x: real
var criterio: logical
var n,k: integer
var elem, modulo_x: real
begin
 modulo_x := abs(x)
 k := 1
 esponenziale := 1.0
 elem := 1.0
 repeat
    elem := elem*modulo_x/float(k)
    esponenziale := esponenziale+elem
    criterio := elem < epsilon_rel(esponenziale)
```

```
      k := k+1
    until criterio
    if x < 0.0
      then
        esponenziale := 1.0/esponenziale
    endif
  end
```

La `function epsilon_rel`, che consideriamo primitiva, determina il valore dell'epsilon relativo al suo argomento.

ESEMPIO:

sviluppare un algoritmo, organizzato come function, per il calcolo del massimo comun divisore.

Il problema del calcolo del massimo comun divisore di due numeri è noto fin dall'antichità: dati due numeri interi positivi a, b trovare il più grande numero intero positivo che divida (esattamente, cioè con resto nullo) entrambi i numeri a, b.

La soluzione può essere calcolata seguendo il procedimento proposto da Euclide (Elementi, libro 7, proposizioni 1,2), che è considerato il primo algoritmo sviluppato e analizzato, nella storia della matematica.

L'idea che è alla base dell'algoritmo discende dalle seguenti proprietà del massimo comun divisore (mcd): Se $a = b$, il mcd è a. Se $a > b$, allora a, b e $a - b$ hanno lo stesso mcd. Inoltre, dividendo a per b si ottiene un quoziente q e un resto r, cioè:

$$a = qb + r$$

se $r = 0$, allora q è il mcd; se $r \neq 0$, si ha che a, b e r hanno lo stesso mcd[5].

Il problema del calcolo del mcd di a, b è equivalente al problema del calcolo del mcd di b, $a - b$ oppure del mcd di b, r.

Come si può sfruttare in modo vantaggioso queste proprietà al fine di sviluppare un algoritmo?

Il problema non è cambiato, in quanto è necessario calcolare ancora un mcd; solo che b e r (oppure b e $a - b$) sono entrambi più piccoli di a.

L'osservazione fondamentale è che a partire dalla nuova coppia di numeri è possibile applicare nuovamente le proprietà precedenti.

Seguiremo l'idea originaria di Euclide, che sceglie b e r come nuova coppia.

[5] Un qualunque numero che divide sia a sia b, deve dividere anche $a - qb = r$ e inoltre un qualunque numero che divide sia b sia r, deve dividere anche $qb + r = a$.

L'algoritmo procede per passi; a ogni passo si considera un problema che ha la stessa soluzione del problema al passo precedente e quindi del problema di partenza (problemi *equivalenti*); questa *catena* di problemi equivalenti termina con un problema di risoluzione banale, cioè il caso $r=0$. In virtù dell'equivalenza, la soluzione del problema banale è la soluzione del problema.

Ciò si suole sintetizzare dicendo che vengono generate successivamente varie *istanze* dello stesso problema con numeri via via più piccoli.

Il processo ripetitivo termina quando il resto è nullo.

L'idea di risolvere un problema trasformandolo ripetutamente in un problema equivalente più *semplice* è di portata generale ed è alla base del progetto di molti algoritmi.

dati di input: *a*, *b*
dato di output: `mcd`
costrutto ripetitivo: `repeat`
operazione ripetuta (al generico passo `i`): calcolare il resto, eventualmente considerare la nuova coppia
predicato di uscita: il resto è nullo.

```
function mcd(a,b): integer
var a,b, dividendo,divisore,resto: integer
var max, min: integer function
begin
      dividendo := max(a,b)
      divisore := min(a,b)
      repeat
          resto := mod(dividendo,divisore)
          dividendo := divisore
          divisore := resto
      until resto = 0
      mcd := dividendo
end

function max(x,y): integer
var x,y: integer
begin
      if x > y
         then
            max := x
         else
            max := y
      endif
end
```

La organizzazione della `function min(x,y)` è analoga alla precedente.

> Nell'approccio incrementale la soluzione del problema viene costruita attraverso la risoluzione di una sequenza di istanze via via *più complesse* dello stesso problema, a partire dal problema banale la cui soluzione è nota.
> Al contrario l'algoritmo per il calcolo del mcd procede per così dire nel verso opposto, cioè utilizzando una sequenza di istanze via via *più semplici* del problema, ma tutte *equivalenti*.
> Si noti che nessuno di questi problemi viene risolto. Ognuno serve solo a generare il successivo problema equivalente, fino al problema la cui soluzione è nota.

ESERCIZI:

1) Verificare attraverso un algoritmo che la somma dei primi n numeri dispari è uguale a n^2.

2) Verificare attraverso un algoritmo che la somma dei cubi dei primi n numeri naturali è uguale al quadrato della somma dei primi n numeri naturali.

3) Verificare attraverso un algoritmo che la somma delle prime n potenze di due

$(1+2+4+8+\ldots+2^{n-1})$ è uguale a 2^n-1.

4) Verificare attraverso un algoritmo che:

$$1\cdot 2+2\cdot 3+3\cdot 4+\ldots+n\cdot(n+1)=\frac{n(n+1)(n+2)}{3}.$$

5) Sviluppare un algorimo per il calcolo della somma dei primi n numeri naturali pari.

6) Verificare attraverso un algoritmo che:

$$\frac{1}{1\cdot 2}+\frac{1}{2\cdot 3}+\frac{1}{3\cdot 4}+\ldots+\frac{1}{n\cdot(n+1)}=\frac{n}{(n+1)}.$$

7) Dato un numero intero n, le sue cifre possono essere ottenute in sequenza, a partire da quella che denota le unità, come il resto della divisione intera per 10. Per esempio sia n=314, allora la prima cifra è il resto di n/10,

cioè 314/10, che è 4, la seconda il resto di (n/10)/10, cioè 31/10, che è 1, la terza, il resto di ((n/10)/10)/10, cioè 3/10, che è 3. Usando la function primitiva `mod`, sviluppare una procedura che visualizzi le cifre di un qualunque numero intero.

8) Dato un numero intero n, le sue cifre nella rappresentazione in base b possono essere ottenute in sequenza, a partire da quella che denota le unità, come il resto della divisione intera per b. Per esempio sia n=314 e b=8, allora la prima cifra è il resto di n/8, cioè 2, la seconda il resto di 39/8, cioè 7, la terza, il resto di 4/8, cioè 4. Usando la function primitiva `mod`, sviluppare una procedura che visualizzi le cifre di un qualunque numero intero in una qualunque base.

9) Sviluppare un algoritmo che conti quanti elementi di un dato insieme di numeri reali appartengono all'unione di due intervalli assegnati.

10) Sviluppare un algoritmo che conti quanti elementi di un dato insieme di numeri interi pari sono di almeno quattro cifre.

11) Sviluppare un algoritmo, organizzato come function, per il calcolo della media geometrica di n numeri (forniti sul dispositivo esterno di input).

La media geometrica di $x_1, x_2, \ldots, x_n$ è $m_g = \sqrt[n]{x_1 x_2 \cdots x_n}$.

12) Sviluppare algoritmi per il calcolo di exp, sin, cos, basati sullo sviluppo in serie di Taylor. Studiare la stabilità / instabilità.

13) Sviluppare un algoritmo per il calcolo del mcd trasformando ad ogni passo il problema del calcolo del mcd di a, b nel problema del calcolo del mcd di b, $a - b$. Confrontare il numero di passi effettuato da questo algoritmo e dalla function `mcd`.

14) Utilizzare la function `distanza_origine` (par 3.1.4) per sviluppare una function `distanza(x,y,xx,yy)` che calcola la distanza tra due punti di coordinate `(x,y)` e `(xx,yy)`.

15) Nel par. 3.1.5 è stato presentata la porzione di algoritmo

```
a:= ...
for i=1,1000
    a:=a/2.
endfor
```

Senza eseguire tale algoritmo, dire quale risultato si avrebbe, per qualunque valore dato in `a:= ...`, se il calcolatore ha un sistema aritmetico

standard IEEE in singola precisione. Implementare l'algoritmo in un linguaggio di programmazione ed eseguire su un calcolatore più volte, variando il valore iniziale di a. Commentare il risultato.

16) Innestando opportunamente due cicli for, scrivere un algoritmo che visualizzi in output:

```
(1 1) (2 1) (2 2) (3 1) (3 2) (3 3) (4 1) (4 2) (4 3) (4 4)
```

e un altro che visualizzi:

```
(1 1) (1 2) (1 3) (1 4) (2 2) (2 3) (2 4) (3 3) (3 4) (4 4).
```

3.2 Il tipo strutturato array

I costrutti ripetitivi consentono di descrivere in modo semplice (cioè scrivendo un breve testo di programma) una sequenza computazionale arbitrariamente lunga. È naturale chiedersi se è possibile indicare in modo semplice una quantità arbitraria di dati.

Nei problemi proposti nei capitoli precedenti, il numero dei dati poteva essere arbitrariamente grande, come nel caso del calcolo della media, della ricerca in un insieme, etc.; tuttavia negli algoritmi che abbiamo finora sviluppato, *l'insieme dei dati non era indicato attraverso un singolo nome*; gli elementi dell'insieme non erano contemporaneamente presenti nella memoria, ma venivano immessi dall'esterno (con l'istruzione read), utilizzati ed eliminati, uno alla volta. Questo approccio non ha la flessibilità necessaria per risolvere moltissimi problemi concreti.

Inoltre, nelle applicazioni i dati sono spesso legati tra loro da relazioni logiche che sono parte integrante del problema. Come vedremo nel seguito, il progetto di un algoritmo è intimamente collegato all'individuazione delle relazioni tra i dati del problema e alla capacità di sfruttare tali relazioni per costruire la soluzione. Pertanto, un linguaggio di programmazione deve avere la possibilità di descrivere l'organizzazione dei dati del problema. Questo obiettivo viene raggiunto utilizzando i *tipi strutturati*.

3.2.1 Strutture dati e variabili strutturate

Nei linguaggi naturali esistono i cosiddetti *nomi collettivi*, che indicano insiemi di oggetti, come coppia di sposi, vettori di numeri, cassettiera, mazzo di carte, elenco telefonico, pila di piatti, fila di persone, albero genealogico, ...

In tutti questi esempi, oltre alla denotazione formale di un insieme con un nome, è possibile osservare che viene indicata anche l'esistenza di partico-

lari relazioni tra gli oggetti dell'insieme. Questo fatto si suole sintetizzare dicendo che vi è una *struttura* nell'insieme degli oggetti.

Inoltre nei linguaggi naturali è anche possibile indicare singolarmente gli oggetti dell'insieme (per esempio la terza carta del mazzo), in un modo che dipende dalla struttura.

Nei problemi applicativi si possono presentare varie forme di relazioni tra i dati.

In una *fila* di persone, per esempio, vi sono proprietà strutturali tra gli elementi che la compongono e che prescindono dal particolare valore dell'elemento (il nome della persona, per esempio).

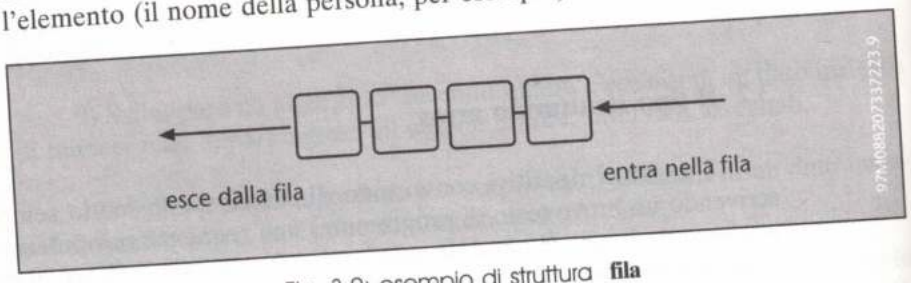


Fig. 3.9: esempio di struttura **fila**

Tali proprietà possono essere sintetizzate nelle risposte alle domande: qual è il primo elemento della fila? qual è l'ultimo elemento? dove si inserisce un nuovo elemento? qual è l'elemento che precede e quello che segue un dato elemento della fila?

In un *albero* genealogico gli elementi sono in una relazione di tipo padre-figli, che consente una organizzazione degli elementi per livelli di generazioni successive.

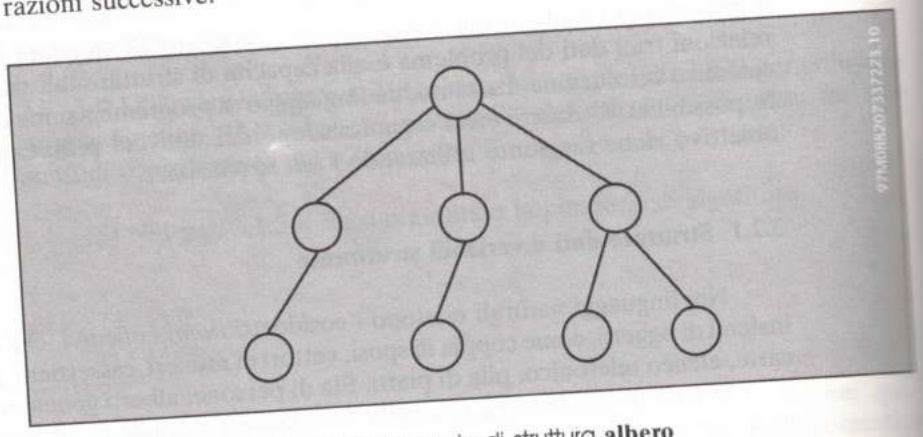


Fig. 3.10: esempio di struttura **albero**

Quali sono gli elementi di una certa generazione? quali sono gli elementi che non hanno figli? Queste domande si riferiscono ad alcune proprietà strutturali che caratterizzano gli elementi di un albero genealogico. È interessante notare che le relazioni che individuano un albero genealogico possono essere utilizzate anche in un contesto diverso, per esempio per caratterizzare la gerarchia di un comando militare o per indicare l'organizzazione di una azienda, delle sue consociate e delle sue filiali.

In moltissime applicazioni i dati sono naturalmente organizzati in *tabelle*. La forma più semplice di tabella è quella costituita da una unica *riga* di caselle.

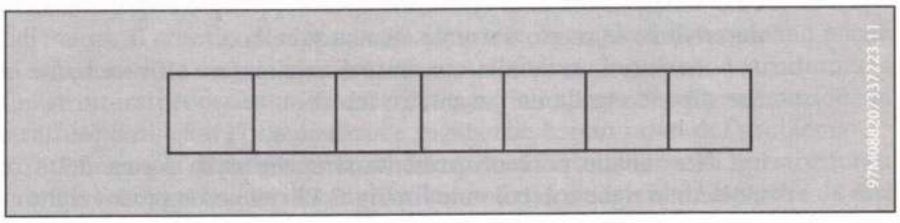


Fig. 3.11: esempio di struttura **tabella**

Ogni casella contiene un dato. Tutti i dati sono dello stesso tipo; quindi si ha una tabella di numeri interi, una tabella di caratteri, etc.

La proprietà strutturale fondamentale della tabella (a una riga) è il fatto che ogni casella (elemento della tabella) è individuabile univocamente attraverso un numero intero, detto *indice*.

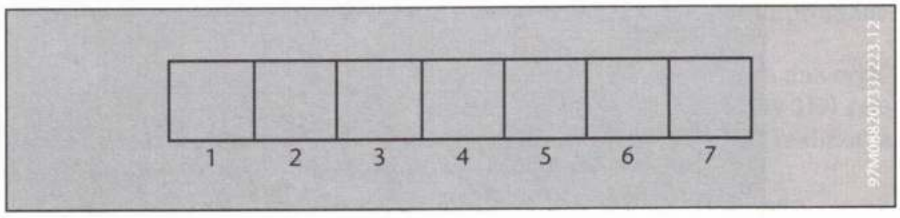


Fig. 3.12: indici di una tabella

Si osservi che in una tabella *non si possono scambiare due caselle*, anche se *si può scambiare il contenuto* di due caselle. Inoltre non si può distruggere alcuna casella e in generale non si può modificare il numero delle caselle.

È sorprendentemente vasta la varietà delle situazioni concrete in cui gli oggetti sono nella relazione descritta dalla tabella (a una riga).

Le case di una strada sono in questa relazione; ogni casa è individuata da un indice progressivo (il numero civico), non è possibile mettere una casa

al posto di un'altra (anche se i traslochi del contenuto sono possibili) e in definitiva anche se è possibile distruggere una casa la strada non si accorcia!

Le camere di un corridoio d'albergo, i cassetti di una scrivania o le ante di un armadio sono anch'essi esempi di elementi di una struttura a tabella.

La medesima struttura si ritrova in oggetti astratti della matematica, come le n-ple (per es. la coppia, la terna,...) definite come gli elementi del prodotto cartesiano tra insiemi.

Un rigo di un testo è un altro esempio di tabella, in cui ogni casella contiene un carattere.

Una sequenza di eventi, come i risultati di un esperimento o di una misurazione ripetuta nel tempo o in generale un insieme finito di valori di una funzione di una variabile (*valori campionati* di una funzione), è descrivibile in modo naturale da una tabella.

In sintesi, la tabella consente di organizzare efficacemente informazioni che dipendono da un parametro intero.

Le tabelle possono presentarsi anche nella forma detta rettangolare, cioè con n righe e m colonne (in Fig. 3.13 vi è l'esempio a 4 righe e 7 colonne).

Ogni elemento della tabella è individuabile attraverso due indici, l'*indice di riga* e l'*indice di colonna*.

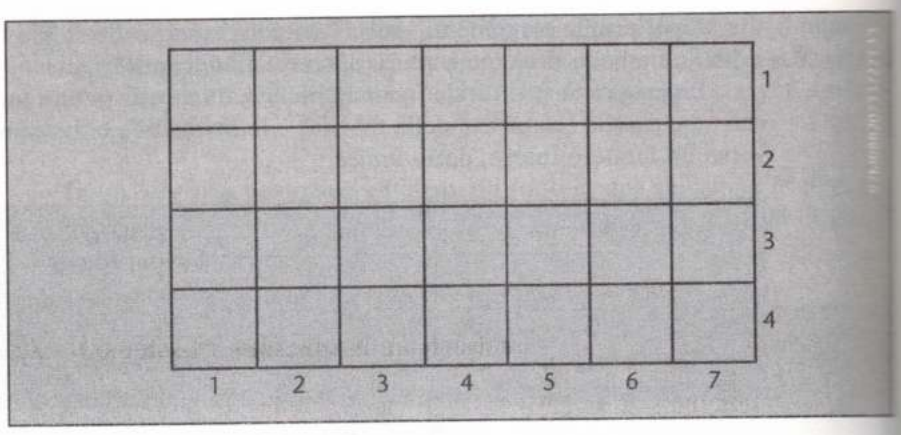


Fig. 3.13: esempio di struttura **tabella rettangolare**

È abbastanza semplice fornire esempi di applicazioni in cui i dati sono organizzati come tabella rettangolare: i caratteri di una pagina di testo, le poltrone di un teatro, le camere di un albergo (i due indici sono il numero del piano e il numero all'interno del corridoio del piano), gli elementi di una matrice, i valori campionati di una funzione di due variabili. Un altro esempio di tabella è lo schermo del calcolatore, la cui risoluzione (per esempio 128

X 1024) è proprio il numero di righe e di colonne degli elementi (pixel) che possono essere illuminati.

In sintesi, la tabella rettangolare consente di organizzare efficacemente informazioni che dipendono da due parametri interi.

In un algoritmo è spesso necessario rappresentare le strutture che esistono nell'insieme dei dati, in modo che le relazioni strutturali possono essere utilizzate per risolvere il problema.

I linguaggi di programmazione consentono di denotare con un nome un insieme di dati strutturati mediante le *variabili strutturate*.

Come nel caso delle variabili scalari, le variabili strutturate sono organizzate in tipi, detti *tipi strutturati*. Un tipo strutturato individua un insieme di valori, un criterio di rappresentazione dei valori e un insieme di operazioni su tali valori. Il criterio di rappresentazione e le operazioni determinano anche le relazioni tra i singoli elementi che costituiscono la variabile strutturata. I tipi strutturati consentono di descrivere nel linguaggio di programmazione le strutture dati, cioè l'organizzazone logica che hanno i dati del problema.

I linguaggi di programmazione hanno un certo numero di tipi strutturati primitivi che realizzano alcune strutture di dati fondamentali; a partire da essi è possibile costruire altri tipi strutturati e descrivere altre strutture di dati. Il numero, le proprietà dei tipi strutturati primitivi e le modalità di realizzazione di nuove strutture differiscono abbastanza sensibilmente da linguaggio a linguaggio e costituiscono uno dei criteri per selezionare il linguaggio più adatto alla descrizione dell'algoritmo per uno specifico problema applicativo.

3.2.2 Proprietà di base del tipo array

Il tipo strutturato primitivo più comune nei vari linguaggi di programmazione è l'*array*.

L'array consente di descrivere un insieme di dati che presenta una organizzazione a tabella; in particolare l'array a una dimensione (array 1D) realizza la tabella a una riga e l'array a due dimensioni (array 2D) realizza la tabella rettangolare.

Le proprietà che caratterizzazano il tipo strutturato array sono:

- ogni elemento di una variabile di tipo array è individuato dal nome della variabile seguito in parentesi da un indice (array 1D) o da due indici (array 2D)[6];
- l'insieme dei valori assumibili dagli indici dell'array (il range degli indici), è detto *shape* dell'array; lo shape è fissato al momento della dichiarazione e, in P-like, non è modificabile;

[6] In molti linguaggi esistono array a più di due dimensioni.

- il numero complessivo di elementi (detto *size*) di una variabile di tipo array è fissato al momento della dichiarazione e, in P-like, non è modificabile;
- tutti gli elementi di una variabile di tipo array sono variabili dello stesso tipo (anche strutturato).

La dichiarazione di tipo di una variabile strutturata specifica il tipo strutturato e il tipo degli elementi.

In P-like la dichiarazione di una variabile di tipo strutturato array è la seguente:

```
var <variabile>: array(<shape>) of <tipo elementi>
```

ESEMPIO:

```
var inflazione_mensile: array(1..12) of real
var temperatura_oraria: array(1..24) of real
var psi: array (-5..7) of integer
var rigo_stampa: array(1..80) of character
var pagina_stampa: array(1..20,1..80) of character
var teta: array(0..3,-1..1) of real
```

Si osservi che il *size* è determinato dallo *shape*, cioè dal range di variabilità dei valori degli indici. Per esempio, `inflazione_mensile` è un array 1D di 12 elementi (size 12) di tipo reale, `psi` è un array 1D di 13 elementi (size 13) di tipo reale, `teta` è un array 2D di 12 elementi (size 12) di tipo reale. Associare un valore a una variabile strutturata significa associare un valore a ogni suo elemento. La figura mostra un esempio di valore della variabile `inflazione_mensile`,

.04	.04	.05	.05	.03	.02	.04	.03	.04	.06	.05	.05
1	2	3	4	5	6	7	8	9	10	11	12

97M0882073372231 13a

in cui si osserva che il valore di `inflazione_mensile(5)` è `0.03` (cioè una inflazione del 3%), quello di `inflazione_mensile(11)` è `0.05` e così via.

Le assegnazioni

```
inflazione_mensile(1):=.05
inflazione_mensile(8):=inflazione_mensile(8)+0.01
i:=10
inflazione_mensile(i):=(inflazione_mensile(i+1)+ .....
                                inflazione_mensile(i-1))/2.
```

generano il seguente valore di `inflazione_mensile`

.05	.04	.05	.05	.03	.02	.04	.04	.03	.04	.05	.04
1	2	3	4	5	6	7	8	9	10	11	12

9788820737223.13b

ESEMPIO:

sviluppare una function per il calcolo della media aritmetica dei valori di un array 1D `a` di tipo reale

dati di input: l'array `a`, il size `n`
dato di output: `media_array`
costrutto ripetitivo: `for`
operazione ripetuta (al generico passo `i`): sommare l'`i`-simo elemento dell'array alla somma degli `(i-1)` precedenti.

```
function media_array(a,n):real
var n:integer
var a:array(1..n) of real
begin
    media_array := 0.0
    for i=1,n do
        media_array := media_array+a(i)
    endfor
    media_array := media_array/float(n)
end
```

Si noti che si tratta di una semplice modifica della `function media_aritm`, sviluppata nel paragrafo 3.1.6; diversamente dalla `function media_aritm`, i numeri da sommare, cioè gli elementi di `a`, sono parametri formali della `function media_array`.

ESEMPIO:

sviluppare un programma per il calcolo della media dei valori dell'inflazione mensile e della temperatura oraria, forniti su dispositivo esterno di input.

dati di input: 12 valori dell'inflazione, 24 valori della temperatura
dato di output: `media_inflazione, media_temperatura`
chiamare due volte la `function media_array`

```
program media_inf_e_temp
var inflazione_mensile:array(1..12) of real
var temperatura_oraria:array(1..24) of real
var media_array:real function
begin
    for i=1,12 do
        read inflazione_mensile(i)
    endfor
    for i=1,24 do
        read temperatura_oraria(i)
    endfor
    print media_array(inflazione_mensile,12)
    print media_array(temperatura_oraria,24)
end
```

ESEMPIO:

sviluppare una function per il calcolo della varianza di un insieme di numeri. L'insieme è rappresentato mediante un array 1D di tipo reale. La varianza di n numeri $a_1, a_2, \ldots, a_n$ è la quantità

$$v = \frac{1}{n-1}\sum_{i=1}^{n}(a_i - \bar{a})^2$$

dove $\bar{a}$ è la media aritmetica dei numeri.

dati di input: l'array `a`, il size n
dato di output: v
costrutto ripetitivo: `for`
operazione ripetuta (al generico passo `i`): sommare l'`i`-simo addendo alla somma degli `(i-1)` precedenti.

```
function varianza_array(a,n): real
var n: integer
var a: array(1..n) of real
var media: real
var media_array: real function
begin
    varianza_array := -1.0
    if n > 1
       then
          varianza_array := 0.0
          media := media_array(a,n)
          for i=1,n do
```

```
        varianza_array := varianza_array+(a(i)-media)^2
      endfor
      varianza_array := varianza_array/float(n-1)
    endif
end
```

Si noti che la function restituisce un risultato negativo nel caso in cui $n = 1$, ovvero la varianza non è definita.

3.2.3 Algoritmi elementari con array

ESEMPIO:

sviluppare una procedura che a partire da un array 1D `a` di `n` componenti generi l'array 1D `diff`, di `n-1` componenti, tale che il valore di `diff(i)` è il valore di `a(i+1)-a(i)`. Tale array è detto *array delle differenze* (in avanti).

dati di input: l'array `a`, il size `n`
dato di output: l'array `diff`
costrutto ripetitivo: `for`
operazione ripetuta (al generico passo `i`): calcolare la differenza tra l'`(i+1)`-simo e l'`i`-simo elemento dell'array `a`.

```
procedure differenze(in:a,n; out:diff)
var n: integer
var a: array(1..n) of real
var    diff: array(1..n-1) of real
begin
      for i=1,n-1 do
        diff(i) := a(i+1)-a(i)
      endfor
end
```

ESEMPIO:

sviluppare un algoritmo, organizzato come function, per il calcolo della sommatoria[7]:

$$\sum_{i=0}^{n} a_i \cos\left(\frac{i2\pi}{n}\right)$$

[7] Si tratta di un problema molto importante del Calcolo Scientifico, che può essere risolto con algoritmi più avanzati e più efficienti di quello presentato (algoritmi FFT).

dati di input: n, l'array contenente gli $n+1$ coefficienti a_i
dato di output: `somma_cos`
costrutto ripetitivo: `for`
operazione ripetuta (al generico passo `i`): sommare l'addendo `i`-simo $a_i \cos\left(\frac{i2\pi}{n}\right)$ alla somma degli (`i-1`) precedenti.

```
function somma_cos(a,n): real
var n: integer
var a: array(0..n) of real
var fattore: real
const pi_greco = 3.1415926
begin
    somma_cos := a(0)
    fattore := 2.0*pi_greco/float(n)
    for i=1,n do
       somma_cos := somma_cos+a(i)*cos(float(i)*fattore)
    endfor
end
```

ESEMPIO:

sviluppare una procedura per il calcolo del massimo (e del suo indice) dei valori di un array 1D.

Si utilizza lo stesso approccio usato per la function `massimo(n)`, sviluppata nel par. 3.1.6.

dati di input: l'array `a`, il size `n`
dato di output: `max_array`, l'indice `i_max`
costrutto ripetitivo: `for`
operazione ripetuta (al generico passo `i`): confrontare l'`i`-simo elemento dell'array con il massimo degli (`i-1`) precedenti, eventuale modifica dei valori di `max_array`, `i_max`.

```
procedure max_val_ind(in:a,n; out:max_array,i_max)
var a: array(1..n) of real
var n, i_max: integer
var max_array: real
begin
      max_array := a(1)
      i_max := 1
      for i=2,n do
         if a(i) > max_array
            then
```

```
                max_array := a(i)
                i_max := i
            endif
        endfor
end
```

L'algoritmo effettua un confronto tra `max_array` e ogni elemento dell'array: il numero totale dei confronti è dato dal *size* dell'array.

Nella `procedure max_val_ind` gli elementi dell'array `a` sono considerati in sequenza, a partire da quello con indice 1, via via fino all'indice `n`. È facile convincersi che qualunque altro modo di considerare in sequenza tutti gli elementi dell'array (anche quello *a caso*) genera un algoritmo corretto.

Per ogni algoritmo del par 3.1.6 in cui i dati di input sono insiemi di dati forniti sul dispositivo esterno di input, è semplice sviluppare una versione dell'algoritmo in cui i dati di input sono organizzati in un array 1D, fornito come parametro.

Per esempio, la function per la ricerca sequenziale (`ricerca_seq(nome,n)` del par. 3.1.6) può essere modificata in:

```
function appartiene_array (x,a,n): logical
var x: character
var a: array(1..n) of character
var n,i: integer
begin
      i := 0
      repeat
          i := i+1
      until x=a(i)   or i=n
      if x=a(i)
          then
              appartiene_array := true
          else
              appartiene_array := false
      endif
end
```

Nel seguito faremo uso, senza svilupparla esplicitamente, di una versione di questa function che agisce su un array di tipo reale.

La costruzione di algoritmi che operano su dati di tipo array è basata sull'individuazione di una strategia efficace di generazione dei valori degli indici, cioè sulla strategia di *accesso* agli elementi dell'array.

ESEMPIO:

sviluppare una function per il calcolo del massimo dei valori di un array 2D.

Come nel caso 1D, il calcolo del massimo in array 2D richiede di effettuare un confonto tra una variabile a cui è associato ad ogni passo il massimo relativo (fino a quel passo) e ogni elemento dell'array: il numero totale dei confronti è dato dal *size* dell'array 2D. L'algoritmo deve prevedere uno scheletro di controllo che generi tutte le coppie di indici degli elementi dell'array.

Nella `function max_2D` gli indici sono generati in modo tale da considerare nell'ordine il primo elemento della prima riga, il secondo elemento della prima riga, ..., l'ultimo elemento della prima riga, e così via riga dopo riga; questa sequenza è detta *accesso per righe*.

dati di input: l'array `a`, il numero di righe `n`, il numero di colonne `m`
dato di output: `max_2D`
costrutto ripetitivo: `for`
operazione ripetuta (al generico passo `i`): accedere alla riga `i`-sima dell'array.

Si noti che accedere alla riga `i`-sima dell'array (cioè a tutti gli elementi della riga) significa generare tutti i valori degli indici di colonna (richiede un altro ciclo for innestato).

```
function max_2D(a,n,m): real
var n,m: integer
var a: array(1..n,1..m) of real
begin
   max_2D := a(1,1)
   for i=1,n do
      for j=1,m do
         if a(i,j) > max_2D
            then
               max_2D := a(i,j)
         endif
      endfor
   endfor
end
```

Lo scheletro di controllo di questo algoritmo, che consiste di due cicli `for` innestati, determina l'esecuzione di `n*m` confronti.

I due esempi che seguono si riferiscono a due problemi classici della teoria degli insiemi, l'unione e l'intersezione di due insiemi. Brevemente, siano $A=\{2,4,6,3\}$ e $B=\{4,3,1\}$ due insiemi, allora $A \cup B=\{2,4,6,3,1\}$ e $A \cap B=\{4,3\}$. Si noti che in generale l'unione è un insieme con un numero di elementi minore o al più uguale alla somma dei numeri degli elementi dei due

insiemi, mentre l'intersezione ha un numero di elementi minore o al più uguale al minimo tra i numeri di elementi dei due insiemi.

Un insieme può essere rappresentato mediante un array 1D, le cui componenti sono gli elementi dell'insieme.

ESEMPIO:

sviluppare una procedura per determinare l'unione di due insiemi.

Siano a e b i due array, rispettivamente di n_a e n_b elementi, che rappresentano i due insiemi *A* e *B*. a, b, n_a e n_b sono dati di input. L'array c, di size minore o uguale a n_a + n_b, è il dato di output, e rappresenta l'unione dei due insiemi. Il valore di c può essere costruito usando l'idea incrementale: si assegna a c uno dei due insiemi, per esempio *A*; ogni elemento dell'altro insieme *B*, che non sia anche elemento di *A*, viene inserito nell'array c. Quest'ultima azione è un problema di ricerca sequenziale nell'array a.

```
procedure unione(in:a,n_a,b,n_b;out:c,n_c): real
var n_a,n_b,n_c: integer
var a: array(1..n_a) of real
var b: array(1..n_b) of real
var c: array(1..n_c) of real
begin
    n_c := n_a
% inizializzazione di c
    for i=1,n_c do
         c(i) := a(i)
    endfor
    for i=1,n_b do
        if ~appartiene_array(b(i),a,n_a)
         then
          n_c := n_c+1
          c(n_c) := b(i)
        endif
    endfor
end
```

Si noti che la dichiarazione dell'array c è formalmente scorretta in molti linguaggi (n_c non è un parametro di ingresso); la tolleriamo nello pseudolinguaggio P-like, perchè consente una immediata comprensione dell'algoritmo; il simbolo ~ denota l'operazione logica di negazione in P-like.

Il numero complessivo di confronti è al più n_b*n_a.

Si noti l'uso dei *commenti* ossia le righe che in P-like iniziano con %, per facilitare la leggibilità del programma.

ESEMPIO:

sviluppare una procedura per determinare l'intersezione di due insiemi.

Siano a e b i due array, rispettivamente di n_a e n_b elementi, che rappresentano i due insiemi *A* e *B*. a, b, n_a e n_b sono dati di input. L'array c, di size minore o uguale a min(n_a,n_b), rappresenta il dato di output, cioè l'intersezione dei due insiemi. Il valore di c può essere costruito usando l'idea incrementale: ogni elemento di uno dei due insiemi, sia *B*, che è anche elemento dell'altro insieme, sia *A*, viene inserito nell'array c. Quest'ultima azione è un problema di ricerca sequenziale nell'array a.

