Fondamenti di informatica

Le basi della programmazione avanzata

Riccardo Cassinis
Dipartimento di Elettronica per l'Automazione, Università di Brescia

Fondamenti di informatica

Le basi della programmazione avanzata

INDICE

PREFAZIONE

È sempre molto difficile affrontare il problema di fornire gli elementi di base di una disciplina in evoluzione così rapida come l'informatica.
Questo volume, concepito come supporto al corso di Fondamenti di Informatica II per allievi ingegneri elettronici, si rivolge a lettori non completamente digiuni dei fondamenti su cui si basa l'informatica, e dà per già acquisite alcune nozioni di base, come il concetto di computabilità e quello di algoritmo. Il testo presuppone inoltre che il lettore abbia un'idea della struttura di base dei calcolatori, e che conosca almeno un linguaggio di alto livello, preferibilmente il Pascal o il C.[1]
Il problema che si pone a questo punto è quello di scegliere fra una trattazione "classica" della materia, che si concentrerebbe forzatamente sullo studio degli algoritmi, e la decisione di affrontare svariati argomenti che, al giorno d'oggi, hanno acquisito un'importanza che fino a qualche anno fa erano ben lontani dall'avere.
In particolare, avendo seguito per diversi anni la formazione degli allievi del corso di laurea in Ingegneria Elettronica, mi sono reso conto del fatto che molto spesso essi vengono preparati ad affrontare bene problemi la cui importanza va progressivamente diminuendo, mentre in tutto il corso di studi non si fa quasi cenno a tecniche di programmazione che stanno sostituendo quelle classiche, ormai in buona parte obsolete. Mi riferisco con questo, ad esempio, alle interfacce a finestre, all'uso del mouse come principale mezzo di interazione con il calcolatore, ai sistemi ipertestuali e alla comunicazione fra calcolatori.
È chiaro che un unico corso universitario non è in grado di coprire tutti gli argomenti citati, ma ritengo che sia possibile, senza cadere nell'approssimazione, fornire una sufficiente quantità di nozioni *di base* su ognuno degli argomenti citati, in modo tale che l'allievo, trovandosi più avanti (e in ogni caso al momento della sua entrata nel

[1] Per la comprensione completa del capitolo 8, dedicato agli algoritmi, la conoscenza del linguaggio C è indispensabile. È quindi auspicabile che essa, se non è già posseduta dal lettore, venga acquisita con uno studio parallelo a quello del presente volume.

mondo del lavoro) costretto ad utilizzare tecniche avanzate di programmazione, possegga quanto meno un certo metodo per approfondire il loro studio.
La mia esperienza professionale mi ha inoltre convinto dell'impossibilità di separare i concetti di hardware e di software in campi, come ad esempio quello dell'automazione industriale, in cui l'interazione fra il calcolatore e il mondo esterno è sempre molto più stretta di quanto non avvenga negli altri settori di applicazione dei calcolatori. Problemi anche semplici di interfacciamento di sistemi costituiscono, per la maggior parte dei neolaureati, ostacoli quasi insormontabili.
La materia trattata è divisa in otto capitoli, dedicati ai seguenti argomenti:

- *la struttura di base dei calcolatori*, contenente i fondamenti dell'analisi e della sintesi dei circuiti digitali, e la struttura costitutiva dei microcalcolatori, con particolare riferimento alle interfacce fisiche con il mondo esterno;
- *la rappresentazione delle informazioni*, che, oltre a richiamare i principali metodi usati per i numeri e per le informazioni testuali, introduce alcuni esempi di rappresentazione di informazioni "non tradizionali" (immagini e suoni);
- *le basi dei sistemi operativi*, viste anche alla luce della programmazione "non modale" usata oramai in quasi tutti i programmi applicativi più diffusi;
- *i principi per la comunicazione fra calcolatori*, dalla semplice comunicazione seriale ai fondamenti delle reti di calcolatori;
- *l'interfaccia uomo-macchina*, con particolare riferimento alle interfacce a finestre;
- *i sistemi ipertestuali*, che stanno entrando di prepotenza anche nel mondo del software "tradizionale";
- *il paradigma di programmazione "message passing"*, alternativa in molti casi estremamente valida ai classici metodi di programmazione;
- *il concetto di complessità applicato agli algoritmi di base*, la cui importanza, determinante nel mondo dell'ingegneria, viene molto spesso trascurata, dando luogo, anche in campo professionale, a software inefficiente e alcune volte inutilizzabile.

La trattazione della materia è completata, ove opportuno, da una serie di problemi ed esercizi. Al termine del volume vengono date le soluzioni di alcuni problemi (contrassegnati da un asterisco), lasciando al lettore, secondo un metodo didattico universalmente riconosciuto valido, l'onere non solo di trovare la soluzione dell'esercizio, ma anche di dimostrare che la soluzione trovata è effettivamente corretta.

Riccardo Cassinis

Ringraziamenti
Ringrazio il Dott. Alessandro Rizzi, che ha fornito buona parte del materiale presentato nel capitolo 8, e che ne ha curato la revisione.
Ringrazio inoltre tutti i colleghi che hanno fornito, con discussioni e materiale, utili suggerimenti e spunti per la trattazione della materia, e in particolare i Proff. Giovanni Guida e Paolo Gubian.

A Zoe, e a Maria e Roberto

1

ELEMENTI COSTITUTIVI DEI CALCOLATORI

Questo capitolo contiene alcuni richiami all'algebra di commutazione, ai circuiti logici e alla struttura dei calcolatori, strettamente finalizzati alla comprensione delle parti seguenti del testo. La trattazione è lungi dall'essere completa: chi volesse approfondire la materia può trovare utili informazioni nella bibliografia.

1.1. L'ALGEBRA DI COMMUTAZIONE E I SUOI OPERATORI

L'algebra di commutazione è basata su:

- un alfabeto composto da due simboli: "**0**" e "**1**" (da cui anche il termine, del tutto equivalente, di *algebra binaria*);
- tre operatori elementari:
  - **NOT** (unario), rappresentato dal simbolo "/" o da una sbarretta orizzontale, detto anche "negazione";
  - **OR** (binario), rappresentato dal simbolo "+" o "∨", detto anche *somma logica*;

- **AND** (binario), rappresentato dal simbolo "•" o "||"detto anche *prodotto logico*.[2]

In altre parole, una variabile x può assumere il valore "0" o il valore "1", e

$$x = 0 \text{ se e solo se } x \neq 1$$

$$x = 1 \text{ se e solo se } x \neq 0$$

Gli operatori possono essere applicati ad una variabile (operatori unari) o a due variabili (operatori binari), e producono a loro volta una variabile il cui valore dipende dal tipo di operatore e dal valore delle variabili applicate.
Per rappresentare il comportamento degli operatori si utilizzano le cosiddette *tavole della verità*, che indicano il valore dell'operatore in corrispondenza di ogni possibile configurazione degli ingressi. Le tavole della verità possono essere indifferentemente disegnate sotto forma di matrici o di tabelle, come si vedrà più avanti.
La figura 1.1 riporta le tavole della verità per gli operatori elementari.

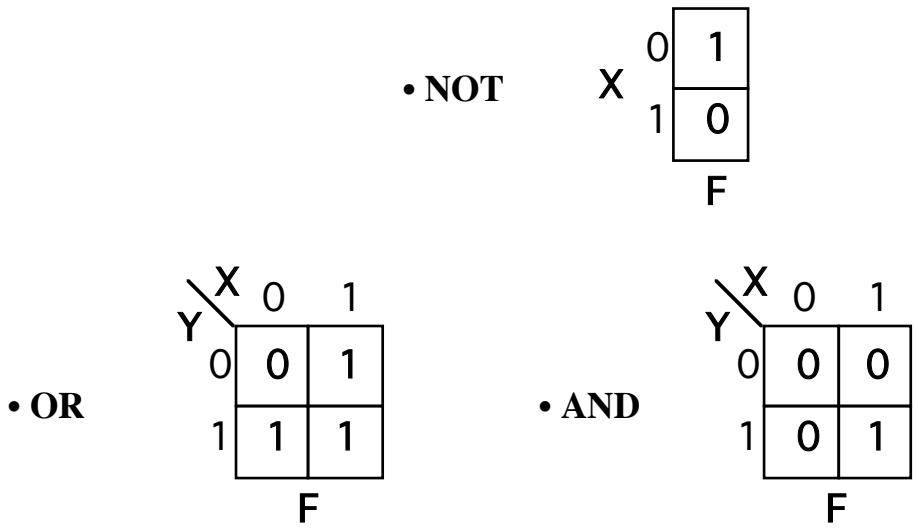


Figura 1.1. - Tavole della verità degli operatori elementari.

In seguito vedremo che gli operatori OR e AND si possono ottenere l'uno come combinazione dell'altro e dell'operatore NOT.
Un'espressione logica è una combinazione di simboli dell'alfabeto (operandi) e di operatori.
L'algebra di commutazione, che studia le proprietà delle espressioni scritte utilizzando i componenti di cui abbiamo appena parlato, è un particolare caso di algebra booleana. Nella pratica comune tuttavia, dal momento che l'algebra di

---

[2] Come è usuale nell'algebra, nella scrittura delle espressioni il simbolo del prodotto logico viene generalmente omesso: $x \bullet y = xy$.

commutazione è di gran lunga la più utilizzata fra tutte le algebre booleane, i due termini sono diventati (impropriamente) sinonimi.

1.1.1. ALCUNE PROPRIETÀ E TEOREMI DELL'ALGEBRA DI COMMUTAZIONE

Le proprietà e i teoremi che seguono sono i più utilizzati nella manipolazione delle espressioni algebriche booleane.
I teoremi sono dimostrabili per induzione matematica perfetta[3].

Negazione:	$\bar{\bar{x}} = x$
And e Or:	$x + 0 = x$ $x + 1 = 1$ $x \bullet 0 = 0$ $x \bullet 1 = x$
Idempotenza:	$x + x = x$ $x \bullet x = x$
Complementazione:	$x + \bar{x} = 1$ $x \bullet \bar{x} = 0$
Commutativa:	$x + y = y + x$ $x \bullet y = y \bullet x$
Associativa:	$x + (y + z) = (x + y) + z$ $x \bullet (y \bullet z) = (x \bullet y) \bullet z$
Distributiva:	$x \bullet (y + z) = x \bullet y + x \bullet z$ $x + (y \bullet z) = (x + y) \bullet (x + z)$
Assorbimento:	$x + (x \bullet y) = x$ $x \bullet (x + y) = x$
Teoremi di De Morgan:	$\bar{x} + \bar{y} = \overline{x \bullet y}$ $\bar{x} \bullet \bar{y} = \overline{x + y}$

I teoremi di De Morgan rivestono una notevole importanza in quanto permettono di scambiare operatori OR con operatori AND e viceversa, a patto di usare un certo numero di operatori NOT.

---

[3] Cioè applicando tutte le possibili configurazioni di variabili, e verificando la correttezza dell'enunciato.

1.1.2. INSIEMI FUNZIONALMENTE COMPLETI

Come già accennato, le proprietà dell'algebra di commutazione (e in particolare i teoremi di De Morgan) permettono, mediante opportune manipolazioni delle espressioni, di sostituire alcuni operatori con altri. Questo fa sì che sia possibile scrivere qualunque espressione combinatoria utilizzando un sottoinsieme dei tre operatori precedentemente definiti. Tali sottoinsiemi vengono definiti *funzionalmente completi*.

- NOT e AND costituiscono un insieme funzionalmente completo;
- NOT e OR costituiscono un insieme funzionalmente completo;
- AND e OR non costituiscono un insieme funzionalmente completo.

1.1.3. OPERATORI UNIVERSALI

Combinando gli operatori visti in precedenza è possibile definire due nuovi operatori, chiamati NOR e NAND. Tali operatori, indicati rispettivamente con i simboli $\downarrow$ e $|$, hanno le espressioni

$$x \downarrow y = \overline{x + y}$$

$$x|y = \overline{x \bullet y}$$

Le loro tavole della verità sono mostrate in figura 1.2.

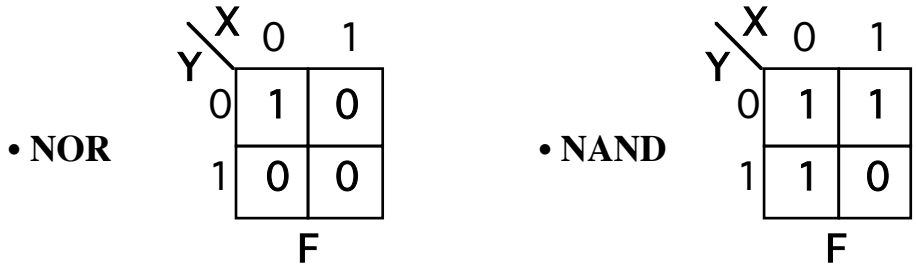


Figura 1.2. - Tavole della verità degli operatori NOR e NAND.

Osserviamo che, per le proprietà già viste,

$$x \downarrow x = \overline{x + x} = \bar{x}$$

$$x|x = \overline{x \bullet x} = \bar{x}$$

Entrambi gli operatori possono cioè essere utilizzati anche come negatori. Quindi, ognuno di essi costituisce da solo un insieme funzionalmente completo. Per questa ragione, essi vengono chiamati *operatori universali*: è possibile scrivere qualunque espressione logica utilizzando solo operatori NOR o solo operatori NAND.

1.1.4. L'OPERATORE XOR

Un altro operatore di uso molto frequente è il cosiddetto "OR esclusivo", o XOR, la cui tavola della verità è mostrata in figura 1.3.

• **XOR**

Y \ X	0	1
0	0	1
1	1	0

F

Figura 1.3. - Tavola della verità dell'operatore XOR.

L'operatore XOR si indica nelle espressioni mediante il simbolo $\oplus$.

1.1.5. ALGEBRA BOOLEANA E CIRCUITI ELETTRICI

Se si stabilisce una relazione tra grandezze elettriche e variabili booleane, è possibile costruire circuiti che implementano operatori logici: che si comportano cioè elettricamente in modo analogo a quello in cui si comportano matematicamente gli operatori logici elementari.
Un esempio di standard comunemente usato, che mette in relazione i livelli di tensione presenti nei diversi punti di un circuito elettrico con le grandezze logiche, è il seguente:

$$0 \le V \le 0.8 \Rightarrow \text{"0"}$$
$$3.2 \le V \le 5 \Rightarrow \text{"1"}$$

I valori di tensione (o, in alcuni casi, di corrente) sono determinati dal tipo di tecnologia impiegata per la costruzione dei circuiti elettronici. Per questa ragione, esistono molti standard diversi. Il principio di funzionamento è comunque lo stesso in ogni caso: nella nostra trattazione faremo sempre riferimento allo standard appena citato.
Possiamo a questo punto parlare di veri e propri "oggetti" (*porte logiche*) che possono essere collegati tra loro per realizzare funzioni combinatorie.
I simboli che si utilizzano per indicare le porte logiche corrispondenti agli operatori definiti finora sono mostrati in figura 1.4.

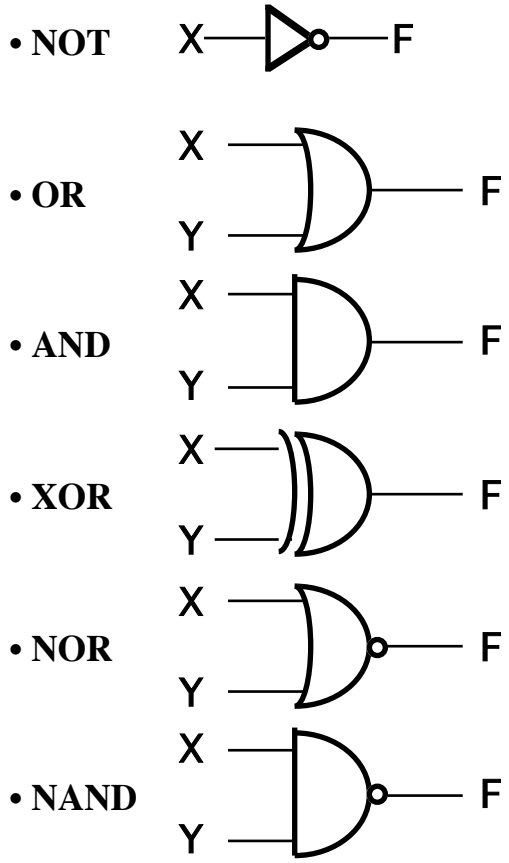


Figura 1.4. - Simboli da utilizzare per indicare gli operatori logici.

Valendosi della proprietà associativa, le porte logiche che implementano operatori definiti come binari possono anche essere costruite con più di due ingressi. Ad esempio, un porta NAND a quattro ingressi ha la tavola della verità mostrata in figura 1.5, ed è rappresentata dal circuito mostrato in figura 1.6.

X_1	X_2	X_3	X_4	f
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Figura 1.5. - Tavola della verità di una porta NAND a 4 ingressi.

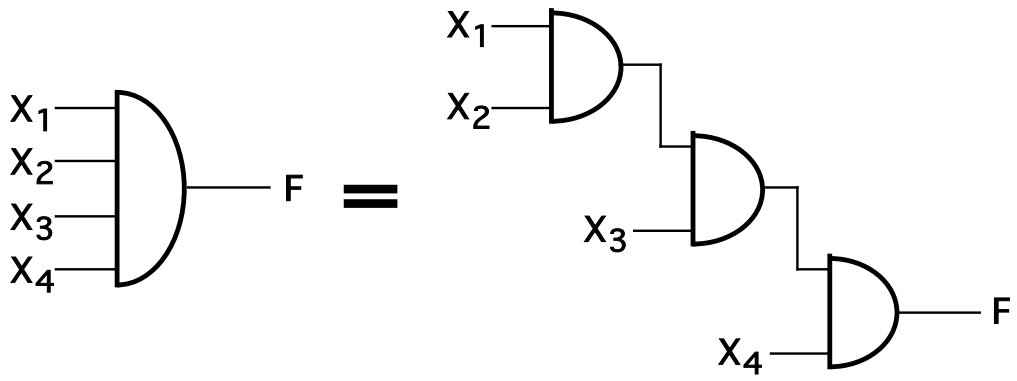


Figura 1.6. - Una porta AND a quattro ingressi.

Il passaggio dall'espressione di una funzione combinatoria al relativo circuito elettrico è immediato: è sufficiente infatti sostituire agli operatori dell'espressione le corrispondenti porte logiche, e collegare gli ingressi e le uscite in modo opportuno. Ad esempio, la funzione

$$y = ((a + b)\overline{cd}) + e$$

è realizzata dal circuito mostrato in figura 1.7.

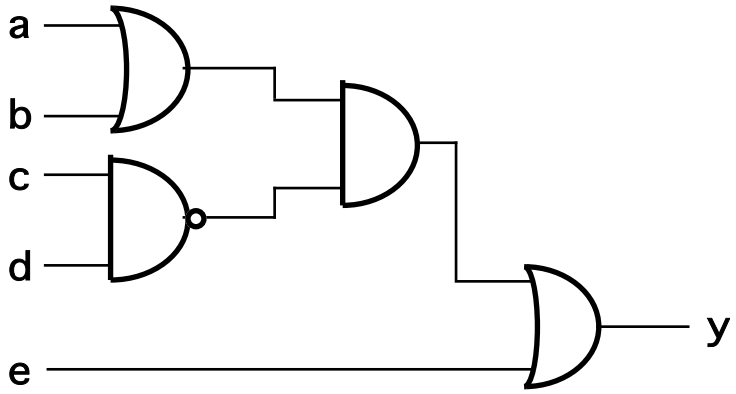


Figura 1.7 - Un circuito logico.

Il collegamento di porte logiche è sempre possibile, a patto di osservare un'importante regola:
In nessun caso è ammesso il collegamento fra le uscite di due o più porte logiche.
Questo per evitare che si creino conflitti fra porte che, per i loro ingressi, dovrebbero avere uscite diverse, come mostrato in figura 1.8.

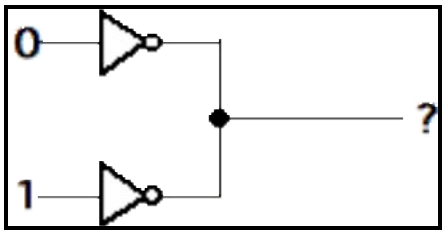


Figura 1.8. - Conflitto fra le uscite di due porte.

Al contrario non vi è nessuna limitazione teorica per quanto riguarda il numero di ingressi che possono essere collegati alla medesima uscita.

1.2. LE RETI COMBINATORIE

Prima di proseguire nella trattazione, è opportuno dare alcune definizioni:

- *rete logica (o semplicemente rete)*: circuito binario dotato di n ingressi e di una uscita;

- *analisi*: operazione consistente nello scrivere la funzione logica corrispondente ad una rete di cui è dato il circuito;
- *sintesi*: operazione opposta alla sintesi, con la quale, data una funzione logica, si progetta la rete che la realizza;
- *rete combinatoria*: circuito binario in cui in ogni istante l'unica uscita è funzione unicamente degli ingressi nel medesimo istante:

$$y(t) = f\left(x_1(t), x_2(t), \ldots, x_n(t)\right)$$

- *funzione mintermine*: funzione combinatoria nella cui tavola della verità compare un solo 1;
- *funzione maxtermine*: funzione combinatoria nella cui tavola della verità compare un solo 0.

Una funzione mintermine è costituita dal prodotto logico di tutte le variabili di ingresso, prese in forma naturale se nella configurazione di valori corrispondente all'unico 1 compaiono col valore 1, in forma complementata se nella stessa configurazione compaiono col valore 0.
La figura 1.9 mostra due funzioni mintermine delle variabili X_1, X_2, X_3 e le relative espressioni.
Una funzione maxtermine è costituita dalla somma logica di tutte le variabili di ingresso, prese in forma naturale se nella configurazione di valori corrispondente all'unico 0 compaiono col valore 0, in forma complementata se nella stessa configurazione compaiono col valore 1.
La figura 1.10 mostra una funzione maxtermine delle variabili X_1, X_2, X_3 e la relativa espressione.
Una funzione di commutazione può essere rappresentata mediante la somma dei suoi mintermini (*prima forma canonica*), come mostrato in figura 1.11.
Il numero dei mintermini da sommare è pari al numero di uni della funzione.
Dualmente, una funzione di commutazione è rappresentabile anche dal prodotto dei suoi maxtermini (*seconda forma canonica*), e in questo caso ha tanti termini quanti sono gli zeri della funzione.

X_1	X_2	X_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

$$f = \overline{X}_1 \bullet X_2 \bullet X_3$$

X_1	X_2	X_3	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

$$f = \overline{X}_1 \bullet \overline{X}_2 \bullet X_3$$

Figura 1.9. - Due funzioni mintermine e le relative espressioni.

X_1	X_2	X_3	f
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

$$f = X_1 + \overline{X}_2 + X_3$$

Figura 1.10. - Una funzione maxtermine e la relativa espressione.

X_1	X_2	X_3	f		f_1		f_2		f_3		f_4
0	0	0	0		0		0		0		0
0	0	1	1		1		0		0		0
0	1	0	1		0		1		0		0
0	1	1	0	=	0	+	0	+	0	+	0
1	0	0	0		0		0		0		0
1	0	1	1		0		0		1		0
1	1	0	0		0		0		0		0
1	1	1	1		0		0		0		1

$$f = f_1 + f_2 + f_3 + f_4 = \overline{X}_1 \bullet \overline{X}_2 \bullet X_3 + \overline{X}_1 \bullet X_2 \bullet \overline{X}_3 + X_1 \bullet \overline{X}_2 \bullet X_3 + X_1 \bullet X_2 \bullet X_3$$

Figura 1.11. - Una funzione combinatoria espressa come la somma dei suoi mintermini.

Sia la prima che la seconda forma canonica sono **uniche**, ma **non necessariamente minime**. Possono cioè esistere rappresentazioni della funzione che impiegano un minor numero di operatori, come sarà mostrato in seguito.

1.2.1. ANALISI DI UNA RETE COMBINATORIA

L'analisi di una rete è la descrizione del suo comportamento. Per una rete combinatoria, questo comportamento dipende in ogni istante esclusivamente dagli ingressi nello stesso istante. In questo caso quindi l'analisi coincide con l'indicazione del valore dell'uscita in corrispondenza di ogni possibile configurazione degli ingressi (tavola della verità).

Sia data la rete rappresentata in figura 1.12.

Il numero di possibili configurazioni in ingresso è: $N = 2^n$, dove n è il numero di ingressi (in questo caso $n = 3 \Rightarrow N = 8$).

La tavola della verità (Fig. 1.13) si ottiene imponendo agli ingressi tutte le possibili configurazioni e valutando di volta in volta l'uscita.

Nel caso di circuiti complessi si può procedere cercando di "spezzare" la rete in parti più piccole e analizzando ciascuna di esse separatamente dalle altre.

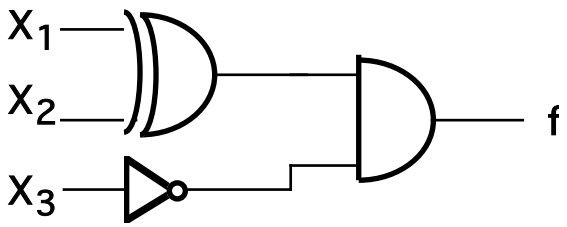


Figura 1.12. - Una rete combinatoria.

X_1	X_2	X_3	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Figura 1.13. - Tavola della verità relativa alla rete di figura 1.12.

1.2.2. SINTESI DI UNA RETE COMBINATORIA

Il primo passo nella sintesi è il verificare se la rete è realizzabile, se cioè il suo comportamento rispetta la definizione di rete combinatoria. Ad esempio, la realizzazione di un circuito contapersone mediante una rete combinatoria è chiaramente impossibile.
I passi successivi verranno illustrati mediante l'aiuto del seguente

> **problema**: costruire una rete logica che permetta di accendere e spegnere una lampada;
> - di notte la luce deve essere sempre accesa se in casa non c'è nessuno;
> - di giorno la luce deve essere sempre spenta se in casa non c'è nessuno.

Graficamente, il dispositivo può essere rappresentato come in figura 1.14, dove il punto interrogativo indica la rete combinatoria da progettare, e L la lampada da controllare.

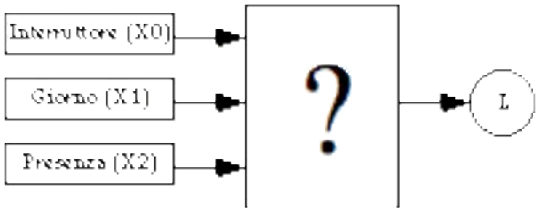


Figura 1.14. - Schematizzazione del problema proposto.

Esistono diverse metodologie per effettuare la sintesi in maniera rigorosa in modo da impiegare il minor numero possibile di componenti (rete minima): ci occuperemo qui del sistema noto col nome di *mappe di Karnaugh*.

1.2.3. LE MAPPE DI KARNAUGH

Le mappe di Karnaugh si basano su una particolare forma delle tavole della verità, ottenuta disponendo le possibili configurazioni di un numero n di variabili binarie ai vertici di un cubo ad n dimensioni (n-cubo o ipercubo), in modo tale che le configurazioni adiacenti (cioè quelle unite da uno spigolo) abbiano solamente un bit di differenza, come si può vedere in figura 1.15 per gli insiemi di una, due e tre variabili binarie.

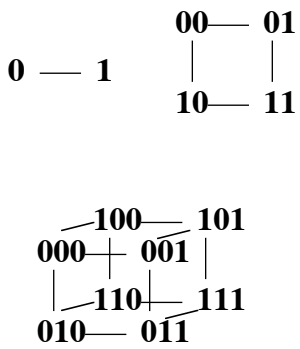


Figura 1.15. - Configurazioni di una, due e tre variabili binarie.

Per motivi di praticità, gli n-cubi vengono poi sviluppati su un piano, come mostrato in figura 1.16 per il caso a tre variabili.

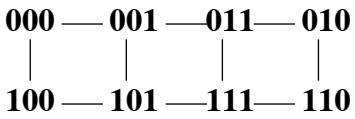


Figura 1.16. - Sviluppo di un *3*-cubo.

Si osservi che, dal momento che la mappa deriva dallo sviluppo di un cubo, i suoi elementi che si trovano su uno spigolo sono da considerarsi adiacenti anche agli elementi omologhi che si trovano sullo spigolo opposto. Nel caso di figura 1.17 l'elemento 000 è adiacente all'elemento 010, e l'elemento 100 è adiacente all'elemento 110.
È possibile in questo modo rappresentare senza grossi problemi funzioni fino a 6 variabili.
Tornando alla sintesi di una rete combinatoria, bisogna in primo luogo ridefinire in modo rigoroso la funzione da sintetizzare compilando la tavola della verità.
Con riferimento all'esempio della lampada possiamo scrivere la tabella di figura 1.17.

X_2	X_1	X_0	L
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Figura 1.17. - Tavola della verità per l'esempio della lampada.

Trascriviamo ora i valori della funzione in una mappa di Karnaugh (Fig. 1.18).
Si osservi che ogni casella contenente un 1 rappresenta un mintermine della funzione, e che ogni casella contenente uno 0 ne rappresenta un maxtermine.

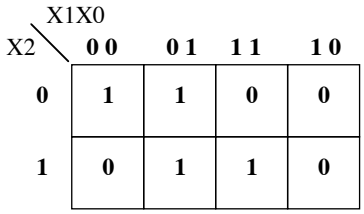


Figura 1.18. - Mappa di Karnaugh per l'esempio della lampada.

La sintesi di una rete minima consiste nel cercare sulla mappa i più grandi raggruppamenti di 1 adiacenti (*implicanti*) che costituiscano sottocubi aventi lati di dimensioni pari a una potenza di 2, e nello scegliere un insieme minimo di implicanti che copra tutti gli 1 della tabella.
Nel caso di una funzione di tre variabili si possono quindi avere implicanti di dimensioni:

$$1 \times 1 \quad 1 \times 2 \quad 1 \times 4$$
$$2 \times 1 \quad 2 \times 2$$

Nel caso in esame esistono tre raggruppamenti di dimensioni maggiori di 1x1, indicati in figura 1.19 con le lettere *a*, *b* e *c*.

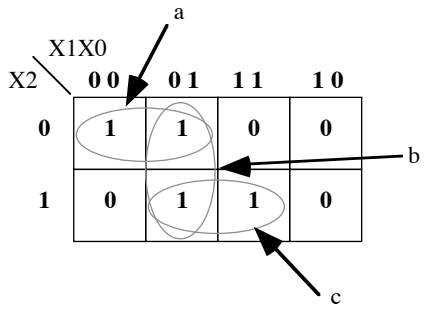


Figura 1.19. - Implicanti possibili.

In questo caso per coprire completamente la tabella sono sufficienti gli implicanti *a* e *c*. Per ognuno di essi si deve ricavare la funzione che lo rappresenta, che è costituita dal prodotto logico di tutte le variabili che hanno lo stesso valore per ogni elemento dell'implicante, prese nella loro forma naturale se compaiono con il valore 1, in quella complementata se compaiono con il valore 0. Nel caso in esame:

Implicante *a*: $\overline{X}_1 \bullet \overline{X}_2$

Implicante *b* (non utilizzato): $\overline{X}_1 \bullet X_0$

Implicante *c*: $X_0 \bullet X_2$

La funzione minima che risolve il problema proposto è data dalla somma degli implicanti *a* e *c*:

$$L = \overline{X}_1 \bullet \overline{X}_2 + X_0 \bullet X_2$$

Il circuito corrispondente è mostrato in figura 1.20.

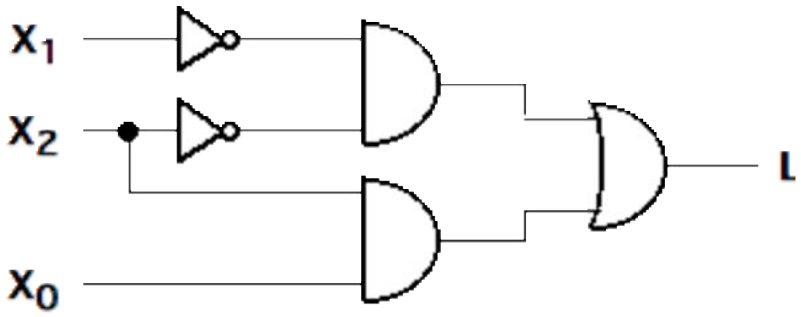


Figura 1.20. - Il circuito completo.

La rete così ottenuta è minima: in questo caso essa è unica, ma in generale possono esistere più soluzioni, tutte minime.
La sintesi mostrata produce una rete *a due livelli* costituita dalla somma di un certo numero di termini prodotto: il metodo è anche utilizzabile dualmente per ottenere reti in forma di prodotto di somme, utilizzando i maxtermini (cioè gli zeri della tabella), e modificando di conseguenza le regole per la sintesi.[4]

1.3. I CIRCUITI INTEGRATI

Alcune funzioni combinatorie sono di impiego molto frequente: di qui l'opportunità di produrre circuiti che le realizzano, in modo da scaricare i progettisti dal compito di dover sintetizzare ogni volta tutte le reti occorrenti.
Le reti in questione vengono realizzate sotto forma di *circuiti integrati*, cioè costruiti su un unico chip semiconduttore. In questo paragrafo saranno trattati solo i circuiti integrati utili per lo sviluppo della trattazione.

[4] Una trattazione più completa dell'argomento delle mappe di Karnaugh, comprendente la sintesi in forma prodotto di somme, il fenomeno delle cosiddette "alee statiche", e la sintesi di reti incompletamente specificate esula dagli scopi di questo corso. Essa può comunque essere reperita nella bibliografia indicata al termine del volume.

1.3.1. PORTE MULTIPLE

Si tratta di insiemi di più reti combinatorie integrate. Alcuni esempi di circuiti integrati di questo tipo sono:

- 6 porte NOT;
- 4 porte NAND a due ingressi;
- 4 porte NAND a tre ingressi;
- 4 porte OR a due ingressi;
- 1 porta NAND a otto ingressi.

1.3.2. DECODIFICATORI

Si tratta di insiemi di reti combinatorie aventi un certo numero n di ingressi e 2^n uscite, di cui una sola è attiva volta per volta (tutte le altre sono a 0). Per ragioni costruttive (il numero di piedini di un circuito è forzatamente limitato), n è in genere non superiore a 4. È opportuno mettere in evidenza che un decodificatore non è una singola rete combinatoria, ma l'insieme di un certo numero di reti combinatorie aventi alcuni componenti in comune.
La figura 1.21 mostra la tavola della verità di un decodificatore a tre ingressi e quindi a otto uscite.
Il circuito corrispondente è mostrato in figura 1.22.
Per comprendere lo schema è sufficiente osservare che ogni uscita del decodificatore rappresenta uno dei mintermini delle variabili di ingresso.

I_2	I_1	I_0	O_0	O_1	O_2	O_3	O_4	O_5	O_6	O_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Figura 1.21. - Tavola della verità di un decodificatore a 3 ingressi.

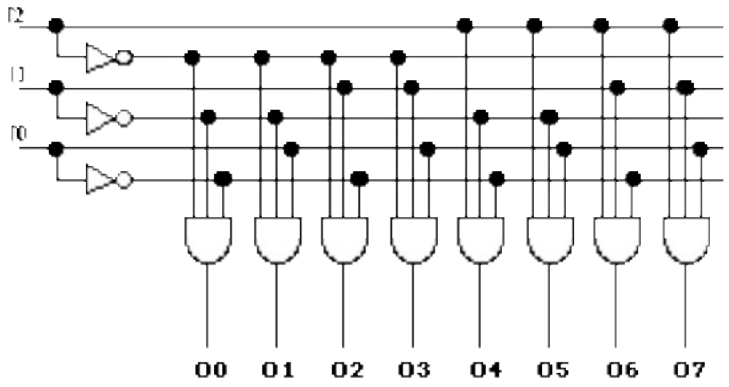


Figura 1.22. - Schema di un decodificatore a tre ingressi.

1.3.3. ROM, PROM, EPROM

Si tratta di insiemi di reti combinatorie usati in svariate applicazioni, in particolare nel campo dei calcolatori.

1.3.3.1. ROM (*Read Only Memory*)

Il dispositivo presenta un numero n di ingressi e un numero k di uscite. Non esiste nessuna relazione tra il numero di ingressi e il numero di uscite.
La ROM è assimilabile ad una tabella avente 2^n righe e k colonne. Ogni elemento della tabella contiene un 1 o uno 0. Sugli ingressi viene presentata la configurazione corrispondente al numero della riga (indirizzo) della tabella di cui si vuole conoscere il contenuto. In ogni istante, sulle uscite è presente il contenuto degli elementi della riga indirizzata.
La *capacità* di una ROM indica le dimensioni della tabella che essa contiene, ed equivale al prodotto del numero di parole per il numero delle uscite, cioè a $2^n \cdot k$ bit. Ad esempio, una ROM da 1024 parole di 8 bit contiene 8192 bit.
La ROM è costituita da due blocchi fondamentali (Fig. 1.23): un decodificatore (o *sezione AND*) e la cosiddetta *sezione OR*.

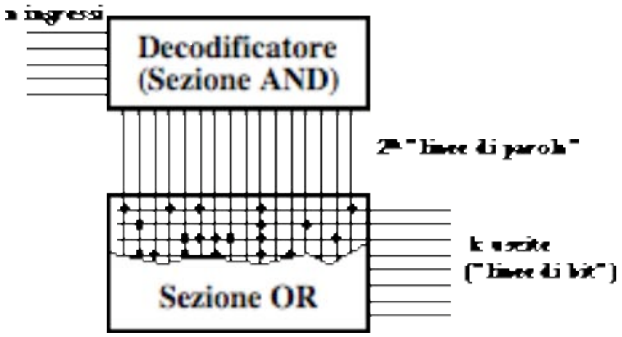


Figura 1.23. - Schema di una ROM.

Il decodificatore è del tutto analogo a quelli visti precedentemente. Dal momento che le sue uscite non devono essere portate all'esterno del dispositivo, come accadeva per i decodificatori visti in precedenza, viene a cadere il vincolo imposto dal numero dei piedini del circuito. È quindi possibile costruire ROM aventi un decodificatore enormemente più grande di quelli visti prima. La sezione OR è costituita da tante porte OR quante sono le linee di bit. Ogni OR ha un ingresso per ogni linea di parola per la quale il corrispondente bit in uscita deve valere 1. Ad esempio, nella ROM di figura 1.23, la prima uscita (quella più in alto) è costituita da un OR a 5 ingressi, collegati rispettivamente alle linee di parola 0, 3, 5, 9, 15.
La configurazione della sezione OR in una ROM è fissa, e viene stabilita in fase di fabbricazione del dispositivo.

1.3.3.2. PROM: *Programmable ROM*

Si tratta di una ROM in cui, all'origine, la sezione OR è completa (ogni linea di parola è collegata a tutte le porte OR). Con opportuni accorgimenti è possibile "bruciare" i collegamenti non desiderati, ottenendo in tal modo la configurazione della tabella che si vuole avere. I collegamenti bruciati non possono più essere ripristinati: una PROM quindi può essere programmata una sola volta.

1.3.3.3. EPROM: *Erasable PROM*

Le EPROM sono PROM in cui, con opportuni accorgimenti (raggi ultravioletti o correnti elettriche) i collegamenti eliminati possono essere ripristinati in blocco, riportando così il dispositivo allo stato originale e rendendolo pronto per una nuova configurazione.
È importante ricordare che, nonostante il nome, ROM, PROM ed EPROM rispondono alla definizione di reti combinatorie, perché in ogni istante le loro uscite sono funzione esclusivamente dei valori degli ingressi.

1.3.4. LE PORTE *TRI-STATE*

Mentre le porte normali hanno due possibili stati di uscita (lo stato 0 e lo stato 1), le porte tri-state hanno tre stati possibili di uscita: 0, 1 e *Hi-Z* . Esse possono essere schematizzate come in figura 1.24 interponendo un interruttore controllabile mediante un segnale logico all'uscita di una normale porta logica. Se l'interruttore è chiuso l'uscita è quella della porta; se invece l'interruttore è aperto l'uscita è virtualmente sconnessa dal circuito (stato *Hi-Z*, o di alta impedenza).

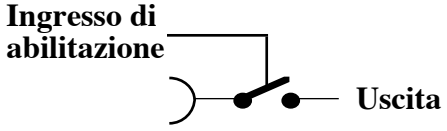


Figura 1.24. - Il concetto di porta tri-state.

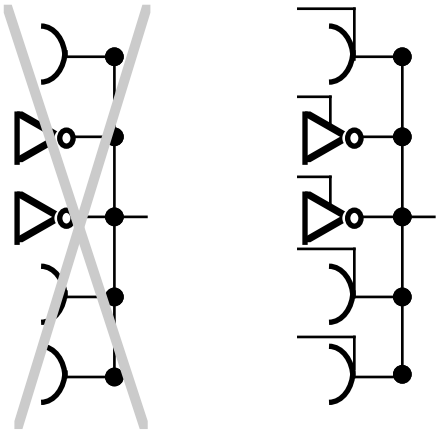


Figura 1.25. - Collegamenti di porte tri-state.

Il fondamentale pregio delle porte tri-state è che esse permettono di "violare" la regola, precedentemente esposta, che vieta di collegare insieme uscite di porte diverse. Le uscite delle porte tri-state possono essere invece collegate insieme (Fig. 1.25), a patto che in qualunque momento una sola di esse sia in uno stato diverso da Hi-Z. In questo caso, è come se al circuito fosse collegata una sola porta.
Un esempio di componente di questo tipo usato molto spesso è la ROM tri-state, il cui schema è mostrato in figura 1.26.
La ROM è analoga a quella vista precedentemente, tranne per il fatto che le porte OR di uscita sono ora porte tri-state. Il segnale /OE attiva le uscite: quando è al valore 0, il circuito si comporta come una ROM normale. Quando invece /OE è al livello 1 l'uscita si trova in uno stato di alta impedenza, e il dispositivo non influisce in alcun modo sul circuito a cui è collegato.

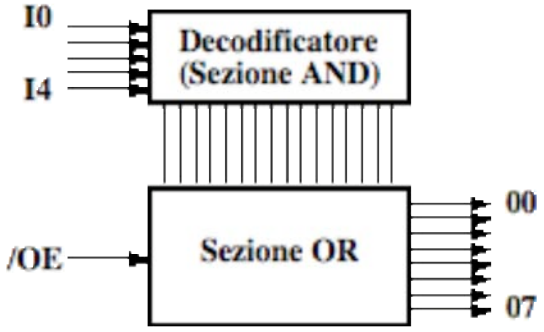


Figura 1.26. - Una ROM con uscite tri-state.

1.4. RETI SEQUENZIALI (MACCHINE SEQUENZIALI)

Sono reti con *n* ingressi e una sola uscita che dipende dal valore degli ingressi e dalla storia passata.
Quindi lo stato dell'uscita non è solamente funzione degli ingressi in un determinato istante, ma dipende anche dai valori che gli ingressi hanno assunto negli istanti precedenti.
Ad esempio, il semplice meccanismo di accensione di una lampada tramite un relais (una pressione sul pulsante accende la lampada, un'altra la spegne, e così via) è una macchina sequenziale, il cui comportamento è quello schematizzato in figura 1.27.

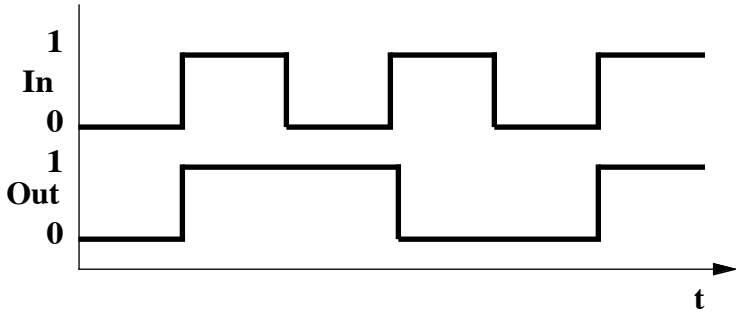


Figura 1.27. - Comportamento di una semplice macchina sequenziale.

Per descrivere compiutamente il comportamento di una macchina sequenziale si utilizza il concetto di *stato*, definito come l'insieme di tutte le variabili d'ingresso, dell'uscita e dello stato precedente.
Per chiarire il concetto, utilizziamo come esempio una semplice macchina con un ingresso e una uscita, descritta nella parte superiore di figura 1.28.
Ad ogni transizione dell'ingresso la macchina si porta in un nuovo stato, secondo il diagramma mostrato nella parte inferiore di figura 1.28. Per ogni stato è indicata l'uscita (all'interno del circoletto), e il nuovo stato in cui la macchina si porta in funzione del valore degli ingressi (indicati sugli archi che collegano gli stati).
Si osservi che, benché il secondo e il quarto stato appaiano uguali (stessi ingressi e stesse uscite), non lo sono, infatti rappresentano due situazioni diverse fra loro. Lo stesso vale per il terzo e il quinto stato.
Una trattazione delle macchine sequenziali esula dagli scopi di questo corso. Saranno qui descritte solo le funzioni di quelle macchine sequenziali, la cui conoscenza è indispensabile per la comprensione del resto del capitolo.

1.4.1. FLIP-FLOP DI TIPO DT (*LATCH*)

È una macchina sequenziale con due ingressi e una sola uscita, e viene chiamato *latch* (dall'inglese *to latch*, bloccare) perché è in grado di memorizzare indefinitamente lo stato di una linea in un certo istante.

La figura 1.29 mostra le connessioni e il comportamento di un flip-flop di tipo DT. Quando l'ingresso T è a 1, l'uscita segue l'andamento dell'ingresso D. I primi valori di Q (fino a quando T non va a 1) non sono definiti perché nel momento in cui il dispositivo comincia a funzionare il livello dell'uscita non è noto. Quando l'ingresso T va a 0, il valore dell'ingresso D in quell'istante viene "bloccato" sull'uscita Q, e vi permane finché T rimane a 0.

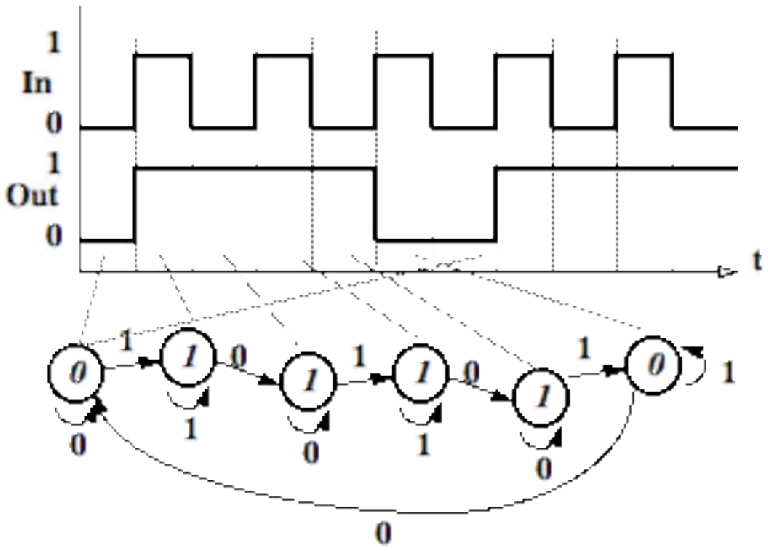


Figura 1.28. - Una macchina sequenziale e i suoi stati.

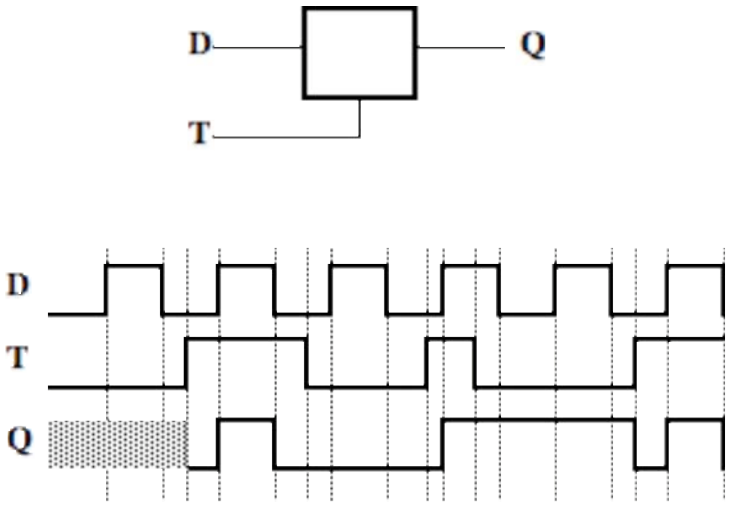


Figura 1.29. - Un flip-flop di tipo DT.

1.4.2. PORTA UNIVERSALE (*LATCH 3-STATE*)

È costituita da 8 flip-flop DT, ognuno seguito da una porta tri-state. I suoi collegamenti sono mostrati in figura 1.30.

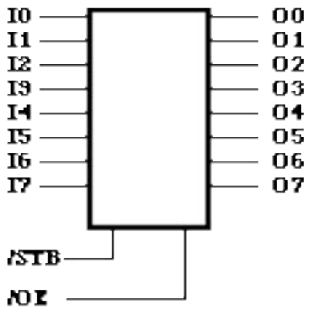


Figura 1.30. - Una porta universale.

Gli ingressi I0-I7 sono collegati agli ingressi D dei rispettivi flip-flop. Quando il segnale /OE, collegato alle abilitazioni di tutte le porte di uscita, diventa basso, sui piedini O1-O7 appare il valore delle uscite dei rispettivi flip-flop. Il piedino /STB (attivo basso) è collegato agli ingressi T di tutti i flip-flop, e serve per indicare l'istante in cui si vuole memorizzare lo stato degli ingressi.
Questa porta è fondamentale per collegare il calcolatore con il mondo esterno; è detta universale perché può funzionare sia in ingresso sia in uscita, come sarà mostrato in seguito.

1.4.3. LA RAM (*RANDOM ACCESS MEMORY*)

La RAM (Fig. 1.31) serve per memorizzare dati e programmi all'interno del calcolatore.
Il funzionamento è simile a quello di una ROM, con l'aggiunta di alcune linee che rendono possibile sia la lettura che la scrittura dei dati. La RAM infatti funziona in due modi diversi a seconda che si voglia leggere o scrivere.
La figura 1.32 mostra le sequenze di segnali durante la lettura di un dato. Si noti che l'ingresso /CE (Chip Enable) serve per abilitare il funzionamento del dispositivo, nel senso che se /CE è alto la memoria non può essere né scritta né letta: mantiene solo le informazioni al suo interno.

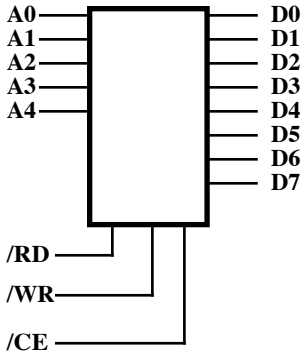


Figura 1.31. - Collegamenti di una RAM.

Le fasi per la lettura di un dato sono:

1. fornire l'indirizzo della parola che si vuole leggere;
2. attivare la memoria mettendo allo stato basso /CE;
3. portare l'ingresso /RD allo stato basso (l'ingresso /WR dovrà essere alto);
4. a questo punto sulle uscite D0-D7 è disponibile il dato;
5. una volta prelevato il dato si può riportare alta la linea /RD, disattivare gli indirizzi e rialzare /CE.

Queste operazioni non possono essere fatte tutte contemporaneamente a causa del cosiddetto *tempo di accesso* della memoria, cioè dell'intervallo di tempo che trascorre tra l'istante in cui si fornisce l'indirizzo e il momento in cui il dato è disponibile.

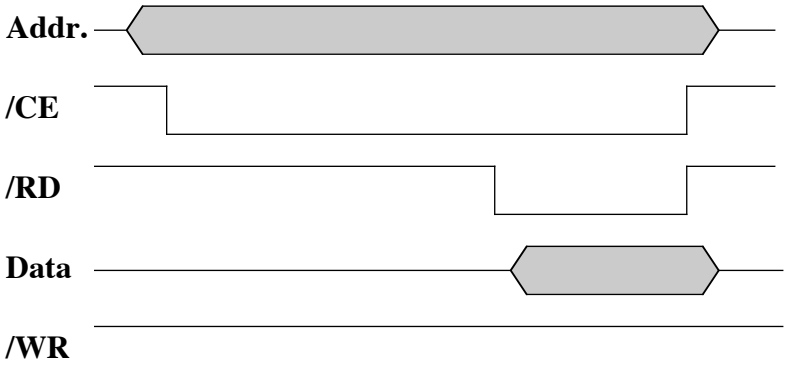


Figura 1.32. - Temporizzazione della lettura di una RAM.

Nel caso in cui si voglia scrivere (Fig. 1.33), il flusso dei dati è inverso: ciò vuol dire che ora i piedini D0-D7 funzionano come ingressi, mentre prima funzionavano come uscite.

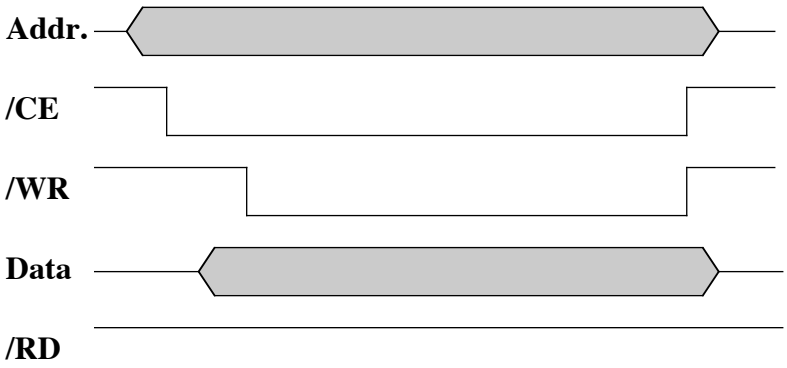


Figura 1.33. - Temporizzazione della scrittura di una RAM.

Per quanto riguarda l'indirizzamento, il procedimento è analogo alla lettura; la scrittura effettiva avviene sul fronte di salita del segnale /WR.

1.5. STRUTTURA DI UN MICROCALCOLATORE

Lo schema a blocchi di un calcolatore visto nel corso di Fondamenti di Informatica I è quello mostrato in figura 1.34.
Lo scopo dell'analisi che sarà ora condotta è di definire il funzionamento del bus e di comprendere il meccanismo dello scambio di informazione fra i moduli.

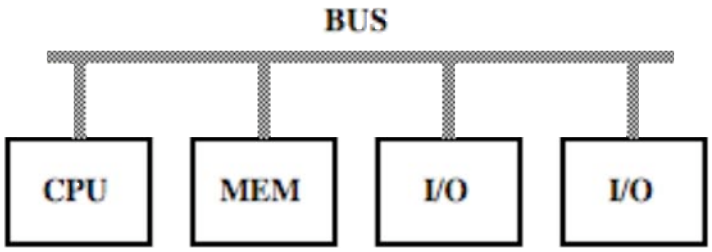


Figura 1.34. - Schema a blocchi di un calcolatore.

1.5.1. I CICLI

La maggior parte delle interazioni che l'unità centrale ha con gli altri componenti consiste nell'esecuzione di cicli di lettura (per acquisire dati dalla memoria o da periferiche) e di scrittura (per memorizzare dati o per mandarli a periferiche). Tali cicli possono essere così descritti a parole:

- Ciclo di lettura (il microcalcolatore acquisisce un dato dalla periferia):
  1. il microcalcolatore manda l'indirizzo;

2. il microcalcolatore segnala che l'indirizzo è valido;
3. il microcalcolatore segnala che è pronto a leggere il dato;
4. il dispositivo indirizzato manda il dato;
5. il microcalcolatore legge il dato.

- Ciclo di scrittura (il microcalcolatore manda un dato verso la periferia):
  1. il microcalcolatore manda l'indirizzo;
  2. il microcalcolatore segnala che l'indirizzo è valido;
  3. il microcalcolatore manda il dato;
  4. il microcalcolatore segnala che il dato è valido;
  5. la periferia acquisisce il dato.

Tutti i dati e i segnali necessari a questi scambi di informazione viaggiano sul bus; dal momento però che per ogni tipo di informazione vengono usate linee diverse, è opportuno suddividere il bus in tre parti distinte, come mostrato in figura 1.35.

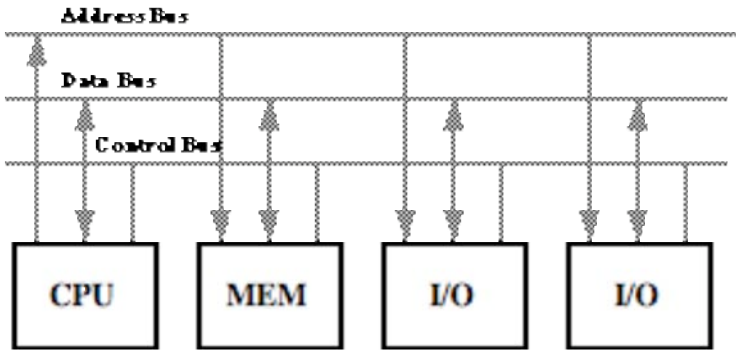


Figura 1.35. - Suddivisione del bus in tre parti.

Il significato dei tre bus così individuati è il seguente:

- Bus indirizzi (*Address bus*):
  - raggruppa le linee che specificano l'indirizzo su cui operare;
  - il numero di linee determina la capacità di indirizzamento del microcalcolatore;
- Bus dati (*Data bus*):
  - raggruppa le linee su cui viaggiano i dati;
  - il numero di linee determina la lunghezza di parola del microcalcolatore;
- Bus di controllo (*Control bus*):
  - raggruppa tutte le altre linee comuni.

È importante osservare che i bus raggiungono tutti i componenti del sistema (CPU, memoria, periferiche), ma che non necessariamente tutti i componenti sono collegati a tutte le linee dei bus.

Consideriamo l'address bus *monodirezionale*, nel senso che è pilotato sempre e solo dalla CPU.
Il data bus viene invece pilotato dalla periferia o dalla CPU a seconda che si debba leggere o scrivere. Si tratta quindi di un bus *bidirezionale* (percorso in un senso o nell'altro, mai in entrambi i sensi contemporaneamente).
Esaminiamo ora la temporizzazione dei cicli di lettura e di scrittura. Come si può vedere, essi sono del tutto compatibili con quelli a suo tempo visti per le RAM.
Ciclo di lettura (Fig. 1.36):
/MREQ (memory request: appartiene al control bus) segnala lo stato del bus indirizzi. Quando è basso l'indirizzo è valido. I nomi degli altri segnali corrispondono a quelli già visti per le RAM.

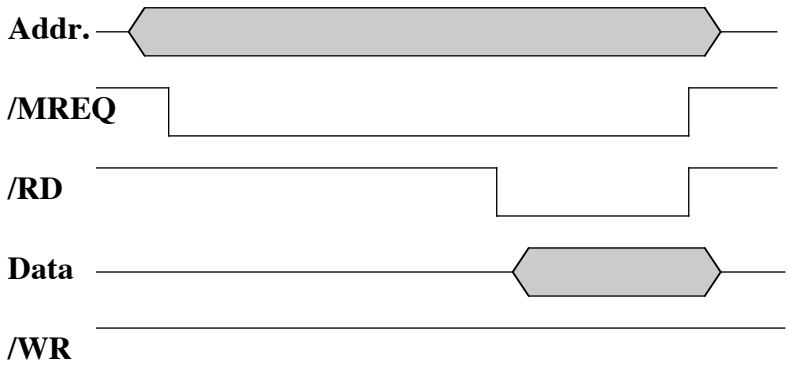


Figura 1.36. - Ciclo di lettura.

Ciclo di scrittura (Fig. 1.37):
l'informazione va nel senso contrario. È la CPU che spedisce i dati fin dall'inizio dell'operazione. /WR è attivo mentre /RD è inattivo.

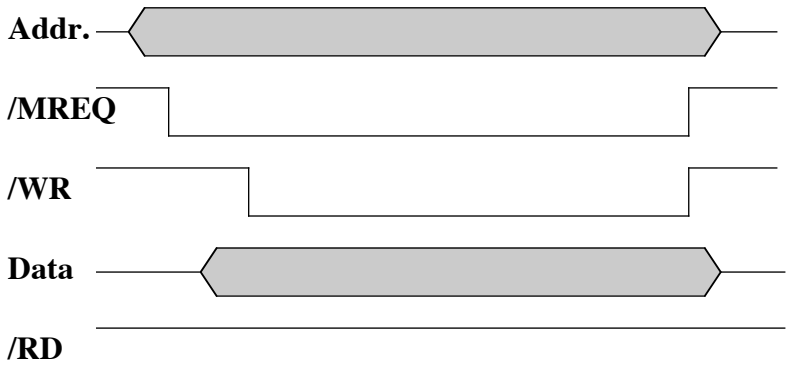


Figura 1.37. - Ciclo di scrittura.

1.5.2. SELEZIONE DEI BLOCCHI DI MEMORIA

Si presenta molto spesso il seguente problema: come collegare più dispositivi al bus? Mostreremo la soluzione con il seguente esempio: si supponga di voler costruire una memoria da 4 Kbyte, utilizzando memorie da 1 Kbyte, e di volerla collegare ad una CPU capace di indirizzare 64 Kbyte. Useremo ovviamente quattro componenti uguali, ed avremo bisogno di un selettore dei moduli di memoria che "colleghi" di volta in volta solamente il modulo giusto, scollegando al contempo tutti gli altri.
Una memoria da 1 Kbyte ha 8 piedini per i dati e 10 piedini per l'indirizzo. Il bus indirizzi del calcolatore indicato ha invece 16 linee (2^{16} = 64K). Gli 8 piedini per i dati di ogni memoria vanno collegati alle 8 linee del data bus. Vediamo ora come vanno collegati i 10 piedini per l'indirizzo.
Vogliamo che la memoria risponda agli indirizzi compresi fra 0 e 4095_{10}, così ripartiti fra i componenti:

Componente	Primo indirizzo	Ultimo indirizzo
Primo	0	1023_{10}
Secondo	1024_{10}	2047_{10}
Terzo	2048_{10}	3071_{10}
Quarto	3072_{10}	4095_{10}

Non ha senso che a una cella sia associato più di un indirizzo. L'indirizzo 3250 deve attivare solo il quarto componente; l'indirizzo 5550 non deve attivare nessun componente, perché non rientra in nessuno dei gruppi di indirizzi definiti nella tabella.
Traducendo in binario la tabella precedente si ha:

A15	**14**	**13**	**12**	**11**	**10**	**09**	**08**	**07**	**06**	**05**	**04**	**03**	**02**	**01**	**00**	
0	**0**	**0**	**0**	**0**	**0**	0	0	0	0	0	0	0	0	0	0	} **RAM 0**
0	**0**	**0**	**0**	**0**	**0**	1	1	1	1	1	1	1	1	1	1	
0	**0**	**0**	**0**	**0**	**1**	0	0	0	0	0	0	0	0	0	0	} **RAM 1**
0	**0**	**0**	**0**	**0**	**1**	1	1	1	1	1	1	1	1	1	1	
0	**0**	**0**	**0**	**1**	**0**	0	0	0	0	0	0	0	0	0	0	} **RAM 2**
0	**0**	**0**	**0**	**1**	**1**	1	1	1	1	1	1	1	1	1	1	
0	**0**	**0**	**0**	**1**	**1**	0	0	0	0	0	0	0	0	0	0	} **RAM 3**
0	**0**	**0**	**0**	**1**	**1**	1	1	1	1	1	1	1	1	1	1	

Ogni memoria è collegata alle 10 linee meno significative dell'address bus. Tutte le altre linee dello stesso bus, più /MREQ, opportunamente combinate, vanno ad un circuito di selezione il cui scopo è quello di abilitare solo uno dei componenti per volta.
I numeri della tabella indicati in carattere tondo si ripetono sempre uguali per ogni componente: queste sono le 10 linee del bus indirizzi che vanno collegate direttamente alle memorie. I numeri in grassetto invece permettono di individuare il gruppo da abilitare: ad esempio, tutti e soli i numeri che appartengono al primo intervallo incominciano con sei zeri[5]. Si può quindi costruire una rete combinatoria

[5] Quindi non servono per discriminare le celle del primo modulo, ma per discriminare il primo modulo dagli altri.

con sette ingressi, la cui uscita valga 0 se e solo se tutti gli ingressi sono 0. Gli ingressi di questa rete sono collegati a /MREQ, A15, A14, …, A10. L'uscita di questa rete, collegata al piedino /CE del primo blocco di memoria, lo abilita solo quando l'indirizzo è valido, ed è compreso fra 0 e 1023_{10}. Si procede analogamente per gli altri blocchi di memoria: le funzioni per abilitarli sono rispettivamente:

$$CE_0 = \overline{\overline{MREQ} + \overline{A_{15}} + \overline{A_{14}} + \overline{A_{13}} + \overline{A_{12}} + \overline{A_{11}} + \overline{A_{10}}}$$

$$CE_1 = \overline{\overline{MREQ} + \overline{A_{15}} + \overline{A_{14}} + \overline{A_{13}} + \overline{A_{12}} + \overline{A_{11}} + A_{10}}$$

$$CE_2 = \overline{\overline{MREQ} + \overline{A_{15}} + \overline{A_{14}} + \overline{A_{13}} + \overline{A_{12}} + A_{11} + \overline{A_{10}}}$$

$$CE_3 = \overline{\overline{MREQ} + \overline{A_{15}} + \overline{A_{14}} + \overline{A_{13}} + \overline{A_{12}} + A_{11} + A_{10}}$$

Lo schema dei collegamenti è quello indicato in figura 1.38.

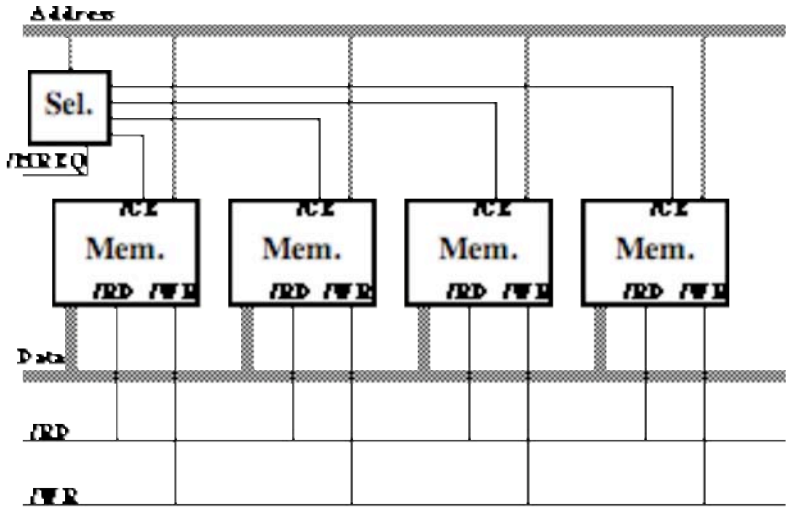


Figura 1.38. - Collegamento della memoria.

1.5.3. PORTE DI USCITA

Si tratta di dispositivi che presentano al mondo esterno almeno un piedino su cui possono comparire zeri e uni sotto controllo di un programma, e sono costruite utilizzando lo schema base mostrato in figura 1.39.[6]

[6] In questo e nel successivo paragrafo sono descritte solo le cosiddette "porte parallele". Gli altri tipi di porte seguono comunque lo stesso schema di base, con l'aggiunta di altri componenti.

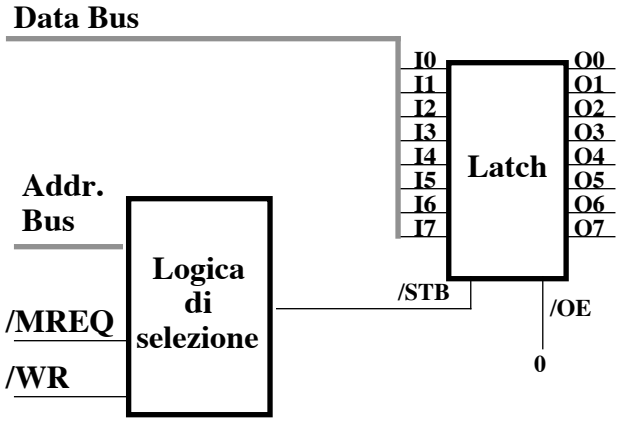


Figura 1.39. - Collegamento di una porta di uscita.

La scrittura su una cella di memoria o su una porta di uscita segue le stesse temporizzazioni dei segnali. Il dato viene mantenuto sui piedini di uscita dal latch fino a che non ne viene scritto un altro. La logica di selezione è fatta in modo da rispondere a un solo indirizzo, secondo i criteri già visti per le memorie.

1.5.4. PORTE DI INGRESSO

Sono analoghe alle precedenti, ma il latch è collegato al contrario, dal momento che l'informazione viaggia in senso opposto (Fig. 1.40).

