Gianluca Rossi

PROGRAMMAZIONE DEI CALCOLATORI
APPUNTI DELLE LEZIONI

(maggio 2017)

Capitolo 1

Problemi, algoritmi, macchine e linguaggi di programmazione

Il computer, nelle sue numerose incarnazioni, è diventato un elemento essenziale della società moderna ed indispensabile strumento della nostra quotidianità. L'uso del calcolatore, infatti, è uscito dai campi tradizionali del calcolo scientifico per entrare in tutte le aree della nostra quotidianità.

Regna, tuttavia, una certa ambiguità sul concetto diffuso di informatica e, per questo, è importante capire che cosa essa *non* è. Chiunque intenda intraprendere percorsi formativi universitari nel campo dell'informatica deve sapere che questa disciplina ha poco in comune con l'"alfabetizzazione informatica" ovvero saper usare un computer per scrivere un testo, navigare in Internet, usare la posta elettronica o installare una stampante: sarebbe come dire che studiare astrofisica consista nell'imparare a usare un telescopio. Allo stesso modo, l'informatica non consiste semplicemente nello scrivere programmi per calcolatore, anche se è naturale aspettarsi da un informatico la capacità di farlo in modo corretto ed efficiente.

In base alla definizione fornita dal gruppo di lavoro dell'*Association for Computing Machinery* nel 1989, l'informatica è "lo studio sistematico dei processi algoritmici che descrivono e trasformano l'informazione: la loro teoria, analisi, progettazione, efficienza, implementazione e applicazione." Da questa definizione risulta chiaro che l'informatica è un complesso di conoscenze scientifiche e tecnologiche che permettono di realizzare quello che si potrebbe chiamare *metodo informatico* o *algoritmico*: così come il metodo scientifico può essere riassunto nel formulare ipotesi che spieghino un fenomeno e nel verificare tali ipotesi mediante l'esecuzione di esperimenti, il metodo informatico consiste nel formulare algoritmi che risolvano un problema, nel trasformare questi algoritmi in sequenze di istruzioni (programmi) per le macchine e nel verificare la correttezza e l'efficacia di tali programmi analizzandoli ed eseguendoli.

Il metodo informatico si sviluppa in una serie di fasi successive ce possiamo schematizzare come segue.

1. *Formulazione del modello*. Dall'interazione con esperti di uno specifico settore applicativo viene formulato un modello matematico che rappresenti il più fedelmente possibile il particolare problema oggetto di studio. Questa fase può risultare particolarmente difficile, a causa soprattutto dei diversi linguaggi che, generalmente, vengono usati in diversi contesti applicativi.

2. *Analisi del modello e progettazione dell'algoritmo*. Questa è probabilmente la fase più "creativa" dell'intero processo per la quale l'informatico ha a disposizione un bagaglio molto vasto di tecniche di progettazione, ma che comunque richiede abilità e fantasia combinate con rigore e impegno.

3. *Implementazione*. In questa fase è necessario conoscere almeno un linguaggio di programmazione che consenta all'informatico di "descrivere" l'algoritmo stesso al calcolatore, chiedendo a quest'ultimo di eseguirlo.

4. *Verifica del programma*. Ha lo scopo di individuare i cosiddetti *bug*, ovvero errori concettuali e/o implementativi: nel caso tali errori vengano rilevati sarà necessario modificare la progettazione dell'algoritmo e/o la sua implementazione.

5. *Verifica del modello*. Si verifica che le soluzioni ottenute siano veramente utili per la risoluzione del problema originario: se così non fosse vorrebbe dire che la modellazione del problema effettuate durante la prima fase nascondeva delle debolezze. Occorre, quindi, modificare il modello e ricominciare l'intero processo.

Da quanto detto sopra si evince che l'applicazione del metodo algoritmico richiede delle conoscenze e competenze di vario genere che l'aspirante informatico dovrà necessariamente far sue.

- *conoscenze matematiche e logico-deduttive*, per proporre soluzioni precise e corrette e per realizzarle in un linguaggio di programmazione;

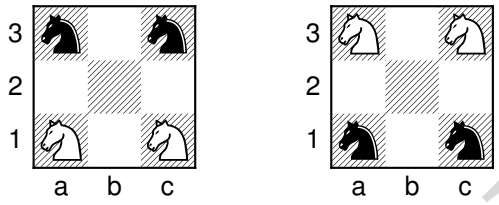
- *conoscenze ingegneristiche*, che permettano di saper modellare il problema in esame, di modulare la soluzione proposta sviluppandola con tecniche che ne garantiscano la manutenibilità;

- *conoscenze di carattere interdisciplinare*, per essere in grado di sviluppare strumenti per settori della società tra i più disparati;

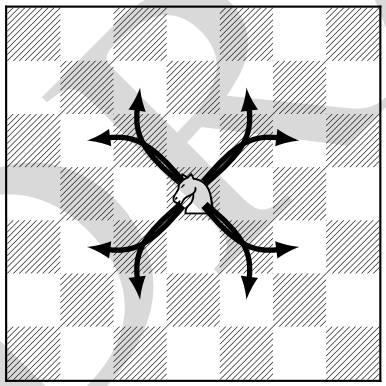
- *conoscenze di carattere etico*, per capire le problematiche di sicurezza, riservatezza e legalità che insorgono nello sviluppo di tali strumenti.