```
procedure intersezione(in:a,n_a,b,n_b;out:c,n_c): real
var n_a,n_b,n_c: integer
var a: array(1..n_a) of real
var b: array(1..n_b) of real
var c: array(1..n_c) of real
begin
    n_c := 0
    for i=1,n_b do
        if appartiene_array(b(i),a,n_a)
         then
           n_c := n_c+1
           c(n_c) := b(i)
        endif
    endfor
end
```

Il numero complessivo di confronti è al più n_b*n_a.

ESEMPIO:

sviluppare una procedura per l'ordinamento di un array 1D.

Il problema dell'ordinamento di un array (*sort*) consiste nel modificare il valore di un array in modo tale che i valori degli elementi costituiscano una sequenza ordinata. Si noti che vi è un unico array che è sia dato di input sia dato di output.

L'ordinamento (che supporremo sempre in ordine crescente) può essere effettuato su array di tipo intero, reale e carattere; infatti per qualunque coppia di dati di questi tipi è possibile stabilire quale dei due dati precede l'altro.

Un semplice algoritmo di ordinamento è basato sull'applicazione dell'approccio incrementale. L'idea consiste nell'ordinare l'array procedendo per porzioni di size via via crescente. Si ordinano i primi due elementi (porzione 1..2), poi si ordina la porzione 1..3, quindi la 1..4 dell'array di

partenza e così via fino alla porzione `1..n`, che implica che l'intero array è stato ordinato. Si noti che, per esempio, per ordinare la porzione `1..3`, essendo già stata ordinata la porzione `1..2`, l'operazione da effettuare è quella di inserire il terzo elemento nella posizione corretta rispetto ai primi due, ottenendo che la porzione `1..3` sia ordinata.

F	D	B	A	M	C	L	N

array disordinato

D	F	B	A	M	C	L	N

ordinamento della porzione `1..2` dell'array

B	D	F	A	M	C	L	N

ordinamento della porzione `1..3` dell'array

In generale, al passo `i` si ha che la porzione `1..i-1` dell'array è ordinata e l'elemento `i`-simo deve essere inserito nella posizione corretta rispetto agli elementi di tale porzione, in modo da ottenere che la porzione `1..i` dell'array (iniziale) sia ordinata. Per questo motivo, l'algoritmo è detto *algoritmo di ordinamento per inserimento*.

dati di input: l'array `a`, (disordinato), il size `n`
dato di output: l'array `a` (ordinato)
costrutto ripetitivo: `for`
operazione ripetuta (al generico passo `i`): inserimento dell'`i`-simo elemento dell'array nella posizione corretta, rispetto ai primi `i-1` elementi, in modo da ottenere che la porzione `1..i` sia ordinata.

```
procedure ord_inserimento(in:n; inout:a)
var n_a,n_b,n_c: integer
var n: integer
var a: array(1..n) of character
var el_da_ins: character
begin
      for i=2,n do
         el_da_ins := a(i)
         j := i=1
         while j >= 1 and el_da_ins < a(j)
            a(j+1) := a(j)
```

```
            j := j-1
          endwhile
          a(j+1) := el_da_ins
        endfor
end
```

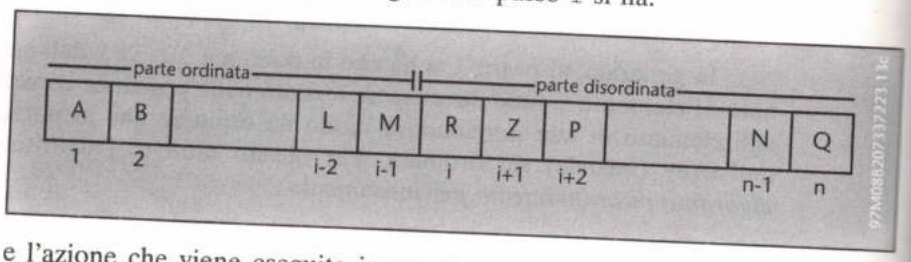
ESEMPIO:

sviluppare un'altra procedura per l'ordinamento di un array 1D.

Un altro semplice algoritmo di ordinamento può essere sviluppato basandosi sull'applicazione dell'idea incrementale alla costruzione dell'array ordinato, nel senso che l'array ordinato viene costruito per passi e all'`i`-simo passo la porzione `1..i` dell'array costituisce la porzione `1..i` dell'array finale ordinato.

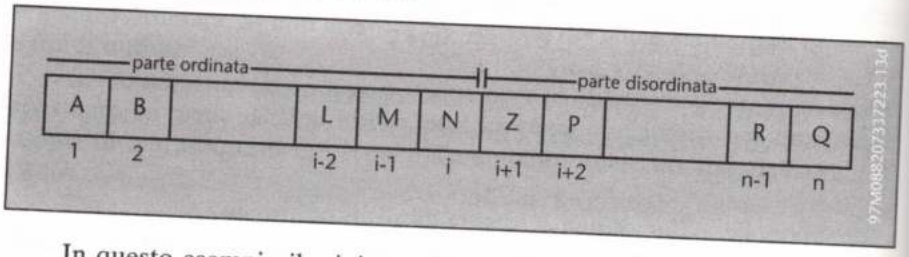
L'idea consiste nel vedere a ogni passo l'array come costituito da due parti: la parte già ordinata e la parte ancora disordinata. All'inizio del processo ripetitivo la parte già ordinata è vuota (e la parte disordinata è l'intero array), al termine la parte ordinata è l'intero array.

Per esempio, all'inizio del generico passo `i` si ha:



e l'azione che viene eseguita in un passo consiste nell'*ingrandire* la parte ordinata (da `1..i-1` a `1..i`) e nell'*accorciare* la parte disordinata (da `i..n` a `i+1..n`). Ciò si ottiene calcolando il minimo valore della parte disordinata e scambiando tale valore con quello dell'elemento di indice `i` dell'array; a questo punto la parte ordinata comprende anche l'elemento di indice `i`, cioè (`1..i`), mentre la parte disordinata è individuata dagli indici `i+1..n`.

Al termine del passo `i` si ha:



In questo esempio il minimo valore della parte disordinata (cioè della porzione `a(i..n)`) è **N** che è il valore dell'elemento di indice `n-1`. Si noti lo scambio tra i valori (**N**,**R**) degli elementi di indice `i` e `n-1`.

Il numero complessivo dei passi è `n-1`. Poichè l'azione che viene ripetuta a ogni passo è la ricerca del minimo (in una parte dell'array), l'algoritmo è detto *algoritmo di ordinamento per selezione di minimo*.

dati di input: l'array `a` (disordinato), il size `n`
dato di output: l'array `a` (ordinato)
costrutto ripetitivo: `for`
operazione ripetuta (al generico passo `i`): calcolo del minimo (e del suo indice) della parte disordinata (`i..n`), scambio del valore minimo con il valore dell'elemento al primo posto della parte disordinata.

```
procedure ord_selezione(in:n; inout:a)
var n, indice_min: integer
var a: array(1..n) of character
begin
      for i=1,n-1 do
         indice_min := i
% ricerca dell'indice dell'elemento minimo
% nella porzione (i+1)..n di a
         for k=i+1,n do
            if a(indice_min) > a(k)
               then
                  indice_min := k
            endif
         endfor
  % scambio dell'elemento minimo con il primo elemento
  % della porzione considerata di a
         scambiare_c(a(i),a(indice_min))
      endfor
 end
```

La `procedure ord_selezione` effettua più volte il calcolo dell'indice dell'elemento minimo di (una parte di) un array 1D; questo potrebbe essere evidenziato nel testo del programma attraverso una chiamata a una procedura. D'altra parte, nell'ottica dell'astrazione funzionale, l'idea dell'ordinamento mediante selezione è indipendente dal dettaglio di come viene calcolato il minimo.

```
procedure ord_selezione(in:n; inout:a)
var n, indice_min: integer
var min_array: character
var a: array(1..n) of character
begin
   for i=1,n-1 do
```

```
      min_val_ind(a(i..n),n-i+1,min_array,indice_min)
  % indice_min e' locale alla porzione di array;
  % spiazzamento di indice_min
        ind_min := indice_min+i-1
        scambiare_c(a(i),a(ind_min))
    endfor
end
```

dove `min_val_ind` è una procedura ottenuta a partire dalla procedura `max_val_ind`, con le semplici modifiche per l'individuazione del minimo invece del massimo.

Bisogna sottolineare che l'uso immediato della procedura `min_val_ind` nell'algoritmo di ordinamento discende dalla possibilità in P-like di denotare direttamente, tra i parametri attuali, una parte dell'array `a`, (quella individuata dal range `i..n`) mediante la notazione `a(i..n)`. Questa notazione è detta *porzione di array*. Si noti che il valore di `indice_min` in uscita da `min_val_ind` si riferisce all'indice locale alla porzione di array considerata e deve pertanto essere *spiazzato*, per essere utilizzato come un normale indice dell'array `a` (per esempio il valore `1` di `indice_min` fa riferimento al primo elemento della porzione `a(i..n)`).

ESEMPIO:

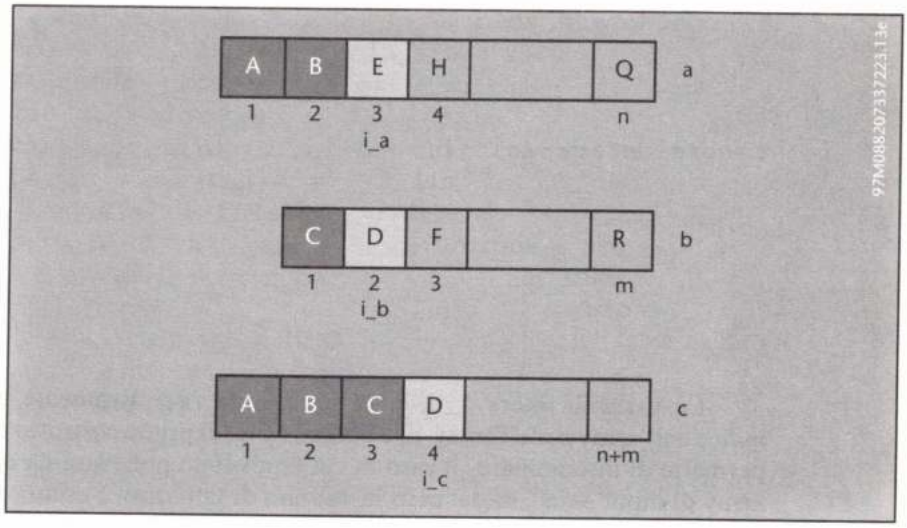
sviluppare una procedura per la fusione di due array 1D ordinati.

Il problema della fusione di array (*merge*) consiste nella generazione di un array ordinato di size `n+m` a partire da due array ordinati di size `n` e `m`. Si noti che si tratta di un particolare problema di determinazione dell'unione di due insiemi.

Per semplicità, si suppone che l'intersezione dei valori dei due array sia vuota.

dati di input: l'array `a` (ordinato), il size `n`, l'array `b` (ordinato), il size `m`
dato di output: l'array `c` (ordinato)
costrutto ripetitivo: `for`
operazione ripetuta (al generico passo `i`): calcolo dell'`i`-simo elemento di `c`.

Il progetto dell'algoritmo si basa sull'idea incrementale. La struttura di controllo è un ciclo `for` di `n+m` passi; l'`i`-simo elemento di `c` è il minimo tra il primo elemento di `a` e il primo elemento di `b` non ancora inseriti in `c`.




```
procedure fusione(in:a,n_a,b,n_b; out:c)
% a,b sono array ordinati, con intersezione nulla
var n_a,n_b,i_a,i_b: integer
var a: array(1..n_a) of character
var b: array(1..n_b) of character
var c: array(1..n_a+n_b) of character
begin
      i_a := 1
      i_b := 1
      for i_c=1,n_a+n_b do
         if i_a =< n_a and i_b =< n_b
% ci sono sia elementi di a sia di b da prendere in
% considerazione
           then
              if a(i_a) < b(i_b)
                 then
                    c(i_c) := a(i_a)
                    i_a := i_a + 1
                 else
                    c(i_c) := b(i_b)
                    i_b := i_b + 1
              endif
           else
% uno dei due array non deve essere considerato
              if i_b > n_b
                 then
```

```
% considerare solo a
                    c(i_c) := a(i_a)
                    i_a := i_a + 1
                else
% considerare solo b
                    c(i_c) := b(i_b)
                    i_b := i_b + 1
            endif
        endif
    endfor
end
```

Le variabili intere `i_a`, `i_b`, `i_c` sono rispettivamente usate come indice dell'array `a`, dell'array `b` e dell'array `c`. Il primo costrutto di selezione permette di discriminare il caso in cui è possibile prelevare da entrambi gli array di input `a` e `b`, dal caso in cui uno di tali array è *esaurito*.

ESEMPIO:

sviluppare una function che calcola il numero di volte in cui una parola compare in una riga di un testo.

Questo è un particolare problema di ricerca (*string matching*); la parola assegnata da ricercare, detta *parola chiave*, è un array 1D di size n di caratteri; il testo assegnato è un array 1D di size `m` di caratteri. L'algoritmo è basato sulla tecnica della ricerca sequenziale.

dati di input: l'array `chiave`, il size `n`, l'array `testo`, il size `m`
dato di output: `conta_chiave`
costrutto ripetitivo: `for`
operazione ripetuta (al generico passo `i`): confronto tra `chiave` e la porzione dell'array `testo` di indici `i..i+n-1`.

```
function  ricerca_chiave(chiave,n,testo,m): integer
var n, m: integer
var uguale: logical
var chiave: array(1..n) of character
var testo: array(1..m) of character
begin
   ricerca_chiave := 0
   for i=1,m-n+1 do
    if uguale(chiave,testo(i..i+n-1),n)
       then
          ricerca_chiave := ricerca_chiave + 1
    endif
```

```
   endfor
end
function  uguale(a,b,n): logical
var n,j: integer
var a,b: array(1..n) of character
begin
   uguale := false
   j := 0
   repeat
      j := j+1
   until a(j) ~= b(j) or j = n
   if a(j) = b(j) and j = n
      then
         uguale := true
   endif
end
```

Il ciclo `for` in `ricerca_chiave` considera esplicitamente ogni elemento dell'array `testo` come possibile inizio della stringa di caratteri da confrontare con `chiave`; la `function uguale` determina se si verifica tale evenienza.

Anche in questo algoritmo è stata utilizzata la convenzione di poter denotare come parametro attuale una parte di `n` elementi (di indici consecutivi) di un array (`testo`), mediante la notazione di porzione di array (`testo(i..i+n-1)`).

Si noti che gli array `chiave` e `testo(i..i+n-1)` hanno lo stesso size[8].

ESEMPIO:

sviluppare una procedura per il calcolo dei valori di una funzione assegnata su una griglia 1D equispaziata.

[8] Il problema della ricerca di una parola chiave può essere risolto da un algoritmo che effettua meno confronti di quelli richiesti dall'algoritmo implementato dalla function `ricerca_chiave`, cioè l'algoritmo Rabin - Karp (questo algoritmo non può essere qui discusso, si veda ad esempio il testo di Cormen et al. in bibliografia). L'idea su cui esso è basato è a prima vista sorprendente: una stringa di caratteri (sia la parola chiave sia il testo) è considerata un numero intero. Ogni carattere, infatti, può sempre essere visto come una cifra del sistema di numerazione che ha come base il numero complessivo dei caratteri dell'alfabeto. L'algoritmo utilizza risultati della teoria dei numeri. Questa idea è alla base anche di tutti gli algoritmi per la *crittografia*, cioè lo sviluppo di codici per codificare con sicurezza quasi assoluta le informazioni.

Infine, il problema dello string matching è di centrale importanza in biologia, per l'analisi delle sequenze di basi nel DNA e di amminoacidi nelle proteine; infatti il progetto "Genoma umano" ha dato impulso a un nuovo settore della biologia, detto *biologia computazionale* o *bioinformatica*.

Una griglia 1D relativa a un fissato intervallo [a,b] è un insieme di punti distinti $x_1 < x_2 < \cdots < x_n$ di tale intervallo. La quantità $h_i = x_{i+1} - x_i$ è l'ampiezza dell'i-simo intervallo di griglia ed è detta *passo* i-simo. L'insieme dei valori $f(x_i)$, $i = 1,\ldots,n$ è la *campionatura* (o *tabulazione*) della funzione f su tale griglia. Se il passo è costante, cioè $h_i = h_j$, $1 \le i, j \le n-1$, allora la griglia è *equispaziata* (o *uniforme*).

dati di input: la function che implementa la funzione f, gli estremi `a`,`b`, il numero `n` dei punti di griglia
dato di output: l'array 1D `f_campionata` che contiene la campionatura della funzione f
costrutto ripetitivo: `for`
operazione ripetuta (al generico passo `i`): generare l' `i`-simo punto della griglia e calcolare il valore della f in tale punto.

```
procedure calcolo_campionatura_1D(in: fun,a,b,n; ....
out: f_campionata)
var fun: real function
var a,b: real
var n: integer
var f_campionata: array(1..n) of real
var passo, punto_di_griglia: real
begin
   passo := abs(b-a)/float(n-1)
   punto_di_griglia := a
   for i=1,n do
       f_campionata(i) := fun(punto_di_griglia)
       punto_di_griglia := a+float(i)*passo
   endfor
end
```

Si noti che tra i parametri formali di input della procedura `calcolo_campionatura_1D` vi è una function. Il nome della function viene effettivamente precisato dal parametro attuale corrispondente, al momento della chiamata della procedura. Si tratta dell'usuale meccanismo di passaggio di parametri esteso al passaggio di una function.

In Fig. 3.14 sono riportati i grafici di tre diverse campionature della funzione $\sin(x)$ sull'intervallo $[0,2\pi]$, relative a griglie equispaziate di 4, 8, 50 punti.

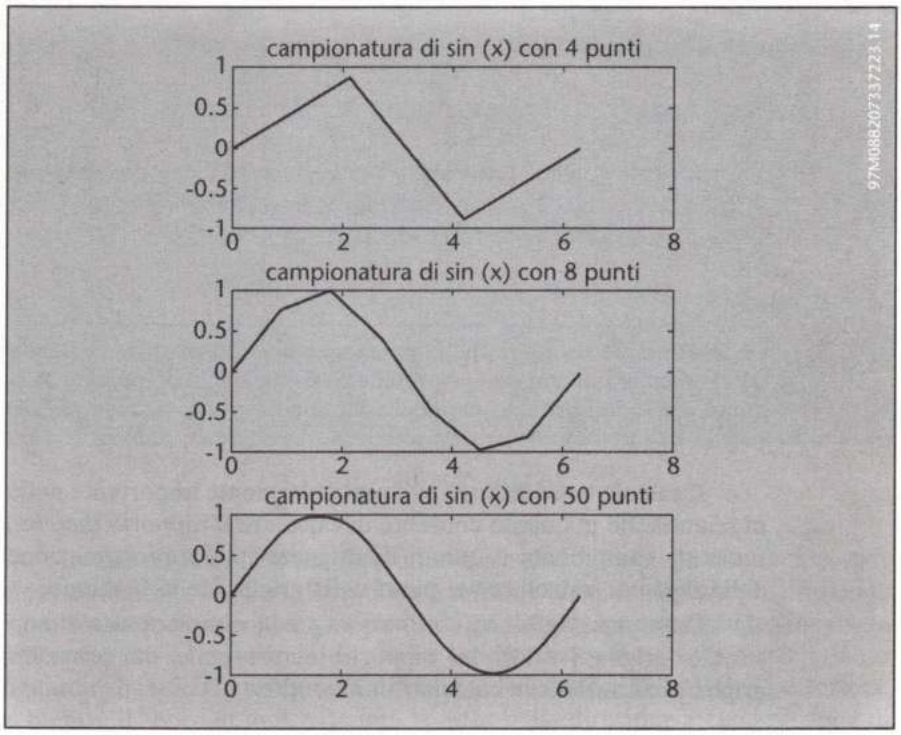


Fig. 3.14: campionatura di sin(x) su diverse griglie

ESEMPIO:

sviluppare una procedura per il calcolo della campionatura su un certo intervallo `[a,b]` di una funzione assegnata (griglia 1D equispaziata) e delle sue *differenze* (in avanti): la i-sima differenza è $f(x_{i+1}) - f(x_i)$. Calcolare inoltre le differenze *di ordine 2* della campionatura, cioè le differenze delle differenze.

dati di input: la function che implementa la funzione *f*, gli estremi `a,b`, il numero `n` dei punti di griglia
dato di output: l'array 1D `f_campionata` che contiene la campionatura della funzione *f*; l'array 1D `f_diff` delle differenze; l'array 1D `f_diff_2` delle differenze seconde

```
procedure difference_engine(in: fun,a,b,n; ....
out: f_campionata, f_diff, f_diff_2)
var fun: real function
var a,b: real
```

```
var n: integer
var f_campionata: array(1..n) of real
var f_diff: array(1..n-1) of real
var f_diff_2: array(1..n-2) of real
begin
   calcolo_campionatura_1D(fun,a,b,n,f_campionata)
   differenze(f_campionata,n,f_diff)
   differenze(f_diff,n-1,f_diff_2)
end
```

È interessante ricordare che la prima macchina calcolatrice di Babbage (cfr. par. 2.1) era in grado di risolvere il problema dell'esempio precedente; in particolare è noto che fu utilizzata nel caso della funzione $x^2 + x + 41$, sulla griglia 0,1,2,..,9.

Il calcolo delle differenze è particolarmente importante nelle applicazioni scientifiche in quanto consente di calcolare il *rapporto incrementale* di una funzione campionata e quindi di ottenere una approssimazione del valore della *derivata*, calcolata nei punti della griglia, della funzione.

Data una griglia $x_1 < x_2 < \cdots < x_n$ e la campionatura di una funzione su tale griglia, i valori del rapporto incrementale sui primi $n - 1$ punti della griglia possono essere calcolati utilizzando:

```
.....
differenze(f_campionata,n,f_diff)
differenze(griglia,n,passo)
for i=1,n-1 do
   rapporto_incrementale(i) := f_diff(i)/passo(i)
endfor
.....
```

Per l'approfondimento di questi aspetti si rimanda a un testo di Calcolo Numerico.

Una griglia 2D relativa a un fissato rettangolo [a,b] × [c,d] è un insieme di punti distinti di coordinate (x_i, y_j), con $x_1 < x_2 < \cdots < x_n$, $y_1 < y_2 < \cdots < y_m$ di tale rettangolo. La quantità $h_i = x_{i+1} - x_i$ è l'ampiezza dell'`i`-simo intervallo di griglia nella direzione x ed è detta *passo* x `i`-simo. La quantità $k_j = y_{j+1} - y_j$ è l'ampiezza del `j`-simo intervallo di griglia nella direzione y ed è detta *passo* y `j`-simo. L'insieme dei valori $f(x_i, y_j)$, $i = 1,\ldots,n$, $j = 1,,\ldots,m$ è la *campionatura* (o *tabulazione*) della funzione f su tale griglia.

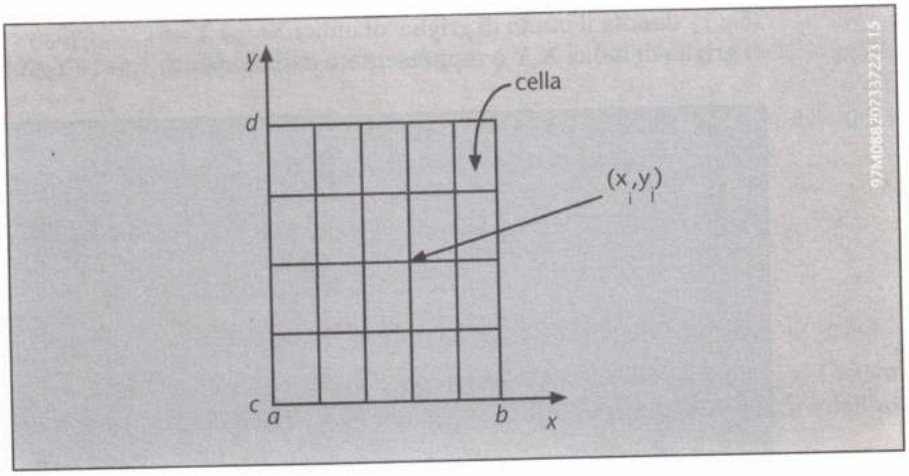


Fig. 3.15: esempio di griglia 2D

In una griglia 2D si usa il termine *linea di griglia* nella direzione x per indicare una riga di punti di griglia e il termine linea di griglia nella direzione y per indicare una colonna di punti di griglia. È opportuno sottolineare la differenza tra la notazione di griglia e la notazione matriciale per quanto riguarda gli indici: in una griglia il primo indice denota la linea nella direzione y, mentre il secondo indice denota la linea nella direzione x (la notazione di griglia rispecchia la notazione (x, y) delle coordinate di un punto del piano).

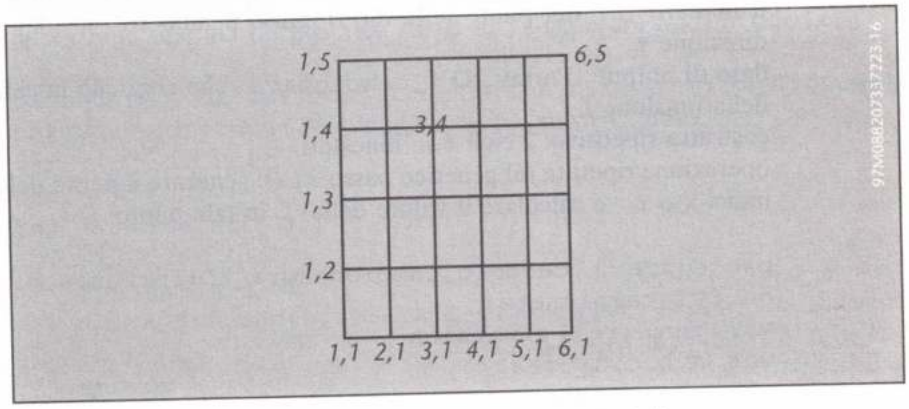


Fig. 3.16: indici nella notazione di griglia

Detti X e Y il primo e il secondo indice di un punto di griglia (nella notazione di griglia mostrata in Fig. 3.16), allora si ha che una griglia, di n punti nella direzione x e m punti nella direzione y, può essere rappresentata mediante un array 2D, di shape `(1..m,1..n)`, in cui l'elemento di indici

`(i,j)` denota il punto di griglia di indici X=`j` e Y=`m+1-i`; viceversa, il punto di griglia di indici X,Y è rappresentato dall'elemento `(m+1-Y,X)` dell'array.

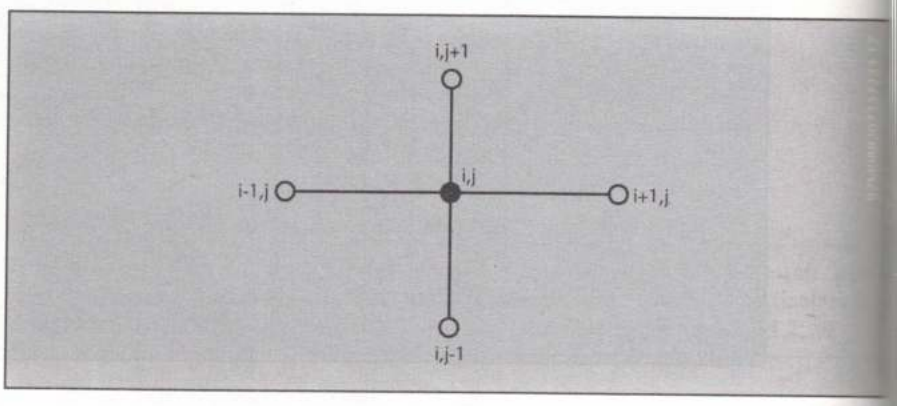


Fig. 3.17: i quattro vicini del punto di griglia i,j
(notazione di griglia)

ESEMPIO:

sviluppare una procedura per il calcolo dei valori di una funzione assegnata su una griglia 2D equispaziata nelle due direzioni x e y.

dati di input: la function che implementa la funzione *f*, gli estremi `a,b,c,d`, il numero `n` dei punti nella direzione x, il numero `m` dei punti nella direzione y
dato di output: l'array 2D `f_campionata` che contiene la campionatura della funzione *f*.
costrutto ripetitivo: 2 cicli `for` innestati
operazione ripetuta (al generico passo `X,Y`): generare il punto della griglia di indici `X` e `Y` e calcolare il valore della *f* in tale punto

```
procedure   calcolo_campionatura_2D(in:fun,a,b,n,c,d,m
out: f_campionata)
var fun: real function
var a,b,c,d: real
var n,m: integer
var f_campionata: array(1..m,1..n) of real
var passo_x,passo_y, ascissa,ordinata: real
begin
 passo_x := abs(b-a)/float(n-1)
 passo_y := abs(d-c)/float(m-1)
 ascissa := a
```

```
 ordinata := c
 for Y=1,m do
    for X=1,n do
        f_campionata(m+1-Y,X) := fun(ascissa, ordinata)
        ascissa := a+float(X)*passo_x
    endfor
    ascissa := a
    ordinata := c+float(Y)*passo_y
 endfor
end
```

La griglia viene costruita per successive linee nella direzione x. Ciò implica che l'array 2D della campionatura è costruito per righe a partire dall'ultima.

ESEMPIO:

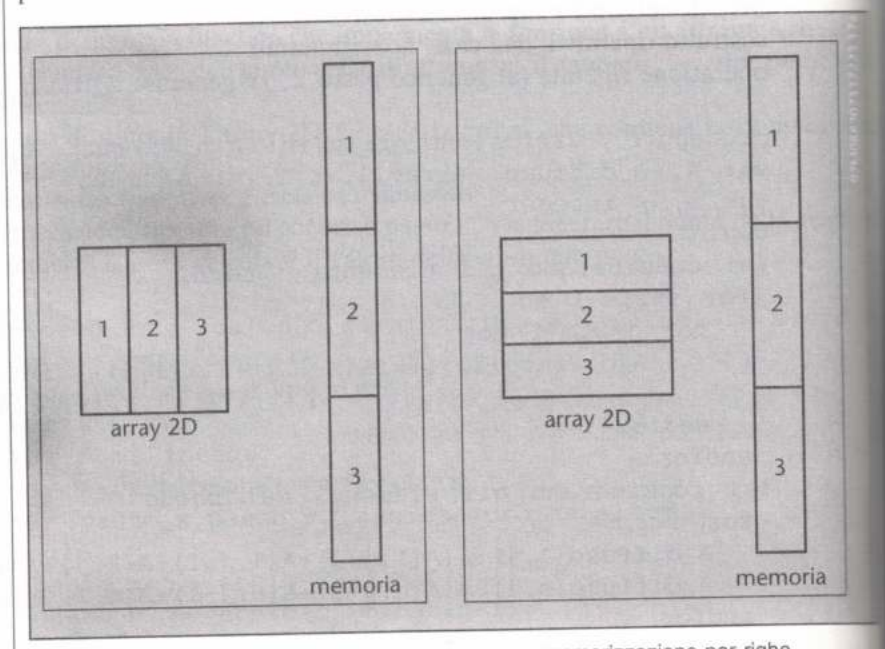
sviluppare una procedura che, a partire da un assegnato array 2D `A`, genera un nuovo array 2D `A_diffuso` tale che l'elemento `A_diffuso(i,j)` è la media dei valori degli elementi (al più quattro) di `A` che sono vicini all'elemento `(i,j)`.

dati di input: l'array `A`, il numero `n` delle righe, il numero `m` delle colonne
dato di output: l'array `A_diffuso`
costrutto ripetitivo: due cicli `for` innestati
operazione ripetuta (al generico passo `i,j`): generare `A_diffuso(i,j)`

```
procedure  diffusione_array(in: A,n,m; out: A_diffuso)
var A, A_diffuso: array(1..n,1..m) of real
var n,m: integer
begin
%si considerano gli elementi interni
 for i=2,n-1 do
    for j=2,m-1 do
        A_diffuso(i,j):=(A(i-1,j)+A(i+1,j)+...
                                A(i,j-1)+A(i,j+1))/4.0
    endfor
 endfor
%si considerano gli elementi sul bordo
 for j=2,m-1 do
    A_diffuso(1,j):=(A(1,j+1)+A(1,j-1)+A(2,j))/3.0
    A_diffuso(n,j):=(A(n,j+1)+A(n,j-1)+A(n-1,j))/3.0
 endfor
 for i=2,n-1 do
```