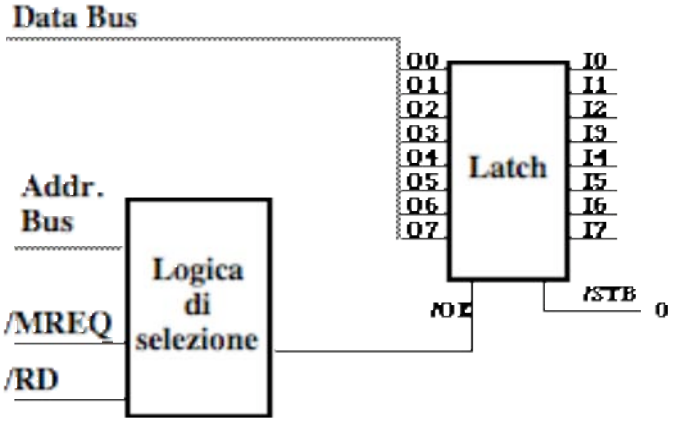


Figura 1.40. - Collegamento di una porta di ingresso.

1.5.5. LE FASI DI ESECUZIONE DI UN'ISTRUZIONE

È opportuno ricordare che l'esecuzione di una istruzione comporta sempre due fasi distinte:

- fase di *fetch* (ricerca): lettura (dalla memoria) dell'istruzione da eseguire
- fase di esecuzione: esecuzione effettiva dell'istruzione

In molti calcolatori, le istruzioni sono costituite da più di una parola, il che implica che la fase di fetch possa comprendere più cicli di lettura. A sua volta, la fase di esecuzione può comprendere uno o più cicli di lettura e/o scrittura su memoria e/o periferiche.

1.5.6. IL COLLEGAMENTO DELLE PERIFERICHE

Il collegamento delle periferiche comporta il rispetto delle seguenti condizioni:

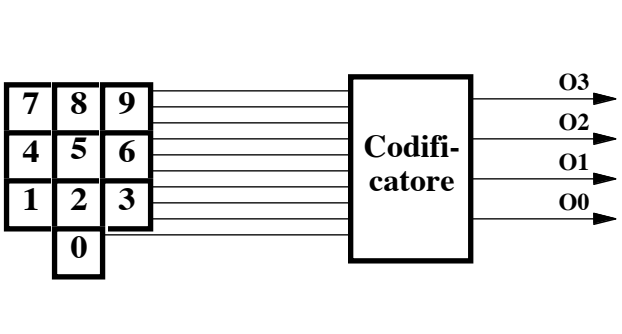
- le informazioni fornite dalle periferiche di ingresso sono significative solo in determinati istanti;
- le informazioni fornite dalle periferiche di ingresso sono significative solo per un determinato tempo;
- le informazioni possono essere fornite alle periferiche di uscita solo in determinati istanti;
- alcune periferiche di uscita **devono** ricevere le informazioni entro un determinato tempo.

Ciò implica la necessità di evitare:

- di acquisire dati che non sono significativi;
- di spedire dati a periferiche che non riescono a gestirli.

1.5.6.1. Un esempio di collegamento

Consideriamo a titolo di esempio una semplice periferica di ingresso, costituita da una tastiera numerica a 10 tasti, seguita da un codificatore che fornisce in uscita il codice binario del tasto premuto, secondo quanto indicato in figura 1.41.



T	O3	O2	O1	O0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
-	0	0	0	0

Figura 1.41. - Una tastiera numerica.

Se non si preme alcun tasto, all'uscita del codificatore comparirà una configurazione costituita da quattro zeri. Tale configurazione è però identica a quella che si ottiene in uscita premendo il tasto "0". Occorre quindi un circuito che permetta di sapere se la configurazione di bit in uscita è significativa, se cioè in un determinato istante uno dei tasti è premuto oppure no. Per far questo, trasformiamo la rete di figura 1.41 in quella di figura 1.42.
Se non si preme alcun tasto, l'uscita /STB ha il valore 1; se viene premuto un tasto qualsiasi, l'uscita /STB ha il valore 0.
La rete è ora completa, in quanto /STB indica se è stato premuto qualche tasto, mentre le altre uscite indicano qual è il tasto premuto.
Vediamo ora come il dispositivo appena illustrato possa essere collegato al calcolatore. Utilizziamo una porta universale, collegata come in figura 1.43, per acquisire al momento opportuno il codice del tasto premuto, e per trasmetterlo alla CPU quando quest'ultima esegue un ciclo di lettura sulla porta in questione.
Premendo un tasto:

- /STB diventa basso;
- il latch fa passare i valori degli ingressi in uscita.

Togliendo il dito dal tasto:

- /STB diventa alto;
- la configurazione viene memorizzata nei flip-flop del latch.

Ora il calcolatore può leggere in qualunque istante la configurazione memorizzata. Tuttavia, per il modo in cui è stato costruito il dispositivo, la CPU non ha modo di sapere se è stato premuto un tasto oppure no. Questo problema può essere risolto in due modi diversi, che saranno esaminati nel prossimo paragrafo.

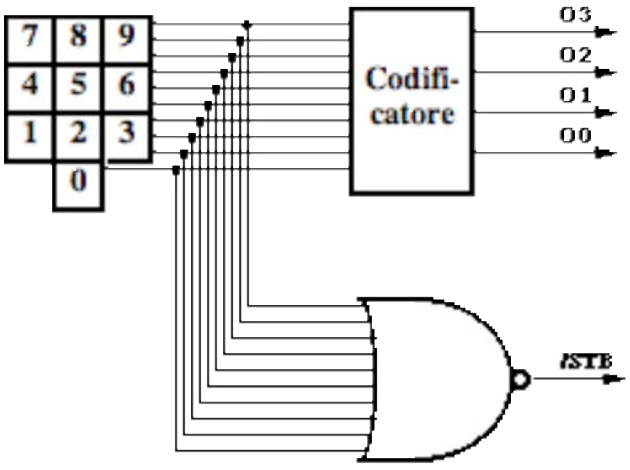


Figura 1.42. - Circuito modificato della tastiera numerica.

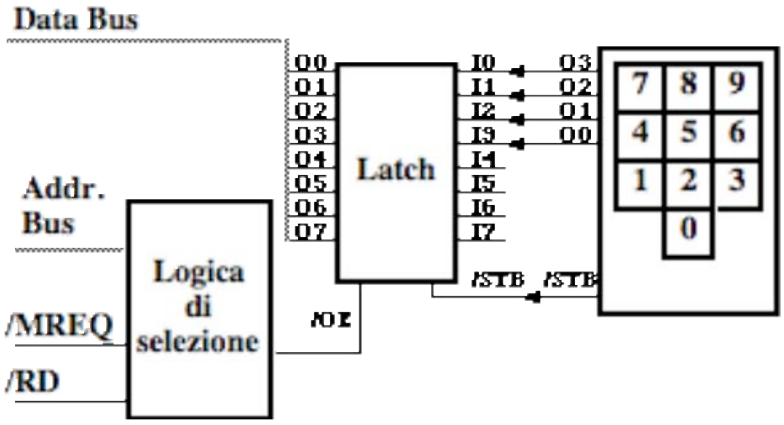


Figura 1.43. - Collegamento della tastiera al calcolatore.

1.5.7. LA GESTIONE DELLE PERIFERICHE

Una volta definito il collegamento elettrico delle periferiche con il calcolatore, descriviamo due diversi metodi per la loro gestione, cioè per governare il trasferimento dei dati da o verso le periferiche stesse.

1.5.7.1. Gestione a controllo di programma

La prima modalità, chiamata gestione *a controllo di programma*, implica che la linea /STB proveniente dalla periferica sia collegata anche a una porta di ingresso del calcolatore.[7]

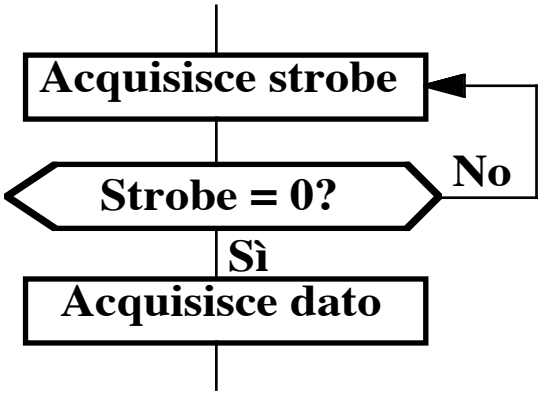


Figura 1.44. - Gestione a controllo di programma.

In questo caso (Fig. 1.44) la CPU è in grado di sapere se è stato premuto un tasto acquisendo ed analizzando lo stato della linea /STB.

---

[7] Diversa da quella a cui sono collegate le linee provenienti dal codificatore.

Questa modalità, per quanto semplice, presenta diversi svantaggi dovuti al fatto che la CPU, mentre attende che una periferica sia pronta ad effettuare un trasferimento, non può compiere nessun'altra operazione.[8]

1.5.7.2. Gestione in interruzione

La modalità di gestione *in interruzione* prevede che il calcolatore venga avvisato direttamente che una periferica è pronta per il trasferimento. Per ottenere ciò, si collega la linea /STB della periferica a un particolare ingresso della CPU. L'invio di un impulso su tale linea causa l'interruzione del programma che la CPU sta eseguendo (da cui il nome), e l'avvio dell'esecuzione di una procedura posta in una determinata zona della memoria.
Questa procedura:

- salva lo stato corrente della CPU;
- acquisisce il dato dalla periferica;
- ripristina lo stato della CPU;
- riprende l'esecuzione del programma dal punto in cui era stata interrotta.

Si noti che le interruzioni sono eventi *asincroni*, nel senso che possono arrivare in qualsiasi istante. Il calcolatore tuttavia le sincronizza, nel senso che, prima di interrompere il processo in atto, porta a termine l'esecuzione dell'operazione in corso; quindi, tra il segnale e l'effettiva interruzione, trascorre un breve lasso di tempo.
Utilizzando la gestione a controllo di programma non è possibile gestire più periferiche contemporaneamente; con la gestione in interruzione esistono invece due modalità per farlo:

- gestione *polled*:
  tutte le interruzioni causano l'esecuzione della stessa procedura, che come prima cosa determina quale periferica ha causato l'interruzione, e poi si comporta di conseguenza;
- gestione *vectored*:
  ogni interruzione causa l'esecuzione di una procedura diversa,[9] che quindi "sa" implicitamente qual è la periferica che ha causato l'interruzione.

---

[8] In questo e nel successivo paragrafo si fa riferimento solo a periferiche di ingresso: il discorso è comunque esattamente lo stesso anche nel caso di periferiche di uscita.

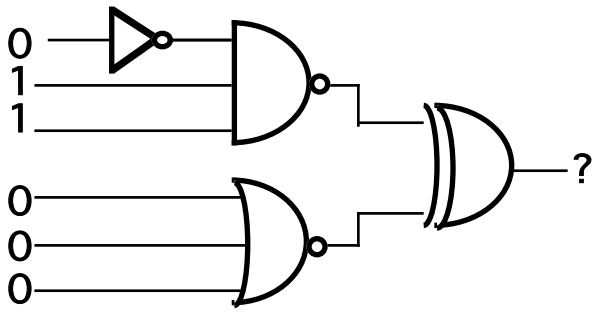
[9] La descrizione dei meccanismi hardware che rendono possibile ciò esula dagli scopi di questo corso.

1.6. PROBLEMI ED ESERCIZI*

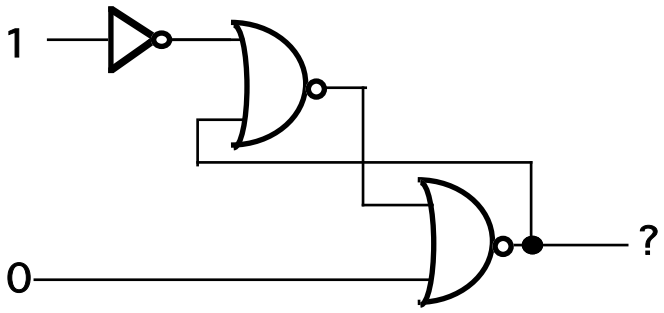
Problema 1.1.
Gli insiemi funzionalmente completi sono quelli:
a. che comprendono tutti gli operatori dell'algebra booleana;
b. che permettono di realizzare qualunque rete combinatoria.

Problema 1.2.
In algebra binaria, l'equazione $x + (x \bullet y) = x$ è vera o falsa?

Problema 1.3.
Determinare il valore dell'uscita della rete che segue, in presenza dei valori di ingresso indicati.



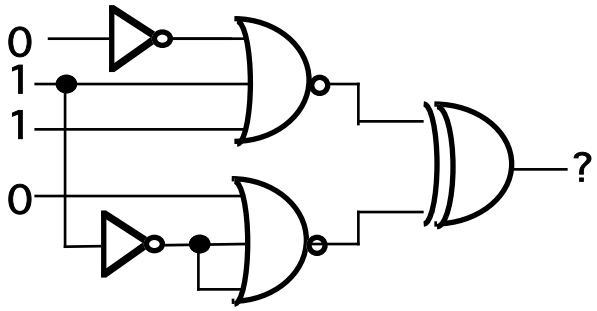
Problema 1.4.*
Determinare il valore dell'uscita della rete che segue, in presenza dei valori di ingresso indicati.



Problema 1.5.
Determinare il valore dell'uscita della rete che segue, in presenza dei valori di ingresso indicati.

---

* In questo e nei successivi capitoli, i problemi di cui è data la soluzione alla fine del volume sono contrassegnati da un asterisco.



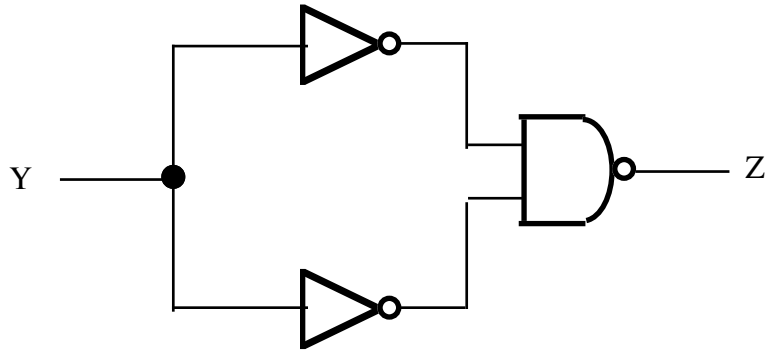
Problema 1.6.*
Dire quale o quali delle espressioni indicate qui di seguito realizzano la funzione della mappa.

X1X2 \ X3X4	00	01	11	10
00	1	1	1	1
01				
11				1
10	1			1

1. $\overline{X}_1\overline{X}_2 + \overline{X}_2\overline{X}_4 + X_1X_3\overline{X}_4$
2. $(X_1 + \overline{X}_2) \bullet (\overline{X}_2 + \overline{X}_3) \bullet (\overline{X}_2 + \overline{X}_4)$
3. $\overline{X}_1\overline{X}_2 + \overline{X}_1\overline{X}_2\overline{X}_4 + X_1X_3\overline{X}_4$
4. $(X_1 + X_2) \bullet (X_2 + X_4) \bullet (\overline{X}_1 + \overline{X}_3 + X_4)$

Problema 1.7.
La prima forma canonica è:
a. unica e minima;
b. unica ma non necessariamente minima;
c. non necessariamente unica e non necessariamente minima.

Problema 1.8.*
Analizzare il circuito che segue, e scrivere la funzione che realizza, ammesso che esista.



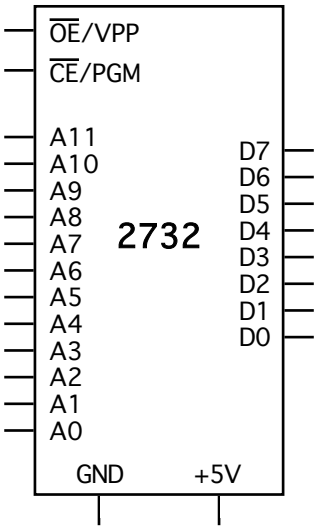
Problema 1.9.*
Le mappe di Karnaugh servono per analizzare il comportamento delle reti combinatorie?

Problema 1.10.
Due uscite di un decodificatore possono essere attive nello stesso istante?

Problema 1.11.
In un decodificatore, il numero delle uscite è sempre superiore al numero degli ingressi?

Problema 1.12.*
La capacità di una RAM, organizzata a parole di 8 bit, che ha in tutto 24 piedini, è di 2048 byte, 4096 byte, 8192 byte, 16384 byte, 32768 byte o 65536 byte?

Problema 1.13.
La figura che segue rappresenta schematicamente una tipica EPROM. Sapendo che "A" sta per *Address* e "D" sta per *Data*, determinare la sua capacità.



Problema 1.14.
Lo stato di una macchina sequenziale è funzione delle sole variabili di ingresso?

Problema 1.15.
Sintetizzare, nel miglior modo possibile, la funzione rappresentata nella tabella che segue.

X1X2 \ X3X4	00	01	11	10
00	1	1	1	1
01	1	1	1	
11				1
10	1			1

Problema 1.16.*
Scrivere la prima forma canonica della funzione rappresentata dalla tabella che segue.

X1X2 \ X3X4	00	01	11	10
00				
01				
11			1	1
10			1	1

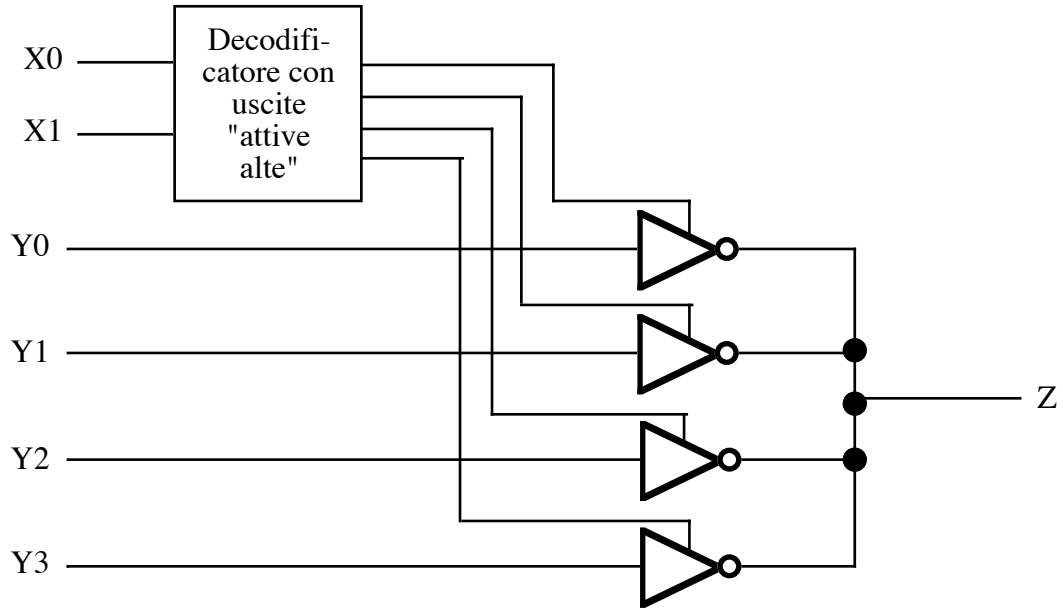
Problema 1.17.
Quanti piedini deve avere un decodificatore a 4 ingressi (escludendo quelli di alimentazione e di massa)?

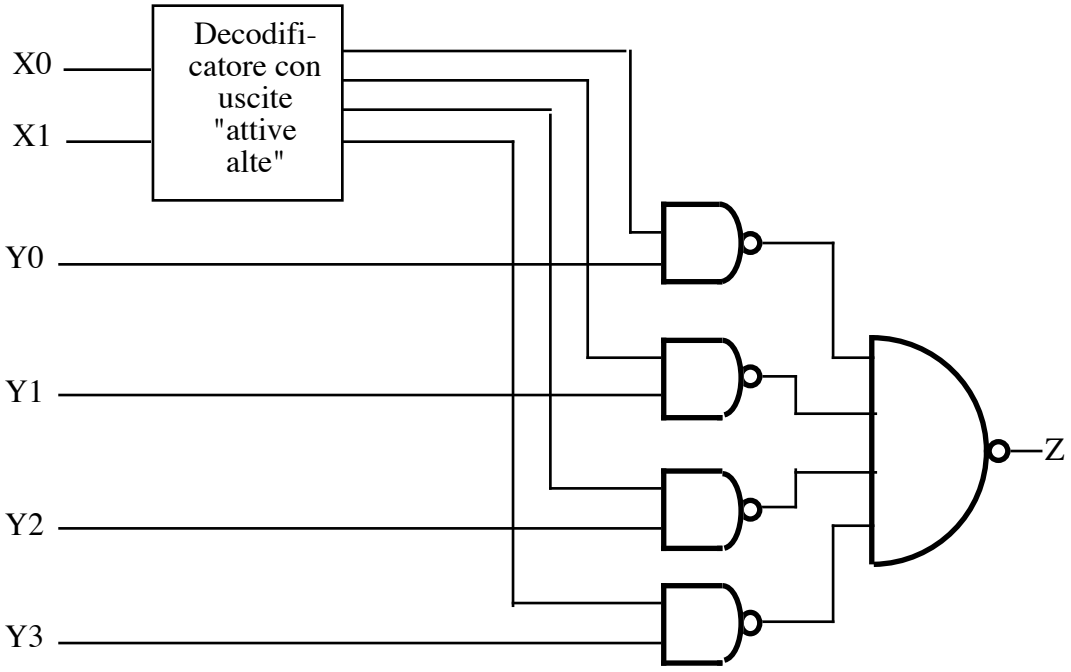
a. 4;
b. 16;
c. 20;
d. 21;
e. 24.

Problema 1.18.
Se un microcalcolatore ha un bus indirizzi composto da 16 linee e un bus dati composto da 8 linee, la massima quantità di memoria indirizzabile è:

a. 4 Kbyte;
b. 8 Kbyte;
c. 32 Kbyte;
d. 56 Kbyte;
e. 64 Kbyte.

Problema 1.19.
Le due reti che seguono sono equivalenti? Se no, perché?





2

LA RAPPRESENTAZIONE DELL'INFORMAZIONE NEI CALCOLATORI

Questo capitolo è dedicato all'integrazione di nozioni, che il lettore certamente già possiede, riguardanti i metodi per rappresentare le informazioni nei sistemi digitali. È opportuno osservare che, con i calcolatori di oggi, la trasformazione e l'elaborazione delle informazioni avvengono in maniera totalmente automatica nella maggior parte dei casi: ciò non richiede quindi, da parte dell'utente, una conoscenza approfondita dei metodi per rappresentare, appunto, le informazioni da trattare. In alcuni casi tuttavia alcune nozioni di base possono essere utili non solo per un fatto di cultura generale, ma anche perché può essere necessario effettuare alcune semplici elaborazioni senza dovere per forza ricorrere ai calcolatori. Abbiamo già visto un esempio di ciò nel primo capitolo, a proposito del progetto delle memorie dei calcolatori.

2.1. CODICI NUMERICI

La nascita dei calcolatori è stata principalmente determinata dalla necessità di disporre di strumenti in grado di eseguire rapidamente operazioni aritmetiche. La logica conseguenza è stata la formalizzazione della rappresentazione dei numeri, che

nella nostra mente sono sempre espressi in forma decimale, in una forma compatibile con la struttura dei calcolatori, che operano con un codice binario.
La rappresentazione di un numero in un calcolatore può assumere varie forme, a seconda che il numero sia intero, reale, ecc., e può occupare una quantità più o meno grande di bit, in dipendenza della precisione con cui si vuole rappresentare il numero stesso.
Ricordiamo le principali modalità di rappresentazione dei numeri utilizzate nei calcolatori:

- interi positivi;
- interi relativi, rappresentati in valore e segno;
- interi relativi, rappresentati in complemento a uno;
- interi relativi, rappresentati in complemento a due;
- numeri razionali, rappresentati in virgola fissa;
- numeri razionali, rappresentati in virgola mobile.

2.1.1. NUMERI INTERI

2.1.1.1. Numeri interi positivi

I numeri rappresentati sono semplicemente la conversione in binario dei corrispondenti numeri decimali. Le grandezze rappresentabili vanno da 0 a $2^N \sqcap 1$, dove N è il numero di bit utilizzati. Quindi, ad esempio, con 8 bit è possibile rappresentare tutti i numeri da 0 a 255, con 16 tutti i numeri da 0 a 65535, ecc.
Ricordiamo l'algoritmo di conversione dei numeri decimali in binari: si divide il numero da convertire per due, tralasciando la parte decimale, e si segna in una apposita colonna il resto dell'operazione, che ovviamente può essere solo o 0 o 1; si ripete il procedimento fino ad avere 0 al quoziente. A questo punto, il numero binario è costituito da tutti i resti, presi in ordine inverso (il bit meno significativo è il primo resto trovato). Un esempio di conversione da decimale a binario è mostrato in figura 2.1.

3	5	5	4	1_{10}	*Resto*	
1	7	7	7	0	**1**	
	8	8	8	5	**0**	
	4	4	4	2	**1**	
	2	2	2	1	**0**	
	1	1	1	0	**1**	
		5	5	5	**0**	
		2	7	7	**1**	
		1	3	8	**1**	↑
			6	9	**0**	
			3	4	**1**	
			1	7	**0**	
				8	**1**	
				4	**0**	
				2	**0**	
				1	**0**	
				0	**1**	

$$35541_{10} = 1000101011010101_2$$

Figura 2.1. - Conversione da decimale a binario.

La conversione opposta (da binario a decimale) si effettua sommando le successive potenze di due, moltiplicate ognuna per il corrispondente bit del numero da convertire, come mostrato in figura 2.2.
Molto spesso, per risparmiare spazio, invece della notazione binaria viene usata quella esadecimale. La conversione da binario a esadecimale è immediata: occorre semplicemente dividere il numero binario in gruppi di 4 bit, partendo dal bit meno significativo, e sostituire ad ogni gruppo la corrispondente cifra esadecimale, ricavabile dalla tabella di figura 2.3. I gruppi di quattro bit sono chiamati, con termine inglese, *nibble*.
Così, per il numero visto precedentemente, è

$$1000 \quad 1010 \quad 1101 \quad 0101_2 = 8AD5_{16}$$

1	•	2^0	=	1	•	1	= 1
0	•	2^1	=	0	•	2	= 0
1	•	2^2	=	1	•	4	= 4
0	•	2^3	=	0	•	8	= 0
1	•	2^4	=	1	•	16	= 16
0	•	2^5	=	0	•	32	= 0
1	•	2^6	=	1	•	64	= 64
1	•	2^7	=	1	•	128	= 128
0	•	2^8	=	0	•	256	= 0
1	•	2^9	=	1	•	512	= 512
0	•	2^{10}	=	0	•	1024	= 0
1	•	2^{11}	=	1	•	2048	= 2048
0	•	2^{12}	=	0	•	4096	= 0
0	•	2^{13}	=	0	•	8192	= 0
0	•	2^{14}	=	0	•	16384	= 0
1	•	2^{15}	=	1	•	32768	= 32768
							35541

Figura 2.2. - Conversione da binario a decimale.

Binario	*Hex*		*Binario*	*Hex*
0000	0		1000	8
0001	1		1001	9
0010	2		1010	A
0011	3		1011	B
0100	4		1100	C
0101	5		1101	D
0110	6		1110	E
0111	7		1111	F

Figura 2.3. - Tabella per la conversione binaria-esadecimale e viceversa.

2.1.1.2. Numeri interi rappresentati in valore e segno

Il modulo del numero viene rappresentato come nel caso precedente; per indicarne il segno, si utilizza un bit aggiuntivo che vale 0 se il numero è positivo, 1 se esso è negativo. Ad esempio:

$$105_{10} = 0110\,1001_2$$
$$-105_{10} = 1110\,1001_2$$

2.1.1.3. Numeri interi rappresentati in complemento a 1

I numeri positivi vengono rappresentati normalmente, mentre quelli negativi sono ottenuti dai corrispondenti positivi complementando ogni cifra. Ad esempio:

$$105_{10} = 0110\,1001_2$$
$$-105_{10} = 1001\,0110_2$$

2.1.1.4. Numeri interi rappresentati in complemento a 2

Per i vantaggi che offre, questa è la rappresentazione più comunemente usata per i numeri interi. Anche qui, i numeri positivi vengono rappresentati normalmente, mentre quelli negativi sommando 1 al corrispondente complemento a 1. Ad esempio:

$$105_{10} = 0110\,1001_2$$
$$-105_{10} = 1001\,0111_2$$

Osserviamo che, con la notazione in complemento a 2, è possibile rappresentare numeri i cui valori vanno da -2^{N-1} a $+2^{N-1}-1$. Per chiarire il concetto, osserviamo la figura 2.4. Essa mostra alcuni numeri binari di 8 bit e i loro corrispondenti decimali. È evidente che il numero 10000000_2 potrebbe rappresentare indifferentemente sia il numero $+128_{10}$ che il numero -128_{10}. Per eliminare l'ambiguità, stabiliamo che esso rappresenti il numero -128: operando in questo modo, abbiamo il vantaggio che tutti i numeri positivi hanno il primo bit a 0, mentre tutti quelli negativi iniziano con un 1.

2.1.2. NUMERI REALI

2.1.2.1. Numeri reali in virgola fissa

Sono rappresentati utilizzando un gruppo di bit per la parte intera, un altro per la parte frazionaria, e un bit per il segno. Le dimensioni e la precisione dei numeri rappresentabili dipendono dalle dimensioni dei gruppi di bit utilizzati. Effettuare operazioni con i numeri binari in virgola fissa è relativamente semplice, in quanto si applicano le stesse regole che si usano per i numeri decimali, e i tempi di calcolo sono abbastanza ridotti. Lo svantaggio è dato dal fatto che la gamma di numeri rappresentabili con un certo insieme di bit è forzatamente limitata: questa rappresentazione è quindi utile in alcuni tipi di calcoli, in cui si devono trattare numeri di dimensioni note a priori.

Base 2	*Base* 10
0000 0000	0
0000 0001	1
0000 0010	2
□	□
01111110	126
01111111	127
1000 0000	□128
1000 0001	□127
□	□
11111110	□2
11111111	□1

Figura 2.4. - Numeri rappresentati in complemento a 2.

2.1.2.2. Numeri reali in virgola mobile

La notazione in virgola mobile è l'esatto equivalente della notazione scientifica usata per i numeri decimali: ogni numero viene rappresentato mediante l'espressione $X = a \bullet 2^b$, dove *a* e *b*, detti rispettivamente *mantissa* e *caratteristica* del numero, sono numeri in complemento a 2. In questo caso, la precisione della rappresentazione dipende dal numero di bit utilizzati per la mantissa, mentre le dimensioni dei numeri rappresentabili dipendono dal numero di bit della caratteristica.

2.2. CODICI ALFANUMERICI

L'evoluzione dei calcolatori, già a pochi anni dal loro apparire, li ha portati a diventare, oltre che elaboratori di numeri, anche elaboratori di altri tipi di informazione, prima fra tutti quella testuale. I simboli che vengono usati per rappresentare testi sono, come è noto, i cosiddetti caratteri *alfanumerici*, cioè l'insieme costituito dalle lettere dell'alfabeto e dalle dieci cifre decimali. A questi vanno aggiunti diversi altri simboli, come lo spazio, i segni di interpunzione, i simboli per indicare il passaggio alla riga o alla pagina successiva, ecc.
Questo insieme, essendo numerabile, può ovviamente essere rappresentato attribuendo in maniera univoca a ciascuno dei suoi elementi un numero (*codice*). Non esistono regole particolari per stabilire questa corrispondenza, che può essere fatta in maniera arbitraria: tuttavia, per garantire la compatibilità della rappresentazione con sistemi diversi, è opportuno che essa sia fatta secondo uno standard universalmente accettato.
Osserviamo che le lettere dell'alfabeto inglese sono 26, per un totale di 52 simboli considerando lettere minuscole e maiuscole; se ad esse aggiungiamo le dieci cifre e

una quarantina fra segni di interpunzione e codici con significati speciali arriviamo ad un centinaio di simboli diversi. Per codificarli in binario, occorrono almeno 7 bit, che permettono un totale di 128 configurazioni diverse.

2.2.1. CODICE ASCII

Dopo un periodo di "anarchia" durato diversi anni, in cui ognuno dei principali costruttori di calcolatori tentava di imporre il proprio metodo di codifica come standard universale, la standardizzazione è stata raggiunta con l'adozione del cosiddetto codice *ASCII* (A*merican* S*tandard* C*ode for* I*nformation* I*nterchange*)[10] che, ad onor del vero, esisteva già da prima, perché era usato nella comunicazione mediante telescriventi. Di questo codice esistono varie versioni, in cui si impiega un diverso numero di bit e che, di conseguenza, comprendono un diverso numero di simboli.

Per molti anni è stato usato il codice ASCII a 7 bit (128 caratteri); in seguito, l'introduzione di terminali sofisticati e la sempre crescente diffusione di programmi per l'elaborazione dei testi hanno resa necessaria la sua espansione per includere le lettere accentate e diversi simboli grafici.

La soluzione è stata quella di aggiungere un ottavo bit al codice, che porta i caratteri rappresentabili a 256, come mostrato in figura 2.5. In questa figura sono riportate le codifiche ASCII relative a tutte le configurazioni da 00 a FF_H. Si osservi che i codici relativi a lettere e numeri sono ordinati in modo crescente. Ciò rende particolarmente semplice l'ordinamento alfabetico di insiemi di caratteri, che è effettuabile confrontando i valori numerici dei codici attribuiti a ogni lettera.

[10] La pronuncia corretta di questa sigla, al contrario di quanto si crede comunemente, è "ASKI".

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	*NUL*	*DLE*		0	@	*P*	`	*p*	*Ä*	*ê*	†	•	¿	–	‡	
1	*SOH*	*DC*1	!	1	*A*	*Q*	*a*	*q*	*Å*	*ë*	°	±	¡	—	·	*Ò*
2	*STX*	*DC*2	"	2	*B*	*R*	*b*	*r*	*Ç*	*í*	¢	□	□	“	‚	*Ú*
3	*ETX*	*DC*3	#	3	*C*	*S*	*c*	*s*	*É*	*ì*	£	≥	÷	”	„	*Û*
4	*EOT*	*DC*4	$	4	*D*	*T*	*d*	*t*	*Ñ*	*î*	§	¥	□	‘	‰	*Ù*
5	*ENQ*	*NAK*	%	5	*E*	*U*	*e*	*u*	*Ö*	*ï*	•	□	□	’	*Â*	*ı*
6	*ACK*	*SYN*	&	6	*F*	*V*	*f*	*v*	*Ü*	*ñ*	¶	∂	□	÷	*Ê*	ˆ
7	*BEL*	*ETB*	'	7	*G*	*W*	*g*	*w*	*á*	*ó*	*ß*	□	«	□	*Á*	˜
8	*BS*	*CAN*	(	8	*H*	*X*	*h*	*x*	*à*	*ò*	□	□	»	*ÿ*	*Ë*	¯
9	*HT*	*EM*	)	9	*I*	*Y*	*i*	*y*	*â*	*ô*	□	□	□	*Ÿ*	*È*	˘
A	*LF*	*SUB*	*	:	*J*	*Z*	*j*	*z*	*ä*	*ö*	□	□	~	□	*Í*	˙
B	*VT*	*ESC*	+	;	*K*	[	*k*	{	*ã*	*õ*	□	ª	*À*		*Î*	°
C	*FF*	*FS*	,	<	*L*	\	*l*	|	*å*	*ú*	¨	º	*Ã*	‹	*Ï*	¸
D	*CR*	*GS*	□	=	*M*	]	*m*	}	*ç*	*ù*	≠	□	*Õ*	›	*Ì*	˝
E	*SO*	*RS*	.	>	*N*	^	*n*	~	*é*	*û*	*Æ*	æ	*Œ*	*fi*	*Ó*	˛
F	*SI*	*US*	/	?	*O*	_	*o*	*DEL*	*è*	*ü*	*Ø*	*ø*	*œ*	*fl*	*Ô*	ˇ

Figura 2.5. - Il codice ASCII a 8 bit.

Le prime due colonne contengono i cosiddetti *non-printing characters*, cioè codici che corrispondono a particolari azioni del terminale che li riceve, ma che non producono la visualizzazione o la stampa di un carattere: ad esempio, *BEL* causa l’attivazione del “campanello” del terminale, *LF* il salto alla riga successiva, e così via.
Occorre anche osservare che, mentre tutti i costruttori si sono uniformati al codice ASCII per quanto riguarda le prime 128 configurazioni, non c’è un accordo completo per la seconda parte della tabella: esiste infatti un secondo standard, detto *ANSI* (A*merican* N*ational* S*tandard* I*nstitute*), che è uguale all’ASCII per le prime otto colonne, ma riporta simboli diversi per le seconde. Inoltre, alcuni calcolatori non seguono né lo standard ASCII, né quello ANSI. Tutto ciò implica che, nel passaggio di informazioni da un calcolatore ad uno di tipo diverso, occorra applicare opportune tabelle di conversione, se si vuole mantenere una completa corrispondenza dei simboli speciali utilizzati. Nessun problema sorge invece per quanto riguarda le lettere e gli altri simboli “normali”.

2.2.2. ALTRI CODICI

Sempre per la rappresentazione di caratteri sono stati usati, e in alcuni casi sono tuttora in uso, codici diversi, quali Hollerith, EBCDIC, ecc. A titolo puramente informativo, riportiamo in figura 2.6 il codice EBCDIC, che è tuttora usato in diversi sistemi IBM.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	*NUL*	*DLE*	*DS*		*SP*	&	□				_					0
1	*SOH*	*DC*1	*SOS*				/		*a*	*j*			*A*	*J*		1
2	*STX*	*DC*2	*FS*	*SYN*					*b*	*k*	*s*		*B*	*K*	*S*	2
3	*ETX*	*TM*							*c*	*l*	*t*		*C*	*L*	*T*	3
4	*PF*	*RES*	*BYP*	*PN*					*d*	*m*	*u*		*D*	*M*	*U*	4
5	*HT*	*NL*	*LF*	*RS*					*e*	*n*	*v*		*E*	*N*	*V*	5
6	*LC*	*BS*	*ETB*	*UC*					*f*	*o*	*w*		*F*	*O*	*W*	6
7	*DEL*	*IL*	*ESC*	*EOT*					*g*	*p*	*x*		*G*	*P*	*X*	7
8		*CAN*							*h*	*q*	*y*		*H*	*Q*	*Y*	8
9	*RLF*	*EM*							*i*	*r*	*z*		*I*	*R*	*Z*	9
A	*SMM*	*CC*	*SM*		[	]		:								
B	*VT*	*CU*1	*CU*2	*CU*3	.	$		#								
C	*FF*	*IFS*		*DC*4	<	*	%	@					~			
D	*CR*	*IGS*	*ENQ*	*NAK*	(	)	□	'								
E	*SO*	*IRS*	*ACK*		+	;	>	=								
F	*SI*	*IUS*	*BEL*	*SUB*	|	□	?	"								

Figura 2.6. - Il codice EBCDIC.

2.3. RAPPRESENTAZIONE DI ALTRE INFORMAZIONI

Negli ultimi anni abbiamo assistito all'introduzione, nel mondo dei calcolatori, di una enorme quantità di software atto ad elaborare informazioni che non possono essere rappresentate semplicemente con i codici di cui abbiamo parlato finora. Si tratta della rappresentazione di oggetti grafici, che possono andare dalle semplici icone che ormai appaiono sullo schermo di tutti i piccoli calcolatori, a disegni tecnici, a immagini fotografiche in bianco e nero e a colori, fino ad arrivare, con l'avvento della cosiddetta *multimedialità*, alla rappresentazione e riproduzione di disegni animati, film e suoni.

È chiaro che ognuna delle categorie di oggetti appena menzionate ha esigenze diverse per quanto riguarda la rappresentazione all'interno di una macchina che è sempre e solo in grado di elaborare informazioni numeriche.
Lo scopo di questo capitolo non è quello di fornire una elencazione dei diversi formati, più o meno standard, che esistono per rappresentare informazioni diverse da quelle numeriche e testuali, ma semplicemente quello di dare un'idea generale sulle problematiche esistenti e sui modi per risolverle.

2.3.1. RAPPRESENTAZIONE DELLE IMMAGINI

Il calcolatore, che per definizione è una macchina che opera su quantità *discrete*, non può rappresentare compiutamente un'immagine, che per sua natura contiene dati *continui*. La rappresentazione finale di un'immagine (sullo schermo o in una stampa) sarà quindi una approssimazione dell'originale, ottenuta dividendolo in tante piccole aree, che prendono il nome di *PIXEL* (*Picture Elements*), a ognuna delle quali si associa un valore di luminosità e, nel caso di immagini a colori, di crominanza.
Tuttavia, esistono due modi fondamentalmente diversi per rappresentare, all'interno del calcolatore, l'immagine: il primo prevede la rappresentazione di ogni singolo pixel, mentre il secondo contiene la descrizione degli oggetti che compongono l'immagine.

Figura 2.7. - Una figura geometrica.

Per chiarire il concetto, consideriamo la figura 2.7. Essa contiene una figura geometrica (l'ovale) ed una scritta. Per rappresentarla, è possibile suddividerla in tanti piccoli quadrati e, per ognuno di essi, specificare se esso deve essere costituito da una superficie bianca o nera, come si può osservare in figura 2.8, che ne mostra l'ingrandimento di una piccola porzione.

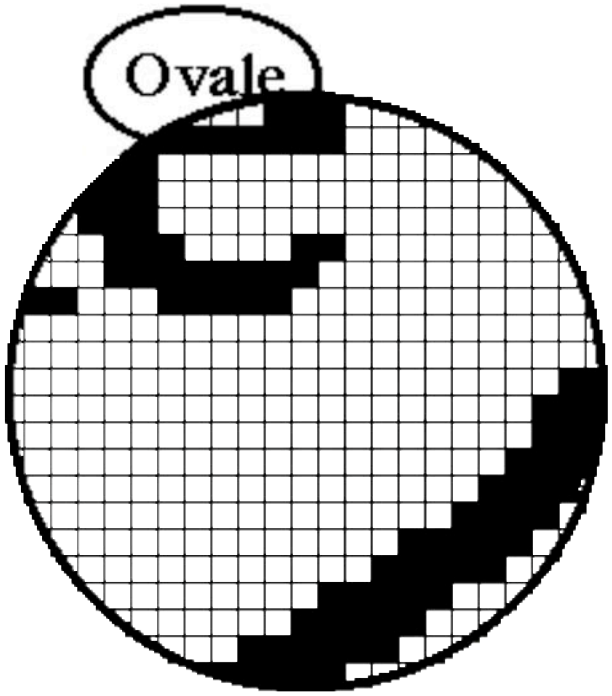


Figura 2.8. - Rappresentazione bitmap della figura 2.7.

Rappresentando l'immagine in questo modo, occorre una quantità di informazione (Fig. 2.9) che dipende dalle dimensioni dell'immagine stessa, dalla risoluzione (cioè dalle dimensioni del singolo pixel) e dal numero di bit adoperati per rappresentare ogni pixel. Infatti, nell'esempio presentato abbiamo ipotizzato che ogni pixel possa essere soltanto o bianco o nero, ma possiamo anche pensare di definire vari livelli di grigio che ogni pixel può assumere. Nella pratica, si utilizzano in genere 16 o 256 livelli di grigio, il che vuol dire che occorrono 4 o 8 bit per ogni pixel.
Nel caso di immagini a colori, occorre triplicare la quantità di informazione: il metodo più comunemente usato è quello di esprimere, per ogni pixel, i valori di luminosità dei tre colori primari rosso, verde e blu. In questo caso, per ogni pixel occorrono da 12 a 24 bit, che permettono di rappresentare rispettivamente $16^3 = 4096$ e $256^3 = 16777216$ colori diversi.
La risoluzione può variare entro limiti molto ampi: normalmente si va dai 72 pixel per pollice tipici degli schermi a raggi catodici a 300 e 600 pixel per pollice per le stampanti a getto di inchiostro o a laser, ma si può arrivare fino a 4000 ed anche 8000 punti per pollice per le fotocompositrici professionali di alta qualità.[11]

[11] Il numero di pixel per pollice viene in genere indicato con la sigla *dpi* (dots per inch). Ricordiamo che un pollice equivale esattamente a 2,54 cm.

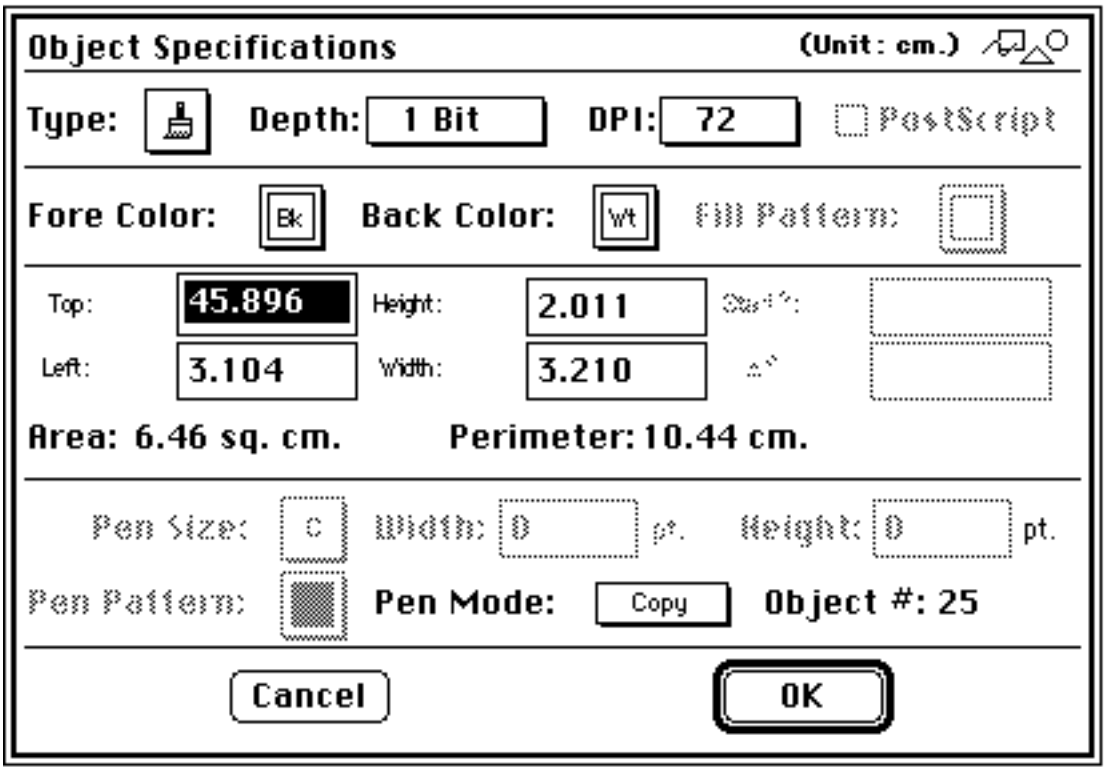


Figura 2.9. - Informazioni corrispondenti all'immagine bitmap di figura 2.7.

È quindi evidente che la quantità di informazione necessaria per rappresentare un'immagine con questo metodo è elevatissima: per esempio, una fotografia a colori di 4x4 pollici (circa 10x10 cm), rappresentata con una risoluzione di 600 pixel per pollice e con 16 milioni di colori[12] richiede, per la sua memorizzazione, più di 17 Mbyte. Sorge quindi la necessità di utilizzare metodi di *compressione* delle informazioni che riducano queste cifre a valori più trattabili.
In alternativa, è possibile memorizzare non l'immagine nel suo complesso, ma una descrizione degli oggetti che la compongono. Ad esempio, il disegno di figura 2.7 potrebbe essere descritto con le informazioni mostrate nelle figure 2.10 e 2.11. È chiaro che, in questo caso, la quantità di informazione da memorizzare è estremamente più piccola, ma che con questo metodo è possibile descrivere solo immagini composte da oggetti definiti a priori. Il metodo quindi non si presta a rappresentare qualunque tipo di immagine, ma va molto bene per tutta una serie di oggetti grafici (ad esempio i disegni tecnici), che sono composti solo da oggetti noti.

[12] Queste specifiche danno una qualità buona, ma non certamente paragonabile a quella di una stampa fotografica.

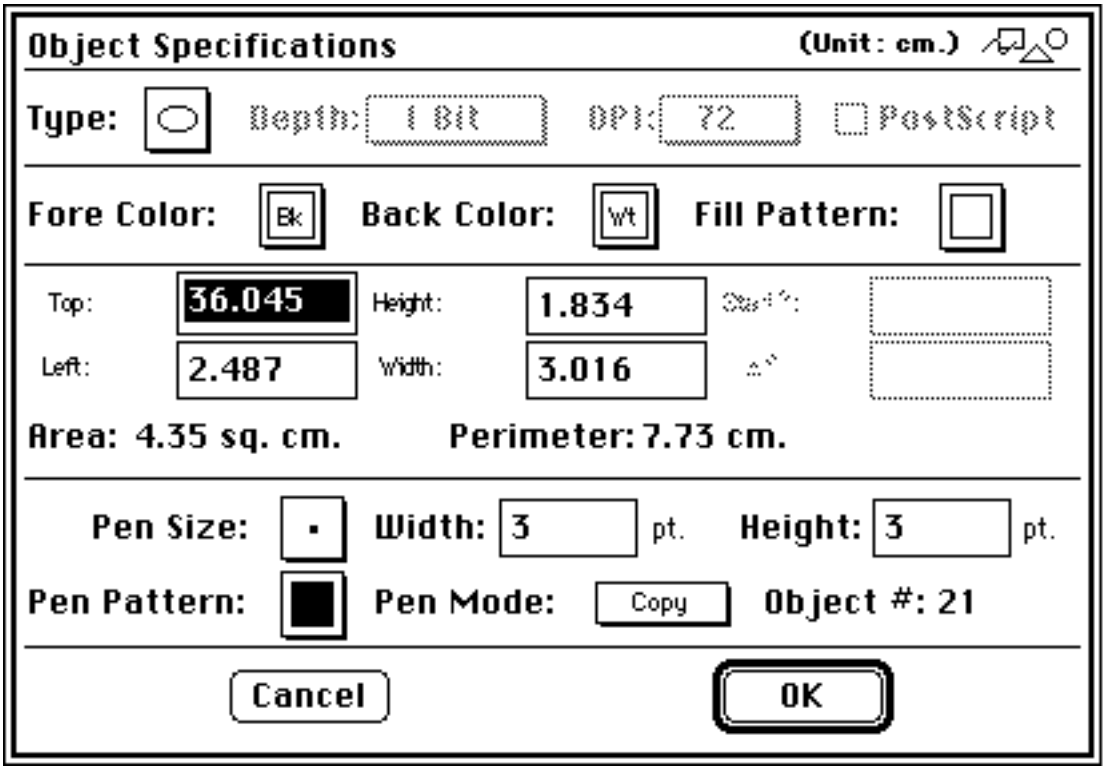


Figura 2.10. - Specifiche dell'ovale di figura 2.7.

La rappresentazione "a oggetti" delle immagini offre altri vantaggi: la possibilità di "collegare" fra loro gli oggetti, in modo che, spostandone uno, si spostino anche quelli collegati, e quella di poter cambiare le loro dimensioni senza dover effettuare calcoli complicati.

2.3.2. RAPPRESENTAZIONE DI SUONI

Anche per i suoni valgono concetti analoghi a quelli visti per l'immagine. Molti calcolatori moderni sono dotati di una periferica per la produzione del suono piuttosto complessa, che è in grado sia di produrre suoni per *sintesi*, cioè basandosi su una descrizione simbolica dei suoni da produrre (forma d'onda, frequenza, durata, intensità), sia convertendo un segnale sonoro *campionato*.
Quest'ultima modalità si basa sul principio che un segnale può essere riprodotto a partire da un certo numero di campioni, presi a distanza sufficientemente ravvicinata l'uno dall'altro (Fig. 2.12). Di ogni campione deve essere registrato il valore, che va quindi convertito in un numero.

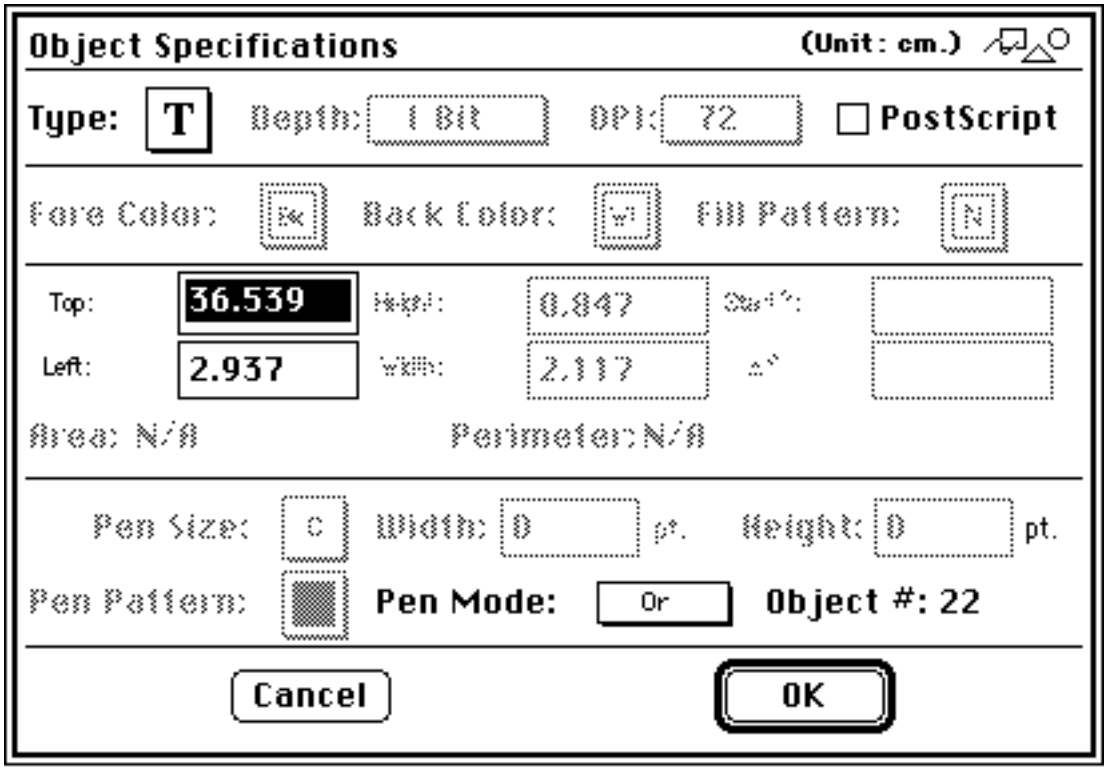


Figura 2.11. - Specifiche della scritta di figura 2.7.

Nel caso dei suoni, la quantità di memoria occorrente dipende dalla durata del suono da registrare, dalla frequenza dei campioni[13] e dalla risoluzione utilizzata. Se, come spesso accade, il suono è stereofonico, e quindi composto da due segnali, le quantità vanno moltiplicate per due. Questo fa sì che per registrare un'ora di musica stereofonica, con una frequenza di campionamento di 44 KHz e una risoluzione di 16 bit (come si fa nei *compact disc*), occorrano quasi 650 Mbyte. Evidentemente anche in questo caso le tecniche di compressione dei dati sono essenziali per ridurre la quantità dei dati.

2.3.3. RAPPRESENTAZIONE DI IMMAGINI IN MOVIMENTO

Come è noto, le immagini in movimento non sono altro che una successione sufficientemente veloce di immagini ferme. Quanto è già stato detto a proposito di queste ultime continua quindi a valere, ma la quantità di dati necessaria aumenta enormemente, dal momento che per avere una buona illusione di movimento bisogna mostrare almeno 18 immagini al secondo (lo standard cinematografico ne prevede 24 e quello televisivo 25). Le tecniche di compressione, in questo caso assolutamente

---

[13] Secondo il teorema di Shannon, la frequenza dei campioni deve essere maggiore del doppio della più alta frequenza contenuta nel segnale da campionare.

indispensabili, si basano soprattutto sul fatto che le immagini successive sono in genere abbastanza simili, ed è quindi sufficiente registrare le differenze fra un'immagine e quella che segue.

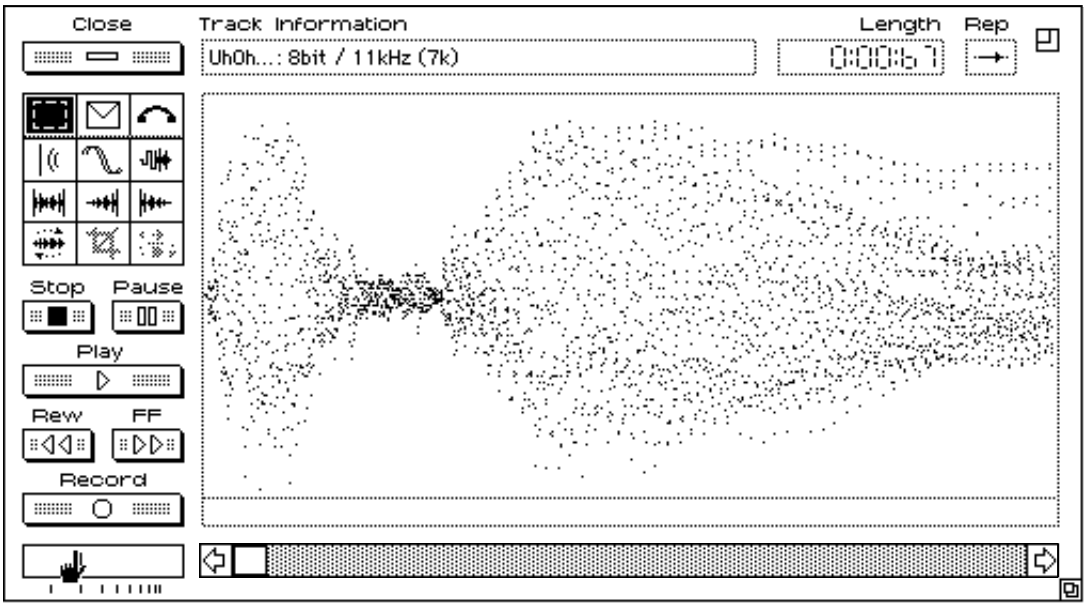


Figura 2.12. - Un suono campionato.

2.4. PROBLEMI ED ESERCIZI

Problema 2.1.*
Convertire i seguenti numeri decimali nella rappresentazione binaria in complemento a 2 a 8 bit:

a. 10;
b. -47;
c. 126;
d. -78;
e. 0.

Problema 2.2.*
Convertire i seguenti numeri decimali nella rappresentazione binaria in complemento a 2 a 16 bit:

a. 170;
b. -4712;
c. 1265;
d. -7808;
e. 0.

Problema 2.3.*
Convertire i seguenti numeri decimali nella rappresentazione esadecimale in complemento a 2 a 16 bit:

a. 320;
b. -15600;
c. 14875;
d. -255.

Problema 2.4.*
Convertire i seguenti numeri esadecimali in complemento a 2 a 16 bit nei corrispondenti numeri decimali:

a. 00FF;
b. 75A8;
c. 914A;
d. 72B5.

Problema 2.5.
La sequenza di numeri che segue rappresenta una stringa codificata con il codice ASCII. Scrivere tale stringa.

42 75 6F 6E 67 69 6F 72 6E 6F 2C 20 6C 65 74 74 6F 72 65 21

Problema 2.6.
Calcolare la quantità di informazione necessaria per memorizzare con il metodo bitmap una immagine in bianco e nero delle dimensioni di 7x10 cm, con la risoluzione di 72 dpi e 256 livelli di grigio.

Problema 2.7.
Calcolare la quantità di informazione necessaria per memorizzare 25 secondi di suono campionato a 22,5 KHz, con una risoluzione di 16 bit.

3

INTRODUZIONE AI SISTEMI OPERATIVI

Il calcolatore, così come è stato analizzato fino ad ora, è totalmente sprovvisto di software: quindi, anche se la conoscenza dei linguaggi di programmazione ci permette di scrivere qualunque programma, dobbiamo ancora esaminare i mezzi che permettono al calcolatore di eseguirlo. Occorre quindi definire le funzioni di un sistema software di supporto all'utente, che chiameremo *Sistema Operativo* (S.O.), la cui funzione fondamentale sia appunto quella di permettere all'utente di interagire con il calcolatore.

3.1. CARATTERIZZAZIONE DEI SISTEMI OPERATIVI

Le funzioni principali del sistema operativo sono:

a. Gestione dei lavori;
b. Supporti per la programmazione;
c. Meccanismi di ingresso/uscita (I/O);
d. Gestione degli archivi.

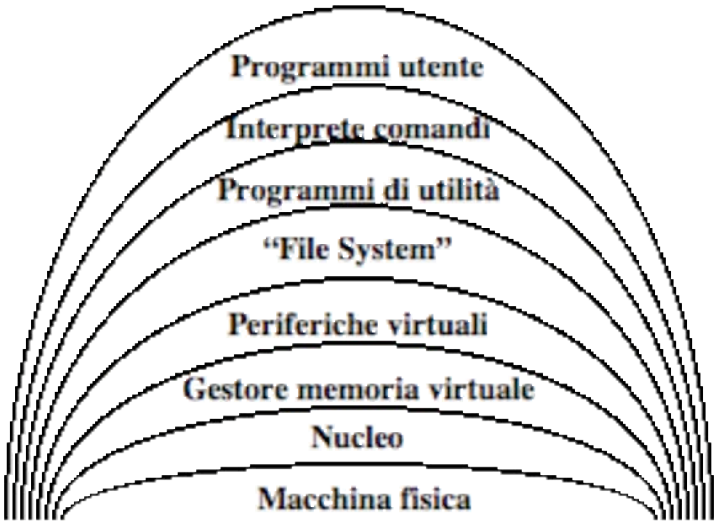


Figura 3.1. - Struttura a strati del sistema operativo.

Definiamo qui di seguito queste funzioni:

a. *gestione dei lavori*: Su quasi tutti i calcolatori "girano" più processi che apparentemente evolvono in parallelo fra di loro; siccome i calcolatori tradizionali (quelli cioè che si rifanno al modello della macchina di Von Neumann) non possono eseguire più di un programma per volta, avendo un solo program counter e una sola serie di circuiti aritmetico-logici, il S.O. si deve occupare dell'esecuzione delle varie parti dei processi usando modalità che diano l'impressione all'utente che tutti i processi stiano evolvendo simultaneamente;

b. *supporti per la programmazione*: Il S.O. fornisce all'utente un insieme di supporti che rendono più facile la preparazione ed il collaudo di programmi (per esempio text editor e debugger);

c. *meccanismi di ingresso/uscita* (I/O): Il S.O. permette la trasportabilità del software da calcolatore a calcolatore dello stesso tipo. Essendo infatti arbitrario l'assegnamento degli indirizzi della memoria e delle porte di un calcolatore, se un programma che utilizza determinati indirizzi per la memoria e per i dispositivi di I/O viene trasportato su un calcolatore analogo ma configurato in maniera differente non può più funzionare. Siccome non si possono né standardizzare gli indirizzi delle porte e della memoria, né adattare ogni programma ad ogni singolo calcolatore, è compito del S.O. unificare i differenti meccanismi di I/O e di accesso alla memoria;

d. *gestione degli archivi*: Un'esigenza fondamentale per l'utente è quella di poter manipolare grosse quantità di informazioni, memorizzate su supporti di massa. Tra le funzioni di un S.O. c'è quindi anche quella di gestione di tali informazioni (File System).

Le funzioni sopra elencate sono estremamente complesse anche per piccoli calcolatori: è perciò utile una divisione delle diverse funzionalità in più livelli, con ogni strato che racchiude tutti gli strati ai livelli inferiori. Ogni livello di una tale struttura gerarchica utilizza, oltre alle proprie, tutte le funzionalità dello strato sottostante e quindi fa qualcosa di più sofisticato rispetto ad esso; infine ogni strato interagisce unicamente con i due strati adiacenti (Fig. 3.1).
Primo livello: hardware. È la macchina base studiata fino a questo momento. Essa è schematizzabile come in figura 3.2.

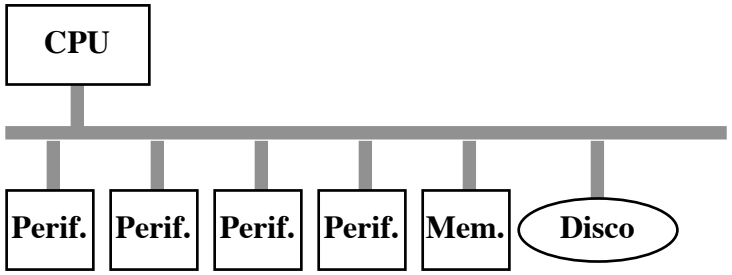


Figura 3.2. - Schematizzazione del calcolatore al primo livello.

Osserviamo che, a rigore, i supporti magnetici (qui indicati con la generica parola *disco*) sono anch'essi delle periferiche; l'uso che di essi si fa nei moderni calcolatori è però tale da suggerire, in questa suddivisione, di catalogarli come un tipo diverso di dispositivi.
Secondo livello: nucleo. È la parte del sistema operativo che si appoggia direttamente all'hardware ed è programmata utilizzando le istruzioni fisiche della macchina (Fig. 3.3).

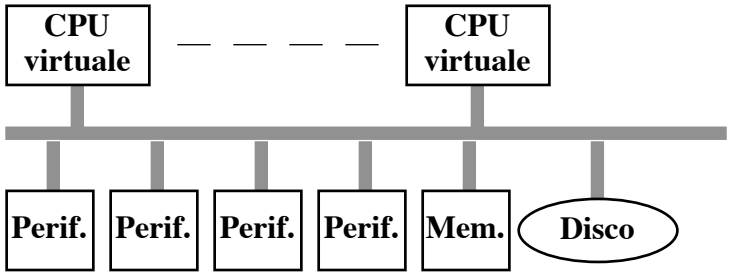


Figura 3.3. - Schematizzazione del calcolatore al secondo livello.

Svolge la funzione di gestione dei lavori vista precedentemente: definisce un certo numero di unità centrali *virtuali* al posto della CPU fisica; il calcolatore visto a questo livello del sistema operativo sembra dunque possedere più unità centrali, una per ogni

processo attivo nel sistema. Inoltre, il sistema operativo fa sì che l'esecuzione di tali programmi avvenga (apparentemente, come abbiamo detto) in parallelo.
Terzo livello: memoria virtuale. In presenza di più CPU virtuali, ognuna di esse ha ovviamente bisogno di una certa quantità di memoria per i dati e per il codice del processo che gira su di essa. Ci si scontra con il fatto che la memoria fisica è invece unica: questo creerebbe dei conflitti se due o più programmi utilizzassero nello stesso istante la stessa zona di memoria, perché uno dei programmi potrebbe leggere o, peggio, modificare i dati di un altro programma. Per evitare questo inconveniente, ad ogni CPU virtuale viene allora assegnata una porzione di memoria (detta anch'essa virtuale), così che la singola CPU virtuale ignora la memoria fisica dell'hardware e vede solamente la propria memoria virtuale (memoria che risulta inaccessibile a tutte le altre CPU virtuali), come mostrato in figura 3.4.
La corrispondenza (*mapping*) fra memoria virtuale e memoria fisica viene effettuata da appositi circuiti hardware; è compito del sistema operativo definire le sezioni (partizioni) di memoria da assegnare a ciascuna CPU virtuale in modo che non sorgano conflitti.

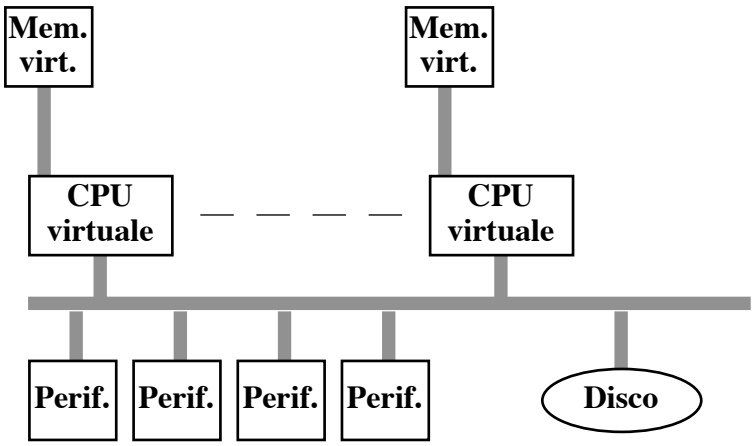


Figura 3.4. - Schematizzazione del calcolatore al terzo livello.

Quarto livello: periferiche virtuali. Il problema appena descritto per la memoria si ripresenta anche per le periferiche: se per esempio una CPU virtuale esegue un processo che prevede la stampa di una frase su una stampante, e nello stesso istante un'altra CPU virtuale esegue un processo analogo, la stampante fisica stamperà le due frasi mischiandole l'una con l'altra in maniera difficilmente prevedibile. Ciò avviene perché le periferiche, così come la memoria, non hanno la possibilità di discriminare i dati in arrivo in funzione del mittente. Il gestore delle periferiche virtuali fa sì che se una periferica fisica è gia utilizzata da qualche processo, nessun altro programma possa utilizzare la stessa periferica finché il primo processo non dichiari di aver finito di usarla (Fig. 3.5).