Trovare il giusto modello. Per chiarire quanto descritto sinora, vediamo un primo esempio di applicazione del metodo algoritmico, in cui la scelta, non immediata, del modello implica, quasi automaticamente, lo sviluppo di un algoritmo estremamente semplice.

Il filosofo Romagnolo del quindicesimo secolo Paolo Guarini ideò il seguente rompicapo basato sul gioco degli scacchi.



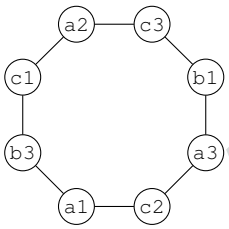
Qual è la sequenza di mosse più breve che consente ai cavalli di passare dalla configurazione a sinistra della figura a quella a destra? Ricordiamo che negli scacchi il cavallo muove di due posizioni in orizzontale ed una posizione in verticale o viceversa. Nella figura che segue sono rappresentate con cerchio bianco le `8` possibili posizioni raggiungibili dal cavallo bianco.



Una possibile soluzione a questo rompicapo potrebbe essere quella di provare tutte le possibili sequenze di mosse dei quattro cavalli selezionando tra di esse quella

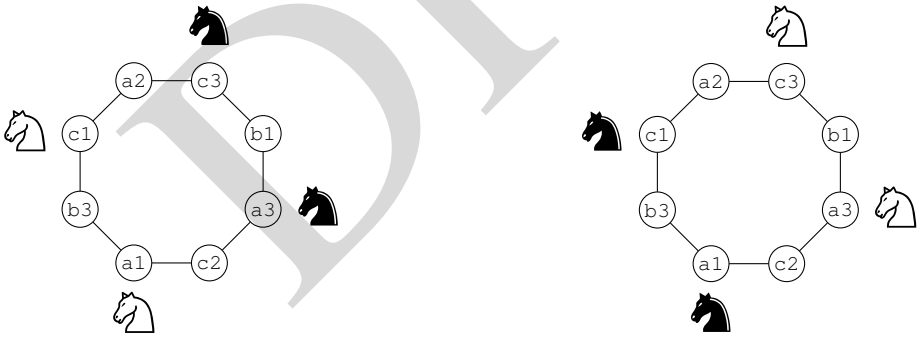
più breve che soddisfi i requisiti del rompicapo stesso. Tuttavia, in tal modo ci renderemmo quasi immediatamente conto che il numero di tali possibili sequenze è molto elevato e che, quindi, tale algoritmo sarebbe del tutto inutilizzabile da un punto di vista pratico.

La soluzione di questo rompicapo può essere ottenuta agevolmente se il problema viene posto in termini diversi ma equivalenti. Da una casella della scacchiera un cavallo può raggiungere un insieme di caselle determinato dalla regola esposta in precedenza. Ad esempio se un cavallo si trova nella casella bianca della riga in basso (casella `b1`) si può spostare soltanto in una delle due caselle nere della riga in alto (caselle `a3` e `c3`) a meno che queste non siano già occupate da un altro cavallo. Quindi esiste una relazione diretta tra la casella `b2` e le caselle `a3` e `c3`; tale relazione rappresenta la possibilità per un cavallo di spostarsi tra le caselle in questione. Si noti che tale relazione è simmetrica in quanto un cavallo che giace su `a3` o `c3` può spostarsi nella casella `b1`. Questa relazione tra le caselle della scacchiera può essere descritta graficamente nel modo che segue.