```
    A_diffuso(i,1):=(A(i-1,1)+A(i+1,1)+A(i,2))/3.0
    A_diffuso(i,m):=(A(i-1,m)+A(i+1,m)+A(i,m-1))/3.0
  endfor
  A_diffuso(1,1):=(A(1,2)+A(2,1))/2.0
  A_diffuso(1,m):=(A(1,m-1)+A(2,m))/2.0
  A_diffuso(n,1):=(A(n-1,1)+A(n,2))/2.0
  A_diffuso(n,m):=(A(n,m-1)+A(n-1,m))/2.0
end
```

La realizzazione della struttura array, cioè il modo in cui un oggetto di tale tipo strutturato è gestito a livello di linguaggio macchina, è molto semplice (si sta facendo riferimento al caso di allocazione statica di memoria), in quanto è una diretta astrazione della organizzazione della memoria di un calcolatore, che è una sequenza (cioè una tabella a una riga) di voci.
L'array 1D viene quindi realizzato mediante una sequenza di voci contigue della memoria. Il numero di voci di tale sequenza è uguale al size dell'array (nell'ipotesi che un elemento dell'array sia rappresentabile in una singola voce).
Anche l'array 2D di m righe e n colonne viene realizzato (nel caso di elementi scalari) mediante una sequenza di mxn voci contigue della memoria. In particolare se l'array 2D viene realizzato memorizzando nella sequenza di voci di memoria dapprima tutta la prima colonna, poi tutta la seconda colonna e così via, si parla di *memorizzazione per* colonne dell'array 2D; se invece la realizzazione prevede di memorizzare dapprima tutta la prima riga, poi tutta la seconda riga e così via, si parla di *memorizzazione per righe* dell'array 2D.



memorizzazione per colonne memorizzazione per righe

L'array in P-like è una struttura statica, nel senso che il numero di componenti deve essere precisato al momento della dichiarazione della variabile strutturata e non è alterabile dall'algoritmo. Come vedremo nel seguito, in Matlab è possibile allocare dinamicamente (cioè durante l'esecuzione del programma) spazio di memoria per un array.

ESERCIZI:

1)
Sia A un array 2D di 31×24 componenti; A(i,j) è la concentrazione di ozono rilevata all'ora j del giorno i. Utilizzando le function `media` e `varianza` determinare media e varianza giornaliera, media e varianza mensile, media e varianza mensile per ogni ora del giorno.

2)
Sviluppare una procedura per il calcolo delle differenze all'indietro della campionatura (1D) di una funzione. La i-sima differenza all'indietro è $f(x_i) - f(x_{i-1})$; la differenza all'indietro non è definita sul primo punto della griglia Per quali funzioni campionate le procedure differenze (in avanti) e differenze all'indietro generano un array a valore nullo?

3)
Determinare la campionatura su una griglia equispaziata di passo 1 della funzione $f(x) = 2^x$. Applicare la procedura differenze a tale campionatura e dare una giustificazione teorica del risultato ottenuto.

4)
Sviluppare un algoritmo per il calcolo delle *differenze centrali seconde* della campionatura (1D) di una funzione su una griglia equispaziata. La i-sima differenza centrale seconda è

$$f(x_{i-1}) - 2f(x_i) + f(x_{i-1}) \qquad , \; i = 2, \ldots, n-1$$

La differenza centrale seconda non è definita nel primo e nell'ultimo punto di griglia.

5)
Per quali array di input (ovvero per quali funzioni campionate) l'algoritmo per le differenze centrali seconde genera un array a valore nullo?

6)
Mostrare teoricamente e poi verificare attraverso l'esecuzione degli algoritmi che applicando a un array prima le differenze in avanti e poi quelle all'indietro, si ottiene il medesimo risultato dato dall'applicazione delle differenze centrali seconde.

7)
Sulla falsariga dell'esercizio precedente, che cosa succede se si effettuano prima le differenze all'indietro e poi le differenze in avanti?

8)
Applicare $n/2$ volte la procedura di differenze centrali seconde a un array il cui valore iniziale è 0,0,0,....0,1,0,.....0, dove l'unico elemento non nullo ha indice $n/2$, utilizzando ogni volta come valore di input il valore di output generato dalla chiamata precedente. Visualizzare i valori degli $n/2$ array così generati.

9)
Applicare la procedura diffusione_array all'array 2D il cui valore è la campionatura della funzione $f(x) = x \cdot e^{(-x^2-y^2)}$ sulla griglia 2D con passo 0.25 che discretizza il quadrato [–2,2] × [–2,2].

10)
Ripetere 10 volte l'applicazione della procedura diffusione_array alla campionatura dell'esercizio precedente, utilizzando ogni volta come valore di input il valore di output generato dalla chiamata precedente. Visualizzare i 10 valori dell'array così generati.

11)
Sviluppare una procedura per determinare se esiste un elemento di un array A di indice (i,j) tale che A(i,j) è il minimo valore nella riga i-sima e il massimo valore nella riga j-sima.

12)
Sviluppare una function che determina il numero di volte in cui le vocali compaiono in un testo.

13)
Modificare la procedura `ord_selezione` in modo tale da calcolare anche il numero complessivo di confronti effettuati per ordinare un array. Ottenere una tabella del tipo

N (size dell'array)	numero confronti
.....	.....
.....	.....

per i seguenti valori di N: 8,16,32,64,128.

14)
Modificare la procedura `fusione` in modo tale da calcolare anche il numero complessivo di confronti effettuati. Ottenere una tabella del tipo

N (size dei due array)	numero confronti
.....	.....
.....	.....

per i seguenti valori di N: 8,16,32,64,128.

15)
Sviluppare una function per calcolare la somma degli elementi di un array 2D che si trovano sul *bordo* dell'array.

Più veloce è il calcolatore, maggiore è l'importanza dell'efficienza dell'algoritmo.
Lloyd N. Trefethen, 1997

3.3 Efficienza di un algoritmo

L'esecuzione di un programma da parte di un calcolatore richiede una certa quantità di tempo (il *tempo di esecuzione*) e una certa quantità di memoria per la rappresentazione dei dati di input, output e locali[9].

Tempo e memoria di un calcolatore sono dette *risorse di calcolo* e la quantificazione dell'uso delle risorse di calcolo necessarie per l'esecuzione di un programma costituisce un importante elemento per la *valutazione delle prestazioni* di un programma.

L'uso delle risorse di calcolo fatto da un programma per risolvere uno specifico problema dipende da due fattori: la potenza del calcolatore e la struttura dell'algoritmo.

La potenza del calcolatore è qui da intendersi nell'accezione ampia di caratteristiche dell'ambiente hardware-software: tempo di clock, organizzazione della memoria e relativi tempi di accesso, tempi di accesso ai dispositivi di I/O, velocità di esecuzione delle istruzioni del linguaggio macchina, parallelismo interno,...., linguaggio di programmazione usato per lo sviluppo, caratteristiche del traduttore,.....

[9] Oltre alla quantità di memoria necessaria per memorizzare le istruzioni del programma.

La struttura dell'algoritmo determina sia l'insieme dei dati di input, output e locali, sia le sequenze computazionali, cioè le sequenze di tutte le istruzioni che devono essere eseguite per calcolare i dati di output.

Il numero totale dei dati e la lunghezza delle sequenze computazionali (il numero totale di istruzioni da eseguire) di un algoritmo sono quantità indipendenti dal calcolatore che esegue il programma; inoltre tali quantità sono proporzionali alla memoria e al tempo richiesti per l'esecuzione da parte di uno specifico calcolatore.

La determinazione di tali quantità costituisce l'analisi dell'*efficienza* di un algoritmo.

3.3.1 Complessità di tempo di un algoritmo

In generale, il numero complessivo delle istruzioni effettuate nel processo di esecuzione di un algoritmo dipende dal numero dei dati di input. Se si considera, per esempio, l'algoritmo per la somma di `n` numeri, si conclude immediatamente che il numero di somme da eseguire è proporzionale a `n` (infatti è `n-1`).

Un rapido esame di tutti gli algoritmi sviluppati nei paragrafi precedenti conferma che in ogni algoritmo la lunghezza della sequenza di operazioni dipende dal numero dei dati di input; chiameremo tale numero *dimensione computazionale del problema*.

Inoltre, un algoritmo è caratterizzato da alcune operazioni che risultano dominanti nella determinazione del tempo di esecuzione: la somma (tra due addendi) nel caso dell'algoritmo per la somma di `n` numeri, e in generale degli algoritmi per il calcolo di sommatorie; il prodotto nel caso dell'algoritmo per il fattoriale e la media geometrica; il confronto (tra due dati) nel caso dell'algoritmo per il calcolo del massimo, del minimo, per la ricerca; il con-

algoritmo	**operazione dominante**	**T(n)**
`somma`	somma	`n`
`somma_quadrati`	somma, prodotto	`n` (somme), `n` (prodotti)
`somma_1`	somma, prodotto	`2n` (somme), `3n`(prodotti)
`media_geom`	prodotto	`n+1`
`massimo`	confronto	`n-1`
`max_2D`	confronto	`n*m`
`fusione`	confronto	`n+m`
`unione`	confronto	`n*m`
`ord_selezione`	confronto, scambio	`n(n-1)/2` confronti, `n-1` scambi
`ricerca_seq`	confronto	`n`

Tab. 3.0: complessità di tempo di alcuni algoritmi di base

fronto e lo scambio di valore tra due elementi nel caso di ordinamento di un array 1D.

Da ciò discende che una indicazione ragionevolmente generale e affidabile del tempo di esecuzione è data dal numero complessivo di volte in cui viene eseguita la (o le) operazione dominante di un algoritmo per un fissato valore della dimensione del problema, o meglio da come tale numero varia al variare della dimensione del problema.

La *complessità di tempo* `T(n)` di un algoritmo è la funzione che esprime il numero di operazioni dominanti in dipendenza della dimensione computazionale `n` del problema.

Nel caso degli algoritmi `max_2D`, `fusione` e `unione` la dimensione del problema è descritta da due parametri `n, m` cioè il numero di righe e di colonne dell'array 2D di input o il size dei due array 1D; quindi la complessità di tempo è `T(n,m)`.

È semplice constatare che un algoritmo la cui struttura è

- ```
  for i=1,n do
     q operazioni dominanti
  endfor
  ```

  ha una complessità di tempo $T(n)=qn$, cioè proporzionale a n

- ```
  for i=1,n do
     q operazioni dominanti
  endfor
  for j=1,n do
     q operazioni dominanti
  endfor
  ```

  ha una complessità di tempo $T(n)=2qn$, cioè proporzionale a n

- ```
  for i=1,n do
    for j=1,n do
      q operazioni dominanti
    endfor
  endfor
  ```

  ha una complessità di tempo $T(n)=qn^2$, cioè proporzionale a n^2

- ```
  for i=1,n do
    for j=1,i do
      q operazioni dominanti
    endfor
  endfor
  ```

  ha una complessità di tempo $T(n)=q+2q+3q+\ldots\ldots+nq = qn(n+1)/2$, cioè proporzionale a n^2

La struttura dell'algoritmo determina sia l'insieme dei dati di input, output e locali, sia le sequenze computazionali, cioè le sequenze di tutte le istruzioni che devono essere eseguite per calcolare i dati di output.

Il numero totale dei dati e la lunghezza delle sequenze computazionali (il numero totale di istruzioni da eseguire) di un algoritmo sono quantità indipendenti dal calcolatore che esegue il programma; inoltre tali quantità sono proporzionali alla memoria e al tempo richiesti per l'esecuzione da parte di uno specifico calcolatore.

La determinazione di tali quantità costituisce l'analisi dell'*efficienza* di un algoritmo.

3.3.1 Complessità di tempo di un algoritmo

In generale, il numero complessivo delle istruzioni effettuate nel processo di esecuzione di un algoritmo dipende dal numero dei dati di input. Se si considera, per esempio, l'algoritmo per la somma di `n` numeri, si conclude immediatamente che il numero di somme da eseguire è proporzionale a `n` (infatti è `n-1`).

Un rapido esame di tutti gli algoritmi sviluppati nei paragrafi precedenti conferma che in ogni algoritmo la lunghezza della sequenza di operazioni dipende dal numero dei dati di input; chiameremo tale numero *dimensione computazionale del problema*.

Inoltre, un algoritmo è caratterizzato da alcune operazioni che risultano dominanti nella determinazione del tempo di esecuzione: la somma (tra due addendi) nel caso dell'algoritmo per la somma di `n` numeri, e in generale degli algoritmi per il calcolo di sommatorie; il prodotto nel caso dell'algoritmo per il fattoriale e la media geometrica; il confronto (tra due dati) nel caso dell'algoritmo per il calcolo del massimo, del minimo, per la ricerca; il con-

algoritmo	**operazione dominante**	**T(n)**
somma	somma	n
somma_quadrati	somma, prodotto	n (somme), n (prodotti)
somma_1	somma, prodotto	2n (somme), 3n(prodotti)
media_geom	prodotto	n+1
massimo	confronto	n-1
max_2D	confronto	n*m
fusione	confronto	n+m
unione	confronto	n*m
ord_selezione	confronto, scambio	n(n-1)/2 confronti, n-1 scambi
ricerca_seq	confronto	n

Tab. 3.0: complessità di tempo di alcuni algoritmi di base

fronto e lo scambio di valore tra due elementi nel caso di ordinamento di un array 1D.

Da ciò discende che una indicazione ragionevolmente generale e affidabile del tempo di esecuzione è data dal numero complessivo di volte in cui viene eseguita la (o le) operazione dominante di un algoritmo per un fissato valore della dimensione del problema, o meglio da come tale numero varia al variare della dimensione del problema.

La *complessità di tempo* `T(n)` di un algoritmo è la funzione che esprime il numero di operazioni dominanti in dipendenza della dimensione computazionale `n` del problema.

Nel caso degli algoritmi `max_2D`, `fusione` e `unione` la dimensione del problema è descritta da due parametri `n`, `m` cioè il numero di righe e di colonne dell'array 2D di input o il size dei due array 1D; quindi la complessità di tempo è `T(n,m)`.

È semplice constatare che un algoritmo la cui struttura è

- ```
  for i=1,n do
     q operazioni dominanti
  endfor
  ```
  ha una complessità di tempo $T(n)=qn$, cioè proporzionale a n

- ```
  for i=1,n do
     q operazioni dominanti
  endfor
  for j=1,n do
     q operazioni dominanti
  endfor
  ```
  ha una complessità di tempo $T(n)=2qn$, cioè proporzionale a n

- ```
  for i=1,n do
    for j=1,n do
      q operazioni dominanti
    endfor
  endfor
  ```
  ha una complessità di tempo $T(n)=qn^2$, cioè proporzionale a n^2

- ```
  for i=1,n do
    for j=1,i do
      q operazioni dominanti
    endfor
  endfor
  ```
  ha una complessità di tempo $T(n)=q+2q+3q+\ldots\ldots+nq =$
  $= qn(n+1)/2$, cioè proporzionale a n^2

```
• for i=1,n do
    for j=1,n do
      for k=1,n do
        q operazioni dominanti
      endfor
    endfor
  endfor
```

ha una complessità di tempo $T(n)=qn^3$, cioè proporzionale a n^3.

```
• i := 0
  repeat
      i := i+1
      q operazioni dominanti
  until 2^i >= n
```

ha una complessità di tempo $T(n)=q\log_2 n$, cioè proporzionale a $\log_2 n$.

Questi esempi mostrano che la complessità di tempo di un algoritmo viene determinata mediante una analisi teorica dell'algoritmo.

Informazioni sulla complessità possono anche essere ricavate sperimentalmente, considerando i tempi di esecuzione del programma su un certo calcolatore. La tecnica consiste nell'eseguire il programma prima per risolvere il problema di dimensione `n` e poi per `2n, 4n, 8n, 16n,` e nell'analizzare i tempi di esecuzione nei vari casi. Per esempio, l'esecuzione di un programma che implementa l'algoritmo `media_array` fornisce[10] i risultati in Tab. 3.1.

n	tempo di esecuzione (sec)	fattore di incremento dei tempi di esecuzione(attuale/precedente)
1000	0.0201	
2000	0.0403	2.004
4000	0.0793	1.967
8000	0.1611	2.031
16000	0.3182	1.975

Tab. 3.1: prestazioni di `media_array`

Si osservi che il fattore di incremento è all'incirca costante; il fatto che sia circa uguale a 2 indica che se si raddoppia la dimensione computazionale allora il tempo di esecuzione raddoppia. La conclusione è che la complessità dell'algoritmo è proporzionale a `n`, ovvero che la complessità è lineare.

In generale, questo tipo di esperimenti permette facilmente di caratterizzare complessità di tipo polinomiale in `n`, come sintetizzato nella Tab. 3.2.

[10] Il programma è stato scritto in Matlab ed eseguito su una ws IBM RS/6000.

fattore di incremento dei tempi di esecuzione, raddoppiando la dimensione computazionale	forma presumibile della complessità di tempo
2	proporzionale a n
4	proporzionale a n^2
8	proporzionale a n^3
16	proporzionale a n^4

Tab. 3.2: fattori di incremento e complessità corrispondente

Una importante osservazione è che la complessità di tempo oltre a dipendere dalla dimensione computazionale, può dipendere anche dai valori dei dati di input. Se si considera l'algoritmo di ricerca sequenziale, si osserva che il numero di confronti può anche essere minore di *n*. Infatti esistono, per esempio, particolari istanze del problema in cui la ricerca può dare esito positivo già al primo confronto.

Nel caso di algoritmi di questo tipo è opportuno determinare la complessità di tempo *del caso peggiore*; in questo modo si ottiene una maggiorazione del numero delle operazioni dominanti.

L'algoritmo di ricerca sequenziale ha una complessità di tempo *del caso peggiore* `T(n) = n`, che corrisponde alle istanze del problema in cui l'elemento che si ricerca non appartiene all'array.

Una analoga analisi può essere fatta per l'algoritmo di unione e di intersezione.

3.3.2 Complessità di spazio di un algoritmo

La determinazione dell'efficienza di un algoritmo rende necessario analizzare anche un altro aspetto della struttura dell'algoritmo.

Si consideri il seguente problema: data una sequenza di n numeri interi ordinata in senso crescente e memorizzata in un array 1D, costruire la sequenza ordinata in senso decrescente. La soluzione del problema è costituita dagli stessi elementi della sequenza di input, ma ordinati nel senso opposto.

Entrambi i seguenti algoritmi risolvono il problema:

```
procedure ordine_inverso_1(in:n;inout:a)
var n: integer
var a: array(1..n) of integer
begin
    for i=1,n/2 do
        scambiare_i(a(i),a(n+1-i))
    endfor
end
```

```
procedure ordine_inverso_2(in:a,n;out:b)
var n: integer
var a,b: array(1..n) of integer
begin
    for i=1,n do
        b(i) := a(n+1-i)
    endfor
end
```

In questi algoritmi l'operazione dominante è l'assegnazione; si ha

- `ordine_inverso_1` : `T(n)=3/2 n`
- `ordine_inverso_2` : `T(n)=n`

La procedura `ordine_inverso_1` ha una complessità di tempo maggiore, ma usa un solo array, quello dove è memorizzata la sequenza di input;

La procedura `ordine_inverso_2` ha una complessità di tempo minore, ma usa due array.

Il confronto dell'efficienza dei due algoritmi sarebbe incompleto se limitato alla sola complessità di tempo, senza tenere conto del numero complessivo dei dati di input, locali e di output.

La *complessità di spazio* `S(n)` di un algoritmo è la funzione che esprime il *size* totale delle strutture dati utilizzate per memorizzare dati di input, locali e di output, in dipendenza della dimensione computazionale `n` del problema.

La complessità di spazio è una indicazione ragionevolmente generale e affidabile dello spazio di memoria necessario per memorizzare i dati del programma che implementa l'algoritmo.

Si noti che la procedura `ordine_inverso_1` utilizza la stessa variabile strutturata di input per costruire e memorizzare il risultato, cioè l'algoritmo opera *in place*.

3.3.3 Complessità asintotica di un algoritmo

Le funzioni di complessità `T(n)` e `S(n)` sono funzioni non decrescenti della dimensione del problema, in quanto, come è intuitivamente evidente, istanze di dimensioni maggiori richiedono un costo computazionale maggiore (o, raramente, uguale) per la risoluzione.

In molti casi, l'analisi dell'efficienza di un algoritmo si ritiene conclusa quando si riesce a caratterizzare l'andamento asintotico di `T(n)` e `S(n)`, cioè il loro comportamento al crescere della dimensione computazionale `n`. Ciò comporta la determinazione solo dei termini dominanti per `n` che tende a infinito, piuttosto che la determinazione completa dell'espressione analitica di `T(n)` e `S(n)`.

Questa analisi della complessità è detta complessità asintotica.

La complessità asintotica viene espressa mediante la seguente notazione detta O grande:

siano $f(n)$ e $g(n)$ due funzioni; si dice che

$$f(n) = O(g(n))$$

se esistono due costanti positive c e N_0 tali che: $f(n) \leq cg(n), \quad n \geq N_0$.

Si noti che la funzione $g(n)$ è una limitazione superiore del comportamento asintotico di $f(n)$, in quanto $f(n) \to \infty$ come $g(n)$ o più lentamente di $g(n)$.

ESEMPIO:

sia

$$f(n) = \sum_{i=0}^{p} a_i n^i$$

cioè $f(n)$ sia un polinomio di grado p; allora si ha $f(n) = O(n^p)$.

ESEMPIO:

sia $f(n) = 2n^2 + 3n + 5$; allora si ha $f(n) = O(n^2)$.

È importante sottolineare che se $f(n) = O(g(n))$, si può concludere solo che $f(n)$ tende all'infinito *al più* come $g(n)$, cioè $f(n)$ può tendere all'infinito più lentamente di $g(n)$. Si ha, per esempio, che se $f(n) = O(n^p)$ allora certamente è anche $f(n) = O(n^{p+1})$, o ancora

$$f(n) = O(n^{p+k}) \text{ con } k \text{ positivo.}$$

Nel caso della complessità di tempo degli algoritmi, è frequente incontrare i seguenti andamenti asintotici:

$O(1)$	costante
$O(\log_2 n)$	logaritmico
$O(n)$	lineare
$O(n \log_2 n)$	lin-log
$O(n^2)$	quadratico
$O(n^k)$	polinomiale di grado k
$O(2^n)$	esponenziale
$O(n!)$	fattoriale

L'andamento asintotico delle funzioni esponenziale e fattoriale rende gli algoritmi che abbiano complessità di tempo $T(n)=O(2^n)$ oppure $T(n)=O(n!)$ praticamente del tutto inutilizzabili per la risoluzione di istanze dei problemi caratterizzate da una dimensione computazionale già dell'ordine di poche decine. La Tab. 3.3 mostra, a conferma di ciò, quali sono le dimensioni dei

problemi che possono essere risolti (rispettivamente in un secondo, un minuto e un'ora) usando algoritmi con alcune funzioni di complessità di tempo e supponendo che l'esecutore degli algoritmi esegua 100 milioni di operazioni (dominanti) in un secondo (100 Mops).

complessità	1 sec	1 minuto	1 ora
$O(\log_2 n)$	10^{16}	$3.6 \cdot 10^{19}$	$1.2 \cdot 10^{23}$
$O(n)$	10^8	$6 \cdot 10^9$	$3.6 \cdot 10^{11}$
$O(n \log_2 n)$	$\approx 4 \cdot 10^6$	$\approx 2 \cdot 10^8$	$\approx 1 \cdot 10^{10}$
$O(n^2)$	10^4	77459	$6 \cdot 10^5$
$O(2^n)$	26	32	38
$O(n!)$	11	12	14

Tab. 3.3: dimensioni dei problemi risolvibili, 100 Mops/sec

L'aumento della velocità di esecuzione dell'esecutore non comporta incrementi significativi, nel caso di complessità esponenziale e fattoriale, come mostrato nella Tab. 3.4, in cui il tempo di esecuzione di una operazione è di 10^{-12} sec.

complessità	1 sec	1 minuto	1 ora
$O(n)$	10^{12}	$6 \cdot 10^{13}$	$3.6 \cdot 10^{15}$
$O(n \log_2 n)$	$\approx 2.5 \cdot 10^{10}$	$\approx 1.4 \cdot 10^{12}$	$\approx 7.5 \cdot 10^{13}$
$O(n^2)$	10^6	$\approx 7.7 \cdot 10^6$	$6 \cdot 10^7$
$O(2^n)$	39	45	51
$O(n!)$	14	16	17

Tab. 3.4: dimensioni dei problemi risolvibili, supponendo una velocità di 1 milione di Mops/sec

Quindi, tranne che nel caso di problemi di dimensioni computazionali *piccole*, gli algoritmi utilizzabili sono quelli che hanno complessità di tempo polinomiale. Per esempio, l'esecuzione di un algoritmo con complessità di tempo fattoriale, da parte di un calcolatore che esegue una operazione in 10^{-12} sec, per risolvere un problema di dimensione computazionale 100, richiederebbe 10^{138} anni per essere conclusa!

Una conclusione fondamentale che è possibile trarre dalle considerazioni precedenti è che *un algoritmo è tecnologia*, nello stesso senso in cui lo è l'hardware del computer.

Infatti, si supponga che un problema di dimensione computazionale $n=10^7$ possa essere risolto sia da un algoritmo a complessità quadratica sia da uno a complessità lin-log; si supponga inoltre di avere a disposizione sia un calcolatore in grado di eseguire 100 milioni di operazioni al secondo (un PC), sia uno più potente, in grado di eseguire 10 miliardi di operazioni al secondo (una WS).

Allora, l'esecuzione dell'algoritmo a complessità lin-log sul PC richiede un tempo inferiore a quello necessario per eseguire l'algoritmo a complessità quadratica sulla WS.

Algoritmo a complessità quadratica (dimensione computazionale del problema $n=10^7$):

$$\frac{(10^7)^2 \text{ operazioni}}{10^8 \text{ operazioni/sec}} = 10^6 \text{ sec} \qquad \text{PC}$$

$$\frac{(10^7)^2 \text{ operazioni}}{10^{10} \text{ operazioni/sec}} = 10^4 \text{ sec} \qquad \text{WS}$$

Algoritmo a complessità lin-log:

$$\frac{10^7 \cdot \log_2 10^7 \text{ operazioni}}{10^8 \text{ operazioni/sec}} \approx 2 \text{ sec} \qquad \text{PC}$$

$$\frac{10^7 \cdot \log_2 10^7 \text{ operazioni}}{10^{10} \text{ operazioni/sec}} \approx 0.02 \text{ sec} \qquad \text{WS}$$

Un miglioramento delle prestazioni si ottiene sia ovviamente con un miglioramento tecnologico in senso stretto (da PC a WS) sia con un *miglioramento dell'algoritmo*.

Questa breve discussione sulla complessità computazionale mostra che un obiettivo fondamentale del progetto di un algoritmo per uno specifico problema deve essere quello di sviluppare un algoritmo efficiente, sia per quanto riguarda la complessità di tempo sia per la complessità di spazio.

Per molti problemi applicativi la scoperta di un nuovo algoritmo con complessità di tempo/spazio inferiore di un ordine di grandezza (per esempio da $O(n^2)$ a $O(n\log_2 n)$) rispetto al migliore algoritmo precedente ha rivoluzionato l'intero settore applicativo, in quanto ha consentito la risoluzione di istanze del problema fino a quel momento non risolvibili in tempi accettabili.

Successi eclatanti come la diminuzione di ordini di grandezza della complessità non sono frequenti; tuttavia anche lo sviluppo di un nuovo algoritmo

che abbia la stessa complessità asintotica di altri algoritmi utilizzati per la risoluzione di quel problema, ma che abbia una costante di proporzionalità minore (per esempio $T(n)=2n^2$ invece di $T(n)=4n^2$) costituisce un miglioramento tecnologico apprezzabile ($T(n)=2n^2$ invece di $T(n)=4n^2$ significa che il problema viene risolto in metà tempo con il nuovo algoritmo).

I seguenti esempi indicano che spesso alcune semplici considerazioni sulla natura specifica del problema da risolvere (dati di input, struttura e proprietà del problema) possono condurre allo sviluppo di un algoritmo efficiente.

ESEMPIO:

sviluppare un algoritmo per la valutazione del valore di un polinomio di grado n $p_n(x)$ in un punto fissato.

Il polinomio è rappresentato nella *forma standard* (o *forma delle potenze*), cioè:

$$p_n(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

Un polinomio di grado n in forma standard è univocamente determinato dagli $n+1$ coefficienti $a_0, a_1, \ldots, a_n$ e quindi rappresentabile computazionalmente mediante un array 1D di size $n+1$, il cui valore è la sequenza dei coefficienti del polinomio.

dati di input: l'array `coef` dei coefficienti, il grado `n`, il punto `c`
dato di output: `valore_pol`
costrutto ripetitivo: `for`
l'algoritmo può essere costruito generalizzando l'algoritmo di valutazione della sommatoria geometrica (`function somma_potenze`), sviluppato nel par 3.1.6

```
function valore_pol(coef,n,c): real
var coef: array(0..n) of real
var n: integer
var c,potenza: real
begin
    valore_pol := coef(0)
    potenza := 1.0
    for i=1,n do
        potenza := potenza*c
        valore_pol := valore_pol+coef(i)*potenza
    endfor
end
```

La complessità di tempo dell'algoritmo implementato nella `function valore_pol` è $T(n)=2n$, avendo individuato la moltiplicazione come operazione dominante.

Un algoritmo più efficiente può essere ottenuto osservando che:

$$p_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n =$$
$$= ((\cdots((a_n x + a_{n-1})x + a_{n-2})x + \cdots)x + a_1)x + a_0$$

L'espressione a secondo membro è detta a *parentesi innestate* ed è usualmente attribuita a Newton. Si parla quindi di polinomio in *forma innestata* o *forma di Newton*.

È immediato verificare che le due espressioni valutano la stessa quantità.

Tuttavia l'algoritmo basato sulla valutazione dell'espressione a parentesi innestate, attraverso la valutazione successiva di un livello di parentesi alla volta, a partire dal livello più interno, ha una complessità di tempo minore di $T(n)=2n$.

L'algoritmo è detto *algoritmo di Horner*, anche se è stato usato per la prima volta da Newton e pare fosse presente nella tradizione matematica cinese già attorno all'anno mille.

```
function valore_pol_horner(coef,n,c): real
var coef: array(0..n) of real
var n: integer
var c: real
begin
    valore_pol_horner := coef(n)
    for i=n-1,0,step=-1 do
        valore_pol_horner := valore_pol_horner*c+coef(i)
    endfor
end
```

Ogni passo del ciclo `for` valuta un livello di parentesi e richiede solo una moltiplicazione (e una addizione). Pertanto, l'algoritmo di Horner ha complessità di tempo $T(n)=n$ (operazione dominante: moltiplicazione); inoltre il numero di addizioni è n, cioè uguale a quello di `valore_pol`.

ESEMPIO:

sviluppare un algoritmo per il calcolo del massimo e del minimo (e dei loro indici) dei valori di un array 1D di tipo reale.

dati di input: l'array a, il size n
dato di output: max_array, l'indice i_max, min_array, l'indice i_min

L'algoritmo può essere costruito richiamando in sequenza una procedura per il calcolo del massimo (e del suo indice) dei valori di un array 1D e una procedura per il calcolo del minimo (e del suo indice). Una procedura per il calcolo del massimo è stata sviluppata nel par 3.2.3 (procedura max_val_ind); la procedura per il calcolo del minimo (diciamola procedure min_val_ind) è immediatamente ottenibile, con semplici modifiche della prima.

```
procedure max_min_1(in:a,n;out:max_array,i_max, ....
                                        min_array,i_min)
var a: array(1..n) of real
var n,i_max,i_min: integer
var max_array,min_array: real
begin
      max_val_ind(a,n,max_array,i_max)
      min_val_ind(a,n,min_array,i_min)
end
```

La complessità di tempo è $T(n)=2n-2$, avendo considerato il confronto come operazione dominante, in quanto sia l'algoritmo per il calcolo del massimo sia quello per il minimo hanno complessità $T(n)=n-1$.

Il seguente algoritmo è basato sull'applicazione dell'approccio incrementale e calcola simultaneamente il valore massimo e minimo su porzioni di size crescente dell'array:

```
procedure
max_min_2(in:a,n;out:max_array,i_max,min_array,i_min)
var a: array(1..n) of real
var n,i_max,i_min: integer
var max_array,min_array: real
begin
      max_array := a(1)
      min_array := a(1)
      i_max := 1
      i_min := 1
      for i=2,n do
          if a(i) > max_array
            then
              max_array := a(i)
              i_max := i
            else
```

```
                if a(i) < min_array
                  then
                      min_array := a(i)
                      i_min := i
                endif
            endif
        endfor
end
```

La complessità di tempo, nel caso peggiore, è ancora $T(n)=2n-2$, anche se esistono valori dei dati di input che danno luogo solo a $n-1$ confronti.

È possibile sviluppare un algoritmo più efficiente (nel caso peggiore), osservando che in `max_min_2` , nel caso peggiore, ogni elemento dell'array viene confrontato due volte. Se invece si confrontano tra loro coppie di elementi successivi dell'array e quindi il massimo della coppia viene confrontato con `max_array` e il minimo con `min_array` si effettuano solo 3 confronti per ogni coppia di elementi dell'array, per un totale di $3n/2$ confronti (per semplicità, assumiamo n pari).

```
procedure max_min(in:a,n;out:max_array,i_max,
                                    min_array,i_min)
var a: array(1..n) of real
var n,i_max,i_min: integer
var max_array, min_array: real
begin
      max_array := a(1)
      min_array := a(1)
      i_max : =1
      i_min := 1
      for i=2,n-1,step=2 do
          if a(i) > a(i+1)
            then
                if a(i) > max_array
                  then
                      max_array := a(i)
                      i_max := i
                  endif
                  if a(i+1) < min_array
                    then
                      min_array := a(i+1)
                      i_min := i+1
                  endif
            else
                  if a(i) < min_array
```