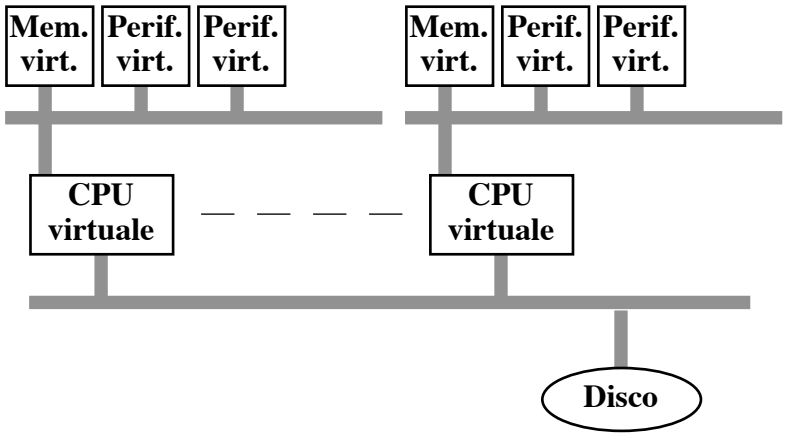


Figura 3.5. - Schematizzazione del calcolatore al quarto livello.

Il lettore si renderà conto a questo punto che una parte delle funzioni del sistema operativo consiste nella allocazione di particolari "risorse" a un processo piuttosto che a un altro. Risorse sono, per quanto abbiamo visto, zone di memoria e periferiche, ma possiamo includere in questo termine anche zone di memoria di massa, tempo di elaborazione,[14] ecc.

Quinto livello: gestore file. La virtualizzazione delle unità di memoria di massa del calcolatore è demandata ad una parte apposita, detta *File System*, del sistema operativo (Fig. 3.6). Dal momento che le memorie di massa sono utilizzate come archivi contenenti svariati tipi di informazioni, la loro gestione è evidentemente più complicata di quella, per esempio, di una stampante. Infatti, oltre a trasferire informazioni da e verso la memoria di massa, il sistema operativo deve anche preoccuparsi di gestire l'archivio di tali informazioni, sia per consentirne la reperibilità, sia per sfruttare opportunamente lo spazio disponibile. Questo modulo consente dunque all'utente di considerare le informazioni come organizzate logicamente in file accessibili attraverso nomi simbolici, di cui si ignora l'organizzazione fisica e, in alcuni casi, addirittura la allocazione su un supporto di memoria piuttosto che su un altro.

[14] Se la CPU sta eseguendo un determinato processo, non può eseguirne nessun altro: per questo motivo, possiamo vedere l'allocazione della CPU fisica ad un processo per un certo periodo come tutte le altre allocazioni di risorse.

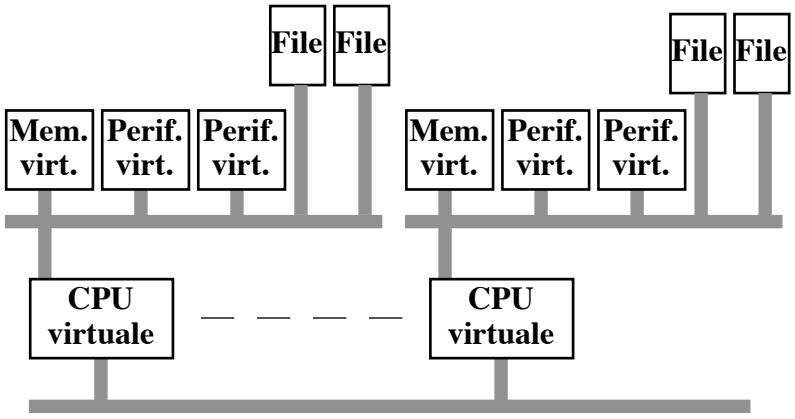


Figura 3.6. - Schematizzazione del calcolatore al quinto livello.

Nei sistemi più evoluti il modulo di gestione dei file comprende e virtualizza non solo le memorie di massa, ma anche le altre periferiche, come mostrato in figura 3.7. Così come si possono creare e scrivere file su disco, la stessa cosa può essere fatta con ogni altra periferica, fatto salvo il fatto che in genere le altre permettono solo o l'ingresso o l'uscita dei dati. I vantaggi di una tale gestione riguardano la comodità del sistema e soprattutto la compatibilità (non interessa più la fisicità di un disco o di una stampante).

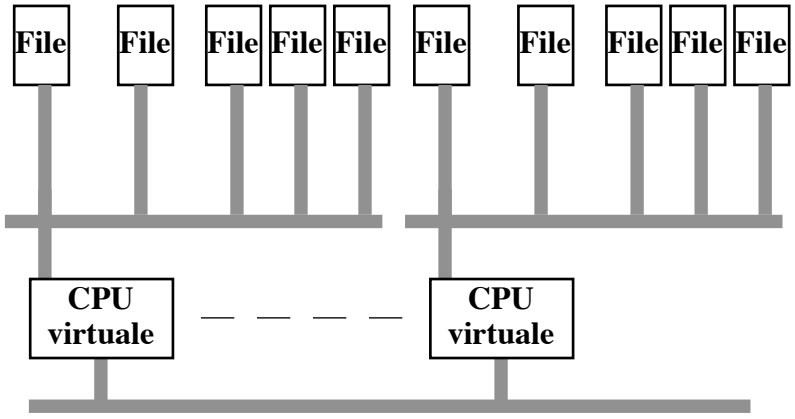


Figura 3.7. - Evoluzione del quinto livello.

Sesto e settimo livello: programmi di utilità e interprete comandi. Si tratta degli strati più esterni del sistema, direttamente visibili all'utente. Per esempio, facendo riferimento all'MS-DOS, il comando

COPY A:*.* C:

viene letto dall'interprete comandi il cui compito è di capire cosa l'utente vuole che il calcolatore faccia. Sulla base di tabelle ed algoritmi l'interprete identifica il comando

dato e i suoi eventuali parametri (mentre se il comando contiene errori è lo stesso interprete a segnalarlo) e attiva di conseguenza un processo che provvede a fare tutte le operazioni necessarie per leggere tutti i file presenti sul disco A e per copiarli sul disco C. I dischi in questo caso sono dunque gestiti da un programma, specializzato nella copia di file. Tale processo si avvale ovviamente del File System che a sua volta, attraverso il gestore delle periferiche si collega alle unità fisiche in questione, mentre il gestore della memoria alloca spazio per i dati letti da A, facendo attenzione che tale zona di memoria non venga utilizzata anche da altri processi.

3.2. CLASSIFICAZIONE DEI SISTEMI OPERATIVI

In questo paragrafo daremo una descrizione funzionale dei più comuni tipi di sistemi operativi.

a. **Sistemi dedicati**. Sono sistemi in cui la macchina è interamente dedicata all'uso da parte di un singolo utente: esiste cioè un unico terminale e l'utente è responsabile di tutto ciò che fa (Personal Computer). Non prevedono in genere alcun tipo di protezione né da errori né da manomissioni volontarie del sistema. L'unica protezione possibile, quando essi contengono dati e/o programmi che si vogliono mantenere riservati, è quella di rendere la macchina fisicamente inaccessibile agli utenti non autorizzati.[15] Non hanno veri e propri S.O.: i loro "sistemi operativi" (ad esempio MS-DOS) mancano spesso di alcune funzioni che caratterizzano i S.O.; per questo sono anche detti programmi di controllo o monitor.

b. **Sistemi a lotti (batch)**. Sistemi in cui tutte le informazioni venivano originariamente perforate su schede, o, più modernamente, su un supporto magnetico; il blocco di informazioni viene inserito tutto insieme nel calcolatore che elabora il tutto senza che l'utente possa intervenire. Il vantaggio principale di questa classe di S.O. consiste nel buon sfruttamento del calcolatore, perché l'utente è costretto a preparare in anticipo tutte le istruzioni e i dati di ingresso e il calcolatore esegue tutto fino alla fine senza mai fermarsi. Lo svantaggio è dato, per contro, dalla scarsissima interattività dal punto di vista dell'utente, che non può interagire in alcun modo con il programma durante la sua esecuzione. I sistemi a schede sono ormai praticamente abbandonati; le modalità batch sono tuttavia mantenute in alcuni sistemi operativi, per consentire l'uso del calcolatore anche in assenza dell'utente.[16]

---

[15] A questo scopo alcuni personal computer sono dotati di un interruttore a chiave che inibisce il funzionamento della tastiera, e che rende quindi impossibile l'avvio di qualunque operazione.

[16] Un uso tipico consiste nel far "girare" programmi di calcolo, che occupano la CPU per molto tempo, in orari in cui il calcolatore non ha molto lavoro (tipicamente di notte), in modo da non disturbare gli altri utenti, e di poter usufruire di tariffe di utilizzo ridotte.

c. **Sistemi interattivi**. Sistemi analoghi a quelli dedicati: l'unica differenza è che le operazioni permesse su un sistema dedicato ad un solo utente alla volta, ora sono utilizzabili da più utenti nello stesso istante; si tratta quindi di sistemi più grossi dei Personal Computer, dotati di svariati terminali, che possono dare notevoli vantaggi in termini di risparmio ed efficienza.

d. **Sistemi in tempo reale (Real Time)**. Il termine Real Time è talvolta ambiguo, perché non è rappresentabile da una misura temporale effettiva. L'idea è che la velocità di risposta di una macchina di questo tipo sia compatibile con le esigenze del processo controllato dalla macchina stessa, esigenze che però possono variare moltissimo da un caso a un altro. È richiesto quindi che a fronte di uno stimolo dell'ambiente si abbia una risposta in un tempo piccolo rispetto alle costanti di tempo proprie dell'ambiente esterno. Il tempo di risposta in sistemi di questo genere può quindi variare da qualche secondo (si pensi ad esempio ad un sistema di prenotazione di voli aerei in tempo reale), a pochi microsecondi (sistema di controllo di un missile).

e. **Sistemi transazionali e sistemi teleprocessing**. Sono sistemi interattivi caratterizzati dal fatto che su di essi gli utenti non sviluppano mai del software. Per esempio, sono sistemi transazionali i terminali bancari collegati ad un calcolatore. Le operazioni sul terminale riguardano l'utilizzo di funzioni che il software presente nel calcolatore mette a disposizione del singolo utente. Tali operazioni sono sempre "chiuse" (hanno cioè sempre un inizio, una fase centrale ed una fine) e sono dette per questo *transazioni*. Quando i terminali accessibili all'utente non sono direttamente collegati al calcolatore (come ad esempio nel caso degli sportelli bancomat), ma sono collegati tramite sistemi di comunicazione, parliamo di sistemi *teleprocessing*.

f. **Sistemi Uni- e Multi-programmati**. Un sistema si dice uniprogrammato se può far girare un solo programma per volta; si dice invece multiprogrammato se in ogni istante possono girare più programmi utente differenti. La quasi totalità dei sistemi operativi odierni permette la multiprogrammazione, anche se in genere l'utente può interagire con un solo programma alla volta.

g. **Sistemi speciali**. Come ogni classificazione, anche quella che abbiamo appena presentato non è in grado di dare una collocazione precisa a tutti i sistemi che possono essere costruiti. Riserviamo perciò questa classe a tutti gli altri tipi di sistemi operativi che non rientrano specificatamente in nessuna delle categorie precedentemente menzionate.

3.3. LA GESTIONE DELLE PERIFERICHE

Esamineremo in questo paragrafo come le periferiche possano essere gestite in modo da soddisfare le esigenze del sistema operativo. A titolo di esemplificazione, considereremo separatamente il caso di una semplice periferica di ingresso, e di una di uscita.

3.3.1. GESTIONE DELLE PERIFERICHE DI INGRESSO

La periferica, che supponiamo collegata secondo lo schema visto nel capitolo 1, genera interruzioni quando ha dati disponibili. Tali interruzioni vengono gestite come mostrato in figura 3.8.
La routine di gestione delle interruzioni acquisisce i dati dalla periferica e li inserisce in una coda (buffer) di N elementi. Il puntatore PIN punta in ogni istante al primo elemento libero del buffer. Il buffer è di norma gestito come se fosse circolare: in altre parole, dopo che è stato inserito un dato nel suo ultimo elemento, si ricomincia ad inserire dati nel primo.
L'estrazione dei dati dal buffer avviene nel modo seguente: se il puntatore POUT (inizialmente posto allo stesso valore di PIN) punta alla medesima posizione del buffer di PIN ciò significa che il buffer è vuoto; in caso contrario possiamo leggere i dati contenuti nel buffer incrementando POUT ogni volta. I dati sono quindi restituiti nello stesso ordine in cui sono entrati.
Il meccanismo illustrato permette di svincolare il processo fisico di ingresso dal processo di utilizzazione dei dati.
Se non ci sono dati, il programma che li attende non prosegue; aspetta che essi vengano introdotti, oppure può essere interrotto e il calcolatore fare qualcos'altro.
La routine di gestione ora descritta assomiglia alla gestione delle periferiche a controllo di programma, ma la differenza sostanziale sta nel fatto che la prima parte funziona comunque a interruzione.
Dal punto di vista del programmatore, due sono le primitive fondamentali del sistema operativo che permettono di acquisire dati da una periferica: una funzione che ritorna il numero di dati disponibili nel buffer al momento in cui viene chiamata, e una procedura che trasferisce in un vettore appartenente al programma di utente un certo numero di dati.

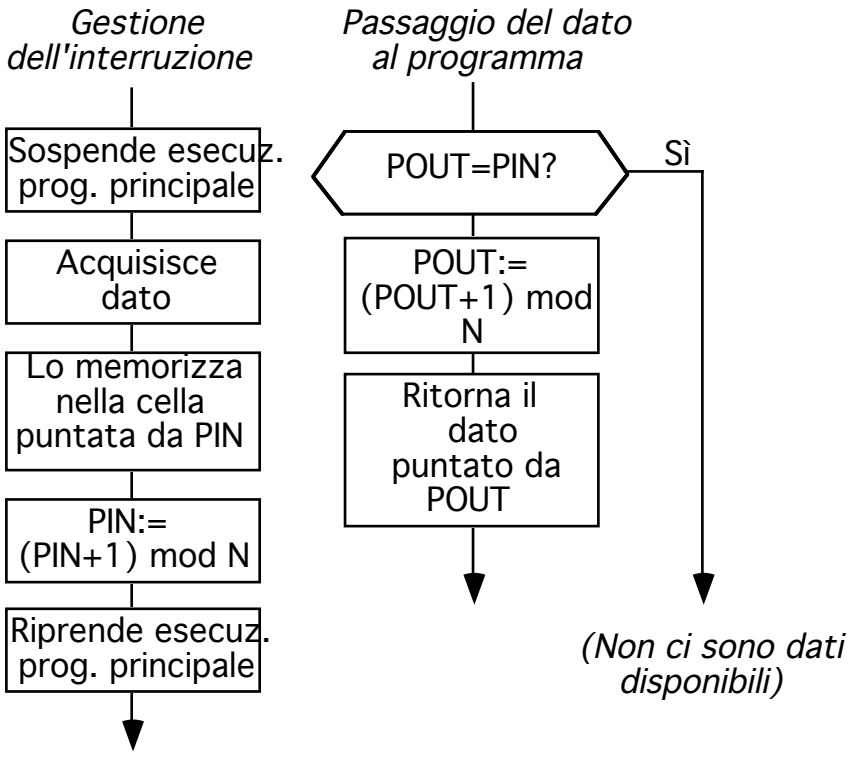


Figura 3.8. - Acquisizione delle informazioni da una periferica.

3.3.2. GESTIONE DELLE PERIFERICHE DI USCITA

La gestione delle periferiche in uscita ricalca in tutto e per tutto quella delle periferiche in ingresso. Le primitive fondamentali che il sistema operativo mette a disposizione in questo caso (Fig. 3.9) sono una procedura che spedisce n dati (forniti come parametro), e una che ritorna il numero di dati ancora presenti nel buffer (cioè non ancora trasferiti alla periferica).
In questo caso, il buffer contiene i dati da spedire alla periferica, e la routine di gestione delle interruzioni prende i dati dal buffer e li invia alla periferica.
La periferica genera interruzioni quando è pronta a ricevere i dati.

3.3.3. OSSERVAZIONI SULLE INTERRUZIONI

Fino a questo momento, abbiamo considerato interruzioni *singole*, che raggiungono cioè il calcolatore "una alla volta". Nella pratica, se il calcolatore è collegato a più periferiche, si può verificare il caso in cui una interruzione viene mandata mentre è in esecuzione la routine di gestione dell'interruzione di un'altra periferica.
In genere, le procedure di gestione delle interruzioni non sono a loro volta interrompibili. Il calcolatore cioè, una volta iniziata la procedura di gestione di una interruzione, ignora ogni altro segnale di interruzione finché la prima gestione non è terminata. Questo è dovuto alla struttura dell'hardware del calcolatore, che, nelle macchine più semplici, non è in grado di salvare lo stato della macchina durante la

gestione di una interruzione. Parliamo in questo caso di *interruzioni ad un solo livello*.
È però possibile, in macchine più sofisticate, definire diversi livelli di interruzione. Si può definire cioè un ordine di priorità nella gestione delle interruzioni: una interruzione è interrompibile solo da un'altra interruzione con priorità più alta.

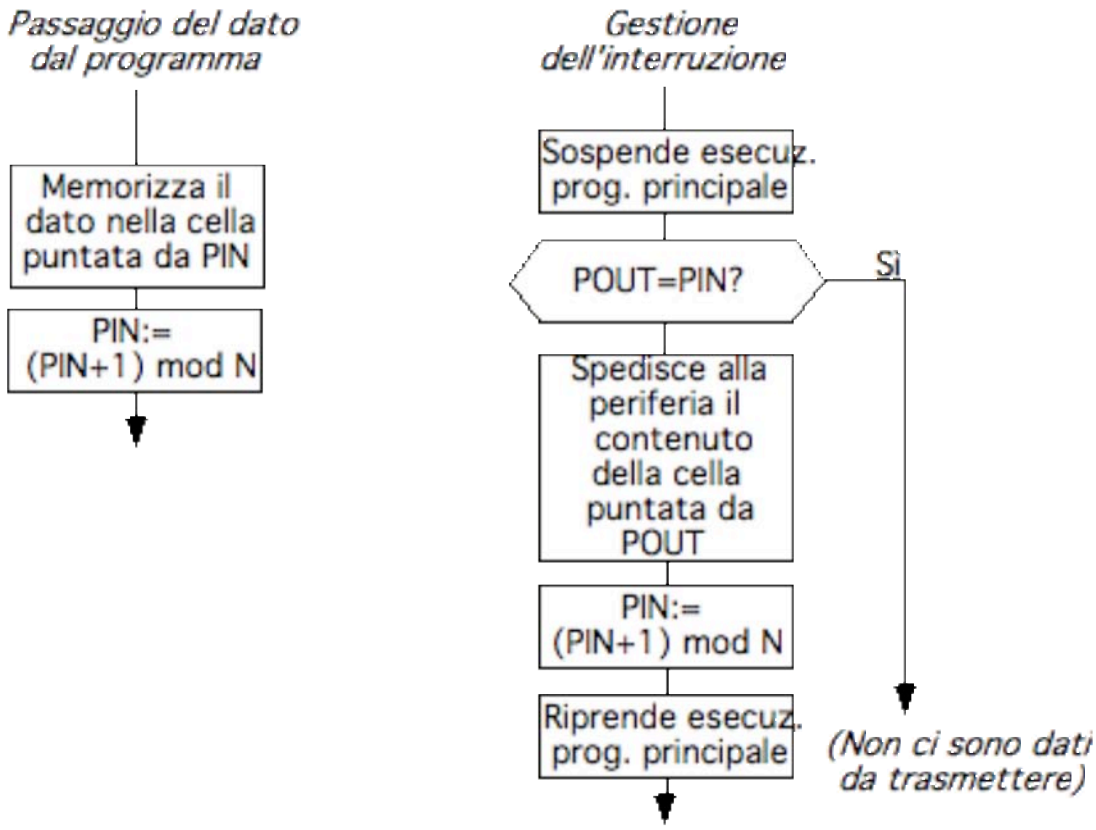


Figura 3.9. - Gestione di una periferica di uscita.

3.4. LA PROTEZIONE DEL SISTEMA

Consideriamo un sistema usato contemporaneamente da più utenti. È fondamentale, per evitare che errori contenuti in un programma di utente possano mettere a repentaglio l'esecuzione degli altri programmi, che vengano rispettati i seguenti punti:

- ogni processo può accedere solo alla memoria che gli è stata assegnata;
- ogni processo può accedere solo alle periferiche che gli sono state assegnate;
- le periferiche devono essere accessibili solo tramite primitive unificate del sistema operativo;

- alcune istruzioni macchina non devono poter essere eseguite.

3.4.1. IL PROBLEMA DELL'ACCESSO AL SISTEMA OPERATIVO

Il programma dell'utente utilizza procedure del sistema operativo ogni volta che deve accedere a risorse (come ad esempio le periferiche) il cui utilizzo diretto gli è negato dalle regole precedentemente esposte.
Tuttavia, a queste procedure il programma dell'utente non può accedere direttamente, utilizzando chiamate di sottoprogrammi. Questo infatti violerebbe la regola che impone ad ogni programma di utilizzare solo la memoria che gli è stata assegnata, e che è fondamentale per evitare che, in presenza di errori, il programma utente possa causare l'esecuzione di operazioni non desiderate e talvolta disastrose. Ciò potrebbe avvenire, ad esempio, chiamando una procedura con un indirizzo sbagliato, come si può vedere in figura 3.10. Una normale chiamata a sottoprogramma infatti implica che l'indirizzo del sottoprogramma da eseguire sia specificato nell'istruzione di chiamata; se, per una ragione qualsiasi, tale indirizzo fosse errato, il controllo potrebbe passare ad una qualunque altra istruzione del sistema operativo, con effetti assolutamente impredicibili e potenzialmente dannosi se non catastrofici.
Per questa ragione, ogni utilizzazione del sistema operativo da parte del programma dell'utente viene gestita come se fosse un'interruzione, sfruttando il fatto che le interruzioni causano il salto a locazioni di memoria stabilite dal sistema, e inalterabili da parte dell'utente. Queste interruzioni "software" vengono realizzate dall'istruzione macchina TRAP *n*, dove a ogni *n* corrisponde l'indirizzo di una particolare procedura o funzione del sistema operativo.

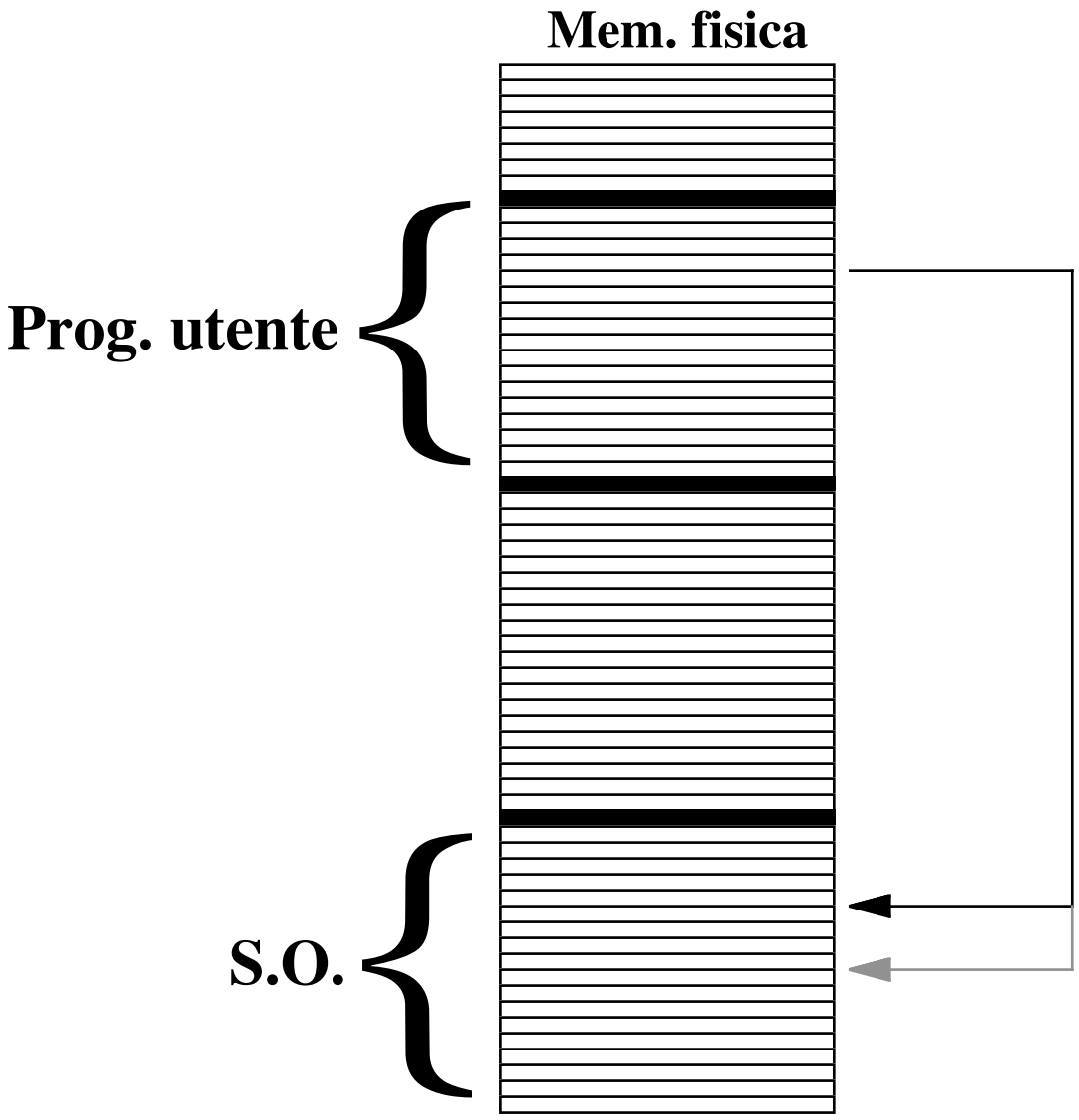


Figura 3.10. - Necessità della protezione degli indirizzi.

3.4.2. I MODI DI FUNZIONAMENTO DEL SUPERVISORE

Il sistema operativo dal canto suo deve poter disporre in maniera completa di tutte le risorse della macchina (memoria, periferiche, insieme delle istruzioni). Il calcolatore ha quindi almeno due distinti modi di funzionamento:

- modo **SUPERVISORE** (*SUPERVISOR*) in cui tutto è permesso, cioè è possibile indirizzare qualunque cella di memoria ed eseguire qualunque istruzione;
- modo **UTENTE** (*USER*): l'hardware, su istruzione del sistema operativo, si incarica di impedire alcuni tipi di operazione. Ogni tentativo di fare cose vietate genera una interruzione.

Le interruzioni sono sempre gestite dal sistema operativo.
Il sistema operativo funziona sempre in modo supervisore; prima di passare il controllo a un programma utente, pone la macchina in modo utente. Dal momento che non esistono istruzioni che permettono di passare dal modo utente al modo supervisore, il programma utente è costretto ad operare in questa modalità.
Quando si verifica un'interruzione (hardware o software), il calcolatore ritorna al modo supervisore, ma a questo punto il controllo è passato al sistema operativo, che la gestisce e, prima di ripassare il controllo al programma utente, riporta la macchina nello stato utente.

3.5. SFRUTTAMENTO DELLE RISORSE

Discutiamo in questo paragrafo alcuni aspetti relativi allo sfruttamento delle risorse dei calcolatori, con particolare riferimento all'esecuzione contemporanea di più processi. Se in memoria sono contemporaneamente presenti più processi, dal momento che uno solo di essi può essere in esecuzione ad un determinato istante, è fondamentale stabilire le modalità con cui il controllo viene passato dall'uno all'altro, cioè la cosiddetta *politica di scheduling*. Per una trattazione completa dell'argomento si rimanda ad un testo di Sistemi Operativi: ragioni di semplicità consigliano qui di definire lo stato di ogni processo come appartenente ad un insieme composto dai due stati *attivo* e *sospeso*.
Ogni processo può quindi essere o attivo, e quindi in esecuzione, o sospeso, cioè in attesa di essere eseguito.
È evidente che ad ogni momento, solo un processo può essere in stato attivo; tutti gli altri contemporaneamente presenti in memoria devono essere nello stato sospeso.

3.5.1. SISTEMA MULTIPROGRAMMATO CON N PROGRAMMI IN MEMORIA

Quando un processo si ferma per via dei tempi di attesa legati all'utilizzo delle periferiche il calcolatore lo sospende, e attiva uno dei processi rimanenti (figura 3.11). Se (come spesso accade) tutti i processi sono in attesa di una periferica, il calcolatore rimane ovviamente in uno stato di attesa.

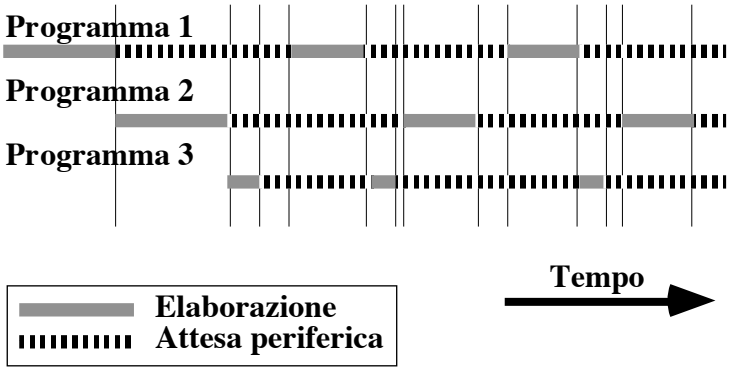


Figura 3.11. - Comportamento di un sistema multiprogrammato.

3.5.2. SISTEMA A DIVISIONE DI TEMPO (*TIME SHARING* O *TIME SLICING*)

I sistemi multiprogrammati non impediscono che il calcolatore sia costretto a riservare tutto il suo tempo ad un unico programma che presenta errori di programmazione come, ad esempio, cicli senza uscita. Infatti, dal momento che il passaggio del controllo da un processo ad un altro avviene solo quando un processo si pone in attesa di una periferica, un programma che impegni in continuazione l'unità centrale, ad esempio quando esegue un'istruzione del tipo

```
while true do;
```

non verrebbe mai sospeso. Tutti gli altri programmi resterebbero indefinitamente in attesa e, come si dice in termini tecnici, morirebbero per fame (*starvation*).
Tutto questo risulta particolarmente grave nei sistemi multiutente, perché un unico programma sbagliato può bloccare del tutto la macchina. Per ovviare a questo inconveniente a volte si preferisce ripartire equamente tra i programmi il tempo ad essi dedicato dal calcolatore, stabilendo che il passaggio del controllo da un programma all'altro avvenga ad intervalli di tempo fissi.
Per far questo, occorre avere a disposizione una particolare periferica, detta orologio in tempo reale (*real time clock*), che genera interruzioni ad intervalli costanti (tipicamente 50 o 60 volte al secondo). Ogni volta che arriva una interruzione da tale periferica, il sistema operativo sospende il processo attivo in quel momento, ed attiva il successivo. Quando viene sospeso l'ultimo processo, viene di nuovo attivato il primo, e il meccanismo si ripete (Fig. 3.12).
Il passaggio del controllo da un programma all'altro è quindi indipendente dagli accessi alle periferiche, e non può essere bloccato da un errore di programmazione, come accadeva nel caso precedente.

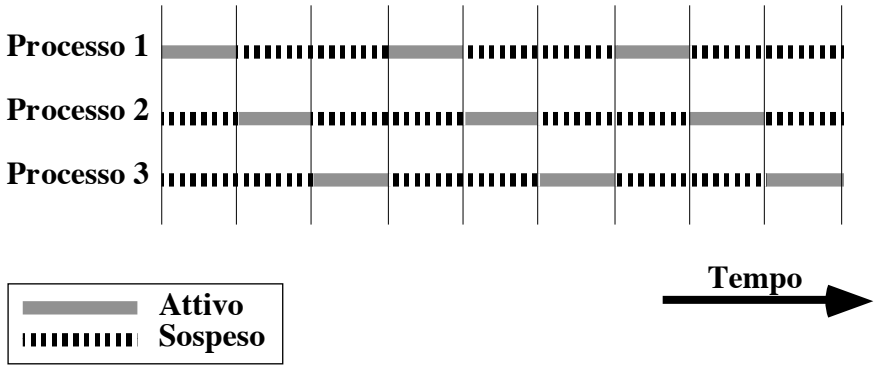


Figura 3.12. - Un sistema a divisione di tempo.

Questo metodo di gestione peggiora però le prestazioni del sistema, in quanto non tiene conto del tempo perso dai programmi in attesa di risposta da parte delle periferiche.

3.5.3. SISTEMA MISTO

È ottenuto fondendo i due sistemi appena visti, allo scopo di eliminare i difetti menzionati. In questo metodo di gestione ogni programma perde la priorità di utilizzo del calcolatore sia se è in attesa della disponibilità di una periferica, sia se è scaduto il tempo a sua disposizione (Fig. 3.13).

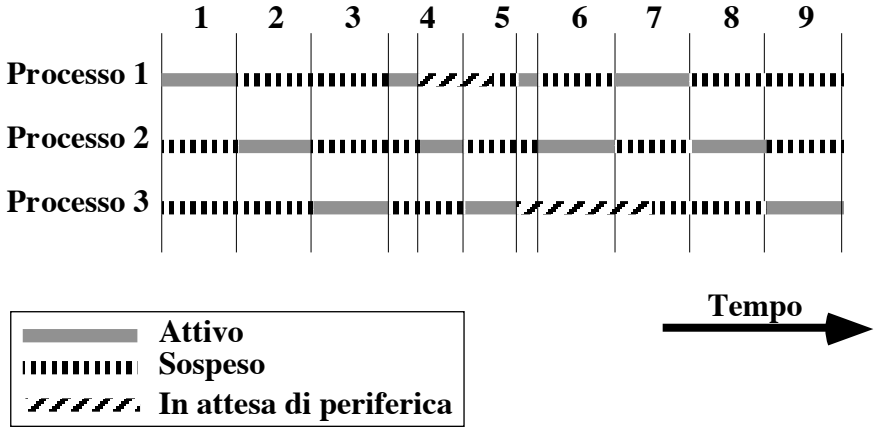


Figura 3.13. - Un sistema misto.

In questo esempio, nei primi tre quanti di tempo il sistema si comporta secondo lo schema del time sharing. Durante il quanto 4, il processo 1 va in attesa di una periferica, per cui il controllo viene immediatamente passato al processo 2. Allo scadere del quanto 4 il controllo passa al processo 3, che però va anch'esso in attesa

di una periferica. Il controllo viene quindi ripassato al processo 1 il quale, avendo nel frattempo ottenuto la disponibilità della periferica, può ripartire. Allo scadere del quanto 6 il controllo dovrebbe passare al processo 3, ma, siccome quest'ultimo è ancora in attesa di periferica, viene invece ripassato al processo 1.
Si noti come questo metodo permette uno sfruttamento dell'unità centrale pari a quello del primo metodo visto, ma senza averne gli inconvenienti. Nella pratica tuttavia può essere desiderabile avere politiche di gestione ancora più sofisticate, che introducano il concetto di *priorità* dei processi. Queste politiche non formano però oggetto di questa trattazione.

3.6. LA COMUNICAZIONE E LA SINCRONIZZAZIONE FRA PROCESSI

Finora abbiamo visto alcuni modi con cui un singolo calcolatore riesce a gestire più processi, come se evolvessero in parallelo. Se la commutazione fra un processo e l'altro è abbastanza frequente (svariate volte al secondo), all'utente sembrerà che questi processi evolvano contemporaneamente.
Ci occuperemo ora di come due o più processi possano comunicare fra di loro. In realtà, è abbastanza intuitivo pensare che due o più processi che girano parallelamente su un calcolatore debbano per un motivo o per l'altro scambiarsi delle informazioni. Infatti, sarebbe illogico pensare che ogni processo sia una unità a sé stante, e che non abbia nessuna relazione con gli altri processi che girano sullo stesso calcolatore. Ad esempio, il processo che tiene aggiornata la data e l'ora sul calcolatore deve in qualche modo essere in grado di poter comunicare a qualche altro processo che giorno e che ora è. Perché questo sia possibile, occorre sia un meccanismo che decide istante per istante qual è il processo che deve girare, sia un meccanismo che permetta a questi processi di scambiarsi informazioni.
Da quanto abbiamo studiato fino ad ora, potremmo affermare che la comunicazione fra due processi sia impossibile, dato che ogni processo può "vedere" solamente una parte della memoria, che è assegnata a lui e a nessun altro; può vedere solamente le periferiche che sono assegnate a lui, e nessun altro processo può accedervi. Quindi due processi non hanno nessun mezzo comune per scambiarsi informazioni, perché sono, per quello che abbiamo visto, indipendenti.
Due ordini di problemi nascono da questo: occorre creare meccanismi che permettano lo scambio di informazioni fra processi scritti dall'utente, e meccanismi per la *sincronizzazione* dei processi. Per capire la problematica usiamo la metafora che segue.
Come vediamo in figura 3.14, due robot, A e B, devono spostare degli oggetti, inizialmente posizionati a sinistra di A, nelle posizioni tratteggiate poste a destra della macchina B. Nella disposizione dei robot indicata, il robot A è in grado di afferrare gli oggetti, perché il suo braccio arriva nella zona dove sono posizionati inizialmente, ma non è in grado di raggiungere la zona dove vanno depositati. Il robot B, dal canto suo, è in grado di raggiungere queste posizioni, ma non può prendere gli oggetti da

dove originariamente sono collocati. Per risolvere il problema bisogna fare in modo che i due robot si aiutino a vicenda (Fig. 3.15).

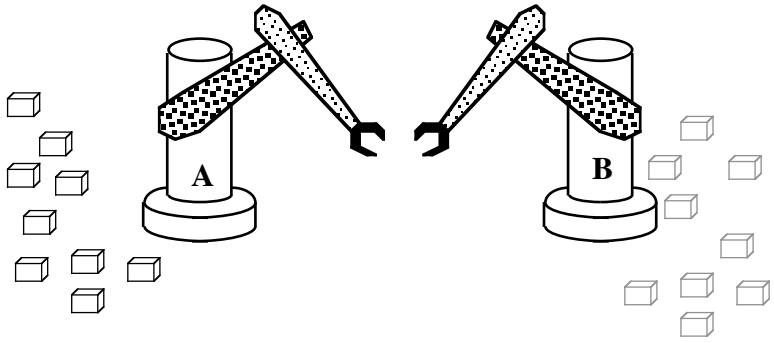


Figura 3.14. - La metafora dei robot.

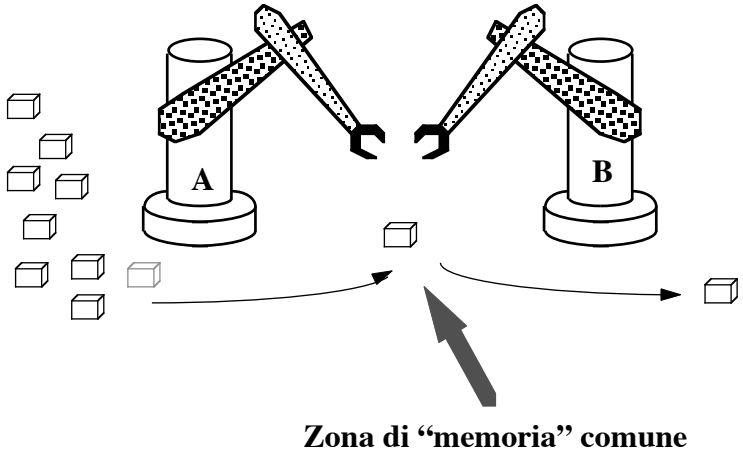


Figura 3.15. - La soluzione del problema.

Il robot A posizionerà gli oggetti al centro (zona intermedia: punto accessibile ad entrambi i bracci dei robot), e il robot B li porterà dove devono essere collocati. Per applicare la metafora alla nostra trattazione, basta pensare che A e B non siano dei robot, ma dei processi, e che gli oggetti da spostare siano informazioni. I due processi sono tali che uno dei due “produce” i dati, e l’altro processo li “consuma”, cioè li utilizza (ad esempio per stamparli). I due processi hanno in comune i dati che devono passarsi. Nel calcolatore l’equivalente del tavolo su cui si appoggiano gli oggetti dell’esempio dei robot è una memoria che ha la particolarità di poter essere accessibile sia al processo A che al processo B. Il sistema dovrà dunque essere fatto in modo tale che ogni processo possa accedere sia alla sua zona di memoria “privata”, sia ad una zona di memoria comune che prende il nome di *memoria globale* (che ogni processo che gira nel calcolatore può leggere o scrivere).

Il fatto che più processi possano accedere ad una stessa zona di memoria è critico, perché tale zona può essere in qualsiasi momento alterata da qualunque altro processo, causando così la crisi del sistema.[17] Possiamo provare a scrivere il programma che i due robot devono eseguire, e che è rappresentato schematicamente in figura 3.16.

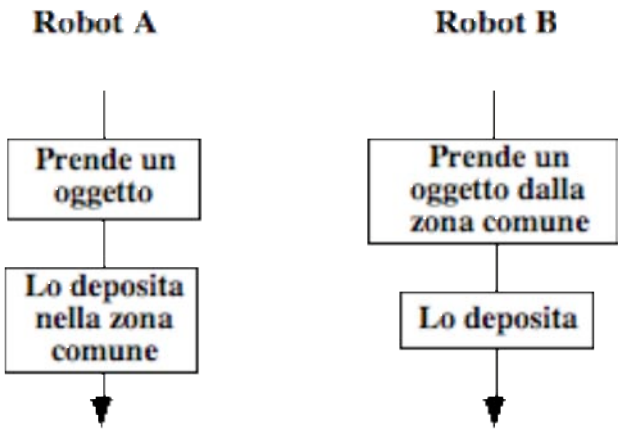


Figura 3.16. - I programmi dei due robot.

Evidentemente, per risolvere il problema proposto, i due programmi devono essere eseguiti ciclicamente. Tuttavia, se il robot A è più veloce del robot B, ad un certo punto si accumuleranno più oggetti nella zona intermedia, mentre, se il robot B è più veloce del robot A, la zona intermedia resterà vuota. Inoltre, i due bracci corrono costantemente il rischio di scontrarsi fra loro. Occorre quindi un metodo per coordinare le azioni dei due robot o, per usare un termine più tecnico, per *sincronizzare* i due processi. Così A e B non evolvono più liberamente, ma in alcuni istanti l'azione di uno dei robot deve essere fermata, fino a quando l'altro non abbia compiuto il suo ciclo (Fig. 3.17).

[17] Vedremo più avanti che, con il meccanismo dei messaggi, tale problema può essere eliminato.

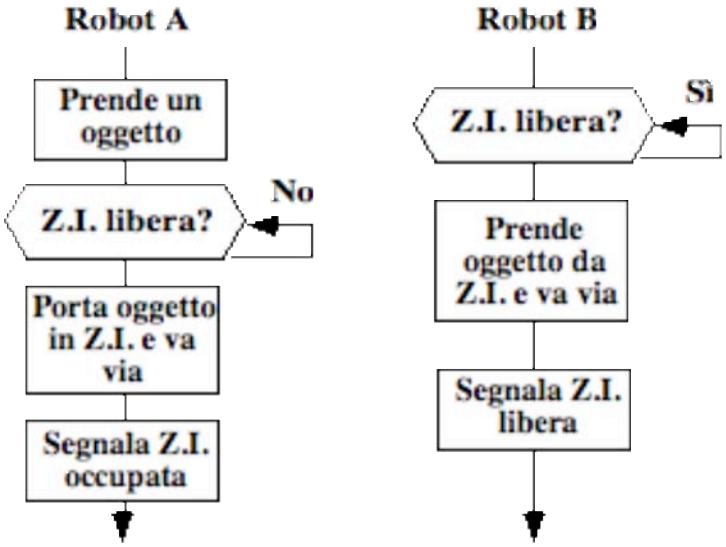


Figura 3.17. - I programmi dei due robot, sincronizzati.

Il robot A deve verificare che la zona intermedia sia libera prima di deporvi un oggetto, e segnalare al robot B quando tale zona è occupata. Il robot B invece deve accertarsi che la zona intermedia sia occupata prima di iniziare il ciclo, e segnalare al robot A che essa è libera quando ha afferrato l'oggetto. La stessa cosa avviene fra due processi che si scambiano dati.
Questa comunicazione prevede lo scambio di una informazione che prende il nome di *semaforo binario*.
La segnalazione dello stato della zona intermedia è binaria perché esistono solo due possibilità: la zona intermedia o è libera o è occupata. Indichiamo lo stato di zona intermedia libera con 1, quello di zona intermedia occupata con 0. Il robot A segnalerà che la zona intermedia è occupata ponendo a 0 il semaforo, e viceversa B segnalerà che essa è libera ponendolo a 1.
Il problema appena discusso, noto come *produttore-consumatore*, era stato implicitamente visto in precedenza. Ad esempio, la routine di gestione delle interruzioni di una periferica di ingresso è un processo produttore, perché "produce" dati e li mette nel buffer, mentre il programma che aspettava questi dati è il processo consumatore. Anche in questo esempio avevamo visto, senza metterla in evidenza, la presenza di un semaforo, perché esisteva un test che verificava se i due puntatori PIN e POUT erano uguali. La risposta a questa domanda può essere ovviamente solo o sì o no, cioè binaria.
Nel caso dei due robot, abbiamo ipotizzato che nella zona intermedia ci possa essere solo un oggetto per volta, mentre nel caso delle periferiche avevamo assunto che il buffer potesse contenere contemporaneamente più dati. Concettualmente però non ci sono differenze: nel caso dei robot invece di considerare un solo oggetto, si potrebbe considerare una "coda" di oggetti, dove il primo messo nella fila, sarà il primo ad essere afferrato dal braccio del robot B.
Se abbiamo un meccanismo di sincronizzazione, come può essere il semaforo binario, una differente velocità dei processi non crea alcun problema. Occorre però tener

presente che esistono alcuni casi in cui la velocità del produttore è fissata da fattori che sono al di fuori del controllo del programmatore. Ad esempio, se il produttore è una telecamera, essa produrrà un flusso continuo di dati che non può essere rallentato. In questo caso il problema può essere risolto solo trovando un consumatore veloce almeno tanto quanto il produttore.
Se invece il processo produttore (come accade per esempio coi dischi magnetici) produce blocchi di dati non interrompibili, ma può essere fermato fra un blocco e l'altro, il problema è solo quello di disporre di una sufficiente quantità di memoria da utilizzare come buffer.
Supponiamo per esempio che il produttore produca blocchi non interrompibili di 100 dati in un secondo, mentre il consumatore può elaborarne solo 10. Sarà sufficiente disporre di un buffer di 100 celle. In questo si possono far produrre 100 dati al produttore, e poi bloccarne l'esecuzione per 9 secondi.
Per finire, osserviamo che non è possibile costruire un sistema produttore-consumatore senza un sistema di sincronismo perché il tempo di esecuzione dei diversi processi non può mai essere noto con sufficiente precisione.

3.7. IL CONCETTO DI MESSAGGIO

Il meccanismo dei semafori appena visto presenta l'ovvio svantaggio di richiedere una zona di memoria comune a più processi. Un errore nella gestione di questa zona può causare malfunzionamenti nel sistema, tanto più probabili quanto maggiore è il numero di processi che cooperano. Per eliminare questo problema, possiamo pensare ad un meccanismo di sincronizzazione che sia gestito interamente dal sistema operativo.
Questo meccanismo, che implica il concetto di *messaggio*, è indicato in figura 3.18. La comunicazione fra i due robot è identica a quella del caso precedente ma, invece che scrivendola su una zona di memoria comune, essa viene comunicata attraverso "lettere" consegnate da un "postino", che nel nostro caso è il sistema operativo. Ogni processo quindi comunica solo con il sistema operativo, scambiando dati solo con esso. L'esistenza di un meccanismo per lo scambio di messaggi elimina la necessità di disporre di una memoria comune.

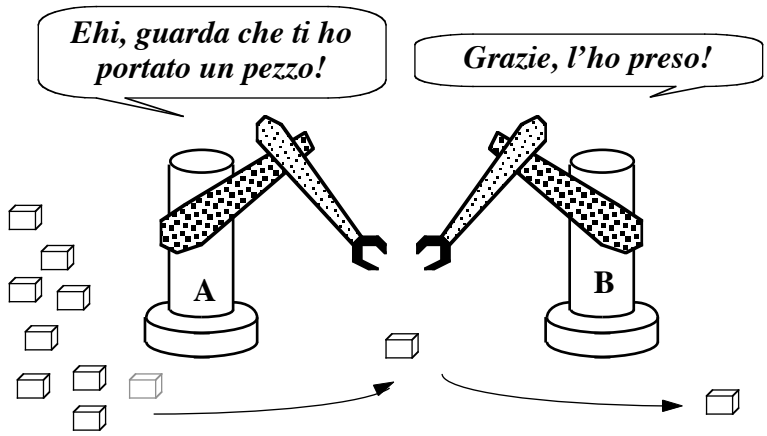


Figura 3.18. - I messaggi usati come primitive di sincronizzazione.

Ogni messaggio contiene quattro tipi di informazione:

- tipo del messaggio;
- destinatario;
- mittente;
- parametri.

Il tipo di messaggio è l'informazione fondamentale, perché contiene il significato del messaggio. L'indicazione del destinatario viene utilizzata dal sistema operativo per il recapito. L'indicazione del mittente viene utilizzata dal destinatario per una eventuale risposta. I parametri sono dati che completano, istanziandola, l'informazione contenuta nel tipo di messaggio. Il mittente, il destinatario e i parametri possono in alcuni casi mancare. Il mittente può mancare perché, per molti tipi di messaggio, al destinatario non interessa chi l'ha mandato. Il destinatario può mancare perché in alcuni casi il messaggio è indirizzato a tutti gli oggetti che possono riceverlo (messaggio *broadcast*). I parametri possono mancare, perché per alcuni messaggi tutta l'informazione necessaria è contenuta nel tipo.
Il meccanismo di scambio dei messaggi sarà largamente ripreso nei capitoli successivi, perché costituisce, oltre ad un sistema utile per la sincronizzazione dei processi, anche un metodo estremamente efficace per la scrittura dei programmi.

3.8. L'ALLOCAZIONE DELLE RISORSE

Un altro problema dei sistemi operativi è quello della allocazione delle risorse. Supponiamo che un processo abbia bisogno di una certa risorsa (ad esempio di una stampante). Se la risorsa è disponibile, il sistema operativo provvede ad assegnarla al processo che ne ha fatta richiesta. Ora, se un secondo processo, mentre il primo sta già stampando, richiede anch'esso la stampante, il sistema operativo risponderà

dicendo che la stampante è già occupata e sospenderà il secondo processo in attesa che la stampante si liberi, a meno che non esista una seconda stampante.
Il sistema operativo si incarica quindi di distribuire le risorse disponibili, cioè di allocarle, nella maniera migliore a chi ne richiede l'uso. Quando ad un processo si è fornita una risorsa, non si può togliergliela, fino a che non abbia finito di utilizzarla, cioè fino a che non sia il processo stesso a cederla. Per risorse intendiamo, come abbiamo già detto, memoria, periferiche, tempo di calcolo ecc.
La allocazione delle risorse è in linea di principio un problema semplice: occorre che il sistema operativo mantenga aggiornate delle tabelle, su cui sono segnate le allocazioni correnti di ogni risorsa; quando i programmi richiedono l'accesso a tali risorse, il sistema provvede al trasferimento dei dati da e verso di esse. Nella pratica sorgono però diversi problemi di cui ora ci occuperemo.
Per illustrare le problematiche relative la trattazione dei sistemi operativi fa riferimento ad un problema classicamente noto come *problema dei cinque filosofi.*

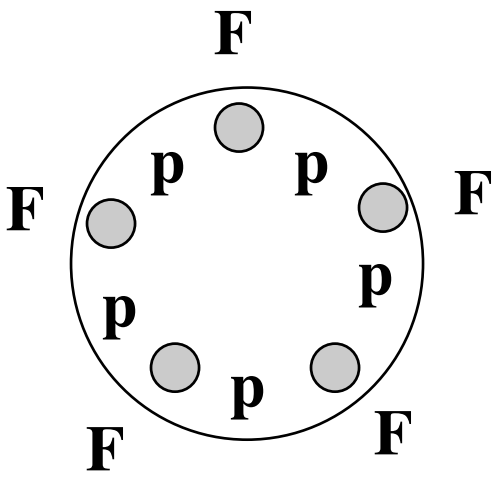


Figura 3.19. - Il problema dei cinque filosofi (versione classica).

Ci sono cinque filosofi che siedono ad un tavolo (Fig. 3.19), e hanno di fronte ad ognuno di loro un piatto di spaghetti. Sul tavolo ci sono anche cinque posate. I filosofi in questione sono personaggi abbastanza schematici, nel senso che sanno fare solo due cose, mangiare o pensare. Non possono fare le due cose contemporaneamente. Inoltre per mangiare hanno bisogno di usare due posate contemporaneamente, e ognuno di loro è in grado di usare solo le posate che ha immediatamente alla sua destra e sinistra.
La prima considerazione che possiamo fare è che intorno al tavolo non più di due filosofi contemporaneamente possono mangiare, dal momento che ci sono cinque posate.

Questa metafora serve per illustrare il problema del cosiddetto *blocco critico*. Per semplificare le cose considereremo il problema con due soli filosofi e due sole posate (Fig. 3.20).

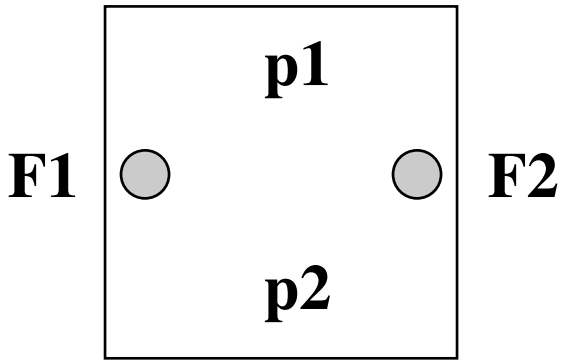


Figura 3.20. - Il problema dei cinque filosofi (versione semplificata).

Utilizzando le regole viste per il problema dei cinque filosofi, il problema si riduce al seguente: se ad un certo momento un filosofo decide che ha fame, prende la posata 1 e la posata 2, e mangia. Se a questo punto il filosofo 2 decide che ha fame, non può mangiare fino a che il filosofo 1 non abbia finito. Questo nonostante le apparenze non è un problema, perché prima o poi il filosofo 1 sarà sazio, e così il filosofo 2 potrà mangiare. Ma se il filosofo 1 è ingordo e non finisce più di mangiare, il filosofo 2 non avrà mai l'opportunità di mangiare. In inglese questo problema si chiama *starvation* (morte per fame).
Un problema ancora più grave è il seguente: inizialmente i filosofi pensano, ad un certo punto il filosofo F1 decide che ha fame e prende la posata p1; contemporaneamente anche il filosofo F2 decide che ha fame e prende la posata p2. A questo punto cosa succede? Tutti e due i filosofi muoiono di fame, perché per mangiare hanno bisogno di due posate. L'unico modo per evitare ciò sarebbe che uno dei due lasciasse la posata, cosa che contraddice il principio che una volta che il sistema operativo assegna una risorsa (posata) ad un processo non può più togliergliela fino che esso non ha finito. Questa è la situazione di *blocco critico* del sistema. Situazione che si verifica quando un processo A ha bisogno di una risorsa X che è allocata ad un altro processo B che è a sua volta in attesa di una risorsa Y che è allocata al processo A. Per evitare il blocco critico occorre fare in modo che le richieste delle risorse siano fatte in blocco e che non possano essere interrotte. Prendendo come esempio i due filosofi, le risorse sono le due posate. Se il filosofo F1 decide che ha fame, deve richiedere al "sistema operativo" le risorse in blocco, cioè la posata p1 e subito dopo la posata p2. Mentre il filosofo F1 chiede le risorse, il filosofo F2 permane in uno stato di sospensione, e non può intervenire nella richiesta delle risorse evitando che si verifichi il problema del blocco critico.

3.9. PROBLEMI ED ESERCIZI

Problema 3.1.*
Scrivere un programma Pascal che simuli una routine di risposta ad interruzione, usando una struttura a stack.

Problema 3.2.*
Una interruzione da una periferica è un evento sincrono o asincrono?

Problema 3.3.*
Le interruzioni generate dal real time clock sono sincrone od asincrone?

Problema 3.4.*
Una SVC è sincrona od asincrona?

Problema 3.5.*
Da chi è fissato l'indirizzo di una routine di risposta ad un'interruzione?

Problema 3.6.*
Cosa viene salvato dall'hardware al verificarsi di un'interruzione?

Problema 3.7.*
Al verificarsi di un'interruzione asincrona il processo in esecuzione in quale stato passa?

Problema 3.8.*
Al verificarsi di un'interruzione sincrona il processo in esecuzione in quale stato passa?

Problema 3.9.*
Quando le interruzioni sono disabilitate cosa succede alle richieste di interruzione?

Problema 3.10.
È possibile che, in un sistema time-sharing, un processo non possa mai girare perché gli altri consumano tutto il tempo di calcolo a disposizione?

Problema 3.11.*
È possibile costruire un sistema comprendente un processo produttore e un processo consumatore senza utilizzare un meccanismo di sincronizzazione fra i due processi?

Problema 3.12.
I buffer delle periferiche sono code o pile?

Problema 3.13.*
I semafori binari servono per sincronizzare più processi o per scambiare informazioni fra processi?

Problema 3.14.*
Il processo A, a cui è allocata la risorsa X, ha bisogno anche della risorsa Y, che è attualmente allocata al processo B (in normale evoluzione). Quindi, A non può evolvere finché la risorsa Y non diventerà disponibile.
Il sistema è in una situazione di blocco critico?

Problema 3.15.*
Il processo A, a cui è allocata la risorsa X, ha bisogno anche della risorsa Y, che è attualmente allocata al processo B, il quale a sua volta ha bisogno della risorsa X.
Il sistema è in una situazione di blocco critico?

Problema 3.16.*
La politica del *time sharing* serve principalmente:
a. a permettere a vari processi di girare "quasi parallelamente" sullo stesso calcolatore;
b. ad eliminare i conflitti nell'allocazione delle risorse;
c. a consentire una corretta gestione del flusso di dati in ingresso e in uscita dal calcolatore.

Problema 3.17.
Più grandi sono i *time slice*, più aumenta l'apparente parallelismo di esecuzione dei processi in un sistema time-sharing?

Problema 3.18.
Un messaggio può essere sempre sostituito da un semaforo binario?
a. vero;
b. falso;
c. vero, a condizione che il sistema operativo metta a disposizione una zona di memoria condivisa.

Problema 3.19.*
Un semaforo binario può essere sempre sostituito da un messaggio?
a. vero;
b. falso;
c. vero, a condizione che il sistema operativo metta a disposizione una zona di memoria condivisa.

4

IL COLLEGAMENTO FRA CALCOLATORI

Con l'evoluzione delle tecnologie informatiche assume una importanza sempre più grande il problema della trasmissione di dati a distanza. Fino a qualche anno fa, il problema si identificava in genere con la necessità di collegare i calcolatori (che erano spesso grandi, costosi e costruiti per servire più utenti contemporaneamente) con terminali che potevano essere posti, a seconda dei casi, nelle loro immediate vicinanze, in altre parti dello stesso edificio o in altri luoghi, collocati anche a grande distanza.
Oggi, con la diminuzione del costo dei calcolatori, si tende a distribuire la potenza di calcolo fra i vari utenti, dotando ognuno di essi di un calcolatore personale di piccola potenza.[18] Ciò non toglie che rimanga, ed anzi si vada accentuando sempre di più, l'esigenza di collegare insieme questi calcolatori per poter condividere dati, programmi e attività lavorative.
È quindi importante conoscere i principi che stanno alla base del collegamento a distanza più o meno grande dei sistemi digitali, avvertendo però che la trattazione completa della materia è estremamente più vasta, e coinvolge anche settori che hanno relativa attinenza allo scopo di questo libro, come ad esempio le telecomunicazioni.

[18] Parlare di "piccola potenza" nel caso dei personal computer non è in realtà molto corretto: la capacità di calcolo e di memoria di un PC odierno è spesso pari, se non superiore, a quella di un grosso calcolatore di qualche anno fa, mentre il suo prezzo è centinaia, se non migliaia di volte inferiore.

4.1. RETI GEOGRAFICHE E RETI LOCALI

Anticipiamo qui una definizione che utilizzeremo più avanti, e che permette di fare una distinzione basata in sostanza sulla estensione dei collegamenti da considerare. Dato un insieme di calcolatori (o, più in generale, di macchine digitali) da collegare insieme, parleremo di *rete locale* se la distanza complessiva da percorrere è piccola, mentre faremo riferimento a una *rete geografica* se essa è grande.
Definire i termini *piccola* e *grande* è, in questo caso, piuttosto arduo: per la nostra trattazione potremo comunque considerare piccole le distanze per cui è possibile operare collegamenti diretti, senza cioè dover fare ricorso a particolari dispositivi per il condizionamento dei segnali da trasmettere.

4.2. PARAMETRI CARATTERIZZANTI I COLLEGAMENTI

È possibile definire un gran numero di parametri ognuno dei quali caratterizza un particolare aspetto del collegamento. Per non appesantire inutilmente la trattazione, ci limiteremo qui a citare quelli fondamentali. Altri parametri saranno introdotti più avanti, quando se ne presenterà l'occasione.
I parametri fondamentali sono:

1. il tipo di collegamento: punto a punto, multipunto, ecc.;
2. la modalità di collegamento: unidirezionale o bidirezionale;
3. la lunghezza del collegamento;
4. il tipo di mezzo trasmissivo: linea semplice, cavo coassiale, linea telefonica, radio, ecc.;
5. la velocità di trasmissione;
6. gli standard elettrici.

Quasi tutti i termini appena citati hanno un significato ovvio, e saranno discussi più avanti. Diamo invece una caratterizzazione migliore al quinto punto, cioè alla velocità di trasmissione.
Essa è definita come la quantità di informazione che può essere trasmessa nell'unità di tempo, ed è espressa, a seconda dei casi, in bit/s oppure in byte/s.
Evidentemente, è valida la relazione

$$1\frac{\text{byte}}{s} = 8\frac{\text{bit}}{s}$$

Osserviamo che i bit/s vengono anche chiamati *baud*, dallo scienziato francese Baudot che fu l'inventore della telescrivente, il primo sistema di trasmissione digitale

delle informazioni completamente automatico.[19] Una velocità di trasmissione di 9600 baud equivale pertanto a 9600 bit/s, cioè a 1200 byte/s.
Le velocità di trasmissione possono variare enormemente a seconda dei sistemi impiegati: si va da poche decine di bit/s a centinaia di Mbyte/s.
Per quanto riguarda gli standard elettrici, la necessità di collegare fra loro apparecchi di marche diverse ha portato alla definizione di numerose "raccomandazioni", emesse da organi internazionali di standardizzazione, che definiscono i livelli dei segnali elettrici, le temporizzazioni, i tipi di connettori da usare, ecc. Due standard molto noti agli utenti di personal computer sono ad esempio RS232C e RS422, che definiscono le caratteristiche di due diversi sistemi per i collegamenti seriali.

4.3. COLLEGAMENTI PUNTO A PUNTO

Intendiamo con questa denominazione i collegamenti in cui i dati emessi da un trasmettitore raggiungono un unico ricevitore. I collegamenti punto a punto possono essere unidirezionali (i dati viaggiano cioè solo in un senso) o, molto più frequentemente, bidirezionali (le due unità collegate si comportano cioè, a seconda dei casi, come trasmettitori o come ricevitori).
Per i nostri scopi, possiamo vedere ogni collegamento bidirezionale come una coppia di collegamenti unidirezionali operanti in senso inverso l'uno rispetto all'altro.

4.3.1. COLLEGAMENTI PARALLELI

Le informazioni nei calcolatori sono in genere memorizzate come gruppi di bit. La lunghezza di questi gruppi può essere qualunque, ma si è ormai, per diverse ragioni, standardizzata. Abbiamo perciò lunghezze di 8 bit o di multipli di 8 bit (in genere 16, 24 o 32 bit).
È ovviamente possibile trasmettere tutti i bit di ogni gruppo simultaneamente, a condizione di disporre di un numero sufficiente di linee.
Le modalità di trasmissione sono identiche a quelle a suo tempo viste per il collegamento delle periferiche: occorrono quindi tante linee quanti sono i bit da trasmettere contemporaneamente, più una per lo strobe, più una per il livello di riferimento di tensione (linea di massa).
I collegamenti paralleli vengono usati solo su brevissime distanze (dell'ordine di pochi metri al massimo). Ciò è dovuto sia a motivi di ordine elettrico (degradazione del segnale su distanze più lunghe), sia, soprattutto, al costo della linea di trasmissione, che deve essere composta da un gran numero di fili.

---

[19] L'alfabeto Morse, infatti, richiedeva una interpretazione che, a quei tempi, poteva essere effettuata solo dall'uomo.

4.3.2. COLLEGAMENTI SERIALI

Per i motivi appena esposti, risulta in genere più conveniente trasmettere le informazioni un bit alla volta, cioè in modo *seriale*. Questo metodo, pur essendo più lento del precedente, offre il grande vantaggio di richiedere un numero molto più limitato di linee.

4.3.2.1. La trasmissione seriale sincrona

Supponiamo di dover trasmettere il byte 01100101. Possiamo pensare, utilizzando un apposito circuito, di disporre i suoi bit in sequenza (cominciando dal meno significativo) e di trasmetterli su un filo elettrico che trasporterà quindi un segnale come quello mostrato nella parte alta della figura 4.1. Ovviamente, per permettere la ricezione di questo segnale, dovremo trasmetterne un secondo (in basso in figura 4.1), analogo al segnale di strobe della trasmissione parallela, e che in questo caso prende il nome di *clock*. Questo secondo segnale serve ad indicare gli istanti in cui il primo è valido, e può quindi essere acquisito e, mediante un circuito che opera in senso inverso a quello del trasmettitore, ritrasformato nella forma parallela originale.
Così facendo, abbiamo bisogno di tre soli fili: quello che porta il segnale, quello che porta il clock, e il filo di massa.
Molto spesso però anche questa soluzione non è soddisfacente, perché i fili a disposizione sono solamente due. È però possibile "conglobare" il segnale di clock all'interno del segnale dei dati. Ciò può essere fatto in varie maniere, fra cui una delle più usate è la cosiddetta *codifica Manchester*.

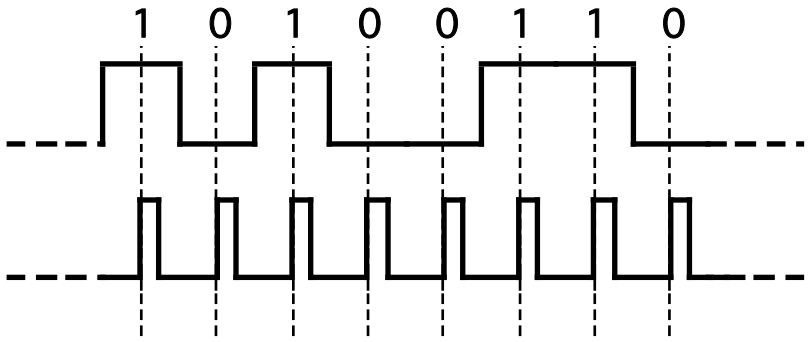


Figura 4.1. - Trasmissione seriale sincrona con segnale di sincronismo separato.

Il metodo è semplice (Fig. 4.2): stabiliamo che, per ogni 1 trasmesso, la linea debba passare dallo stato 0 allo stato 1, e che, per ogni 0, la linea debba passare dallo stato 1 allo stato 0. Per ogni bit quindi deve obbligatoriamente esserci una transizione di stato: se si devono trasmettere due o più bit uguali, occorrerà aggiungere una seconda transizione per riportare il segnale nello stato corretto.

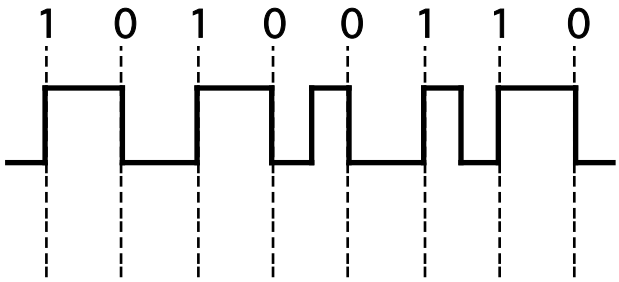


Figura 4.2. - Trasmissione seriale sincrona con codifica Manchester.

Anche la ricezione è semplice: una volta individuato il primo cambiamento di stato, è sufficiente attendere un tempo pari a circa tre quarti della durata di un singolo bit prima di ricominciare ad osservare lo stato della linea. Ogni transizione rilevata viene trasformata nel simbolo corrispondente (1 o 0 a seconda dei casi).
La comunicazione sincrona funziona bene solo quando il flusso dei dati è continuo. Infatti, qualunque sia il metodo utilizzato, occorre che prima di iniziare la trasmissione vera e propria vengano inviati alcuni caratteri di controllo che permettono la sincronizzazione iniziale, che però viene persa ogni volta che il flusso di dati si interrompe. Per questo, si inseriscono caratteri di riempimento (*idle*) quando non ci sono dati da trasmettere.
Se per una qualunque ragione il sincronismo viene perso durante la trasmissione di un blocco di dati, esso può essere ripristinato solo all'inizio di un nuovo blocco.