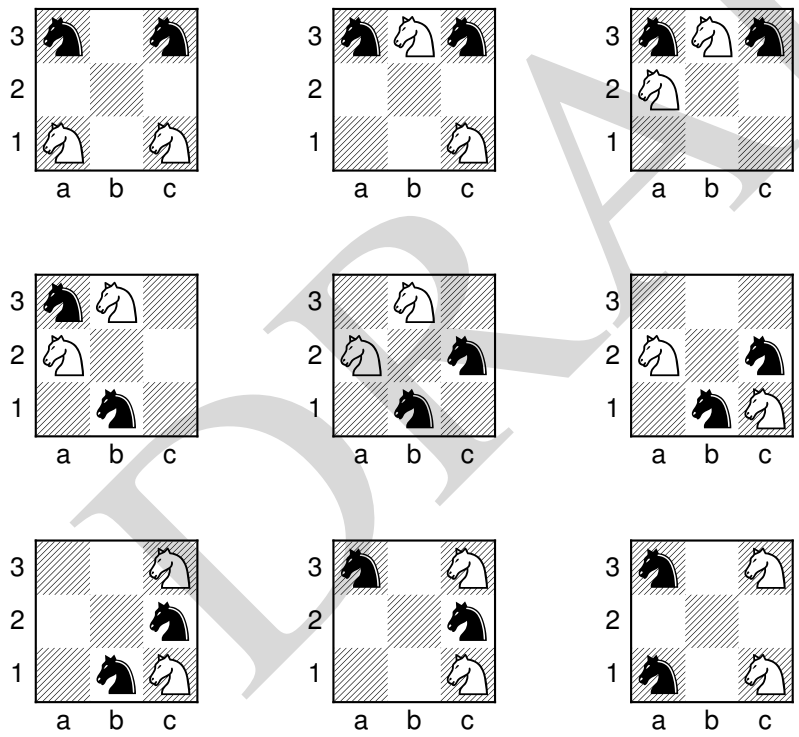
Questa rappresentazione evidenzia con una linea la relazione di raggiungibilità tra caselle: un cavallo può spostarsi tra due caselle con una mossa solo se queste sono *adiacenti* ovvero collegate con da linea. Si osservi che non compare la casella `b2` in quanto questa non può essere raggiunta da nessun'altra casella.

Ora possiamo posizionare i nostri cavalli sul diagramma a cerchio anziché sulla scacchiera e riformulare il rompicapo di Guarini.



Potendo spostare un cavallo dalla sua casella ad una adiacente libera, qual è la sequenza di mosse più breve che consente ai cavalli di raggiungere la configurazione a destra della figura partendo da quella a sinistra?

La rappresentazione grafica del rompicapo di Guarini e la sua successiva riformulazione, ci permettono di progettare abbastanza facilmente un algoritmo risolutivo del rompicapo stesso. A tale scopo, iniziamo con l'osservare che i cavalli devono muoversi nella stessa direzione, altrimenti, prima o poi, questi saranno costretti ad incrociarsi e quindi a occupare la stessa casella. In base a tale osservazione, è naturale progettare il seguente procedimento algoritmico: spostiamo un cavallo alla volta di una casella in senso orario (oppure antiorario) fintanto che i cavalli bianchi in `a1` e `c1` vadano rispettivamente in `c3` e `a3` e i cavalli neri in `c3` e `a3` vadano rispettivamente in `a1` e `c1`. Con quattro mosse, i quattro cavalli raggiungono le posizioni finali per un totale, quindi, di sedici mosse. Di seguito sono mostrate le prime otto mosse che permettono di "ruotare" la disposizione dei cavalli di 90 gradi in senso anti-orario. Lo studente può a questo punto dedurre le successive otto mosse che consentiranno di raggiungere la configurazione finale.



L'esempio appena descritto mostra come la formulazione del giusto modello, con cui rappresentare il problema sotto esame, sia una passo particolarmente importante del metodo algoritmico, in quanto può rendere la progettazione dell'algoritmo molto più semplice. Nell'esempio la progettazione dell'algoritmo di rotazione dei cavalli è stata una naturale conseguenza del modello grafico adottato per la formulazione del problema.

Un altro esempio di questo tipo è il problema del calcolo della radice quadrata di un numero x: $\sqrt{x} = y$ se e solo se $y^2 = x$. Questa definizione non ci dice nulla su come calcolare $\sqrt{x}$; dal punto di vista di un matematico essa è ineccepibile ma dal punto di vista di un informatico che si propone di sviluppare un algoritmo per il calcolo di $\sqrt{x}$ è poco utilizzabile. Per risolvere il problema occorre cambiare modello e rappresentare la radice quadrata di x in un altro modo. Per fortuna la matematica ci dice anche che se z_0 è un qualunque numero positivo allora

$$\sqrt{x} = \lim_{n \to +\infty} z_n \quad \text{dove } z_n = \frac{1}{2}\left(z_{n-1} + \frac{x}{z_{n-1}}\right).$$

Questo modo alternativo di indicare il numero $\sqrt{x}$ è sicuramente meno intuitivo ma suggerisce un metodo per il calcolo effettivo della radice cercata:

1. Stabilisci un margine di errore $\texttt{eps} > 0$;
2. Fissa $z = x$;
3. Fintanto che $|z^2 - x| > \texttt{eps}$ assegna a z il nuovo valore $\frac{1}{2}\left(z + \frac{x}{z}\right)$;
4. Concludi approssimando $\sqrt{x}$ con il valore di z.