```
                    then
                      min_array := a(i)
                      i_min := i
                endif
                if a(i+1) > max_array
                  then
                      max_array := a(i+1)
                      i_max := i+1
                endif
            endif
        endfor
    end
```

La complessità di tempo è $T(n)= 3n/2$.

3.3.4 Complessità intrinseca di un problema

Un aspetto interessante dello studio della complessità computazionale consiste nel generalizzare l'analisi in modo tale da determinare, oltre alla complessità di un certo algoritmo, anche limitazioni inferiori di complessità per classi di algoritmi (o per tutti gli algoritmi) per la risoluzione di uno specifico problema. Questo approccio, che è proprio della *teoria della complessità*, consente di concludere che la complessità di un algoritmo per un dato problema deve avere almeno un certo andamento asintotico e rende dunque possibile stabilire l'*ottimalità* di certi algoritmi. Un tale risultato non è facile da ottenere e le limitazioni inferiori di complessità e l'esistenza di algoritmi ottimali[11] sono state stabilite solo per alcune classi di problemi.

Nel caso in cui sia stata determinata l'ottimalità di un algoritmo, si può allora parlare di *complessità computazionale intrinseca* del problema, assumendo cioè che la complessità del problema sia la complessità dell'algoritmo risolutore ottimale.

È possibile per esempio dimostrare che l'algoritmo di Horner è ottimale. Allora si può concludere che la complessità computazionale intrinseca del problema della valutazione di un polinomio di grado n in un punto (polinomio rappresentato nella forma standard) è $T(n)=n$ (prodotti e somme) con $S(n)=n+1$.

È naturale classificare i problemi in base alla loro complessità intrinseca. Un problema la cui complessità intrinseca è $O(n^k)$, con k costante positiva, è

[11] Si ricorda che la *qualità complessiva* di un algoritmo, soprattutto nel caso del calcolo scientifico, è determinata non solo dall'*efficienza*, ma anche dalla *stabilità* dell'algoritmo.

detto *problema risolvibile in un tempo polinomiale*. La maggior parte dei problemi considerati nei paragrafi precedenti sono di questo tipo. È evidente che un problema caratterizzato da un elevato valore di k (per esempio $k > 10$) richiede un tempo di esecuzione per la sua risoluzione che può risultare inaccettabile in pratica già per piccoli valori della dimensione n (questo può accadere anche nel caso di complessità lineare del tipo $O(1000^k n)$. Tuttavia, se paragonato a un problema a complessità intrinseca esponenziale ($O(a^n)$, con $a > 1$), un qualunque problema a complessità intrinseca polinomiale risulta più semplice. Infatti, è facile convincersi che esiste certamente un valore della dimensione computazionale a partire dal quale la risoluzione del problema a complessità intrinseca esponenziale richiede un tempo di esecuzione maggiore di quello necessario per risolvere il problema a complessità intrinseca polinomiale.

I problemi a complessità intrinseca polinomiale costituiscono l'insieme dei problemi trattabili.

Al contrario, *i problemi a complessità intrinseca esponenziale costituiscono l'insieme dei problemi intrattabili.*

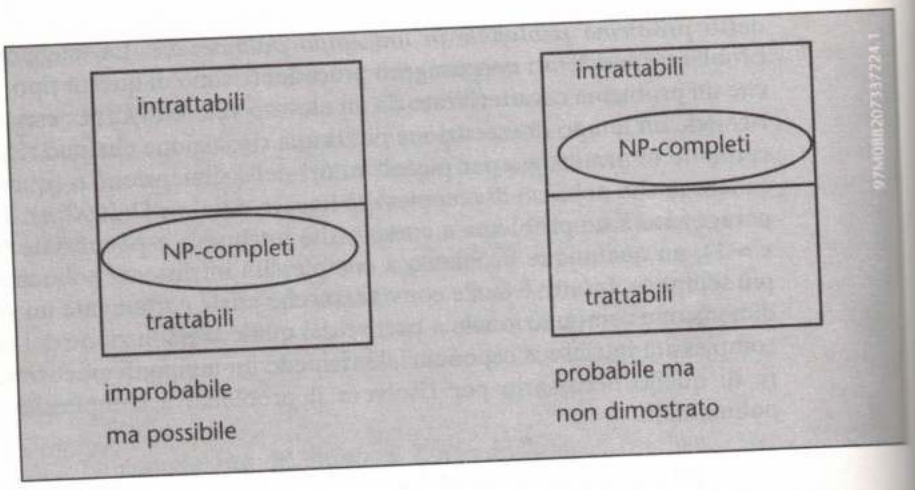
Si noti che complessità del tipo $O(n!)$, $O(a^{n^n})$ indicano crescite più rapide rispetto alla complessità esponenziale. Una complessità almeno esponenziale è anche detta *complessità non polinomiale.*

Un problema a complessità intrinseca non polinomiale è intrattabile nel senso che può essere risolto in pratica solo per piccoli valori della dimensione computazionale.

Un esempio di problema di tale natura è quello della generazione di tutte le permutazioni di un insieme di n oggetti. Poichè il numero di possibili permutazioni è $n!$, nell'ipotesi che la generazione di una permutazione richieda un numero costante di scambi, si può immediatamente concludere che il problema è a complessità intrinseca fattoriale (operazione dominante: scambio).

I problemi che sono necessariamente riconducibili al problema della generazione di *tutte* le permutazioni dell'insieme dei dati di input sono intrattabili.

C'è un consistente insieme di problemi che hanno la seguente proprietà: si conoscono algoritmi a complessità non polinomiale per la loro risoluzione ma per nessuno di tali problemi è stato possibile dimostrare che l'algoritmo ottimale è non polinomiale! Ciò significa che non è stato dimostrato che la complessità intrinseca di questi problemi è non polinomiale. L'insieme di questi problemi è detto insieme dei *problemi NP-completi.*



Le implicazioni teoriche legate ai problemi NP-completi non vengono qui trattate; si rimanda a testi di Teoria della complessità per un loro approfondimento[12].

Per esempio, sono NP-completi i problemi

- dato un insieme finito S di numeri naturali e un fissato numero naturale c, trovare un sottoinsieme di S tale che la somma dei suoi elementi è uguale a c;
- dato un insieme di città collegate da una rete stradale e una fissata lunghezza c, trovare un percorso di lunghezza uguale a c;
- dato un insieme di città collegate da una rete stradale, trovare un percorso di lunghezza minima che parta da una città, tocchi tutte le città una sola volta e termini nella città di partenza.

Infine, è importante sottolineare che i problemi intrattabili sono in realtà asintoticamente intrattabili, ma una particolare istanza di qualunque problema a complessità intrinseca non polinomiale può essere risolta se la sua dimensione computazionale è *piccola*. Nelle applicazioni concrete il grado di trattabilità dipende dall'algoritmo risolutore (cioè dalla sua effettiva complessità non polinomiale) e dalla velocità del calcolatore che lo esegue.

ESERCIZI

1)
Riempire la seguente tabella, effettuando esperimenti sui tempi di esecuzione dell'algoritmo di ordinamento per selezione (par. 3.2.3). L'esercizio richiede

[12] Vedere per esempio il testo di Cormen, Leiserson e Rivest per ulteriori riferimenti bibliografici.

che sia noto il tempo di esecuzione dell'operazione significativa sul calcolatore che viene utilizzato (è ricavabile sperimentalmente, come?).

n	tempo di esecuzione (sec)	tempo stimato mediante T(n)
1000		
2000		
4000		
8000		
16000		

2)
Sviluppare un algoritmo per la campionature di tutte le funzioni introdotte nel par. 3.3.3, che descrivono le complessità asintotiche di tempo più comuni. Utilizzare la griglia *logaritmicamente uniforme* $10, 10^2, 10^3, 10^4, 10^5, 10^6, 10^7, 10^8$.

3)
Determinare la complessità di spazio e di tempo degli algoritmi di ordinamento, ricerca e fusione.

3.4 Algoritmi per problemi elementari con matrici

L'uso delle matrici pervade tutta la disciplina del calcolo e lo sviluppo di algoritmi per la risoluzione di problemi coinvolgenti matrici è un aspetto fondamentale dell'Informatica applicata e del Calcolo Scientifico.

Nel seguito vengono discussi gli aspetti computazionali di base legati alla rappresentazione delle matrici mediante strutture dati opportune e allo sviluppo degli algoritmi per alcune operazioni fondamentali sulle matrici.

3.4.1 Definizioni generali e partizionamenti

Una *matrice A* può essere definita[13] come una collezione di $m \times n$ numeri organizzata secondo lo schema a tabella:

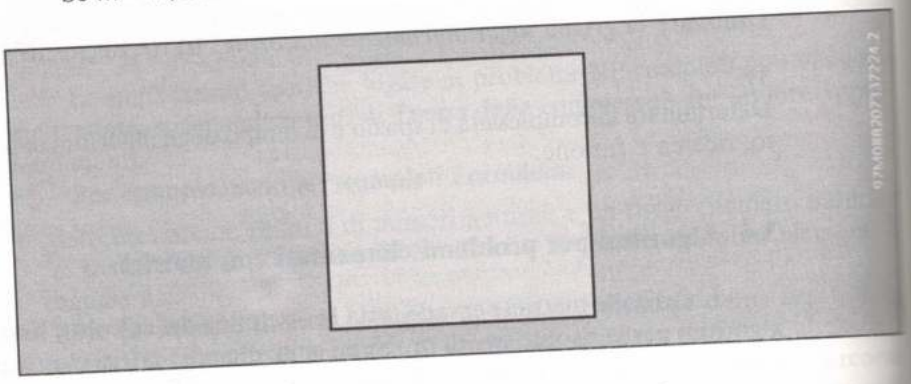
$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \tag{3.4.1}$$

[13] Questa definizione è qui usata a fini pratici; un'altra definizione potrebbe essere: sia I_{mn} l'insieme delle coppie ordinate *(i,j)* di numeri interi , con $1 \le i \le m$, $1 \le j \le n$. Siano inoltre I un insieme, *a* una funzione definita in I_{mn} a valori in I. Indichiamo con a_{ij} l'immagine in *a* di *(i,j)*. L'insieme dei valori di *a* disposti come in (3.5.1) è una matrice di tipo $m \times n$ su I.

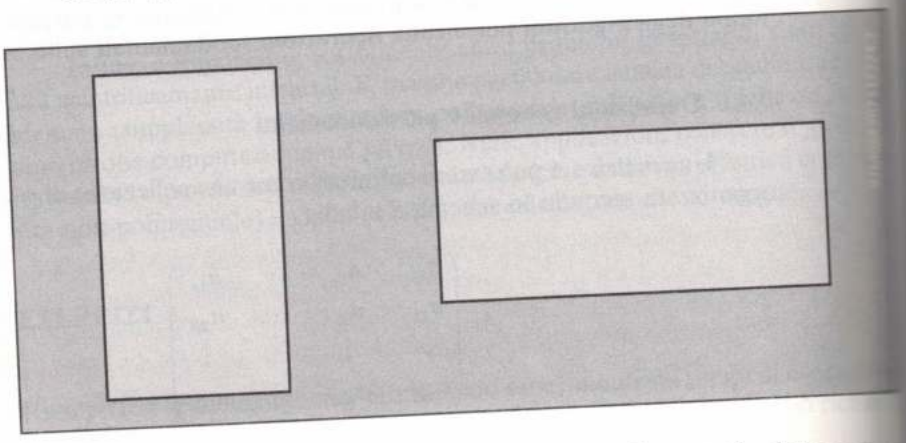
La matrice A ha m *righe*, n *colonne*. Il numero a_{ij} è detto *elemento di* A di *posto* i,j. Di seguito sono evidenziate, come esempio, la seconda riga e la n-sima colonna della matrice A:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{1n} \\ a_{21} & a_{22} & a_{2n} \\ \\ a_{m1} & a_{m2} & a_{mn} \end{pmatrix}$$

Se $m = n$, la matrice è detta *quadrata* (di ordine n):



Se $m \neq n$, la matrice è detta *rettangolare* $(m \times n)$



Particolare rilevanza hanno la matrice rettangolare $m \times 1$, detta *vettore colonna* di m elementi:

$$c = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{pmatrix}$$

e la matrice rettangolare $1 \times n$, detta *vettore riga* di n elementi:

$$r^T = (r_1, r_2, \ldots, r_n).$$

La notazione evidenzia che, quando non altrimenti specificato, un vettore è inteso come vettore colonna; infatti un vettore riga è denotato con l'apice T (si legge: trasposto), intendendo che il vettore r (senza apice T) è un vettore colonna. L'apice T indica l'operazione di *trasposizione*.

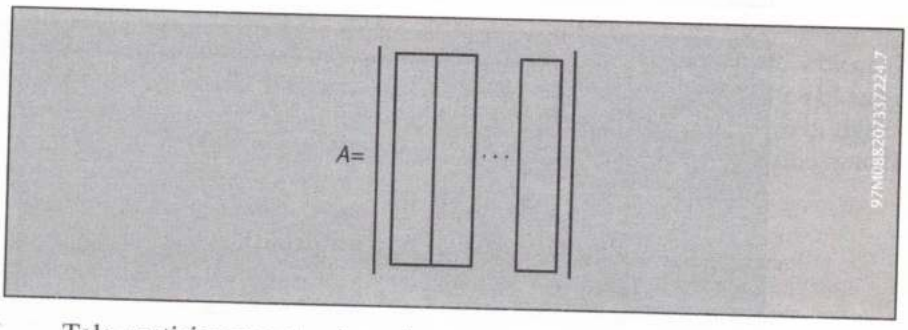
Due matrici sono *uguali* se hanno lo stesso numero di righe e di colonne e inoltre hanno gli elementi uguali per ogni posto.

È convenzione indicare sinteticamente particolari insiemi di posti (di elementi) di una matrice: la Fig. 3.22 a pagina seguente mostra la *diagonale principale*, il *triangolo (trapezio) strettamente superiore* e il *triangolo (trapezio) strettamente inferiore*, nei vari casi. L'unione della diagonale principale e del triangolo strettamente superiore prende il nome di *triangolo superiore*, mentre l'unione della diagonale principale e del triangolo strettamente inferiore è detto *triangolo inferiore*.

È spesso conveniente vedere una matrice come costituita da sottoinsiemi di righe e di colonne, cioè secondo un *partizionamento a blocchi*.

I partizionamenti più semplici sono quello per colonne e quello per righe.

Nel partizionamento *per colonne* una matrice è vista come formata da n vettori colonna:



9788820733722 4 7

Tale partizionamento viene denotato con

$$A = (a_{*1}, a_{*2}, \ldots, a_{*n})$$

dove a_{*i} indica l'i-sima (vettore) colonna di A.

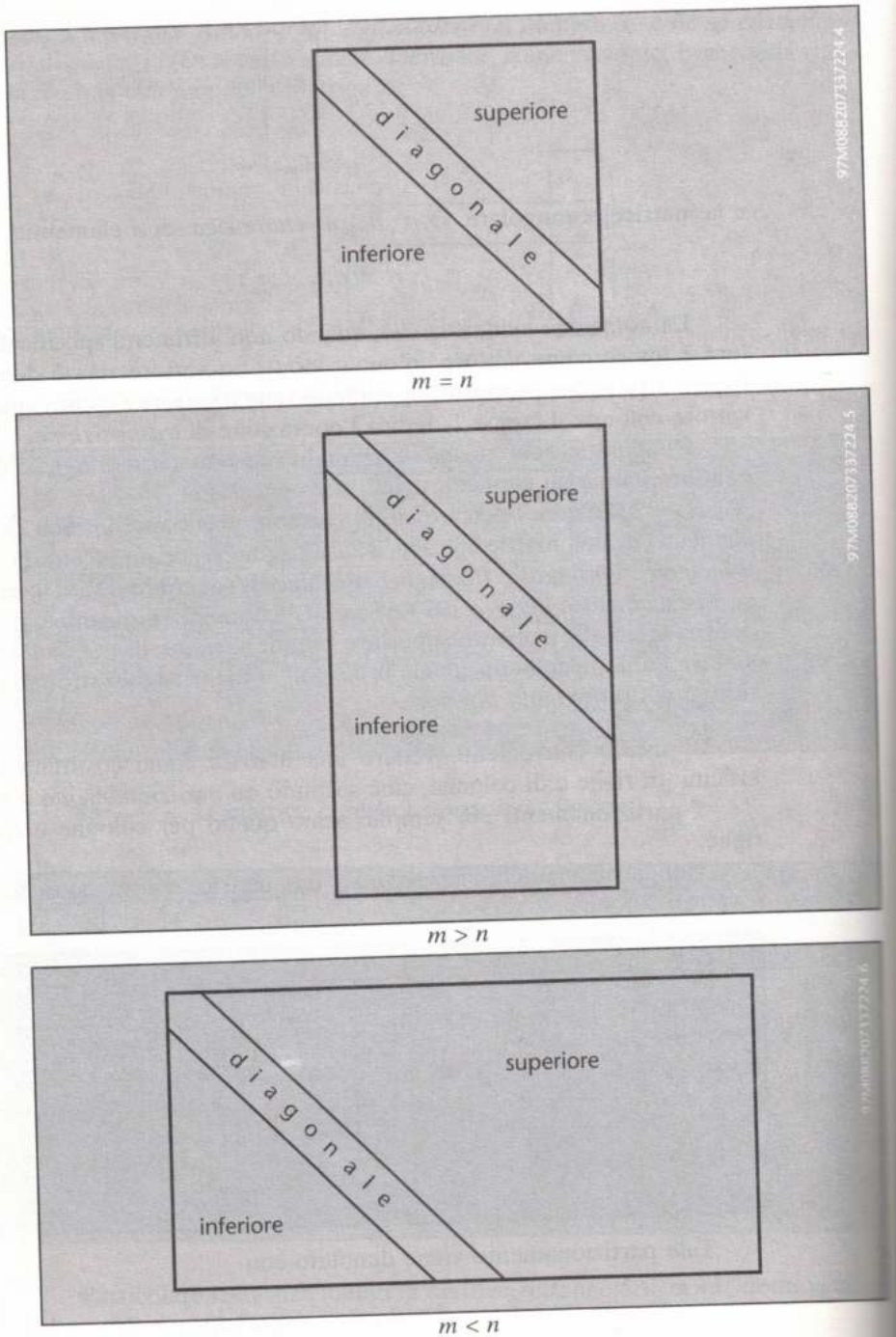
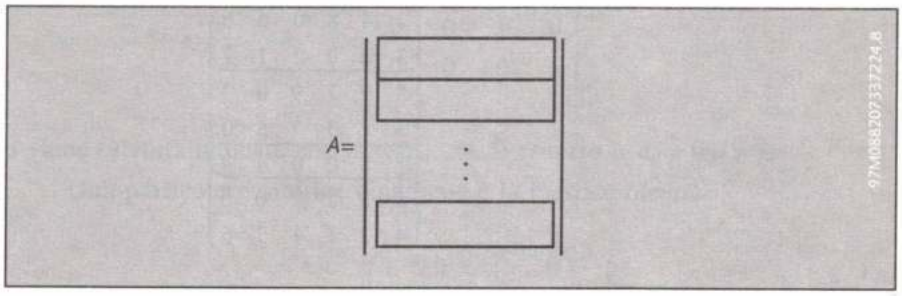


Fig. 3.22: diagonale principale, triangoli superiore e inferiore di matrici

Analogamente, nel partizionamento *per righe* una matrice è vista come formata da m vettori riga:



Tale partizionamento viene denotato con

$$A = \begin{pmatrix} a_{1\bullet}^T \\ a_{2\bullet}^T \\ \vdots \\ a_{m\bullet}^T \end{pmatrix}$$

dove $a_{i\bullet}^T$ indica l'i-sima (vettore) riga di A.

Una generalizzazione dei partizionamenti precedenti è il *partizionamento a blocchi*, in cui una matrice è vista come formata da un certo numero di matrici (con un numero inferiore di righe e di colonne) dette *blocchi*: Per esempio, la seguente matrice A è una matrice 3×3 a blocchi:

$$A = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix}$$

dove A_{ij} indica il blocco di posto *ij di A*. Si noti che, per esempio, i blocchi $A_{11}\, A_{12}\, A_{13}$ formano la prima *riga di blocchi* e che necessariamente i blocchi che compongono una riga di blocchi devono avere lo stesso numero di righe. Analoga considerazione vale per una *colonna di blocchi*. Due partizionamenti a 3×4 blocchi della stessa matrice sono esemplificati di seguito:

$$\left(\begin{array}{cc|c|cc|c} 2 & 4 & 8 & 0 & 6 & 5 \\ 3 & 0 & 0 & 2 & 1 & 7 \\ \hline 5 & 9 & 7 & 9 & 0 & 3 \\ 3 & 2 & 3 & 7 & 5 & 0 \\ 7 & 8 & 0 & 0 & 4 & 4 \\ \hline 5 & 8 & 2 & 2 & 0 & 5 \\ 4 & 3 & 1 & 1 & 7 & 6 \end{array}\right), \quad \left(\begin{array}{c|cc|c|cc} 2 & 4 & 8 & 0 & 6 & 5 \\ \hline 3 & 0 & 0 & 2 & 1 & 7 \\ 5 & 9 & 7 & 9 & 0 & 3 \\ 3 & 2 & 3 & 7 & 5 & 0 \\ 7 & 8 & 0 & 0 & 4 & 4 \\ \hline 5 & 8 & 2 & 2 & 0 & 5 \\ 4 & 3 & 1 & 1 & 7 & 6 \end{array}\right),$$

Un partizionamento in 4×1 blocchi della stessa matrice è

$$\left(\begin{array}{cccccc} 2 & 4 & 8 & 0 & 6 & 5 \\ 3 & 0 & 0 & 2 & 1 & 7 \\ \hline 5 & 9 & 7 & 9 & 0 & 3 \\ 3 & 2 & 3 & 7 & 5 & 0 \\ \hline 7 & 8 & 0 & 0 & 4 & 4 \\ \hline 5 & 8 & 2 & 2 & 0 & 5 \\ 4 & 3 & 1 & 1 & 7 & 6 \end{array}\right)$$

3.4.2 Caratteristiche strutturali

Nell'approccio computazionale è fondamentale classificare le matrici in base ad alcune caratteristiche che sono individuabili immediatamente esaminando il numero degli elementi nulli e degli elementi non nulli e i posti da essi occupati nella matrice. Tali proprietà sono dette *caratteristiche strutturali* di una matrice.

La prima distinzione è tra matrici *piene* e matrici *sparse*. Una matrice è piena se la maggior parte dei suoi elementi è non nulla. Una definizione più precisa è la seguente: sia A una matrice $m \times n$; A è piena se il numero degli elementi non nulli è proporzionale a $m \cdot n$.

Una matrice è sparsa se la maggior parte dei suoi elementi è nulla.

Più precisamente: sia A una matrice $m \times n$, ed nnz il numero di elementi non nulli; si definisce grado di sparsità (o coefficiente di sparsità) della matrice, la quantità:

$$sp = \frac{m \times n - nnz}{m \times n}$$

La matrice ha un elevato grado di sparsità se sp è molto vicino a 1. In particolare, se la matrice ha tutti gli elementi non nulli, $sp=0$; se invece la matrice ha tutti gli elementi nulli, allora $sp=1$.

Un'altra proprietà che consente di operare una distinzione utile dal punto di vista computazionale è la presenza di un semplice criterio che consenta di determinare il posto degli elementi nulli o non nulli di una matrice. In tal caso la matrice viene detta *strutturata*.

Le seguenti matrici sono sparse e strutturate (la x indica gli elementi che possono essere non nulli).

La matrice *diagonale* è caratterizzata dal fatto che sono nulli gli elementi che non appartengono alla diagonale principale:

$$D = \begin{pmatrix} x & 0 & 0 & 0 \\ 0 & x & 0 & 0 \\ 0 & 0 & x & 0 \\ 0 & 0 & 0 & x \end{pmatrix}$$

e viene talvolta indicata con *diag(x,...,x)*. Il criterio è $a_{ij} = 0, i \neq j$.

Una particolare matrice diagonale è la matrice *identità*:

$$I = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{pmatrix}$$

In una matrice *tridiagonale* sono nulli gli elementi che non appartengono alla diagonale principale, alla prima diagonale superiore e alla prima diagonale inferiore:

$$T = \begin{pmatrix} x & x & 0 & 0 \\ x & x & x & 0 \\ 0 & x & x & x \\ 0 & 0 & x & x \end{pmatrix}$$

cioè $a_{ij} = 0, |i-j| > 1$.

Una matrice tridiagonale può essere considerata come un caso particolare di matrice *a banda*, in cui gli elementi non nulli compaiono solo su un certo numero di diagonali contigue. La seguente è una matrice a banda, con semiampiezza sinistra q e semiampiezza destra p:

$$B = \begin{array}{c} \\ q \left\{ \begin{array}{c} \\ \\ \\ \end{array} \right. \end{array} \overbrace{\begin{pmatrix} x & \cdots & x & 0 & \cdots & 0 \\ \vdots & \ddots & & \ddots & \ddots & \vdots \\ x & & \ddots & & \ddots & 0 \\ 0 & \ddots & & \ddots & & x \\ \vdots & \ddots & \ddots & & \ddots & \vdots \\ 0 & \cdots & 0 & x & \cdots & x \end{pmatrix}}^{p}$$

cioè $a_{ij} = 0$, $j-i > p$, $i-j > q$.

Una importante classe di matrici strutturate è quella delle matrici *triangolari,* distinte in triangolari *superiori* (gli elementi non nulli sono nel triangolo superiore e sono nulli tutti gli elementi del triangolo strettamente inferiore)

$$U = \begin{pmatrix} x & x & \dots & x \\ 0 & x & \dots & x \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & x \end{pmatrix}$$

$a_{ij} = 0 \, , i > j$

e in triangolari *inferiori* (gli elementi non nulli sono nel triangolo inferiore e sono nulli tutti gli elementi del triangolo strettamente superiore)

$$L = \begin{pmatrix} x & 0 & \dots & 0 \\ x & x & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ x & x & \dots & x \end{pmatrix}$$

$a_{ij} = 0 \, , \, i<j$.

La struttura delle matrici strettamente triangolari superiori e inferiori è ovvia.

Un esempio di matrice piena strutturata è fornito dalla classe delle matrici di *Toeplitz* in cui gli elementi su ogni diagonale sono uguali

$$V = \begin{pmatrix} d & e & f & g \\ c & d & e & f \\ b & c & d & e \\ a & b & c & d \end{pmatrix}$$

È possibile avere matrici in cui sono presenti contemporaneamente varie proprietà strutturali; la seguente matrice è tridiagonale di Toeplitz

$$\begin{pmatrix} 2 & -1 & 0 & 0 \\ 1 & 2 & -1 & 0 \\ 0 & 1 & 2 & -1 \\ 0 & 0 & 1 & 2 \end{pmatrix}$$

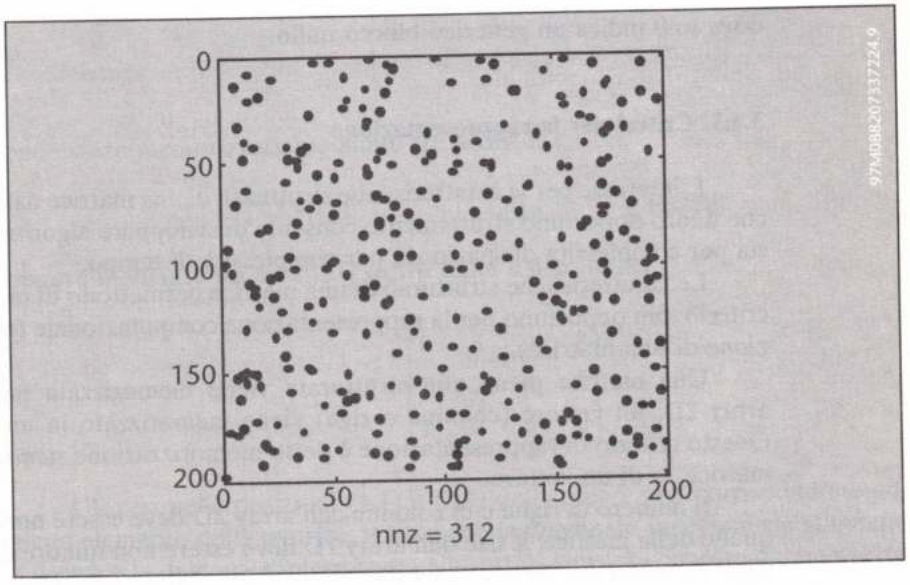
Una matrice sparsa non strutturata è una matrice in cui non esiste una semplice regola che consenta di individuare la posizione degli elementi nulli e non nulli, come per esempio la seguente matrice:

$$A = \begin{pmatrix} 6. & 0. & 0. & 3. & 0. \\ 4. & 1. & 0. & 0. & 2. \\ 7. & 0. & 6. & 0. & 8. \\ 0. & 0. & 9. & 5. & 0. \\ 0. & 0. & 0. & 0. & 3. \end{pmatrix}$$

Per una matrice sparsa non strutturata è utile considerare il *coefficiente di densità*, che indichiamo con d, definito come il rapporto tra il numero degli elementi non nulli e il numero complessivo degli elementi. La quantità $1 - d$ è allora uguale a sp.

Nel caso della matrice A, si ha che $nnz = 11$, $d = 0.44$ (cioè 44%) e $sp = 0.56$. Si noti che questa matrice è qui introdotta solo come esempio; infatti il suo ordine è piccolo e il suo coefficente di sparsità non è grande

Un esempio più significativo è il seguente, che mostra una matrice di ordine 200, in cui sono evidenziati con un punto solo gli elementi non nulli:



la sua densità è 0.0156 (1.56%) a cui corrisponde il coefficiente di sparsità 0.9844 (98.44%).

Nelle applicazioni concrete è frequente operare con matrici sparse di ordine elevato (>10000) e con alto coefficiente di sparsità (99.9%).

Talvolta un opportuno partizionamento a blocchi evidenzia una particolare caratteristica strutturale. Per esempio la matrice:

$$\begin{pmatrix} 2 & 4 & 9 & 0 & 0 & 0 \\ 3 & 0 & 6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 9 & 7 & 0 \\ 0 & 0 & 0 & 7 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 0 & 0 & 6 \end{pmatrix}$$

è *diagonale a blocchi*, con tre righe e tre colonne di blocchi, cioè

$$\begin{pmatrix} A_{11} & 0 & 0 \\ 0 & A_{22} & 0 \\ 0 & 0 & A_{33} \end{pmatrix}$$

dove lo 0 indica un generico blocco nullo.

3.4.3 Criteri per la rappresentazione

L'interesse per le caratteristiche strutturali di una matrice nasce dal fatto che il loro opportuno sfruttamento consente di sviluppare algoritmi efficienti sia per complessità di spazio sia per complessità di tempo.

Le caratteristiche strutturali di una matrice permettono di individuare il criterio più opportuno per la rappresentazione computazionale (*memorizzazione* di una matrice).

Una matrice piena non strutturata viene memorizzata mediante un array 2D; un vettore (colonna o riga) viene memorizzato in un array 1D. Questo criterio di rappresentazione è detto memorizzazione *standard* di una matrice o di un vettore.

Il numero di righe e di colonne dell'array 2D deve essere non minore di quello della matrice; il size dell'array 1D deve essere non minore del numero degli elementi del vettore. La scelta più comune è di far coincidere il numero di righe e di colonne dell'array con il numero di righe e di colonne della matrice, anche se talvolta può essere conveniente una memorizzazione su array di size maggiore del numero degli elementi della matrice.

Se la matrice è piena e strutturata, sparsa strutturata oppure sparsa non strutturata, è più efficiente ricorrere a memorizzazioni diverse da quella standard.

ESEMPIO:

una matrice piena strutturata di Toeplitz di m righe e n colonne viene memorizzazta mediante un array 1D di size $m+n-1$, che è il numero di diagonali della matrice. La matrice di Toeplitz V mostrata precedentemente viene memorizzata in un array 1D dal valore $[a, b, c, d, e, f, g]$.

ESEMPIO:

una matrice tridiagonale (che è una matrice sparsa strutturata) di ordine n viene memorizzata in tre array 1D di size $n, n-1$ e $n-1$, dove sono memorizzate rispettivamente, la diagonale principale della matrice, la prima diagonale superiore e la prima diagonale inferiore; una soluzione alternativa, anche se sostanzialmente equivalente, consiste nel memorizzare la matrice in un array 2D di tre righe e n colonne.

Per esempio la matrice tridiagonale

$$T = \begin{pmatrix} 7 & 5 & 0 & 0 \\ 1 & 3 & 2 & 0 \\ 0 & 2 & 4 & 8 \\ 0 & 0 & 6 & 1 \end{pmatrix}$$

può essere memorizzata mediante tre array 1D:

$$\texttt{d} = [7,3,4,1]\ ; \quad \texttt{f} = [5,2,8]\ ; \quad \texttt{c} = [1,2,6];$$

oppure mediante un array 2D di tre righe e 4 colonne:

$$\mathrm{T} = \begin{bmatrix} 0 & 5 & 2 & 8 \\ 7 & 3 & 4 & 1 \\ 1 & 2 & 6 & 0 \end{bmatrix}$$

Gli zeri nelle posizioni (1,1) e (3,4) dell'array `T` non corrispondono ad alcun elemento della matrice; il fatto che la diagonale superiore sia allineata a destra e la diagonale inferiore sia allineata a sinistra consente di memorizzare la j-sima colonna della matrice T nella j-sima colonna dell'array `T`.

In generale, una matrice a banda di ordine n e di ampiezza totale $q + p + 1$, viene memorizzata in un array 2D di $q + p + 1$ righe e di n colonne.

In modo analogo, le diagonali possono essere memorizzate ciascuna in una colonna dell'array 2D (invece che in una riga), cioè:

$$T = \begin{bmatrix} 0 & 7 & 5 \\ 1 & 3 & 2 \\ 2 & 4 & 8 \\ 6 & 1 & 0 \end{bmatrix}$$

ESEMPIO:

una matrice a banda, in cui alcune diagonali *interne* sono nulle, come nel caso

$$A = \begin{pmatrix} 2 & 0 & 3 & 0 & 0 \\ 1 & 5 & 0 & 4 & 0 \\ 0 & 2 & 6 & 0 & 5 \\ 0 & 0 & 1 & 2 & 0 \\ 0 & 0 & 0 & 3 & 4 \end{pmatrix}$$

dove la diagonale sopra la diagonale principale è interna e nulla, può essere memorizzata nel cosiddetto formato *diagonale con offset*, cioè mediante un array 2D che contiene, per colonne, gli elementi delle diagonali non nulle e un array 1D che indica quale diagonale è memorizzata in ogni colonna. L'array 2D `B` e l'array 1D `OFFSET` rappresentano la matrice A:

$$B = \begin{bmatrix} 0 & 2 & 3 \\ 1 & 5 & 4 \\ 2 & 6 & 5 \\ 1 & 2 & 0 \\ 3 & 4 & 0 \end{bmatrix}$$

OFFSET	-1	0	2

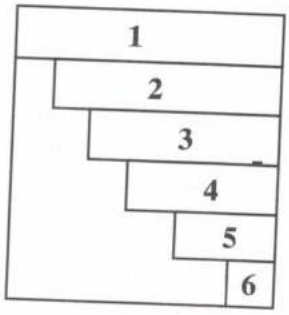
`OFFSET(1)`, cioè `-1`, indica che la prima colonna di `B` contiene la prima diagonale al di sotto della diagonale principale di `A`; `OFFSET(2)`, cioè `0`, indica che la seconda colonna di `B` contiene la diagonale principale; `OFFSET(3)`, cioè `2`, indica che la terza colonna di `B` contiene la seconda diagonale al di sopra della diagonale principale.

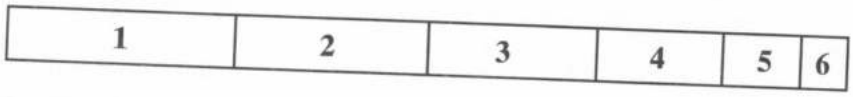
In questo modo l'elemento `B(i,j)` dell'array contiene l'elemento di A di indici `i,i+OFFSET(j)`.

ESEMPIO:

una matrice triangolare viene considerata piena se è di ordine piccolo (< 1000), mentre viene considerata sparsa se è di ordine elevato. Nel primo caso, viene memorizzata in modo standard. Nel secondo caso, la matrice viene memorizzata in un array 1D, secondo il cosiddetto criterio *compattato* (*packed form*).

L'esempio si riferisce alla memorizzazione di una matrice triangolare superiore di ordine 6:





Il size dell'array 1D è uguale al numero degli elementi del triangolo superiore di una matrice quadrata, cioè

$$n+(n-1)+\ldots+2+1=n(n+1)/2\,.$$

Nel caso di matrice sparsa non strutturata (e quindi anche di vettori sparsi) esistono varie tecniche che possono essere utilizzate per una rappresentazione computazionale efficiente.

Si consideri per esempio la matrice, già introdotta in 3.5.2,

$$A=\begin{pmatrix} 6. & 0. & 0. & 3. & 0. \\ 4. & 1. & 0. & 0. & 2. \\ 7. & 0. & 6. & 0. & 8. \\ 0. & 0. & 9. & 5. & 0. \\ 0. & 0. & 0. & 0. & 3. \end{pmatrix}$$

Il criterio di memorizzazione più semplice è il *formato a coordinate*. Esso consiste di tre array 1D: il primo contiene tutti gli elementi non nulli della matrice; il secondo contiene gli indici di colonna di tali elementi; il terzo contiene gli indici di riga di tali elementi.

La matrice A nel formato a coordinate è rappresentata dai tre array:

```
A_nonzero        [3. 2. 6. 4. 1. 6. 8. 9. 5. 3. 7.]
indice_colonna   [4  5  3  1  2  1  5  3  4  5  1 ]
indice_riga      [1  2  3  2  2  1  3  4  4  5  3 ]
```

L'elemento in `A_nonzero(k)` ha per *coordinate* il valore di `indice_riga(k)`, e il valore di `indice_colonna(k)`; tali valori sono la coppia di indici che individuano il posto nella matrice A. Per esempio, `A_nonzero(7)` contiene il valore 8., che è l'elemento di A di posto `(3,5)`, cioè i valori di `indice_riga(7)` e di `indice_colonna(7)`.

Il size dei tre array è `nnz`; inoltre, `A_nonzero` è un array di tipo reale, mentre gli altri due array sono di tipo intero.

Infine, gli elementi in `A_nonzero` non devono necessariamente rispettare l'ordine di posizione nella matrice, anche se spesso sono memorizzati per righe o per colonne.

La memorizzazione per righe dà luogo a

```
A_nonzero        [6. 3. 4. 1. 2. 7. 6. 8. 9. 5. 3.]
indice_colonna   [1  4  1  2  5  1  3  5  3  4  5 ]
indice_riga      [1  1  2  2  2  3  3  3  4  4  5 ]
```

Si può osservare che l'array `indice_riga` contiene informazioni ridondanti; ciò è ancora più evidente nel caso di matrici di ordine molto elevato (>10000), in cui il numero di elementi non nulli di una riga può essere grande.

Questa osservazione porta al criterio di memorizzazione nel *formato a riga sparsa compressa*, che è una semplice modifica del formato a coordinate. Il criterio consiste nell'uso di tre array 1D; i primi due array contengono gli elementi non nulli e i loro indici di colonna, come nel formato a coordinate per righe; il terzo array contiene solo `m+1` indici (`m` è il numero di righe di A), che indicano dove è memorizzato, nei primi due array, il primo elemento non nullo delle varie righe di A.

La matrice A nel formato a riga sparsa compressa è rappresentata dai tre array:

```
A_nonzero        [6. 3. 4. 1. 2. 7. 6. 8. 9. 5. 3.]
indice_colonna   [1  4  1  2  5  1  3  5  3  4  5 ]
inizio_riga      [1  3  6  9  11 12]
```

L'elemento in `A_nonzero(k)` ha come indice di colonna (in A) il valore di `indice_colonna(k)`; il suo indice di riga non è memorizzato esplicitamente, ma deve essere ricostruito esaminando il valore dell'array `inizio_riga`. Infatti, il valore di `inizio_riga(r)` è un puntatore logico all'array `A_nonzero` che indica dove inizia la memorizzazione degli elementi non nulli della `r`-sima riga della matrice A. L'ultimo elemento di `inizio_riga` è il size di `A_nonzero + 1`.

Nell'esempio in esame, poichè il valore di `inizio_riga(1)` è 1, si ha che la *prima* riga inizia con `A_nonzero(1)`; poichè il valore di `inizio_riga(2)` è 3, si ha che la *seconda* riga inizia con `A_nonzero(3)`; e così via.

Il numero di elementi non nulli nella `r`-sima riga è dato da `inizio_riga(r+1)- inizio_riga(r)`; quelli dell'ultima riga sono `inizio_riga(m+1)-inizio_riga(m)`.

In `A_nonzero` gli elementi non nulli sono memorizzati per riga.

In modo del tutto analogo, si può definire il *formato a colonna sparsa compressa*, basato sugli array `A_nonzero` e `indice_riga`, in cui viene memorizzato esplicitamente solo l'indice di riga degli elementi non nulli, mentre un terzo array `inizio_colonna` punta all'elemento iniziale non nullo delle varie colonne.

Un altro criterio di memorizzazione è il *formato Itpack*. Questo formato risulta vantaggioso se il massimo numero di elementi non nulli per riga (che denoteremo con `nnzr`) è piccolo rispetto all'ordine della matrice.

Il criterio consiste nell'uso di due array 2D di `m` righe e `nnzr` colonne. Il primo array contiene riga per riga gli elementi non nulli della matrice; il secondo array, di tipo intero, contiene gli indici di colonna dei corrispondenti elementi del primo array.

La matrice A nel formato Itpack è rappresentata dai due array `A_nonzero` e `indice_colonna`.

Le righe di A che hanno un numero di elementi non nulli minore di `nnzr` danno luogo a righe di `A_nonzero` contenenti elementi nulli, che sono fittizi; ciò viene indicato dalla presenza di un indice di colonna fittizio (=0) nell'array `indice_colonna`.

$$\texttt{A_nonzero} = \begin{bmatrix} 6. & 3. & 0. \\ 4. & 1. & 2. \\ 7. & 6. & 8. \\ 9. & 5. & 0. \\ 3. & 0. & 0. \end{bmatrix} \qquad \texttt{indice_colonna} = \begin{bmatrix} 1 & 4 & 0 \\ 1 & 2 & 5 \\ 1 & 3 & 5 \\ 3 & 4 & 0 \\ 5 & 0 & 0 \end{bmatrix}$$

I *vettori sparsi*, che nelle applicazioni sono quasi sempre non strutturati, sono in genere memorizzati con la rappresentazione standard, cioè in un array 1D di size uguale al numero degli elementi del vettore e con gli elementi nulli memorizzati esplicitamente.

In situazioni particolari, quando il vettore ha un elevato numero di elementi e un coefficiente di densità molto piccolo, può essere conveniente ricorrere alla rappresentazione in *formato a coordinate*, analoga a quella delle matrici. Essa consiste di due array 1D, in cui sono memorizzati gli elementi non nulli e i loro indici. Per esempio il vettore sparso

$$x = (0,0,6,0,0,0,2,3,0,0,0,0,0,8,0,5,0,0,1,0,0,0,0,0,4)^T$$

è rappresentato mediante gli array:

```
x_nonzero     [6 2 3  8  5  1  4]
indice        [3 7 8 14 16 19 25]
```

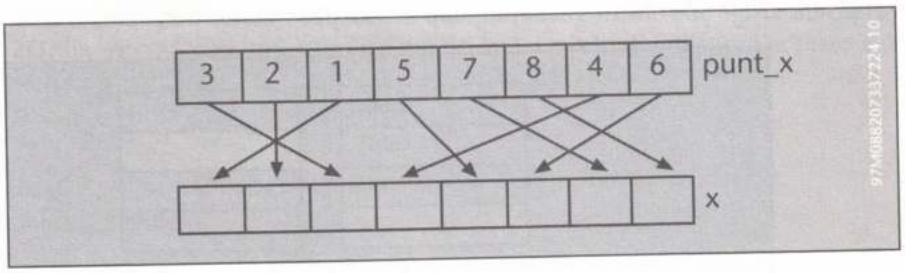
Si noti che in ogni caso (memorizzazione standard o formato a coordinate) i vettori riga e i vettori colonna sono rappresentati nello stesso modo.

La distinzione tra vettori riga e colonna avviene generalmente a livello di progetto dell'algoritmo e non viene esplicitamente rappresentata nella struttura dati. Nei problemi in cui è necessario memorizzare l'informazione sulla struttura (riga o colonna) di un vettore, si può ricorrere a una banale codifica convenzionale di tale informazione, per esempio memorizzando una coppia di interi che indica il numero delle righe e delle colonne.

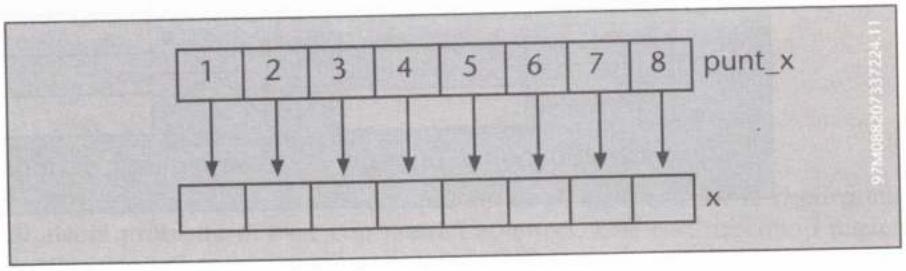
In alcuni casi, si ha a che fare con vettori e matrici pieni non strutturati e memorizzati in formato standard, ma gli array 1D e 2D usati per la rappresentazione contengono gli elementi dei vettori e delle matrici in un ordine diverso: in altre parole, i generici elementi x_i e a_{ij} non necessariamente vengono memorizzati in `x(i)` e `A(i,j)`, dove `x` e `A` sono array 1D e 2D. Questo fatto si suole sintetizzare dicendo che *l'ordine logico non coincide con l'ordine fisico*. In tal caso la memorizzazione richiede, oltre all'array dei valori, uno (o più) array di indici, detti *indici puntatori*, che consentono di conservare la corrispondenza tra posizione fisica (nell'array) con quella logica (nel vettore o nella matrice).

Per esempio, sia `x` l'array 1D dei valori e sia `punt_x` l'array 1D degli indici puntatori, come nell'esempio in figura; allora si ha che il primo elemento (nell'ordinamento logico) di `x` trova al posto 3 (cioè x_1 è memorizzato in `x(3)`), come indicato dal valore di `punt_x(1)`, cioè è il valore associato a `x(punt_x(1))`; il secondo elemento (nell'ordinamento logico) di `x` si trova al posto 2, come indicato dal valore di `punt_x(2)`, cioè è `x(punt_x(2))`; il terzo elemento (nell'ordinamento logico) di `x` si trova al posto 1, come indicato dal valore di `punt_x(3)`, cioè è `x(punt_x(3))`; e così via. La

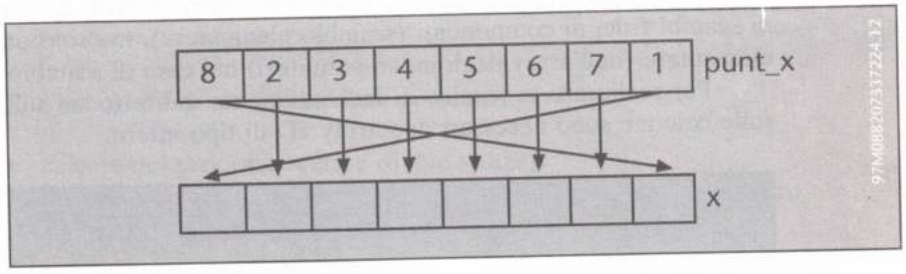
frecce riportate in figura vogliono solo enfatizzare il *significato* degli indici puntatori.



Ad esempio, per scambiare (logicamente) il primo e l'ultimo elemento di un array si procede come schematizzato nelle seguenti figure:



prima: ordinamento logico e fisico coincidono

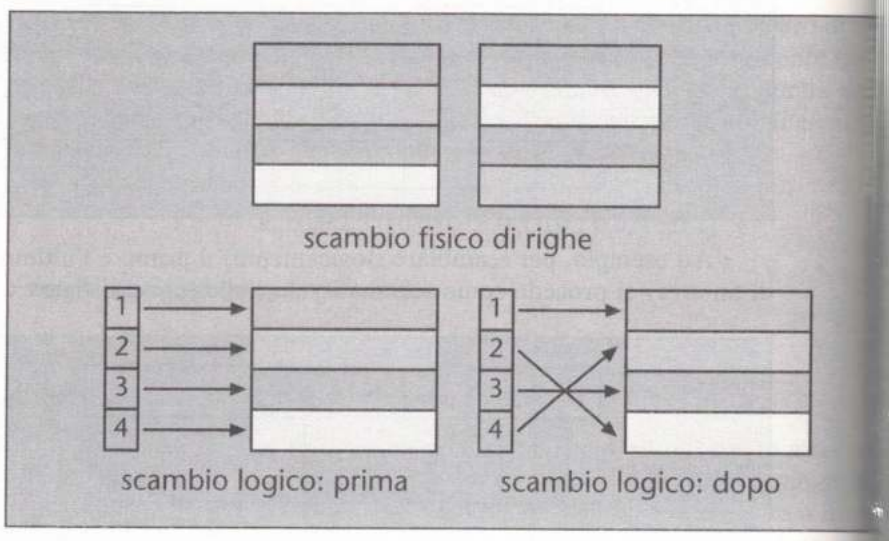


dopo: il primo e l'ultimo elemento sono stati scambiati (logicamente)

Quando un elemento di un array è individuato attraverso il valore di un elemento di un altro array (come nel caso di `x(punt_x(i))`) si dice che c'è un *indirizzamento indiretto* dell'array x.

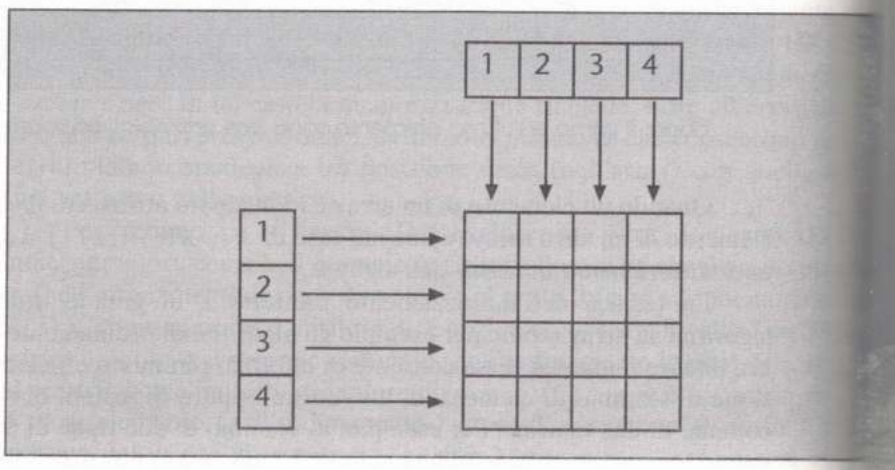
La tecnica dell'indirizzamento indiretto è di grande utilità in vari algoritmi su array (come per esempio gli algoritmi di ordinamento) e di algebra lineare numerica. Essa consente di effettuare in modo efficiente l'operazione di scambio di elementi di un vettore oppure di scambi di righe e/o di colonne di una matrice. Per esempio, lo scambio di due righe di una matrice piena, memorizzata in modalità standard, può essere effettuato senza scam-

biare le corrispondenti righe dell'array 2D (scambio fisico), ma solo scam biando i corrisponedenti elementi dell'array 1D degli indici puntato all'array 2D.



L'uso degli indici puntatori aumenta la complessità di spazio di algoritmo, ma può ridurre la complessità di tempo, rispetto all'operazione scambio. Infatti lo scambio di due righe di un array 2D di m colonne richie m scambi fisici di componenti (scambio elementare), ma solo uno scam elementare (nell'array degli indici puntatori) nel caso di scambio logico.

Per realizzare la tecnica di indirizzamento indiretto sia sulle righe sulle colonne sono necessari due array 1D di tipo intero.



Il primo, `ind_riga`, ha size uguale al numero di righe dell'array 2D; Il secondo, `ind_colonna`, ha size uguale al numero di colonne dell'array 2D. Tale tecnica può essere utilizzata in qualunque algoritmo che agisce sull'array 2D, sia `array`, con una semplice modifica del modo di indicare l'accesso agli elementi dell'array:
sostituire tutti gli accessi agli elementi di `array`, del tipo

```
array(i,j)
```

con i seguenti

```
array(ind_riga(i),ind_colonna(j))
```

Gli eventuali scambi sono effettuati solo sui due array di indici puntatori `ind_riga(i)`, `ind_colonna` e non su `array`.

3.4.4 Nuclei computazionali di base

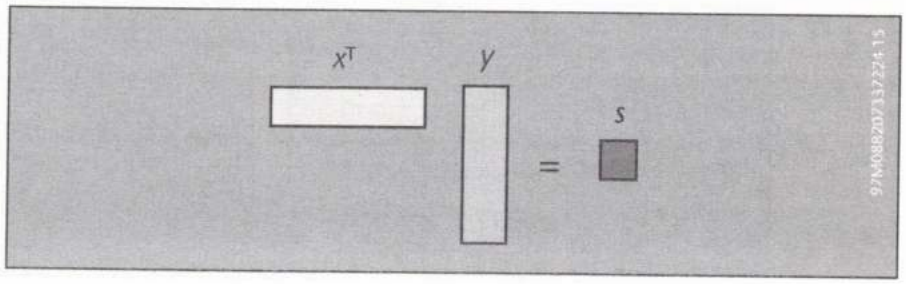
In questo paragrafo vengono presentati gli algoritmi per la risoluzione di alcuni problemi di base con matrici e vettori. Essi costituiscono i nuclei computazionali di ogni problema applicativo che coinvolge matrici e vettori.

La struttura degli algoritmi dipende in modo sostanziale dalle tecniche per la rappresentazione di matrici e vettori, cioè dal criterio di memorizzazione che viene utilizzato.

Vengono discussi i seguenti problemi:

- calcolo del prodotto scalare di due vettori;
  il prodotto scalare di due vettori x e y di n elementi è il numero

$$s = x^T y = \sum_{i=1}^{n} x_i \cdot y_i = x_1 y_1 + x_2 y_2 + \ldots + x_n y_n$$



- calcolo della somma di un vettore e di un multiplo di un altro vettore

$$z = x + \alpha y = (x_1 + \alpha y_1, x_2 + \alpha y_2, \ldots, x_n + \alpha y_n)^T$$

- calcolo della somma di due matrici $C = A + B$

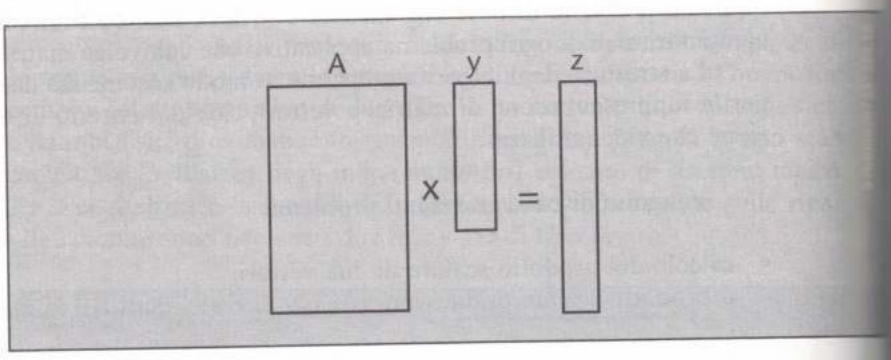
$$c_{ij} = a_{ij} + b_{ij} \qquad i = 1:m, j = 1:n$$

con A, B, C matrici di m righe e n colonne

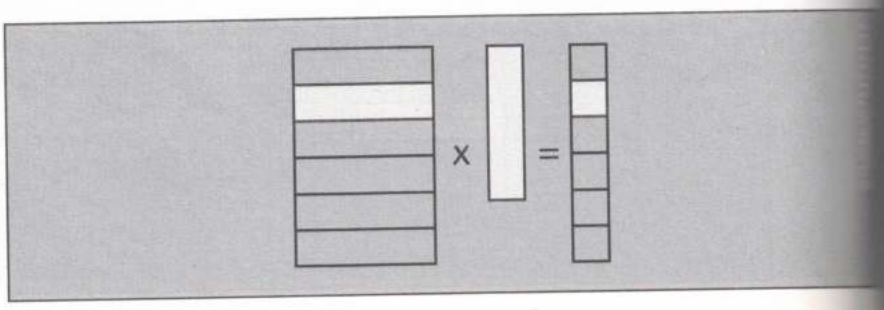
- calcolo del prodotto di una matrice per un vettore $Ay = z$

$$z_i = \sum_{j=1}^{n} a_{i,j} y_j \quad , i = 1:m$$

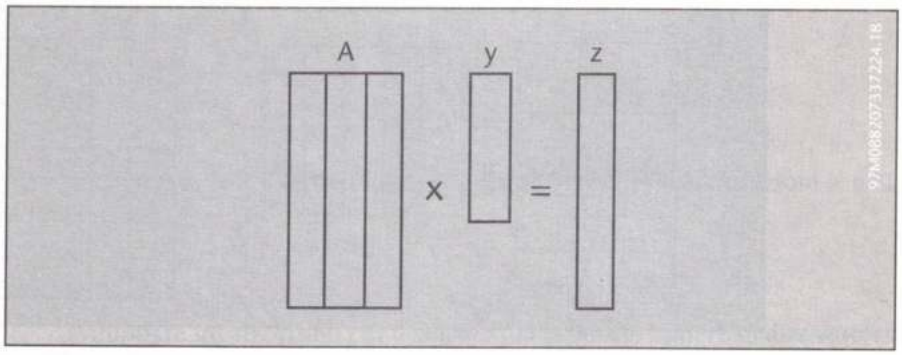
con A matrice di m righe e n colonne, y vettore di n componenti, z vettore di m componenti

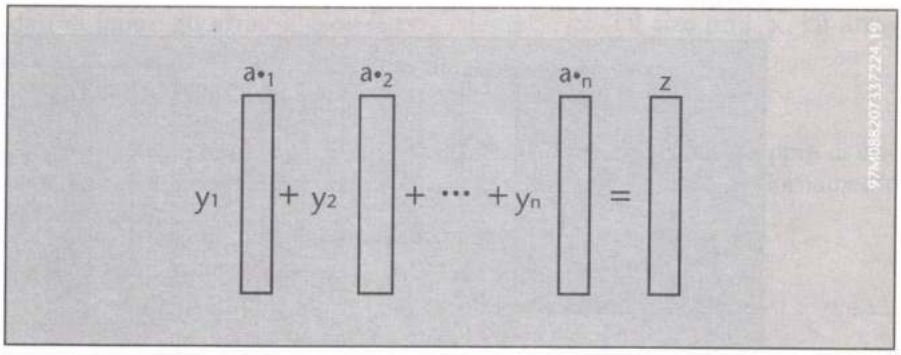


Il prodotto è definito in termini di prodotti scalari; infatti z_i è il prodotto scalare della i-sima riga di A e il vettore y.



Il prodotto può essere anche visto in termini di somma di un multiplo di un vettore a un altro vettore:

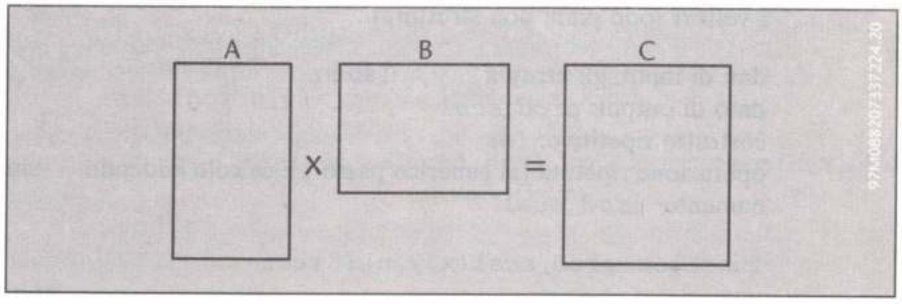




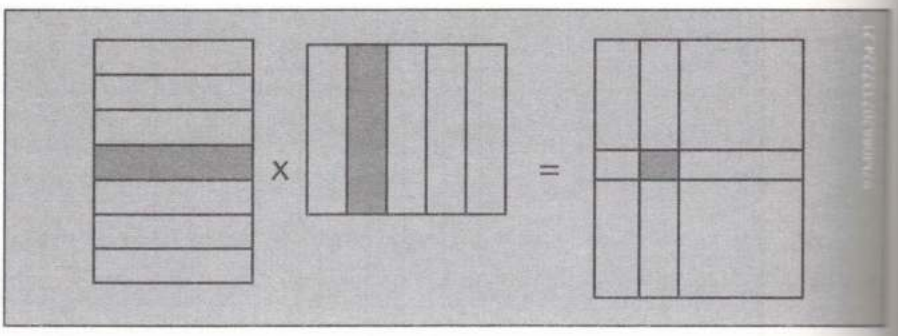
cioè il vettore prodotto è la combinazione lineare delle colonne della matrice A con coefficienti costituiti dalle componenti di y, cioè

$$Ay = (a_{\bullet 1}, \ldots, a_{\bullet n})y = a_{\bullet 1}y_1 + \ldots + a_{\bullet n}y_n = z$$

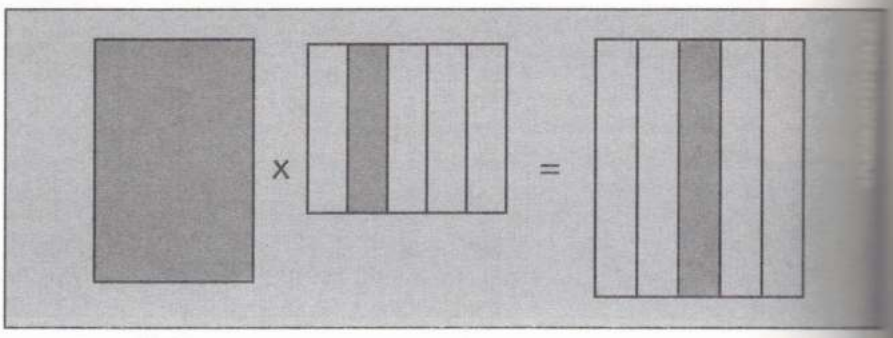
- calcolo del prodotto di due matrici $A = AB$, con A matrice di m righe e n colonne, B matrice di n righe e l colonne, C matrice di m righe e l colonne.



Il prodotto è definito in termini di prodotti scalari; infatti c_{ij} è il prodotto scalare della i-sima riga di A e della j-sima colonna di B.



Il prodotto può essere anche visto in termini di prodotti matrice per vettore



cioè, j-sima colonna di C è data dal prodotta della matrice A per la j-sima colonna di B

$$AB = A(b_{\bullet 1}, \ldots, b_{\bullet l}) = (Ab_{\bullet 1}, \ldots, Ab_{\bullet l}) = C = (c_{\bullet 1}, \ldots, c_{\bullet l})$$

ESEMPIO:

sviluppare un algoritmo per il calcolo del prodotto scalare di due vettori; i vettori sono pieni non strutturati.

dati di input: gli array `x, y,` il size `n`.
dato di output: `prod_scal`
costrutto ripetitivo: `for`
operazione ripetuta (al generico passo `i`): calcolo addendo `i`-simo e aggiornamento `prod_scal`.

```
function prod_scal(x,y,n): real
var n: integer
```

```
var x,y: array(1..n) of real
begin
      prod_scal := 0.0
      for i=1,n do
            prod_scal := prod_scal + x(i)*y(i)
      endfor
end
```

La complessità di tempo di questo algoritmo è `n` moltiplicazioni e `n-1` addizioni.

ESEMPIO:

sviluppare un algoritmo per il calcolo del prodotto scalare di due vettori; i vettori sono sparsi e rappresentati con il formato a coordinate.

dati di input: gli array `x_nonzero`, `indice_x`, il size `nnz_x`, gli array `y_nonzero`, `indice_y`, il size `nnz_y`.
dato di output: `prod_scal_sp`
costrutto ripetitivo: `while`
operazione ripetuta (al generico passo): individuazione di una coppia di elementi non nulli di ugual posto dei due vettori, eventuale aggiornamento `prod_scal_sp`.

```
function prod_scal_sp(x_nonzero, ....
        indice_x,nnz_x,y_nonzero,indice_y,nnz_y): real
var nnz_x,nnz_y,k_y: integer
var x_nonzero: array(1..nnz_x) of real
var y_nonzero: array(1..nnz_y) of real
var indice_x: array(1..nnz_x) of integer
var indice_y: array(1..nnz_y) of integer
begin
  prod_scal_sp := 0.0
  k_y := 1
  for k_x=1,nnz_x do
      while indice_y(k_y)<indice_x(k_x) and k_y<nnz_y
          k_y := k_y + 1
      endwhile
      if indice_y(k_y) = indice_x(k_x)
       then
        prod_scal_sp := prod_scal_sp + x_nonzero(k_x) * ...
                        y_nonzero(k_y)
      endif
  endfor
end
```

La complessità di tempo di questo algoritmo è min(nnz_x, nnz_y) moltiplicazioni e addizioni.

ESEMPIO:

sviluppare un algoritmo per il calcolo del prodotto scalare di due vettori; il primo è pieno (rappresentazione standard), il secondo è sparso, (rappresentazione in formato compresso).

dati di input: l'array x, il size n, gli array y_nonzero, indice_y, il size nnz_y.
dato di output: prod_scal_fsp
costrutto ripetitivo: while
operazione ripetuta (al generico passo): individuazione di una coppia di elementi non nulli di ugual posto dei due vettori, eventuale aggiornamento prod_scal_sp.

```
function prod_scal_fsp(x,n,y_nonzero,indice_y, ...
                                nnz_y): real
var n,nnz_y: integer
var x: array(1..n) of real
var y_nonzero: array(1..nnz_y) of real
var indice_y: array(1..nnz_y) of integer
begin
    prod_scal_fsp := 0.0
    for k=1,nnz_y do
         prod_scal_fsp:= prod_scal_fsp + y_nonzero(k)*...
                        x(indice_y(k))
    endfor
end
```

Si noti l'indirizzamento indiretto dell'array x. La complessità di tempo di questo algoritmo è nnz_y moltiplicazioni e addizioni.

ESEMPIO:

sviluppare un algoritmo per il calcolo della somma di un vettore e di un multiplo di un altro vettore; i vettori sono pieni (memorizzazione standard).

dati di input: gli array x, y, il size n, il numero α
dato di output: l'array z
costrutto ripetitivo: for
operazione ripetuta (al generico passo i):
calcolo del valore della componente i-sima dell'array z, cioè del valore $z_i = x_i + \alpha y_i$.

```
procedure vet_xpay(in: x,y,n,alfa; out: z)
var n: integer
var x,y,z: array(1..n) of real
var alfa: real
begin
      for i=1,n do
         z(i) := x(i) + alfa * y(i)
      endfor
end
```

La complessità di tempo di questo algoritmo è `n` moltiplicazioni e `n` addizioni.

L'algoritmo può essere immediatamente modificato per risolvere il problema del cosiddetto *aggiornamento in place*:

$$x \leftarrow x + \alpha y$$
$$x = (x_1 + \alpha y_1, x_2 + \alpha y_2, \ldots, x_n + \alpha y_n)^T$$

ESEMPIO:

sviluppare un algoritmo per il calcolo della somma di un vettore e di un multiplo di un altro vettore (aggiornamento *in place*); i vettori sono sparsi (rappresentazione in formato a coordinate non ordinato).

dati di input: gli array `x_nonzero`, `indice_x`, il size `nnz_x`, gli array `y_nonzero`, `indice_y`, il size `nnz_y`, il numero α, l'ordine `n` dei vettori
dato di output: gli array `x_nonzero`, `indice_x`

L'idea è di

utilizzare un array locale (sia `y_pieno`) per rappresentare in formato standard uno dei due vettori (sia `y`);

accedere sequenzialmente alle componenti di `x_nonzero`; se il corrispondente elemento di `y_pieno` è non nullo, la componente di `x_nonzero` è aggiornata e l'elemento di `y_pieno` diventa zero;

accedere sequenzialmente alle componenti di `y_nonzero`; se il corrispondente elemento di `y_pieno` è non nullo, viene aggiunta una nuova componente alla struttura dati `x_nonzero` e l'elemento di `y_pieno` diventa zero.

Si noti che in questo problema il numero delle componenti non nulle in output del vettore x può essere maggiore del numero delle sue componenti non nulle in input.

È importante osservare che, poichè il tipo array in P-like richiede che il numero di componenti sia predefinito e non modificabile, l'array non è la scelta ottimale per la rappresentazione dei dati di questo problema.

Tuttavia, una soluzione di ragionevole compromesso[14] può essere ottenuta dichiarando array di size maggiore del numero degli elementi non nulli in input del vettore x (nella procedura `vet_xpay_sp` si è assunto come size `2*nnz_x`, nell'ipotesi che il numero di elementi non nulle di x sia non minore di quello di y), in modo da poter ospitare i nuovi elementi non nulli.

Questo fenomeno di aumento del numero degli elementi non nulli in un vettore sparso (o in una matrice sparsa) è detto *fill-in* (*riempimento*).

```
procedure vet_xpay_sp(in: y_nonzero,indice_y,nnz_y, ....
            alfa, n; inout: x_nonzero,indice_x,nnz_x)
var nnz_x,nnz_y,n: integer
var x_nonzero: array(1..2*nnz_x) of real
var indice_x: array(1..2*nnz_x) of integer
var y_nonzero: array(1..nnz_y) of real
var indice_y: array(1..nnz_y) of integer
var alfa: real
var y_pieno: array(1..n) of real
var i:integer
begin
% costruzione della rappresentazione standard di y
   for k=1,n do
      y_pieno(k) := 0.0
   endfor
   for k=1,nnz_y do
      y_pieno(indice_y(k)):= y_nonzero(k)
   endfor
% eventuale aggiornamento degli elementi di x_nonzero
   for k=1,nnz_x do
      i := indice_x(k)
      if y_pieno(i) ~= 0.0
         then
            x_nonzero(k) := x_nonzero(k)+alfa*y_pieno(i)
            y_pieno(i) := 0.0
       endif
   endfor
% determinazione del fill-in nell'array x_nonzero
   for k=1,nnz_y do
      i := indice_y(k)
      if y_pieno(i) ~= 0.0
```

[14] In P-like l'array è allocato staticamente; la soluzione più efficiente richiede l'allocazione dinamica.

```
            then
               nnz_x := nnz_x+1
               x_nonzero(nnz_x) := alfa*y_pieno(i)
               indice_x(nnz_x) := i
               y_pieno(i) := 0.0
         endif
      endfor
   end
```

ESEMPIO:

sviluppare un algoritmo per il calcolo del prodotto di una matrice per un vettore; la matrice e il vettore sono pieni non strutturati.

dati di input: gli array `A, y,` il numero di righe `m`, il numero di colonne `n`
dato di output: l'array `z`
costrutto ripetitivo: `for`
operazione ripetuta (al generico passo `i`): calcolo del valore della componente `i`-sima dell'array `z`, cioè del prodotto scalare tra la riga `i`-sima di A e y.

```
procedure mat_vet(in: A,m,n,y; out: z)
var m,n: integer
var A: array(1..m,1..n) of real
var y: array(1..n) of real
var z: array(1..m) of real
begin
      for i=1,m do
% calcolo del prodotto scalare
         z(i) := 0.0
         for j=1,n do
            z(i) := z(i) + A(i,j) * y(j)
         endfor
      endfor
end
```

La complessità di tempo di questo algoritmo è $m \cdot n$ moltiplicazioni e $m \cdot n$ addizioni.

ESEMPIO:

sviluppare un algoritmo per il calcolo del prodotto di una matrice per un vettore (la matrice e il vettore sono pieni non strutturati), usando la procedura `vet_xpay(in: x,y,n,alfa; out: z)`.

dati di input: gli array A, y, il numero di righe m, il numero di colonne n
dato di output: l'array z
costrutto ripetitivo: for
operazione ripetuta (al generico passo j): sommare al vettore delle somme parziali la j-sima colonna di A moltiplicata per y_j.

```
procedure mat_vet(in: A,m,n,y; out: z)
var m,n: integer
var A: array(1..m,1..n) of real
var y: array(1..n) of real
var z: array(1..m) of real
begin
      for i=1,m do
         z(i) := 0.0
      endfor
      for j=1,n do
         vet_xpay(z,A(1..m,j),m,y(j),z)
      endfor
end
```

La notazione di porzione di array A(1..m,j) indica la j-sima colonna dell'array A.

Si noti che l'algoritmo precedente e la seguente porzione di algoritmo

```
begin
 for i=1,m do
    z(i) := 0.0
 endfor
 for j=1,n do
     for i=1,m do
         z(i) := z(i) + A(i,j)*y(j)
     endfor
 endfor
end
```

determinano la medesima sequenza computazionale.

La complessità di tempo è m·n moltiplicazioni e m·n addizioni.

ESEMPIO:

sviluppare un algoritmo per il calcolo del prodotto di una matrice per un vettore; la matrice è sparsa non strutturata (rappresentazione in formato a riga sparsa compressa) e i vettori sono pieni (rappresentazione standard).

dati di input: gli array `A_nonzero, indice_colonna, inizio_riga, nnz, y,` il numero di righe `m`, il numero di colonne `n`
dato di output: l'array `z`
costrutto ripetitivo: `for`
operazione ripetuta (al generico passo `i`): calcolo del valore della componente `i`-sima dell'array `z`, cioè del prodotto scalare tra la riga `i`-sima di A e y.

```
procedure mat_vet_sp(in: A_nonzero,nnz, ....
            indice_colonna, inizio_riga, y,m,n; out: z)
var nnz,m,n: integer
var A_nonzero: array(1..nnz) of real
var indice_colonna: array(1..nnz) of integer
var inizio_riga: array(1..m) of integer
var y: array(1..n) of real
var z: array(1..m) of real
var primo,ultimo: integer
begin
   for i=1,m do
% individuazione riga i-sima in A_nonzero
   primo := inizio_riga(i)
   ultimo := inizio_riga(i+1) - 1
% calcolo del prodotto scalare
   z(i) := 0.0
   for j=primo,ultimo do
      z(i) := z(i)+A_nonzero(j)*y (indice_colonna(j))
   endfor
  endfor
end
```

La complessità di tempo di questo algoritmo è data dal numero degli elementi non nulli della matrice, cioè `nnz` moltiplicazioni e O `(nnz)` addizioni. La complessità può anche essere espressa in termini del coefficiente di sparsità s della matrice e risulta essere $m \cdot n \cdot (1 - s)$.

ESEMPIO:

sviluppare un algoritmo per il calcolo del prodotto di una matrice quadrata per un vettore; la matrice è sparsa strutturata (rappresentazione in formato diagonale con offset e i vettori sono pieni (rappresentazione standard).

```
procedure mat_vet_off(in: B,n,n_diag,offset,y; out: z)
var n,n_diag: integer
var B: array(1..n,1..n_diag) of real
```

```
var offset: array(1..n_diag) of integer
var y,z: array(1..n) of real
var k: real
begin
      for i=1,n do
         z(i) := 0.0
         for j=1,n_diag do
              k:=i+offset(j)
              if k>0 and k<=n
                 then
                    z(i):=z(i)+B(i,j)*y(k)
              endif
         endfor
      endfor
end
```

La complessità di tempo di questo algoritmo è data dal numero degli elementi non nulli della matrice, cioè `nnz` moltiplicazioni e addizioni.

ESEMPIO:

sviluppare un algoritmo per il calcolo del prodotto di due matrici; le matrici sono piene non strutturate.

dati di input: gli array `A`, `B`, il numero `m` di righe di `A`, il numero `n` di colonne di `A`, il numero `l` di colonne di `B`
dato di output: l'array `C`
costrutto ripetitivo: `for i=1,m; for j=1,l` (innestati)
operazione ripetuta (al generico passo `i,j`): calcolo del valore della componente `i,j`-sima dell'array `C`, cioè del prodotto scalare tra la riga `i`-sima di `A` e la colonna `j`-sima di `B`.