4.3.2.2. La trasmissione seriale asincrona

Le ragioni appena menzionate fanno sì che, quando il flusso dei dati è discontinuo, come accade nella grande maggioranza dei casi, sia preferibile la modalità di trasmissione *asincrona*. In quest'ultima, il segnale di sincronismo non viene trasmesso affatto, e il ricevitore si sincronizza con il trasmettitore sfruttando il fatto che entrambi sono provvisti di un orologio, e quindi il ricevitore può conoscere con sufficiente precisione gli istanti in cui i dati trasmessi sono validi.
La trasmissione (Fig. 4.3) è basata sul seguente meccanismo: ogni bit viene mantenuto sulla linea per un certo tempo, fisso e noto al ricevitore. Prima di iniziare la trasmissione viene inviato un bit che è sempre 1 (*bit di start*), poi vengono trasmessi in sequenza i bit che compongono il carattere (normalmente 8, 9 se è presente il bit di parità di cui si parlerà più avanti), ed infine viene inviato un bit (*bit di stop*) che conclude la trasmissione del carattere. Il processo si può ripetere indefinitamente; fra un carattere e l'altro può intercorrere un tempo di qualunque lunghezza, al limite nullo.

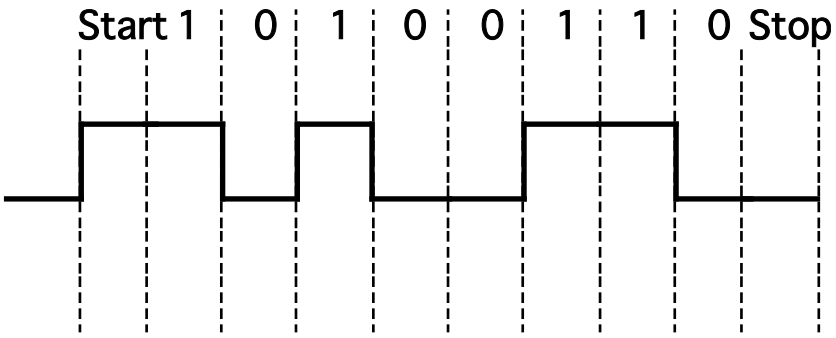


Figura 4.3. - Trasmissione seriale asincrona.

La ricezione del carattere avviene in questo modo: quando il ricevitore avverte il primo fronte di salita del segnale (bit di start), attende un tempo pari ad una volta e mezza la durata di un bit, poi campiona il segnale 9 o 10 volte ad intervalli pari alla durata di un bit. Il valore della linea ad ogni campione viene assunto come valore del bit corrispondente. L'ultimo campione, che viene preso in corrispondenza del bit di stop, non apporta informazione, ma serve solo a verificare che lo stato della linea in quel momento sia 0. Se ciò non si verifica, c'è stato evidentemente un errore (*framing error*).
Dal momento che non esiste più un sincronismo diretto fra il ricevitore ed il trasmettitore, e che, come abbiamo detto, il ricevitore deve conoscere la velocità del trasmettitore, si è resa necessaria una standardizzazione delle velocità per consentire il collegamento fra apparecchiature di marche diverse. Le velocità più comunemente usate oggi per il collegamento seriale asincrono sono 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 38400 baud.

4.3.3. LA TRASMISSIONE SU LINEE TELEFONICHE

Quando la distanza fra il ricevitore ed il trasmettitore aumenta, la trasmissione su normali fili diventa problematica perché le loro caratteristiche elettriche fanno sì che i segnali siano attenuati e deformati. Inoltre molto spesso non è addirittura possibile avere a disposizione linee dedicate, ed occorre rivolgersi ad altri tipi di mezzi trasmissivi.
Un sistema di comunicazione che si trova con grande facilità in quasi ogni luogo è il telefono: purtroppo però esso è stato progettato per trasmettere la voce, e non i segnali di cui ci stiamo occupando. Se infatti provassimo a trasmettere direttamente un segnale come quello di figura 4.3, all'altro capo della linea telefonica vedremmo un segnale simile a quello di figura 4.4, che ha ben poco a che fare con l'originale. Possiamo però trasformare il segnale in partenza, prima di immetterlo sulla linea, in un segnale più adatto a transitare sulla linea stessa. Questo procedimento che, con le variazioni del caso, è adottato per tutti i sistemi di comunicazione non diretti (telefono, radio, fibre ottiche, ecc.) è chiamato *modulazione*. Di esso esistono infinite varianti: una molto usata nel caso delle linee telefoniche è la modulazione FSK (*frequency shift keying*), che consiste nel trasmettere un'onda sinusoidale di frequenza

appropriata al sistema di trasmissione usato. Questa frequenza può assumere due valori diversi, a seconda che essa debba rappresentare uno 0 o un 1 (figura 4.5).

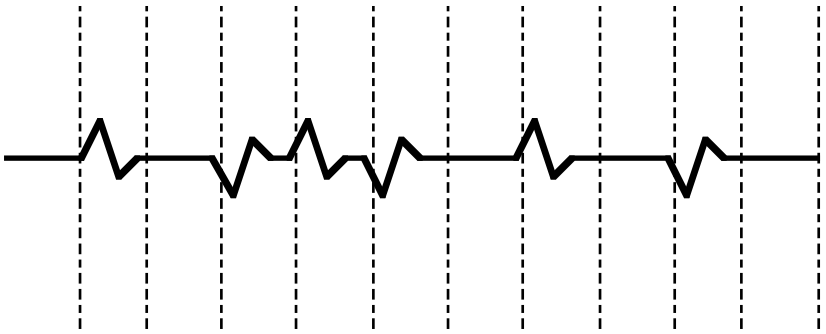


Figura 4.4. - Il segnale di figura 4.3 trasmesso attraverso una linea telefonica.

Ovviamente, per poter utilizzare il segnale, all'altro capo della linea dovremo avere un dispositivo (*demodulatore*) che ritrasformi il segnale modulato in quello originale. Siccome in genere la comunicazione su una linea è bidirezionale (ci sono cioè dati che viaggiano in entrambi i sensi), ad ogni capo sono necessari sia un modulatore per i dati in partenza che un demodulatore per quelli in arrivo. È allora conveniente conglobare i due apparecchi in uno solo, che prende il nome di *modem* (*MODulator - DEModulator*).

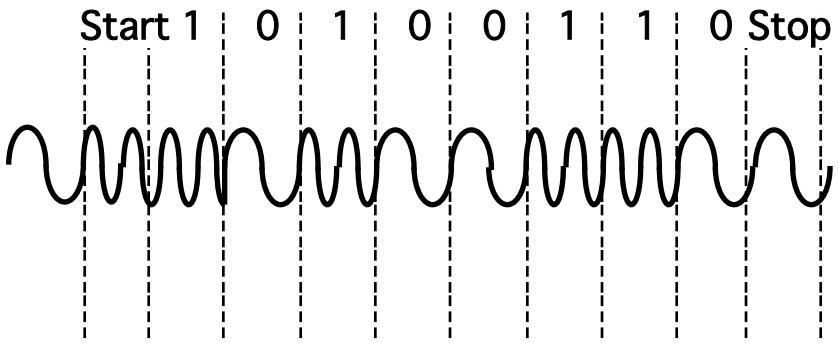


Figura 4.5. - Il segnale di figura 4.3 modulato FSK.

Osserviamo che utilizzando i modem è possibile trasmettere dati seriali sia in modo sincrono che asincrono. Osserviamo anche che, con l'introduzione dei microprocessori, i modem sono diventati strumenti estremamente efficaci, in grado per esempio di stabilire automaticamente, in base alle caratteristiche della linea e dell'altro modem a cui sono collegati, la massima velocità di trasmissione e di modificarla se le caratteristiche della linea cambiano, di implementare

autonomamente sistemi di compressione dei dati e di correzione automatica degli errori, ecc.

4.3.4. LA CORREZIONE DEGLI ERRORI

Le linee lungo le quali vengono trasmessi i dati sono affette da rumore, il quale può, in misura maggiore o minore a seconda dei casi, causare alterazioni dei dati in transito. Questo fa sì che, in generale, non si abbia la certezza che i dati ricevuti siano gli stessi che erano stati originariamente trasmessi. Dal momento che la correttezza della trasmissione è un fattore molto importante, e in alcuni casi addirittura vitale, occorre studiare meccanismi che permettano di rilevare la presenza di eventuali errori intervenuti durante la trasmissione dei dati.
Dobbiamo fare una netta distinzione fra due diversi sistemi: quelli che permettono la *rilevazione* degli errori, e quelli che permettono anche la loro *correzione*. Rilevare un errore significa semplicemente disporre di un sistema in grado di stabilire che, durante la trasmissione di un certo insieme di dati, è intervenuto un disturbo che li ha alterati, senza fornire alcuna informazione su quali siano i dati alterati; correggere un errore significa, al contrario, disporre di un meccanismo che sia in grado non solo di rilevare l'errore ma anche, con opportuni sistemi, di determinare dove esso si è verificato, e quindi di modificarlo in modo da riportarlo alla sua forma originale.
Nei sistemi di trasmissione di dati la rilevazione assume in genere importanza maggiore della correzione:[20] infatti, è in genere possibile, una volta rilevato un errore, fare in modo che l'intero blocco di dati alterato sia ritrasmesso.
Inoltre, con l'eccezione del metodo del CRC che sarà descritto in seguito, i codici correttori ed autocorrettori (ad esempio i codici di Hamming) richiedono che venga trasmessa una quantità aggiuntiva di informazione che renderebbe in genere inefficiente il sistema di trasmissione.

4.3.4.1. Il modello della linea di trasmissione

Per quanto concerne il problema degli errori, la linea di trasmissione può essere semplicemente modellizzata come mostrato in figura 4.6.

[20] Al contrario di quanto avviene ad esempio per le memorie.

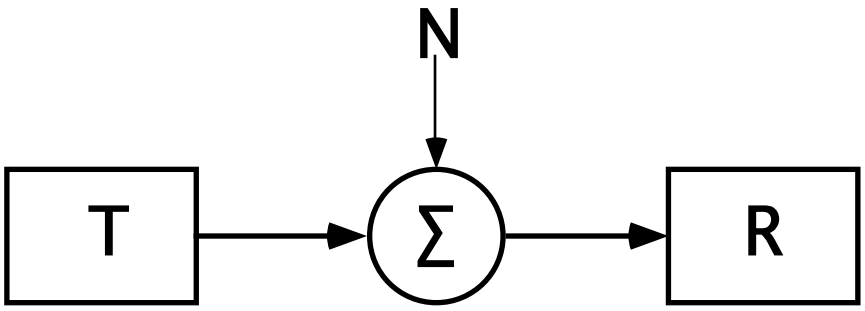


Figura 4.6. - Modello del canale di trasmissione.

In essa, il trasmettitore T emette un flusso di dati che, essendo binari, è costituito da una serie di zeri e di uni. La linea è rappresentata da un sommatore che aggiunge ai dati trasmessi il rumore N, anch'esso rappresentato da una serie di bit. Il ricevitore R ha quindi in ingresso un segnale che è dato dalla somma del segnale originariamente trasmesso e del rumore. È importante osservare che il sommatore opera la somma modulo 2, cioè secondo la tabella dell'operatore XOR, che è riportata in figura 4.7.

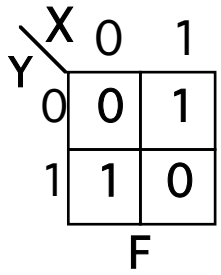


Figura 4.7. - Tabella della somma modulo 2.

Ciò vuol dire che, se il rumore è 0, il segnale passa inalterato, mentre, se il rumore è 1, il suo effetto non si propaga nel tempo: ogni disturbo modifica solo ed esclusivamente il bit che viene trasmesso in quel momento, complementandone il valore, mentre i bit precedenti e seguenti mantengono il loro valore originario. Questo modello corrisponde bene alle caratteristiche di quasi tutti i canali di trasmissione che sono affetti da disturbi di tipo impulsivo i quali possono momentaneamente modificare i segnali trasmessi. Occorre anche osservare che la frequenza di uni nel rumore deve essere ragionevolmente bassa perché il canale possa essere utilizzato: è chiaro che, quanto più aumenta il rumore, tanto più problematica diventa la trasmissione dei dati, fino a diventare del tutto impossibile.

4.3.4.2. Il bit di parità (*parity bit*)

Il primo e più semplice meccanismo per la rilevazione degli errori è il cosiddetto *bit di parità*. Esso consiste nell'aggiungere ad ogni parola trasmessa un bit, il cui valore è

0 o 1 a seconda che il numero di uni nella parola trasmessa sia pari o dispari. In altre parole, il bit di parità serve a fare in modo che il numero complessivo di uni contenuti nella parola sia sempre pari.
Ad esempio, per la parola

0 0 1 1 1 0 1 0

il bit di parità è 0, mentre per la parola

0 1 1 0 1 0 0 0

il bit di parità è 1.
Il funzionamento del sistema a questo punto è evidente: il trasmettitore aggiunge ad ogni parola trasmessa il bit di parità, e il ricevitore controlla, per ogni parola ricevuta, che il numero totale di uni in essa contenuti sia pari. Se esso è dispari, è certamente intervenuto un errore: non siamo in grado però, dal momento che non si tratta di un sistema di correzione, di capire qual è il bit che è stato alterato.
Osserviamo anche che se, durante la trasmissione di una parola, intervengono due errori (o comunque un numero pari di errori), essi non saranno rilevati, perché i loro effetti si annullano. Dal momento però che in molti canali di trasmissione gli errori sono sporadici, la probabilità che due bit della stessa parola siano errati è molto bassa, e l'applicazione del metodo è quindi giustificata.

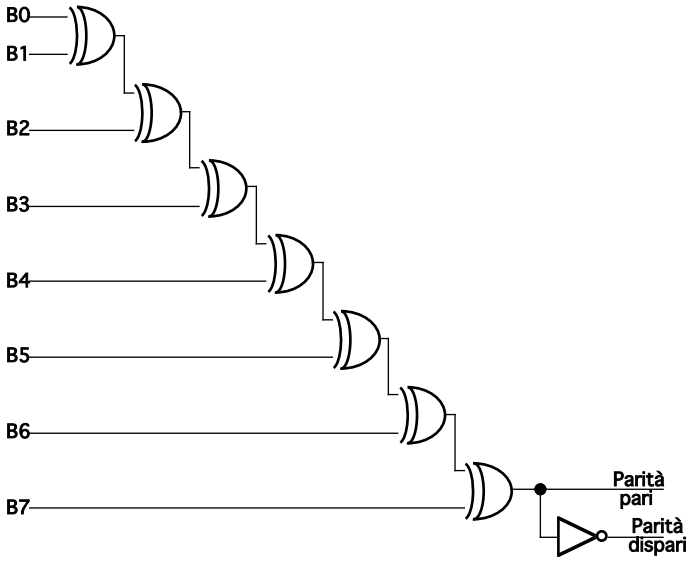


Figura 4.8. - Circuito per il calcolo del bit di parità.

Il bit di parità di cui ci siamo occupati finora, che rende pari il numero di uni contenuti nella parola, è detto *bit di parità pari*. In modo del tutto analogo può essere definito il *bit di parità dispari*, che è semplicemente il complemento del precedente. Il calcolo del bit di parità è molto semplice, e può essere effettuato ad esempio con il circuito mostrato in figura 4.8.

La figura 4.9 mostra invece un circuito per la verifica della parità pari.

4.3.4.3. Controllo di parità longitudinale e trasversale

Le limitazioni appena viste del sistema basato sul controllo di parità possono essere ridotte, quando i dati da trasmettere sono più di uno, estendendo il metodo in modo da comprendere nel calcolo più di una parola.
A titolo di esempio, consideriamo il blocco di dati mostrato in figura 4.10.

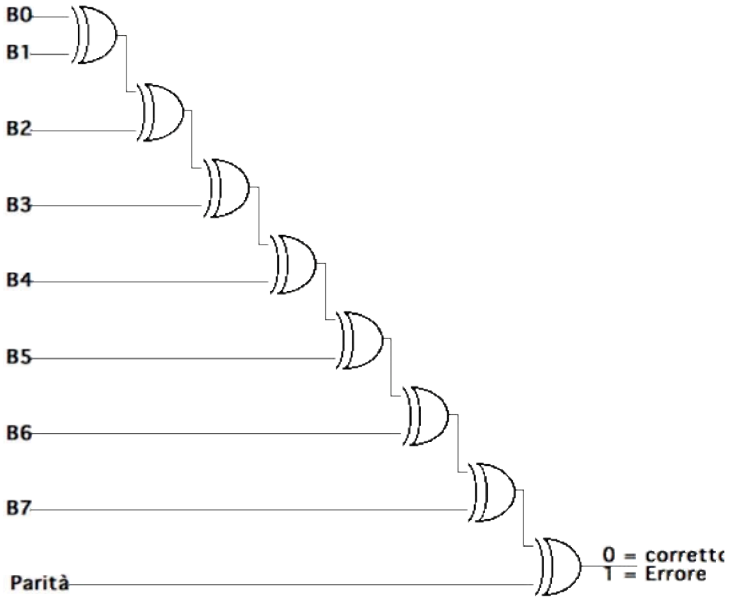


Figura 4.9. - Circuito per la verifica della parità pari.

B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
0	0	1	1	0	1	1	1
1	1	1	0	0	0	1	1
0	0	1	0	1	0	0	0
0	0	0	0	0	0	0	0
1	1	0	0	1	1	0	0
0	0	1	1	0	0	1	1
1	0	1	0	1	0	1	1
1	1	1	1	0	0	0	0
1	0	0	1	1	0	1	0
1	1	1	0	0	0	1	0

Figura 4.10. - Un blocco di 10 byte.

Secondo quanto abbiamo appena detto, possiamo aggiungere ad ogni byte il bit di parità, come si può vedere in figura 4.11. Questo bit prende anche il nome di *bit di parità trasversale*.

P	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
1	0	0	1	1	0	1	1	1
1	1	1	1	0	0	0	1	1
0	0	0	1	0	1	0	0	0
0	0	0	0	0	0	0	0	0
0	1	1	0	0	1	1	0	0
0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	1
0	1	1	1	1	0	0	0	0
0	1	0	0	1	1	0	1	0
0	1	1	1	0	0	0	1	0

Figura 4.11. - Gli stessi dati con il bit di parità trasversale.

A questo punto è possibile, utilizzando lo stesso algoritmo, calcolare il bit di parità per ogni colonna, come mostrato in figura 4.12. L'insieme di questi nuovi bit di parità costituisce un'ulteriore parola, detta *parità longitudinale* o *checksum*, e permette di rilevare anche un numero pari di errori sulla stessa parola.

P	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
1	0	0	1	1	0	1	1	1
1	1	1	1	0	0	0	1	1
0	0	0	1	0	1	0	0	0
0	0	0	0	0	0	0	0	0
0	1	1	0	0	1	1	0	0
0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	1
0	1	1	1	1	0	0	0	0
0	1	0	0	1	1	0	1	0
0	1	1	1	0	0	0	1	0
1	0	0	1	0	0	0	0	0

Figura 4.12. - Aggiunta della parità longitudinale.

P	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
1	0	0	1	1	0	1	1	1
1	1	1	1	0	0	0	1	1
0	0	0	**0**	0	1	**1**	0	0
0	0	0	0	0	0	0	0	0
0	1	1	**1**	0	1	**0**	0	0
0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	1
0	1	1	1	1	0	0	0	0
0	1	0	0	1	1	0	1	0
0	1	1	1	0	0	0	1	0
1	0	0	1	0	0	0	0	0

Figura 4.13. - Quattro errori che si compensano a vicenda.

È evidente che questo metodo, pur essendo più sofisticato del semplice bit di parità, non è immune da errori, nel senso che possono comunque verificarsi insiemi di errori di trasmissione che non vengono rilevati. Ad esempio, se il blocco di dati viene alterato come in figura 4.13, in cui gli errori sono segnati in grassetto, il controllo di parità trasversale e longitudinale non rivela nulla di anormale, perché tutti gli errori si compensano a vicenda.

Se la probabilità che, in un determinato canale di trasmissione, si verifichino errori non rilevabili come quelli appena mostrati non è trascurabile, occorre fare ricorso a metodi più sofisticati, come quello del CRC, di cui si parlerà più avanti.

4.3.4.4. Un altro metodo per il calcolo del checksum

Un secondo metodo per il calcolo del checksum, che presenta qualche vantaggio rispetto al precedente, è basato sull'algoritmo che segue.

Durante la trasmissione dei dati, si esegue la somma di tutte le parole trasmesse. Al termine di ogni blocco, si fa il complemento a 2 della somma calcolata, e lo si trasmette. Il ricevitore, dal canto suo, somma anch'esso tutti i dati ricevuti, compreso il checksum. Al termine, se non si sono verificati errori, il totale deve ovviamente essere zero: in caso contrario, si ha la certezza che si è verificato qualche errore.

In pratica, dal momento che i dati trasmessi hanno una lunghezza di parola di 8 bit, si può operare in due modi diversi:

a. la somma viene mantenuta in una parola di 8 bit, e i bit eccedenti vengono semplicemente scartati;
b. la somma viene mantenuta in due parole di 8 bit, per un totale di 16 bit. Anche in questo caso gli eventuali bit eccedenti vengono scartati, ma occorre osservare che perché ciò accada occorre che il blocco sia composto da almeno 257 parole.

Un esempio del primo metodo è mostrato in figura 4.14.

	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
Parola 1	1	0	0	1	0	1	1	0
Parola 2	1	1	0	0	0	1	0	0
Parola 3	0	1	0	1	0	1	1	1
Parola 4	0	0	0	1	0	1	1	1
Parola 5	0	0	1	0	1	0	1	1
Somma	1	1	1	1	0	0	1	1
Checksum	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**1**

Figura 4.14. - Il checksum (a 8 bit) di un blocco di cinque parole.

Il lettore può verificare che, trascurando i riporti, la somma delle cinque parole e del checksum è uguale a zero.

Nel caso del checksum a 16 bit il procedimento è identico. Ovviamente, dal momento che la lunghezza delle parole trasmesse è sempre di 8 bit, occorrono due parole per trasmettere il checksum.

4.3.4.5. Il CRC (*Cyclic Redundancy Check*)

Nel caso in cui, per le cattive caratteristiche della linea di trasmissione da utilizzare, o per l'importanza delle informazioni da trasmettere, i metodi finora visti non diano sufficienti garanzie di affidabilità, si può ricorrere a un metodo che, seppure più complicato, offre garanzie di sicurezza molto maggiori. Esso si basa sul concetto di macchina lineare, cioè di una macchina sequenziale realizzata utilizzando solo tre tipi di elementi:

- il ritardo elementare *R*: si tratta di un elemento che ritarda di una unità di tempo la variabile presentata al suo ingresso;
- il sommatore in modulo *p*, che esegue la somma (appunto in modulo *p*) dei suoi ingressi;
- il moltiplicatore per una costante, la cui uscita è costituita dal prodotto (effettuato in modulo *p*) dell'ingresso per una costante.

Nel nostro caso, della intera classe delle macchine lineari hanno interesse quelle, dette *registri a scorrimento generalizzati*, che sono costituite dalla generica struttura riportata in figura 4.15.

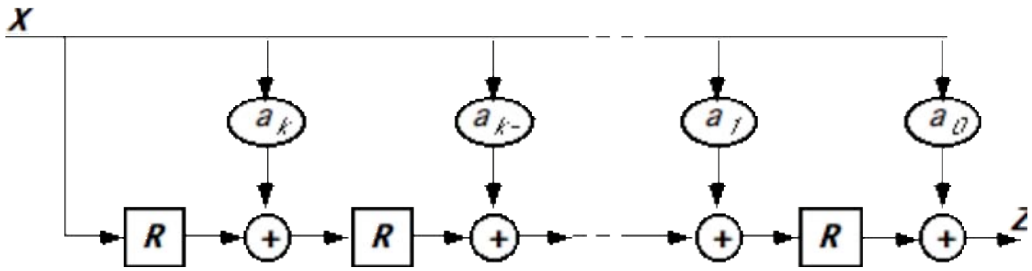


Figura 4.15. - Un registro a scorrimento generalizzato.

Osserviamo inoltre che, operando con numeri binari, le costanti $a_0, a_1, \ldots, a_k$ possono assumere solo i valori 0 o 1: il significato dei moltiplicatori è quindi quello di "circuito aperto" (se la costante vale 0) o di "collegamento effettuato" (se la costante vale 1).
Se indichiamo convenzionalmente l'operazione di ritardo come il "prodotto dell'ingresso per il ritardo elementare", possiamo scrivere, per la generica rete di figura 4.15, la relazione fra ingresso e uscita:

$$Z = a_0 X + a_1 RX + a_2 R^2 X + \ldots + A_{k-1} R^{k-1} X + a_k R^k X$$

dove con R^i abbiamo indicato l'azione di ritardo per *i* unità di tempo.
Meglio ancora, possiamo definire la *funzione di trasferimento* della macchina come

$$T = \frac{Z}{X} = a_0 + a_1 R + a_2 R^2 + \ldots + A_{k-1} R^{k-1} + a_k R^k$$

Come esempio, consideriamo la macchina rappresentata in figura 4.16.

La sua funzione di trasferimento è

$$T = \frac{Z}{X} = 1 + R + R^4$$

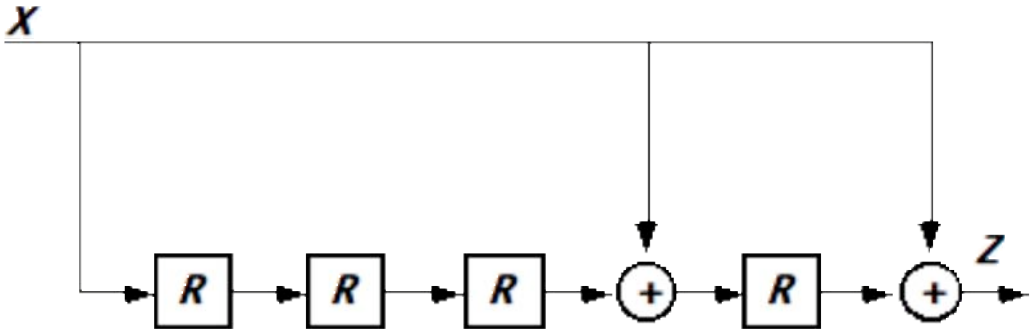


Figura 4.16. - Un esempio pratico di registro a scorrimento generalizzato.

Supponiamo ora che, inizialmente, la macchina sia *inerte*, cioè che tutte le uscite degli elementi di ritardo siano al valore 0. Ci interessa considerare la cosiddetta *risposta all'impulso*, cioè il comportamento dell'uscita quando all'ingresso viene applicato un 1, seguito da una sequenza di zeri. È facile verificare che, per la macchina di figura 4.16, tale risposta sarà 11001. Infatti, osservando l'espressione generica della funzione di trasferimento, è facile identificare la risposta all'impulso con la sequenza

$$a_0 \bullet 1, a_1 \bullet 1, \square , a_{k\square 1} \bullet 1, a_k \bullet 1$$

Nel caso in esame, dati i valori dei coefficienti, abbiamo quindi la sequenza

$$1 \bullet 1, 1 \bullet 1, 0 \bullet 1, 0 \bullet 1, 1 \bullet 1$$

cioè appunto 11001. Nota la risposta all'impulso, è facilissimo calcolare la risposta della macchina a qualunque sequenza di ingresso. Osserviamo infatti che, essendo la macchina lineare ed inizialmente inerte, vale per essa il ben noto principio di sovrapposizione degli effetti: se consideriamo quindi l'ingresso (che sarà costituito da una sequenza di zeri e di uni) come una serie di impulsi applicati in istanti opportuni, l'uscita della macchina sarà calcolabile come la somma delle uscite dovute ai singoli impulsi. Supponiamo per esempio di applicare all'ingresso della macchina la sequenza $X = 1101001$: avremo, ricordando che le somme devono essere effettuate in modulo 2,

$t=0$	$X=1$	$Z=$	1	1	0	0	1	0	0	0	0	0	0	0
$t=1$	$X=1$	$Z=$		1	1	0	0	1	0	0	0	0	0	0
$t=2$	$X=0$	$Z=$			0	0	0	0	0	0	0	0	0	0
$t=3$	$X=1$	$Z=$				1	1	0	0	1	0	0	0	0
$t=4$	$X=0$	$Z=$					0	0	0	0	0	0	0	0
$t=5$	$X=0$	$Z=$						0	0	0	0	0	0	0
$t=6$	$X=1$	$Z=$							1	1	0	0	1	0
		$Z=$	1	0	1	1	0	1	1	0	0	0	1	0

Supponiamo ora di avere a disposizione una macchina, sempre lineare, la cui funzione di trasferimento sia l'inversa di quella appena considerata. La relazione fra ingresso e uscita della macchina diretta può essere riscritta nella forma

$$X = \frac{1}{a_0}\left(Z \oplus a_1 RX \oplus a_2 R^2 X \oplus \cdots \oplus A_{k-1} R^{k-1} X \oplus a_k R^k X\right)$$

che, ricordando che in modulo 2 è $-a_i = a_i$, dà luogo ad una macchina la cui struttura generica è mostrata in figura 4.17. Osserviamo che la costruzione di una macchina inversa è sempre possibile, a condizione che nella macchina diretta sia $a_0 \neq 0$.
Per la macchina del nostro esempio, la struttura sarà quella di figura 4.18.
La relazione ingresso-uscita di questa macchina è

$$X = Z + RX + R^4 X$$

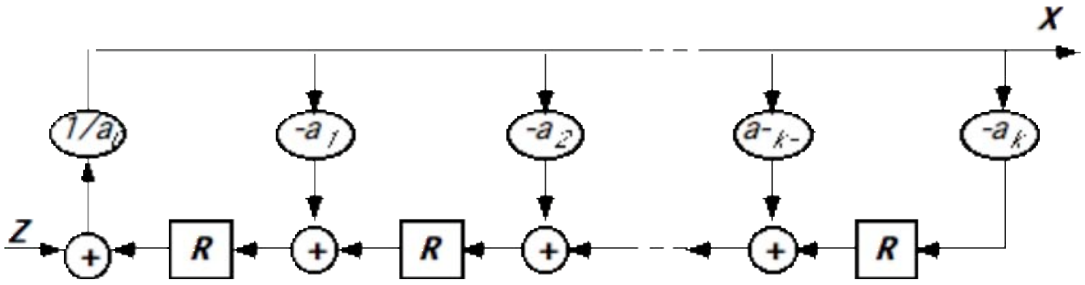


Figura 4.17. - La generica macchina inversa.

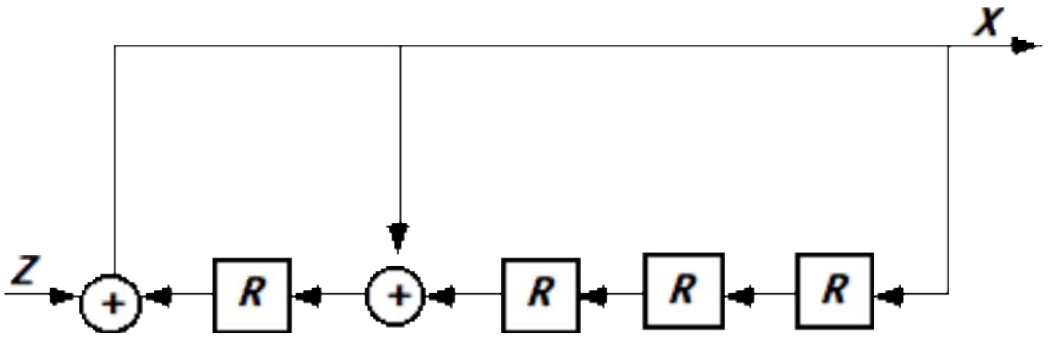


Figura 4.18. - La macchina inversa di quella di figura 4.16.

Dal momento che anche questa macchina è lineare, possiamo identificarne la risposta all'impulso, che è 111101011001000. Osserviamo che, al contrario della macchina diretta, in quella inversa la risposta è periodica, data la presenza di anelli di reazione che "riportano" il segnale di uscita all'ingresso.
Se ora proviamo ad applicare alla macchina inversa l'uscita di quella diretta quando quest'ultima viene stimolata da un impulso, troveremo all'uscita della macchina inversa ancora un impulso. La verifica di quanto appena affermato è lasciata al lettore.
Sulla disponibilità di una macchina diretta e di una macchina inversa è basato il metodo del controllo ciclico di ridondanza, che forma oggetto del presente paragrafo. Infatti, possiamo immaginare di costruire un sistema in cui i segnali, prima di essere trasmessi, vengano fatti passare attraverso una macchina diretta (che in questo caso viene chiamata *codificatore*) e, prima di essere acquisiti al termine della linea di trasmissione, vengano fatti passare attraverso un *decodificatore*, avente una funzione di trasferimento inversa, secondo lo schema di figura 4.19.

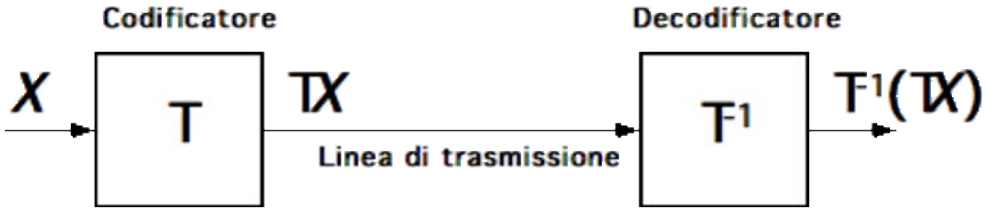


Figura 4.19. - Impiego di un codificatore e di un decodificatore.

Ricordando che le macchine sono lineari, è evidentemente $T \bullet T^{-1} = 1$, e quindi il segnale in uscita dal decodificatore è uguale a quello presente all'ingresso del codificatore.

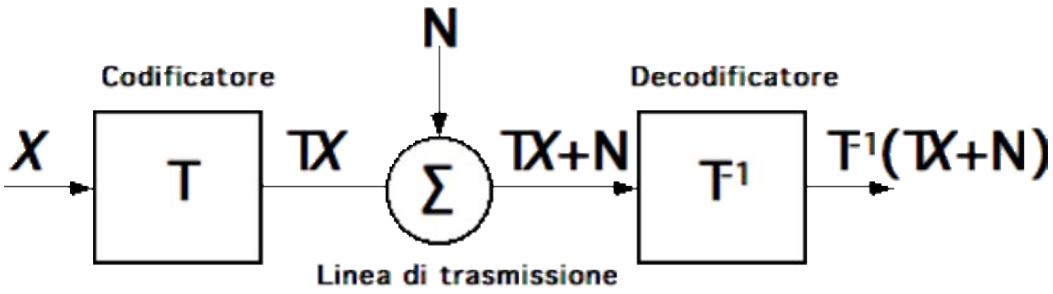


Figura 4.20. - Trasmissione su una linea affetta da rumore.

Se ora supponiamo che la linea sia affetta da rumore, ed applichiamo il modello presentato a suo tempo, avremo lo schema di figura 4.20.
Dal momento che vale il principio di sovrapposizione degli effetti, possiamo però scrivere

$$T^{-1}(TX + N) = T^{-1}TX + T^{-1}N = X + T^{-1}N$$

Ora, se il ricevitore conoscesse il messaggio trasmesso, sarebbe immediato identificare il rumore: dal momento però che ci troviamo in una situazione ben diversa (il ricevitore non conosce, ovviamente, il messaggio trasmesso) possiamo operare in questo modo:

- si aggiunge in coda ad ogni messaggio trasmesso un certo numero di bit, la cui configurazione sia nota (questo è, propriamente parlando, il CRC);
- dopo la ricezione, si controlla che la configurazione degli ultimi bit sia quella della coda del messaggio originario;
- se la configurazione non è quella corretta, ciò può essere dovuto solo al fatto che al messaggio originario si è aggiunto del rumore, i cui effetti, visto che la risposta all'impulso della macchina inversa è periodica, si propagano anche alla coda del messaggio ricostruito.

Sotto determinate ipotesi, è addirittura possibile comprendere qual è il punto del messaggio in cui si è verificato l'errore e, sottraendo la risposta all'impulso della macchina inversa al messaggio decodificato, ricostruire quello originale. In questo caso, abbiamo implementato un sistema in grado non solo di rilevare errori di trasmissione, ma anche di correggerli.
Con qualche variazione, il metodo appena descritto è oggi largamente usato in moltissimi sistemi per la trasmissione dei dati. Ovviamente, per permettere la rilevazione di errori complessi, la macchina lineare utilizzata deve avere un numero piuttosto elevato di elementi di ritardo: una macchina utilizzata internazionalmente in diversi standard ha la relazione ingresso-uscita

$$Z = X + R^4X + R^{11}X + R^{15}X$$

4.3.5. I PROTOCOLLI DI LINEA

Una volta appreso come rilevare e/o correggere gli errori, dobbiamo occuparci del problema della trasmissione di informazioni su mezzi trasmissivi affetti da rumore. Come abbiamo già detto, nella trasmissione di dati a distanza non ha grande importanza che il sistema sia in grado di correggere gli errori perché, qualora se ne rilevi uno, il ricevitore può chiedere al trasmettitore di rispedire l'intero blocco che è stato ricevuto in maniera errata: ciò comporta una perdita di tempo, che è però giustificata dalla maggiore semplicità ed efficienza dei sistemi che si limitano a rilevare gli errori rispetto a quelli che possono anche correggerli.
Vediamo allora i principi base che permettono di assicurare una trasmissione corretta anche in presenza di rumore. Essi sono:

- i dati da trasmettere devono essere raggruppati in sequenze di una certa lunghezza. A seconda dei casi questa lunghezza può essere fissa o variabile: l'importante è che ogni sequenza abbia un inizio e una fine. Chiamiamo queste sequenze *messaggi* o *pacchetti*;
- ogni pacchetto deve portare, oltre ai dati, un numero che permetta di distinguerlo da tutti gli altri pacchetti trasmessi;
- ogni pacchetto deve essere trasmesso con un sistema che permetta la rilevazione efficace degli errori. Il sistema che riceve il pacchetto deve essere in grado cioè di capire con sicurezza se ciò che gli è arrivato è corretto oppure no.

Dal momento che, come si vedrà, la trasmissione di un pacchetto richiede scambio di informazioni in entrambi i sensi, non è più opportuno parlare di trasmettitore e ricevitore, perché entrambe le unità assumono alternativamente i due ruoli. Parleremo invece di *primario* riferendoci all'unità che emette il pacchetto, e di *secondario* riferendoci a quella che lo riceve.
Il più semplice protocollo implementabile con le condizioni appena menzionate si basa sul seguente principio:

a. il primario trasmette il pacchetto;
b. il secondario, una volta ricevuto il pacchetto e verificato che è corretto, spedisce un pacchetto, detto *acknowledge* (riconoscimento) o più brevemente *ACK*, che indica convenzionalmente l'avvenuta ricezione del pacchetto spedito. Si tratta dell'esatto equivalente della ricevuta di ritorno delle lettere raccomandate;
c. il primario, con la ricezione dell'ACK, sa che il pacchetto è arrivato a destinazione, ed invia il successivo.

Questo è ciò che accade se non si verificano errori. Dal momento però che supponiamo la linea soggetta a rumore, possono anche darsi i due casi seguenti:

a. il pacchetto originale arriva alterato;
b. l'ACK arriva alterato.

In entrambi i casi, dal momento che la ricezione di un pacchetto alterato da errori di trasmissione equivale a non aver ricevuto nulla, il primario, non ricevendo l'ACK, trascorso il tempo ragionevolmente occorrente per la trasmissione e l'elaborazione dei

dati (detto *tempo di guardia*), invia nuovamente il pacchetto. Dalla situazione b deriva la necessità di distinguere univocamente i pacchetti numerandoli: se ciò non fosse fatto, il secondario non sarebbe in grado di comprendere che i dati dell'ultimo pacchetto arrivato erano già stati ricevuti, e che il pacchetto stesso è stato rispedito semplicemente perché il corrispondente ACK è andato perso.
Lo schema appena descritto, detto *idle RQ* (richiesta con stati di attesa), è la base di tutta una serie di protocolli di linea, che tendono ad aumentare il flusso di dati riducendo i tempi in cui la linea non trasporta segnali. Un primo, evidente miglioramento delle prestazioni potrebbe essere ottenuto introducendo un altro pacchetto, detto di *non riconoscimento* (NACK), usato per segnalare la ricezione di un pacchetto errato. In questo caso, se il pacchetto originario arriva alterato, il secondario trasmette un pacchetto NACK, e il primario può immediatamente ritrasmettere il pacchetto, senza dover attendere la fine del tempo di guardia.

4.3.5.1. Il controllo di flusso

In alcuni casi, è possibile considerare le linee virtualmente prive di errori. Ciò è giustificato quando i collegamenti sono molto brevi (un calcolatore e un terminale posto nelle immediate vicinanze, ad esempio) o quando un eventuale errore non avrebbe conseguenze catastrofiche (questo è vero per quasi tutti i terminali interattivi non dedicati a funzioni particolarmente critiche), oppure ancora quando, pur essendo la linea soggetta a rumore, alle sue estremità si trovano dispositivi in grado di correggere automaticamente gli errori, come i modem sofisticati di cui abbiamo già parlato. In questo caso, l'unico problema che sorge è quello del cosiddetto *controllo di flusso*, a cui è dedicato questo paragrafo.
Supponiamo di avere un calcolatore collegato con un terminale mediante una linea seriale bidirezionale. Ciò implica o l'esistenza di due linee fisicamente separate, o di una coppia di modem che siano in grado di convogliare entrambe le linee sullo stesso canale di trasmissione. Infatti, un terminale è costituito in realtà da due periferiche distinte: la tastiera, che manda dati verso il calcolatore, e il video, che mostra i dati ricevuti dal calcolatore. Potremo avere quindi uno dei due casi illustrati in figura 4.21, che sono del tutto equivalenti rispetto al problema del controllo di flusso.
In una situazione del genere, possono darsi casi in cui una delle unità non è disponibile ad accettare i dati che le vengono inviati: per esempio, l'utente può voler bloccare il flusso di dati che arrivano al display, per avere il tempo di leggere quelli che sono già mostrati sullo schermo. Oppure, il calcolatore può non essere disponibile ad accettarne, perché impegnato in altre operazioni (per esempio, un lungo calcolo matematico).
Occorre allora un meccanismo che permetta di interrompere e di far riprendere a piacimento il flusso dei dati. Ciò può essere ottenuto mediante due altre linee, come mostrato in figura 4.22. Ognuna di queste linee, quando il suo livello logico è alto, abilita la corrispondente interfaccia a trasmettere dati, mentre, se il suo livello è basso, la blocca.
Nel caso in cui le due interfacce non siano collegate direttamente, è compito dei modem codificare in maniera opportuna lo stato di queste linee.
La presenza delle linee per il controllo del flusso permette di definire un protocollo che viene normalmente chiamato *hardware handshaking*.

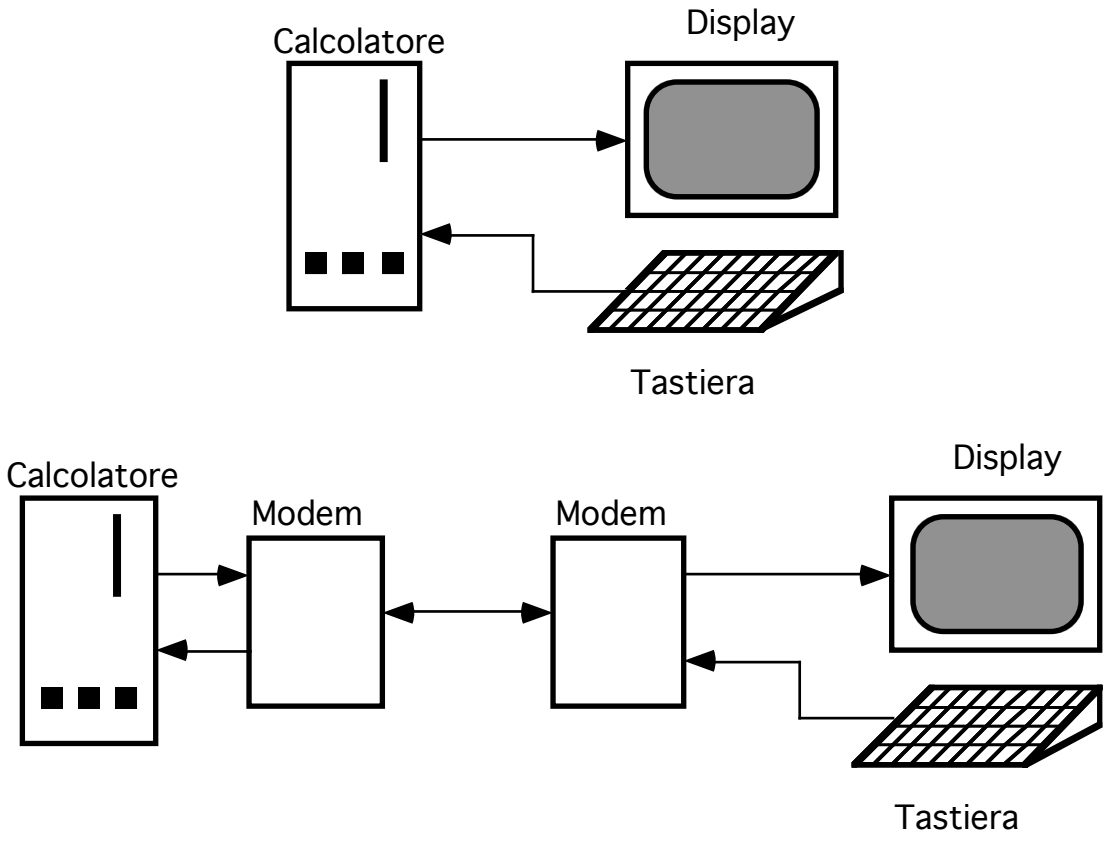


Figura 4.21. - Due possibili collegamenti fra un calcolatore e un terminale.

Quando il collegamento è effettuato direttamente (senza quindi l'impiego di modem) può essere poco conveniente dover utilizzare cinque fili per il collegamento.[21] In questo caso si può ricorrere ad un metodo diverso, detto *protocollo XON-XOFF* o *software handshaking*, che consiste nell'effettuare il collegamento usando solo i tre fili classici, e nell'assegnare un particolare significato a due dei caratteri che possono essere trasmessi. Il meccanismo è il seguente: quando l'unità ricevente non è più in grado di accettare dati, invia al trasmettitore un carattere convenzionale (XOFF). La sua ricezione causa la sospensione della trasmissione. Quando poi il ricevitore sarà nuovamente in grado di accettare dati, trasmetterà un secondo carattere convenzionale (XON), che riabilita la trasmissione dei dati.

[21] Ricordiamo che il quinto filo, non mostrato nelle figure, è quello che serve per stabilire il livello di riferimento delle tensioni, cioè il cosiddetto *filo di massa*.

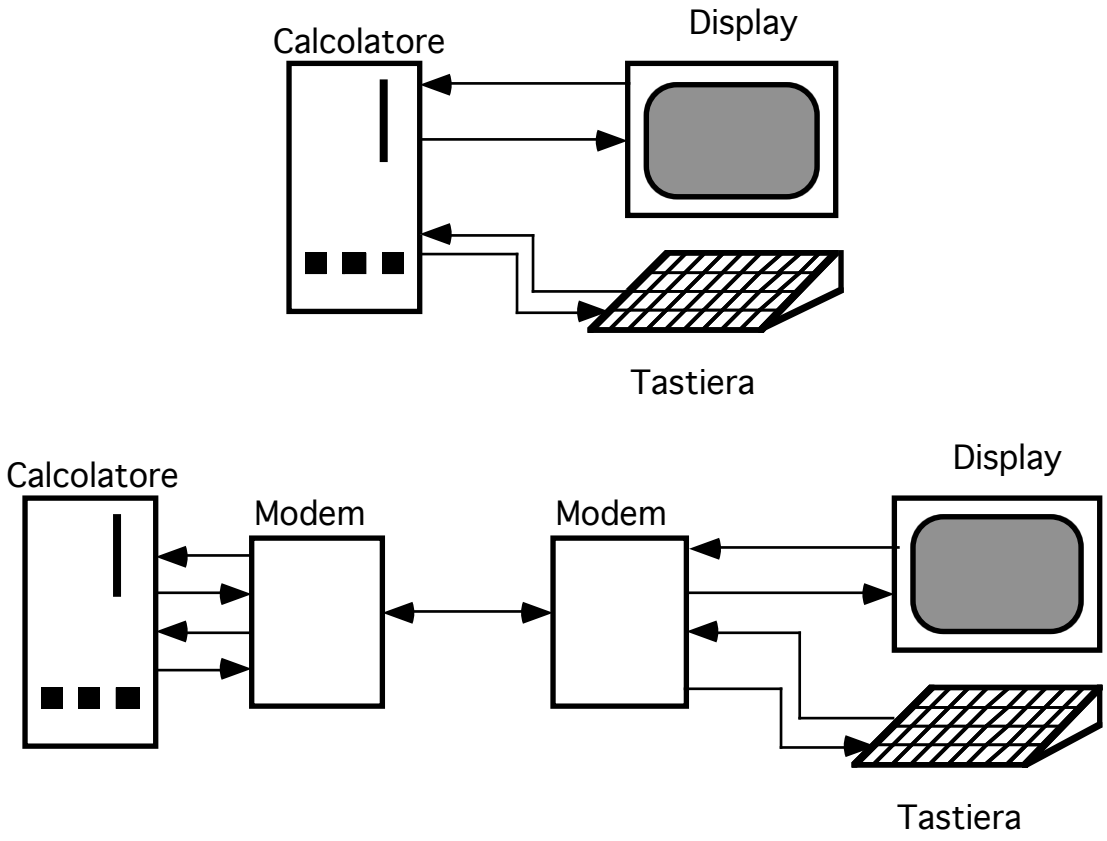


Figura 4.22. - Aggiunta delle linee per il controllo del flusso.

4.4. COLLEGAMENTI MULTIPUNTO

Fino a questo momento ci siamo occupati di sistemi in cui esistevano un unico trasmettitore e un unico ricevitore. Il problema si complica quando si vuole invece utilizzare lo stesso mezzo trasmissivo per collegare fra loro più di due unità, permettendo a ciascuna di esse di trasmettere dati.
La trattazione completa dell'argomento esula del tutto dagli scopi del presente volume: vogliamo dare qui solo i cenni fondamentali di una materia che in questi anni ha raggiunto livelli di sofisticazione (e di complicazione) estremamente elevati. Basti pensare, a questo proposito, che sarebbe ormai virtualmente possibile, utilizzando svariati sistemi di comunicazione, collegare in un'unica rete tutti i calcolatori del mondo. Oltre alle linee telefoniche, di cui abbiamo già parlato, sono infatti disponibili sistemi di comunicazione progettati appositamente per collegare calcolatori (in Italia la rete Itapac), o per fornire un servizio di trasmissione mista suono-immagini-dati, come la rete telefonica digitale ISDN che sta progressivamente soppiantando le reti telefoniche tradizionali. È inoltre possibile collegare anche calcolatori siti in zone disabitate e/o inaccessibili, utilizzando i sistemi di radiotelefonia cellulare e satellitare.

4.4.1. TOPOLOGIA DELLE RETI

Indipendentemente dal fatto che si tratti di reti locali o di reti geografiche,[22] possiamo distinguere due topologie fondamentali di rete: le cosiddette *reti a stella* e quelle *ad anello*.

La differenza (Fig. 4.23) sta nel fatto che nel primo caso tutti i dispositivi presenti sulla rete sono collegati fisicamente insieme, perché la rete è costituita da un cavo che li collega tutti, mentre nel secondo la rete è formata da una serie di collegamenti punto a punto, ognuno dei quali collega singolarmente due dispositivi.

Nel caso delle reti a stella, qualunque dato trasmesso da una unità presente sulla rete viene ricevuto da tutte le altre, mentre nelle reti ad anello i dati emessi da una unità sono ricevuti solo da quella immediatamente seguente, che deve provvedere a ritrasmetterli alla successiva, e così via finché essi non siano arrivati a destinazione.

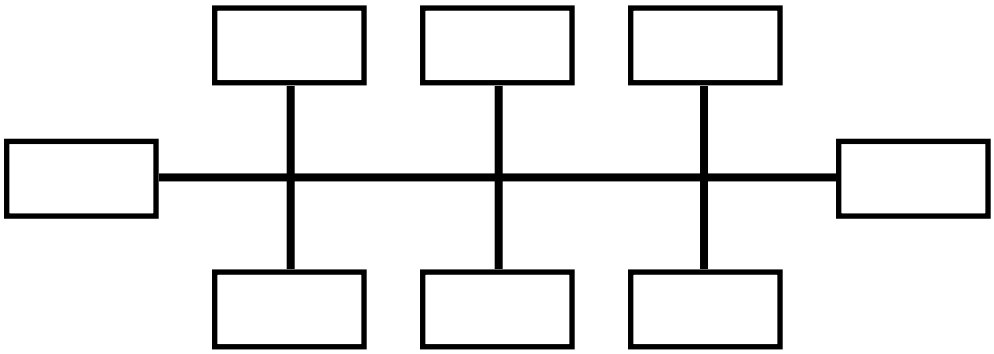
In entrambi i casi possono essere utilizzate tecniche a pacchetto simili a quelle viste prima, ma è necessario che ogni pacchetto porti con sé due informazioni aggiuntive: l'identificazione del destinatario (per sapere a chi è destinato) e quella del mittente (per sapere a chi inviare l'ACK o, più in generale, la risposta).

4.4.2. METODI DI GESTIONE DEI CONFLITTI

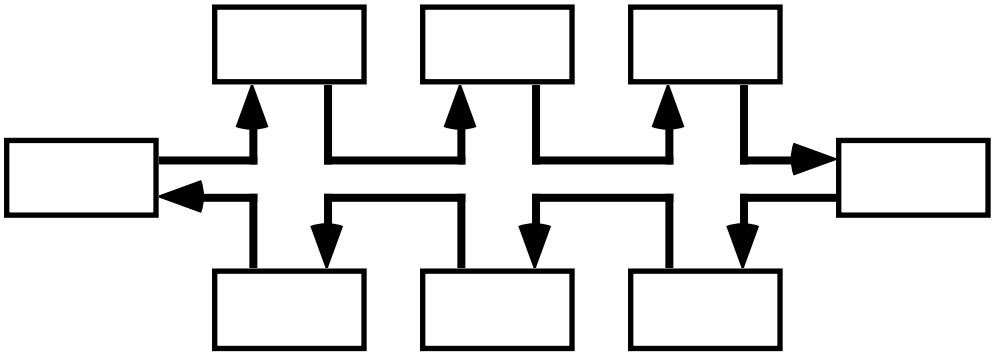
Un secondo problema, tipico delle reti a stella, è quello del conflitto che può verificarsi se due o più unità iniziano a trasmettere nello stesso momento. In questo caso infatti i segnali emessi dalle unità che stanno trasmettendo, dovendo passare sullo stesso mezzo trasmissivo, si sommano, dando luogo ad un segnale globalmente incomprensibile. Fondamentalmente esistono due sistemi, descritti di seguito, per eliminare il problema.

---

[22] Va osservato che le reti geografiche, per la loro complessità, utilizzano in genere strutture che sono combinazioni di quelle qui presentate.



a. - Rete a stella



b. - Rete ad anello

Figura 4.23. - Topologie fondamentali delle reti.

4.4.2.1. Reti a gettone

Viene definito un "gettone" (*token*), il cui possesso garantisce il diritto a trasmettere. Il gettone è una entità logica, ed esiste un procedimento atto a garantire che, quando la rete inizia a funzionare, solo una unità ne sia in possesso. A questo punto, questa unità invia sulla rete i pacchetti che deve trasmettere, certa del fatto che tutte le altre, non avendo il gettone, sono in ascolto. Quando la trasmissione è terminata, essa "passa" il gettone all'unità successiva, che a sua volta trasmette i suoi pacchetti, e poi passa il gettone. Se una unità non ha dati da trasmettere, si limita a passare il gettone alla successiva.

Dal momento che l'ultima macchina della catena ripassa il gettone alla prima, ogni dispositivo viene periodicamente in possesso del gettone, e ha quindi la possibilità di inviare dati.

Il metodo del gettone richiede un hardware piuttosto semplice, a fronte però di una certa complicazione del software, sia per l'inizializzazione della rete, sia per garantire l'esistenza e l'unicità del gettone anche in presenza di errori di trasmissione o di guasti alle unità.

4.4.2.2. Reti a rilevazione delle collisioni

Il secondo metodo, più semplice dal punto di vista software ma che richiede un hardware più sofisticato, è il cosiddetto *CSMA/CD* (*Carrier Sense Multiple Access with Collision Detection*), che si basa sul seguente protocollo:

a. quando una unità deve trasmettere dati, si accerta in primo luogo che la rete sia inattiva, cioè che non ci siano trasmissioni in corso da parte di altre unità (*Carrier Sense*). Ovviamente, ciò non impedisce che altre unità effettuino il medesimo ragionamento nello stesso istante, e che quindi più unità comincino a trasmettere contemporaneamente;

b. durante la trasmissione, le unità che trasmettono restano anche in ascolto sulla rete, verificando che ciò che ricevono sia uguale a ciò che stanno trasmettendo. Se più unità hanno iniziato la trasmissione, ognuna di esse si accorgerà del fatto, perché riceverà un segnale "ingarbugliato", costituito dalla somma di tutti quelli che vengono trasmessi contemporaneamente (*Collision Detection*). Se ciò si verifica, ogni unità sospende immediatamente ciò che sta facendo, dopo aver trasmesso una sequenza di dati atta a rendere certamente irriconoscibili i dati appena trasmessi, in modo che tutte le unità all'ascolto siano costrette a scartarli;

c. passato un certo tempo, ogni unità che aveva rilevato una collisione "ritenta" la trasmissione con gli stessi criteri. Dal momento che il tempo che passa prima della ritrasmissione è casuale,[23] la probabilità che si verifichi una nuova collisione è piuttosto bassa.

Occorre osservare che il metodo CSMA/CD funziona bene se il flusso di informazioni sulla rete non è troppo elevato rispetto alla capacità massima della rete stessa. Quando il flusso di dati aumenta, aumenta anche la probabilità di collisioni e, dal momento che ogni collisione causa un certo numero di ritrasmissioni, ciò fa aumentare ulteriormente il flusso di dati. Questo può portare al blocco completo della rete, se non vengono prese precauzioni opportune.

Tuttavia, i vantaggi offerti da questo metodo, la cui implementazione più famosa è la rete Ethernet, sono talmente notevoli che esso va progressivamente soppiantando il metodo del gettone.

4.5. PROBLEMI ED ESERCIZI

Problema 4.1.*

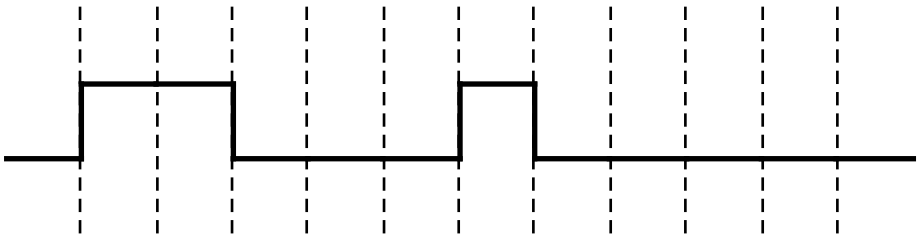
La figura che segue rappresenta il segnale presente su una linea seriale asincrona. Il carattere di 8 bit trasmesso è

a. 11000100;

---

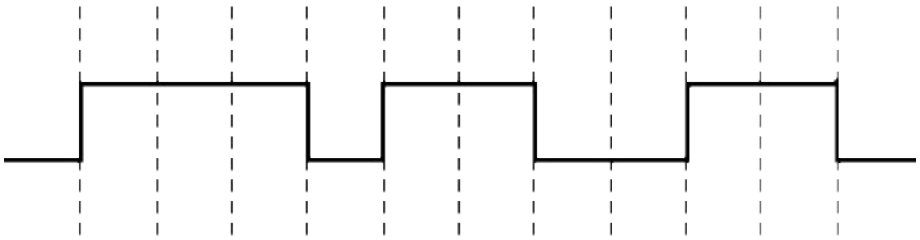
[23] Esiste un sofisticato meccanismo per ottimizzare questi tempi di attesa, in funzione del carico della rete.

b. 00010001;
c. 10001000;
d. non si può determinare: c'è un errore.



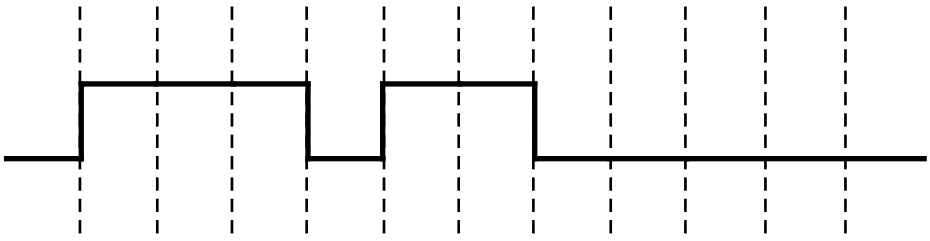
Problema 4.2.*
La figura che segue rappresenta il segnale presente su una linea seriale asincrona. Il carattere di 8 bit trasmesso è

a. 11101100;
b. 10011011;
c. 11011001;
d. non si può determinare: c'è un errore.



Problema 4.3.*
La figura che segue rappresenta il segnale presente su una linea seriale asincrona. La velocità di tale linea è di

a. 110 baud;
b. 300 baud;
c. 1200 baud;
d. 14400 baud.



Problema 4.4.
La trasmissione dei dati a brevissima distanza (pochi centimetri) avviene normalmente in forma parallela?

Problema 4.5.*
La trasmissione dei dati su linee telefoniche avviene normalmente in forma parallela?

Problema 4.6.*
La velocità media di una linea seriale asincrona su cui vengono trasmessi caratteri a 1200 baud è di
a. 171,43 caratteri/s;
b. 150 caratteri/s;
c. 133,33 caratteri/s;
d. 120 caratteri/s;
e. 109,09 caratteri/s;
f. non si può determinare: dipende anche da altri fattori.

Problema 4.7.
Quanti bit occorrono, globalmente, per trasmettere su una linea seriale asincrona 1500 parole di 8 bit con controllo di parità pari alla velocità di 9600 baud?

Problema 4.8.*
Calcolare il checksum (a 8 bit) per il seguente blocco di dati:
01000010
01110101
10111011
00000000
10111100

Problema 4.9.
Calcolare il bit di parità pari per le parole che seguono:
a. 10011101;
b. 00000000;
c. 11110000;
d. 11101010.

Problema 4.10.*
Il bit di parità dispari permette di rilevare un qualunque numero di errori di trasmissione?

Problema 4.11.*
Il bit di parità dispari permette di correggere un qualunque numero di errori di trasmissione?

Problema 4.12.*
Il bit di parità pari può essere usato nella trasmissione parallela dei dati?

Problema 4.13.*
Il checksum permette di correggere errori di trasmissione?

Problema 4.14.*
Se non vengono effettuati troncamenti, il checksum di un blocco di 256 byte può avere il valore massimo
a. -256;
b. -512;
c. -65280;
d. -65536.

Problema 4.15.*
A parità di quantità di informazione "utile" circolante, una rete a gettone deve trasportare più o meno informazione di una rete Ethernet?

Problema 4.16.*
Ha senso utilizzare Ethernet per un collegamento punto a punto fra un calcolatore e un terminale?

Problema 4.17.*
Il token ha senso in un collegamento punto a punto fra un calcolatore e un terminale?

Problema 4.18.
In una rete Ethernet, quando viene rilevata una collisione, i messaggi che hanno colliso vengono ritrasmessi immediatamente, dopo un tempo fisso o dopo un tempo variabile?

5

L'INTERFACCIA CON L'UTENTE

Un punto fondamentale dei sistemi operativi è costituito dalla cosiddetta *interfaccia utente*, cioè da quell'insieme di procedure che consentono all'uomo di comunicare con la macchina. Nella nostra trattazione considereremo solo sistemi interattivi. L'interfaccia utente riguarda direttamente sia il sistema operativo, sia i programmi applicativi.
Esistono due tipi fondamentali di interfaccia: quella che fa riferimento alla cosiddetta *programmazione modale* (*modal*) (ad esempio MS-DOS), e quella che utilizza la *programmazione non modale* (*modeless*) (tipo WINDOWS).

5.1. LA PROGRAMMAZIONE MODALE

È basata sul concetto di *comando*. Consideriamo come esempio un editor.
All'inizio dell'esecuzione, il programma stampa una lista delle possibili opzioni fra cui l'utente può scegliere quella che di volta in volta vuole usare. Se si vuole inserire del testo si sceglie l'opzione corrispondente. Il sistema a questo punto è pronto ad accettare caratteri. Al termine, occorre dirgli che l'inserzione di nuovi caratteri è terminata. Questa funzione di solito è attribuita al tasto ESC. Fatto questo, il calcolatore torna allo stato precedente e aspetta che gli venga fornito un nuovo comando. Se proviamo a tradurre tutto ciò in uno schema a blocchi otteniamo quanto mostrato in figura 5.1.

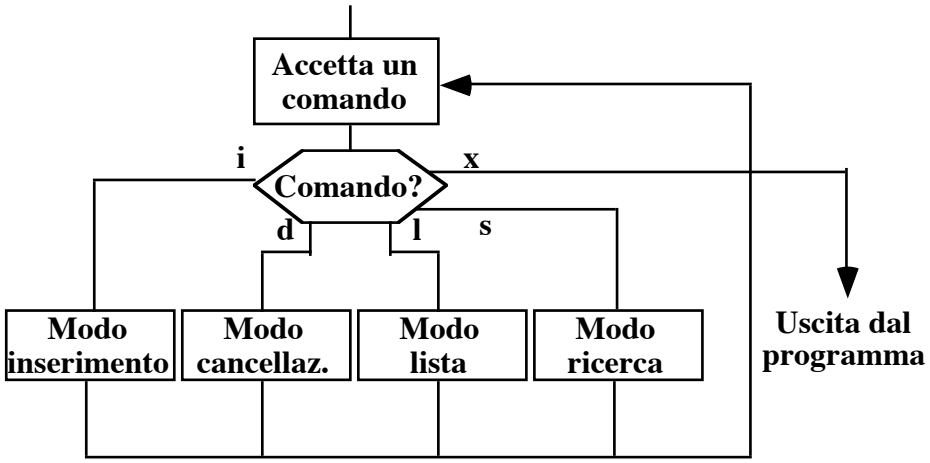


Figura 5.1. - Schema a blocchi di un editor modale.

La codifica dello schema di figura 5.1 porta ad un programma del tipo:

```
program editor;

  var
    answer: char;

begin

  repeat
    write('Comando? ');
    readln(answer);
    case answer of
      'i':
        insert;
      'd':
        delete;
      'l':
        list;
      's':
        search;
      'x':
        ;
      otherwise
        writeln('Comando non valido');
    end;
  until answer = 'x';
end;
```

dove `insert`, `delete`, ecc. sono procedure, ognuna delle quali compie le operazioni specificate, e da cui si esce solo con un apposito comando dato dall'utente, oppure, come nel caso della procedura `list`, al termine della loro esecuzione.
La programmazione modale trovava la sua naturale applicazione in quei sistemi in cui la comunicazione con l'utente si svolgeva attraverso telescriventi o semplici terminali video alfanumerici. La successiva introduzione di terminali più sofisticati (video grafici, a colori) e soprattutto di sistemi diversi per l'introduzione dei dati (primo fra

tutti il mouse) da una parte, e le aumentate capacità del software dall'altra, hanno favorito lo sviluppo di un diverso tipo di interfacciamento, che fa riferimento alla cosiddetta *programmazione non modale* (*modeless*).

5.2. LA PROGRAMMAZIONE NON MODALE

Il concetto base è che, al contrario di quanto avveniva nella programmazione modale, tutto ciò che il programma può fare è disponibile contemporaneamente.
Per descrivere il funzionamento e i metodi di implementazione della programmazione non modale faremo riferimento a quanto avviene nel sistema operativo dei calcolatori Apple Macintosh™ (Macintosh OS), i quali, pur presentando alcune fondamentali carenze dal punto di vista della multiprogrammazione, hanno il vantaggio, rispetto ad altri sistemi, di essere gestiti in maniera estremamente lineare, e quindi di facile comprensione.
È importante tuttavia sottolineare che la semplice lettura di un testo non può dare altro che una conoscenza estremamente superficiale delle modalità di programmazione di un sistema non modale. L'unico (e fortemente raccomandato) metodo di apprendimento consiste nella sperimentazione pratica su calcolatore, che può essere effettuata utilizzando appositi programmi-esempio che tutti i produttori di compilatori mettono a disposizione, e che possono essere studiati solo sulla macchina, modificandoli ed integrandoli secondo le istruzioni date dai manuali e, in qualche caso, dagli stessi programmi.

5.2.1. L'INTERFACCIA UTENTE IN UN SISTEMA DI PROGRAMMAZIONE NON MODALE

Rimandando alla seconda parte di questo capitolo la trattazione più dettagliata dei vari componenti di un'interfaccia uomo-macchina in un sistema non modale, è opportuno mettere fin da ora in evidenza che, nei moderni calcolatori, tale interfaccia è estremamente complessa. Senza addentrarci in discussioni sui sistemi multimediali, osserviamo che già nei casi più semplici l'utente si trova di fronte ad uno schermo "pieno" di informazioni, come quello che appare in figura 5.2, che rappresenta lo schermo del calcolatore mentre veniva preparata questa pagina.

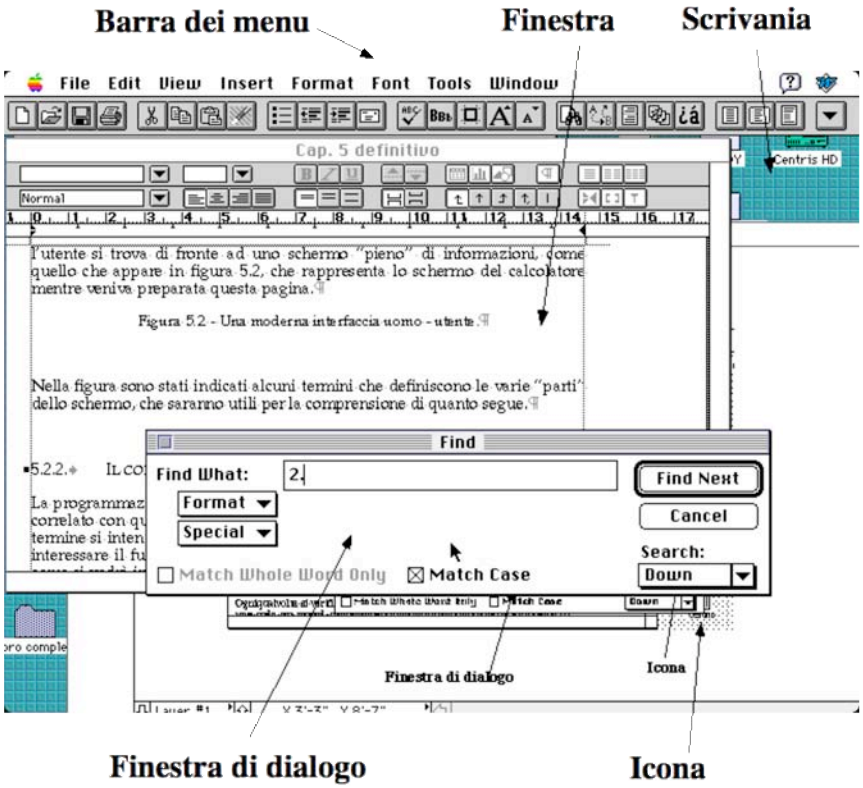


Figura 5.2. - Una moderna interfaccia uomo - utente.

Nella figura sono stati indicati alcuni termini che definiscono le varie “parti” dello schermo, che saranno utili per la comprensione di quanto segue.

5.2.2. IL CONCETTO DI EVENTO

La programmazione non modale è basata sul concetto di *evento*, strettamente correlato con quello di interruzione (hardware o software), dove con questo termine si intende in modo generico il fatto che è successo qualcosa che può interessare il funzionamento del programma. Ad ogni evento corrisponde, come si vedrà in seguito, la generazione di un opportuno messaggio.
Ogniqualvolta si verifica una interruzione, il sistema operativo inserisce in una coda un record, che descrive appunto l’evento che si è verificato. È importante osservare che gli eventi vengono gestiti completamente dal sistema operativo, senza alcun intervento da parte dei programmi applicativi, che si limitano a sfruttarli.
Il record che definisce l’evento ha la seguente struttura:

```
type

point=record          {utilizzato per definire la posizione di un
                        punto sullo schermo}
  x,y:integer;
end;
```

```
EventRecord=record
  What:integer;      {event code: tipo di evento, ad esempio tasto
                       premuto}
  Message:longint; {event message: informazione associata al
                       messaggio, ad esempio codice del tasto
                       premuto}
  When:longint;      {ticks since startup[24]: istante in cui si è
                       verificato l'evento}
  Where:point;       {mouse location: posizione del mouse nel
                       momento in cui si è verificato l'evento}
  Modifiers:integer; {modifier flags: ulteriori eventuali
                       informazioni relative all'evento}
end;
```

A titolo di esempio, sono qui di seguito riportati alcuni possibili tipi di eventi.

```
const
nullevent=0;     {ogni tic (60 volte al secondo) viene generato un
                   nullevent se non è successo nient'altro}
mousedown=1;     {il pulsante del mouse è stato premuto}
mouseup=2;       {il pulsante del mouse è stato rilasciato}
keydown=3;       {è stato premuto un tasto della tastiera}
keyup=4;         {è stato rilasciato un tasto della tastiera}
autokey=5;       {un tasto della tastiera è stato premuto tanto
                   a lungo da iniziare la funzione di autorepeat}
updateevt=6;     {occorre ridisegnare una zona dello schermo che
                   prima era coperta}
diskevt=7;       {è stato inserito un floppy disc)
activateevt=8; {una finestra dello schermo è diventata attiva}
networkevt=9;    {si è verificato un evento sulla rete di
                   comunicazione}
```

5.2.3. L'UTILIZZAZIONE DEGLI EVENTI

Una volta che gli eventi sono stati acquisiti dal sistema operativo, essi devono essere utilizzati o dal sistema operativo stesso, o dai programmi applicativi che stanno girando. Nella filosofia del Macintosh OS, tutti gli eventi che si sono verificati sono resi disponibili al programma applicativo, il quale deciderà se sfruttarli, gestendoli opportunamente, oppure no. In quest'ultimo caso, essi possono essere nuovamente resi disponibili al sistema operativo per la loro gestione.
Mentre i programmi modali in ogni fase accettano solo alcuni eventi previsti, i programmi non modali devono in qualunque istante essere in grado di gestire qualunque cosa succeda. Lo schema a blocchi corrispondente è mostrato in figura 5.3, dove i blocchi indicati genericamente con "gestisci" provvedono ad effettuare le operazioni appropriate per ogni tipo di evento che si sia verificato.

---

[24] Nel Macintosh, il *tick* è l'unità fondamentale di tempo, corrispondente all'intervallo fra due successive interruzioni dell'orologio in tempo reale, e corrisponde a 1/60 di secondo.

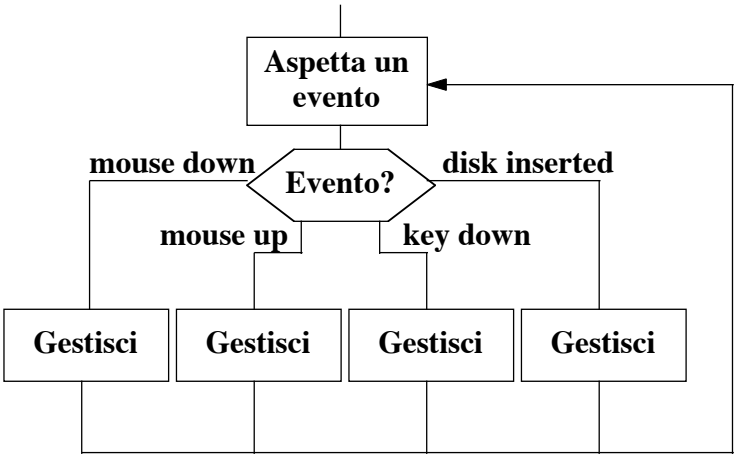


Figura 5.3. - Schema a blocchi di un sistema di programmazione non modale.

È fondamentale osservare che, mentre in un programma modale, una volta entrati nella gestione di un particolare modo di funzionamento, se ne esce solo su comando dell'utente, in questo caso la gestione di ogni evento deve essere quanto più rapida possibile, in modo da poter tornare immediatamente alla gestione dell'evento successivo.
Questo metodo, sebbene abbia notevoli vantaggi dal punto di vista dell'interfaccia con l'utente, presenta diverse complicazioni per quel che riguarda la gestione degli eventi, perché il significato di ogni evento è in genere legato al contesto in cui esso si è verificato. In altre parole, non è possibile stabilire un unico sottoprogramma di gestione per ogni evento, in quanto il suo significato può variare in dipendenza di diversi altri fattori.
Per chiarire meglio il concetto, analizzeremo nel dettaglio la gestione di un singolo evento. Supponiamo per esempio che l'evento sia di tipo *mousedown*, cioè che l'utente abbia premuto il pulsante del mouse.
Una volta stabilito che l'evento da gestire è di tipo *mousedown*, il controllo passa al blocco relativo (il primo a sinistra nello schema di figura 5.3). Dal momento che il significato di un evento mousedown dipende dalla posizione in cui si trovava il mouse al momento in cui si è verificato, occorre un secondo blocco di selezione che trasferisca il controllo alla procedura appropriata in relazione alla posizione del mouse, come si può vedere in figura 5.4.

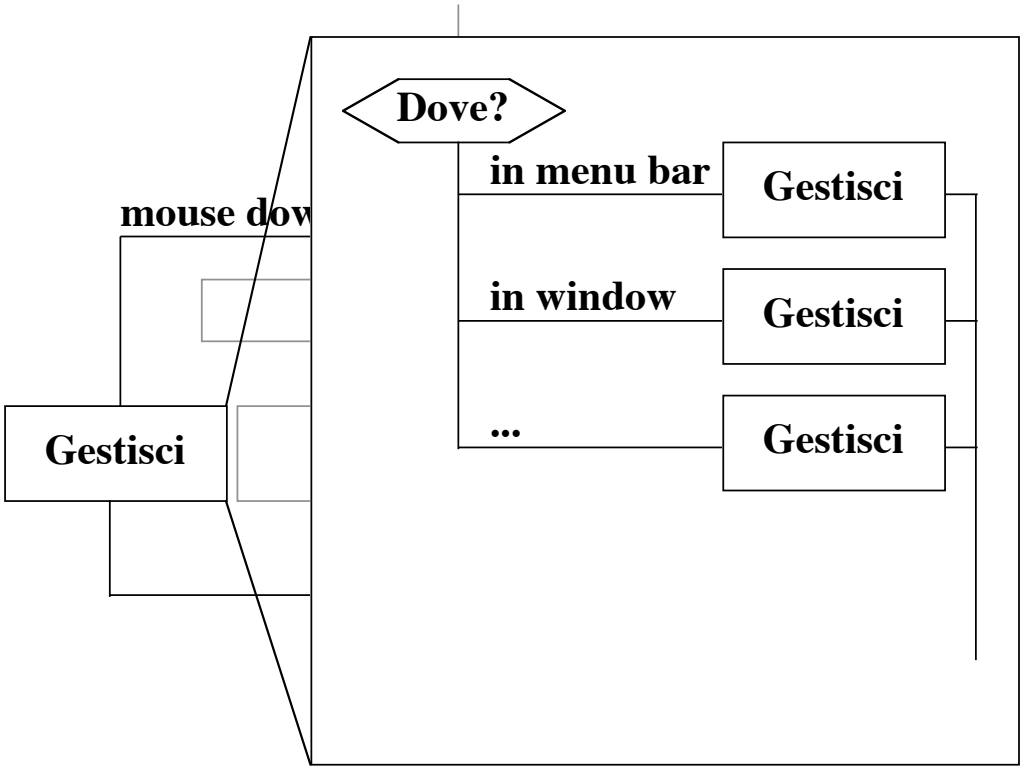


Figura 5.4. - La gestione di un evento mousedown.

Supponiamo che l'evento si sia verificato mentre il mouse si trovava sopra la barra dei menu: occorre allora una ulteriore selezione (Fig. 5.5), per determinare quale menu sia stato scelto dall'utente; ancora una volta una selezione smisterà il controllo fra diverse procedure a seconda del menu prescelto (primo, secondo ecc.). Infine, bisognerà con un'ulteriore selezione dirigere l'esecuzione del programma a seconda dell'item (elemento del menu) che è stato selezionato. Qualcosa di analogo succede anche per tutti gli altri eventi. Questa serie di blocchi di selezione nidificati permettono di gestire tutti i possibili eventi che si possono generare.

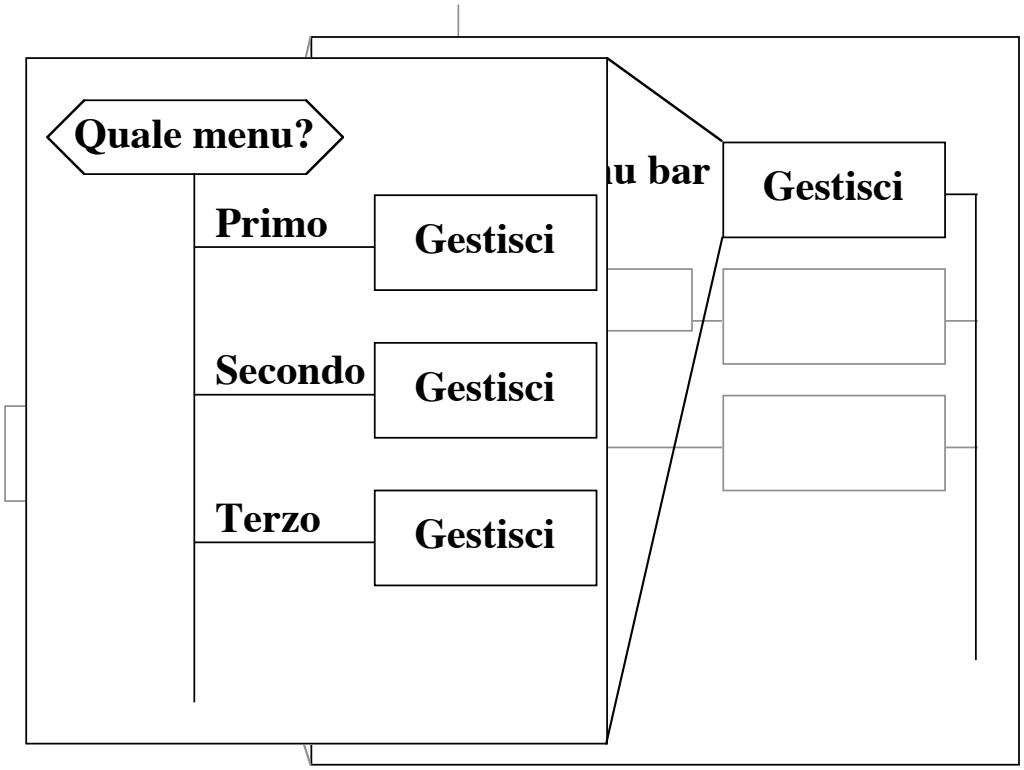


Figura 5.5. - La gestione di un evento mousedown.

5.2.4. UN ESEMPIO DI PROGRAMMA NON MODALE

Vediamo ora come quanto esposto nel paragrafo precedente possa tradursi in un programma. Utilizzeremo come esempio un programma "generico", in cui le operazioni da eseguire in seguito al verificarsi degli eventi non sono definite, perché questo non ha importanza ai fini della trattazione. A seconda poi delle necessità, un programma come quello che viene presentato può essere completato opportunamente, e costituisce quindi lo scheletro di qualsiasi programma non modale.
Per semplicità, è stata omessa la dichiarazione delle variabili globali.

5.2.4.1. Il programma principale

In un programma non modale il programma principale è sempre estremamente semplice, e contiene solo la chiamata a tre procedure:

```
program main;
begin {main}

  initall;

  maineventloop;

  closeall;

end.{main}
```

Le funzioni delle tre procedure sono le seguenti:
`initall`: contiene tutto ciò che serve ad inizializzare il programma: allocazioni di memoria dinamica, inizializzazione di variabili globali, apertura di file e di canali di comunicazione, ecc.;
`maineventloop`: costituisce la procedura che gestisce gli eventi, ed è quindi il corpo del programma: come vedremo, essa è costituita da un ciclo che viene ripetuto continuamente, finché non si determinano le condizioni per la terminazione del programma;
`closeall`: conclude il programma chiudendo i file aperti e i canali di comunicazione, liberando la memoria dinamica allocata, ecc.

5.2.4.2. La procedura `maineventloop`

Dal momento che il programma applicativo non può accedere direttamente alla coda degli eventi, è disponibile una funzione di sistema (`GetNextEvent`) che, se la coda non è vuota, restituisce il primo evento in essa contenuto.
Tale funzione è definita come

```
function GetNextEvent(EventMask:byte; var Event:eventrecord):boolean;
```

e ritorna il valore `true` se nella coda c'è almeno un evento disponibile.
`EventMask` è un parametro il cui valore permette di specificare a quali tipi di evento il programma è interessato: esso permette cioè di escludere gli eventi il cui verificarsi non influisce sul funzionamento del programma.
`Event` è il parametro in cui viene posto il primo evento disponibile nella coda, se ne esiste almeno uno di un tipo specificato dalla `EventMask`.

```
procedure maineventloop;

  var
    myevent: eventrecord;
    code: integer;
    whichwindow: windowptr;
    whichwindow1: windowpeek;
    thechar: char;
    doneflag: boolean;
    savePort: GrafPtr;
    whichcontrol: controlhandle;
    mycontrol: integer;


begin {maineventloop}

  doneflag := false; {quando diventerà true la maineventloop avrà
                        termine}

  repeat {until doneflag}
    if GetNextEvent(everyEvent, myEvent) then
      begin
        case myEvent.what of

          mouseDown:
            handlemousedown;

          keyDown, autoKey:
            handlekeydown;

          activateEvt:
            handleactivateevt;

          updateEvt:
            handleupdateevt;

          otherwise
            ;
        end {of event case}
      end;

  until doneFlag
end; {maineventloop}
```

Come già detto, il programma ripete continuamente il ciclo e, se c'è un evento disponibile, passa il controllo alla procedura deputata alla gestione di tale evento. Per determinare di volta in volta di che tipo di evento si tratti si esamina il campo `what` del record, che contiene appunto il tipo di evento.
Esamineremo ora in dettaglio la procedura `handlemousedown`.

5.2.4.3. La procedura `handlemousedown`

Dal momento che lo schermo del calcolatore contiene diverse finestre, e che ad ogni finestra è associato un gran numero di "pulsanti" e di controlli, questa procedura è ovviamente molto complessa. Diverse procedure e funzioni di sistema rendono però il compito del programmatore più agevole: ad esempio, la funzione `FindWindow` permette, utilizzando il campo `where` del record che contiene l'evento, di

determinare se esso si è verificato quando il mouse era sulla barra dei menu, sulla scrivania (la parte dello schermo non occupata da alcuna finestra), o su una finestra. L'esempio qui di seguito riportato è incompleto, per mantenerne la comprensibilità ad un livello accettabile.

```
procedure handlemousedown;
begin
  code := FindWindow(myEvent.where, whichWindow);
  case code of

    inMenuBar:
      handlemenu(myevent, doneflag);

    inSysWindow:
        SystemClick(myEvent, whichWindow) {restituisce
                    temporaneamente il controllo
                    al sistema operativo per la gestione degli
                    eventi relativi alla scrivania, che non
                    interessano certamente il programma utente}

    inDrag:
        DragWindow(whichWindow, myEvent.where, dragRect) {sposta
                    la finestra sulla scrivania}

    inGoAway:
          hidewindow(whichwindow); {nasconde una finestra}

    inContent:
      begin
        {Varie azioni, dipendenti da qual è la finestra su cui
        si è verificato l'evento}
      end; {in Content}

    otherwise
      ;
  end
end;
```

5.2.4.4. La procedura `handlemenu`

Particolarmente interessante è la procedura `handlemenu`, che viene chiamata quando l'evento `mousedown` si è verificato quando il mouse si trovava sulla barra dei menu.
I menu a tendina sono costituiti da una serie di parole chiave (`menuname` nell'esempio che segue). Queste parole appaiono nella parte superiore dello schermo. Quando si preme il pulsante su una di queste parole, appare sotto di essa una serie di altre parole (`menuitem`) fra cui, spostando opportunamente il mouse, l'utente può scegliere quella che gli interessa. La gestione del processo appena descritto è effettuata dal sistema operativo; il programma utente deve quindi solo determinare quali sono il menu e l'item selezionati, e comportarsi di conseguenza.

```
procedure HandleMenu (myevent: eventrecord; var doneflag: boolean);

begin
```

```
  menuname := hiword(menuselected);
  menuitem := loword(menuselected); {Ogni menu ha una serie di
                                     "item", cioè di comandi dei
                                     quali uno è quello
                                     selezionato}

  case menuname of
    1:
      case menuitem of
        1:
          {gestisce il primo item del primo menu}

        otherwise
          begin
            getitem(applemenuhandle, loword(menuselected), name);
            refnum := opendeskacc(name);      {Gli altri item del
                      primo menu si riferiscono a funzioni del
                      sistema operativo, a cui viene demandata
                      la gestione}
          end;
      end;

    2: {Secondo menu}
      case menuitem of
        1: {Primo item del secondo menu}
          {gestione del primo item del secondo menu}
;
        2: {Secondo item del secondo menu}
          {gestione del secondo item del secondo menu}
;
        3: {Terzo item del secondo menu}
          {gestione del terzo item del secondo menu}
;
        4:
          doneflag := true;   {Questo è l'item che causa la
                               terminazione del programma}
        otherwise
      end;
    3: {Terzo menu}

……

      end;

    otherwise
  end;
end;{handlemenu}
```

Un programma così concepito può risultare complesso, ma i vantaggi per l'utente sono notevoli. Infatti la versatilità di un'interfaccia a finestre che una programmazione di questo tipo gli offre è immensamente superiore a quella delle tradizionali interfacce modali.

5.2.5. LE FINESTRE

Dopo aver visto il meccanismo che permette la gestione degli eventi in un sistema non modale, esaminiamo i componenti più importanti dell'interfaccia stessa, iniziando dalle finestre, di cui un esempio è illustrato in figura 5.6.

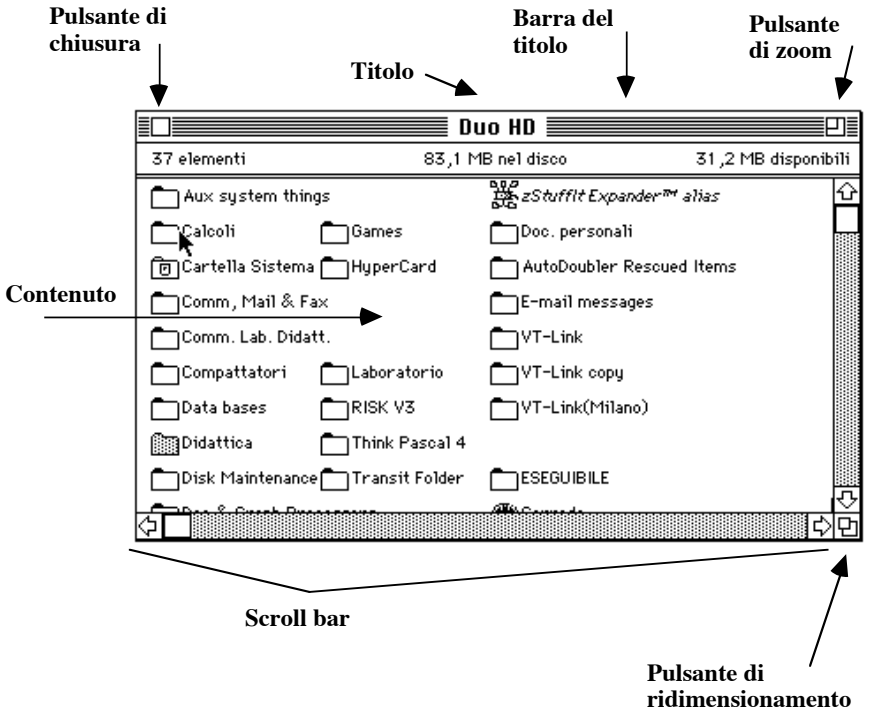


Figura 5.6. - Un esempio di finestra.

Una finestra presenta in genere:

- un *pulsante di chiusura* che può servire a molteplici scopi: far scomparire la finestra dallo schermo, chiudere il file su cui si sta operando, chiudere anche il programma in esecuzione oppure ridurre la finestra a icona, cioè ad una rappresentazione miniaturizzata della finestra (chiaramente non si legge su di essa il contenuto della finestra: per far ciò occorre premere il pulsante del mouse su di essa e la finestra ricomparirà mentre l'icona si cancella);
- il *titolo* (nome della finestra);
- la *barra del titolo* che serve a spostare la finestra dove si vuole (tenendo il pulsante premuto su di essa e spostando il mouse si vedrà la finestra seguire i suoi spostamenti);
- il *contenuto della finestra*, che in questo caso è un insieme di file;

- un *pulsante di zoom* che trasforma le dimensioni attuali della finestra in quelle massime consentite dallo schermo;
- un *pulsante di ridimensionamento* con il quale si può ingrandire o rimpicciolire a piacere la finestra;
- due *scroll bar* che permettono di risolvere il seguente problema: mentre la finestra può al massimo assumere le dimensioni dello schermo, l'ideale foglio che essa rappresenta potrebbe anche essere più grande. Lo scopo degli scroll bar è proprio quello di spostare idealmente la finestra sopra l'ipotetico foglio. Se si preme sulle frecce si ottengono piccoli spostamenti, se si preme sulle barre grigie si ottengono spostamenti più grandi e se si tiene premuto sul quadratino e si sposta il mouse si può portare la finestra in qualunque posizione sopra il foglio stesso.

5.2.6. LE FINESTRE DI DIALOGO

5.2.6.1. Finestre di dialogo non modali

Le finestre illustrate nel paragrafo precedente contengono in generale testo o disegni. In alcune situazioni è però necessario avere a disposizione finestre di tipo diverso, di cui la figura 5.7 dà un esempio.

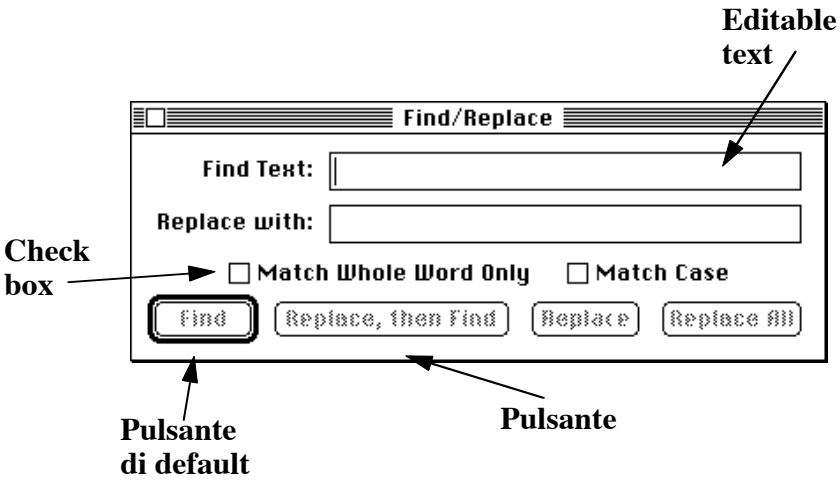


Figura 5.7. - Un esempio di dialogo non modale.

Queste finestre vengono chiamate *finestre di dialogo non modali*, e il loro principale scopo è quello di permettere uno scambio di informazioni fra il calcolatore e l'utente secondo un formato molto standardizzato.
Esse presentano all'utente testo e disegni, e possono contenere:

- righe di *editable text* nelle quali possono essere inseriti caratteri alfanumerici;
- tre tipi diversi di pulsanti:
  button, che sono pulsanti veri e propri;
  check box, che possono essere assimilati ad interruttori ai quali vengono associate variabili binarie;
  radio button, che sono insiemi di più pulsanti, uno solo dei quali può essere attivo in un determinato istante.

5.2.6.2. Finestre di dialogo modali

I tipi di finestre visti finora non sono ancora sufficienti: per la gestione di eventi prioritari (che richiedono cioè una risposta dall'utente prima che il processo possa proseguire) vengono usate le *finestre di dialogo modali* (Fig. 5.8).

Figura 5.8. - Un esempio di dialogo modale.

Mentre le finestre finora viste avevano il pulsante di chiusura (e perciò potevano essere fatte scomparire dallo schermo in qualunque istante), quelle non modali, che non hanno tale pulsante, obbligano l'utente a dare una risposta prima di poter fare qualunque altra cosa. Esse infatti scompaiono automaticamente dallo schermo solo dopo che l'utente ha "premuto" uno dei pulsanti che vi appaiono.
Nell'esempio della figura 5.8, dopo aver tentato di copiare un file là dove già ne esisteva uno, il calcolatore chiede se lo debba sostituire o se debba annullare l'istruzione di copiatura.

5.2.6.3. Gli *alert*

Altro esempio di finestra modale è l'*alert* (Fig. 5.9) nel quale addirittura si deve solo dare la conferma di aver recepito il messaggio del calcolatore.

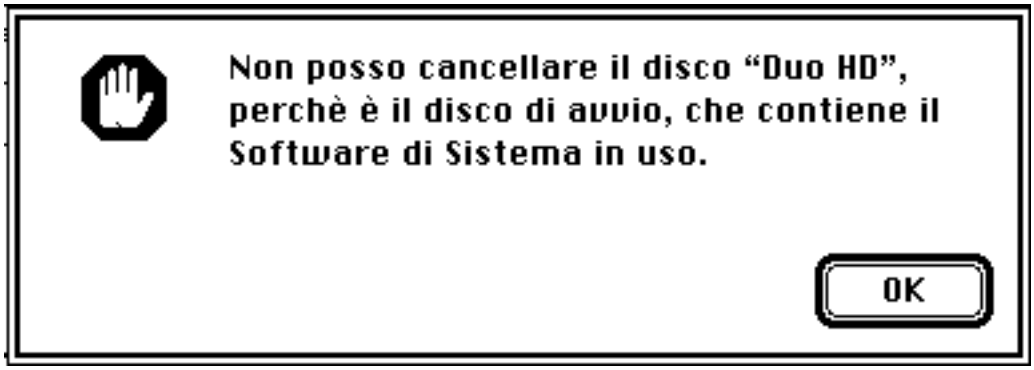


Figura 5.9. - Un *alert*.

Nell’esempio della figura 5.9, dopo un tentativo di formattazione del disco fisso, il calcolatore ha avvisato l’utente che non è possibile cancellare il software di sistema in uso.

5.3. CONCLUSIONI

È opportuno a questo punto riassumere i concetti fondamentali visti in questo capitolo. La programmazione non modale:

- permette maggiore flessibilità nelle operazioni (il sistema non modale reagisce a qualunque cosa succeda al calcolatore);
- richiede meno sforzo mnemonico all’utente (può mostrare molte cose sullo schermo, per cui non c’è bisogno di impararle a memoria);
- permette una maggiore facilità di apprendimento (un sistema non modale ben congegnato può essere utilizzato anche senza il manuale d’uso, ed è possibile costruire efficacissimi manuali *on line*, cioè consultabili sullo stesso schermo su cui appare il programma);
- richiede un maggiore sforzo programmativo nella scrittura dei programmi applicativi (maggior comprensione delle funzioni del sistema operativo);
- in genere è meno efficiente della programmazione modale, ma tale inefficienza è di solito trascurabile rispetto ai vantaggi che ne derivano.

Per quanto riguarda la suddivisione delle funzioni fra il sistema operativo e il programma applicativo, ricordiamo che:

- il sistema operativo
  - gestisce le interruzioni;
  - le trasforma in eventi;
  - inserisce gli eventi in una coda;

    - rende disponibili gli eventi al programma applicativo;
- il programma applicativo
    - gestisce gli eventi che gli interessano, lasciando nella coda quelli relativi ad altri programmi eventualmente presenti.

5.4. PROBLEMI ED ESERCIZI

Problema 5.1.*
I programmi modali devono continuamente tenere sotto controllo tutti gli eventi che si verificano (pressione di tasti, spostamenti del mouse, inserzioni di dischetti, ecc.)?

Problema 5.2.*
I programmi modali sono più facili da scrivere di quelli non modali?

Problema 5.3.*
I programmi non modali devono continuamente tenere sotto controllo tutti gli eventi che si verificano (pressione di tasti, spostamenti del mouse, inserzioni di dischetti, ecc.)?

Problema 5.4.*
La programmazione non modale serve principalmente:

a. a rendere più veloce l'esecuzione dei programmi;
b. a svincolare l'utente da procedure rigidamente codificate, permettendogli di interagire con il calcolatore in modo più libero;
c. ad aumentare l'*intelligenza* dei calcolatori.

6

I SISTEMI IPERTESTUALI

Da alcuni anni il prefisso *iper* è entrato di prepotenza in molti aspetti della nostra vita di tutti i giorni. In maniera più o meno impropria, parole come *ipermercato* e *iperteso* vengono riproposte in continuazione. Una delle ultime in ordine di apparizione è la parola *ipertesto*, che costituisce l'argomento del presente capitolo.

6.1. IL CONCETTO DI IPERTESTO

Per introdurre il concetto di ipertesto, possiamo considerare la differenza che passa tra un romanzo ed un quotidiano. Entrambi sono costituiti da un testo stampato su carta. In alcuni casi al testo possono essere abbinate delle figure.
Esiste però una differenza fondamentale: mentre il romanzo è pensato per essere letto *sequenzialmente*, cioè partendo dalla prima pagina e terminando con l'ultima, il giornale, impaginato secondo le tendenze moderne, non prevede un ordine particolare di lettura: non solo, ma la prima pagina contiene la massima quantità possibile di argomenti, nel senso che in essa sono citate tutte le notizie più importanti contenute nel fascicolo. Dal momento però che lo spazio disponibile sulla sola prima pagina non è sufficiente, esse sono citate in forma succinta, rimandando il lettore ad ulteriori articoli, stampati nelle pagine interne, che espandono ed integrano quanto accennato nei trafiletti di prima pagina. L'ordine di lettura del giornale, a differenza di quello del romanzo, è quindi *casuale*, nel senso che ogni lettore, osservando la prima pagina, individua gli argomenti che gli interessano; per ognuno di essi, legge l'articolo (o gli articoli) relativi nelle pagine interne, poi ritorna alla prima pagina per cercare altre

notizie interessanti, e ripete questo procedimento fino ad aver esaurito l'interesse (o il tempo a disposizione).[25]
Mentre abbiamo la ragionevole certezza che un romanzo sarà letto nello stesso modo da tutti, ognuno legge il giornale in maniera differente.
Un po' approssimativamente, utilizzando l'esempio appena fatto, potremmo dire che un ipertesto è semplicemente, come un giornale, una raccolta di testi non sequenziali. Non è neppure necessario che in un ipertesto le varie parti siano strettamente correlate fra di loro: nel caso del giornale il legame è rappresentato solo dal fatto che le notizie riportate sono in qualche modo attuali, ma possono riguardare fatti, luoghi e persone che non hanno alcun rapporto le une con le altre.

6.2. GLI IPERTESTI NEI CALCOLATORI

Come accade per molte cose relative all'informatica, il concetto di ipertesto è stato espresso molto tempo prima della nascita dei calcolatori elettronici. Il primo ricercatore che ha intuito l'importanza di organizzare l'informazione secondo quelli che sono gli attuali metodi usati negli ipertesti è stato Vannevar Bush che, fin dal 1945, metteva in evidenza la necessità di disporre di strumenti di ricerca associativa, fatti in modo tale da poter passare da un elemento ad un altro in maniera diretta, senza ricorrere ad indici.
Ovviamente, questi meccanismi sono stati resi possibili solo dal miglioramento della tecnologia dei calcolatori, tanto che dobbiamo attendere il 1960 per vedere la nascita del primo sistema ipertestuale, ad opera di Ted Nelson che, per inciso, ha anche coniato la parola *hypertext* che ora è diventata di uso così comune.
La nascita dei calcolatori personali ha poi dato grande impulso allo sviluppo dei sistemi ipertestuali, mettendoli a disposizione di una larga fascia di utenti. La prima casa costruttrice che si è mossa in questo senso è stata Apple, che ha creato HyperCard™ per la linea dei suoi calcolatori Macintosh. HyperCard è descritto in dettaglio nel prossimo capitolo.

6.3. GLI IPERTESTI IN PRATICA

Per chiarire meglio il concetto di ipertesto, consideriamo come secondo esempio un ipotetico codice di leggi. Come ogni codice, esso è costituito da un certo numero di articoli, raggruppati in un libro diviso in una serie di capitoli dedicati ognuno ad un diverso argomento. Ogni capitolo contiene un certo numero di articoli, ed ogni articolo è diviso in commi, come mostrato in figura 6.1.
La ricerca delle informazioni può essere fatta sfruttando il fatto che gli articoli sono disposti sequenzialmente. In genere esiste anche un indice che riporta, per ogni

---

[25] Ovviamente ci sono anche altri modi per leggere il giornale, ad esempio sfogliandolo. Il lettore frettoloso però userà di preferenza il metodo descritto nel testo.

articolo, un riassunto sintetico e la pagina in cui esso è stampato. Questo indice tuttavia, anche se costituisce un valido aiuto alla consultazione del testo, non contiene alcuna informazione aggiuntiva, tanto è vero che esso viene compilato semplicemente estraendo alcune informazioni dal testo secondo un algoritmo molto semplice.
L'organizzazione della materia è quindi strettamente gerarchica. Per i nostri scopi è più conveniente rappresentarla con un albero, secondo quanto mostrato in figura 6.2.
Se ogni comma fosse indipendente da ogni altro, la strutturazione dell'informazione appena vista sarebbe sufficiente a garantire una consultazione completa. Invece, sappiamo bene che in campo legale (ma il ragionamento vale, come vedremo, anche in molti altri campi) molto spesso le leggi fanno riferimento ad altre leggi. Per conoscerne una non è quindi sufficiente leggere gli articoli che la compongono, ma bisogna anche consultare altre parti del testo che contengono gli articoli a cui essa fa riferimento, i quali a loro volta possono fare riferimento ad ulteriori articoli. Sarebbe quindi più corretto, per rappresentare i legami fra le informazioni, utilizzare un grafo, e non più un albero, come esemplificato in figura 6.3.
In quest'ultima figura sono stati indicati mediante frecce grigie i cosiddetti *riferimenti incrociati*. È chiaro che a questo punto l'organizzazione di un libro non è più ottimale per garantire la ricerca di tutte le informazioni, perché, ogni volta che si trova un riferimento ad un altro articolo o ad un altro comma, è necessario cercare sull'indice la pagina in cui esso è contenuto. In un certo senso la lettura di un articolo comporta una serie di "salite" e di "discese" attraverso i rami dell'albero, perché non è possibile passare direttamente da una foglia all'altra.

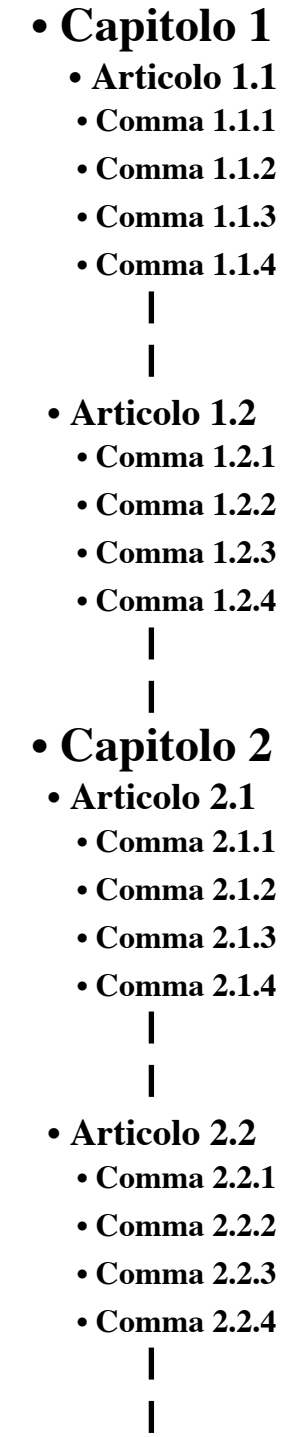


Figura 6.1. - L'organizzazione di un codice legale.

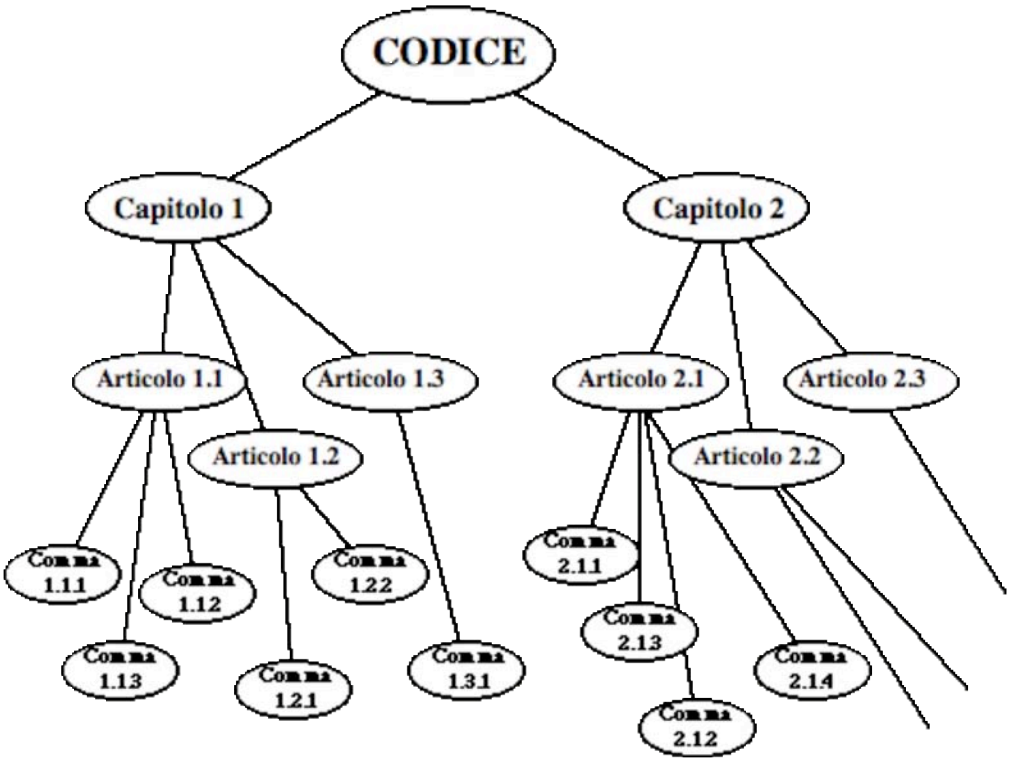


Figura 6.2. - Rappresentazione ad albero della struttura di figura 6.1.

Il meccanismo sarebbe molto più semplice se, per ogni riferimento incrociato, fosse indicata la pagina in cui esso è contenuto. I riferimenti, che in qualunque codice sono del tipo

…secondo quanto stabilito dal secondo comma dell'art. 25

dovrebbero essere quindi trasformati in frasi strutturate come la seguente:

…secondo quanto stabilito dal secondo comma dell'art. 25 [pag. 352]

Chiaramente, tutto questo può essere fatto anche in un libro stampato, anche se dobbiamo osservare che ciò non è affatto semplice se la materia di cui si tratta, come accade per le raccolte di leggi, varia nel tempo, dal momento che nuove leggi vengono promulgate e vecchie leggi sono abrogate.

Supponiamo ora che l'intero codice di cui abbiamo parlato, anziché essere stampato, sia registrato su una memoria elettronica. Supponiamo anche che esista un meccanismo di indirizzamento di ogni singolo comma tale per cui sia possibile accedervi rapidamente ed efficientemente.

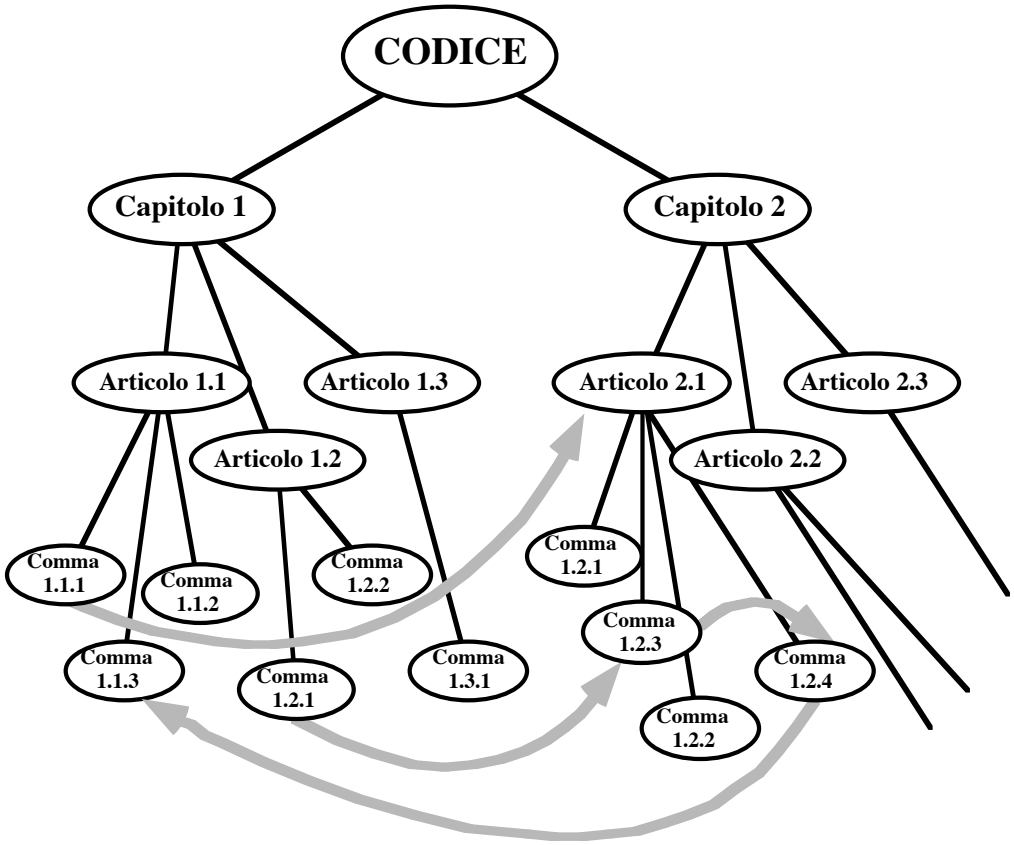


Figura 6.3. - Rappresentazione del codice mediante un grafo.

Non è difficile a questo punto pensare di poter far apparire sullo schermo una frase del tipo mostrato in figura 6.4.

```
Art. 3.1. L'abigeato è punito con la reclusione da sei mesi
a quattro anni, secondo quanto disposto dal quarto e quinto
comma dell'art. 2.14.
```

Figura 6.4. - Una schermata ipertestuale.

La differenza fondamentale rispetto ad un testo di tipo tradizionale è rappresentata dal fatto che alcune parole appaiono sottolineate, o comunque evidenziate rispetto al resto del testo. Ciò indica convenzionalmente che esse fanno riferimento ad altre informazioni.

```
Art. 2.14. Le pene detentive possono essere commutate in
sanzioni amministrative secondo i seguenti criteri:

Pene detentive di durata inferiore a 7 giorni: ammenda da L.
100.000 a L. 1.000.000;

Pene detentive da 1 a 30 giorni: ammenda da L. 1.000.000 a
L. 10.000.000;

Pene detentive da 1 a 6 mesi: ammenda da L. 10.000.000 a L.
100.000.000.

Le pene detentive di durata superiore a 6 mesi non possono
essere commutate in sanzioni amministrative.
```

Figura 6.5. - Una seconda schermata ipertestuale.

Utilizzando l'interfaccia uomo-macchina di cui abbiamo parlato nel capitolo precedente, possiamo pensare che, puntando il mouse su queste parole e premendone il pulsante, il testo sullo schermo venga sostituito da quello a cui il testo stesso fa riferimento, come si può vedere in figura 6.5.
I meccanismi di indirizzamento disponibili rendono tutto ciò possibile in maniera *automatica*, senza cioè che l'utente debba preoccuparsi in alcun modo di effettuare in prima persona la ricerca.
Da questo punto di partenza sono possibili infinite variazioni e miglioramenti. Ad esempio, una simbologia molto usata nei sistemi ipertestuali prevederebbe, nel caso in esame, il testo mostrato in figura 6.6, dove, oltre alla sottolineatura già vista, appaiono anche due parole sottolineate in modo discontinuo. Il significato della sottolineatura è: *di questa parola esiste una definizione in un glossario* e, premendo il tasto del mouse su di essa, appare in un'altra zona dello schermo una finestra contenente la definizione della parola in questione.

```
Art. 3.1.3. L'abigeato è punito con la reclusione da sei
mesi a quattro anni, secondo quanto disposto dal quarto
comma dell'art. 2.14.
```

Figura 6.6. - Una schermata ipertestuale più sofisticata.

Nell'esempio appena visto l'informazione contenuta nel sistema è suddivisa in un certo numero di elementi, che costituiscono i nodi del grafo di figura 6.3. Ad ogni nodo sono associati, senza particolari distinzioni, i collegamenti (*link*) che tale nodo

ha con altri nodi. Il meccanismo di indirizzamento deve essere fatto in modo tale che, aggiungendo ulteriori nodi, le modalità di indirizzamento dei nodi preesistenti non cambino.[26] Questo fa sì che l'inserimento di nuove informazioni comporti solamente l'aggiunta di nuovi link, ma non richieda mai la modifica di link già esistenti.
Qualche problema sorge invece quando occorre effettuare la cancellazione di un nodo. In questo caso infatti occorre esaminare tutto il contenuto del sistema, per eliminare i collegamenti al nodo che è stato eliminato.[27]

6.4. IL PROBLEMA DELLA NAVIGAZIONE

È facile immaginare che, se il sistema ipertestuale di cui ci occupiamo contiene informazioni strutturate in maniera complessa (un ottimo esempio è dato dalla giungla legislativa di alcuni Paesi), l'utente, che inizialmente cercava informazioni su un problema semplice, possa "smarrirsi" in una serie di articoli di legge che ne richiamano altri che a loro volta fanno riferimento ad altri, perdendo il filo del ragionamento e non riuscendo più a tornare al punto di partenza.
Per questa ragione, tutti i sistemi ipertestuali prevedono dei meccanismi che aiutano l'utente a ritornare al punto da cui era originariamente partito. Due sono i sistemi più comunemente usati:

a. ogni volta che viene attivato un nuovo link, l'informazione del nodo a cui esso punta viene mostrata in una nuova finestra, posta sopra a tutte le altre. Le informazioni già mostrate sullo schermo non vengono cancellate, ma solo "nascoste" da quelle apparse dopo. In questo caso, è sufficiente chiudere tutte le nuove finestre per ritornare alle informazioni precedenti;

b. il sistema memorizza tutti i link che sono stati attivati e l'ordine cronologico con cui sono stati percorsi, e dispone di comandi che permettono di ripercorrerli all'indietro. In questo caso sullo schermo appaiono solo le informazioni relative al nodo correntemente in esame, ma è possibile in maniera semplice, e senza che l'utente debba ricordare alcunché, far riapparire i nodi precedenti, fino a tornare a quello di partenza.

---

[26] Non può quindi essere usato, ad esempio, un criterio basato sul numero progressivo di ogni nodo all'interno del sistema.

[27] Questa operazione, relativamente semplice quando tutte le informazioni sono concentrate in un unico supporto, può diventare molto complessa, se non addirittura impossibile, quando le informazioni sono disseminate in luoghi diversi, come si vedrà più avanti.

6.5. IL PROBLEMA DELLA RICERCA

Accade molto spesso che l'utente che deve consultare una base di dati non abbia una conoscenza esatta di ciò che sta cercando. Nel caso del codice, è abbastanza improbabile che l'utente ricordi con esattezza il numero dell'articolo che tratta il problema di cui si sta occupando.
Per questo è indispensabile dotare i sistemi ipertestuali di efficaci meccanismi per la ricerca delle informazioni, che può essere effettuata in due modi diversi: o mediante un indice, strutturato esattamente come quello di un libro, che permetta di accedere alle informazioni desiderate, o cercando gli argomenti che interessano direttamente all'interno del sistema. Il primo metodo può ovviamente sfruttare il meccanismo di collegamento già visto: ogni riga dell'indice è di per sé un link che punta al nodo che contiene il testo che interessa. Il secondo metodo permette invece di specificare una o più *parole chiave* le cui occorrenze vengono listate in modo da costituire una specie di indice dei soli nodi che contengono appunto la (o le) parole chiave. Nel seguito, vedremo in pratica come funzionano entrambi i metodi.

6.6. VERSO IL CONCETTO DI MULTIMEDIALITÀ

Finora abbiamo supposto che l'informazione contenuta in ogni nodo fosse esclusivamente testuale. Sappiamo però che i calcolatori di oggi possono rappresentare, oltre all'informazione testuale, anche quella grafica, sia in forma statica (disegni, fotografie, ecc.), sia in forma dinamica (disegni animati, filmati, ecc.). Inoltre, dotandoli di opportune periferiche, è anche possibile fare in modo che essi riproducano suoni preregistrati o sintetizzati.
Nel concetto di ipertesto si tende a prescindere dal tipo di informazione contenuta in ogni nodo: quando viene raggiunta l'informazione desiderata, il sistema determina automaticamente il tipo di dati che la costituiscono, e attiva, sempre in maniera automatica, la procedura più opportuna per rappresentarla.
L'idea di multimedialità prevede appunto che l'informazione contenuta in ogni nodo possa essere presentata all'utente mediante il canale più opportuno. Nei sistemi più evoluti, infatti, si accoppiano al calcolatore altri sistemi (televisori, amplificatori ad alta fedeltà, strumenti musicali elettronici, ecc.) per ottenere la migliore qualità possibile nella rappresentazione di ogni informazione.

6.7. DISTRIBUZIONE DELL'INFORMAZIONE

Un altro punto molto importante del concetto di ipertesto è che, al contrario di quanto avviene nelle basi di dati tradizionali, non è affatto necessario che tutta l'informazione a cui il sistema può accedere debba risiedere sullo stesso supporto. Ciò è opportuno per due ragioni: la prima è che esistono diversi tipi di supporti, con

caratteristiche diverse; la seconda, è che così facendo si può evitare di duplicare inutilmente le informazioni.
Per chiarire il primo di questi due concetti, osserviamo ad esempio la differenza fra un disco magnetico e un videodisco: il primo contiene una quantità relativamente limitata di informazioni (centinaia di Mbyte), con un tempo di accesso molto ridotto (pochi millisecondi), mentre il secondo contiene una grande quantità di informazioni (svariati Gbyte), ma con un tempo di accesso dell'ordine dei decimi di secondo. Se allora dobbiamo costruire un sistema contenente sia testo che filmati, sarà conveniente mantenere tutta l'informazione testuale e quella relativa ai link fra i nodi sul supporto magnetico, utilizzando il videodisco solo per ciò che non può essere contenuto sul disco magnetico, e cioè per le immagini, che come è noto comportano una grande quantità di informazione.
Relativamente alle duplicazioni, osserviamo che in molti casi l'informazione a cui si fa riferimento da un nodo è contenuta in una base di dati completamente diversa: nel nostro esempio, il luogo più ovvio dove andare a cercare la definizione di "abigeato" non è il codice civile, ma il vocabolario della lingua italiana. Potremmo certamente trasportare tutte le definizioni delle parole usate nel codice dal vocabolario al codice, ma ciò comporterebbe un'inutile duplicazione, e quindi uno spreco di memoria, oltre ad evidenti complicazioni nella "manutenzione" delle informazioni stesse. È quindi molto più logico stabilire link che collegano diverse basi di dati, piuttosto che trasportare masse di informazioni dall'una all'altra.
Infatti, accade molto spesso che le informazioni cambino rapidamente: si pensi per esempio ad un sistema ipertestuale che contenga informazioni turistiche sulle principali città italiane, includendo anche le previsioni del tempo ricavate da una base di dati dell'Aeronautica Militare. Dal momento che queste ultime cambiano ogni sei ore, sarebbe necessario effettuare con grande frequenza il trasferimento di una massa non indifferente di dati, che con ogni probabilità verrebbero poi utilizzati solo in minima parte. È molto più logico il meccanismo che permette di prelevare i dati direttamente dalla fonte, solo se e quando servono effettivamente.
Questo concetto sta assumendo una enorme importanza con il diffondersi delle reti di calcolatori, sia su scala locale che mondiale. L'esempio più eclatante è WWW (*World Wide Web*), di cui parleremo più avanti.

6.8. ESEMPI DI IPERTESTI

A puro titolo esemplificativo, saranno mostrati in questo paragrafo alcuni esempi di ciò che può essere fatto utilizzando sistemi ipertestuali.
Ovviamente, non è possibile ricondurre in un libro stampato la potenza espressiva di un ipertesto multimediale: si consiglia perciò il lettore di sperimentare personalmente, utilizzando un calcolatore, le infinite variazioni possibili sul tema.
I campi di possibile applicazione dei sistemi ipertestuali sono pressoché illimitati. In questo capitolo faremo riferimento solo ad alcuni di essi, ma occorre che il lettore tenga presente che le possibili variazioni sul tema sono limitate praticamente solo dalla fantasia umana.

6.8.1. IPERTESTI PER LA DIDATTICA

La didattica è uno dei campi in cui gli ipertesti trovano le migliori possibilità di applicazione. Questo deriva dal fatto che l'insegnamento è sempre costituito dalla trasmissione di una serie di informazioni concatenate le une alle altre in maniera non necessariamente sequenziale. Dovendo sostituire o integrare un docente umano con un mezzo meccanico, la filosofia ipertestuale rappresenta quanto di più efficiente sia stato fino ad oggi inventato.
Il primo esempio riportato riguarda un sistema per lo studio delle particelle subatomiche e delle loro interazioni, di cui vediamo 4 schermate successive. Come si vede in figura 6.7, l'interazione con l'utente avviene utilizzando i pulsanti di cui abbiamo già parlato: l'uomo non deve mai ricordare o scrivere il nome di un nodo già visitato, perché la navigazione fra i diversi nodi avviene in maniera del tutto automatica.
Dalle figure 6.8, 6.9 e 6.10 si può vedere come le informazioni testuali siano mescolate con quelle grafiche per migliorare la chiarezza di uso del sistema.
La figura 6.9 ad esempio contiene solo grafica, ma ognuno degli elementi disegnati è in realtà un link ad un nodo che lo descrive in dettaglio.
Sempre dal campo della didattica osserviamo un secondo esempio, riguardante un atlante anatomico della mano (Figg. 6.11 e 6.12). Anche qui l'informazione grafica è predominante, per non dire esclusiva: premendo il pulsante del mouse su una qualunque parte del disegno appare il nome della parte indicata, a cui possono essere aggiunte ulteriori informazioni.

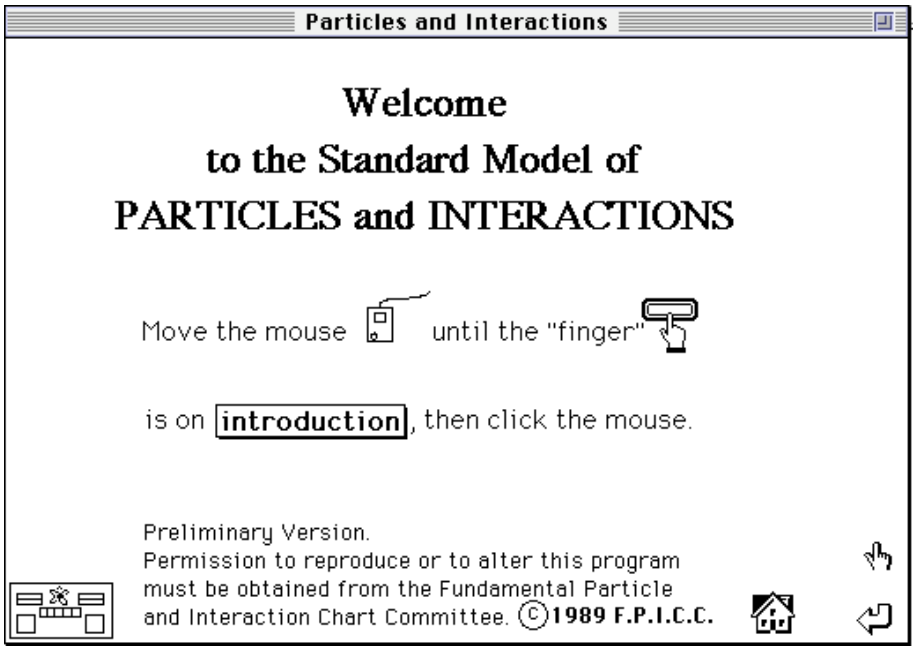


Figura 6.7. - Sistema didattico per lo studio delle particelle subatomiche (1).

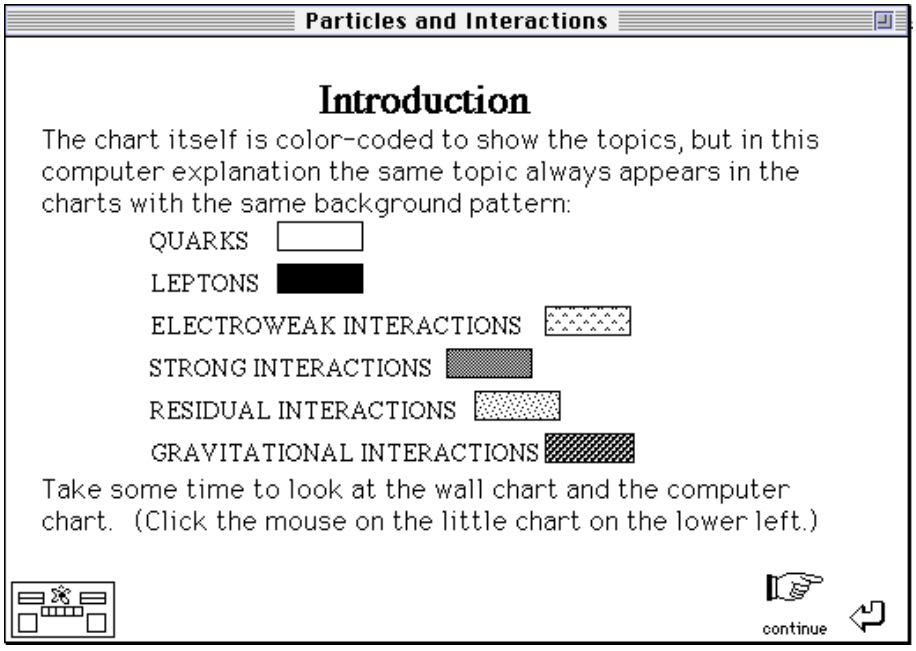


Figura 6.8. - Sistema didattico per lo studio delle particelle subatomiche (2).

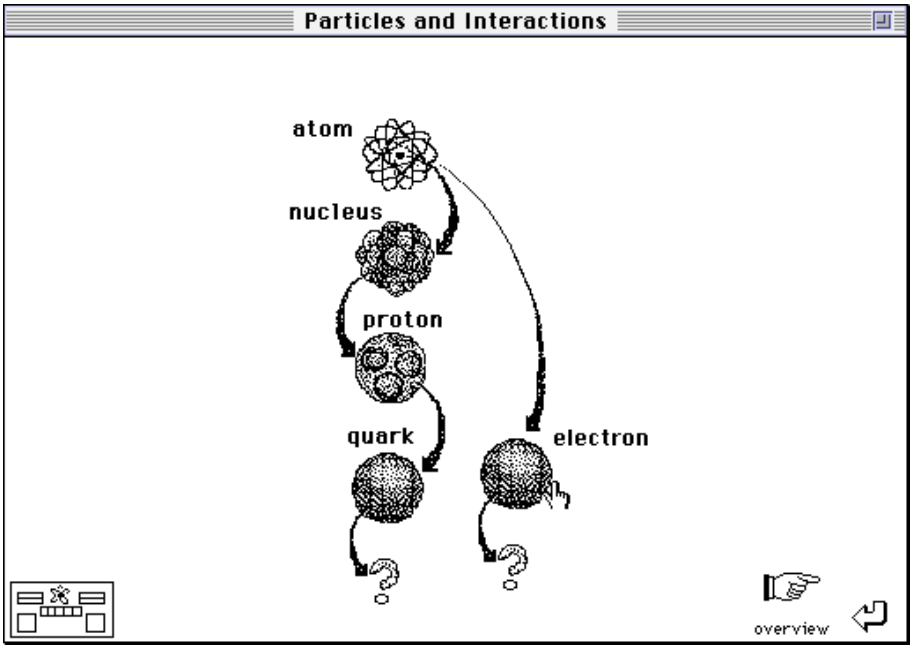


Figura 6.9. - Sistema didattico per lo studio delle particelle subatomiche (3).

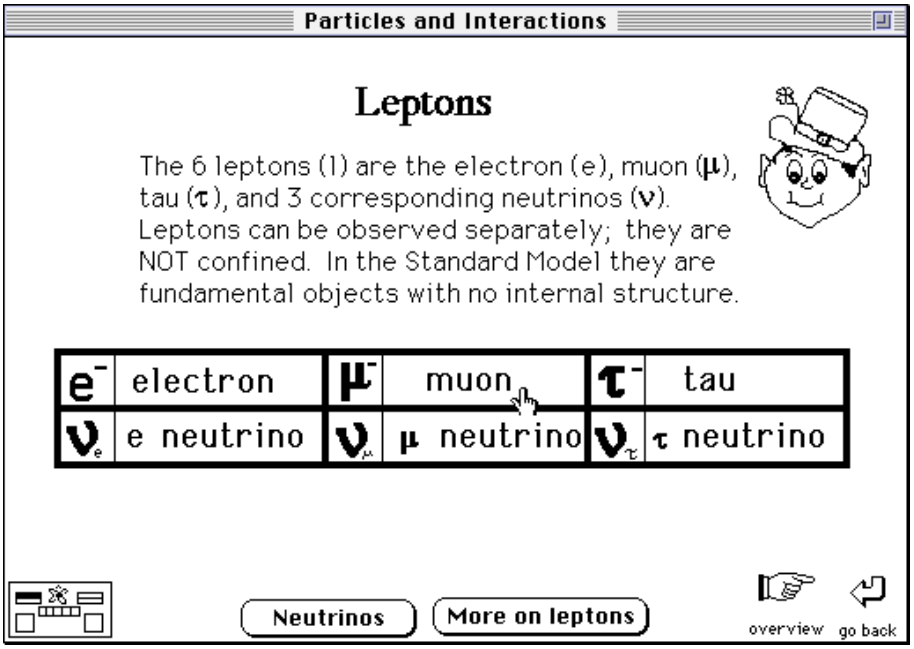


Figura 6.10. - Sistema didattico per lo studio delle particelle subatomiche (4).

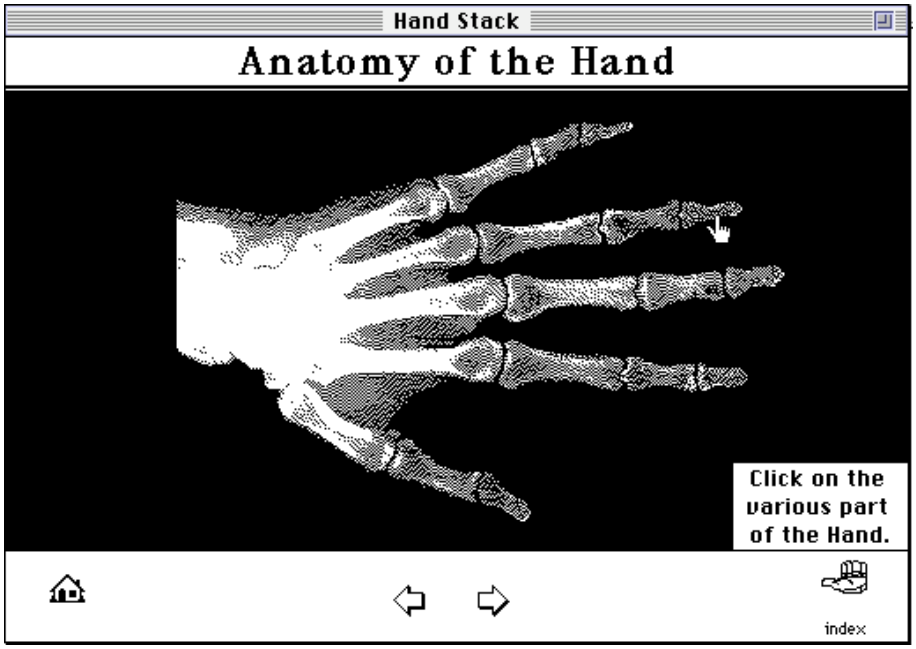


Figura 6.11. - Un atlante anatomico in HyperCard (1).