```
procedure mat_mat(in: A,m,n,B,l; out: C)
var m,n,l: integer
var A: array(1..m,1..n) of real
var B: array(1..n,1..l) of real
var C: array(1..m,1..l) of real
begin
      for i=1,m do
         for j=1,l do
% calcolo del prodotto scalare
            C(i,j) := 0.0
            for k=1,n do
               C(i,j) := C(i,j) + A(i,k) * B(k,j)
```

```
            endfor
        endfor
    endfor
end
```

La complessità di tempo di questo algoritmo è `m·n·l` moltiplicazioni e addizioni.

Un importante aspetto del calcolo del prodotto di due matrici è la domanda circa l'ottimalità, dal punto di vista dell'efficienza, dell'algoritmo precedente e quindi la complessità computazionale intrinseca del problema della moltiplicazione di due matrici. La soprendente risposta è che

- esistono algoritmi con complessità minore di `m·n·l` moltiplicazioni;
- non è stata ancora individuata né una limitazione inferiore di complessità, né il corrispondente algoritmo, che consenta di stabilire la complessità intrinseca del problema.

Il famoso algoritmo di Strassen (dal nome del matematico tedesco Volker Strassen, che lo sviluppò nel 1969) è un esempio di algoritmo di prodotto con complessità di tempo minore di `m·n·l`. Nel caso di matrici quadrate di ordine n, che tratteremo brevemente, la sua complessità è $O(n^{2.807})$.
Siano A,B,C matrici di ordine $n = 2^k$, con k intero positivo.
Se $k = 1$, allora

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

posto:

$$\begin{aligned} & s_1 = (a_{11} + a_{22})(b_{11} + b_{22}) \ ; \ s_2 = (a_{21} + a_{22})b_{11} \ ; \ s_3 = a_{11}(b_{12} - b_{22}) \ ; \\ & s_4 = a_{22}(b_{21} - b_{11}) \ ; \\ & s_5 = (a_{11} + a_{12})b_{22} \ ; \ s_6 = (a_{21} - a_{11})(b_{11} + b_{12}) \ ; \ s_7 = (a_{12} - a_{22})(b_{21} + b_{22}) \end{aligned} \tag{1}$$

allora:

$$\begin{aligned} c_{11} &= s_1 + s_4 - s_5 + s_7 & c_{12} &= s_3 + s_5 \\ c_{21} &= s_2 + s_4 & c_{22} &= s_1 - s_2 + s_3 + s_6 \end{aligned} \tag{2}$$

con un costo di 7 moltiplicazioni e 18 addizioni.
Se $k > 1$, allora si partizionano le matrici A,B,C ognuna in quattro blocchi quadrati di ordine $n/2$

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \times \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

e si applicano le formule (1) e (2), estendendole ai blocchi invece che alle singole componenti.
La complessità di tempo dell'algoritmo di Strassen si può ricavare osservando che

il prodotto di due matrici di ordine n si riduce al calcolo di sette prodotti di matrici di ordine $n/2$, cioè

$$T(n) = 7 \cdot T\left(\frac{n}{2}\right)$$

Poichè $n = 2^k$ e $T(1) = 1$, si ha

$$T(n) = 7 \cdot T\left(\frac{n}{2}\right) = 7\left(7 \cdot T\left(\frac{n}{4}\right)\right) = 7^2 T\left(\frac{n}{4}\right) = 7^3 T\left(\frac{n}{8}\right) = \ldots$$
$$= 7^k T(1) \equiv 7^k$$

La quantità $7^k \equiv 7^{\log_2 n}$ può essere facilmente espressa in termini di n, la dimensione computazionale del problema:

$$7^{\log_2 n} = n^x \Rightarrow x = \log_2(7) = 2.807...$$

e quindi si può concludere che

$$T(n) = n^{2.807}$$

I problemi elementari di calcolo delle matrici esaminati in questo paragrafo costituiscono i nuclei computazionali di base di ogni problema applicativo che coinvolga matrici e vettori. L'implementazione di tali algoritmi in un sofware efficiente e la disponibilità di quest'ultimo in un calcolatore è un aspetto molto importante dell'Informatica applicata e del calcolo scientifico in quanto determina l'efficienza complessiva dei programmi applicativi.

BLAS, Basic Linear Algebra Subprograms, è un insieme di programmi per tali nuclei computazionali di base (consultare il sito http://www.netlib.org per approfondimenti).
Blas contiene, fra gli altri, programmi per la risoluzione dei problemi elementari prima descritti, per vettori e matrici piene non strutturate, memorizzate in formato standard, e per matrici a banda.

ESERCIZI:

1)
Sviluppare una procedura per il calcolo della trasposta di una matrice piena non strutturata.

2)
Sviluppare una procedura per il calcolo della trasposta di una matrice sparsa non strutturata.

3)
Sviluppare una procedura che individui il massimo elemento di una matrice (piena non strutturata) e scambi la 1 riga (o colonna) con con la riga (o colonna) di tale elemento.

4)
Sviluppare una procedura che individui il massimo elemento di una matrice (piena non strutturata) e scambi la 1 riga e la 1 colonna rispettivamente con la riga e la colonna di tale elemento.

5)
Sviluppare una procedura per il prodotto esterno (tensoriale) di due vettori a e b, ovvero la matrice C i cui elementi sono c(i,j)=a(i)*b(j).

6)
Sviluppare una versione della procedura per il calcolo del prodotto di una matrice tridiagonale per un vettore.

7)
Sviluppare una versione della procedura per il calcolo del prodotto matrice per vettore, dove la matrice è data da un prodotto esterno (vedi Esercizio 5).

8)
Sviluppare una versione della procedura per il calcolo del prodotto matrice per vettore, che usi la procedura `vet_xpay` e una matrice memorizzata in un array 2D con 2 array 1D di indici puntatori.

9)
Sviluppare una procedura per il calcolo del prodotto matrice per vettore. La matrice è sparsa non strutturata memorizzata nel formato Itpack.

10)
Sviluppare una procedura per il calcolo del prodotto matrice per vettore. La matrice è sparsa non strutturata memorizzata nel formato colonna compatta.

11)
Sviluppare una procedura per il calcolo del prodotto matrice per matrice che usi la procedura `mat_vett`.

12)
Sviluppare una procedura per il calcolo del prodotto di due matrici tridiagonali.

13)
Sviluppare una procedura per il calcolo del prodotto di due matrici triangolari (con entrambe le memorizzazioni).

14)
Sviluppare una procedura per il calcolo del prodotto B=TA di una matrice T triangolare superiore i cui elementi non nulli sono tutti uguali a 1 per una matrice qualunque A. Verificare che la prima colonna della matrice prodotto B è uguale alla prima colonna di A, la seconda colonna di B è uguale alla somma della prima colonna e della seconda di A, la i-sima colonna di B è uguale alla somma delle prime i colonne di A.

15)
Sia A una matrice *mxn* dove a(i,j) indica il voto riportato dallo studente i nell'esame j; sia inoltre w un vettore di n componenti, dove w(j) indica il peso dell'esame j (numero crediti). Determinare le medie pesate degli esami di tutti gli studenti.

16)
Riscrivere l'algoritmo di ordinamento per selezione di un array 1D `A` in modo tale che l'array `A` non sia permutato fisicamente ma l'ordinamento sia memorizzato in un array di indici puntatori. L'intestazione sia `procedure ord_sel_punt(in:A,n;inout: punt_A)`

17)
Verificare che il vettore delle differenze in avanti di un vettore `a`, calcolate dalla procedura `differenze` del par. 3.2.3, è ottenibile dal risultato del prodotto della matrice D per il vettore `a`, dove

$$D=\begin{pmatrix} -1 & 1 & 0 & \cdots & 0 \\ 0 & -1 & 1 & 0 & 0 \\ 0 & \cdots & -1 & 1 & 0 \\ \vdots & & \cdots & \ddots & \vdots \\ 0 & \cdots & & 0 & -1 \end{pmatrix}$$

Capitolo quarto

MATLAB

Un linguaggio di programmazione è uno strumento per pensare.

Per Brinch Hansen, 2000

La missione del Calcolo Scientifico è quella di fornire alla comunità scientifica degli strumenti software efficaci ed efficienti.

Gene H. Golub, 1990

All'inizio degli anni 80 il matematico computazionale Cleve Moler sviluppò un sistema interattivo per effettuare computazioni con matrici, da utilizzarsi durante le esercitazioni di laboratorio degli studenti del suo corso di Calcolo numerico. Il sistema, che prese il nome di MATLAB (MATrix LABoratory), era costituito da un semplice linguaggio che consentiva di richiamare in modo interattivo le procedure di un package di algebra lineare numerica (in particolare LINPACK e EISPACK) e da un sistema di visualizzazione grafica.

Negli anni 90 Matlab ha avuto una evoluzione e una diffusione che ne fanno attualmente il sistema numerico più usato, sia in ambito didattico sia in ambito applicativo, in svariati campi del calcolo scientifico.

In questo capitolo Matlab viene presentato soprattutto come un linguaggio di programmazione ad alto livello, sottolineando la struttura di base del linguaggio, l'organizzazione dei programmi e i criteri (o stili) di programmazione. Vengono anche riportate alcune caratteristichepeculiari del Matlab che riguardano le operazioni su matrici e vettori, anche di tipo sparso. Altre caratteristiche di Matlab legate alla risoluzione di problemi matematici specifici (sistemi di equazioni, fitting di dati, sistemi di equazioni differenziali, calcolo simbolico, analisi di segnali, statistica, ecc.) non sono qui illustrate; per una loro trattazione si rimanda a testi di Calcolo Numerico e alla letteratura specializzata in Matlab (*).

(*) Informazioni su Matlab possono essere reperite agli indirizzi:
`http://www.mathworks.com`
`http://www.mathtools.com`

Matlab è un esempio di linguaggio (a paradigma) funzionale, cioè basato sull'utilizzo estensivo di procedure di tipo function.

Il traduttore del linguaggio Matlab è un interprete.

4.1 Organizzazione di una computazione

Una computazione Matlab richiede l'attivazione del programma interprete e procede in modo interattivo, secondo due modalità:

- inserendo da terminale una istruzione alla volta: l'istruzione viene eseguita e il risultato eventualmente visualizzato sullo schermo;
- richiedendo l'esecuzione di un programma Matlab.

Un esempio della prima modalità è il seguente

```
» 7+2
ans =
    9
»
```

dove

- » è il simbolo di *prompt* di Matlab e denota che l'interprete è pronto a ricevere istruzioni;
- `7+2` è l'istruzione inserita, cioè la valutazione di una espressione aritmetica;
- i dati visualizzati dopo `ans =` (`ans` sta per *answer*, cioè risposta) sono il risultato dell'esecuzione dell'istruzione.

Invece un esempio della seconda è:

```
» mioprog
ans =
    xxxxxxxxx
»
```

dove `mioprog` è il nome di un programma Matlab e `xxxxxxxxx` sta per il risultato dell'esecuzione di `mioprog`.

Un programma Matlab è un insieme di istruzioni memorizzate in un file di tipo testo il cui nome deve avere l'estensione `.m` e, per questo, è detto m-file; nell'esempio precedente il nome del file è `mioprog.m`.

Matlab dispone di un ambiente di sviluppo che comprende anche un *Editor/Debugger* per la scrittura e l'aggiornamento del testo di un programma.

4.2 Tipi primitivi, variabili e assegnazione

In Matlab vi sono:

- tre tipi primitivi scalari: il tipo reale, il tipo complesso e il tipo carattere;
- cinque[1] tipi primitivi strutturati: la matrice numerica piena, la matrice numerica sparsa, la stringa di caratteri, il record (detto *structure*), l'array di celle (detto *cell array*).

Non esistono dichiarazioni esplicite di tipo: il tipo di una variabile viene determinato automaticamente al momento della definizione della variabile, cioè attraverso una assegnazione o una lettura da supporto esterno.

Il tipo predefinito di una variabile è la matrice (piena), i cui elementi sono tutti dello stesso tipo. Una matrice (m×n) è realizzata come un array bidimensionale di dati di tipo numerico (reale, complesso). Gli scalari sono considerati matrici (1×1), i vettori colonna sono considerati matrici (m×1) ed i vettori riga matrici (1×n).

L'istruzione di assegnazione è denotata con il simbolo `=`. Le seguenti sono istruzioni di assegnazione in Matlab:

```
» a=3
a =
   3
```

assegna il valore di tipo reale 3 alla variabile `a`;

```
» alfa=[3 5.22 8 -1.1]
alfa =
   3.0000    5.2200    8.0000   -1.1000
```

definisce un vettore riga `alfa` di 4 componenti;

```
» b=[2; 5; -1]
b =
   2
   5
  -1
```

definisce un vettore colonna `b` di 3 componenti;

[1] Altri tipi, come ad esempio `uint8`, non vengono qui trattati.

```
» A=[1 2 0;0 5 6;7 0 9;0 4 7]
A =
   1  2  0
   0  5  6
   7  0  9
   0  4  7
```

definisce una matrice A di 4 righe e 3 colonne;

```
» beta=alfa
beta =
   3.0000    5.2200    8.0000   -1.1000
```

definisce un vettore riga beta con le stesse componenti di alfa;

La function input è utile per guidare l'immissione di dati da tastiera; infatti

```
» raggio = input('inserire il valore del raggio ')
```

visualizza sullo schermo

```
inserire il valore del raggio _
```

e il sistema resta in attesa dell'immissione di un numero da tastiera (conclusa dall'invio), che diviene il valore associato alla variabile raggio; per esempio inserendo il valore 2.5 si ha

```
inserire il valore del raggio 2.5
raggio =
    2.5000
```

Sono equivalenti le istruzioni

```
» a=3;
» a=3.0;
» a=0.3e1;
```

che assegnano il valore di tipo reale 3 alla variabile a.

La presenza del simbolo **;** alla fine di una istruzione indica che il risultato non deve essere visualizzato. La visualizzazione del valore associato a una variabile si ottiene semplicemente specificando il nome della variabile:

```
» a
a =
   3
» alfa
alfa =
   3.0000    5.2200    8.0000   -1.1000
```

oppure utilizzando la function primitiva `disp`

```
» disp(alfa)
   3.0000    5.2200    8.0000   -1.1000
```

I dati di tipo reale vengono rappresentati in memoria sempre in doppia precisione, anche se normalmente vengono visualizzate solo le prime 5 cifre significative; in particolare, se il valore della variabile (di tipo reale) è un vettore o una matrice di numeri interi, questi vengono visualizzati come interi.

L'istruzione `format long` richiede all'interprete la visualizzazione di tutte le cifre significative:

```
» r=1/3
r =
   0.3333
» format long
» r
r =
   0.33333333333333
» format
» r
r =
   0.3333
```

Le istruzioni

```
» d=2+3i
d =
   2.0000 + 3.0000i
» C=[i 2 3+0.1i;0 5i 6]
C =
   0 + 1.0000i      2.0000            3.0000 + 0.1000i
   0                0 + 5.0000i       6.0000
```

definiscono variabili di tipo complesso; in particolare, la prima assegnazione definisce una variabile scalare `d` e la seconda una variabile `C` di tipo matrice (2×3).

L'accesso alle singole componenti di una variabile array si denota facendo seguire il nome della variabile da parentesi tonde contenenti un numero di indici adeguato alle dimensioni dell'array:

```
» (b(1)+alfa(k))/A(1,1)^2-3*A(1,k+1)
ans =
   -5.1000
```

Si noti, nell'esempio precedente, che l'interprete valuta una espressione anche se questa non fa parte di una operazione di assegnazione; il valore dell'espressione viene automaticamente associato alla variabile speciale ans.

Il tipo matrice è rappresentato in memoria con un array 2D. L'allocazione di spazio è dinamica ed è effettuata implicitamente; ad esempio l'istruzione

```
» alfa=[alfa 22 44]
alfa =
   3.0000   5.2200   8.0000  -1.1000  22.0000  44.0000
```

alloca due nuove componenti per il vettore `alfa` e assegna loro i valori 22 e 44.

```
» b=[b; -3.1]
b =
   2.0000
   5.0000
  -1.0000
  -3.1000
```

alloca una nuova componente per il vettore b e le assegna il valore `-3.1`

```
» A=[A b]
A =
   1.0000   2.0000        0   2.0000
        0   5.0000   6.0000   5.0000
   7.0000        0   9.0000  -1.0000
        0   4.0000   7.0000  -3.1000
```

alloca una nuova colonna per la matrice `A` e le assegna il valore del vettore (colonna) `b`.

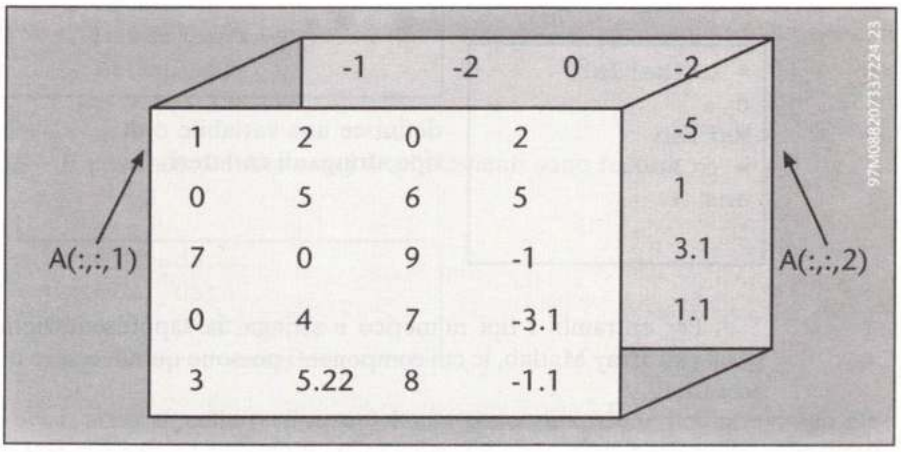
```
» A=[A; beta]
A =
   1.0000    2.0000         0    2.0000
        0    5.0000    6.0000    5.0000
   7.0000         0    9.0000   -1.0000
        0    4.0000    7.0000   -3.1000
   3.0000    5.2200    8.0000   -1.1000
```

alloca una nuova riga per la matrice `A` e le assegna il valore del vettore (riga) `beta`.

Tale approccio consente di costruire matrici con più di due dimensioni:

```
» A(:,:,2) = -A;
```

definisce una matrice `A` tridimensionale di 5 righe, 4 colonne e 2 facce (componenti nella terza dimensione):



Si noti, negli esempi precedenti, la diversa notazione per indicare l'aumento del numero delle colonne (concatenazione per colonne), l'aumento del numero delle righe (concatenazione per righe) e l'aumento del numero di dimesioni di una matrice.

Il tipo matrice sparsa viene rappresentato in memoria mediante il formato a colonna sparsa compressa (vedi par. 3.5.3), cioè mediante tre array 1D.

```
» A=[1 2 0;0 5 6;7 0 9;0 4 7]
» B=sparse(A)
B =
(1,1)        1
(3,1)        7
(1,2)        2
(2,2)        5
(4,2)        4
(2,3)    6
(3,3)    9
(4,3)    7
```

definisce una variabile B di tipo matrice sparsa, con gli stessi elementi della matrice (piena) A. Gli elementi nulli di B non sono memorizzati e quindi l'uso di variabili di tipo matrice sparsa può portare a computazioni più efficienti sia in termini di spazio di memoria sia, come vedremo, in termini di numero di operazioni e di tempo di esecuzione. La function primitiva sparse è il costruttore di matrici sparse.

Il tipo stringa di caratteri viene rappresentato come un array 1D in cui ogni elemento rappresenta un carattere della stringa.

```
» c='Matlab'
c =
Matlab
» c(3)
ans =
t
```

definisce una variabile c di tipo stringa di caratteri;

Per entrambi i tipi numerico e stringa, la rappresentazione interna è basata su array Matlab, le cui componenti possono quindi essere direttamente accessibili.

Matlab consente di gestire anche il tipo logico, rappresentando il valore "vero" come 1 ed il valore "falso" come 0. Sono inoltre definiti gli usuali operatori logici:

```
» ~1
ans =
    0
```

~ **"not"**

```
» 0|1
ans =
    1
```

| **"or"**

```
» 0&1
ans =
    0
```

& **"and"**

Espressioni logiche si ottengono anche tramite gli operatori relazionali definiti su tipi di dato ordinati; ad esempio:

```
» n=2; n>1 & n<5
ans =
   1
» n==2
ans =
   1
```

Una variabile di tipo record, detta in Matlab `structure`, viene definita tramite i suoi *campi*, come nell'esempio:

```
» data.giorno=29;
» data.mese='aprile';
» data.anno=1999;
» data
data =
   giorno: 29
     mese:'aprile'
     anno:1999
```

È possibile definire array i cui elementi sono record:

```
» date(1)=data;
» date(2)=data;
» date(2).giorno=30;
```

L'array di celle (*cell array*) è una generalizzazione dell'array; agli elementi di un tale array, detti *celle*, è possibile associare dati, anche strutturati, di tipo diverso. Ad esempio, la variabile `A` può essere definita come array di celle nel modo seguente:

```
» A(1,1)={1};
» A(1,2)={[2 -4 5;6 1 3]};
» A(2,1)={data};
» A(2,2)={'nel mezzo del cammin'};
» A
 [           1]      [2x3 double]
 [  1x1 struct]      [1x20 char ]
```

oppure equivalentemente:

```
» A{1,1}=1;
» A{1,2}=[2 -4 5;6 1 3];
» A{2,1}=data;
» A{2,2}='nel mezzo del cammin';
```

La variabile `A` risulta un array di 4 celle (2×2), le cui componenti sono di tipo diverso (sia scalare sia strutturato); una cella particolare, per esempio la cella 1,2, si denota con A(1,2) oppure con A{1,2} ed è direttamente accessibile.

1	2-4 5 6 1 3
29 'aprile' 1999	nel mezzo del cammin

Fig. 4.1: l'array di celle A

In pratica, l'array di celle può essere visto come un contenitore ai cui elementi è possibile associare un qualsiasi tipo di dato. Esso consente di organizzare, con maggiore flessibilità rispetto al tipo `structure` e all'array di `structure`, dati di tipo o dimensioni differenti.

Record e array di celle possono essere definiti anche attraverso i costruttori `struct` e `cell`.

La deallocazione completa di spazio si effettua mediante l'istruzione `clear`

`» clear`	dealloca tutte le variabili
`» clear a A`	dealloca solo le variabili `a` ed `A`

Anche la deallocazione parziale di spazio, per matrici e array di celle, si effettua utilizzando il simbolo `[]`:

```
» b(4)=[]
b =
    2
    5
   -1
» beta(2)=[]
beta =
    3.0000    8.0000   -1.1000
```

elimina la quarta componente di `b` e la seconda componente di `beta`.

Per eliminare un campo di una variabile record, si utilizza la function primitiva `rmfield`

```
» rmfield(data,'mese')
ans =
  giorno: 29
    anno: 1999
```

In ogni caso, una nuova assegnazione definisce il nuovo valore e anche il nuovo tipo di una variabile; è quindi formalmente corretta la sequenza di istruzioni

```
» gamma=[3 4 -8.1 9];
» gamma=3;
» gamma='Matlab';
```

L'insieme costituito da tutte le variabili che vengono definite in una computazione interattiva prende il nome di *area di lavoro* (workspace).

I comandi `who` e `whos` consentono di avere informazioni sulle variabili in quel momento presenti nell'area di lavoro.

Per esempio, per l'elenco delle variabili allocate:

```
» who
Your variables are:

A         C         aa        b         c         r
B         a         ans       beta      d
```

e per avere tutte le informazioni sulle variabili allocate:

```
» whos
Name       Size     Elements     Bytes    Density   Complex
A        5 by 4        20         160       Full      No
B        4 by 3         8         108     0.6667      No
C        2 by 3         6          96       Full      Yes
a        1 by 1         1           8       Full      No
aa       1 by 1         1           8       Full      No
ans      1 by 2         2          16       Full      No
b        3 by 1         3          24       Full      No
beta     1 by 3         3          24       Full      No
c        1 by 6         6          48       Full      No
d        1 by 1         1          16       Full      Yes
r        1 by 1         1           8       Full      No
Grand total is 52 elements using 516 bytes
```

4.3 Operazioni strutturali su matrici

Negli esempi del paragrafo precedente si è mostrato che il costruttore di matrici viene denotato con `[...]`, mentre l'accesso selettivo a una componente (o, come vedremo, a un sottoinsieme di componenti) di una matrice si indica con `(...)`.

Un altro costruttore, specifico per vettori, è basato sull'uso del simbolo **:** (due punti).

La convenzione generale è `<inizio>:<passo>:<fine>`, dove si può omettere `:<passo>` se questo vale `1`.

```
» g=1:2:9
g =
   1     3     5     7     9
» n=5;
» p=n:-1:-3
p =
   5     4     3     2     1     0    -1    -2    -3
» q=1:n
q =
   1     2     3     4     5
» x=-pi:(2*pi)/100:pi;
```

Si noti l'uso della variabile predefinita `pi` a cui è associato il valore π.[2]

La convenzione per denotare una porzione di una matrice è basata ancora sull'uso del simbolo "**:**" : una porzione di matrice viene in generale indicata inserendo nel selettore `(...)` un vettore o una matrice che contiene gli indici degli elementi desiderati. Ad esempio, per denotare solo alcune componenti di un vettore:

```
» beta
beta =
   3.0000    5.2200      8.0000   -1.1000
» beta(1:3)
ans =
   3.0000    5.2200    8.0000
» beta([1 3])
ans =
   3.0000    8.0000
```

oppure per denotare una sottomatrice di matrice:

```
» A
A =
  1.0000  2.0000       0   2.0000
       0  5.0000  6.0000   5.0000
  7.0000       0  9.0000  -1.0000
       0  4.0000  7.0000  -3.1000
  3.0000  5.2200  8.0000  -1.1000
```

⇨

```
» A(2:3,2:4)
ans =
  5    6    5
  0    9   -1

» D=A(1:3,1:3)
D =
  0    5    6
  7    0    9
```

```
» j=[1 3];
» k=[1 3 4];
» A(j,k)
ans =
   1    0    2
   7    9   -1
```

seleziona la sottomatrice formata dalla prima e terza riga e dalle colonne 1, 3, 4 di `A`;

[2] Altre variabili predefinite sono: `eps` per l'epsilon macchina; `i,j` per l'unità immaginaria; `realmax` e `realmin` per il massimo e il minimo numero floating point rappresentabile; `inf` per infinito; `NaN` per il risultato di una operazione matematicamente indefinita, come ad esempio 0/0, 0*`inf`

In particolare

```
» A(2,:)
ans =
     0   5   6   5
```

denota la seconda riga di A;

```
» A(:,1)
ans =
   1
   0
   7
   0
   3
```

denota la prima colonna di A;

```
» A(5:-1:1,:)
ans =
   3.0000    5.2200    8.0000   -1.1000
        0    4.0000    7.0000   -3.1000
   7.0000         0    9.0000   -1.0000
        0    5.0000    6.0000    5.0000
   1.0000    2.0000         0    2.0000
```

denota la matrice con le righe invertite rispetto ad A.

L'uso di vettori di indici puntatori è immediato in Matlab:

```
» vet=[10 70 20 50 30 80 90 40 60];
» punt=[1 3 5 8 4 9 2 6 7];
» disp(vet(punt))
   10 20 30 40 50 60 70 80 90
```

e consente di denotare in modo naturale le permutazioni di righe e di colonne delle matrici:

```
» mat=[0 0 10 20; 0 0 30 40; 50 60 70 80]
mat =
   0    0   10   20
   0    0   30   40
  50   60   70   80

» punt_riga=[3 2 1];
» punt_colonna=[4 3 1 2];
» disp(mat(:,punt_colonna))
   20    10    0    0
   40    30    0    0
   80    70   50   60

» disp(mat(punt_riga,punt_colonna))
   80    70   50   60
   40    30    0    0
   20    10    0    0
```

La notazione di sottomatrice di matrice e la possibilità di allocare dinamicamente nuove righe e colonne consentono di costruire facilmente matrici a blocchi e di estrarre blocchi da matrici:

```
» A11=[1 1 1; 1 1 1];
» A12=[2 2; 2 2];
» A21=[3 3 3];
» A_blocchi=[A11 A12;A21 [4 4]]
ans =
   1  1  1  2  2
   1  1  1  2  2
   3  3  3  4  4

» disp(A_blocchi(2:3,3:5))
   1  2  2
   3  4  4

» A22=A_blocchi(3,4:5)
A22 =
   4  4
```

4.4 Operazioni aritmetiche matriciali

È conveniente suddividere le operazioni aritmetiche Matlab, coinvolgenti le matrici, in due classi: le operazioni matriciali dell'algebra delle matrici e dei vettori (prodotto scalare, prodotto matrice vettore, prodotto matrice matrice,..) e le operazioni che sono ripetute componente per componente sugli elementi di una matrice.

Le operazioni matriciali sono denotate con i simboli aritmetici usuali (+, -, *). Ad esempio:

```
» A=[1 2 0; 0 5 6; 7 0 9; 0 4 7];
» C=[1 0 3; 4 2 1; 7 3 9; 0 4 1];
» A+C
ans =
   2     2     3
   4     7     7
  14     3    18
   0     8     8
```

A+C effettua la somma di due matrici

```
» A+10
ans =
   11    12    10
   10    15    16
   17    10    19
   10    14    17
```

effettua la somma dello scalare 10 a tutti gli elementi della matrice A

```
» A+3*C
ans =
   4     2     9
  12    11     9
  28     9    36
   0    16    10
```

effettua il prodotto della matrice C per lo scalare 3 e la somma di tale matrice con A

```
» A*b
ans =
   12
   19
    5
   13
```

effettua il prodotto matrice per vettore (si noti che b deve essere un vettore colonna)

```
» A*D
ans =
   1    12    12
  42    25    84
  70    14    81
  49    20    87
```

effettua il prodotto righe per colonne di due matrici.

L'istruzione seguente richiede l'esecuzione del prodotto tra matrici di `size` non appropriato

```
» A*C
??? Error using ==> *
Inner matrix dimensions must agree.
```

L'operazione di trasposizione di una matrice si indica con il simbolo " ' "

```
» C=C'
C =
   1    4    7    0
   0    2    3    4
   3    1    9    1

» b'
ans =
   2  5  -1
```

La trasposizione trasforma un vettore riga in un vettore colonna e viceversa[3].

Come caso particolare di prodotto di due matrici, il prodotto scalare tra due vettori (colonna) `h, g` viene denotato dall'istruzione:

```
» h=[2 5 -1]'; g=[3 -4 1]';
» h'*g
ans =
  -15
```

[3] Nel caso di matrice complessa la trasposizione effettua anche l'operazione di coniugazione (cambio del segno della parte immaginaria) degli elementi.

mentre l'istruzione:

```
» g*h'
ans =
   6    15    -3
  -8   -20     4
   2     5    -1
```

denota il prodotto di un vettore colonna per un vettore riga e, pertanto, produce una matrice; il prodotto `h*g` non è matematicamente definito; infatti:

```
» h*g
??? Error using ==> *
Inner matrix dimensions must agree.
```

Infine, l'istruzione:

```
» D^2
ans =
   1    12    12
  42    25    84
  70    14    81
```

denota il quadrato di una matrice; tale operazione è equivalente al prodotto matriciale di una matrice per se stessa; ciò può essere verificato nel seguente modo:

```
» D*D-D^2
ans =
   0     0     0
   0     0     0
   0     0     0
```

Senza entrare nel dettaglio, si riportano di seguito solo alcuni esempi per mostrare come lo stesso operatore di divisione (quello a sinistra ****) denoti, in funzione dei particolari dati cui esso è applicato, diverse operazioni di Algebra Lineare Numerica.

Sia dato il sistema $\boldsymbol{Ax=b}$, dove $\boldsymbol{A}(m \times n)$, $x(n \times 1)$, $\boldsymbol{b}(m \times 1)$ sono array reali, allora l'istruzione

```
» x=A\b
```

restituisce:

1. la soluzione del sistema (mediante l'*algoritmo di Gauss con pivoting parziale*), se questo è compatibile determinato.

```
» A=[1 0 2;-2 1 1;3 4 0];
  b=[7 3 11]';
» x=A\b
x =
    1.0000
    2.0000
    3.0000
```

2. la *soluzione ai minimi quadrati* se il sistema è incompatibile

```
» A=[1 2;1 2;3 -4];
  b=[7 3 11]';
» x=A\b
x =
    4.2000
    0.4000
```

3. una *soluzione particolare* (con al più r componenti non nulle dove r è il *rango* di $\boldsymbol{A}$), se il sistema è indeterminato[4].

```
» A=[1 0 2;-2 1 1]; b=[7 3]';
» xp=A\b; xp'
ans =
    0.2000         0    3.4000
» xo=null(A,'r'); xo'
ans =
 -2    -5     1
```

I seguenti esempi mostrano alcune operazioni su matrici a blocchi:

[4] La soluzione generale $\boldsymbol{x}$ di un sistema indeterminato si scrive:

$$\boldsymbol{x} = \boldsymbol{x}^{(p)} + \lambda \boldsymbol{x}^{(o)}, \quad \forall \lambda \in \mathbf{R}$$

dove $\lambda \boldsymbol{x}^{(o)}$ è la soluzione generale di $\boldsymbol{Ax=0}$
$\boldsymbol{x}^{(p)}$ è una soluzione particolare di $\boldsymbol{Ax=b}$.