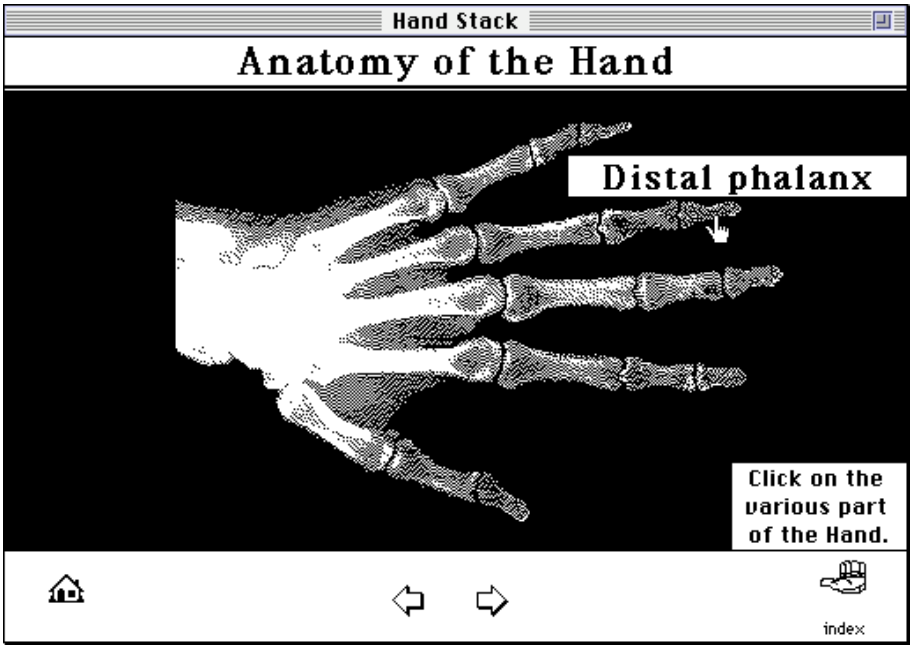


Figura 6.12. - Un atlante anatomico in HyperCard (2).

La possibilità di associare ai link informazioni diverse si rivela molto utile nel campo della didattica. Ad esempio, per lo studio della cardiologia è possibile realizzare un atlante analogo a quello appena visto, a cui però sono associati i suoni risultanti dall'auscultazione del torace in situazione normale e in varie patologie. Ovviamente, tutto ciò non è in alcun modo realizzabile con i soli libri stampati.

6.8.2. MANUALI ON-LINE

Un altro fondamentale campo di applicazione dei sistemi ipertestuali è quello della manualistica. Un manuale, qualunque sia il sistema di cui tratta, è sempre pieno di riferimenti incrociati. Inoltre, se il sistema descritto è complesso, la sua consultazione, se è stampato in forma tradizionale, è resa problematica dalla difficoltà di reperire rapidamente gli argomenti che interessano il lettore. Le figure 6.13 - 6.28 rappresentano diverse schermate del manuale di HyperCard.

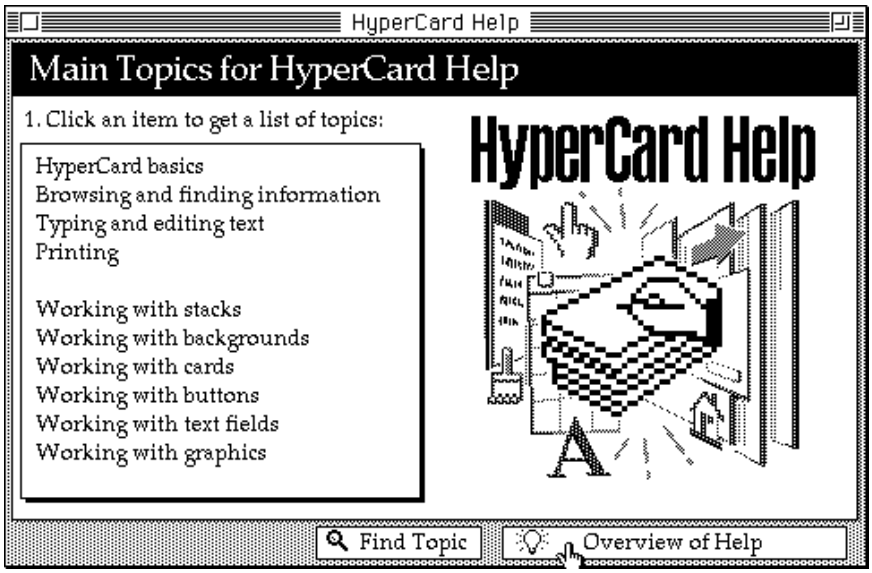


Figura 6.13. - Un manuale *on line* (1).

Osserviamo in figura 6.13 il primo metodo per il reperimento delle informazioni: nella schermata di apertura (quella cioè che appare quando si comincia ad usare il manuale) viene presentato un indice degli argomenti disponibili. Ogni riga dell'indice rappresenta un link alla pagina corrispondente. Sono inoltre disponibili alcuni pulsanti che permettono di ottenere altri tipi di informazione. Ad esempio, facendo click sul pulsante *Overview of Help* si ottiene la finestra di figura 6.14, da cui è possibile spostarsi mediante altri pulsanti (Figg. 6.15 - 6.19).

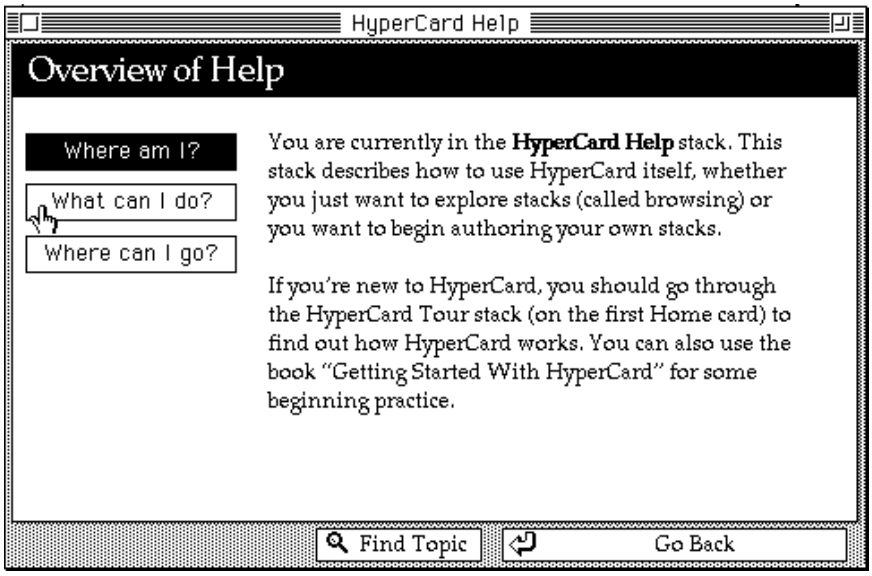


Figura 6.14. - Un manuale *on line* (2).

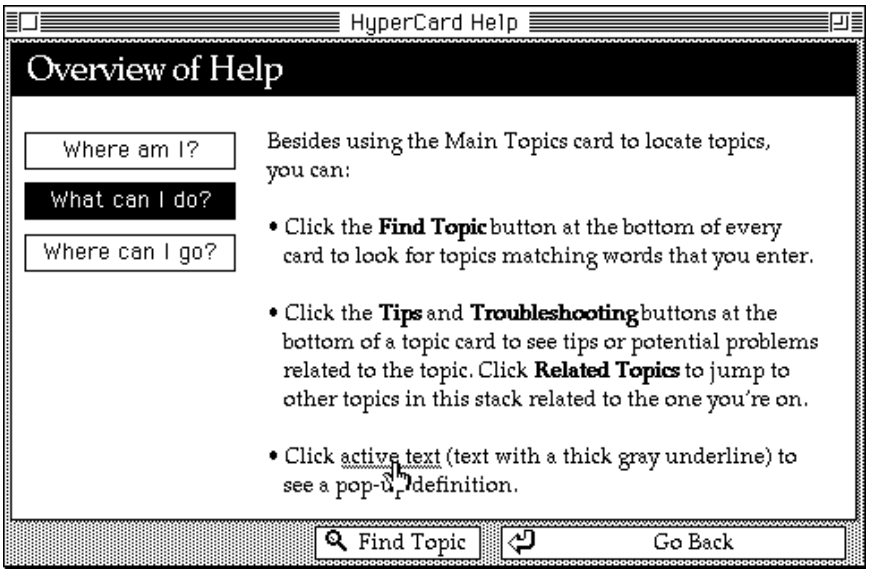


Figura 6.15. - Un manuale *on line* (3).

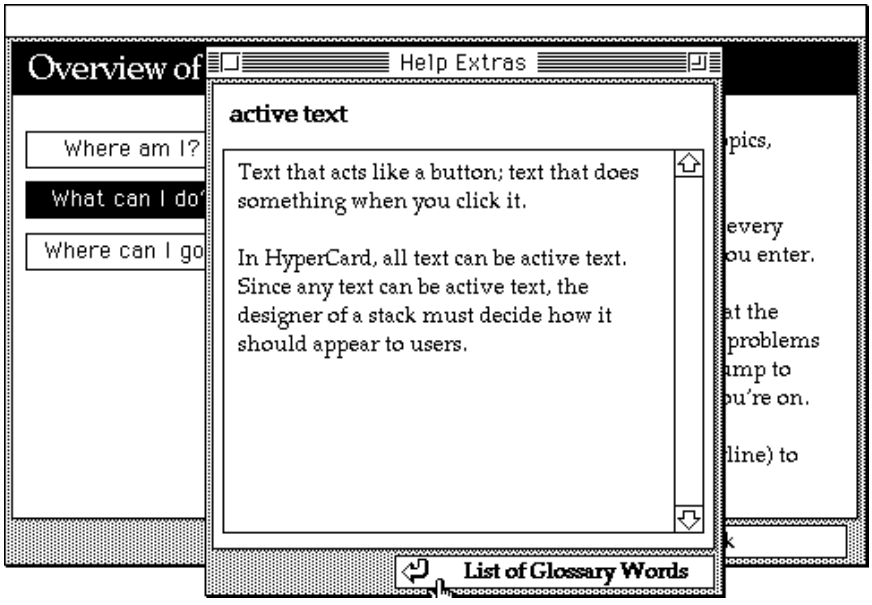


Figura 6.16. - Un manuale *on line* (4).

Osserviamo in figura 6.15 che è disponibile la funzione di glossario descritta precedentemente: facendo click sulle parole sottolineate si attiva una finestra che contiene la loro definizione. Osserviamo che anche il glossario è a sua volta organizzato come un ipertesto, in cui è possibile spostarsi con le modalità già viste (Figg. 6.16 - 6.18).

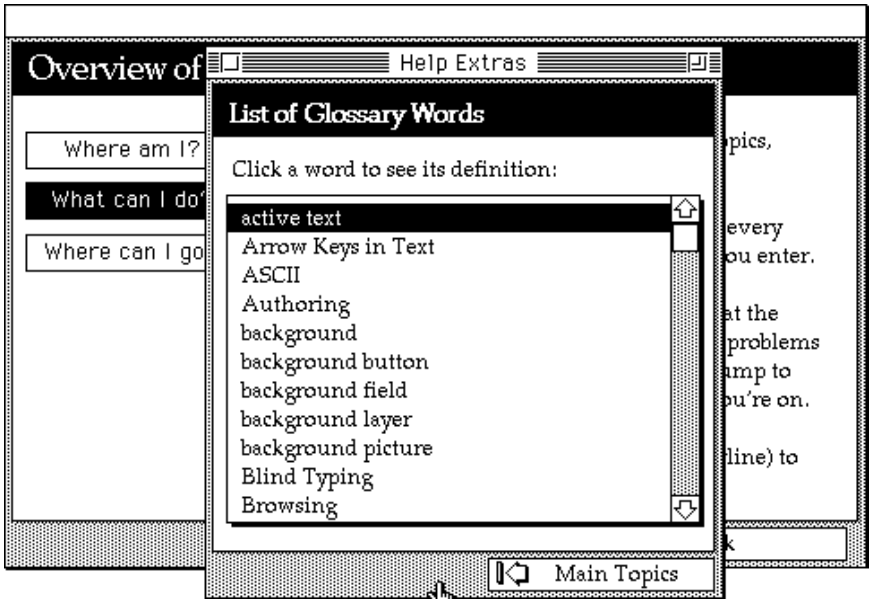


Figura 6.17. - Un manuale *on line* (5).

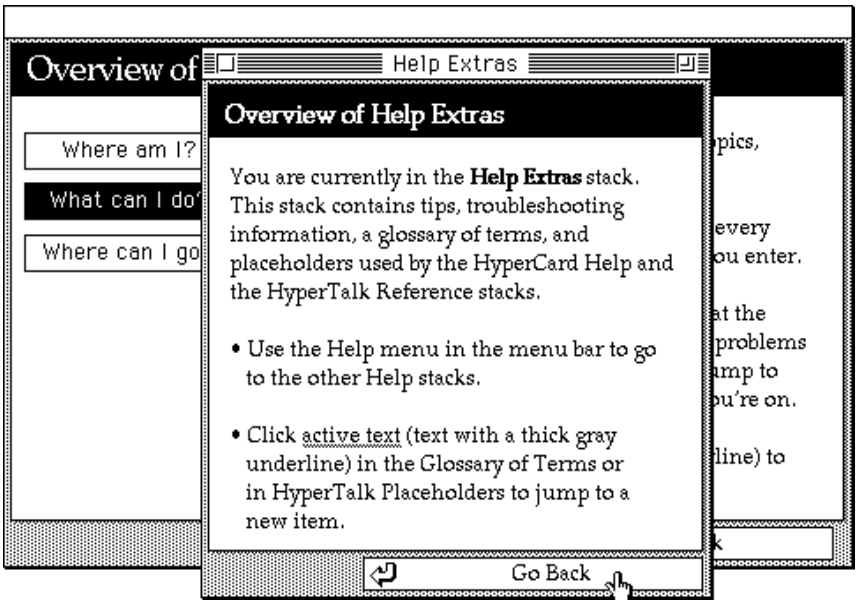


Figura 6.18. - Un manuale *on line* (6).

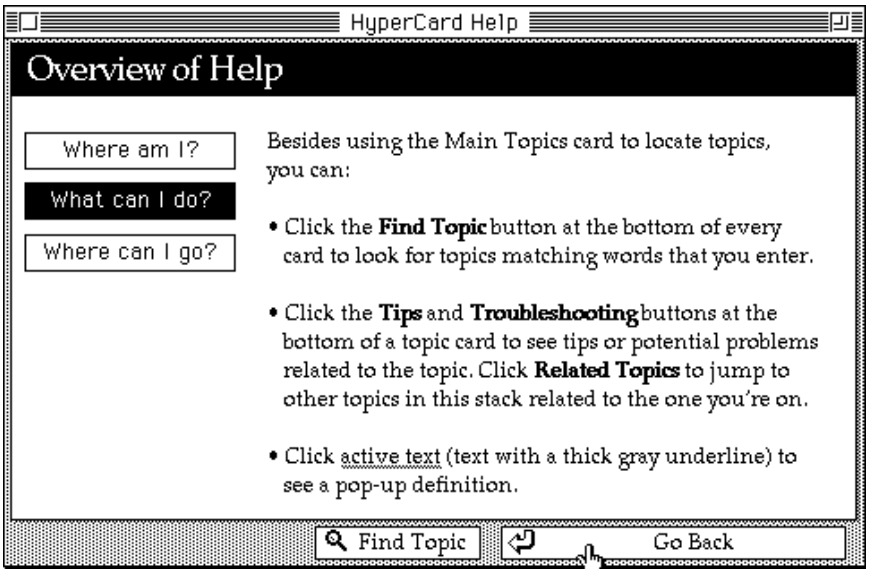


Figura 6.19. - Un manuale *on line* (7).

Tornando alla schermata di apertura (Fig. 6.20) osserviamo che, facendo click su una riga dell'indice, appare in un secondo riquadro una lista più dettagliata degli argomenti contenuti in quella voce (Fig. 6.21), da cui è possibile poi passare alle pagine che contengono la descrizione effettiva (Fig. 6.22).

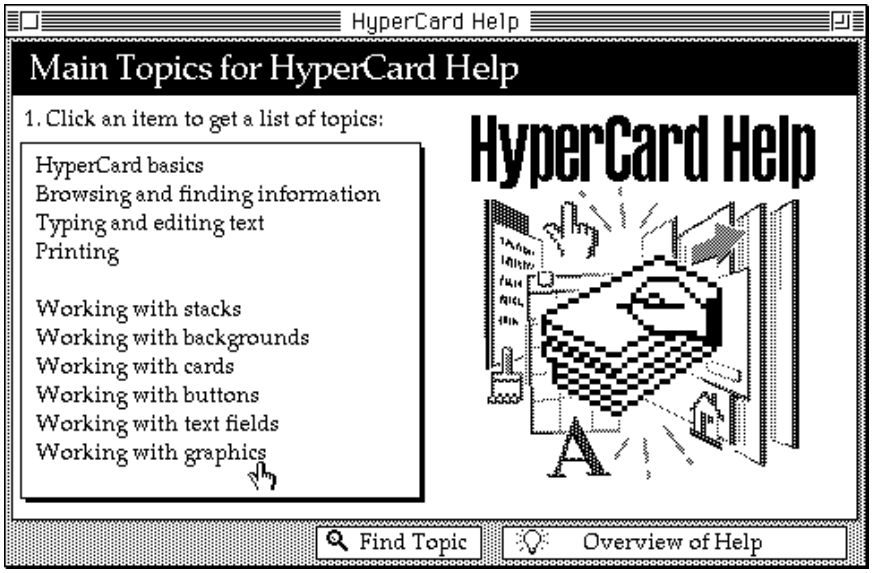


Figura 6.20. - Un manuale *on line* (8).

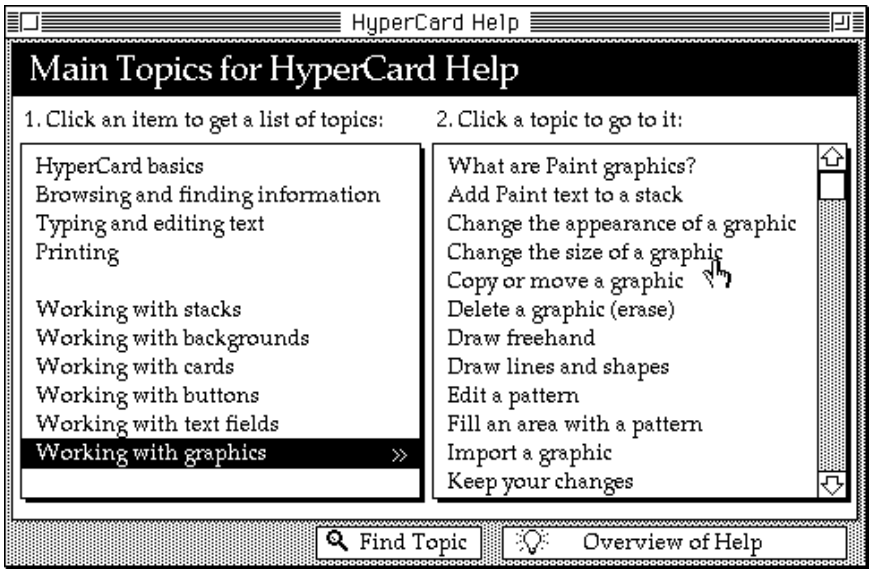


Figura 6.21. - Un manuale *on line* (9).

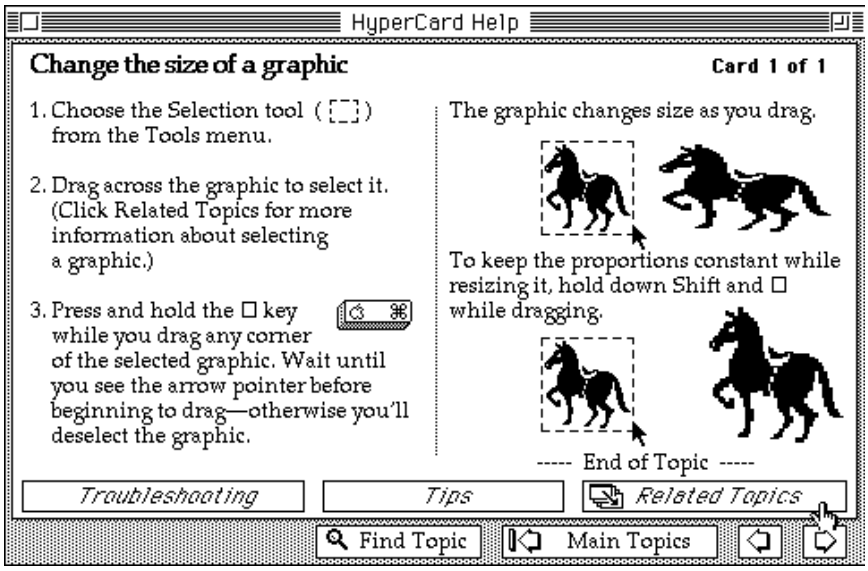


Figura 6.22. - Un manuale *on line* (10).

Per ogni argomento è disponibile un elenco degli argomenti logicamente collegati a quello che si sta esaminando (Figg. 6.23 - 6.25).
Osserviamo anche (Figg. 6.25 e 6.26) che è sempre possibile ritornare al punto da cui si era partiti, perché il sistema memorizza i nodi visitati. Il ritorno può quindi avvenire in maniera del tutto automatica.

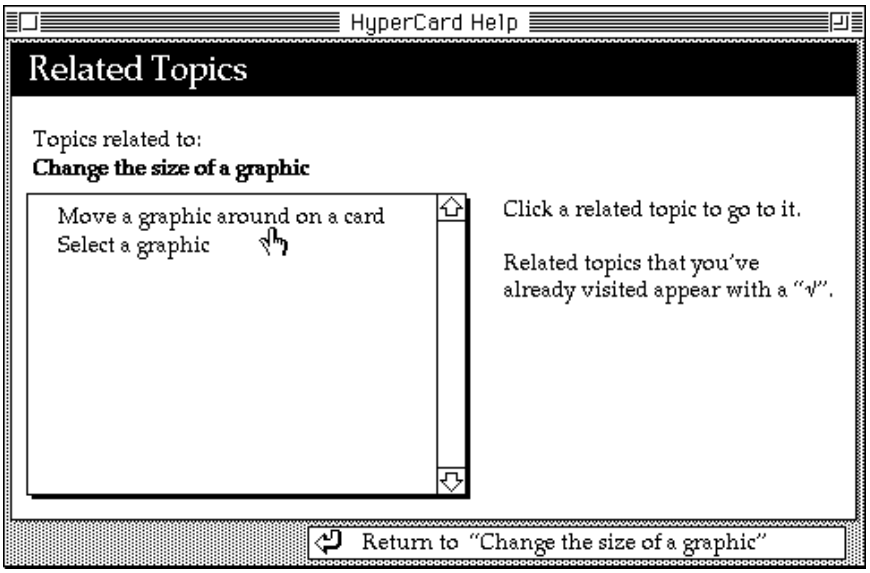


Figura 6.23. - Un manuale *on line* (11).

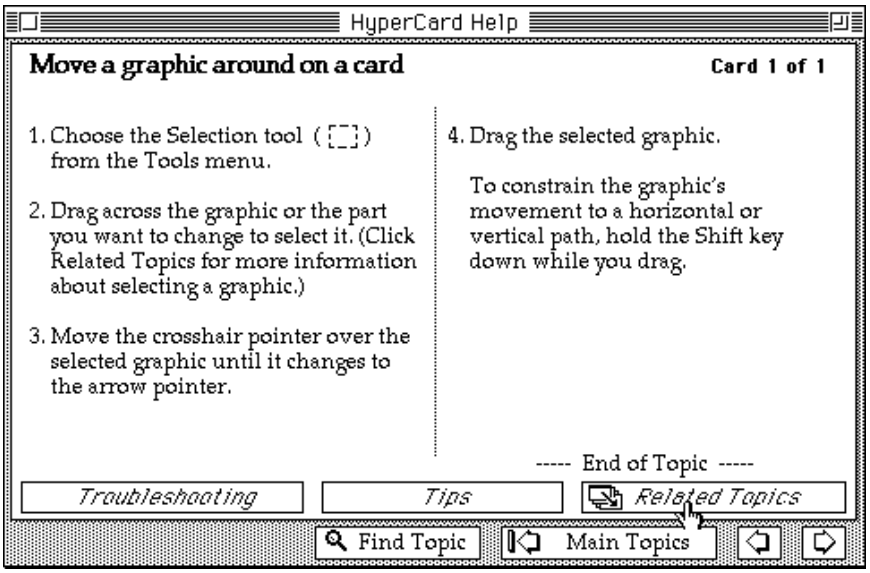


Figura 6.24. - Un manuale *on line* (12).

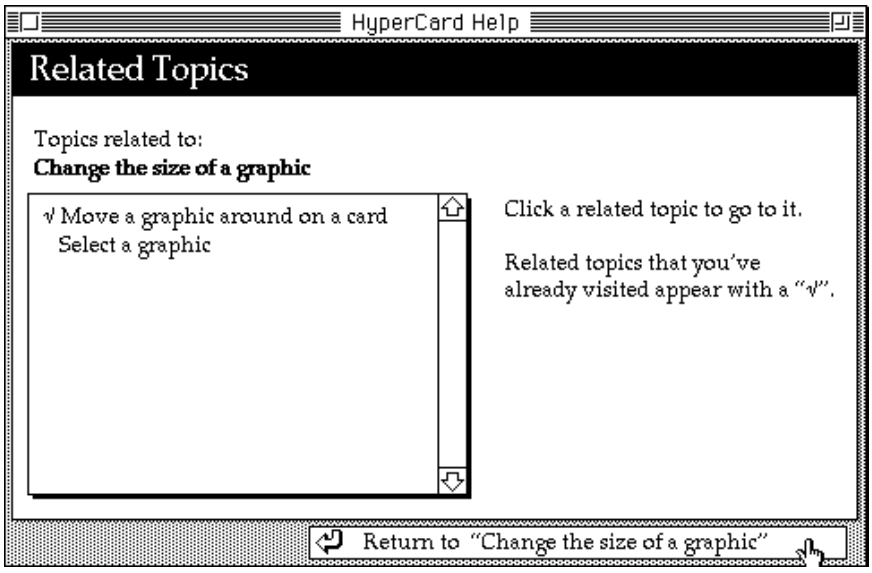


Figura 6.25. - Un manuale *on line* (13).

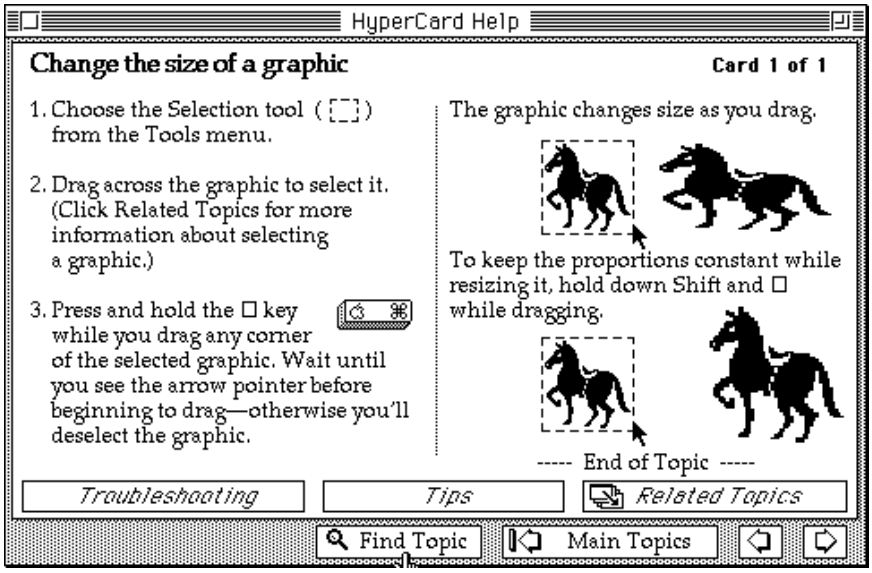


Figura 6.26. - Un manuale *on line* (14).

Per finire, diamo uno sguardo alla seconda modalità di reperimento delle informazioni. In questo ipertesto è possibile attivare una ricerca su una parola chiave. Per esempio, definendo come parola chiave la parola "window" (Fig. 6.27) si ottiene la lista mostrata in figura 6.28, che contiene tutti i nodi in cui appare la parola cercata. Il passaggio a questi nodi avviene con le stesse modalità viste per la navigazione normale.

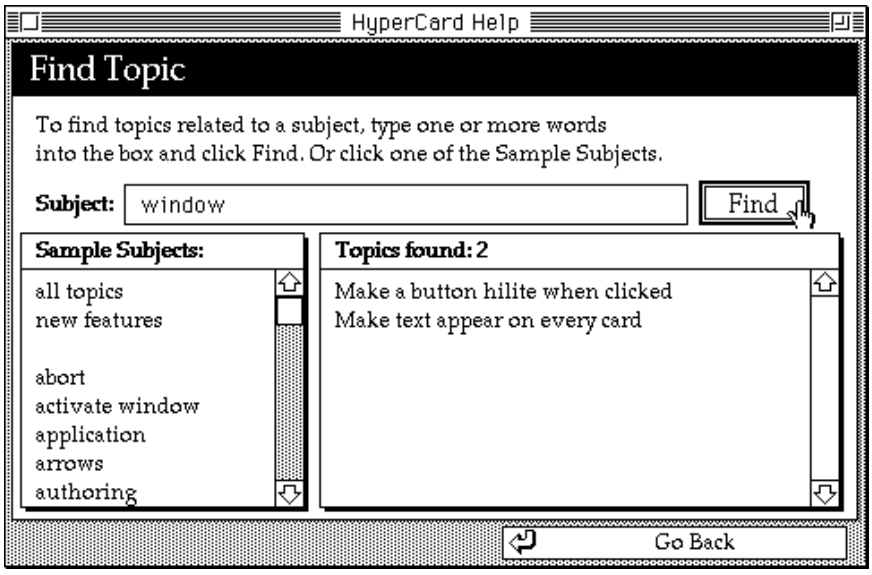


Figura 6.27. - Un manuale *on line* (15).

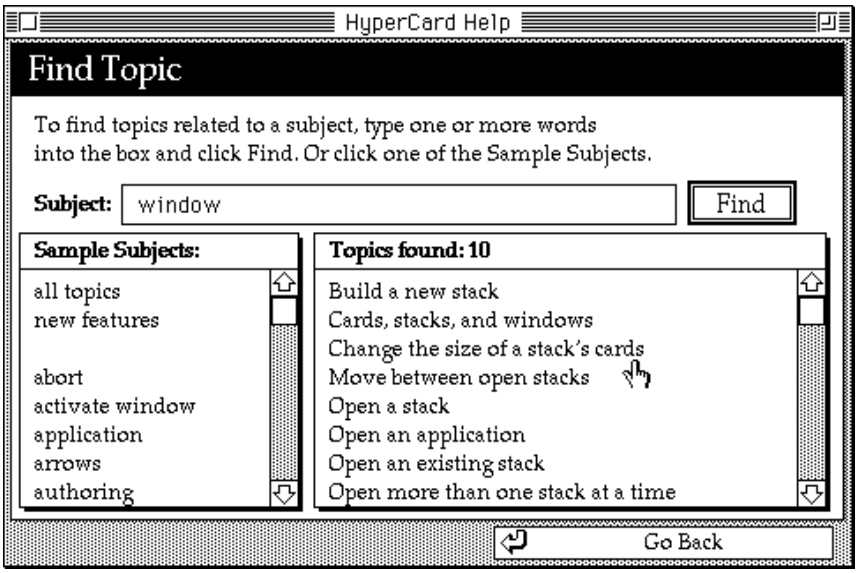


Figura 6.28. - Un manuale *on line* (16).

6.8.3. IPERTESTI SU SCALA MONDIALE

Per terminare questo capitolo accenniamo a WWW (*World Wide Web*), il sistema ipertestuale più esteso attualmente esistente. I principi su cui si basa sono quelli già esposti, con la differenza che i nodi si trovano su calcolatori diversi, situati in ogni parte del mondo. Ovviamente, perché il sistema possa funzionare, è necessario che i nodi siano collegati tramite una rete che consenta il trasferimento delle informazioni

in tempo reale. WWW utilizza Internet che, con diversi milioni di calcolatori collegati, è la rete informatica più estesa attualmente esistente.
WWW permette il trasferimento di informazione testuale, binaria generica (ad esempio per trasportare programmi), grafica (ferma e in movimento) e sonora, definendo protocolli diversi per ogni tipo. I nodi sono contenuti in macchine (*server*) collegate alla rete, e disponibili a fornire queste informazioni a seguito di richieste provenienti dalla rete stessa. Sono inoltre disponibili diversi servizi ausiliari, come la posta elettronica.
Vediamo il funzionamento di WWW osservando il link relativo ad una associazione scientifica, che vuole pubblicizzare le proprie iniziative (Fig. 6.29). Essa ha definito un nodo principale (*Home page*), che contiene diversi link che puntano alla descrizione di varie iniziative. Seguendo uno di essi, vediamo l'elenco delle prossime conferenze (Fig. 6.30), da cui possiamo passare all'annuncio di una conferenza che si svolgerà nel 1996 a Kaiserslautern (Germania), mostrato in figura 6.31. Ancora, possiamo ottenere informazioni sul luogo dell'evento (Fig. 6.32). È interessante osservare che, in questo caso, la home page si trova su un calcolatore sito in Belgio, mentre l'annuncio della conferenza, il cui organizzatore è italiano, è in un calcolatore dell'università di Brescia. Le informazioni sulla città di Kaiserslautern sono ovviamente contenute in calcolatori localizzati nel municipio di quella città. Operando in questo modo, le informazioni sono locali rispetto a chi ha il compito di mantenerle aggiornate, e possono essere recuperate da ogni parte del mondo utilizzando un calcolatore collegato ad Internet, su cui giri un opportuno sistema ipertestuale. Ancora una volta, osserviamo che, dal momento che l'indirizzamento è gestito in maniera automatica, l'utente non ha nessun bisogno di sapere dove risiedono le informazioni che sta osservando: l'unica cosa che egli deve fornire al calcolatore è l'indirizzo della home page, che d'altra parte può a sua volta essere ottenuto automaticamente utilizzando appositi strumenti per la ricerca delle informazioni.

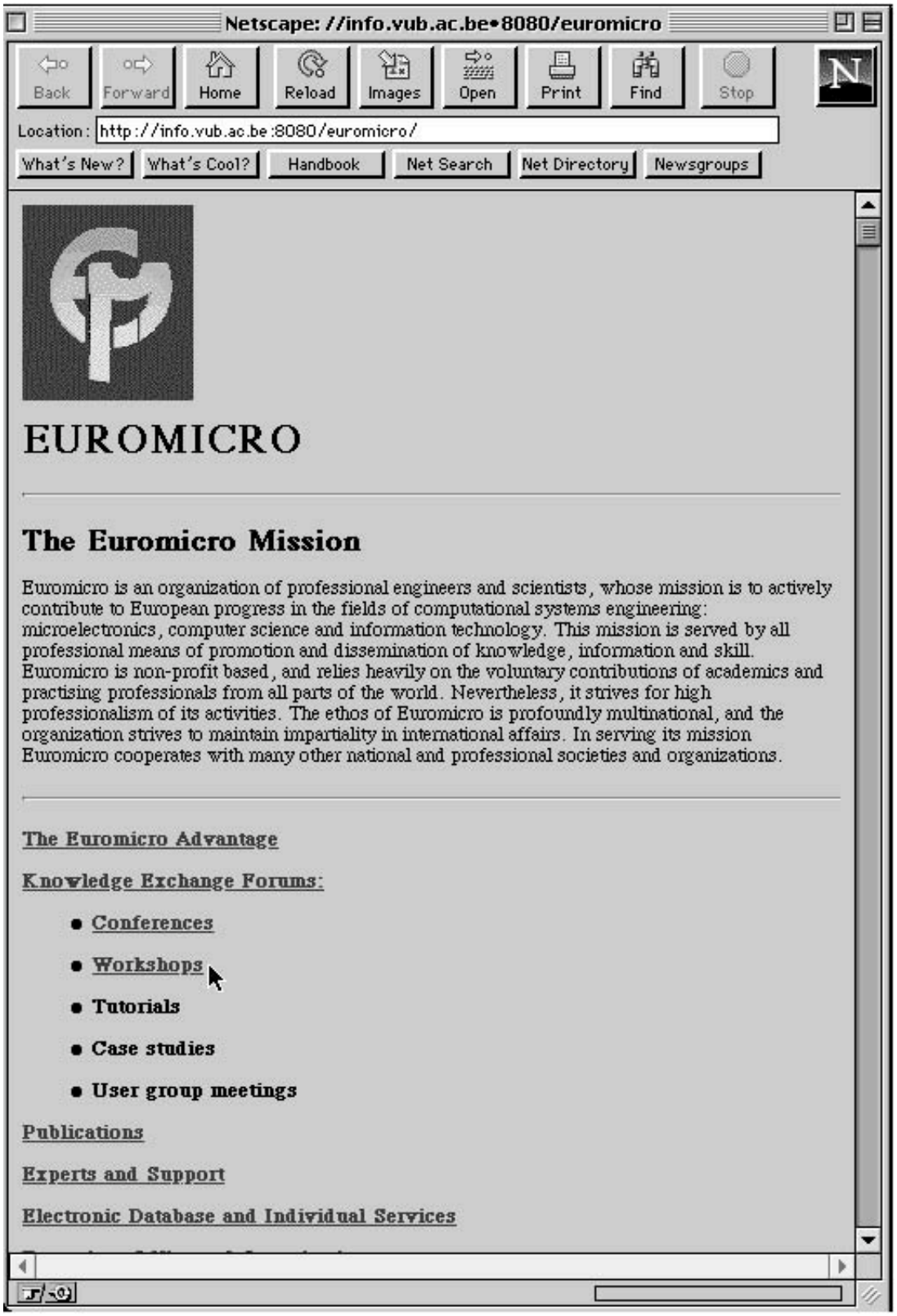


Figura 6.29. - La "Home page" di Euromicro.

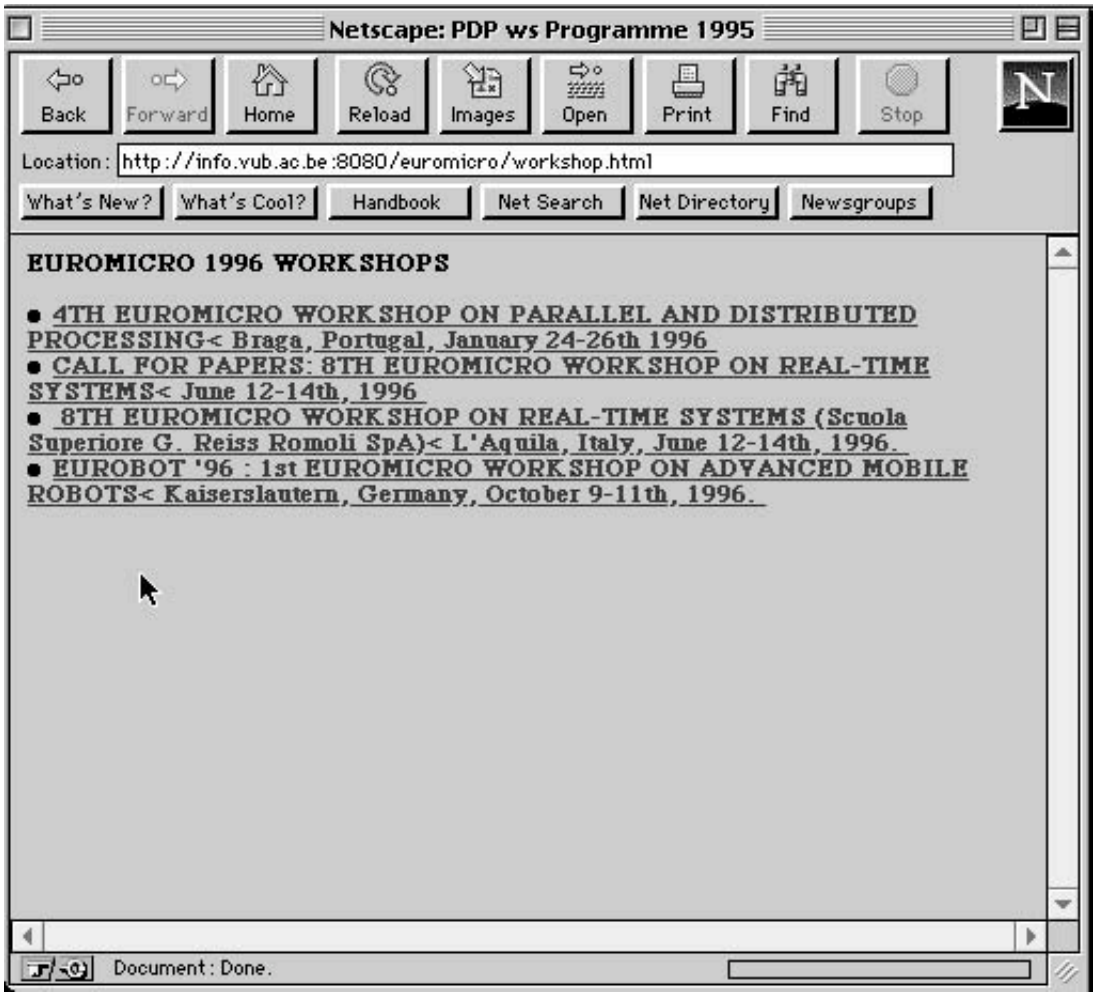


Figura 6.30 - L'elenco delle conferenze.

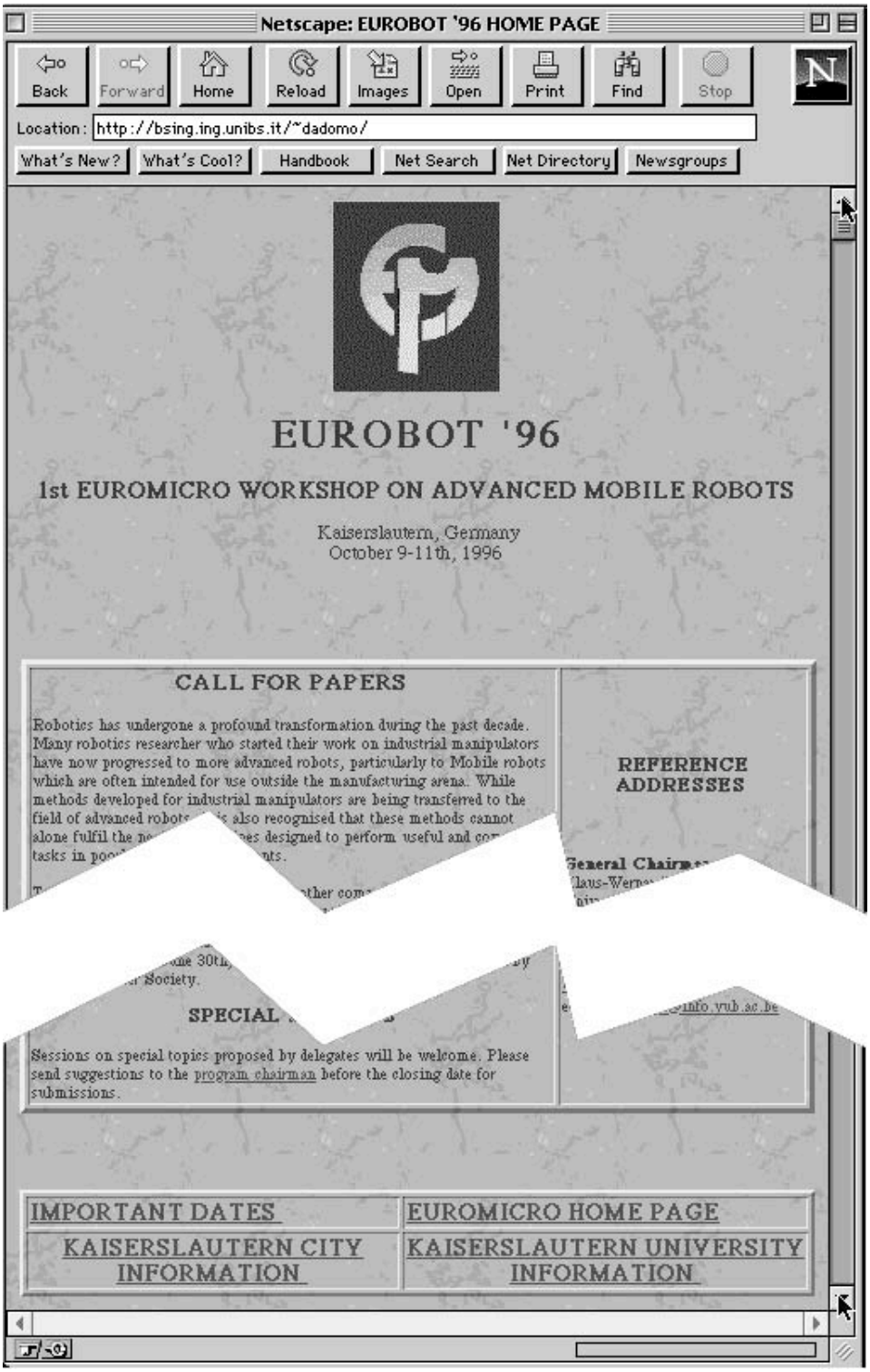


Figura 6.31. - La pagina relativa alla conferenza.

Figura 6.32. - Una pagina relativa alla città di Kaiserslautern.

6.9. PROBLEMI ED ESERCIZI

Problema 6.1.*
Quale fra le affermazioni che seguono si applica meglio agli ipertesti?

a. la possibilità di usare file molto grandi;
b. la possibilità di utilizzare messaggi per far comunicare gli oggetti tra di loro;
c. la possibilità di assegnare ad ogni oggetto particolari reazioni quando esso venga sollecitato opportunamente.

Problema 6.2.*
Il concetto di ipertesto implica soprattutto:

a. grande efficienza;
b. grande facilità di programmazione;
c. grande facilità di uso;
d. ottima interfaccia utente-macchina.

Problema 6.3.
Descrivere in una relazione di non più di due pagine il concetto di multimedialità.

7

UN ESEMPIO DI IPERTESTO: HYPERCARD E HYPERTALK

Questo capitolo è dedicato all'esame di una filosofia di programmazione dei calcolatori che è per molti versi differente da quelle viste in precedenza: esamineremo infatti un sistema, basato sul concetto di ipertesto visto nel capitolo precedente, che utilizza il cosiddetto paradigma *message passing* per semplificarne la programmazione, riducendo drasticamente la complicazione della programmazione non modale senza perderne i vantaggi.
È opportuno osservare che HyperCard, di cui il capitolo tratta, è un sistema completo di programmazione, la cui descrizione richiederebbe, da sola, un intero volume: si daranno qui solo le informazioni essenziali alla comprensione di quanto c'è di nuovo nel paradigma *message passing*, rimandando alla bibliografia per una descrizione completa del sistema.
Inoltre, come avviene in questi casi, per la comprensione completa della materia sarebbe indispensabile avere sottomano un calcolatore su cui sperimentare e approfondire le informazioni via via fornite.

7.1. HYPERCARD

Tra i molti sistemi di programmazione message passing disponibili sul mercato, la scelta di studiare HyperCard è principalmente dovuta al fatto che esso, oltre a combinare in un unico sistema le caratteristiche più interessanti degli ipertesti e dei

sistemi message passing, è stato concepito per funzionare su calcolatori di piccole dimensioni ed è quindi accessibile a tutti gli utenti. HyperCard gira sui calcolatori della famiglia Macintosh: esistono prodotti simili o, in alcuni casi, praticamente uguali,[28] adatti a funzionare su macchine del tipo PC IBM.
HyperCard è stato concepito in origine come un database configurabile dall'utente secondo le sue specifiche esigenze. Da questo punto di vista, esso non si differenzia da altri database di impiego molto più diffuso (ad esempio DBIII). Anzi, la sua efficienza nel caso di basi di dati di grandi dimensioni (maggiori di un paio di Mbyte) è piuttosto limitata.
La differenza con un database di tipo tradizionale comincia a diventare evidente se si osserva che gli aspetti grafici sono privilegiati in HyperCard, sia dal punto di vista del contenuto delle basi di dati (si possono costruire con facilità basi di dati grafiche, che contengono cioè immagini oltre ad informazioni testuali, come vedremo più avanti), sia dal punto di vista delle operazioni che possono essere compiute dall'utente. Infatti, tutti i comandi per l'inserzione, la cancellazione e la ricerca di informazioni possono essere dati utilizzando gli strumenti grafici già visti a proposito della programmazione non modale: menu, pulsanti, dialoghi, *alert*, ecc.
Per esemplificare quanto abbiamo appena affermato, osserviamo la figura 7.1, che rappresenta lo schermo del calcolatore in due diversi momenti dell'uso di HyperCard con un database che contiene ricette di cucina.
Nella parte alta della figura è mostrata la schermata iniziale: essa è costituita da una serie di simboli grafici la cui interpretazione è immediata. Spostando il puntatore (rappresentato dalla piccola mano con l'indice teso) sul simbolo che interessa e premendo il pulsante del mouse,[29] si ottiene l'effetto desiderato. Ad esempio, facendo click sulla piccola freccia in basso a destra si ottiene la seconda schermata della figura, che contiene a sua volta altri simboli grafici, oltre ad informazioni testuali.

---

[28] Ad esempio PLUS.

[29] In gergo, questa operazione è detta "fare click" (*to click on*) su un oggetto, in questo caso un pulsante.

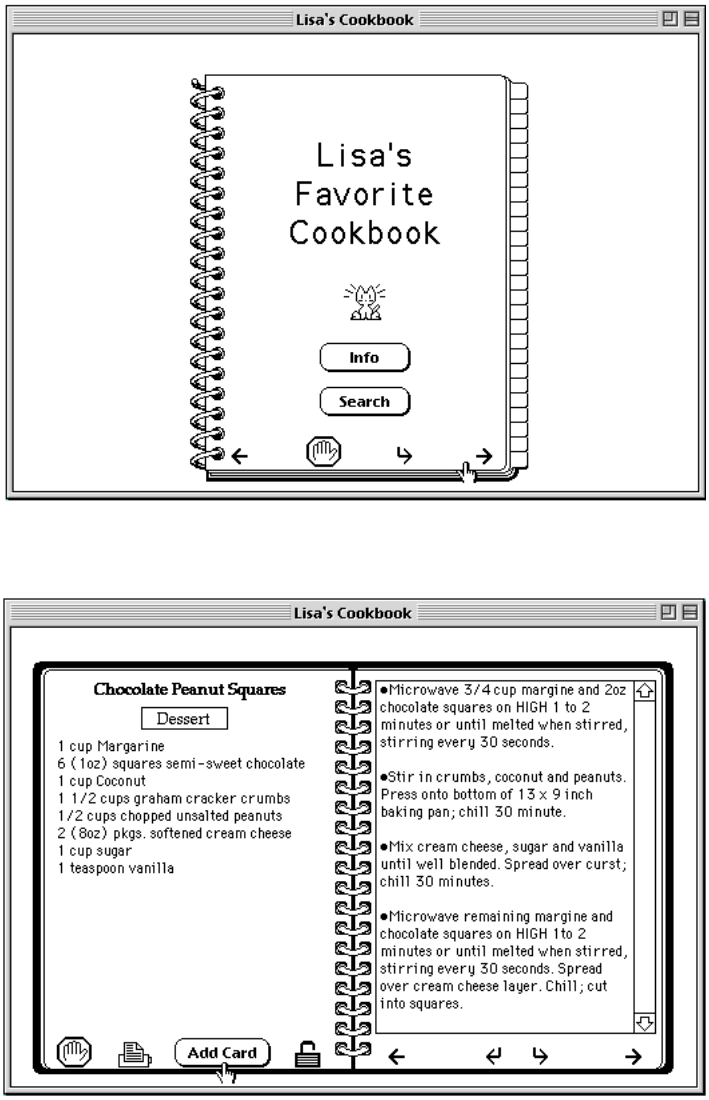


Figura 7.1. - Due schermate di HyperCard.

7.1.1. HYPERCARD COME DATABASE

Ci occuperemo innanzitutto dell'uso di HyperCard come un normale database. HyperCard opera su file che, nel gergo particolare del linguaggio, vengono chiamati *stack*. Ogni stack contiene al suo interno un certo numero di oggetti, che saranno descritti in dettaglio nel paragrafo seguente.

7.1.2. LE ENTITÀ FONDAMENTALI DI HYPERCARD

Le entità fondamentali (oggetti) di HyperCard sono:

- stack;

- cornici;
- schede;
- campi;
- pulsanti;
- grafica.

Come abbiamo già detto, viene chiamato stack un file che contiene un certo numero di oggetti che HyperCard è in grado di utilizzare.
Dal momento che HyperCard è in primo luogo un database, vengono messe a disposizione dell'utente le cosiddette *cornici*, ognuna delle quali rappresenta la definizione del formato di un insieme di schede. Per chiarire il concetto, pensiamo ad una biblioteca che raccolga libri e dischi.

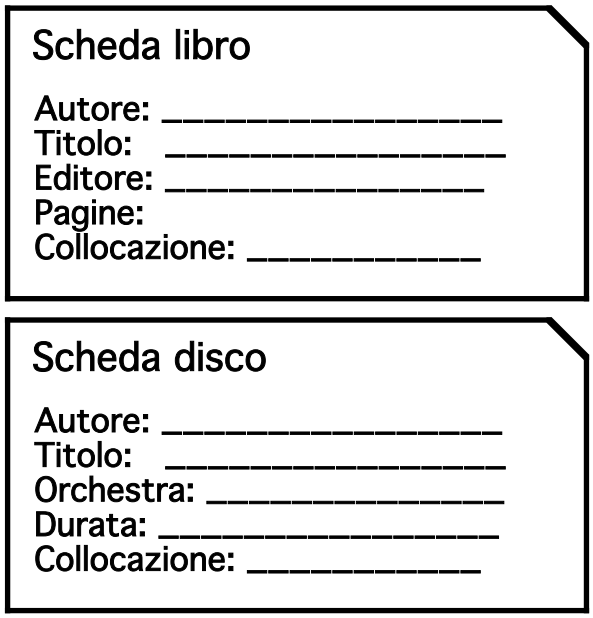


Figura 7.2. - Due formati di schede.

Il suo catalogo (nel nostro caso lo stack) sarà composto da un armadio, contenente un certo numero di schede (supponiamo che ne esista una per ogni libro o disco archiviato). È evidente che, data la diversa natura degli oggetti a cui si riferiscono, le schede per l'archiviazione dei libri saranno differenti da quelle destinate ai dischi, ma tutte le schede per i libri avranno lo stesso formato, e le differenze fra l'una e l'altra saranno solo nel contenuto delle singole voci. Lo stesso discorso può essere fatto per le schede destinate alla catalogazione dei dischi. Un esempio molto semplificato di quanto appena detto è mostrato in figura 7.2.

Nulla poi vieta che le schede, ordinate secondo un certo criterio, siano “mischiate” le une con le altre, come si può vedere in figura 7.3.
Nella terminologia di HyperCard, ogni tipo di scheda è chiamato *cornice* (più propriamente *background*, cioè sfondo, in inglese).
Le parti compilabili delle schede, cioè quelle che contengono informazioni diverse per ogni scheda, sono chiamate *campi* (*fields*).
Esistono poi i pulsanti, che sono del tutto analoghi, sia come aspetto che come funzioni, a quelli già visti nella descrizione della programmazione non modale: ad essi possono essere associate informazioni testuali o grafiche.
Per finire, è possibile definire oggetti grafici generici (disegni, fotografie, ecc.).
Campi, pulsanti e grafica possono essere associati alle cornici (e in questo caso essi compariranno su tutte le schede appartenenti alla cornice in questione), oppure ad una singola scheda (compariranno cioè solo sulla scheda a cui appartengono).
La figura 7.4 riassume i diversi oggetti appena descritti e ne mostra le interdipendenze.
È opportuno osservare che uno stack contiene sempre almeno una cornice, e che ogni cornice contiene sempre almeno una scheda.

7.1.3. COSTRUZIONE DI UNO STACK HYPERCARD

Dal momento che non è un programma applicativo dedicato, HyperCard non può svolgere nessuna funzione utile all’utente se non viene prima programmato. Una caratteristica di HyperCard è che, perché esso possa funzionare, deve per forza esistere uno stack identificato da un nome che non può essere cambiato: *home* (*casa*). In altre parole, quando si fa partire HyperCard, il programma cerca un file del tipo opportuno che si chiama *home* e se non lo trova l’esecuzione si blocca. Il file *home* deve quindi sempre essere aperto durante il funzionamento del sistema.

Figura 7.3. - Un archivio con schede di due formati diversi.

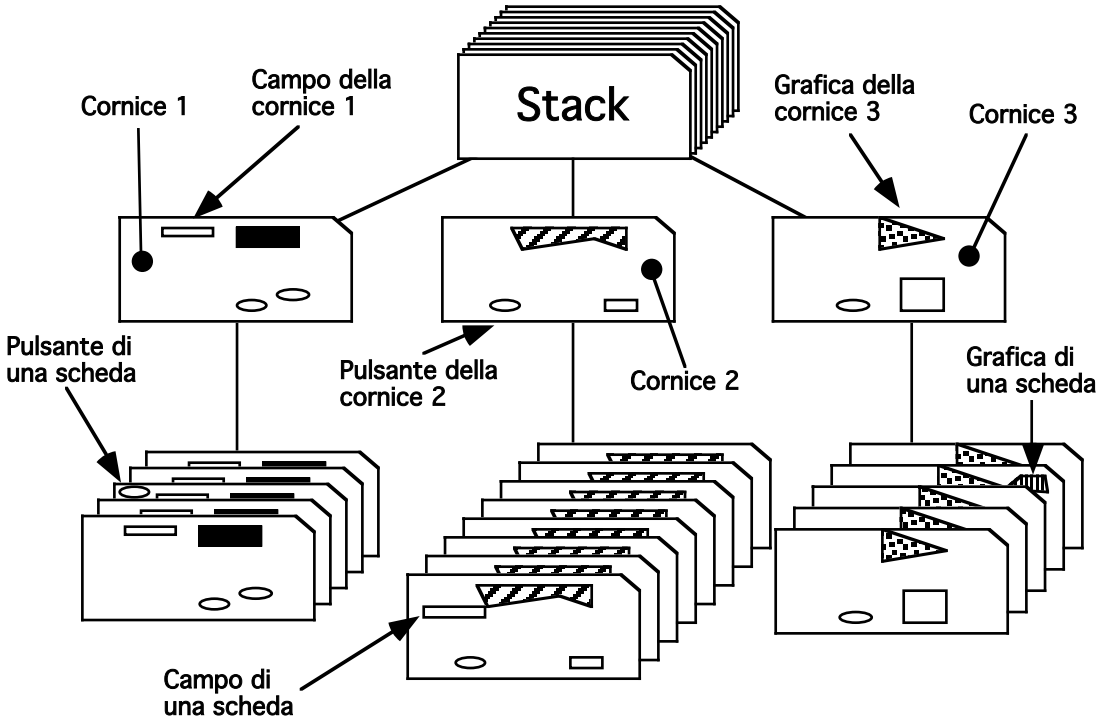


Figura 7.4. - Interdipendenze fra gli oggetti di HyperCard.

Le figure 7.5, 7.6 e 7.7 mostrano le informazioni relative a uno stack appena creato, e quindi ancora vuoto. Come già detto, uno stack vuoto contiene una cornice ed una scheda. Osserviamo che sullo schermo del calcolatore viene sempre presentata una scheda: le cornici e gli stack non possono essere visualizzati direttamente. Le modifiche alle cornici vengono effettuate utilizzando una qualunque scheda appartenente alla cornice che si vuole modificare, che in quel momento funge da "prototipo" della cornice stessa.
Per avere informazioni su schede, cornici e stack si sceglie l'opportuno comando dalla barra dei menu: si tratta della voce *Objects*, sotto la quale risiedono le varie opzioni *Stack info…*, *Background info…* e *Card info…*.

7.1.4. IDENTIFICAZIONE DEGLI OGGETTI IN HYPERCARD

Chiedendo informazioni sull'unica cornice contenuta nello stack vuoto osserviamo che la cornice non ha alcun nome (Fig. 7.6). Questo può apparire strano, in quanto essa deve essere ovviamente identificabile all'interno del programma.

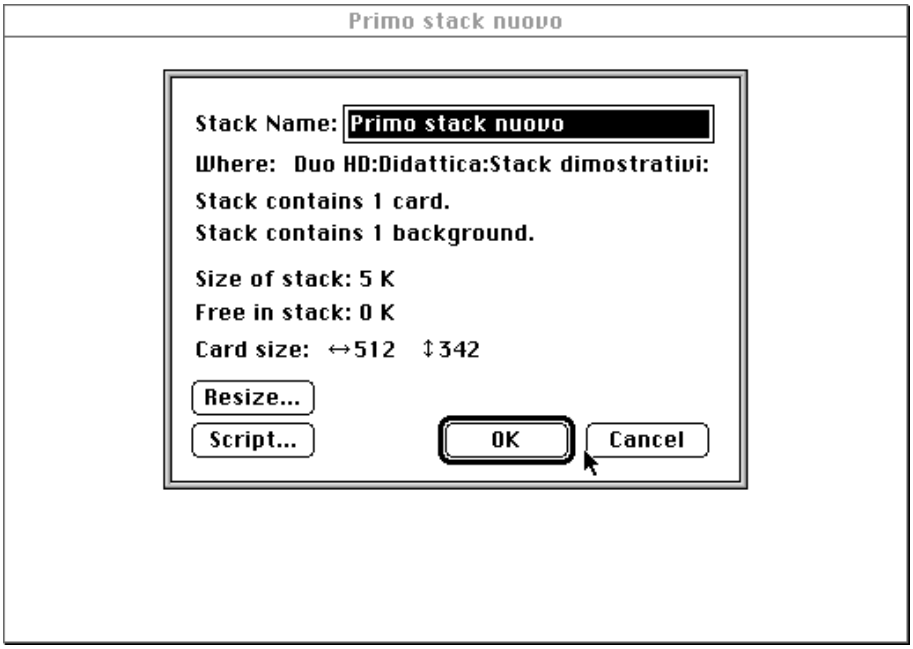


Figura 7.5. - Informazioni relative a uno stack.

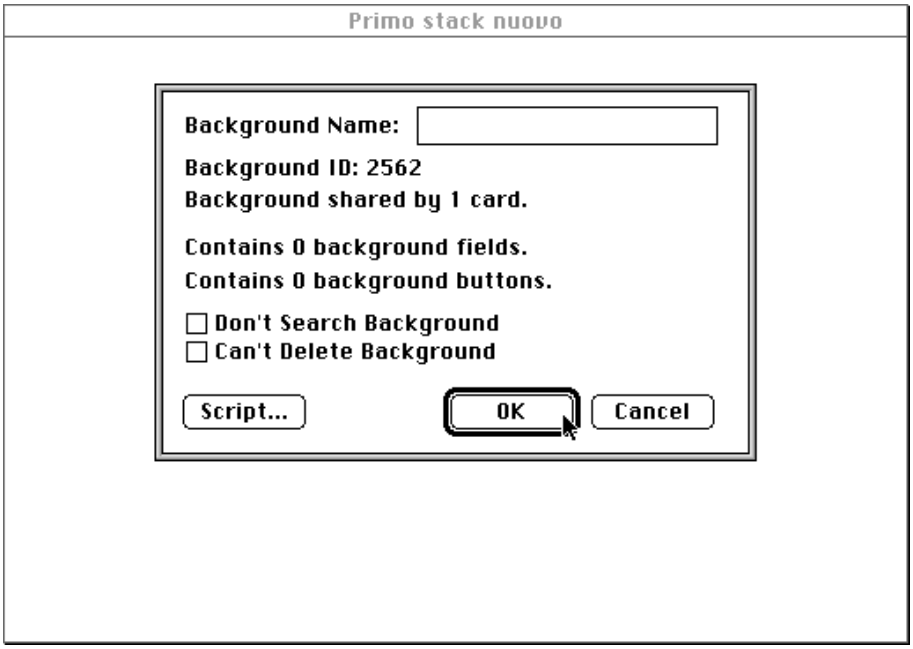


Figura 7.6. - Informazioni relative a una cornice.

La ragione è che ogni oggetto di HyperCard può essere identificato in tre modi diversi. I tre identificatori sono:

- *nome*: una sequenza di caratteri alfanumerici, assegnabile dall'utente ed opzionale;
- *numero d'ordine*: rappresenta il numero dell'oggetto relativamente alla classe a cui appartiene (ad esempio, il pulsante N° 4); questa numerazione può essere cambiata dall'utente;
- *ID* (*Identity*): un numero assegnato dal sistema ad ogni oggetto appena viene creato. Questo numero viene assegnato con criteri di univocità: in uno stack non possono esistere due oggetti dello stesso tipo con lo stesso ID. L'utente non può cambiare l'ID di un oggetto.

Il numero d'ordine viene utilizzato per stabilire l'ordine con cui vengono disegnati gli oggetti sullo schermo. Infatti, anche se lo schermo è piano (e quindi ha solo due dimensioni) l'attribuzione di un numero d'ordine permette di definire una serie di "livelli" ideali a cui situare i vari oggetti, in modo tale che, se essi sono parzialmente o totalmente sovrapposti, quello con il numero più alto "copra" quelli con numeri più bassi.

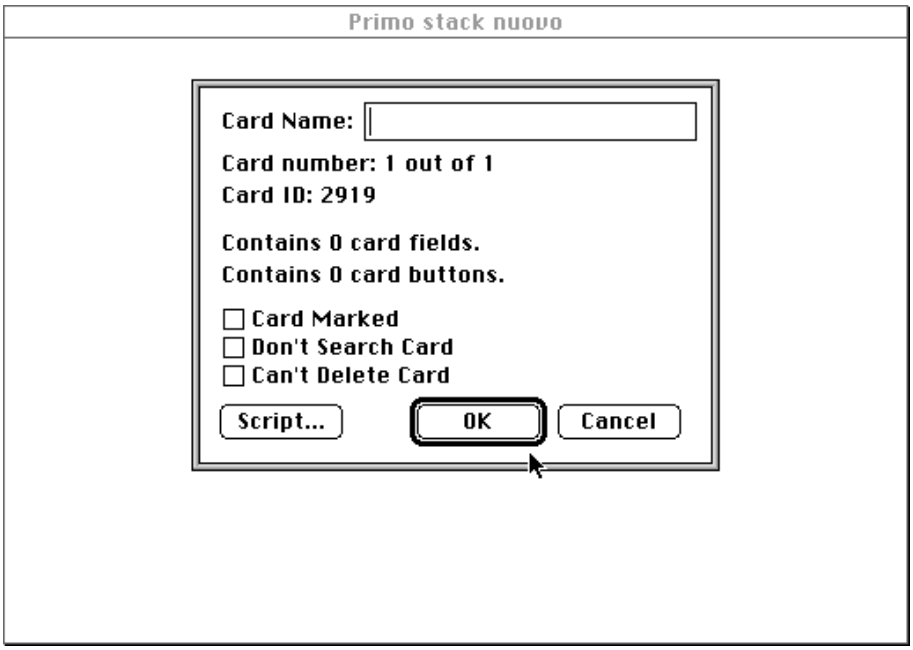


Figura 7.7. - Informazioni relative a una scheda.

Fra le informazioni che vengono fornite a proposito della cornice, si vede che viene segnalato anche il numero di pulsanti e campi associati ad essa, preceduti dalla parola *background*. Ricordiamo infatti che pulsanti e campi possono essere associati sia a cornici che a schede, ed è necessario poter distinguere gli oggetti associati alle cornici da quelli associati alle schede. Per questo si parla di *background fields/buttons* nel caso siano associati a cornici e di *card fields/buttons* se sono associati a schede.

Ogni cornice contiene almeno una scheda, della quale è possibile avere le informazioni sempre mediante una pressione del pulsante del mouse sull'opzione opportuna (Fig. 7.7).

7.1.4.1. Le proprietà degli oggetti

Ogni oggetto HyperCard è dotato di un certo numero di *proprietà*, alcune delle quali possono essere modificate dall'utente secondo le sue specifiche esigenze.
Le proprietà sono parametri associati agli oggetti, che ne completano la definizione. Alcune di esse sono numeriche, altre alfanumeriche, altre infine booleane.
È inutile qui elencarle tutte: basterà ricordare che le proprietà sono diverse a seconda del tipo di oggetto a cui si riferiscono, ad eccezione di alcune, comuni a tutti gli oggetti, come ad esempio il nome e gli altri identificatori, le dimensioni e la posizione sullo schermo, ecc.

7.1.4.2. Gli *script*

Il dialogo che mostra le informazioni relative ad ogni oggetto contiene anche un pulsante di nome *Script…* che consente di associare azioni all'oggetto che si sta esaminando, secondo modalità che vedremo più avanti.
La possibilità di associare azioni non predefinite agli oggetti costituisce la caratteristica più interessante di HyperCard, e sarà oggetto di trattazione approfondita.

7.1.4.3. Costruiamo un primo stack

Come primo esercizio, supponiamo di voler utilizzare HyperCard come database, e precisamente come un indirizzario con rubrica telefonica.
Prima di tutto occorre creare un nuovo stack: selezioniamo la sottovoce *New stack* della voce *File* della barra dei menu.
A questo punto il sistema chiede il nome del nuovo stack da creare con il dialogo mostrato in figura 7.8. Lo chiamiamo "Indirizzi".
Facendo click sul pulsante *New*, il calcolatore presenta un nuovo display (Fig. 7.9) nel quale è sparito lo stack precedente, che è stato sostituito da "Indirizzi". In realtà lo stack che avevamo attivato prima non è stato chiuso, ma è rimasto nascosto da "Indirizzi" che si è sovrapposto ad esso.

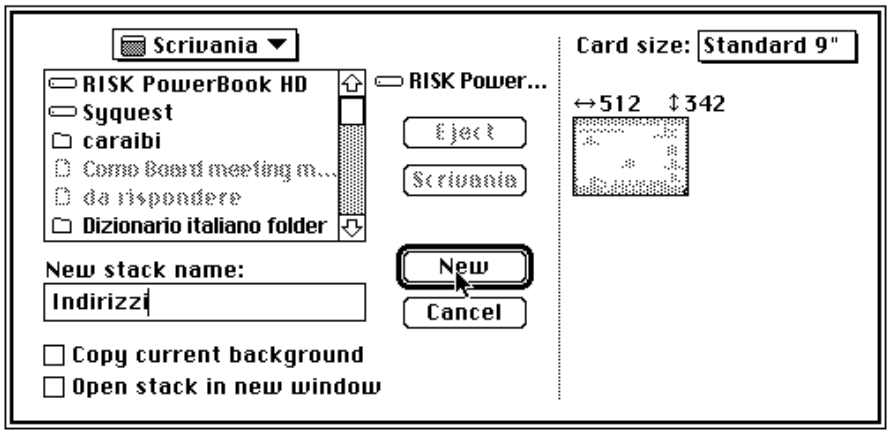


Figura 7.8. - Il dialogo per la creazione di un nuovo stack.

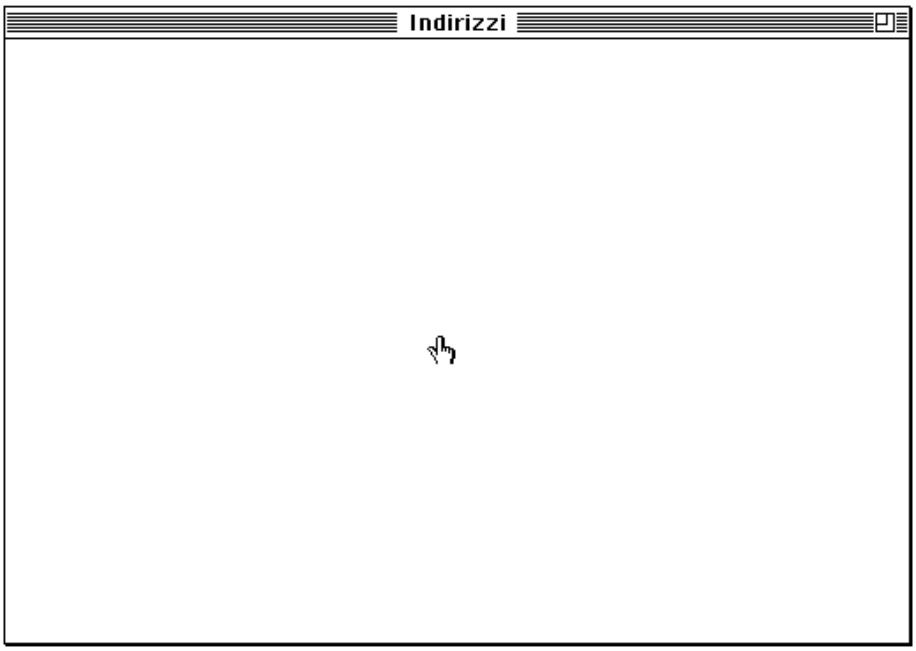


Figura 7.9. - Uno stack appena creato.

Ora dobbiamo definire i campi da inserire in questo stack. Utilizziamo la opzione *Objects* della barra dei menu, voce *New field*. Appare un campo vuoto, di cui possiamo variare posizione e dimensioni trascinandone opportunamente gli angoli con il mouse (Fig. 7.10).

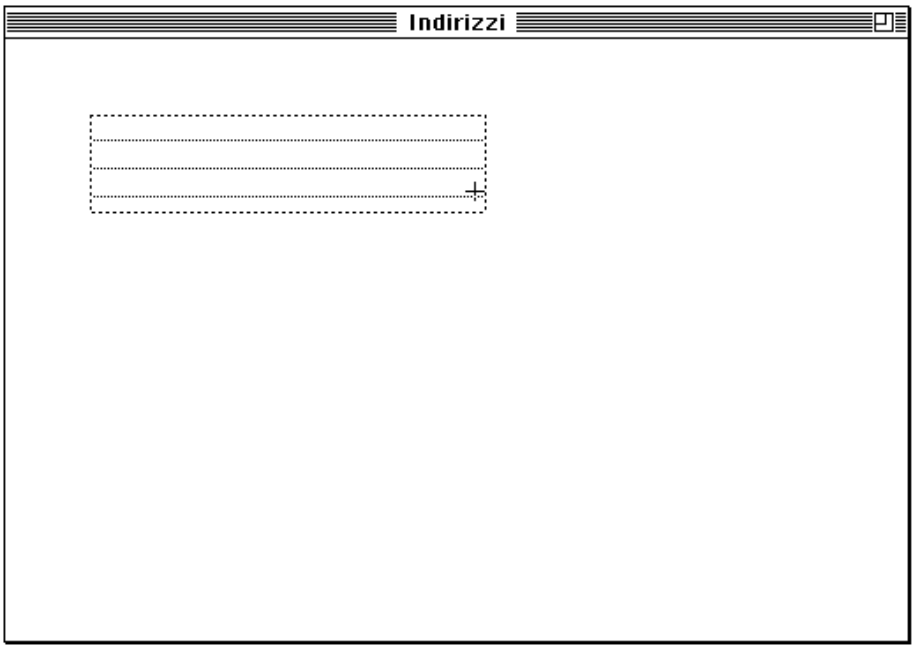


Figura 7.10. - Creazione del primo campo.

Volendo creare altri campi simili possiamo fare ricorso al menu *Edit*, fra le cui sottovoci scegliamo *Copy field*. A questo punto il campo è stato copiato in un apposito buffer nella memoria del calcolatore. Se sempre nel menu *Edit* scegliamo l'opzione *Paste field*, il contenuto di questo buffer viene copiato nuovamente sulla scheda. Osserviamo che in questo momento nella scheda ci sono due campi, ma se ne vede uno solo perché essi sono perfettamente sovrapposti. Basta collocare il mouse sopra l'unico campo che si vede e trascinarlo per accorgersi che il campo precedente era nascosto da quello appena creato come sua copia (Fig. 7.11). Allo stesso modo possiamo creare altri campi, fino ad averne uno per il cognome, uno per il nome, uno per l'indirizzo e uno per il numero di telefono, fino ad avere la scheda configurata come in figura 7.12.
A questo punto lo stack appena creato può già essere utilizzato. Infatti, esplorando i comandi disponibili nella barra dei menu ne troviamo un insieme che permette di inserire nuove schede (*New card*), di cancellare una scheda (*Delete card*), e di spostarci da una scheda ad un'altra (*Go next card*, *Go previous card*, ecc.).

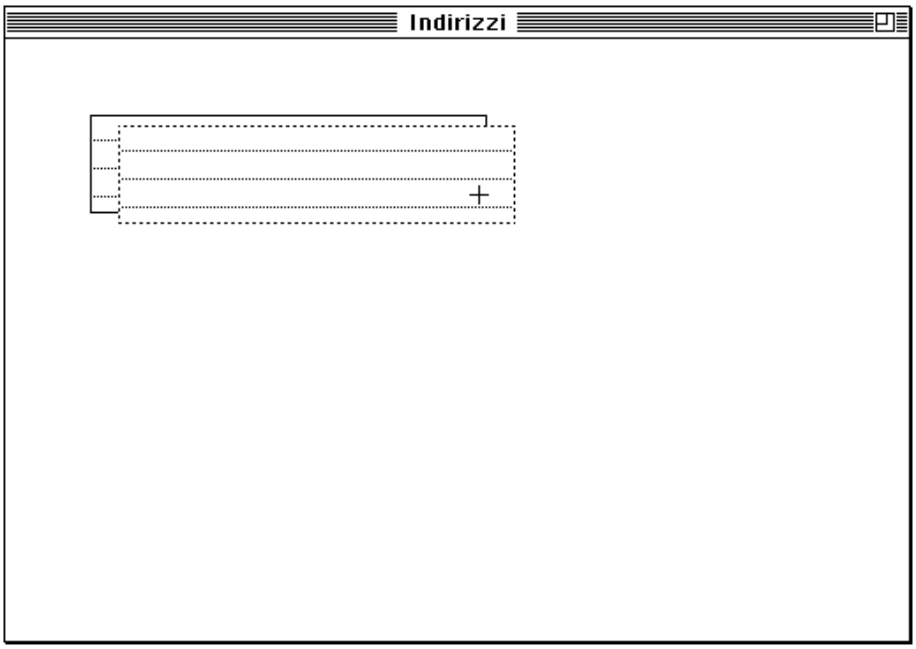


Figura 7.11. - Due campi parzialmente sovrapposti.

Quanto appena detto può apparire complesso e di difficile comprensione, ma occorre osservare che le modalità di definizione degli oggetti in uno stack sono esattamente uguali a quelle utilizzate in tutti gli altri sistemi di programmazione non modale, e sono assolutamente intuitive. Bastano pochi minuti perché una persona totalmente inesperta impari a compiere le operazioni appena descritte, così come avviene per i vari programmi per l'elaborazione dei testi e per il disegno di uso più comune.

7.1.4.4. Associare azioni agli oggetti

Sempre facendo riferimento all'esempio del paragrafo precedente, osserviamo che fare ricorso alla barra dei menu ogni volta che si vuol passare da una scheda alla successiva può alla lunga risultare pesante. Il problema si risolve creando un pulsante, situato in posizione opportuna sulla cornice, facendo click sul quale si ottenga il passaggio alla scheda successiva. Per far questo, creiamo un pulsante che verrà visualizzato con un'icona che raffigura una freccia verso destra per illustrare intuitivamente il suo significato (Fig. 7.13).

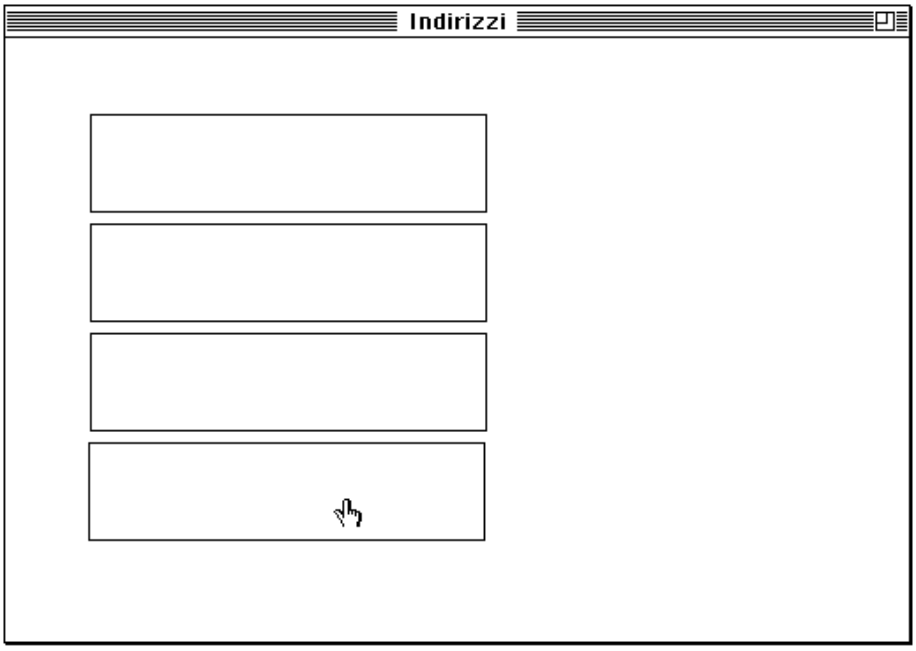


Figura 7.12. - I quattro campi sono stati definiti.

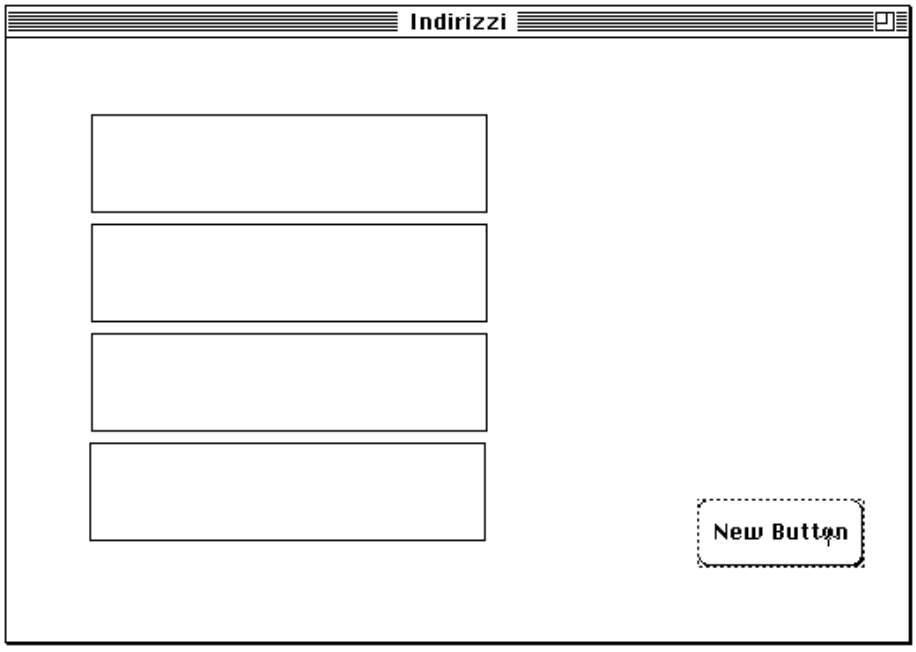


Figura 7.13. - Creazione di un pulsante.

La creazione di un pulsante avviene in modo analogo a quella di un campo; nel dialogo di definizione (Fig. 7.14) è presente un pulsante (*Icon*) che permette di scegliere l'icona più adatta[30] o, se occorre, di crearne una nuova (Fig. 7.15).

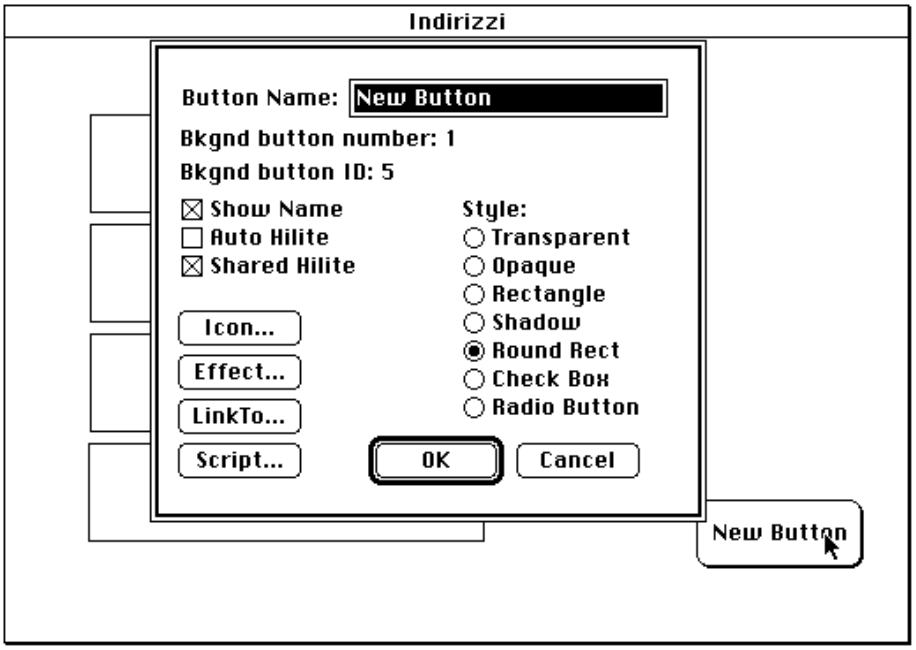


Figura 7.14. - Il dialogo per la definizione delle proprietà di un pulsante.

Ora che il pulsante è stato creato, occorre associargli l'azione desiderata: in questo caso il passaggio alla scheda successiva. Per ottenere questo, premiamo il pulsante *Script* nel suo dialogo di definizione. Appare una finestra di testo, in cui inseriremo il "programma" rappresentato in figura 7.16.
Il significato di questo programma è evidente: più avanti studieremo le modalità con cui esso (e programmi molto più complicati) possono essere scritti.
Facendo ora click sul pulsante appena definito otteniamo il passaggio alla scheda successiva. Dal momento che il pulsante è stato creato sulla cornice, esso apparirà su tutte le schede nella medesima posizione.

[30] In questo caso, il termine "icona" si riferisce a un piccolo disegno che rappresenta simbolicamente l'azione del pulsante.

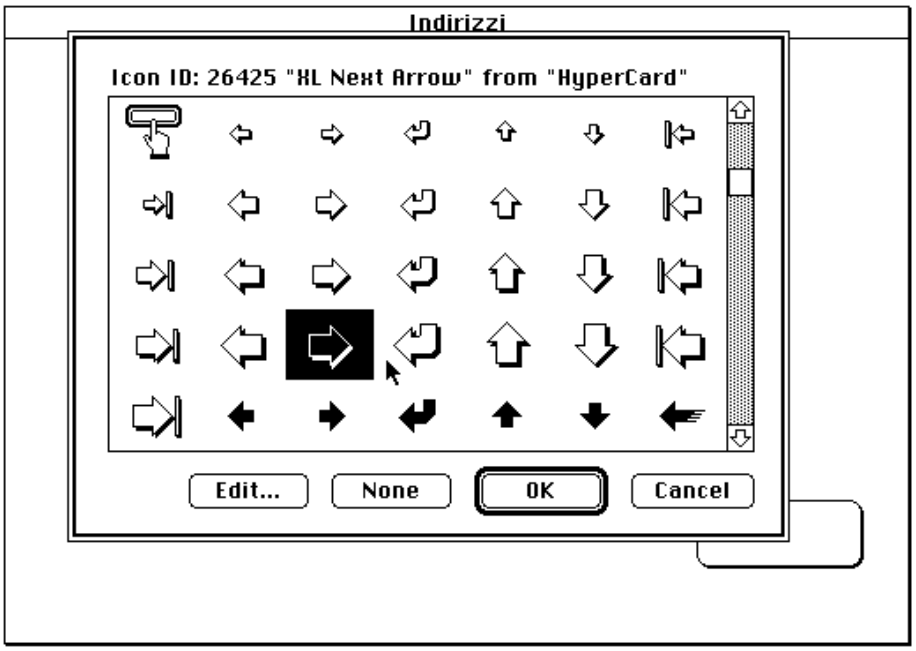


Figura 7.15. - Il dialogo per la definizione delle icone.

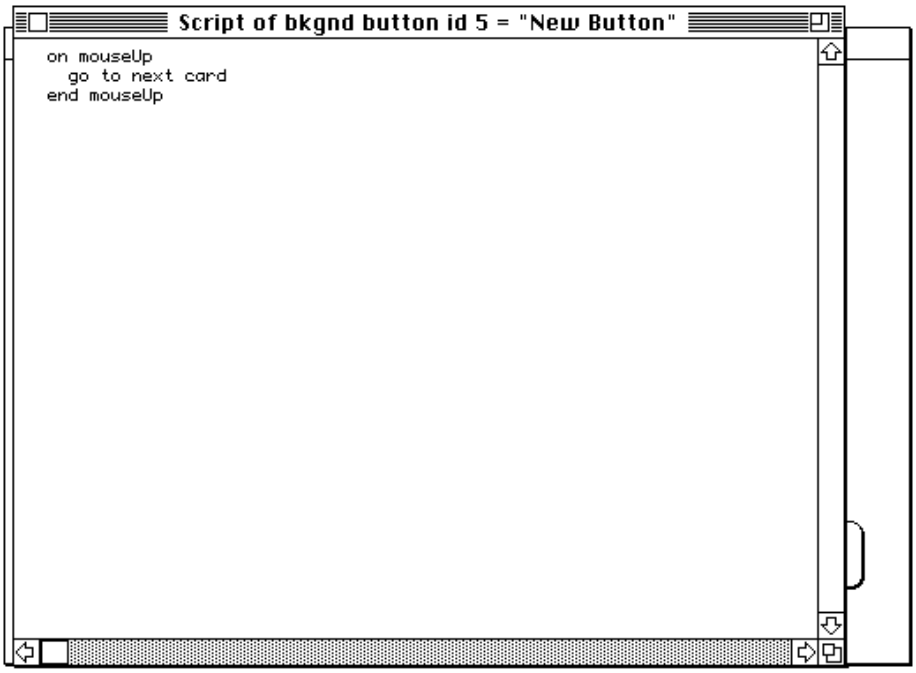


Figura 7.16. - Inserzione di uno script.

In modo del tutto analogo possiamo creare un pulsante per passare alla scheda precedente, che avrà ovviamente un'icona diversa (freccia rivolta verso sinistra), e lo script:

```
on mouseup
  go to previous card
end mouseup
```

Un pulsante che permetta di creare una nuova scheda avrà invece lo script:

```
on mouseup
  domenu "New card"
end mouseup
```

Al termine, la cornice che abbiamo creato avrà l'aspetto mostrato in figura 7.17, e lo stack sarà pronto per essere utilizzato.

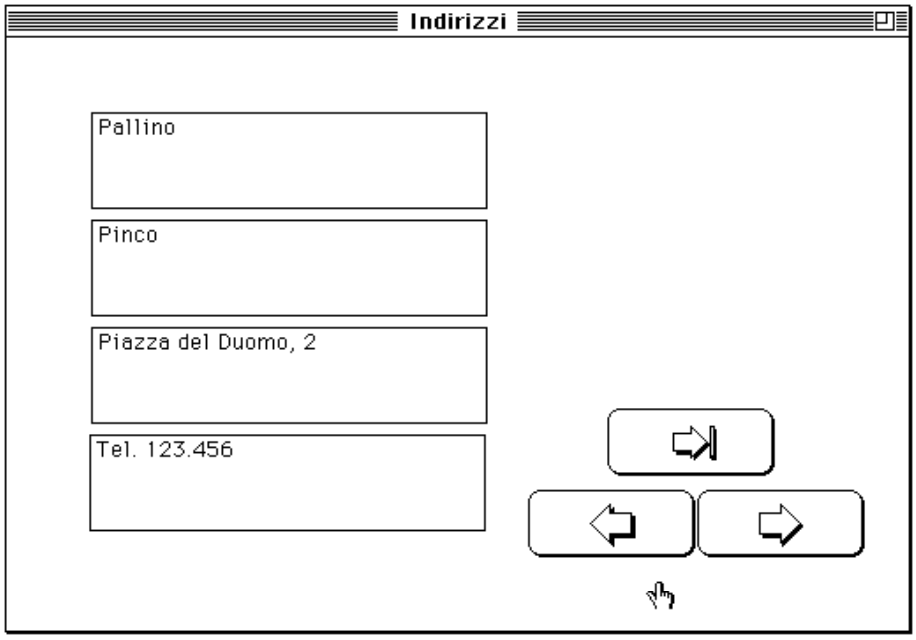


Figura 7.17. - Lo stack pronto per essere usato.

7.1.5. LE FUNZIONI DI DATABASE DI HYPERCARD

Quanto appena detto, unitamente ad una esplorazione dei comandi disponibili sulla barra dei menu, ci porta a concludere che le principali funzioni di database offerte da HyperCard sono:

- possibilità di definire cornici;
- possibilità di creare schede e di inserire informazioni al loro interno;
- possibilità di avere più cornici in uno stack;

- disponibilità di diverse funzioni di *browsing*[31]
  - prossima scheda;
  - scheda precedente;
  - prima scheda dello stack;
  - ultima scheda dello stack;
  - n-esima scheda dello stack;
  - ultime schede visitate: questa funzione permette di rivedere in senso inverso le schede che sono apparse sullo schermo;
  - schede recenti: HyperCard mantiene una coda delle schede visitate, grazie alla quale è possibile scegliere una qualunque delle schede viste per ultime;
- ricerca di informazioni: è possibile trovare le schede che contengono determinate stringhe alfanumeriche;
- possibilità di ordinare le schede secondo diversi criteri.

7.1.6. UN ESEMPIO PIÙ COMPLESSO

Si vuole ora creare una scheda con un solo pulsante e un solo campo, fatta in modo tale che facendo click sul pulsante appaia una scritta nel campo della scheda.
La creazione del pulsante e del campo vanno fatte con le modalità già viste. Questa volta vogliamo però definire il pulsante in maniera diversa, il che vuol dire cambiarne alcune proprietà rispetto alla definizione standard. Per far ciò, occorre fare apparire le attuali proprietà del pulsante, che vengono mostrate utilizzando il comando *Button info…* del menu *Objects*, che fa apparire il dialogo mostrato in figura 7.18.
Vogliamo che il pulsante abbia il contorno ombreggiato, e per questo scegliamo l'opzione *Shadow*.
Assegnamo al pulsante il nome "XYZ", ma decidiamo di non farlo apparire sullo schermo, preferendogli una icona che sceglieremo tra quelle disponibili, visualizzandole con l'opzione *Icon….*
La proprietà *Auto Hilite* fa in modo che il pulsante, se viene premuto il pulsante del mouse su di esso, cambi il suo aspetto grafico per tutto il tempo durante il quale il pulsante del mouse viene tenuto premuto.
Una volta eseguite tutte le modifiche necessarie alle proprietà del pulsante, si esce dalla finestra aperta con *Button info…* e appare la scheda contenente il pulsante. Per uscire dal modo di funzionamento di modifica dei pulsanti e ripristinare il modo di funzionamento normale (*browse*) si utilizza l'opportuno comando del menu *Tools*.

---

[31] Il termine equivale all'italiano "sfogliare", nel senso di "sfogliare un libro".

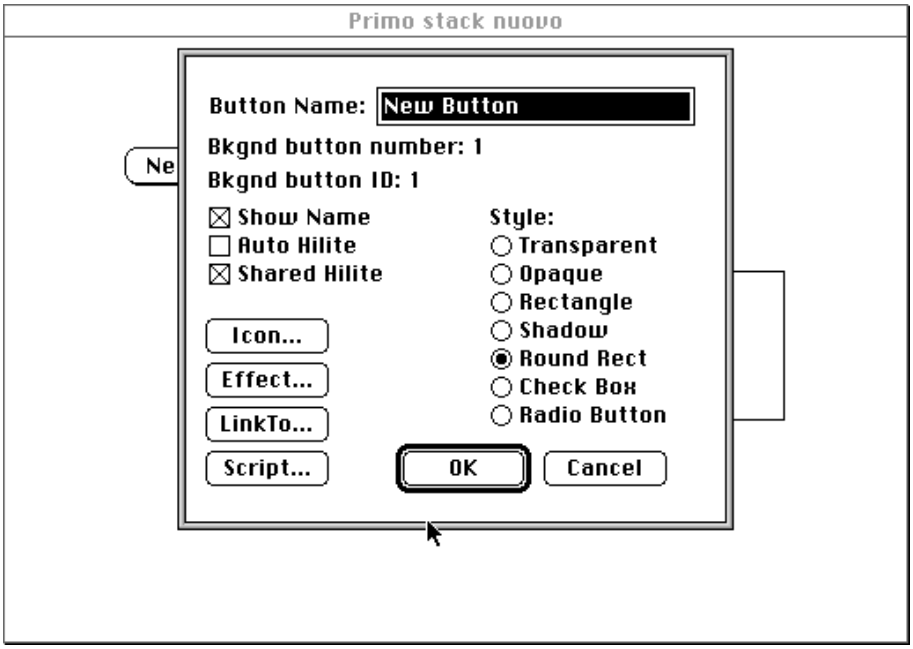


Figura 7.18. - Informazioni relative a un pulsante.

Sempre attraverso il menu *Tools* si possono variare le proprietà dei campi (Fig. 7.19). Nel nostro caso, assegnamo il nome “Risultato” al campo che abbiamo creato sulla scheda, scegliamo il tipo e le dimensioni del carattere con il quale dovrà essere visualizzato il contenuto del campo, scegliamo la formattazione con la quale questo contenuto dovrà essere visualizzato e in genere manipoliamo le proprietà fino ad ottenere un campo dalle caratteristiche desiderate.
Per fare in modo che una pressione sul pulsante “XYZ” causi l’azione desiderata, dovremo infine associargli lo script

```
on mousedown
  beep
  put “ciao!” into card field “risultato”
end mousedown
```

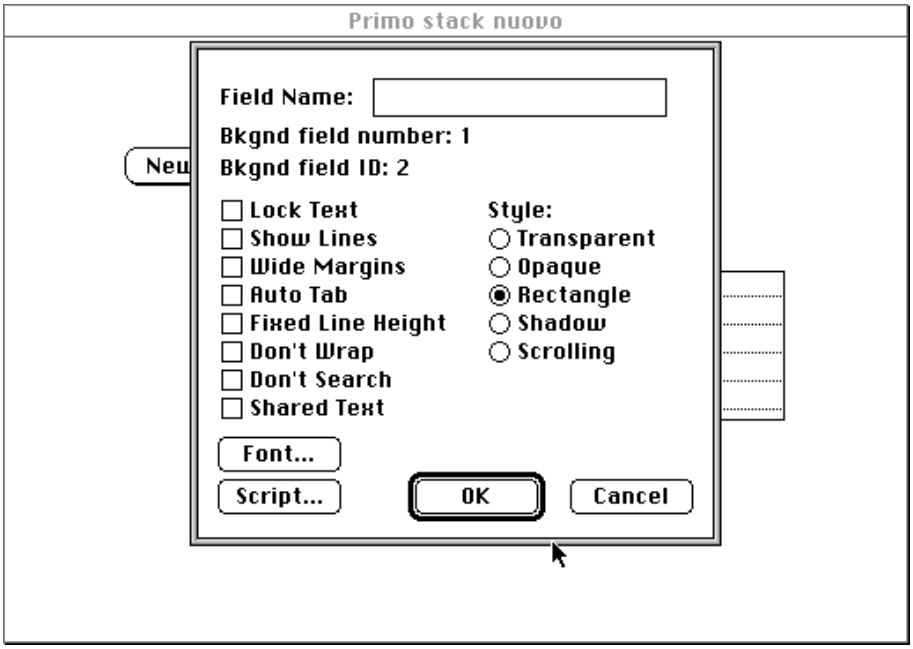


Figura 7.19. - Informazioni relative a un campo.

Se vogliamo anche che rilasciando il tasto del mouse la scritta venga cancellata dal campo e che venga emesso un altro segnale acustico, dovremo aggiungere allo script quanto segue:

```
on mouseup
  beep
  put empty into card field “risultato”
end mouseup
```

7.2. HYPERTALK

Nel paragrafo precedente, oltre agli oggetti che compongono HyperCard, abbiamo visto alcuni semplici esempi di script. Il linguaggio con cui essi sono scritti ha nome HyperTalk™ ed è stato pensato in modo da assomigliare per quanto possibile alla lingua naturale inglese.
Un “programma” HyperTalk è costituito da tante sezioni, ognuna delle quali svolge una determinata azione, che sono fisicamente separate le une dalle altre e sono in genere associate agli oggetti a cui fanno riferimento.
Ogni sezione è a sua volta scritta in un linguaggio che, sia pure con alcune differenze, assomiglia agli altri linguaggi ad alto livello studiati in precedenza.
Le varie sezioni del programma vengono eseguite quando ricevono opportuni “messaggi”. Questo costituisce la base del paradigma *message passing*, e la differenza fondamentale con gli altri linguaggi di programmazione.
Diamo ora una descrizione delle principali caratteristiche di HyperTalk.

7.2.1. I MESSAGGI

Nel capitolo dedicato alla programmazione non modale si era visto che essa offre innegabili vantaggi, soprattutto dal punto di vista dell'interfaccia utente. Dal punto di vista del programmatore invece essa comporta difficoltà dovute al bisogno di gestire in un'unica unità di programma tutti gli eventi che si verificano.
La naturale evoluzione della programmazione non modale consiste nel far sì che gli eventi vengano distribuiti alle parti del programma a cui competono in modo da decentrare le funzioni di controllo e da renderle singolarmente più semplici. Per far questo, si può associare al sistema operativo un sistema di supporto alla programmazione secondo il seguente paradigma:
Il sistema di programmazione:

- riceve gli eventi;
- gestisce direttamente quelli che riguardano la macchina fisica (ad esempio, le interruzioni dell'orologio in tempo reale);
- trasforma gli altri in messaggi;
- spedisce questi messaggi agli oggetti di cui è composto il programma applicativo.

Con questo metodo ogni evento viene comunicato solo agli oggetti direttamente interessati o, per meglio dire, ogni oggetto riceve solamente gli eventi che gli interessano.
Il programma applicativo, ora, risulta essere:

- composto di tanti oggetti;
- ogni oggetto ha un programma associato (ogni programma è un pezzo indipendente dai programmi degli altri oggetti);
- il programma gestisce i messaggi che gli arrivano (tali messaggi sono inviati all'oggetto dal sistema di programmazione).

Ogni messaggio è identificato da:

- *nome*: indica il tipo di messaggio;
- *parametri*: aggiungono ulteriori specificazioni al nome del messaggio, e possono essere in un numero qualsiasi. Molto spesso, i messaggi non hanno nessun parametro, perché il loro nome è sufficiente ad identificarli completamente;
- *destinatario*: in casi particolari è possibile definire il destinatario del messaggio. Ciò in genere non viene fatto, perché esiste un meccanismo che regola in modo automatico la trasmissione dei messaggi.

I parametri dei messaggi possono essere:

- *costanti*;
- *variabili*.

I messaggi possono essere generati:

- dall'hardware, in concomitanza con il verificarsi di determinati eventi;

- dal software, in quanto ogni sezione di programma HyperCard può generare messaggi.

Per chiarire i concetti appena esposti, esaminiamo la sequenza di ciò che accade quando si fa click su un pulsante:

1. si genera una interruzione;
2. il sistema operativo la trasforma in un evento (in questo caso di tipo mousedown) e lo inserisce nella coda degli eventi insieme ai suoi parametri ausiliari (posizione del mouse, istante in cui è stato fatto click, ecc.);
3. il controllo ritorna al sistema di programmazione che riceve l'evento, confronta la posizione del mouse con le coordinate del pulsante con la freccia verso destra e conclude che l'evento riguarda tale pulsante;
4. il sistema di programmazione manda un messaggio *mousedown* al pulsante in questione;
5. al pulsante è associato un programma, detto *Script*, costituito da uno o più *Handler*; ogni handler è un insieme di istruzioni associate ad un dato messaggio, e tali istruzioni sono eseguite solo se il messaggio interessa il relativo handler.

7.2.2. GLI HANDLER

Uno handler, che nella sintassi di HyperTalk è identificato dalla frase di apertura

```
on <nome del messaggio>
```

e da quella di chiusura

```
end <nome del messaggio>
```

è una sezione di programma (*script*) associata ad un oggetto, che viene eseguita quando tale oggetto riceve il messaggio appropriato.
Nei primi esempi di questo capitolo abbiamo visto azioni associate ai messaggi *mousedown* e *mouseup*, che vengono generati rispettivamente quando il pulsante del mouse viene premuto o rilasciato sopra un determinato oggetto.
Quando un handler riceve un messaggio, quest'ultimo viene cancellato, a meno che non si voglia che esso continui a "viaggiare" nel sistema: in tal caso bisogna mettere nel corpo dello handler l'istruzione

```
pass <nome-messaggio>
```

7.2.3. IL PASSAGGIO DEI MESSAGGI

Abbiamo detto che i messaggi vengono inviati agli oggetti direttamente interessati: questo vuol dire che ad esempio i messaggi di mouseup o di mousedown sono spediti direttamente ai pulsanti o ai campi su cui è stato fatto click. Esistono però casi in cui

non è possibile determinare un particolare pulsante o campo a cui il messaggio si riferisce: in questo caso, il messaggio viene mandato alla scheda che appare in quel momento sullo schermo.

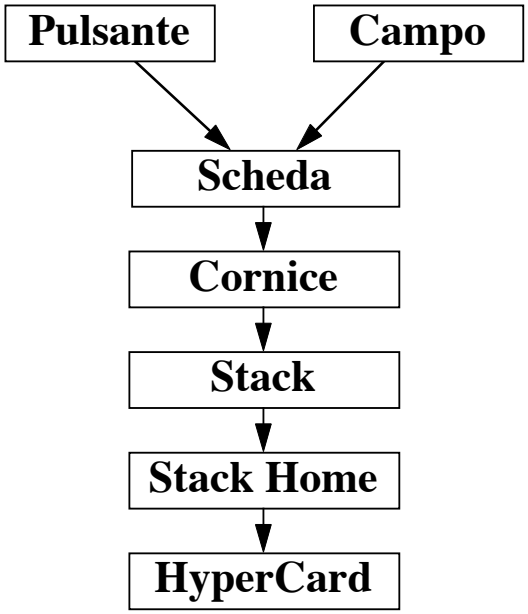


Figura 7.20. - La gerarchia di passaggio dei messaggi in HyperCard.

Qualunque sia l'oggetto che riceve il messaggio, se al suo interno non è contenuto uno handler che lo gestisca, il messaggio inizia un "viaggio" fra i diversi oggetti che compongono lo stack secondo un preciso itinerario che è mostrato in figura 7.20. Se, in tutta la catena indicata in figura, non esiste uno handler che gestisca un determinato messaggio, quest'ultimo viene inviato ad HyperCard, il quale ha la capacità di gestirne in modo automatico un certo numero. Ovviamente, se si tratta di un messaggio generato dal programma dell'utente, e quindi sconosciuto ad HyperCard, viene data una segnalazione di errore.

7.2.4. LE VARIABILI

Possiamo considerare tre tipi di variabili: numeriche, alfanumeriche, booleane. Al contrario di quanto avviene in altri linguaggi, in HyperTalk le variabili non devono essere dichiarate preventivamente: la loro dichiarazione avviene implicitamente all'atto della definizione, cioè quando sono usate per la prima volta. All'interno del sistema, tutte le variabili sono rappresentate come stringhe alfanumeriche, qualunque sia il loro tipo: ciò permette, a prezzo di una leggera inefficienza, una gestione estremamente sofisticata e allo stesso tempo semplice. Ad esempio, una sezione di programma del tipo

```
put “Io ho “ into var1
put “2” & “9” into var2
put “ anni” into var3
put var1 & var2+1 & var3 into var4
```

fa sì che, alla fine dell’esecuzione, la variabile var4 contenga la stringa “Io ho 30 anni”, cosa impossibile da realizzare nella maggior parte degli altri linguaggi.
Dal momento che tutte le variabili hanno la stessa forma, il sistema deve verificarne il tipo ogni volta che effettua un’operazione su di esse. Per esempio, le istruzioni

```
   put “ciao” into pippo
   add “1” to pippo
```

sono sintatticamente corrette, ma semanticamente sbagliate, perché in fase di esecuzione, quando il sistema cerca di sommare `1`, si accorge che `pippo` è una variabile alfanumerica.
Le variabili sono locali, cioè definite e utilizzabili solo all’interno dello handler cui appartengono.
Esiste anche la possibilità di definire variabili globali scrivendo la parola `global` davanti al nome della variabile prima che questa venga utilizzata: ciò renderà accessibile la variabile da qualunque handler che contenga la stessa dichiarazione. Ad esempio, se lo script dello stack contiene il seguente handler:

```
on openstack[32]
   global pippo
   put 0 into pippo
end openstack
```

e quello del pulsante “X” lo script

```
on mouseup
   global pippo
   add 1 to pippo
end mouseup
```

e lo script del pulsante “Y” contiene lo handler

```
on mouseup
   global pippo
   beep pippo
end mouseup
```

facendo click su quest’ultimo, il sistema emetterà un numero di segnali acustici pari al numero di volte che è stato premuto il pulsante “X”.

7.2.4.1. Un esempio di passaggio dei messaggi

Vediamo ora un esempio più complesso di passaggio di messaggi, che include anche diverse particolarità di HyperTalk.
Iniziamo la descrizione con lo script di un pulsante, a cui diamo il nome “Pulsante 1”, che contiene due handler:

```
Script of card button ID 1=“Pulsante 1”
```

---

[32] `openstack` è un messaggio che viene mandato automaticamente all’atto dell’apertura di uno stack.

```
on mouseup -- inizio dello handler del messaggio mouseup

   put empty into card field “risultati”
     -- empty è una costante predefinita che equivale ad una stringa
     -- vuota; questa istruzione quindi “svuota” il campo della
   -- scheda “risultati”

   messx  -- Spedisce il messaggio “messx”. “messx” è un messaggio
          -- definito dall’utente, e non ha parametri

end mouseup -- fine dello handler del messaggio mouseup

on messx -- inizio dello handler del messaggio “messx”

   get fmess(the long name of me) -- vedi la descrizione dettagliata
                                  -- più avanti

   pass messx -- rimette in circolazione il messaggio “messx”

end messx -- fine dello handler del messaggio “messx”
```

In HyperCard è predefinita la variabile `it`, che viene usata da molti comandi del linguaggio: per esempio, `get` è un comando che ha il significato di prendere il valore dell’espressione che segue (in questo caso `fmess(the long name of me)`) e di metterlo nella variabile `it`.
`Fmess` è una funzione e `the long name of me` è il parametro ad essa passato; dal momento che le chiamate delle funzioni seguono lo stesso percorso dei messaggi, le funzioni possono essere contenute nello stesso handler in cui sono chiamate, o in qualunque script di livello gerarchico superiore.
Per quel che riguarda il parametro `the long name of me`, osserviamo che `the long name` indica il nome dettagliato di un oggetto, mentre `of` è una forma alternativa alle parentesi (sarebbe stato lo stesso scrivere `the long name (me)`) e `me` è l’oggetto cui appartiene lo script; quindi `the long name of me` indica il nome dettagliato dell’oggetto cui appartiene lo script.
Possiamo definire la funzione `fmess` nello script dello stack Home:

```
function fmess arg
   beep
   put “messaggio passato da” && arg & return & return¬
   after card field “risultati”
   wait 5 seconds
end fmess
```

`Arg` è il parametro formale della funzione, a cui v iene fatto corrispondere, nel caso in esame, il parametro attuale `the long name of me`; `&` è un operatore che indica la congiunzione di due stringhe, `&&` indica la congiunzione di due stringhe con l’interposizione di uno spazio, `after` accoda la stringa al campo senza cancellare il suo precedente contenuto.
In pratica, `fmess` accoda il nome dell’oggetto che ha passato il messaggio nel campo “risultati”. Nel caso in esame, la funzione `fmess` non ha un valore di ritorno. Essa potrebbe essere quindi sostituita dallo handler di un opportuno messaggio. In questo caso, lo handler di `messx` avrebbe potuto essere

```
on messx

   fmess(the long name of me)

   pass messx

end messx -- fine dello handler del messaggio "messx"
```

e la funzione `fmess` avrebbe dovuto essere sostituita dallo handler

```
on fmess arg
   beep
   put "messaggio passato da" && arg & return & return¬
   after card field "risultati"
   wait 5 seconds
end fmess
```

Abbiamo notato quanto precede per mettere in evidenza il fatto che handler e funzioni sono molto simili fra loro. L'unica differenza è che le funzioni possono avere un parametro di ritorno, e possono essere chiamate all'interno di espressioni, mentre gli handler di messaggi no: le differenze sono le stesse che passano fra procedure e funzioni nel linguaggio Pascal.
Se si vuole attribuire un valore di ritorno ad una funzione, occorre usare l'istruzione

```
return <valore di ritorno>
```

Tornando al nostro esempio, proviamo ad inserire lo handler del messaggio `messx` in tutti gli oggetti dello stack. Se ora premiamo il pulsante, appariranno nel campo "risultati" i nomi di tutti gli oggetti attraverso cui è passato il messaggio `messx`: ciò ci permette di verificare quanto affermato a proposito della gerarchia di trasmissione dei messaggi. Se invece togliamo la frase `pass messx` da uno degli handler, osserveremo che la trasmissione del messaggio si arresta in corrispondenza di quello handler, come affermato in precedenza.

7.2.5. LO HELP ON-LINE DI HYPERCARD

Come abbiamo visto nel capitolo precedente, HyperCard dispone di un sistema di help on-line, che è costituito da due stack: HyperCard Help e HyperTalk Reference. Essi contengono tutte le istruzioni necessarie per utilizzare, rispettivamente, HyperCard e HyperTalk.

7.2.6. UN ULTIMO ESEMPIO

Viene qui presentata la traccia della soluzione di un ultimo problema, il cui completamento è lasciato al lettore. Si tratta di creare uno stack, contenente un'unica scheda, con le seguenti specifiche: si vuole che in un riquadro dello schermo appaia l'ora corrente.
I passaggi da seguire sono i seguenti:

1. si costruisce un nuovo stack assegnandogli un nome e, volendo, altre proprietà (ad esempio la posizione sullo schermo);

2. si crea un campo e lo si chiama, per esempio, col nome “ora corrente”;
3. si crea uno handler, da inserire nello script della scheda, del tipo:

```
on idle[33]
   put the long time into card field “ora corrente”
   pass idle
end idle
```

Aggiungiamo ora, come ulteriore specifica, la possibilità che lo stack agisca da sveglia: vogliamo cioè che il calcolatore emetta un segnale acustico ad un’ora prefissata. Dovremo eseguire i seguenti passi:

4. si crea un campo dove inserire l’ora della sveglia, lo si posiziona, e lo si chiama “ora sveglia”;
5. allo handler precedentemente mostrato si aggiungono le istruzioni:

```
if the long time is card field “ora sveglia”
   then
     repeat for 5
       beep
       wait 30 ticks
     end repeat
end if
```

Un passo successivo potrà essere quello di creare dei pulsanti premendo i quali aumenteranno o diminuiranno automaticamente il valore di ora e minuti della sveglia.

7.3. PROBLEMI ED ESERCIZI

Questo paragrafo contiene, oltre agli esercizi, ulteriori informazioni su HyperCard e HyperTalk. Per questo, alcune soluzioni sono state riportate subito dopo l’esercizio a cui si riferiscono, anziché nell’appendice.
È opportuno comunque ribadire che una completa conoscenza del sistema richiede che venga effettuata una estensiva sperimentazione *in vivo*, utilizzando il sistema e provando a scrivere vari tipi di stack.

Problema 7.1.
La sintassi dello handler

```
on mouseup
   beep
   pass mousedown
end mouseup
```

a. è corretta;

---

[33] Si ricordi che `idle` è un messaggio che viene mandato periodicamente, in concomitanza con le interruzioni dell’orologio in tempo reale.

b. è imprecisa;
c. è scorretta.

Soluzione 7.1.
La risposta è c, in quanto ogni handler può passare solo il messaggio che lo attiva: in questo caso quindi sarebbe corretta solo un'istruzione del tipo `pass mouseup`
Osserviamo per inciso che la risposta b non ha senso, in quanto ogni linguaggio di programmazione ha una sintassi governata da precise regole: la sintassi può essere solo giusta o sbagliata, e non vi sono altre possibilità.

Problema 7.2.
Una variabile intera in HyperTalk è rappresentata con precisione minore di una variabile reale?

Soluzione 7.2.
Falso: in HyperCard non esistono variabili di tipo predefinito, e l'utente non definisce né tipi né variabili all'inizio del programma o delle procedure, come avviene ad esempio per il Pascal, ma si limita ad assegnare un nome ad una variabile solamente quando la utilizza.
Il tipo viene scelto automaticamente da HyperCard nel momento dell'assegnazione e può quindi variare nel tempo se cambia il tipo di dato in essa contenuto.

Problema 7.3.
Quando si fa click su un pulsante, viene mandato un messaggio *mousedown* a tutti i pulsanti della scheda che si trova sullo schermo?

Soluzione 7.3.
Falso: il messaggio viene spedito al pulsante (o, più in generale, all'oggetto) direttamente interessato e, se non viene gestito o viene passato, risale la gerarchia di HyperCard; quindi è solo uno il pulsante della scheda visualizzata che riceve il messaggio.
Un caso leggermente diverso è rappresentato dai campi, per i quali occorre considerare lo stato della proprietà *locktext*:

- facendo click su un campo con *locktext* attiva il messaggio viene mandato al campo che lo ha generato, poi segue la gerarchia di HyperCard come nel caso del pulsante;
- se *locktext* non è attiva significa che si desidera poter scrivere direttamente nel campo. In questo caso il messaggio viene utilizzato dal sistema per attivare le operazioni di editing sul campo, senza ulteriori effetti.

Problema 7.4.
I messaggi inviati ad una scheda sono passati nell'ordine a:
- scheda;
- cornice a cui appartiene la scheda;

- stack a cui appartiene la cornice;
- stack *Home*;
- HyperCard.

Soluzione 7.4.
Vero, ma bisogna ricordare che questa strada viene percorsa solo se il messaggio non viene gestito o viene gestito e passato.
L'ordine di trasmissione dei messaggi può essere modificato utilizzando il comando `send <nome oggetto>` che manda il messaggio direttamente all'oggetto desiderato.
Un'altra possibilità consiste nell'inserire in una lista alcuni stack mediante l'uso dell'istruzione `start using <nome stack>`: ciò permette di passare i messaggi agli stack contenuti nella lista prima che allo stack Home e successivamente a HyperCard.
Il comando opposto `stop using <nome stack>` cancella dalla lista lo stack <nome stack>.
Questi comandi permettono di sfruttare script già presenti senza doverli riscrivere o copiare appesantendo inutilmente il codice.

Problema 7.5.
Per costruire un sistema per la contabilità di uno studio professionale, è meglio usare
a. un ipertesto;
b. un foglio elettronico;
c. un programma in C scritto ad hoc.

Soluzione 7.5.
La risposta corretta è la b.
La scelta di un ipertesto non ha molto senso in quanto questo è adatto soprattutto per funzioni di database, mentre la contabilità di uno studio professionale presenta molte funzioni di calcolo.
La scelta di un foglio elettronico è la più indicata in quanto tale tipo di software è acquistabile sul mercato, con notevole risparmio di tempo anche perché tale prodotto è nato appositamente per questo genere di applicazioni.
Il programma in C non è consigliabile in questo caso perché il suo sviluppo necessita di molto tempo, e quindi denaro, soprattutto la prima volta; il discorso cambierebbe se si avesse intenzione di creare un programma da utilizzare su larga scala, quindi per molti utenti, o se si prevedesse di riutilizzare tutto o parte del programma per altre applicazioni; ciò comporterebbe comunque uno studio complesso e la conoscenza degli eventuali sviluppi.

Problema 7.6.
La proprietà *lockscreen*, quando è attiva, serve principalmente a
a. evitare confusioni all'utente;
b. rendere più veloci le operazioni che implicano l'esame di molte schede;
c. bloccare la trasmissione dei messaggi.

Soluzione 7.6.
La risposta corretta in questo caso non è univoca: infatti l'utilità principale di tale comando, che fa in modo che lo schermo del calcolatore non venga modificato finché essa è attiva, è di aumentare la velocità di esecuzione del programma ma, indirettamente, si evita di creare confusione all'utente che vedrebbe solo dei flash decisamente poco utili.
La proprietà che blocca la trasmissione dei messaggi è invece *lockmessages*.

Problema 7.7.
Il programma

```
set lockscreen to true
put "Buongiorno"
wait 120 seconds
set lockscreen to false
```

ha effetto sullo schermo

a. immediato;
b. dopo due minuti;
c. mai.

Soluzione 7.7.
L'effetto di *lockscreen* si ha solamente sulla visualizzazione delle schede, non sulla *message box* o su eventuali messaggi di errore: la risposta corretta è quindi la c.

Problema 7.8.
Il programma

```
Function conta frase
   get the number of words of frase
end conta
```

ritorna il numero di parole contenute nel parametro *frase*?

Soluzione 7.8.
La funzione conta il numero di parole della frase (parametro) e lo salva nella variabile `it`. Manca però l'istruzione di assegnamento di `it`. alla funzione conta: `return it`.

Problema 7.9.
Una funzione può avere più di un parametro?

Soluzione 7.9.
Come per il Pascal le funzioni possono avere un qualunque numero di parametri; in HyperCard si possono definire anche funzioni con un numero variabile di parametri.

Problema 7.10.
Nell'ipotesi che lo stack corrente contenga un certo numero di volte la stringa "ANTONIO", lo script

```
find "Antonio"
```

a. non la trova mai;
b. la trova.

Soluzione 7.10.
HyperCard non fa distinzione fra lettere maiuscole e minuscole, quindi la parola verrà trovata.

Problema 7.11.
Nell'ipotesi che lo stack contenga un certo numero di volte la stringa "Antonio", lo script

```
repeat
   find "Antonio"
end repeat
```

a. si ripete all'infinito;
b. esce dopo aver trovato l'ultima occorrenza della stringa "Antonio";
c. segnala errore dopo aver trovato l'ultima occorrenza della stringa "Antonio".

Soluzione 7.11.
Questo è un ciclo senza uscita: infatti, per come è costruita la struttura dati, le schede sono disposte ad anello chiuso ovvero la scheda successiva all'ultima è la prima e viceversa, quindi dopo aver trovato tutte le schede contenenti la stringa in questione il programma ripartirà dall'inizio.
HyperCard memorizza l'ultima scheda trovata come punto di partenza nella ricerca e segnala l'errore solamente se percorre un giro completo senza trovare nulla: quindi, se esiste almeno una scheda con la stringa cercata, si ha un ciclo infinito.

Problema 7.12.*
Un campo non ha nessun attributo particolare. Facendo click su di esso,
1 non succede nulla;
2 viene mandato un messaggio *mousedown*;
3 inizia una procedura di editing.

Problema 7.13.
La frase

```
char 1 to 25 of line 1 of field "campo x"
```

a. è un *chunk*;
b. è non ha senso.

Soluzione 7.13.
Questo è un *chunk*, cioè uno dei molteplici modi con cui è possibile identificare parti del contenuto di un campo o di una variabile, per i quali si rimanda al manuale del sistema.

Problema 7.14.
Lo script che segue ha senso?

```
on mouseup
   on idle
   …
   end idle
end mouseup
```

Soluzione 7.14.
I messaggi vengono gestiti dagli handler in ordine ed uno per volta. Ogni handler può gestire un solo messaggio (quello che lo attiva) quindi al suo interno non è possibile annidare la gestione di altri messaggi.

Problema 7.15.*
A ogni campo deve necessariamente essere associato uno script che contenga almeno uno handler?
a. sì, sempre;
b. no;
c. no, ma non avrebbe senso.

Problema 7.16.*
A ogni pulsante deve necessariamente essere associato uno script?
a. sì, sempre;
b. no;
c. no, ma non avrebbe senso.

Problema 7.17.*
A ogni scheda deve necessariamente essere associato uno script?
a. sì, sempre;
b. no;
c. no, ma non avrebbe senso.

Problema 7.18.*
A ogni stack deve necessariamente essere associato uno script che contenga almeno uno handler?
a. sì, sempre;
b. no;
c. no, ma non avrebbe senso.

Problema 7.19.*
A prescindere dalla perdita di qualche effetto grafico, ogni pulsante potrebbe essere sostituito da un campo?

Problema 7.20.*
A prescindere dalla perdita di qualche effetto grafico, ogni campo potrebbe essere sostituito da un pulsante?

Problema 7.21.*
È possibile scrivere una funzione in HyperTalk che abbia più di un parametro di ritorno?
a. sì, sempre;
b. no, mai;
c. sì, ma solo usando variabili globali;
d. si, ma solo se la funzione è inclusa nello script che la chiama.

Problema 7.22.*
HyperCard ha un equivalente della struttura di controllo *case* del Pascal?

Problema 7.23.*
HyperTalk ha la possibilità di definire vettori e matrici?

Problema 7.24.*
HyperTalk permette la definizione di tipi di dati?

Problema 7.25.*
I messaggi che non incontrano uno handler scritto dall'utente che li gestisca vengono inviati a HyperCard?

Problema 7.26.*
I messaggi devono obbligatoriamente avere almeno un parametro?

Problema 7.27.*
I messaggi, una volta che sono stati gestiti da uno handler, continuano il loro viaggio verso i livelli più alti?
a. no, mai;
b. sì, sempre;
c. solo se il programmatore lo vuole.

Problema 7.28.
I nomi degli oggetti possono essere cambiati:
a. dall'utente, indipendentemente dal valore della proprietà *userlevel*;
b. da uno script, indipendentemente dal valore della proprietà *userlevel*.

Soluzione 7.28.
b. Possono essere cambiati dall'utente solo se la proprietà *userlevel* è ad un livello sufficientemente alto.

Problema 7.29.*
I numeri di ID degli oggetti possono essere cambiati:
a. dall'utente, indipendentemente dal valore della proprietà *userlevel*;
b. da uno script, indipendentemente dal valore della proprietà *userlevel*;
c. non possono essere cambiati.

Problema 7.30.*
I parametri dei messaggi di HyperTalk vengono passati per nome o per valore?

Problema 7.31.*
I parametri dei messaggi possono essere solo variabili numeriche?
a. vero;
b. falso: non si definisce il tipo delle variabili;
c. falso: possono essere anche variabili definite come stringhe alfanumeriche;
d. falso: a parte il fatto che il tipo delle variabili non è definito, possono essere anche costanti.

Problema 7.32.*
Fra quelli elencati qui sotto, il pregio principale di HyperCard è
a. la velocità di esecuzione dei programmi;
b. la facilità di scrittura dei programmi;
c. la disponibilità di una ricca libreria di funzioni matematiche.

Problema 7.33.
In HyperCard, è possibile cancellare un pulsante di una cornice da una sola scheda della cornice stessa?

Soluzione 7.33.
No. Tutt'al più è possibile "coprirlo" con un disegno, con un altro pulsante o con un campo.

Problema 7.34.*
È possibile che esista una cornice che non contiene nessuna scheda?

Problema 7.35.*
È possibile che esista uno stack che non contiene nessuna cornice?

Problema 7.36.*
È possibile definire un campo che esista in una sola scheda, anche se la cornice contiene più schede?

Problema 7.37.*
È possibile definire un campo della cornice che abbia lo stesso testo per tutte le schede della cornice stessa e che, se si cambia il suo contenuto in una scheda, rifletta questo cambiamento in tutte le altre schede della stessa cornice?

Problema 7.38.*
È possibile definire un campo della cornice che abbia un testo diverso per ogni scheda della cornice stessa?

Problema 7.39.*
È possibile definire un pulsante che esista in una sola scheda, anche se la cornice contiene più schede?

Problema 7.40.*
È possibile definire un pulsante della cornice che abbia un nome diverso per ogni scheda della cornice stessa?

Problema 7.41.*
È possibile mandare un messaggio senza specificarne il destinatario?

Problema 7.42.*
È possibile scrivere uno handler che contiene al suo interno un altro handler?

Problema 7.43.*
In HyperCard, il messaggio è l'unico modo che un oggetto ha per passare informazioni ad un altro oggetto?

Problema 7.44.*
Il testo di un campo dichiarato *locked* non può essere cambiato in nessun modo?

Problema 7.45.*
In HyperTalk, le funzioni definite in uno script possono essere chiamate:
a. solo dallo script in cui sono definite;
b. solo da alcuni script;
c. da tutti gli script dello stack attivo;
d. da tutti gli script di tutti gli stack aperti.

Problema 7.46.*
Le variabili definite in uno handler hanno valore anche per gli altri handler contenuti nello stesso script?

Problema 7.47.*
Le variabili globali hanno valore solo per gli handler e le funzioni contenuti nello stesso script in cui è contenuta la loro definizione?

Problema 7.48.*
Lo stesso handler può gestire messaggi con nomi diversi?

Problema 7.49.
Lo stesso messaggio può essere gestito da più di uno handler?
a. sì, perché i messaggi raggiungono comunque tutti gli oggetti dello stack;
b. no;
c. sì, ma solo se gli handler non appartengono al medesimo script;
d. sì, ma solo con accorgimenti particolari.

Soluzione 7.49.
d. Occorre infatti che lo handler che gestisce il messaggio contenga l'istruzione `pass <nome messaggio>` perché esso sia rimesso in circolazione.

Problema 7.50.
In HyperCard, mediante uno script è possibile creare un nuovo pulsante sulla cornice corrente?

Soluzione 7.50.
Sì: mediante l'istruzione domenu è possibile fare eseguire qualunque comando della barra dei menu.

Problema 7.51.*
In HyperCard, si può mandare un messaggio a uno specifico oggetto?
a. sì;
b. no: i messaggi seguono sempre la gerarchia prestabilita.

Problema 7.52.*
Sulla stessa scheda possono esistere due campi con lo stesso nome?
a. sì;
b. no;
c. sì, ma a condizione che uno sia un campo della scheda e l'altro sia un campo della cornice.

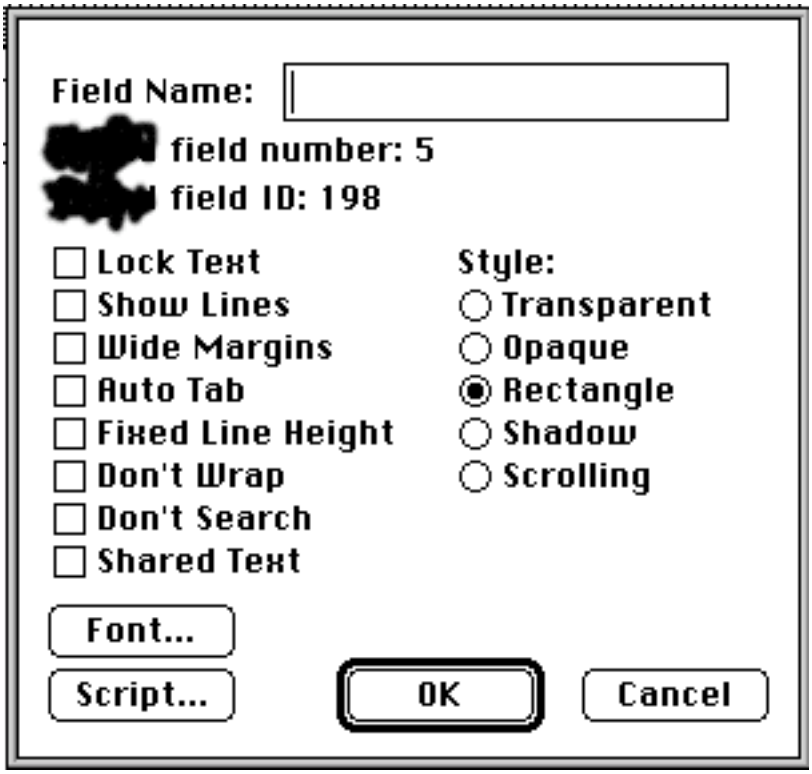
Problema 7.53.*
Un campo della cornice definito *shared text* può avere un testo diverso per ogni scheda?

Problema 7.54.*
Un campo di una cornice può apparire in posizioni diverse sulle varie schede della cornice stessa?

Problema 7.55.*
Un messaggio indica sempre che si è verificato un evento fisico (tasto premuto, pulsante del mouse premuto, ecc.)?

Problema 7.56*
In HyperTalk devono essere dichiarate:
a. tutte le variabili;
b. solo le variabili locali;
c. solo le variabili globali;
d. nessuna variabile.

Problema 7.57.*
La figura che segue rappresenta il dialogo di definizione di un campo. Determinare cosa c'è scritto sotto la cancellatura.



a. *Card*;
b. *Bkgnd*;
c. potrebbero esserci entrambe le parole;
d. non è possibile determinarla.

Problema 7.58.*
Se l'utente risponde "Sì", lo script

```
on mouseUp
   answer "Cancello tutte le schede?" with "Sì" or "No"
   if it <> "Sì" then exit mouseUp
   lock screen
   go second card of bg "risultati"
   put the number of cards of background "risultati" into numero
```

```
   repeat with n=1 to numero-1
     put "Deleting... "&numero-n
     domenu "delete card"
   end repeat
   hide message
end mouseUp
```

cancella realmente tutte le schede della cornice `risultati?`

a. sì, tutte;
b. tutte meno una;
c. tutte meno due;
d. una sola;
e. nessuna.

Problema 7.59.*

Se lo script

```
on mouseup
   go to stack me
end mouseup
```

segue appartiene a un campo col testo bloccato, il cui nome è "Conti" e il cui contenuto è "Home", facendo click su di esso si salta

a. da nessuna parte: lo script è assurdo;
b. allo stack "conti";
c. allo stack "home".

Problema 7.60.*

Una scheda può appartenere a due cornici diverse?

8

ALGORITMI E COMPLESSITÀ

Questo capitolo si propone di dare al lettore un insieme di algoritmi di base, utili in un gran numero di applicazioni pratiche. Infatti, nella maggior parte dei casi, lo sviluppo di programmi per calcolatore comporta la risoluzione di problemi che, pur con differenze nella struttura e nelle dimensioni dei dati, appartengono sempre alle stesse categorie. Fra essi possiamo elencare quelli relativi al mantenimento di elenchi (inserzione, cancellazione, ordinamento, ricerca di dati), quelli per la manipolazione di stringhe alfanumeriche, ecc.
Esistono manuali che raccolgono gli algoritmi di impiego più comune: in questo capitolo, noi elencheremo gli algoritmi di uso più frequente relativi al mantenimento di elenchi.
Contemporaneamente, ci porremo il problema, di fondamentale importanza nella pratica, di riuscire in qualche modo a valutare preventivamente il tempo di esecuzione di un particolare algoritmo, in quanto, in certe situazioni, il tempo stesso può divenire un fattore assolutamente vincolante.
Per far questo, dovremo introdurre il concetto di *complessità di calcolo* che, come vedremo, costituisce un indice in base al quale è possibile valutare l'efficienza di un determinato algoritmo in una determinata situazione.

8.1. IL CONCETTO DI COMPLESSITÀ

Qualunque algoritmo implementato su un calcolatore richiede fondamentalmente due risorse: memoria per il programma e per i dati, e tempo per essere eseguito. Per ogni

problema risolvibile, esistono in teoria infiniti algoritmi che lo risolvono: anche se scartiamo a priori quelli inutilmente complicati, molto spesso la scelta non si riduce ad un unico algoritmo, perché tipi diversi di dati da elaborare possono rendere più conveniente la scelta di un algoritmo rispetto ad un altro.
In altre parole, per un determinato problema non esiste necessariamente un algoritmo ottimo in assoluto.
La necessità di disporre in maniera più o meno grande delle due risorse prima menzionate introduce il concetto di *complessità*, cioè di una funzione che leghi le dimensioni delle risorse occorrenti ai dati da trattare. A seconda che si faccia riferimento all'occupazione di memoria o al tempo di calcolo, parleremo di *complessità spaziale* o di *complessità temporale*. Nella pratica, quest'ultima è molto più importante della prima, soprattutto in seguito agli sviluppi tecnologici che hanno reso la memoria un bene di costo molto limitato. La rapidità di calcolo invece, nonostante il costante incremento della velocità degli elaboratori, è sempre un fattore importantissimo nella determinazione della bontà e, in alcuni casi, addirittura dell'applicabilità di un determinato algoritmo.
La valutazione del tempo di calcolo non sarebbe particolarmente complessa se esso fosse invariabile: al limite, sarebbe sufficiente scrivere l'algoritmo e farlo girare sul calcolatore prescelto misurandone il tempo di esecuzione. Dobbiamo però tenere presente che, in genere, il numero e il tipo dei dati che l'algoritmo dovrà trattare non sono noti al momento della programmazione, in quanto dipendono dall'applicazione che dell'algoritmo verrà fatta. Questo può portare a spiacevoli conseguenze: alcuni algoritmi mostrano grande efficienza con certi insiemi di dati, e diventano penosamente lenti con altri. Oppure, mantenendo uguale il tipo di dati, essi si comportano egregiamente quando i dati sono pochi, ma il loro tempo di esecuzione aumenta vertiginosamente quando i dati aumentano.
Per chiarire il concetto, supponiamo di aver a che fare con un algoritmo che, durante l'esecuzione, debba calcolare tutte le permutazioni dei dati in ingresso. È chiaro che, dal momento che le possibili permutazioni di n elementi sono $n!$, se l'algoritmo impiega, supponiamo, 10 ms per ogni iterazione, con 5 dati in ingresso la sua esecuzione richiederà 1,2 s; con 10 dati occorreranno circa 10 ore e con 50 quasi 10^{55} anni!
Il concetto di complessità così definito si basa sulla considerazione che le prestazioni di un algoritmo siano proporzionali in qualche modo alla quantità di dati da elaborare e quindi alla quantità di istruzioni da eseguire, considerazione che è in generale valida ma necessita di alcune precisazioni che verranno fatte in seguito.
Non è facile, dato un algoritmo, scrivere una formula che metta rigorosamente in relazione n con il tempo di esecuzione; è però possibile stimare un tempo massimo di esecuzione. Quindi, immaginando un grafico che rappresenti nel piano dimensioni/tempo le prestazioni di un dato algoritmo in modo rigoroso, quello che si può fare realisticamente è elaborare una curva più semplice, ma simile a quella reale che, per ogni n, abbia sempre un valore maggiore o uguale a quello del tempo realmente necessario, cioè che massimizzi la funzione. Quello che si ottiene è un'idea abbastanza realistica di come si potrà comportare l'algoritmo scelto e quindi la possibilità di valutarne le prestazioni.