```
» A_blocchi_tr=[A11' A21';A12' A22']

     1  1  3
     1  1  3
     1  1  3
     2  2  4
     2  2  4

» AperAT=[A11*A11'+A12*A12'   A11*A21'+A12*A22';
     A21*A11'+A22*A12'   A21*A21'+A22*A22'];
» A_blocchi_tr'*A_blocchi_tr-AperAT

     0  0  0
     0  0  0
     0  0  0
```

Sia nel calcolo della trasposta di una matrice a blocchi sia nel prodotto tra matrici a blocchi, i singoli blocchi sono coinvolti nello stesso modo in cui si opera sugli elementi scalari di una matrice 'classica' nelle stesse operazioni.

4.5 Operazioni componente per componente

Le operazioni di somma e sottrazione di matrici agiscono, per definizione, componente per componente. Prodotto, divisione ed elevamento a potenza componente per componente si denotano premettendo il simbolo "`.`" all'operatore aritmetico. Per esempio:

calcola il prodotto componente per componente dei due vettori (per inciso si noti che l'operazione `y*z` non è definita);

calcola il quoziente componente per componente

```
» y=[4 6 9];
» z=[1 0 2];
» y.*z
ans =
   4     0    18
» ans./y
ans =
   1     0     2
```

Altri esempi di operazioni, che agiscono componente per componente, sono:

```
» y.^z
ans =
   4     1    81

»  y.^2
ans =
   16    36        81
```

```
» D.^2
ans =
    1     4     0
    0    25    36
   49     0    81

» D.^2 - D.*D
ans =
   0     0     0
   0     0     0
   0     0     0
```

La modalità componente per componente è molto utile quando si usano funzioni campionate su una griglia; come già osservato, questa è la situazione usuale nell'approccio numerico. Le istruzioni

```
» griglia=0:0.1:2;
» funzione_campionata=(5*x.^2 +3*x)./(2*x.^3 +1);
```

generano una griglia 1D uniforme sull'intervallo [0,2] con passo 0.1 e il corrispondente insieme di valori campionati della funzione $\frac{5x^2+3x}{2x^3+1}$.

È importante sottolineare che la notazione di sottomatrice di matrice e le operazioni componente per componente costituiscono gli strumenti di base dello stile di programmazione a parallelismo sui dati, che è lo stile più naturale per la descrizione degli algoritmi in Matlab.

4.6 Confronto e operazioni logiche con matrici

Le operazioni di confronto agiscono su due variabili di tipo matrice (piena o sparsa) e sono denotate in Matlab dai simboli `==` (uguale), `~=` (diverso), `>` (maggiore), `>=` (non minore), `<` (minore), `<=` (non maggiore). La caratteristica fondamentale di tali operazioni è che il risultato è ancora una matrice, dello stesso size degli operandi, i cui elementi sono valori di verità, denotati in Matlab con `1` (vero) oppure con `0` (falso):

```
» A < B
ans =
   1     0     0
   1     0     0
   0     1     0
   1     0     0
```

```
» A=[1 2 0; 0 5 6; 7 0 9; 0 4 7];
» B=[2 1 0;9 -7 -1; 7 1 8; 1 4 6];
» [A == B A ~= B]
ans =
   0   0   1   1   1   0
   0   0   0   1   1   1
   1   0   0   0   1   1
   0   1   0   1   0   1
```

Le operazioni di confronto agiscono componente per componente; il risultato dell'operazione su una singola coppia di elementi è `1` se il confronto è vero, oppure `0` se il confronto è falso.

Le operazioni di confronto possono coinvolgere anche scalari, come per esempio

```
» 3 < 5
ans =
   1
```

Analogamente, le operazioni logiche not, and e or, quando applicate a matrici, agiscono in modalità componente per componente

```
» ~A
ans =
   0     0     1
   1     0     0
   0     1     0
   1     0     0
```

il risultato dell'operazione su un singolo elemento è `1` se l'elemento è nullo, oppure `0` se l'elemento è non nullo;

```
» A&B
ans =
   1     1     0
   0     1     1
   1     0     1
   0     1     1
```

il risultato dell'operazione sulla singola coppia di elementi è 1 se entrambi gli elementi sono non nulli, oppure 0 se entrambi gli elementi sono nulli;

```
» A|B
ans =
   1     1     0
   1     1     1
   1     1     1
   1     1     1
```

il risultato dell'operazione sulla singola coppia di elementi è 1 se almeno uno degli elementi è non nullo, oppure 0 se entrambi gli elementi sono nulli.

4.7 Function e script

I programmi scritti in Matlab, memorizzati in file detti *m-file*, sono di due tipi: *script file* e *function file*, le cui differenze si possono riassumere in Tab. 4.1.

Tipi di *m-files*	
Script file	*Function files*
non ha intestazione	ha intestazione
non ha parametri di input e di output	ha parametri di input e di output
opera sulle variabili nel *workspace*: usa quelle già attive (anche eliminandole) oppure ne crea di nuove	opera su variabili interne (*local workspace*) ed eventualmente su variabili dichiarate globali (*global*)
è interpretato a ogni chiamata	è compilato all'atto della prima chiamata
nome del file: qualsiasi	nome del file: uguale a quello della function[5]

Tab. 4.1: differenze tra script e function file

[5] Esistono anche dei function files locali (detti sub function): si ottengono inserendo le loro linee di programma nello stesso file di un m-file che lo richiama. In tal modo però non sono richiamabili da altri m-file.

L'uso estensivo delle function è la caratteristica fondamentale della programmazione in Matlab. Le principali funzionalità del linguaggio sono infatti realizzate attraverso l'insieme delle function primitive. Ulteriori funzionalità sono aggiungibili, come usuale in programmazione, sviluppando il corrispondente algoritmo e organizzando l'algoritmo come una function.

È consentito l'approccio ricorsivo, denotato come al solito attraverso l'autorefenziamento di una function.

La dichiarazione

```
global <lista_variabili>
```

denota che le variabili in `<lista_variabili>` sono globali tra tutte le function che contengono la stessa dichiarazione; se la dichiarazione compare in uno script oppure viene immessa nella finestra comandi in modalità interattiva, allora si ottengono variabili globali nel *workspace*.

4.7.1 Script

Un programma *script* è semplicemente un insieme di istruzioni Matlab individuato da un nome. Al contrario di una function, ad esso non si applica il concetto di località delle variabili: ciò significa, da un lato, che le variabili definite all'interno di uno script appartengono tutte all'area di lavoro e, dall'altro, che le istruzioni di uno script possono far riferimento a qualsiasi variabile presente nell'area di lavoro, al momento dell'esecuzione.

ESEMPIO:

script per il calcolo del coefficiente binomiale

```
% script file coefbinom.m
%
n=input('valore di n');
k=input('valore di k (k = n)');
binom=prod(1:n)/(prod(1:k)*prod(1:n-k));
```

dove `prod(1:n)` calcola il fattoriale di `n` per ogni $n \geq 0$.

Uno *script* viene eseguito semplicemente inserendo il nome dello *script file* (senza l'estensione *.m*) al prompt di Matlab: per esempio, utilizzando lo script file `coefbinom.m` precedente, basta digitare

```
» coefbinom
```

in una sessione matlab per ottenere l'esecuzione dello script stesso.

Dal punto di vista operativo, la chiamata di uno script equivale a inserire interattivamente, una dopo l'altra, le singole istruzioni che lo costituiscono. In alcuni linguaggi lo script viene detta *macroistruzione*.

4.7.2 Function non primitive

Un *function file* Matlab deve contenere la seguente intestazione:

```
function [<lista_par_output>]=nome_function>
                                    (<lista_par_input>)
```

oppure, nel caso in cui la `<lista_par_output>` sia costituita da un'unica variabile,

```
function <nome_var> = <nome_function>
                              (<lista_par_input>)
```

Solo in quest'ultimo caso la chiamata della function può avvenire all'interno di una espressione.

L'ultima istruzione della function chiude il corpo della function e determina il ritorno all'unità di programma chiamante. A tale scopo si può anche usare l'istruzione `return`. Come usuale, nel corpo della function devono essere definite le variabili che compaiono in `<lista_par_output>`.

ESEMPIO:

function per il fattoriale di un numero *n* (non ricorsivo)

```
function nfat=fat(n)
    k=n;nfat=k;
    while k>2,
       k=k-1;
       nfat=nfat*k;
    end
```

Le variabili definite all'interno di una function sono locali alla function. In questo esempio `k` è locale.

ESEMPIO:

function per il fattoriale di un numero *n* (ricorsivo)

```
function nfat=fat_ric(n)
```

```
    if n<=2
        nfat=n;
    else
        nfat=n*fat_ric(n-1);
    end
```

ESEMPIO:

function ricorsiva per l'algoritmo di Strassen (par 3.5.4)

```
 function C=strassen(A,B)
% La function STRASSEN calcola il prodotto matriciale
% tra le matrici A e B mediante l'algoritmo di
% Strassen
% (complessità di tempo T(n)=0(n^2.8)).
%
    [m,n] = size(A);
    if n == 1
        C = A*B;
    else
        nmezzi = n/2; p = 1:nmezzi; q = (nmezzi+1):n
        P1 = strassen(A(p,p)+A(q,q),B(p,p)+B(q,q));
        P2 = strassen(A(q,p)+A(q,q),B(p,p));
        P3 = strassen(A(p,p),B(p,q)+B(q,q));
        P4 = strassen(A(q,q),B(p,q)+B(q,q));
        P5 = strassen(A(p,p)+A(p,q),B(q,q);
        P6 = strassen(A(q,p)-A(p,p),B(p,p)+B(p,q);
        P7 = strassen(A(p,q)-A(q,q),B(q,p)+B(q,q);
        C = [P1+P4-P5+P7    P3+P5;
              P2+P4     P1-P2+P3+P6];
    end
```

4.7.3 Alcune function primitive

Le function primitive sono classificabili in

function di base	*function avanzate*[6]
matematiche	matematiche
relazionali e logiche	algebra lineare numerica
azioni elementari su matrici	fitting di dati

[6] Altre funzionalità avanzate sono organizzate in package Matlab specifici, detti toolbox.

azioni elementari su matrici sparse
azioni elementari su stringhe
analisi di dati
grafica
risoluzione di equazioni differenziali
integrazione numerica
equazioni non lineari e ottimizzazione
statistica computazionale
calcolo simbolico
analisi di segnali

L'istruzione » `help <nome_function>` visualizza alcune informazioni sulle funzionalità e sulle specifiche della function `<nome_function>`.

Negli esempi che seguono si illustra l'uso di alcune delle function primitive di base.

La caratteristica principale delle function Matlab è che esse in generale hanno matrici come parametri di input[7] e di output.

Le function matematiche di base agiscono componente per componente.

Le function matematiche di base e le operazioni aritmetiche componente per componente consentono di calcolare i valori delle principali funzioni matematiche su una qualunque griglia.

```
» x=[0.2 1.1 1.5];
» y= cos(x)
y =
   0.9801    0.4536    0.0707

» z= 3*sqrt(2*x)
z =
   1.8974    4.4497    5.1962
» s= sin(x)./x
s =
   0.9933    0.8102    0.6650
» u= log(x).*exp(abs(tan(log10(x))))
u =
   -3.7300    0.0993    0.4844
```

Le function logiche (`any` e `all`) agiscono su una matrice e restituiscono un solo valore logico.

[7] Il passaggio dei parametri di input di una function avviene secondo il meccanismo di sostituzione di valore.

In particolare

```
» any(y)
ans =
   1
» any(A)
ans =
   1  1  1
```

restituisce il valore 1 (vero) se almeno un elem del vettore y è non nullo, e restituisce il valore (falso) se tutti gli elementi sono nulli; se l'argom è una matrice, la function opera per colonne e re tuisce un vettore riga di 1 e 0

```
» all(y)
ans =
   1
» all(A)
ans =
   0  0  0
```

restituisce il valore 1 (vero) se tutti gli elemen vettore y sono non nulli, e restituisce il valore (falso) se almeno uno è nullo; se l'argomento è matrice, la function opera per colonne e restit un vettore riga di 1 e 0

Naturalmente, si ha che

```
» any(y) == ~all(y)
ans =
   1

» any(all(A))
ans =
   0
```

```
» vet=[3.1 0 -2.7 2.5 0];
» find(vet)
ans =
   1  3  4
```

La function find restit un vettore riga degli in degli elementi non nulli di vettore;

oppure

```
» [i_riga, i_colonna]=find(A);
» disp([i_riga';i_colonna'])
   1  3  1  2  4  2  3  4
   1  1  2  2  2  3  3  3
```

due vettori contenenti risp tivamente gli indici di riga di colonna degli elementi nulli di una matrice (ag anche su variabili di tipo ma trice sparsa)

Sfruttando la gestione dei valori logici in Matlab, la function `find` consente di individuare le componenti di un array soddisfacenti la proprietà espressa nel parametro della function: per esempio

```
» find(vet>2)
ans =
        1     4
» vet(find(vet>2))
ans =
3.1000  2.5000
```

restituisce gli indici degli elementi di `vet` che hanno un valore maggiore di `2` e visualizza tale sottovettore.

Varie azioni strutturali su matrici sono implementate attraverso function primitive:

```
» diag(A)'
ans =
   1   5   9
```

estrae la diagonale principale della matrice `A`;

```
» diag(A,1)
ans =
   2
   6
```

estrae la diagonale sopra la diagonale principale della matrice `A`

```
» diag(A,-1)
ans =
   0
   0
   7
```

estrae la diagonale sotto la diagonale principale della matrice `A`.

Al contrario `diag` agisce come un costruttore di matrice quando l'argomento è un vettore

```
» b=[2 5 -1]';
» diag(b)
ans =
   2     0     0
   0     5     0
   0     0    -1
```

```
» b=[2 5 -1]';
» diag(b,1)
ans =
   0     2     0     0
   0     0     5     0
   0     0     0    -1
   0     0     0     0
```

Si è già osservato che, in Matlab, l'operazione denotata da alcune function dipende dagli operandi: per la function `diag` il risultato dipende dal `size` degli operandi. Ad esempio si ha che

```
» diag(diag(A))
ans =
   1     0     0
   0     5     0
   0     0     9
```

definisce una matrice diagonale con la stessa diagonale di `A`

```
» diag(b,1)+diag(b,-1)
ans =
   0     2     0     0
   2     0     5     0
   0     5     0    -1
   0     0    -1     0
```

definisce una matrice con le due subdiagonali principali uguali al vettore `b`

L'istruzione

```
» tril(A)
ans =
   1     0     0
   0     5     0
   7     0     9
   0     4     7
```

estrae il triangolo inferiore (alla diagonale principale) della matrice `A`; analogamente, `triu(A)` estrae il triangolo superiore.

Varie function agiscono come costruttori di particolari vettori e matrici; ad esempio

```
» x=linspace(1,2,5)
x =
   1.0000    1.2500    1.5000    1.7500    2.0000
```

definisce un vettore(riga) di 5 componenti che costituiscono una griglia di passo costante sull'intervallo [1,2];

```
» x=logspace(1,2,5)
x =
   10.0000   17.7828   31.6228   56.2341  100.0000
```

definisce un vettore (riga) di 5 ccomponenti che costituiscono una griglia il cui logaritmo del passo è costante sull'intervallo $[10^1,10^2]$.

Cioè l'istruzione `logspace(a,b,c)` è equivalente a `x=10.^(a+[0:c-1]*(b-a)/(c-1))`.

Altri esempi di function predefinite sono:

```
» zeros(m,n);
» ones(m,n);
» eye(m,n);
```

definiscono matrici, di size `m,n`, rispettivamente ad elementi nulli, ad elementi unitari ed una matrice avente tutti uno sulla diagonale principale (è la matrice identica se `m=n`).

```
» max(b)
ans =
   5
```

calcola il massimo elemento del vettore `b`;

```
» max(A)
ans =
   7     5     9
```

definisce il vettore (riga) degli elementi massimi di ciascuna colonna della matrice `A`;

```
» max(max(A))
ans =
   9
```

calcola il massimo elemento della matrice `A`.

In modo del tutto analogo agisce la function `min`, per il calcolo del minimo.

```
» sum(b)
ans =
   6
```

calcola la somma delle componenti del vettore `b`;

```
» sum(A)
ans =
   8    11    22
```

definisce il vettore (riga) delle somme degli elementi di ciascuna colonna della matrice `A`;

```
» sum(sum(A))
ans =
   41
```

calcola la somma di tutti gli elementi della matrice `A`.

In modo del tutto analogo agisce la function `prod`, per il calcolo del prodotto.

Invece, le function `cumsum` e `cumprod` producono array dello stesso `size` dell'argomento contenenti rispettivamente le somme ed i prodotti parziali delle componenti; per esempio

```
» b=[2 5 -1];
» cumsum(b)
ans =
   2    7    6
```

definisce il vettore (riga come `b`) le cui componenti contengono nell'ordine la prima componente di `b`, la somma delle prime due componenti di `b` e la somma delle tre componenti di `b`

```
» A=[1 2 0
     0 5 6
     7 0 9
     0 4 7];
» cumsum(A)
ans =
   1    2    0
   1    7    6
   8    7   15
   8   11   22
```

applica `cumsum` ad ogni vettore colonna di `A`

Si possono usare le function `prod` e `cumprod` per calcolare rispettivamente il fattoriale di `n` (con `n` opportuno affinchè non provochi overflow!) ed il fattoriale di tutti gli interi tra 1 ed `n`:

```
» n=5; prod(2:n)
ans =
   120
» cumprod(1:n)
ans =
   1   2   6   24   120
```

La function

```
» sort(b)
ans =
  -1   2   5
```

definisce il vettore ottenuto da `b` ordinando in modo crescente le sue componenti; se l'argomento è una matrice, l'ordinamento è applicato alle singole colonne.

La function

```
» diff(b)
ans =
   3    -6
```

definisce il vettore delle differenze in avanti di b, cioè un vettore le cui componenti contengono la differenza tra due elementi consecutivi del vettore b.

Il medesimo risultato è ottenibile mediante

```
» b(2:length(b))-b(1:length(b)-1);
```

dove `length` è la funzione che restituisce il numero di componenti del vettore argomento.

Una stessa function Matlab può avere chiamate con un numero diverso di parametri; ad esempio, si è visto che la function `max` se chiamata con un solo parametro di uscita restituisce il massimo di un vettore, invece chiamata con due parametri di uscita

```
» [max_b,i_max]=max(b)
max_b =
   5
i_max =
   2
```

determina valore ed indice dell'elemento massimo del vettore b;

```
» [num_row,num_col]=size(A)
num_row =
   4
num_col =
   3
```

determina il numero di righe e il numero di colonne della matrice A;

La function `size` è utile anche quando si voglia definire una matrice speciale le cui dimensioni sono uguali a quelle di una matrice data; ad esempio le istruzioni

```
» I=eye(size(A))
» O=zeros(size(A))
» Z=ones(size(A))
```

definiscono rispettivamente la matrice identica, la matrice nulla e la matrice con elementi unitari, tutte con le stesse dimensioni di A.

La function `nnz` determina il numero di elementi non nulli di una matrice (anche di tipo sparso)

```
» V = 5*S
V =
   (1,1)        5
   (3,1)       35
   (1,2)       10
   (2,2)       25
   (4,2)       20
   (2,3)       30
   (3,3)       45
   (4,3)       35
```

definisce la variabile V di tipo matrice sparsa, poichè S è una matrice sparsa;

ancora, l'assegnazione

```
» Z = V+S
Z =
   (1,1)        6
   (3,1)       42
   (1,2)       12
   (2,2)       30
   (4,2)       24
   (2,3)       36
   (3,3)       54
   (4,3)       42
```

definisce la variabile Z di tipo matrice sparsa, poichè S e V sono matrici sparse;

```
» V = V';
» W = V*S
W =
   (1,1)      250
   (2,1)       10
   (3,1)      315
   (1,2)       10
   (2,2)      225
   (3,2)      290
   (1,3)      315
   (2,3)      290
   (3,3)      830
```

sul tipo sparso sono primitive le operazioni di trasposizione di matrice e di prodotto tra matrici sparse[8]

[8] Per le convenzioni riguardanti espressioni miste, che coinvolgono sia variabili di tipo matrice sparsa sia variabili di tipo matrice piena, si rimanda alla documentazione Matlab.

Il seguente esempio consente di verificare che l'uso di variabili di tipo matrice sparsa dà luogo a computazioni più efficienti:

```
» A_piena = [.......];
» B_piena = [.......];
» flops(0)
» tic; A_piena*B_piena;tempo_prodotto_pieno=toc;
» costo_prodotto_pieno=flops;
» A_sparsa = sparse(A_piena);
» B_sparsa = sparse(B_piena);
» flops(0)
» tic;A_sparsa*B_sparsa;tempo_prodotto_sparso=toc;
» costo_prodotto_sparso=flops;
» ['costo prodotto pieno' 'costo prodotto sparso']
» [costo_prodotto_pieno costo_prodotto_sparso]
» ['tempo prodotto pieno' 'tempo prodotto sparso']
» [tempo_prodotto_pieno tempo_prodotto_sparso]
```

Si noti l'uso delle istruzioni di analisi delle prestazioni:

`flops(0)`	azzera il contatore delle operazioni floating point;
`flops`	visualizza il contatore delle operazioni floating point;
`tic`	azzera il cronometro parziale;
`toc`	visualizza il cronometro parziale.

La function `sparse` permette di definire una variabile di tipo matrice sparsa direttamente a partire da tre vettori contenenti rispettivamente gli elementi non nulli della matrice ed i relativi indici ed, eventualmente, le dimensioni della matrice sparsa che si vuole generare. Ricordando che una matrice sparsa di Matlab può essere solo bidimensionale, la sintassi più generale è

```
» S=sparse(I,J,e_non_nulli,m,n,max_num_e_non_nulli);
```

dove il vettore `e_non_nulli` specifica i valori non nulli da inserire nelle posizioni specificate in `I`, `J` ed i parametri (opzionali) `m`, `n`, `max_num_e_non_nulli` rispettivamente il numero di righe e di colonne ed il massimo numero di elementi non nulli da allocare per la matrice.

Il seguente esempio mostra la definizione della variabile `Trid_sparsa` di tipo matrice sparsa con il valore (matrice tridiagonale):

$$\begin{pmatrix} 2 & 1 & 0 & 0 \\ -1 & 3 & 5 & 0 \\ 0 & -5 & 3 & 1 \\ 0 & 0 & -1 & 2 \end{pmatrix}$$

```
» Trid_sparsa = sparse(1:4,1:4,[2 3 3 2],4,4,4) + ...
                sparse(1:3,2:4,[1 5 1],4,4,3) + ...
                sparse(2:4,1:3,[-1 -5 -1],4,4,3);
```

Per definire matrici sparse tridiagonali, o più in generale "a banda", oltre a `sparse` si può usare la function `spdiags` che definisce una matrice sparsa a partire dagli elementi della diagonale principale e di quelle ad essa parallele; per esempio la precedente matrice `Trid_sparsa` può essere equivalentemente definita (si noti l'offset) come:

```
» d_prin=[2 3 3 2]';
» d_sup=[0 1 5 1]';
» d_inf=[-1 -5 -1 0]';
» Trid_sparsa = spdiags([d_inf d_prin d_sup],-1:1,4,4);
```

La function `find` determina i valori non nulli, insieme con i loro indici di riga e di colonna, di una qualunque variabile, sia di tipo matrice piena sia di tipo matrice sparsa[9]:

```
» S=sparse(A);
» [I,J,e_non_nulli]=find(S);
» [I,J,e_non_nulli]=find(A_piena);
```

le due `find` forniscono entrambe gli stessi vettori `I`, `J` ed `e_non_nulli`.

La maggior parte delle function cui sono demandate le principali azioni su variabili di tipo matrice hanno una corrispondente function che effettua la stessa azione su variabili di tipo matrice sparsa; per esempio le function `spdiags(A)` e `diag(A)`, dove `A` è di tipo pieno o di di tipo sparso, entrambe estraggono la diagonale di `A`.

[9] `find(b)` dove `b` è un vettore definisce il vettore degli indici degli elementi non nulli di `b`.

Altri esempi sono:

```
» speye(m,n);
» spones(S);
» sprand(S);
```

Si noti che `spones` e `sprand` definiscono rispettivamente una matrice sparsa, con la stessa struttura di `S`, ma con tutti gli elementi non nulli uguali a uno, ed una matrice sparsa, con la stessa struttura di `S`, i cui elementi non nulli sono numeri casuali.

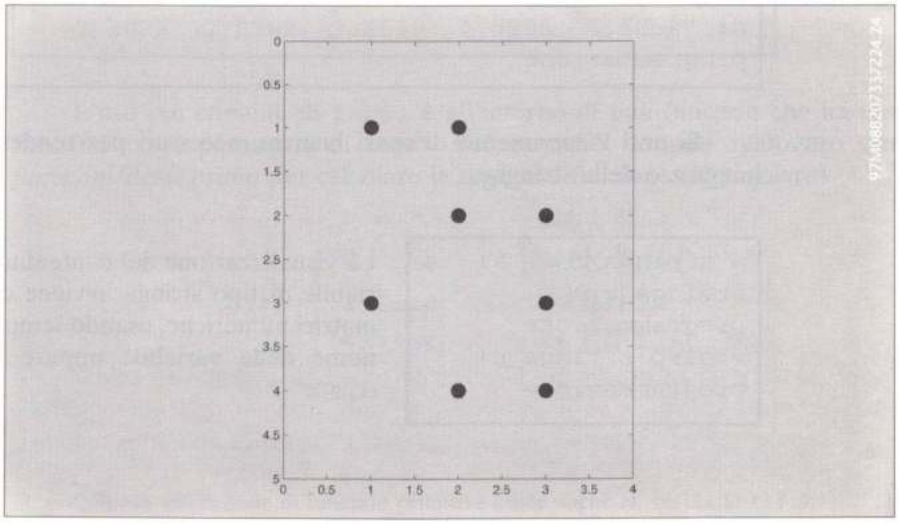
La function `spfun` permette di valutare una qualunque funzione (implementata come function Matlab) con un argomento di tipo matrice sparsa o piena, restituendo il risultato di tipo matrice sparsa; per esempio:

```
» spfun('cos',S);
» spfun('exp',A);
```

L'indice di densità di una matrice definito come rapporto tra il numero degli elementi non nulli e il numero totale di elementi può essere calcolato mediante

```
» nnz(S)/(prod(size(S))
ans =
   0.6667
```

Una function utile nelle computazioni con matrici sparse, è `spy` che visualizza la struttura di sparsità di una matrice visualizzando dove sono posizionati gli elementi non nulli; ad esempio `spy(S)` produce



Matlab ha un certo numero di function primitive per azioni su stringhe di caratteri.

```
» length('Matlab');
ans =
   6
```

La function `length` determina il numero di caratteri una stringa

Ricordando che una stringa Matlab è una matrice di caratteri, due (o più) variabili di tipo stringa possono essere concatenate nel modo usuale (concatenazione per colonne)

```
» stringa_a='laboratorio';
» stringa_b=' di ';
» stringa_c='programmazione';
» stringa=[stringa_a stringa_b stringa_c]
stringa =
laboratorio di programmazione
```

oppure concatenate verticalmente (concatenazione per righe) purchè le stringhe abbiano lo stesso numero di caratteri:

```
» a_str='laboratorio   '; b_str='di          ';
» stringa=[a_str; b_str; stringa_c]
stringa =
laboratorio
di
programmazione
```

Si noti l'inserimento di spazi bianchi, necessari per rendere uguali le lunghezze delle stringhe.

```
» stringa_c
stringa_c =
programmazione
» disp(stringa_c)
programmazione
```

La visualizzazione del contenuto di una variabile di tipo stringa avviene come per le matrici numeriche, usando semplicemente il nome della variabile oppure la function `disp`:

```
» strcmp(stringa,stringa_a)
ans =
0
» strncmp(stringa,stringa_a,11)
ans =
1
```

La function `strcmp` confronta le due variabili di tipo stringa restituendo un risultato booleano: 1 se le variabili sono uguali, 0 altrimenti.

La function `strncmp(...,...,n)` confronta invece i primi `n` caratteri delle due stringhe.

L'istruzione `eval(<stringa_testo>)` è una richiesta di interpretare il contenuto della variabile `<stringa_testo>` come il testo di una istruzione Matlab e di eseguire l'istruzione stessa; per esempio

```
» stringa = '3+5';
» eval(stringa);
ans =
   8
```

L'istruzione `feval('fun',<argomenti_di_fun>)` esegue la function il cui nome è la stringa `fun` con gli argomenti specificati; `fun` può essere una function primitiva[10] o una qualunque function definita in un M-file. Per esempio

```
» c=feval('log',x);
ans =
   2.3026  2.8782  3.4539  4.9295  4.6052
```

L'uso più comune di `feval` è all'interno di una function che ha una function tra i parametri di input. Per esempio, avendo realizzato due *function-file*: il primo per calcolare la seguente funzione (triangolare)

$$f(x)=\begin{cases} 0 & \text{se} \quad x\in\,]-\infty,0]\cup[2,+\infty[\\ x & \text{se} \quad x\in\,]0,1] \\ 2-x & \text{se} \quad x\in\,]1,2[\end{cases}$$

[10] Per la valutazione di function primitive esiste anche la `builtin('fun',...)`.

```
% function file mia_fun.m
%
function y=mia_fun(x)
        y=zeros(size(x));
        punt=find(x>1 & x<2);
        y(punt)=2-x(punt);
        punt=find(x>0 & x<=1);
        y(punt)=x(punt);
```

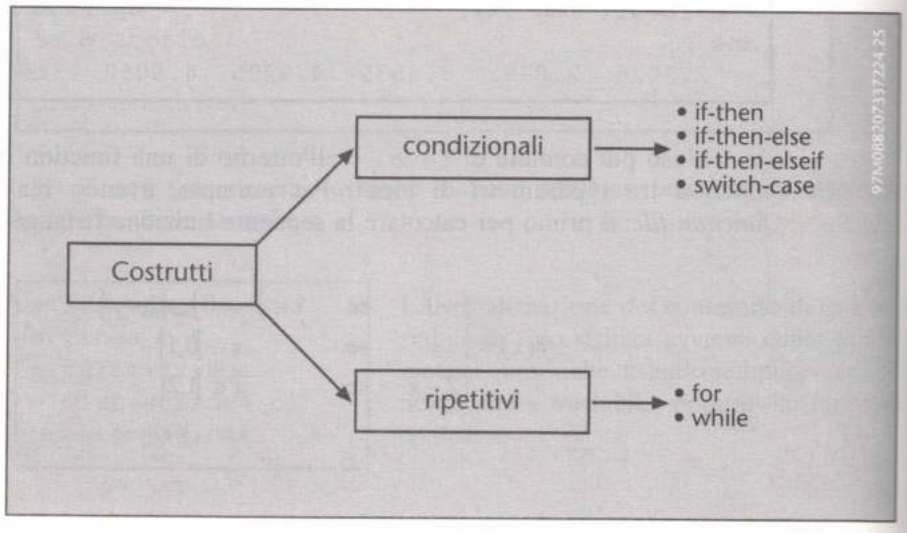
ed il secondo per campionare una qualsiasi funzione $\phi(x)$ in un intervallo [a, b]:

```
% function file campiona.m
function f_campionata=campiona(fun,a,b,n_campioni)
griglia=linspace(a,b,n_campioni);
f_campionata=feval(fun,griglia);
```

si può allora richiamare la function `campiona` specificando la funzione da campionare di volta in volta.

4.8 Costrutti di controllo

In Matlab esistono i seguenti costrutti di controllo:



Il costrutto di selezione `if_then_else` è realizzato in Matlab con

```
» if <espressione>
        <comandi_1>
  else
        <comandi_2>
  end
```

dove

- `<espressione>` è in generale una matrice di valori logici;
- `<comandi_1>` sono le istruzioni che vengono eseguite se *tutti* gli elementi di `<espressione>` sono *non nulli* (cioè considerati veri);
- `<comandi_2>` sono le istruzioni che vengono eseguite se *almeno* un elemento di `<espressione>` è *nullo* (cioè considerato falso).

Il costrutto di selezione in Matlab è una generalizzazione del corrispondente costrutto in P-like, dove invece `<espressione>` è un predicato, cioè una espressione logica il cui valore scalare è *vero* oppure *falso*.

Il costrutto di selezione in Matlab prevede anche la forma semplificata

```
» if <espressione>
      <comandi_1>
  end
```

e la forma multipla `if-elseif`, come nell'esempio

```
» if A>B
      A=A-B;
  elseif B>A
```

```
» n=input('valore di n (<20) ');
» switch n
  case {1,2,3,5,7,11,13,17,19}
     disp('n è numero primo')
  case {4,8,16}
     disp('n è potenza di 2')
  case 9
     disp('n è potenza di 3')
   otherwise
     disp('n ha vari fattori primi')
  end
```

Il costrutto iterativo `for` viene denotato in Matlab con :

```
» for <indice>=<val_iniz:passo:val_fin>
        <comandi>
  end
```

dove `val_iniz` e `val_fin` sono rispettivamente il valore iniziale e finale di `indice`; e `passo` è il passo di incremento[11]. Per esempio, i seguenti due cicli for innestati costruiscono la *matrice di Hilbert* di dimensione `n`:

```
» for i=1:n
     for j=1:n
        A(i,j)=1/(i+j-1);
     end
end
```

Il cotrutto iterativo `while` viene denotato con:

```
» while <espressione>
    <comandi>
  end
```

[11] Più in generale si può avere `for <indice_ciclo>=<vettore>` e il ciclo viene eseguito assegnando a ogni iterazione a `<indice_ciclo>` il valore di un elemento di `<vettore>`.

I <comandi> nel ciclo vengono eseguiti se *tutti* gli elementi di <espressione>, che può essere una matrice, sono *non nulli* (cioè sono considerati veri).

La seguente function implementa la funzionalità della function primitiva any:

```
function y = mia_any(x)
y=0; k=1;
while (k<= length(x) & y == 0)
        if x(k) ~=0
    y=1;
end
k=k+1;
end
```

Si noti che quanto detto a proposito di <espressione> nel costrutto if continua a valere anche nel caso del costrutto while.

In Matlab non è realizzato il costrutto repeat.

> È importante sottolineare che la possibilità di effettuare chiamate innestate a function, le azioni componente per componente e la potente notazione per le porzioni di matrici, cioè in breve lo stile di programmazione funzionale ed a parallelismo sui dati offerto dal linguaggio di Matlab, riducono in modo drastico la necessità di ricorrere ai costrutti iterativi (e anche al costrutto di selezione) nella descrizione di un algoritmo.

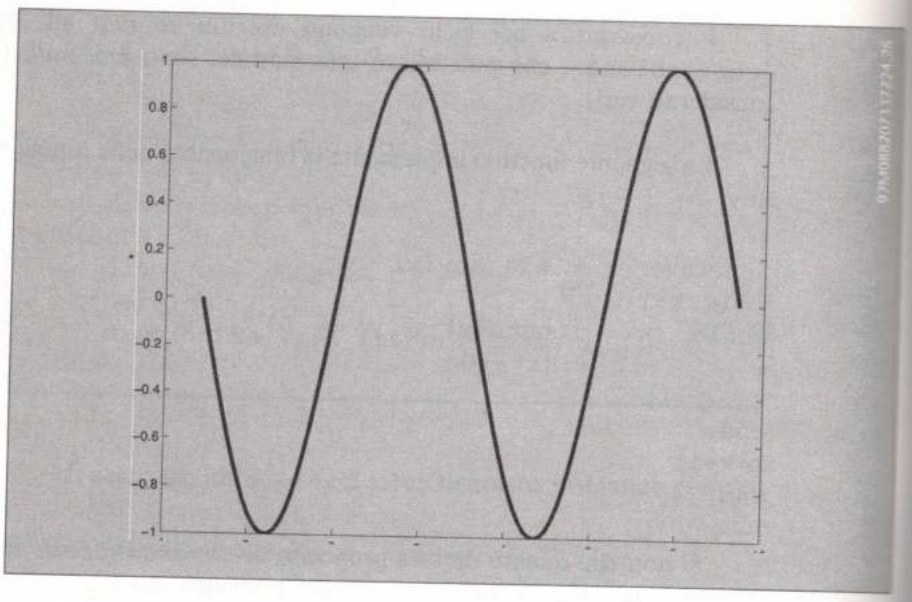
4.9 Grafica

Matlab consente di realizzare grafici bidimensionali e tridimensionali.

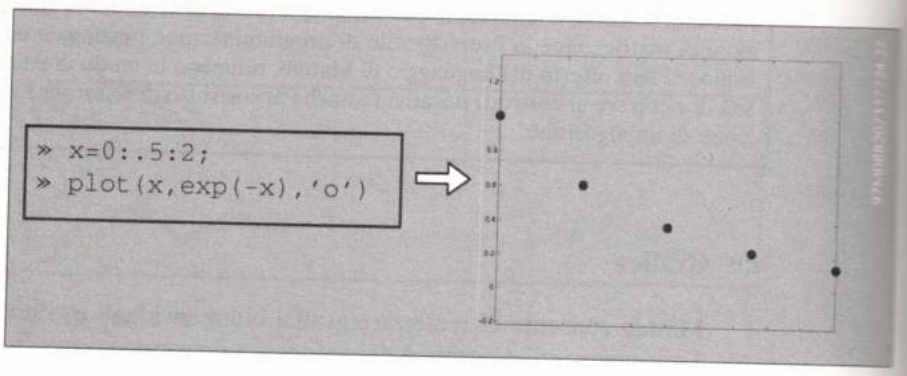
La visualizzazione grafica dei valori di un vettore y rispetto a quelli di un vettore x si ottiene con la function

```
plot(x,y)
```

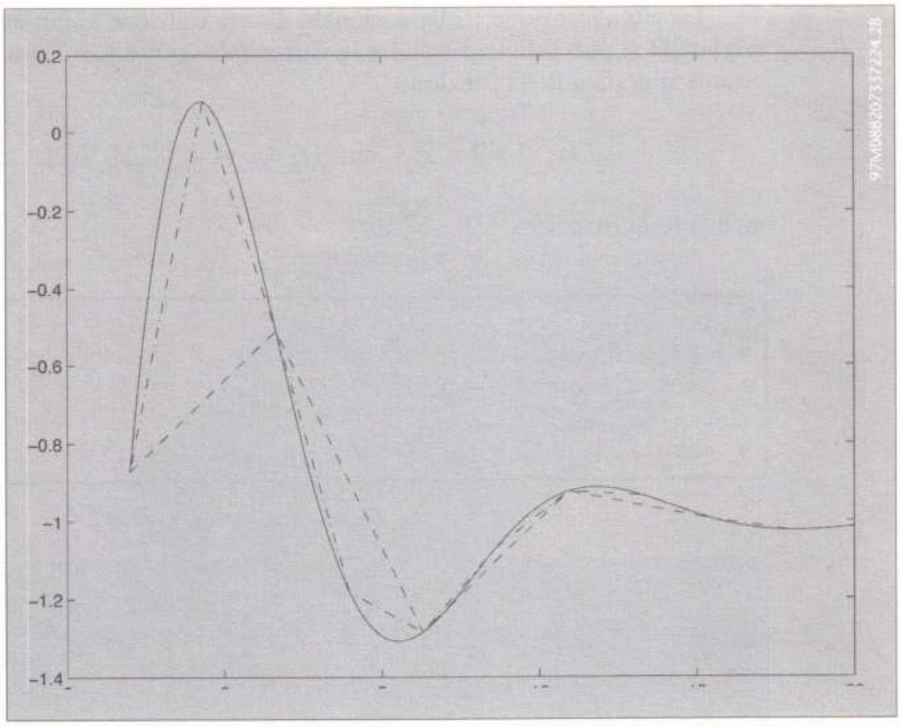
che genera, se necessario, una nuova finestra contenente il grafico. Per esempio se il valore di y è quello restituito da campionamento('sin',-pi,3*pi,100), il grafico visualizzato è



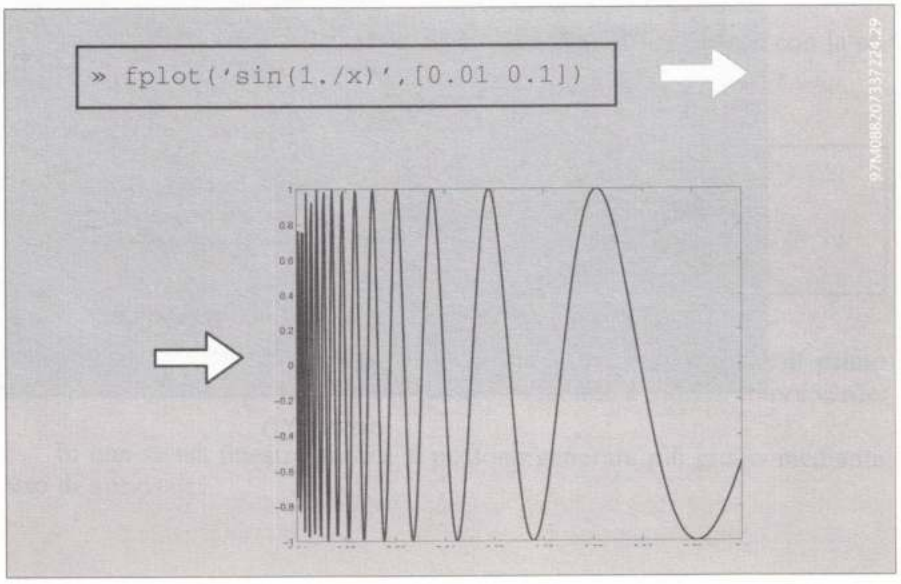
Una possibile visualizzazione per punti, utile per nel caso di dati intrinsecamente discreti si ottiene come mostra il seguente esempio



Se la griglia, e quindi la corrispondente campionatura della funzione, è sufficientemente fitta, il risultato visivo è una curva liscia. Si guardi, ad esempio, nel grafico seguente le tre curve ottenute campionando una stessa funzione su tre griglie che contengono rispettivamente 100, 10 e 5 punti.



Il modo più semplice di produrre il grafico di una funzione *f(x)* è tramite la function `fplot`; ad esempio

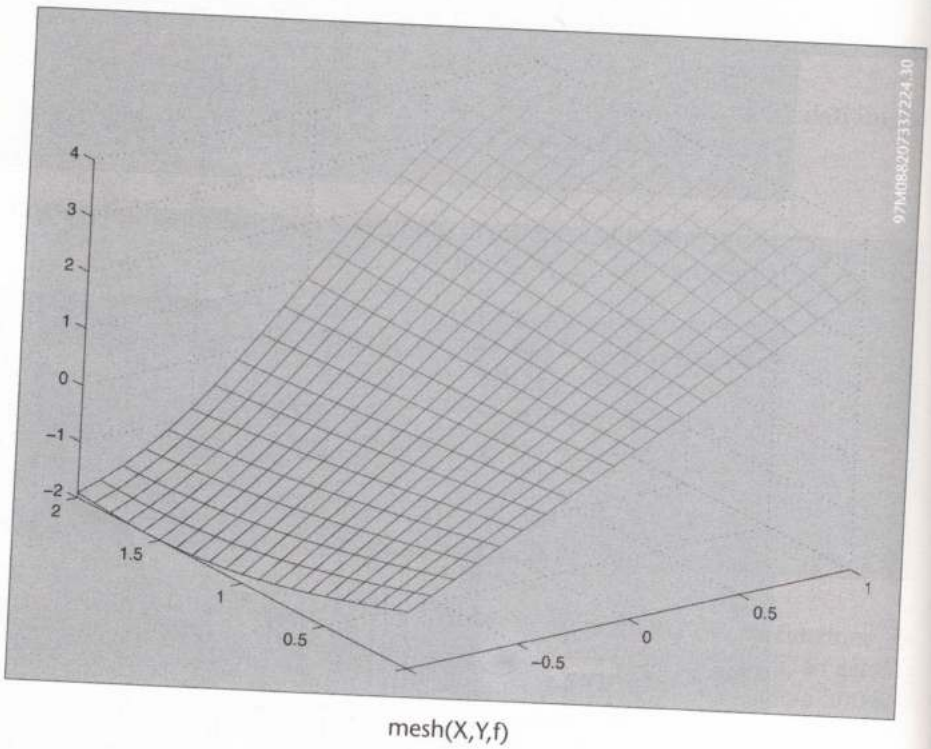


La visualizzazione tridimensionale di una funzione campionata su una griglia 2D si può ottenere tramite le istruzioni `mesh` e `surf`; ad esempio si ottiene il grafico della funzione

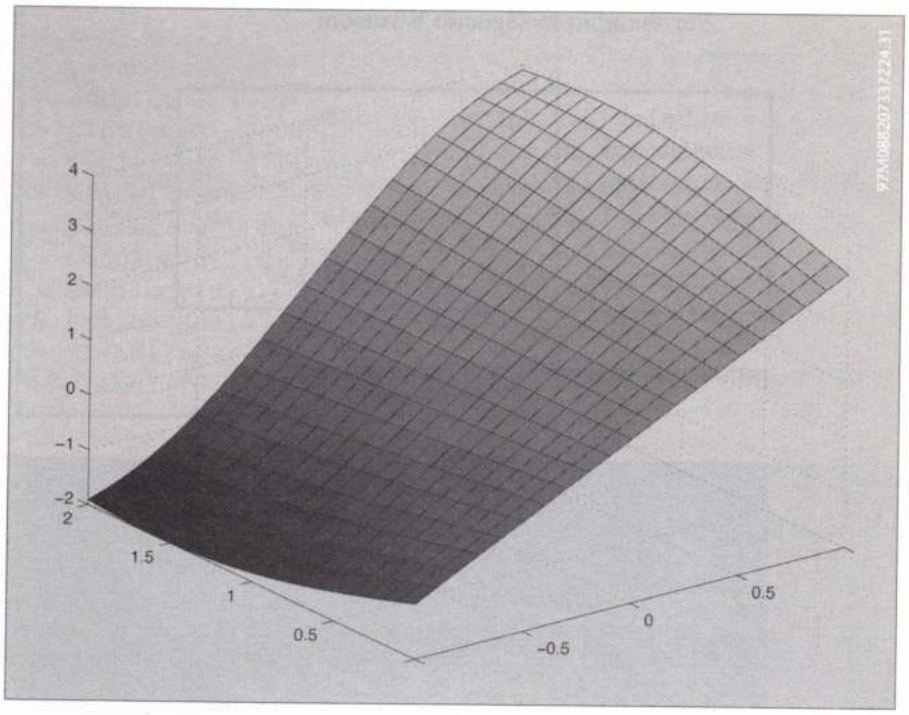
$$f(x, y) = 1 + x^2 + \sin(xy), \quad (x, y) \in [-1,1], [0,2]$$

mediante le istruzioni

```
» x=-1:0.1:1;
» y=0:0.1:2;
» [X,Y}=meshgrid(x,y);
» f=1+X.*2+sin(X.*Y);
» mesh(X,Y,f)     % oppure     surf(X,Y,f))
```



mesh(X,Y,f)



surf(X,Y,f)

La visualizzazione nella stessa figura di più grafici si ottiene con la seguente

```
» plot(x1,y1,x2,y2)
```

oppure con

```
» plot(x1,y1,'r.')
» hold on
» plot(x2,y2,'g-')
» hold off
```

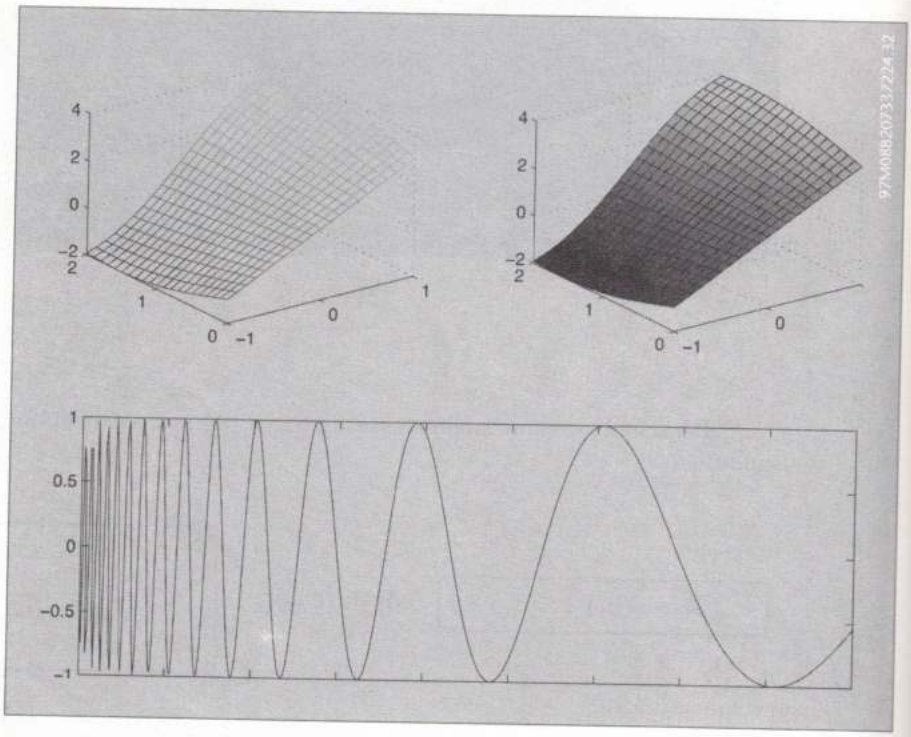
dove le opzioni, racchiuse tra apici nella parentesi, indicano che il primo grafico è disegnato a punti di colore rosso e il secondo a tratti di colore verde.

In una stessa finestra grafica si possono generare più grafici mediante l'uso di `subplot`.

Per esempio le seguenti istruzioni

```
» subplot(2,2,1)
» mesh(X,Y,f)
» subplot(2,2,2)
» surf(X,Y,f)
» subplot(2,1,2)
» fplot('sin(1./x)',[0.01 0.1])
```

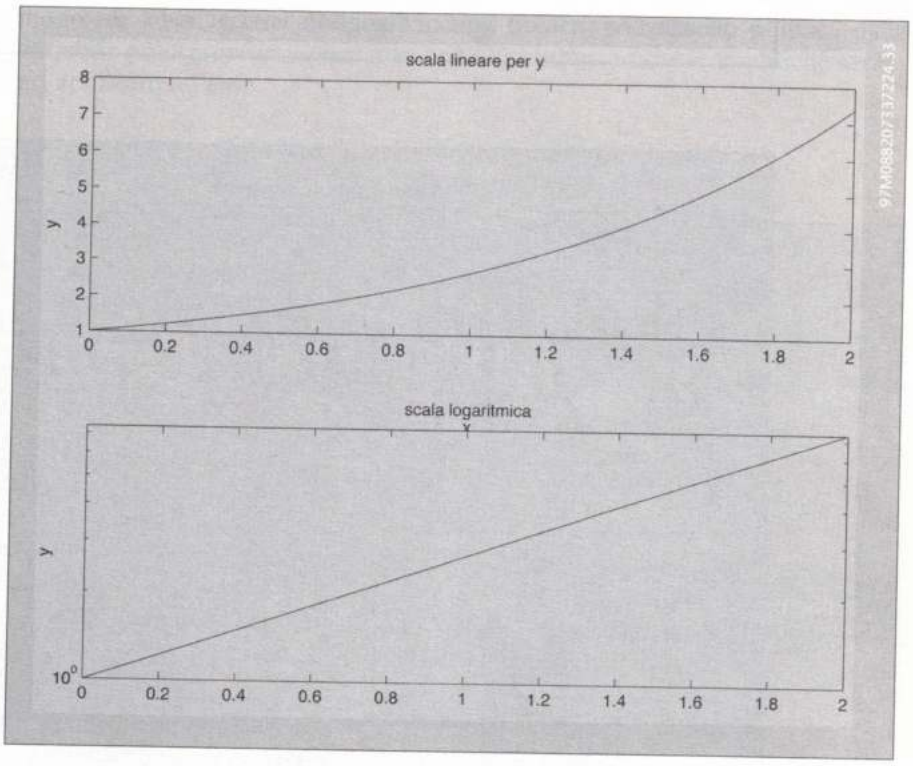
producono la seguente finestra grafica



Mediante la function `subplot(m,n,k)` la finestra grafica è logicamente divisa in una matrice (di `m` righe ed `n` colonne) di grafici ciascuno numerato da 1 ad `mn` procedendo dall'angolo alto-sinistro all'angolo basso-destro.

Altre function per la grafica 2D sono utilizzate nell'esempio che segue

```
» x=0:.01:2;
» y=exp(x);
» subplot(2,1,1)
» plot(x,y)
» title('scala lineare per y')
» xlabel('x')
» ylabel('y')
» subplot(2,1,2)
» semilogy(x,y)
» title('scala logaritmica')
» xlabel('x')
» ylabel('y')
```

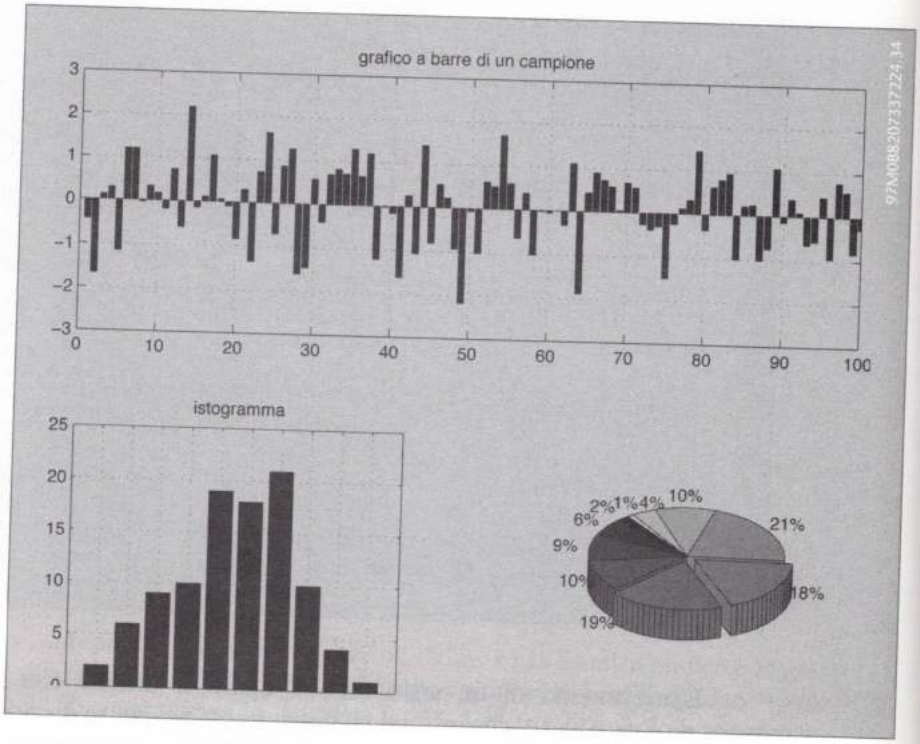


L'inserimento di un testo in una figura si ottiene con la function `text(x,y,';;;')` che inserisce il testo `';;;'` nel punto di coordinate `x,y` della figura (nel sistema di riferimento della figura), oppure con la function `gtext(';;;')` che inserisce il testo nel punto della figura dove si clicca con il mouse.

Per una trattazione completa delle numerose function per la grafica si rimanda a un manuale di Matlab.

L'esempio seguente mostra alcune function per la visualizzazione di vettori di dati.

```
» x=randn(100,1);
» subplot(2,1,1); bar(x)
» axis([0 100 -3 3]); grid
» title('grafico a barre di un campione','color','r')
» n=hist(x); subplot(2,2,3); bar(n)
» axis([0 11 0 25]); grid
» title('istogramma', 'color','r')
» subplot(2,2,4); pie3(n,[0 0 0 0 1 1 0 0 0 0])
» title('grafico a torta 3D','color','r')
```



4.10 Grafica avanzata

Il sottosistema grafico di Matlab consente di creare, oltre a grafici e animazioni, interfacce interattive come menù, pulsanti e barre di scorrimento. La gestione delle finestre grafiche è di tipo *object oriented*: l'intero schermo è il nodo radice di una struttura gerarchica (albero) in cui ogni nodo, accessibile tramite un puntatore (*handle*), possiede degli attributi (*object properties*) modificabili dall'utente.

Ogni nodo proviene da un unico nodo padre (*parent*) e conduce ad eventuali nodi figli (*children*). Solo lo schermo non ha padre; mentre i suoi figli sono le varie finestre grafiche aperte. Indipendentemente dalle finestre grafiche aperte solo una è attiva in quel momento: è attiva la finestra cui punta la variabile predefinita `gcf` (*graphic current figure handle*). Allo stesso modo tra i vari grafici disegnati in una finestra solo quello puntato dalla variabile `gca` (*graphic current axes handle*) è attivo e tutte le modifiche saranno applicate ad esso.

La Fig. 4.2 visualizza l'*albero* degli oggetti grafici in Matlab.

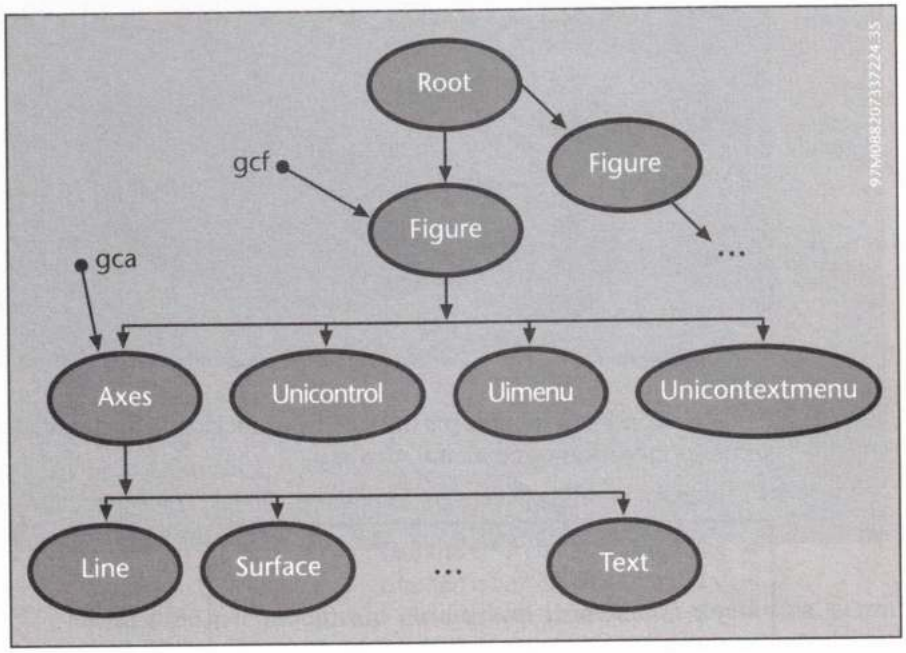
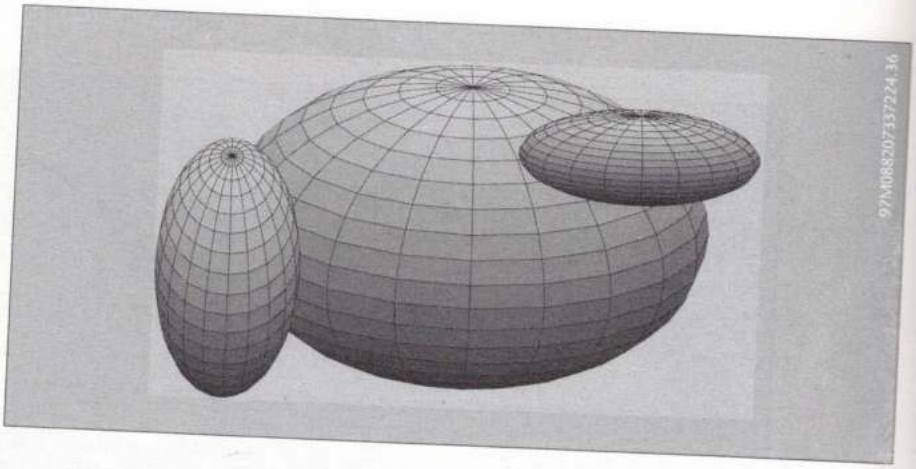


Fig. 4.2: albero degli oggetti grafici

Rimandando i dettagli a un manuale specifico, si vuole, con qualche esempio, mostrare come la modifica di un attributo di un oggetto può cambiarne l'aspetto. Le function che consentono rispettivamente di determinare il valore degli attributi e di modificarlo sono `get` e `set`.

Il primo esempio sovrappone il grafico di tra superfici ottenute da una medesima superficie sferica che però viene disegnata in finestre grafiche di dimensioni diverse:

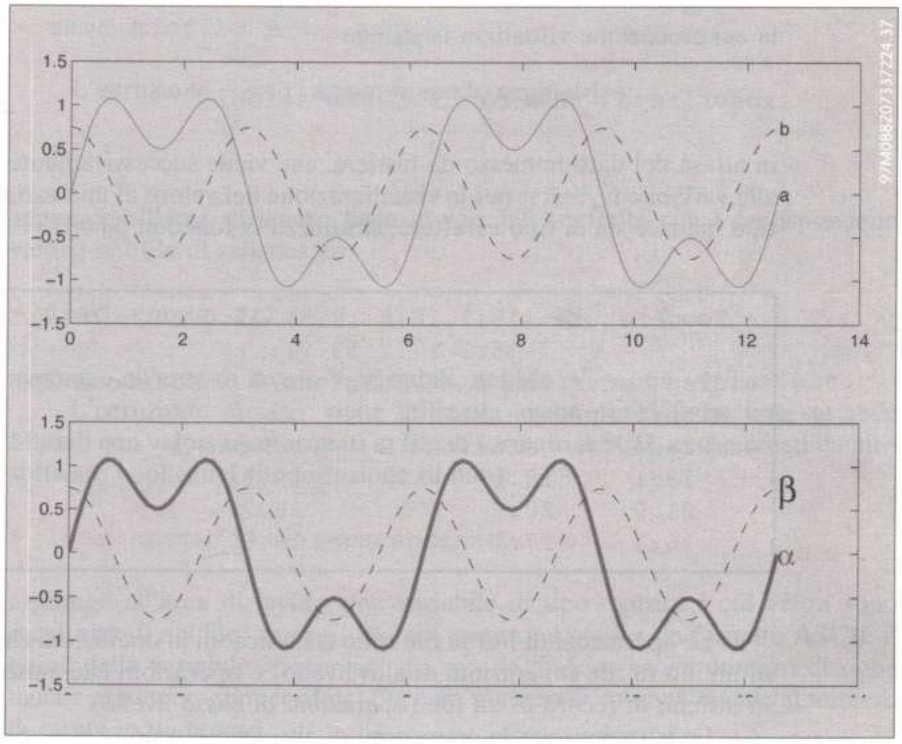
```
» h(1)=axes('Position',[0 0 1 1]);
» sphere
» h(2)=axes('Position',[0 0 .3 .8]);
» sphere
» h(3)=axes('Position',[.5 .5 .5 .3]);
» sphere
» set(h,'Visible','off')
```



Il secondo esempio, a partire da due grafici identici, cambia l'aspetto del secondo modificandone alcuni attributi.

```
» x=linspace(0,4*pi,100);
» y1=sin(x)+.5*sin(3*x);
» y2=.75*cos(2*x);
» subplot(2,1,1)
» plot(x,y1,'r',x,y2,'b')
» text(x(100),y1(100),'a');
» text(x(100),y2(100),'b');
```

```
» subplot(2,1,2)
» plot(x,y1,'r',x,y2,'b')
» text(x(100),y1(100),'a');
» text(x(100),y2(100),'b');
» pt=get(gca,'Children');
» set(pt(4),'LineWidth',2.5)
» set(pt(1),'Color','c', …
     'FontName','Symbol', …
     'FontSize',20)
» set(pt(2),'Color','m', …
     'FontName','Symbol', …
     'FontSize',16)
```



Molto utile per modificare gli attributi in modalità interattiva, senza doverne ricordare il nome ed i possibili valori, è il ricorso all'interfaccia *Graphics Property Editor* di Matlab. Basta immettere

```
» propedit(<handle>)
```

perchè si apra una finestra che visualizza gli attributi dell'oggetto puntato da `<handle>` e consente di modificarli.

4.11 Input e Output

Le operazioni di I/O in Matlab riguardano tastiera e schermo oppure file di memoria secondaria.

Nell'utilizzo interattivo, l'immissione di dati e istruzioni da tastiera, e la visualizzazione dei risultati sono parte integrante del linguaggio.

Nell'utilizzo di script e function è demandata alla function `input` l'immissione di dati da tastiera, come ad esempio

```
n_dati=input('inserire il numero di osservazioni')
```

la cui esecuzione visualizza la stringa

```
inserire il numero di osservazioni_
```

in attesa del dato immesso da tastiera, che viene successivamente assegnato alla variabile `n_dati`; per la visualizzazione del valore di una variabile, sia di tipo matrice sia di tipo carattere, si utilizza la function `disp`.

```
» Temperature=[18.1 18.8 19.5; 21.0 20.5 26.7;
               16.3 17.7 19.9];
» disp('stazione 1    stazione 2    stazione 3')
» disp(Temperature)
stazione 1    stazione 2    stazione 3
   18.1       18.8           19.5
   21.0       20.5           26.7
   16.3       17.7           19.9
```

Le operazioni di I/O su file sono classificabili in operazioni che agiscono su un intero file (operazioni di alto livello) e operazioni che agiscono anche su insiemi di record di un file (operazioni di basso livello).

Le istruzioni per le operazioni di alto livello sono `save` e `load`.

L'istruzione

```
» save
```

crea il file, detto talvolta *di salvataggio*, `matlab.mat` e vi memorizza nomi e valori di tutte le variabili nel *workspace*. L'estensione `.mat` indica che i

dati sono rappresentati nel file secondo il formato binario standard di Matlab. Invece

```
» save A b
```

memorizza nel file di salvataggio solo le variabili A e b. In generale, l'istruzione

```
» save <nome file> <lista variabili>
```

crea il file <nome file>.mat e vi memorizza nomi e valori di tutte le variabili in <lista variabili>; si può ottenere la la memorizzazione in un file testo (formato ASCII) aggiungendo nell'istruzione l'opzione -ascii, come nell'esempio

```
» save miofile A b -ascii
```

L'istruzione load agisce in modo simmetrico:

```
» load
```

aggiunge all'area di lavoro tutte le variabili (definite con i corrispondenti valori) nel file di salvataggio;

```
» load <nome file>
```

aggiunge all'area di lavoro le variabili nel file <nome file>.mat.

L'istruzione load viene utilizzata anche per definire una variabile Matlab con valori memorizzati in file di formato ASCII, anche creati da altri software applicativi (importazione di dati):

```
» load <nome file>.<estensione>
```

aggiunge all'area di lavoro una variabile di tipo matrice i cui valori sono memorizzati nel file <nome file>.<estensione> in formato ASCII; il nome della variabile è <nome file>; la matrice ha un numero di righe uguale al numero di record del file e un numero di colonne uguale al numero di campi in un record.

Per esempio, se il file Temperature.dat è il file testo costituito dai tre record

```
18.1      18.8      19.5
21.0      20.5      26.7
16.3      17.7      19.9
```

allora, l'istuzione

```
» load Temperature.dat
```

definisce la variabile `Temperature` di tipo matrice di `size` 3,3, il cui valore è:

```
» Temperature
Temperature =
   18.1      18.8          19.5
   21.0      20.5          26.7
   16.3      17.7          19.9
```

Le istruzioni per le operazioni di basso livello sono

`fread` e `fwrite`, che agiscono su file binari;
`fscanf` e `fprintf`, che agiscono su file testo.

Inoltre le istruzioni `fopen` e `fclose` effettuano le operazioni di apertura e chiusura di un file. Nel seguito si mostano alcuni esempi di uso di alcune istruzioni di basso livello; per approfondimenti si rinvia alla documentazione Matlab.

Il comando per aprire un file è `fopen` e richiede due dati di input: la stringa indicante il nome del file e una stringa che serva a precisare la modalità di apertura, da scegliere tra:

- `'r'` lettura
- `'w'` scrittura
- `'a'` scrittura in aggiunta
- `'r+'` per scrivere e leggere.

`fopen` restituisce in output un numero che identifica il file e che va richiamato qualora si voglia operare sul file; per esempio,

```
» id=fopen('dati.txt','a')
```

apre un file di nome `dati.txt`, per potervi scrivere senza cancellare il vecchio contenuto, attribuendo il numero che lo identifica alla variabile `id`. Per terminare la lettura o scrittura del file basta il comando

```
» fclose(id).
```

La funzione `printf` permette di convertire dati in stringhe di caratteri, e di scrivere stringhe di caratteri in un file. Essa richiede due dati di input:

- l'identificativo del file su cui scrivere (se omesso, l'output sarà visualizzato sullo schermo)
- la stringa da scrivere, eventualmente con caratteri speciali per formattare l'output.

ESEMPIO: creare una tabella con i valori della funzione *sin(x)* per valori di x tra 0 e π:

```
» x=linspace(0,pi,10);
» y=[x;sin(x)];
» fi=fopen('tabella.txt','w');
» fprintf(fi,'valori della funzione sin\n\n');
» fprintf(fi,'%6.2f %10.7f\n',y);
» fclose(fi);
```

La prima chiamata a `fprintf` permette di scrivere nel file `tabella.txt` il titolo 'valori della ...', poi di andare a capo e di lasciare una linea vuota con il comando `\n`.

La seconda chiamata a `fprintf` consente di scrivere i valori della tabella, formattandoli secondo le specifiche che seguono il carattere **%**. Le specifiche di conversione ammesse sono

`%e` notazione esponenziale
`%f` notazione in virgola fissa.

4.12 Algoritmi elementari in Matlab

Diamo qui solo pochi esempi di come alcuni algoritmi elementari, già sviluppati in P-like nel capitolo 3, possano essere implementati usando il linguaggio Matlab. Si vuole mostrare soprattutto come il linguaggio Matlab sia utilizzabile secondo lo stile di programmazione per così dire classico, cioè per tradurre in modo immediato gli algoritmi descritti in P-like. Non viene utilizzato lo stile avanzato di programmazione in Matlab detto a parallelismo sui dati.

ESEMPIO: calcolo della somma di potenze.

```
function y = sommapotenze(x,n)
y = 1;
potenza = 1;
for i=1:n
    potenza = potenza*x;
    y = y + potenza;
end
```

ESEMPIO: calcolo del massimo e del minimo elemento di un array 1D e dei loro indici

dati di input: l'array `a`, il size `n`
dato di output: `max_a,i_max,min_a,i_min`
costrutto ripetitivo: `for (step 2):`
operazione ripetuta (al generico passo `i`): confronto tra la coppia costituita dall'`i`–simo e dall'`(i+1)`–simo elemento dell'array ed eventuale modifica delle variabili di output.

```
function [max_a,i_max,min_a,i_min] = max_min(a,n)
% calcolo del massimo e del minimo elemento di un array
% 1D e dei loro indici.
% massima efficienza nel caso peggiore.
max_a = a(1); i_max = 1;
min_a = a(1); i_min = 1;
for i=2:2:n-1
    if a(i) > a(i+1)
        if a(i) > max_a
           max_a = a(i);
           i_max = i;
      end
       if a(i+1) < min_a
          min_a = a(i+1);
          i_min = i+1;
       end
   else
       if a(i) < min_a
          min_a = a(i);
          i_min = i;
      end
      if a(i+1) > max_a
```

```
            max_a = a(i+1);
            i_max = i+1;
        end
    end
end
```

ESEMPIO: ordinamento per inserimento di un array 1D.

```
function a = ord_ins(a,n)
% ordinamento per inserimento 'in place'
for i=2:n
    el_da_ins = a(i);
    % inserimento dell'i-simo elemento al suo
    % posto nella porzione (1..i) di a
    j=i-1;
    while j >= 1 & el_da_ins < a(j)
        a(j+1)=a(j);
        j=j-1;
    end
    a(j+1) = el_da_ins;
end
```

ESEMPIO: ordinamento per selezione di un array 1D.

```
function a = ord_sel(a,n)
% ordinamento per selezione di minimo 'in place'
for i=1:n-1
    [min_array,i_min] = min_val_ind(a(i:n),n-i+1);
    % notare che i_min e' un indice locale della
    % porzione di array; esso va reso globale mediante
    % la somma dello spiazzamento (i-1)
    [a(i),a(i_min+i-1)] = scambiare(a(i),a(i_min+i-1));
end

function [min_array,i_min] = min_val_ind(a,n)
% calcolo del minimo elemento di un array 1D e
% del suo indice
min_array = a(1);
i_min = 1;
for i=2:n
    if a(i) < min_array
```

```
            min_array = a(i);
            i_min = i;
        end
    end
```

Si ricorda comunque che l'ordinamento è una operazione primitiva in Matlab (function `sort`).

ESEMPIO: fusione di due array 1D ordinati.

```
function c = fusione(a,b)
% fusione dei due array 1D a,b
% a e b sono array ordinati, con intersezione nulla.
n_a = length(a); n_b = length(b);
i_a = 1; i_b = 1;
for i_c=1:n_a+n_b
    if i_a <= n_a & i_b <= n_b
%ci sono sia elementi di a sia di b da considerare
        if a(i_a) < b(i_b)
            c(i_c) = a(i_a);
            i_a = i_a+1;
        else
            c(i_c) = b(i_b);
            i_b = i_b+1;
        end
    else
% uno dei due array non deve essere considerato
  if i_b > n_b
      %considerare solo a
      c(i_c) = a(i_a);
      i_a = i_a+1;
  else
      %considerare solo b
      c(i_c) = b(i_b);
      i_b = i_b+1;
  end
end
```

ESEMPIO: prodotto matrice (memorizzata in formato standard) per vettore.

```
function z = mat_vet(A,m,n,y)
% prodotto matrice per vettore;
```

```
% matrice e vettore pieni non strutturati, formato
% standard versione che richiama la function vet_xpay
for i=1:m
    z(i) = 0;
end
% calcolo della combinazione lineare delle colonne di A
% utilizzando gli elementi di y come coefficienti.
for j=1:n
    z = vet_xpay(z,A(1:m,j),m,y(j));
end

function vet = vet_xpay(x,y,n,alfa)
% somma di un vettore e di un multiplo di un altro
% vettore;
% vettori pieni
for i=1:n
    vet(i) = x(i)+alfa*y(i);
end
```

La seguente function è una versione alternativa a `mat_vet` ed è stata sviluppata utilizzando lo stile di programmazione a parallelismo sui dati, alcune function di base e la notazione di porzione di array di Matlab.

```
function z = mat_vet_1(A,y)
% prodotto matrice per vettore;
% matrice e vettore pieni non strutturati, formato
standard
% versione che richiama la function vet_xpay
    z = zeros(size(A,1),1);
% calcolo della combinazione lineare delle colonne di A
% utilizzando gli elementi di y come coefficienti.
for j=1:size(A,2)
    z = z+A(:,j)*y(j);
end
```

Si ricorda comunque che l'operazione di prodotto matrice per vettore è primitiva in Matlab `(z=A*y)`.

Come anticipato nel prologo del libro, Matlab deve essere il linguaggio di sviluppo del software relativo a tutti gli algoritmi presentati e a tutti gli esercizi proposti.

Il lettore è quindi invitato a realizzare il proprio insieme di script e function che implementano gli algoritmi che sono stati sviluppati nel testo.

Nella fase iniziale dell'apprendimento, il lettore dovrebbe utilizzare lo stile di programmazione classico per tradurre in Matlab gli algoritmi descritti in P-like. In una fase più avanzata, gli algoritmi dovrebbero essere implementati usando lo stile di programmazione a parallelismo sui dati e le function primitive di base di Matlab.

Bibliografia per approfondimenti

1. H.H. Goldstine *The computer from Pascal to von Neumann*, Princeton University Press, 1972.
2. A.S. Tanenbaum *Architettura del computer*, Prentice Hall, 1991.
3. T.H. Cormen, C.E. Leiserson, R.L. Rivest *Introduzione agli Algoritmi*, Jackson, 1995.
4. R. G. Dromey *Algoritmi Fondamentali*, Jackson, 1990.
5. C. Van Loan *Handbook of matrix computation*, SIAM, 1987.
6. W. J. Palm *Matlab 6 per l'ingegneria e le scienze*, McGraw-Hill, 2001.