Gli algoritmi che verranno presentati sono scritti nel linguaggio C che è un linguaggio ad alto livello e che di conseguenza permette di creare programmi in grado di essere eseguiti da hardware con caratteristiche differenti.
Si presenta quindi la necessità di distinguere il concetto di *quante volte viene eseguita ogni istruzione* da quello di *quanto tempo occorre per eseguire ogni istruzione*. Infatti il numero di istruzioni da eseguire non fornisce sempre un'indicazione utile.
Ciò è facilmente dimostrabile con un esempio: se si considerano due algoritmi A e B, con A che implica l'esecuzione di 100 istruzioni mentre B di sole 50 (quindi A con complessità 100 e B 50) ma ogni istruzione di B necessita di un tempo tre volte maggiore di quello impiegato per eseguire le istruzioni di A, è chiaro che l'algoritmo più efficiente dal punto di vista del tempo di esecuzione è A, pur avendo una complessità maggiore.
Se poi usiamo gli stessi algoritmi su di una macchina diversa che esegue le istruzioni di B in un tempo di poco maggiore di quello per le istruzioni di A, vediamo che l'algoritmo più efficiente questa volta risulta essere B. Quindi, concludendo, la dimensione di un algoritmo è in genere un buon metodo di valutazione, è tuttavia approssimativo e non assoluto.

8.2. ALGORITMI E TEMPI DI CALCOLO

Ogni algoritmo che costruiamo per risolvere un problema è caratterizzato da un suo tempo di calcolo che determina la quantità di tempo necessaria affinché tale algoritmo venga completamente eseguito.
Ecco qui di seguito alcuni esempi che ci possono facilmente chiarire come tale tempo possa variare a seconda di quale algoritmo venga eseguito.
Tempo di calcolo proporzionale a

- 1 per azzerare il primo elemento di un vettore di N elementi;
- N per sommare 1 a tutti gli elementi di un vettore di N elementi;[34]
- N^2 per eseguire due cicli `for` intrecciati :

```
for i := 0 to N-1 do
for j := 0 to N-1 do...
```

- N^3 per eseguire tre cicli intrecciati;
- $\log N$ (vedi esempi successivi);
- $N \log N$ (vedi esempi successivi).

[34] È facile pensare come questo coefficiente di proporzionalità potrebbe variare se occorresse mettere in memoria, a più riprese, parti più o meno grandi del vettore v quando tale vettore non possa risiedere in memoria per intero.

8.3. LA *O-NOTATION* (*BIG O-NOTATION*)

Dopo aver visto gli esempi riguardanti i tempi di calcolo di vari algoritmi, è opportuno dare alcune definizioni che ci permettano di valutare approssimativamente il tempo di calcolo per un generico algoritmo che vogliamo studiare.

Definizione

Date due funzioni $g(n)$ e $f(n)$, $g(n)$ è $O(f(n))$ se esistono due costanti c e N tali che per ogni n preso da N in poi sia verificata l'espressione $g(n) \leq cf(n)$.

Vediamo un esempio di applicazione di questa definizione.

Sia data la funzione $f(n) = 5n^2 + 15$.

$5n^2 + 15 = O(n^2)$, perché $5n^2 + 15 \leq 6n^2$ per $n \geq 4$

Definizione

Una funzione $f(n)$ si dice monotonicamente crescente se $f(n_1) \geq f(n_2)$ per ogni $n_1 \geq n_2$.

Teorema

Per ogni costante $c > 0$, per ogni $a > 1$ e per ogni funzione $f(n)$ monotonicamente crescente è verificata la seguente equazione:

$(f(n))^c = O(a^{f(n)})$

ovvero ogni funzione esponenziale "cresce più in fretta" di una funzione polinomiale.

Per concludere, presentiamo due lemmi che legano le complessità di sottoparti di algoritmi.

Lemma 1: $f(n) = O(s(n)), g(n) = O(r(n)) \Rightarrow f(n) + g(n) = O(s(n) + r(n))$

Lemma 2: $f(n) = O(s(n)), g(n) = O(r(n)) \Rightarrow f(n)g(n) = O(s(n)r(n))$

Essi sono utili quando si debba valutare la complessità globale di algoritmi composti da parti la cui complessità è nota.

8.4. ALGORITMI PER I VETTORI

Nei paragrafi che seguono saranno presentati gli algoritmi più comunemente usati per effettuare operazioni su insiemi di dati strutturati in vettori. Adotteremo vettori composti da un certo numero di elementi, ognuno dei quali è un record contenente due campi numerici: il primo è la cosiddetta *chiave*, in base alla quale saranno effettuate ricerche e ordinamenti, e il secondo rappresenta il contenuto informativo vero e proprio del record. La definizione della struttura dati usata in tutti gli esempi è quindi:

```
static struct node
   { int key; int info; };
static struct node a[maxN+1];
static int N;
```

Per poter effettuare le operazioni che seguono è necessario, prima di ogni altra cosa, inizializzare la struttura dati, in modo da portarla in una situazione nota. Ciò si ottiene semplicemente con la procedura:

```
seqinitialize()
  { N=0; }
```
Da questo momento in poi, la variabile N conterrà il numero di elementi validi presenti nel vettore.

8.4.1. INSERZIONE E CANCELLAZIONE DI ELEMENTI IN UN VETTORE

L'inserzione di un elemento può essere effettuata in due modi: il primo, più semplice, prevede che l'elemento venga inserito dopo l'ultimo elemento utile contenuto nel vettore:

```
seqinsert (int v, int info)
  {a[++N].key=v; a[N].info=info;}
```
Si osservi che la complessità di questo metodo è costante, in quanto non dipende né dalle dimensioni del vettore, né dal numero di elementi già utilizzati.
Il secondo metodo prevede invece che ogni elemento sia inserito già "al posto giusto", cioè in una posizione tale da mantenere gli elementi del vettore ordinati secondo un certo criterio. Questo secondo metodo sarà trattato più avanti, quando si parlerà dell'algoritmo di *insertion sort*.
Per quanto riguarda la cancellazione, osserviamo che essa comporta lo spostamento di tutti gli elementi che seguono all'indietro di una posizione, per riempire il vuoto lasciato dall'elemento che è stato tolto. L'esempio che segue elimina l'x-esimo elemento del vettore:

```
seqdelete (int x)
  {
    while (x<N);
      {
        a[x].key=a[x+1].key;
        a[x].info=a[x+1].info;
        x++;
      }
    N--;
  }
```
È evidente che il numero di cicli occorrenti per completare il programma dipende dalla posizione dell'elemento da cancellare e dal numero di elementi utili nel vettore. Dal momento che, statisticamente, occorrerà un numero di cicli pari ad N/2, possiamo concludere che questo algoritmo ha complessità linearmente crescente con N.

8.4.2. RICERCA DI ELEMENTI IN UN VETTORE

Per cercare un elemento in un vettore, cioè per recuperare il contenuto informativo corrispondente a una determinata chiave, esistono diversi metodi, di cui il più elementare, valido sia per vettori ordinati che non, consiste semplicemente nell'esaminare in sequenza tutti gli elementi del vettore, fino a trovare quello con la chiave desiderata. Per semplificare la programmazione, è opportuno effettuare questa ricerca in senso inverso, partendo cioè dall'ultimo elemento e andando verso il primo:

```
int seqsearch (int v)
  {
    int x = N+1;
    a[0].key=v; a[0].info =-1;
    while (v != a[--x].key);
    return a[x].info;
  }
```

Per evitare che, se l'elemento cercato non esiste, il programma non abbia mai termine, il metodo più semplice consiste nell'inserire la chiave dell'elemento cercato nella posizione 0 del vettore (che non è altrimenti utilizzata), e nel farle corrispondere un contenuto informativo sicuramente diverso da quello di tutti gli altri elementi. Nel nostro esempio abbiamo utilizzato il valore -1, supponendo che tutti gli altri record contengano valori diversi da quest'ultimo. Quindi, l'algoritmo appena presentato ritornerà il contenuto informativo dell'elemento cercato se esso esiste, e il valore -1 in caso contrario.
Per quanto riguarda la complessità di questo algoritmo, vale lo stesso ragionamento fatto a proposito della cancellazione, e cioè che la sua complessità è dell'ordine di N/2.
Infine, osserviamo che questo algoritmo si comporta esattamente nello stesso modo sia che il vettore contenga dati ordinati, sia che essi siano disposti in ordine casuale.
Se il vettore in cui occorre effettuare la ricerca è ordinato, possiamo utilizzare un algoritmo, basato sulla cosiddetta *ricerca binaria* o *dicotomica*, che ha un'efficienza enormemente superiore, soprattutto se il vettore è grande.
Esso si basa sul concetto che se, dato un vettore di elementi ordinati in modo crescente, consideriamo quello centrale, l'elemento cercato sarà nella parte sinistra del vettore se esso è più piccolo di quello centrale, e nella parte destra in caso contrario. A questo punto possiamo ripetere la ricerca nello stesso modo, avendo scartato metà degli elementi del vettore. Il procedimento si ripete finché non si trova l'elemento cercato, o finché la lunghezza del sottovettore individuato non diventa nulla, il che significa che l'elemento cercato non esiste.
L'algoritmo è il seguente:

```
int binsearch (int v)
  {
    int l=1; int r=N; int x;
    while (r>=l)
      {
        x=(l+r)/2;
        if (v < a[x].key) r = x-1; else l = x+1;
        if (v == a[x].key) return a[x].info;
      }
    return -1;
  }
```

Dal momento che ad ogni iterazione le dimensioni del vettore da considerare si dimezzano, la complessità dell'algoritmo è dell'ordine di $\log_2 N$, quindi cresce molto lentamente all'aumentare di N, come si può vedere in figura 8.1.

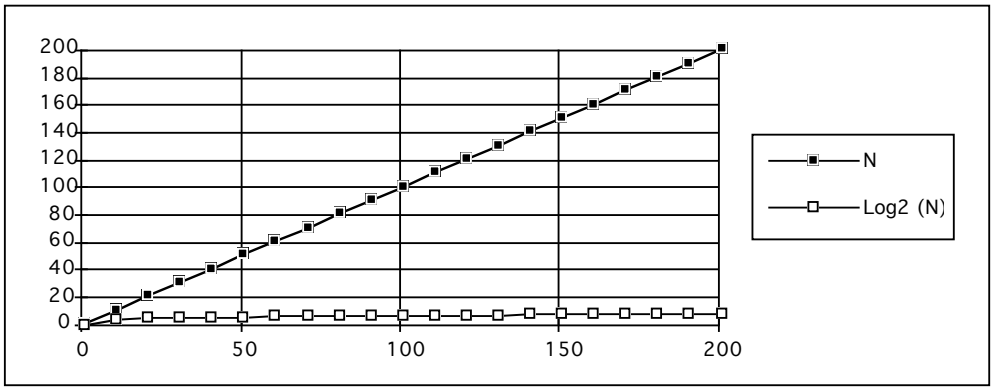


Figura 8.1. - Confronto fra due complessità di calcolo.

8.4.3. ALGORITMI PER L'ORDINAMENTO DI UN VETTORE

Sia dato un vettore V formato da N elementi di un tipo che, in questo e nei successivi esempi, supporremo intero,[35] disposti in ordine del tutto casuale, da ordinare in senso crescente,[36] In questi esempi, la chiave secondo cui viene fatto l'ordinamento è il contenuto stesso degli elementi.
A seconda di come vogliamo costruire un algoritmo che risolva tale problema possiamo decidere di lasciare inalterate oppure di scambiare le posizioni di elementi uguali durante l'operazione cosiddetta di *sorting* del vettore.

8.4.3.1. Algoritmo semplice

Questo è il primo algoritmo risolutivo del problema di ordinamento di un vettore i cui elementi siano per esempio dei numeri interi.
Esso confronta di volta in volta un elemento, a partire dal primo, con tutti i rimanenti del vettore scambiandolo di posto con un eventuale numero minore di questo. Il ciclo termina quando tutti i numeri saranno stati confrontati e conseguentemente ordinati (Fig. 8.2).

---

[35] Il ragionamento non cambia per tipi di dati differenti.

[36] Questa operazione è detta in inglese *sorting*.

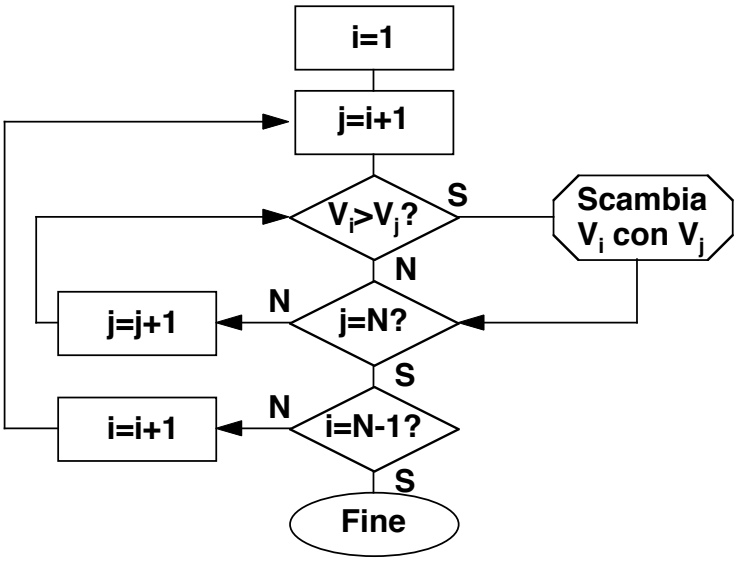


Figura 8.2. - Il primo algoritmo di ordinamento.

Osservando lo schema a blocchi di questo algoritmo, osserviamo che è impossibile prevedere con esattezza il suo tempo di esecuzione perché, mentre il numero di cicli (e quindi di confronti) che devono essere eseguiti è determinabile, e vale $\sum_{i=1}^{n-1} i \approx \frac{n^2}{2}$, il numero di scambi dipende dal grado di disordine degli elementi del vettore. Il numero di scambi è minimo (0 scambi) se il vettore viene presentato già ordinato, e massimo $\left(\sum_{i=1}^{n-1} i\right)$, quando il vettore è ordinato esattamente nel senso opposto a quello desiderato.

Anche se il numero esatto di scambi è difficile da determinare, si può dare un risultato statistico sul valore di questo numero sulla base di un campione di prove, come è mostrato dal grafico di figura 8.3.

In esso si nota un andamento lineare del numero di scambi effettuati in funzione del numero N di elementi considerati.

In conclusione, considerato l'andamento lineare del numero di scambi, possiamo affermare che la complessità di questo algoritmo è dell'ordine di $\frac{N^2}{2}$.

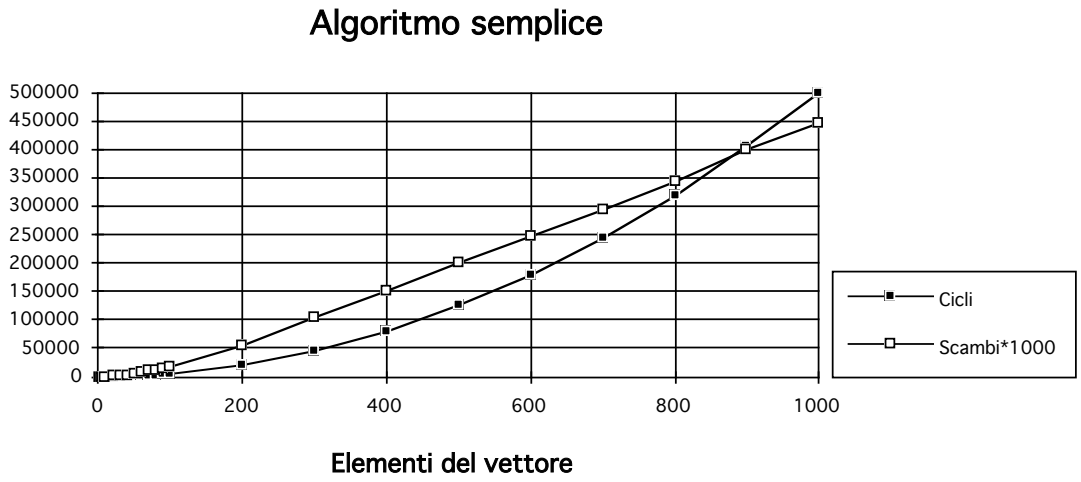


Figura 8.3. - Confronti e scambi per l'algoritmo semplice.

8.4.3.2. *Selection Sort*

L'algoritmo *Selection Sort*, di cui è mostrato qui di seguito il programma in C che lo implementa, offre un mezzo più efficiente per l'ordinamento di un generico vettore.

```
selection (int a[], int N)
  {
    int i, j, min, t;
    for (i=1; i<N; i++)
      {
        min=i;
        for (j=i+1; j<=N; j++)
          if (a[j] < a[min]) min=j;
        t=a[min]; a[min]=a[i]; a[i]=t;
      }
  }
```

Il ciclo più interno dell'algoritmo serve a determinare l'indice del più piccolo numero fra gli (N-i-1) elementi rimasti. Questo elemento viene poi scambiato con l'i-esimo elemento.
Si noti, inoltre, che l'algoritmo effettua un solo scambio ogni volta che il ciclo interno viene eseguito.
L'algoritmo è migliore del precedente perché in genere prevede un numero di scambi minore che non l'altro.

8.4.3.3. *Insertion Sort* e *Bubble Sort*

Insertion Sort e *Bubble Sort* sono due altri semplici algoritmi di ordinamento, di cui vengono qui fornite le implementazioni. *Insertion Sort* è di comprensione immediata, e riproduce l'algoritmo che usiamo quando dobbiamo ordinare un pacco di schede: ognuna di esse viene presa, ed "infilata" al posto che le compete all'interno della parte del pacco già ordinata, spostando in avanti di una posizione tutte le altre.

```
insertion (int a[], int N)
  {
```

```
    int i, j, v;
    for (i=2; i<=N; i++)
      {
        v=a[i]; j=i;
        while (a[j-1] > v)
          { a[j] = a[j-1]; j--; }
          a[j] = v;
      }
  }
```

Bubble Sort è meno intuitivo del precedente: per la sua comprensione, occorre osservare che i due cicli di cui è composto hanno gli indici che variano in senso opposto: quello più esterno viene decrementato ad ogni iterazione, mentre quello più interno viene incrementato. *Bubble Sort* è poco efficiente: il suo maggior vantaggio consiste nella estrema semplicità del codice che lo implementa.

```
bubble (int a[], int N)
  {
    int i, j, t;
    for (i=N; i>=1; i--)
      for (j=2; j<=i; j++)
        if (a[j-1] > a[j])
          { t=a[j-1]; a[j-1]=a[j]; a[j]=t;}
  }
```

8.4.3.4. Efficienza degli algoritmi fondamentali

I tre algoritmi visti in precedenza possono essere confrontati per determinarne l'efficienza. Come si può vedere nella tabella che segue, è impossibile determinare quale di essi sia in assoluto il migliore, perché le loro prestazioni dipendono dal numero e dal tipo dei dati su cui devono operare.

Algoritmo	Confronti	Scambi	Note
SELECTION	$\frac{N^2}{2}$	N	(2)
INSERTION	$\frac{N^2}{4}$	$\frac{N^2}{8}$	(1)
BUBBLE	$\frac{N^2}{2}$	$\frac{N^2}{2}$	

(1) Lineare per vettori "quasi ordinati"
(2) Lineare per vettori con grandi record e piccole chiavi

È importante notare che l'algoritmo *Insertion Sort* diventa di complessità lineare se il vettore è quasi ordinato: questo perché tale algoritmo confronta un elemento con il suo precedente quindi, se il vettore è già ordinato, non si effettuano scambi e il numero di confronti è pari agli elementi del vettore stesso. Ovviamente la differenza nel caso di vettore quasi ordinato è minima e giustifica perciò l'estendibilità dei risultati già trovati anche a questo caso. È chiaro perciò che, dovendo aggiungere un esiguo numero di elementi in coda ad una struttura molto grande e già ordinata, usare l'algoritmo *Selection Sort* costituisce un passo in avanti.

8.4.3.5. *Quicksort*

La complessità quadratica degli algoritmi di ordinamento appena visti ne rende problematica l'applicazione a casi in cui il numero di elementi da ordinare sia grande. La ricerca di algoritmi meno complessi, in cui cioè il tempo di esecuzione cresca meno che quadraticamente con l'aumentare del numero di elementi, ha dato luogo ad algoritmi che richiedono più memoria, ma meno tempo, per la loro esecuzione. Gli esempi più tipici sono *Shell Sort*, *Distribution Counting* e, soprattutto, *Quicksort*. Quest'ultimo è un algoritmo ricorsivo che si basa sull'idea di prendere un elemento a caso nel vettore, e di disporre tutti quelli minori alla sua sinistra, e tutti quelli maggiori alla sua destra; ripetendo ricorsivamente il procedimento per i due sottovettori che si sono così creati, si ottiene alla fine il vettore ordinato. L'algoritmo è molto semplice:

```
quicksort (int a[], int l, int r)
  {
    int i;
    if (r>1)
      {
        i=partition(l, r);
        quicksort(a, l, i-1);
        quicksort(a, i+1, r);
      }
  }
```

Per comprenderne il funzionamento, consideriamo il vettore

```
   4    7    3   22   11    9    6
```

e supponiamo di scegliere come elemento di partizione quello che contiene il numero 11. Per sapere che posto avrà tale numero dobbiamo utilizzare un criterio che porterà tutti gli elementi minori di 11 alla sua sinistra e quelli maggiori alla sua destra. Consideriamo, quindi, l'elemento che deve essere sistemato e, partendo da sinistra, lo confrontiamo con gli altri elementi del vettore: se ne troviamo uno che non rispetta il criterio suddetto interrompiamo la scansione.
Ripetiamo il procedimento partendo dall'elemento più a destra e, non appena troviamo un elemento minore di quello da sistemare lo scambiamo con quello trovato nella prima scansione:

```
   4    7    3    6   11    9   22
```

Questo processo verrà ripetuto sino a che i puntatori usati per esaminare le posizioni a destra e a sinistra non arrivino a puntare allo stesso elemento del vettore. A questo punto gli elementi a destra di 11 sono tutti più grandi e quelli a sinistra di tale numero sono tutti più piccoli: abbiamo realizzato così un ordinamento parziale:

```
   4    7    3    6    9   11   22
```

Il passo seguente dell'algoritmo prevede di ordinare i due sottovettori costituiti dai valori rispettivamente a destra e a sinistra del numero considerato (11 nel nostro esempio), e di ripetere il procedimento finché tutto il vettore non sia stato ordinato.
Una nota importante è rappresentata dal fatto che, al fine di evitare un funzionamento non ottimale dell'algoritmo, in ogni ricorsione si deve partizionare il vettore di partenza in due sottovettori di dimensioni non troppo diverse. Infatti, partizioni

disuguali porterebbero ad un'eccessiva profondità di ricorsione, e quindi, pur restando uguale il tempo di esecuzione, la quantità di memoria occorrente potrebbe superare la capacità del calcolatore.
La parte più difficile da realizzare dell'algoritmo è quindi la procedura *partition*, che deve dividere il vettore in due parti approssimativamente uguali. La sua struttura dipende dalla distribuzione dei dati in ingresso: con dati disposti casualmente, una buona implementazione potrebbe essere la seguente:

```
quicksort (int a[], int l, int r)
  {
    int v, i, j, t;
    if (r>l)
      {
        v=a[r]; i=l-1; j=r;
        for (;;)
          {
            while (a[++i] < v);
            while (a[--j] > v);
            if (i >= j) break;
            t=a[i]; a[i]=a[j]; a[j]=t;
          }
        t=a[i]; a[i]=a[r]; a[r]=t;
        quicksort(a, l, i-1);
        quicksort(a, i+1, r);
      }
  }
```

Quicksort usa in media $2N\ln_2 N$ confronti.

8.5. STRUTTURE A LISTA

In molti casi la memorizzazione dei dati in vettori ordinati comporta problemi dovuti alla lentezza delle operazioni di inserimento e di ricerca. Altri problemi possono sorgere quando i vari elementi hanno dimensioni differenti, e quindi occupano quantità diverse di memoria. Ciò è particolarmente avvertibile se i dati da trattare sono stringhe alfanumeriche.
In questi casi, appaiono più interessanti le strutture cosiddette *a lista* in cui, come è noto, ad ogni dato sono associati uno o più *puntatori* che costituiscono il collegamento con gli elementi contigui. Le figure 8.4, 8.5, 8.6 e 8.7 mostrano i quattro tipi più comunemente usati di liste.

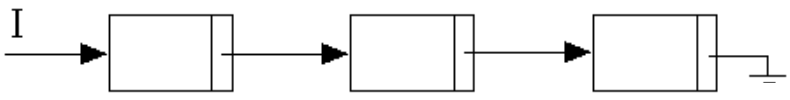


Figura 8.4. - Lista lineare.

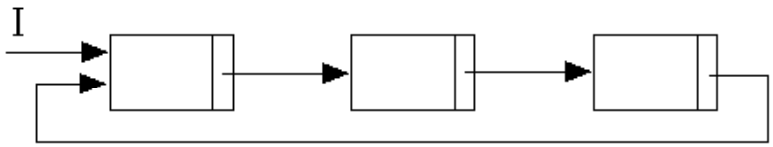


Figura 8.5. - Lista circolare.

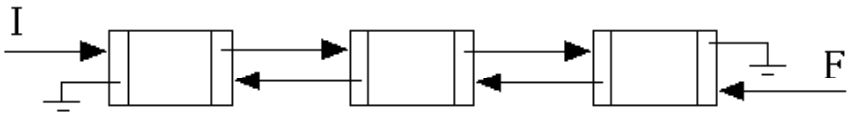


Figura 8.6. - Lista lineare doppia.

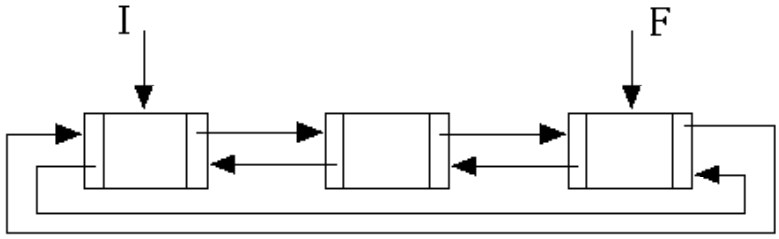


Figura 8.7. - Lista circolare doppia.

È importante anche osservare che l'allocazione della memoria per gli elementi di una lista può avvenire dinamicamente: la memoria necessaria può cioè essere allocata solo al momento dell'inserzione di un nuovo elemento, e resa nuovamente libera quando un elemento viene cancellato.

8.5.1. INSERZIONE IN UNA LISTA LINEARE

Le più comuni operazioni effettuabili sulle liste sono qui di seguito illustrate con una serie di esempi. Tutti i programmi forniti sono commentati al loro interno: la discussione sarà quindi ridotta al minimo indispensabile.
Il primo programma esempio descrive la costruzione di una lista lineare semplice (Fig. 8.4) nella quale vengono memorizzati una serie di numeri letti da tastiera. A differenza dell'utilizzo di array, nella lista lineare viene allocato e utilizzato esattamente lo spazio necessario alla memorizzazione dei numeri introdotti dall'utente, la cui quantità non è nota a priori.
La struttura dell'elemento base della lista è ridotta all'essenziale: un intero per la memorizzazione del dato ed un puntatore all'elemento successivo.

```
struct node
{
int    iNum;
struct node *psnNext;
}

main()
{
int    iX;
struct node *psnStart, *psn;

psnStart = NULL;
printf("Introdurre una sequenza di interi,"
       "seguita da #:\n");
while (scanf("%d", &iX) > 0)
   {
   psn = psnStart;
   psnStart = (struct node *) malloc(sizeof(struct node));
   if (psnStart == NULL)
      {
      printf("Memoria insufficiente");
      exit(1);
      }
   psnStart->iNum = iX;
   psnStart->psnNext = psn;
   }
printf("\nIn ordine inverso, sono stati "
               "letti i numeri seguenti:\n");
for (psn=psnStart ; psn!=NULL ; psn=psn->psnNext)
   printf("%5d\n", psn->iNum);
}
```

8.5.2. LISTA LINEARE CON SENTINELLA

Negli esempi che seguono, implementando una lista lineare semplice, viene "sprecato" un elemento, detto sentinella, il cui unico scopo è quello di semplificare le operazioni di creazione, inserimento e ricerca.

8.5.2.1. Struttura

```
typedef   struct  node
       {
       int    iNum;
       struct node *psnNext;
       } NODOLISTA, *PTRNODOLST;
```

8.5.2.2. Creazione

```
PTRNODOLST  creanodo()
{
PTRNODOLST  psn;

psn=(PTRNODOLST) malloc(sizeof(NODOLISTA));
if (psn == NULL)
```

```
   {
   printf("Memoria insufficiente \n");
   exit (1);
   }
return psn;
}


PTRNODOLST  psn, psnStart, psnEnd;


psnStart = psnEnd = creanodo();
```

Nota: la lista è vuota, non contiene nessun dato, il nodo creato è la sentinella

8.5.2.3. Inserimento alle estremità della lista

Il codice seguente effettua un inserimento in cima alla lista di un nodo contenente il valore iX.

```
psn = psnStart;
psnStart = creanodo();
psnStart->psnNext = psn;
psnStart->iNum = iX;
```

Il codice seguente effettua invece un inserimento in fondo alla lista di un nodo contenente il valore iX.

```
psn = psnEnd;
psnEnd= creanodo();
psn->psnNext = psnEnd;
psn->iNum = iX;
```

8.5.2.4. Ricerca

Il principale svantaggio delle liste rispetto ai vettori è costituito dal fatto che non è possibile indirizzare direttamente i vari elementi, dal momento che le loro posizioni in memoria non sono note: per trovare un elemento, occorre scandire la lista dall'inizio. Il codice seguente effettua un ricerca lungo la lista con sentinella dell'elemento contenente il valore iX.

```
psnEnd->iNum = iX;
psn = psnStart;
while (psn->iNum != iX)
   psn = psn->psnNext;
if (psn == psnEnd)
   {/*  non trovato  */}
else
   {/*  trovato nel nodo puntato da psn  */}
```

Se la lista è ordinata in ordine crescente di iX l'algoritmo invece è

```
psnEnd->iNum = iX;
psn = psnStart;
while (psn->iNum < iX)
   psn = psn->psnNext;
if (psn->iNum == iX && psn != psnEnd)
```

```
   {/*  trovato nel nodo puntato da psn  */}
else
   {/*  non trovato  */}
```

8.5.2.5. Inserimento

Quando si inserisce un nuovo nodo in una lista, occorre riposizionare opportunamente i puntatori:

```
/*  inserimento di un nodo che conterrà il valore iX
     esattamente prima del nodo puntato da psn  */
psnQ=creanodo();
if (psn == psnEnd)
   psnEnd = psnQ;
else
   *psnQ = *psn;         /*  copia l'intero nodo  */
psn->psnNext = psnQ;
psn->iNum = iX;
```

8.5.2.6. Cancellazione

```
/*  cancellazione del nodo puntato da psn  */
psnQ = psn->psnNext;
if (psnQ ==psnEnd)
   psnEnd = psn;
else
   *psn = *psnQ;
free (psnQ);
```

8.5.3. UN ESEMPIO: IMPLEMENTAZIONE DI UNO STACK MEDIANTE LISTE LINEARI

Questo algoritmo implementa uno stack mediante un vettore. Ricordiamo che lo stack è un particolare tipo di memoria, detta anche LIFO (*Last In - First Out*) da cui i dati possono essere prelevati solo in ordine inverso rispetto a quello con cui sono stati inseriti.

```
#define  STSIZE 1000

int  aiStack[STSIZE], iStptr=0;
/* aiStack[iStptr] è la prima locazione libera dello stack */

void  push(int  iX)
{
if (iStptr == STSIZE)
   errore("Overflow dello stack");
else
   aiStack[iStptr++] = iX;
}

int  pop(void)
{
if (iStptr == 0)
   errore("Impossibile prelevare!  stack vuoto");
```

```
else
   return(aiStack[--iStptr]);
}
```

Il principale difetto del programma appena visto deriva dal fatto che il vettore deve essere dimensionato in anticipo, e quindi occupa memoria anche se non è utilizzato. Utilizzando una lista lineare, a prezzo di un codice più complicato e di un calo di efficienza, eliminiamo l'inconveniente appena citato. L'algoritmo relativo è:

```
typedef   struct  node
        {
        int     iNum;
        struct node *psnNext;
        } NODOLISTA, *PTRNODOLST;

PTRNODOLST  psnStart, psn;

void  push(int iX)
{
psn = (PTRNODOLST) malloc(sizeof(NODOLISTA));
if (psn == NULL)
   errore("Overflow dello stack");
else
   {
   psn->psnNext = psnStart;
   psn->iNum = iX;                        /* ins. dato */
   psnStart = psn;
   }
}

int  pop(void)
{
int     iX;
if (psnStart == NULL)
   errore("Impossibile prelevare!  stack vuoto");
else
   {
   iX = psnStart->iNum;                         /* prel. dato */
   psn = psnStart;
   psnStart = psnStart->psnNext;
   free(psn);
   return(iX);
   }
```

8.5.4. LISTE CIRCOLARI DOPPIE CON SENTINELLA

Le liste doppie offrono il vantaggio, rispetto a quelle semplici, di poter essere percorse in entrambi i sensi. Ciò può risultare utile in diverse circostanze. Ovviamente, ogni elemento di una lista doppia dovrà avere due puntatori: uno che punti all'elemento seguente, e uno che punti a quello precedente.
Anche qui viene usato un elemento, non contenente dati, come sentinella.

```
typedef  struct node
{
```

```
int    iNum;
struct node    *psnS, psnD;
}  NODOLISTA, *PTRNODOLST;

PTRNODOLST  psnStart;
```

8.5.4.1. Creazione lista vuota

Il programma che segue crea una lista circolare doppia con sentinella “vuota”, che contiene cioè la sola sentinella. La struttura creata è visibile in figura 8.8.

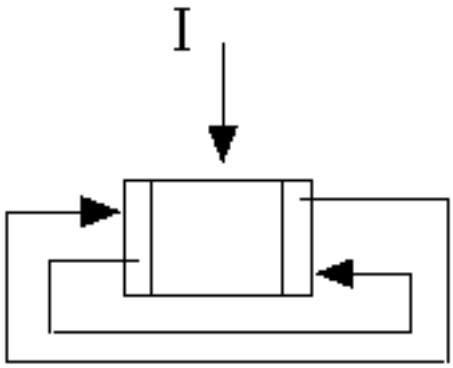


Figura 8.8. - Una lista circolare doppia vuota.

```
psnStart = (PTRNODOLST) malloc (sizeof(NODOLISTA));
if (psnStart == NULL)
   {
   errore("Memoria Insufficiente");
   exit (1);
   }
psnStart->psnD = psnStart->psnS = psnStart;
```

8.5.4.2. Inserimento

Le porzioni di codice che seguono permettono l’inserimento di un elemento contenente il dato iX a sinistra del puntatore di inizio lista (sostituibile con il puntatore ad un elemento qualsiasi). Nella seconda parte, equivalente alla prima, è utilizzato un doppio riferimento indiretto.

```
PTRNODOLST  psn, psnQ;
psn = (PTRNODOLST) malloc (sizeof(NODOLISTA));
if (psn == NULL)
   {
   errore("Memoria Insufficiente");
   exit (1);  /* non e' necessario uscire */
   }

psn->iNum = iX;
```

```
psnQ = psnStart->psnS;
psnStart->psnS = psn;
psn->psnD = psnStart;
psnQ->psnD = psn;
psn->psnS = psnQ;


PTRNODOLST  psn, psnQ;
psnQ = (PTRNODOLST) malloc (sizeof(NODOLISTA));
if (psnQ == NULL)
{
  errore("Memoria Insufficiente");
  exit (1);
  }
psnQ->psnS = psn->psnS;
psnQ->iNum = iX;
psnQ->psnD = psn;
psn->psnS->psnD = psnQ;  /*  !  */
psn->psnS = psnQ;
```

8.5.4.3. Cancellazione del nodo puntato da psn

```
psn->psnS->psnD = psn->psnD;
psn->psnD->psnS = psn->psnS;
free(psn);
```

Attenzione: occorre verificare che `psn` sia diverso da `psnStart` o da qualsiasi altro puntatore utile (ad esempio il nodo corrente).

8.6. ALBERI BINARI

Le strutture ad albero permettono ricerche più rapide rispetto alle liste.
Particolarmente importanti nel nostro caso sono gli alberi binari, in cui ogni nodo è caratterizzato dal fatto di avere al massimo due figli, come mostrato in figura 8.9.
Alcune definizioni e proprietà degli alberi binari sono elencate qui di seguito:

- esiste un solo genitore per ogni nodo tranne che per il nodo radice;
- due puntatori diversi si riferiscono a due nodi diversi: se così non fosse, si tratterebbe di un grafo;
- negli alberi binari possono esistere per ogni nodo 0, 1 o 2 figli;
- si definisce altezza di un albero il livello del nodo più profondo;
- la struttura dell'albero è intrinsecamente ricorsiva: ogni sottoalbero è a sua volta un albero.

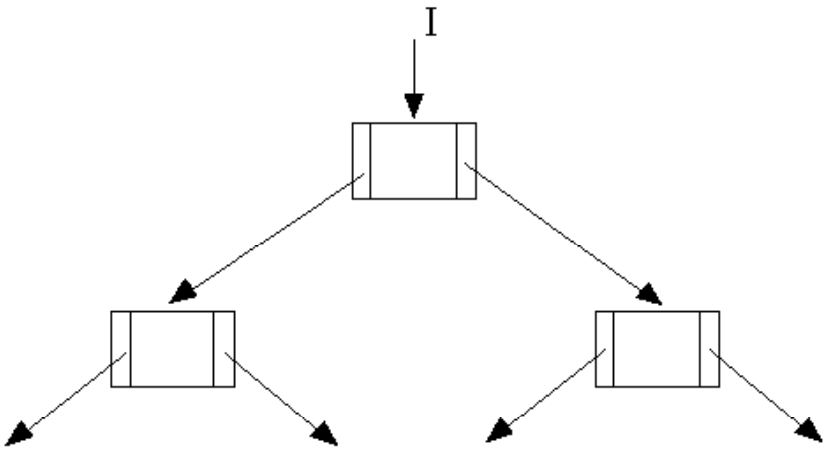


Figura 8.9. - Un albero binario.

8.6.1. ESTRAZIONE DI ELEMENTI DA UN ALBERO BINARIO

Per estrarre gli elementi di un albero binario, occorre procedere ricorsivamente secondo questo algoritmo:

- se l'albero è vuoto:
  - non fare nulla;
- altrimenti:
  - estrarre il sottoalbero di sinistra;
  - estrarre il nodo radice;
  - estrarre il sottoalbero di destra.

L'ordine di estrazione degli elementi dipende ovviamente dal procedimento che è stato utilizzato per costruire l'albero.

8.6.2. RICERCA DI UN ELEMENTO IN UN ALBERO BINARIO ORDINATO

L'algoritmo che segue serve per cercare un elemento all'interno di un albero binario ordinato. Anch'esso è ricorsivo:

- se l'albero è vuoto:
  - chiave non trovata;
- altrimenti:
  - se la chiave è inferiore al nodo:
    applicare l'algoritmo nel sottoalbero sinistro;
  - se la chiave è superiore al nodo:
    applicare l'algoritmo nel sottoalbero destro;
  - altrimenti:
    chiave trovata: è nel nodo in esame.

Come esempio, vediamo un programma che fornisce un conteggio relativo alla frequenza di parole lette in un file di testo.

Il programma costruisce un albero di ricerca binario, compie ricerche all'interno di esso e ne stampa il contenuto.
È possibile ricercare una data parola all'interno dell'albero, per sapere quante volte essa ricorre.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>


typedef struct node
{
char   *pcWord;
int       iCount;
struct node *psnLeft, *psnRight;
}            NODOALB, *PTRNODOALB;


PTRNODOALB  buildtree (PTRNODOALB);
int         fillbuffer (FILE *);
PTRNODOALB  addnode(PTRNODOALB);
char      *my_alloc (int);
void      printtree (PTRNODOALB);
PTRNODOALB  search (PTRNODOALB);


char acBuffer[100];

main()
{
PTRNODOALB  psnRoot, psnPtr;

psnRoot = NULL;
psnRoot = buildtree(psnRoot);
printf("\nDistribuzione di frequenza:\n\n");
printtree(psnRoot);

while (
printf("\n\nIntrodurre una parola o &&& per terminare: "),
fillbuffer(stdin),
strcmp(acBuffer, "&&&")
            ) /* &&& segnale di terminazione */
  {
  psnPtr = search(psnRoot);
  if (psnPtr == NULL)
    printf("Non riscontrata.\n");
  else
    printf("Numero di occorrenze: %d\n",
                    psnPtr->iCount);
  }
}

PTRNODOALB  buildtree (PTRNODOALB  psnRoot){
char acFilname[40];
FILE *fp;
int    iIndice;
```

```
printf("File di input: ");
scanf("%s", acFilname);
fp = fopen(acFilname, "r");
if (fp == NULL)
   {
   printf("File non disponibile");
   exit(1);
   }
while (iIndice = fillbuffer(fp))      /* cioe' iIndice != 0 */
   if (iIndice > 0)
     psnRoot = addnode(psnRoot);
fclose (fp);
return psnRoot;
}

int  fillbuffer (FILE *fp){
char *pcDest, *pcSource, c;

if (fscanf(fp, "%s", acBuffer) <= 0)
   return 0;
if (strcmp(acBuffer, "&&&") == 0)
   return 1;
pcDest = pcSource = acBuffer;
while (c = *pcSource++)            /* cioe' while c != '\0' */
   {
   c = toupper(c);
   if (isalpha(c))
     *pcDest++ = c;
   }
*pcDest = '\0';
return (pcDest > acBuffer ? 1 : -1);       /* -1: stringa vuota */
}
PTRNODOALB  addnode (PTRNODOALB psn)
{
int  iIndice;

if (psn == NULL)
   {
   psn = (PTRNODOALB) my_alloc(sizeof(NODOALB));
   psn->pcWord = my_alloc(strlen(acBuffer) + 1);
   strcpy(psn->pcWord, acBuffer);
   psn->iCount = 1;
   psn->psnLeft = psn->psnRight = NULL;
   }
else
   {
   iIndice = strcmp(acBuffer, psn->pcWord);
   if (iIndice < 0)
     psn->psnLeft = addnode(psn->psnLeft);
   else if (iIndice > 0)
        psn->psnRight = addnode(psn->psnRight);
     else
       psn->iCount ++;
   }
return psn;
}
```

```
char  *my_alloc (int  iN)
{
char *pc;

pc = (char *) malloc(iN);
if (pc == NULL)
   {
   printf("Memoria insufficiente\n");
   exit(1);
   }
return pc;
}



void  printtree(PTRNODOALB  psn)
{
if (psn != NULL)
   {
   printtree(psn->psnLeft);
   printf("%5d %s\n", psn->iCount, psn->pcWord);
   printtree(psn->psnRight);
   }
}



PTRNODOALB search(PTRNODOALB  psn)
{
int  iIndice;

if (psn == NULL)
   return NULL;
iIndice = strcmp(acBuffer, psn->pcWord);
if (iIndice <0)
   return(search(psn->psnLeft));
else if (iIndice >0)
     return(search(psn->psnRight));
   else
     return (psn);
}
```

8.6.3. ALBERI BINARI BILANCIATI

Si definisce *albero binario bilanciato in altezza* un albero nel quale ogni nodo possiede due sottoalberi la cui altezza differisce al massimo di 1.
Si definisce *albero binario perfettamente bilanciato* un albero nel quale ogni nodo possiede due sottoalberi il cui numero di nodi differisce al massimo di 1.
Se un albero è perfettamente bilanciato è anche bilanciato in altezza, mentre il contrario non è necessariamente vero.
Le strutture ad albero viste finora non prevedevano invece alcun limite alla lunghezza dei diversi rami. Osserviamo che, con riferimento alla figura 8.10, gli algoritmi finora

presentati operano ugualmente sia su alberi bilanciati (come quello a sinistra), che su alberi sbilanciati (come quello a destra), ma che un albero completamente sbilanciato è in realtà una lista, in quanto la sua struttura è la stessa. Con alberi di questo tipo si perdono i vantaggi già menzionati a proposito della velocità della ricerca. Su un albero completamente sbilanciato infatti una ricerca dicotomica degenera in una sequenziale. La velocità di ricerca è proporzionale alla bilanciatura dell'albero.

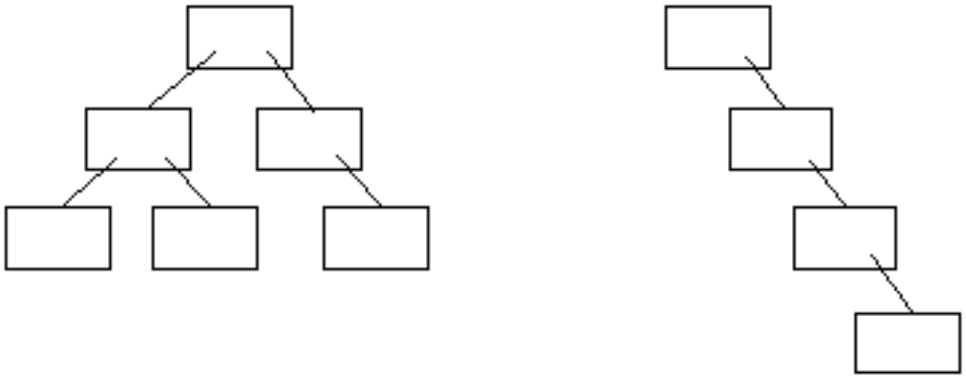


Figura 8.10. - Un albero bilanciato e uno sbilanciato.

8.6.4. COSTRUZIONE DI UN ALBERO BINARIO PERFETTAMENTE BILANCIATO

Il programma che segue crea un albero perfettamente bilanciato a partire da un insieme di dati a condizione che:

- i dati siano in ordine;
- se ne conosca il numero.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct node
{
int       iNum;
struct node *psnLeft, *psnRight;
}            NODOALB, *PTRNODOALB;

PTRNODOALB  pbtree (int  iN);
void      printtree(PTRNODOALB  psn);

FILE *fp;



void  printtree(PTRNODOALB  psn)
```

```
{
if (psn != NULL)
   {
   printtree(psn->psnLeft);
   printf("%5d \n", psn->iNum);
   printtree(psn->psnRight);
   }
}
main()
{
int    iN;
PTRNODOALB  psnRoot;

fp = fopen("DATI", "r");
if (!fp)
   {
   printf("file non apribile!");
   exit(1);
   }
fscanf(fp, "%d", &iN);
psnRoot = pbtree(iN);   /* iN è il numero di volte che la funzione
ricorsiva
            pbtree verrà eseguita */
fclose(fp);
printtree(psnRoot);
}


PTRNODOALB  pbtree (int  iN)
{
int    iLeft, iRight;
PTRNODOALB  psn;

if (iN == 0)
   return NULL;
iLeft = (iN-1) >> 1;
iRight = iN - iLeft - 1;
psn = (PTRNODOALB)  malloc (sizeof(NODOALB));
psn->psnLeft = pbtree(iLeft);
fscanf(fp, "%d", & psn->iNum);
psn->psnRight = pbtree(iRight);
return psn;
}
```

8.6.5. RICERCA NON RICORSIVA

Mediante l'utilizzo di puntatori a puntatori a nodi si può realizzare una funzione di ricerca non ricorsiva, utile quando la disponibilità di memoria per il programma è limitata:

```
PTRPTRNODO  search(PTRPTRNODO  ppsn, int  iX)
{
PTRPTRNODO  ppsnQ;

ppsnQ = ppsn;
```

```
while (*ppsnQ != NULL  &&  iX != (*ppsnQ)->iNum)
   if (iX<(*ppsnQ)->iNum)
     ppsnQ = &(*ppsnQ)->psnLeft;
   else
     ppsnQ = &(*ppsnQ)->psnRight;
return ppsnQ;
}
```

La funzione appena presentata prevede i seguenti casi particolari:

- se l'albero è vuoto, restituisce l'indirizzo della radice, che conterrà NULL;
- se l'albero è costituito dal solo nodo radice contenente il valore cercato, restituisce l'indirizzo della radice, che conterrà il valore cercato;
- se l'albero è costituito dal solo nodo radice contenente un valore diverso da quello cercato, restituisce l'indirizzo di uno dei puntatori della radice (a seconda che il valore presente sia minore o maggiore di quello cercato), che conterrà il valore NULL.

Usando la funzione `search` non ricorsiva, è possibile costruire una funzione per l'inserimento di nuovi nodi:

```
void  insert(PTRPTRNODO  ppsn, int  iX)
{
PTRPTRNODO  ppsnQ;

ppsnQ = search(ppsn, iX);
if (*ppsnQ == NULL)
   {
   *ppsnQ = (PTRNODO)  malloc(sizeof(NODO));
   if (*ppsnQ == NULL)
     {
     printf("Memoria Insufficiente \n");
     exit (1);
     }
   else
     {
     (*ppsnQ)->iNum = iX;
     (*ppsnQ)->psnLeft = (*ppsnQ)->psnRight = NULL;
     }
   }
}
```

8.6.6. CANCELLAZIONE DI UN SOTTOALBERO

La cancellazione di un intero sottoalbero, a partire da un determinato nodo, si effettua mediante il seguente algoritmo:

```
void    deltree(PTRPTRNODO  ppsn){
if (*ppsn != NULL)
   {
   deltree(&(*ppsn)->psnLeft);
   deltree(&(*ppsn)->psnRight);
```

```
   free(*ppsn);
   *ppsn = NULL;
   }
}
```

8.6.7. CANCELLAZIONE DI NODI DA ALBERI BINARI

In un archivio strutturato ad albero è pratica comune, quando si deve cancellare un nodo, marcarlo in modo particolare, senza effettivamente "toglierlo" dalla struttura. Ciò semplifica l'operazione, ma può non risultare efficiente se le cancellazioni e gli inserimenti sono molti. In ogni caso, non eliminare fisicamente i dati cancellati comporta uno spreco di memoria.
Il problema nel cancellare un nodo è che non è sufficiente liberare la memoria occupata: bisogna anche ricostruire i puntatori dei nodi contigui in modo tale che non vadano perse le caratteristiche di ordinamento dell'albero.
Nel caso in cui il nodo da cancellare non abbia figli o ne abbia uno solo, si può usare la funzione

```
void   delnode(PTRPTRNODO  ppsn){
PTRPTRNODO  ppsnQQ;
PTRNODO     psn, psnQ;
if (*ppsn != NULL)
   {
   psn = *ppsn;
   if (psn->psnRight == NULL)
      {
      *ppsn = psn->psnLeft;
      free(psn);
      }
   else if (psn->psnLeft == NULL)
      {
      *ppsn = psn->psnRight;
      free(psn);
      }
   else ............
```

In generale invece la funzione da usare è

```
void   delnode(PTRPTRNODO  ppsn){
PTRPTRNODO  ppsnQQ;
PTRNODO     psn, psnQ;

if (*ppsn != NULL)
   {
   psn = *ppsn;
   if (psn->psnRight == NULL)
      {
      *ppsn = psn->psnLeft;
      free(psn);
      }
   else if (psn->psnLeft == NULL)
      {
      *ppsn = psn->psnRight;
      free(psn);
```

```
    }
  else
    {
    ppsnQQ = &psn->psnLeft;
    while ((*ppsnQQ)->psnRight != NULL)
      ppsnQQ = &(*ppsnQQ)->psnRight;
    psnQ = *ppsnQQ;
         /* elemento più dx del sottoalbero di sx */
           /* cioè il più grande tra gli inferiori */
    *ppsnQQ = psnQ->psnLeft;
      /* ricollega il sottoalbero sx di psnQ se esiste */
    psn->iNum = psnQ->iNum;
         /* sostituisco solo il dato non il nodo */
    free(psnQ);
    }
  }
}
```

8.7. PROBLEMI ED ESERCIZI

Problema 8.1.*
Dovendo effettuare 20000 ricerche in un vettore di 20000 elementi (non ordinati) quale fra queste soluzioni sceglieresti?

a. userei un algoritmo di ricerca sequenziale;
b. userei un algoritmo di ricerca binaria;
c. prima ordinerei il vettore usando *quicksort*, e poi userei un algoritmo di ricerca binaria.

Problema 8.2.*
Dovendo effettuare un numero molto limitato di inserzioni (tipicamente una al giorno) in un vettore ordinato di grandi dimensioni quale fra queste soluzioni sceglieresti?

a. inserirei ogni elemento già al posto giusto;
b. inserirei gli elementi in fondo e poi ordinerei il vettore;
c. è indifferente.

Problema 8.3.*
Dovendo effettuare una unica ricerca in un vettore di 3 elementi (ordinati) quale fra queste soluzioni sceglieresti?

a. userei un algoritmo di ricerca sequenziale;
b. userei un algoritmo di ricerca binaria.

Problema 8.4.
È possibile applicare la ricerca binaria a file non ordinati?

Problema 8.5.
È possibile applicare la ricerca sequenziale a file non ordinati?

Problema 8.6.
In un vettore già ordinato, l'inserzione di un nuovo elemento implica una complessità
a. costante;
b. linearmente crescente;
c. quadratica.

Problema 8.7.*
La complessità di un algoritmo è in genere proporzionale al numero di cicli che esegue il programma che lo implementa?

Problema 8.8.*
La complessità di un algoritmo è in genere proporzionale al numero di righe di cui è composto il programma che lo implementa?

Problema 8.9.*
La sola conoscenza della complessità di un algoritmo permette di valutare il suo tempo di esecuzione?

Problema 8.10.
Quale dei tre metodi elementari di ordinamento è più veloce per un vettore già ordinato?
a. *selection sort*;
b. *insertion sort*;
c. *bubble sort*.

Problema 8.11.
Quale dei tre metodi elementari di ordinamento è più veloce per un vettore ordinato in senso inverso?
a. *selection sort*;
b. *insertion sort*;
c. *bubble sort*.

Problema 8.12.
Quicksort è un algoritmo ricorsivo?

Problema 8.13.*
Se su un certo calcolatore l'algoritmo A impiega minor tempo dell'algoritmo B, si può concludere che l'algoritmo A è meno complesso dell'algoritmo B?

Problema 8.14.
Di quanti puntatori ha bisogno un elemento di una lista circolare doppia?
a. 1;
b. 2;
c. 4.

Problema 8.15.*
È sempre possibile effettuare una ricerca sequenziale su una lista circolare semplice?

Problema 8.16.*
È sempre possibile effettuare una ricerca dicotomica su un albero anche se i suoi elementi non sono ordinati?
a. sì;
b. no;
c. sì, se l'albero è piccolo.

Problema 8.17.*
È sempre possibile scandire un albero anche se i suoi elementi non sono ordinati?
a. sì;
b. no;
c. sì, ma si rischia di perdere gli elementi fuori ordine.

Problema 8.18.
È sempre possibile trasformare una lista circolare doppia in un albero binario?

Problema 8.19.
In un albero perfettamente bilanciato è possibile cancellare qualsiasi nodo?
a. sì;
b. no;
c. tutti tranne il nodo radice.

Problema 8.20.
Quale di queste affermazioni, riferite ad una lista circolare doppia, è falsa?
a. può contenere le stesse informazioni di una lista lineare semplice;
b. contiene sempre le stesse informazioni di una lista lineare semplice con lo stesso numero di nodi;
c. può essere scandita nello stesso modo di una lista circolare semplice.

Problema 8.21.
In un albero perfettamente bilanciato è possibile inserire un nodo in qualsiasi punto?
a. sì;
b. no;
c. in tutti i punti tranne che al posto del nodo radice.

Problema 8.22.*
Quale di queste affermazioni è falsa?
a. un albero perfettamente bilanciato è sempre bilanciato in altezza;
b. un albero bilanciato in altezza può non essere perfettamente bilanciato;
c. un albero bilanciato in altezza tra tutti i possibili alberi è quello che garantisce la ricerca più efficente.

Problema 8.23.*
Di quanti puntatori ha bisogno ogni elemento di un albero binario bilanciato?

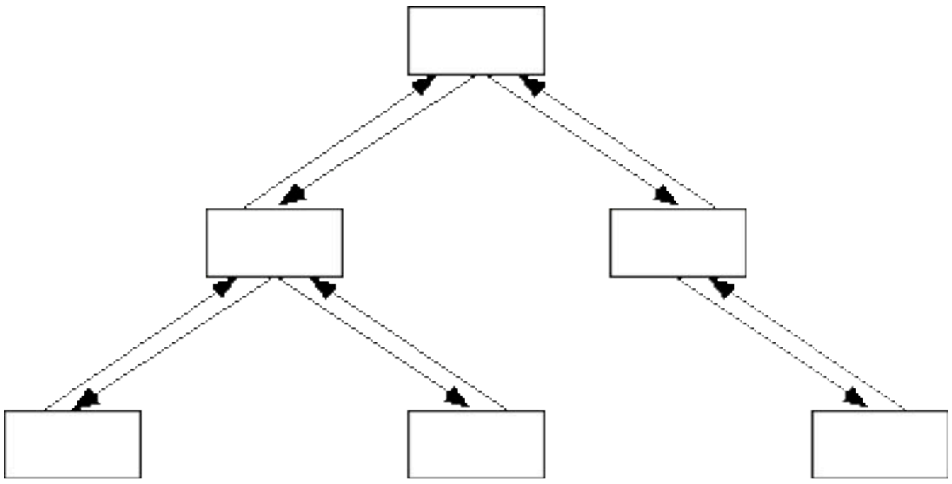
a. 1;
b. 2;
c. 4.

Problema 8.24.*
Quando è conveniente utilizzare un albero binario al posto di una lista per memorizzare dei dati ordinati?

a. quando i dati sono facilmente separabili in coppie;
b. quando sono necessarie numerose ricerche sulla chiave di ordinamento;
c. quando i dati sono equidistribuiti sui valori della chiave di ordinamento.

Problema 8.25.
Per realizzare l'albero della figura è corretta la struttura riportata qui di seguito? Motivare la risposta.

```
struct  nodo
{
int  iInfo;
struct  nodo   *a, *b, *c;
};
```



BIBLIOGRAFIA

La bibliografia sull'argomento trattato in questo volume è sterminata, e ogni tentativo di condensarla in poche pagine costringerebbe fatalmente ad omettere pubblicazioni di importanza a volte anche fondamentale. Inoltre, la costante apparizione di nuove opere rende assolutamente impossibile la compilazione di una lista realmente aggiornata.
Dal momento però che questo volume tratta argomenti di base, si è preferito citare solo i volumi che sono stati consultati più di frequente durante la sua stesura.
La bibliografia è stata suddivisa per capitoli, ma va tenuto presente che diverse opere sono di carattere generale, e contengono quindi la trattazione di argomenti pertinenti a diverse parti del presente volume.

OPERE DI CONSULTAZIONE GENERALE

- A. V. Aho, J. D. Ullmann, *Fondamenti di Informatica*, Zanichelli Editore S.p.A., Bologna, 1994.
- P. Della Vigna, C. Ghezzi, R. Morpurgo, *Fondamenti di Informatica*, CLUP, Milano, 1990.
- F. De Paoli, D. Mandrioli, *Fondamenti di Informatica*, McGraw-Hill Libri Italia S.r.l., Milano, 1995.
- C. Ghezzi, D. Mandrioli, *Informatica Teorica*, Città Studi, Milano, 1994.
- G. Guida, *Fondamenti di Informatica: Algoritmi, Programmi, Sistemi di Elaborazione*, Masson Editoriale ESA, Milano, 1996.

CAPITOLO 1

- J. L. Hennessy, D. A. Patterson, *Computer Architecture: a Quantitative Approach*, Morgan-Kaufmann, San Mateo, CA, 1990.
- Z. Kohavi, *Switching and Finite Automata Theory*, Tata McGraw-Hill Publishing Co. Ltd., Bombay-New Delhi, 1970.
- L. A. Leventhal, *Introduction to Microprocessors: Software, Hardware, Programming*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978.
- M. Sami, R. Negrini, *Calcolatori Elettronici*, CLUP, Milano, 1986.
- A. S. Tanenbaum, *Structured Computer Organization*, 3rd ed., Prentice-Hall, Englewood Cliffs, NJ, 1990.

CAPITOLO 2

- M. Boni, *Informatica*, Apogeo Editrice di Informatica, Milano, 1993.

CAPITOLO 3

- F. Tisato, R. Zicari, *Sistemi Operativi: Architettura e Progetto*, Città Studi, Milano, 1991.

CAPITOLO 4

- F. Halsall, *Sistemi di comunicazione e reti di computer*, Masson - Addison-Wesley, Milano, 1987.

CAPITOLO 5

- Apple Computer, Inc., *Inside Macintosh*, Addison-Wesley Publishing Co., Reading, MA, 1992.

CAPITOLO 6

- P. Seyer, *Understanding Hypertext: Concepts and Applications*, Windcrest Books, Blue Ridge Summit, PA, 1991.

- N. Woodhead, *Hypertext and Hypermedia*; Theory and Applications, Addison Wesley Publishing Co., Reading, MA, 1991.

CAPITOLO 7

- A. E. Stanley, *HyperTalk & HyperText*, Apogeo, Milano, 1991.
- D. Winkler, S. Knaster, *Cooking with HyperTalk 2.0*, Bantam Books, New York, NY, 1990.

CAPITOLO 8

- G. Di Battista, F. Vargiu, *Dal Linguaggio Pascal al linguaggio C*, Masson Editoriale ESA, Milano, 1994.
- N. Graham, *Learning C++*, McGraw-Hill Inc., New York, NY, 1991.
- D. E. Knuth, *The Art of Computer Programming*, Addison Wesley, Reading, MA, 1973.
- U. Manber, *Introduction to Algorithms: A Creative Approach*, Addison Wesley Publishing Co., Reading, MA, 1989.
- R. Sedgewick, *Algorithms in C*, Addison Wesley Publishing Co., Reading, MA, 1990.

APPENDICE

SOLUZIONE DI ESERCIZI SELEZIONATI

Vengono qui proposte le soluzioni di alcuni dei problemi ed esercizi presentati nel corso del volume. Non è stata data la soluzione di quei problemi per cui essa è facilmente verificabile o reperibile all'interno del testo.

A.1. ESERCIZI DEL CAPITOLO 1.

Soluzione 1.4: non è decidibile: si tratta di una macchina sequenziale, e non sono date le condizioni iniziali.

Soluzione 1.6: la 1, la 2 e la 3. La 3 però non è minima.

Soluzione 1.8: $Z = Y$.

Soluzione 1.9: falso: servono per sintetizzarle.

Soluzione 1.12: considerando che due piedini servono per alimentazione e massa, 8 per i dati e 3 per le linee di controllo, ne restano 11 per l'indirizzamento. La capacità della RAM è quindi di $2^{11} = 2048$ byte.

Soluzione 1.16: $F = X_1X_2X_3X_4 + X_1\overline{X}_2X_3X_4 + X_1X_2X_3\overline{X}_4 + X_1\overline{X}_2X_3\overline{X}_4$

A.2. ESERCIZI DEL CAPITOLO 2.

Soluzione 2.1:
a. 00001010;
b. 11010001;
c. 01111110;
d. 10110010;
e. 00000000.

Soluzione 2.2:
a. 0000000010101010;
b. 1110110110011000;
c. 0000010011110001;
d. 1110000110000000;
e. 0000000000000000.

Soluzione 2.3:
a. 0140;
b. C310;
c. 3A1B;
d. FF01.

Soluzione 2.4:
a. 255;
b. 30120;
c. -28342;
d. 29365.

A.3. ESERCIZI DEL CAPITOLO 3.

Soluzione 3.1:

```
program Routine;

  var
  stack:array[0..100] of byte;
  pin:byte;
  car:byte;

  Procedure interrupt
    begin
      stack[pin]:=car;
      pin:=pin+1;
      if pin=101 then pin:=0
    end;

begin
  pin:=0
end.
```

Soluzione 3.2: asincrono.

Soluzione 3.3: si tratta di interruzioni esterne, e come tali sono asincrone in quanto il calcolatore non è in alcun modo in grado di determinare l'istante nel quale esse arriveranno.

Soluzione 3.4: è sincrona in quanto interna ad un programma. L'istante della sua esecuzione è quindi prevedibile.

Soluzione 3.5: dall'hardware. In alcuni casi il software può modificarlo, ma il processo di salto alla routine di gestione dell'interruzione è comunque gestito completamente dall'hardware.

Soluzione 3.6: l'indirizzo a cui tornare dopo che l'interruzione è stata gestita. Gli altri parametri significativi del calcolatore vengono generalmente salvati dal software, all'inizio della routine di gestione dell'interruzione, e ripristinati subito prima della sua fine.

Soluzione 3.7: viene semplicemente sospeso, per il tempo necessario a gestire l'interruzione.

Soluzione 3.8: non ci sono differenze fra interruzioni esterne ed interne: il programma viene in ogni caso sospeso.

Soluzione 3.9: normalmente vengono perse. Se vengono memorizzate non ne devono comunque arrivare più di una dalla stessa periferica.

Soluzione 3.11: no.

Soluzione 3.13: per sincronizzare più processi.

Soluzione 3.14: no.

Soluzione 3.15: sì.

Soluzione 3.16: a.

Soluzione 3.19: c.

A.4. ESERCIZI DEL CAPITOLO 4.

Soluzione 4.1: b.

Soluzione 4.2: d.

Soluzione 4.3: c.

Soluzione 4.5: no: non sarebbe possibile.

Soluzione 4.6: non si può determinare: dipende anche da altri fattori. La velocità massima è di 120 caratteri/s.

Soluzione 4.8: 11010010.

Soluzione 4.10: no: solo un numero pari di errori.

Soluzione 4.11: no.

Soluzione 4.12: sì.

Soluzione 4.13: no.

Soluzione 4.14: c.

Soluzione 4.15: più informazione di una rete Ethernet.

Soluzione 4.16: no.

Soluzione 4.17: no.

A.5. ESERCIZI DEL CAPITOLO 5.

Soluzione 5.1: falso: devono tenere sotto controllo solo gli eventi che, istante per istante, possono interessare l'esecuzione del programma.

Soluzione 5.2: vero, perché hanno una complessità minore. È però anche vero che quelli non modali sfruttano sempre lo stesso canovaccio, a cui è necessario di volta in volta apportare solo le correzioni opportune.

Soluzione 5.3: vero.

Soluzione 5.4: b.

A.6. ESERCIZI DEL CAPITOLO 6.

Soluzione 6.1: c.

Soluzione 6.2: d, c.

A.7. ESERCIZI DEL CAPITOLO 7.

Soluzione 7.12: il click attiverà una procedura di editing del testo contenuto nel campo.

Soluzione 7.15: b.

Soluzione 7.16: c.

Soluzione 7.17: b.

Soluzione 7.18: b.

Soluzione 7.19: vero.

Soluzione 7.20: falso, perché i pulsanti non possono contenere informazioni testuali.

Soluzione 7.21: c.

Soluzione 7.22: no.

Soluzione 7.23: no.

Soluzione 7.24: no.

Soluzione 7.25: vero.

Soluzione 7.26: falso.

Soluzione 7.27: c.

Soluzione 7.29: c.

Soluzione 7.30: per valore.

Soluzione 7.31: d.

Soluzione 7.32: b.

Soluzione 7.34: no.

Soluzione 7.35: no.

Soluzione 7.36: sì.

Soluzione 7.37: sì, basta abilitare la proprietà *shared text*.

Soluzione 7.38: sì, è una situazione del tutto normale.

Soluzione 7.39: sì.

Soluzione 7.40: no.

Soluzione 7.41: sì, è ciò che si fa normalmente.

Soluzione 7.42: no.

Soluzione 7.43: falso: esistono anche le variabili globali.

Soluzione 7.44: può essere cambiato da uno script.

Soluzione 7.45: b. Le chiamate delle funzioni seguono lo stesso ordine del passaggio dei messaggi.

Soluzione 7.46: no, a meno che non siano variabili globali.

Soluzione 7.47: no, hanno valore per tutti gli stack attivi.

Soluzione 7.48: no.

Soluzione 7.51: a.

Soluzione 7.52: a. Non si crea ambiguità, perché i numeri di ID sono comunque diversi.

Soluzione 7.53: no, ovviamente.

Soluzione 7.54: no.

Soluzione 7.55: falso: i messaggi possono essere generati anche da handler e funzioni.

Soluzione 7.56: c.

Soluzione 7.57: b: la proprietà *shared text* è definita solo per i campi della cornice.

Soluzione 7.58: b.

Soluzione 7.59: c.

Soluzione 7.60: no.

A.8. ESERCIZI DEL CAPITOLO 8.

Soluzione 8.1: c.

Soluzione 8.2: c.

Soluzione 8.3: a.

Soluzione 8.7: vero.

Soluzione 8.8: falso.

Soluzione 8.9: falso.

Soluzione 8.13: falso.

Soluzione 8.15: sì.

Soluzione 8.16: b.

Soluzione 8.17: sì.

Soluzione 8.22: c.

Soluzione 8.23: b.

Soluzione 8.24: b.