Questa descrizione risulta essere abbastanza formale e priva di ambiguità tanto da poter essere utilizzata da una persona che non conosce cosa sia la radice quadrata di un numero ma sia in grado di eseguire operazioni elementari. Non solo, codificata in un certo modo potrebbe essere eseguita da una macchina in modo del tutto automatico. Ecco un esempio:

```
1 eps = 0.0001; z = x;
2 WHILE ( abs(z*z - x) > eps) {
3   z = (z + x/z)/2;
4 }
5 RETURN z;
```

La riga 1 codifica i primi due punti della prima descrizione: abbiamo scelto un valore arbitrario per eps, abbiamo definito una *variabile* z a cui abbiamo assegnato il valore di x. Le righe da 2 a 4 sono caratterizzate dalla parola chiave WHILE: qui non si

fa altro che codificare il terzo punto della precedente espressione ovvero si continua ad eseguire il *corpo* del WHILE, vale a dire tutto ciò che è racchiuso tra le parentesi graffe, fintanto che l'espressione tra parentesi tonde risulta essere vera, ovvero il valore assoluto di $\texttt{z}^2 - \texttt{x}$ risulta maggiore di `eps`. Quando l'espressione tra parentesi tonde diventa falsa si smette di eseguire la riga 3 e si passa ad eseguire la riga 5 che è in qualche modo la stessa cosa dell'ultimo punto della prima descrizione.

L'esempio precedente mostra un primo esempio non banale di *algoritmo*. Questo lo possiamo definite come una descrizione dettagliata e non ambigua della sequenza di operazioni "elementari" che permette, a partire dai dati iniziali (*input*), di ricavare in un tempo finito la soluzione (*output*) del determinato problema. Quando codificato in maniera opportuna, l'algoritmo diventa un *programma* che può essere eseguito in modo automatico da un agente (un calcolatore, un dispositivo meccanico, un circuito elettronico, e perché no, un essere umano).

Limiti del metodo algoritmico Per ogni problema è sempre possibile trovare un algoritmo che lo risolva? La risposta a questa domanda, ovvero esistono dei problemi per i quali non esistono algoritmi che li risolvano, è positiva. Uno di questi è il problema della *fermata*. Prima di definire questo problema osserviamo che un programma potrebbe non terminare ovvero divergere. Ad esempio se nell'algoritmo per il calcolo della radice quadrata dimenticassimo la riga 3 il valore di `z` resterebbe invariato e quindi il test del WHILE non sarebbe mai falso. Quindi ha senso domandarsi se un programma con un determinato input converge (ovvero termina) o diverge. Il problema della fermata lo riassumiamo nel seguente modo: dato un programma `P` ed un suo dato di ingresso `x` è possibile stabilire in maniera automatica se `P` con input `x` termina? Alternativamente, esiste un algoritmo che con dati di ingresso `P` e `x` restituisce TRUE se `P` con input `x` termina e FALSE altrimenti?

In alcuni casi è possibile dimostrare che il programma termina come nel caso del nostro algoritmo per la radice quadrata: poiché $\texttt{z}_\texttt{n}$ tende a $\sqrt{\texttt{n}}$ allora da un certo punto in poi $\texttt{z}^2 - \texttt{x} \leqslant \texttt{eps}$ per ogni `eps` positivo.

Dimostreremo che, in generale, non può esistere un algoritmo che sia in grado di stabilire se un arbitrario programma con un arbitrario dato di ingresso converge. Come vedremo più avanti nel capitolo, un programma è codificato mediante una sequenza di simboli presi dallo stesso alfabeto utilizzato per codificare i dati. Quindi un programma può essere a sua volta un dato di ingresso di un altro programma o parte di esso. Supponiamo che esista un programma chiamato `Termina` che risolve il problema della fermata, ovvero `Termina(P,x)` restituisce TRUE se `P` con input `x` termina e FALSE altrimenti. Poiché questo programma esiste, può essere utilizzato come sotto-programa di una altro programma.

```
Paradosso(P) {
  WHILE ( Termina( P, P ) == TRUE) {

  }
}
```

Il programma `Paradosso` con input `P` termina se e solo se il programma `P` con input `P` non termina. Se, come abbiamo assunto, il programma `Termina` esiste questo ha una codifica e quindi anche il programma `Paradosso` ne ha una. Questa codifica può essere usata come input per lo stesso `Paradosso`. Il programma `Paradosso` con input la sua codifica, ovvero `Paradosso(Paradosso)` può terminare oppure divergere.

- Se `Paradosso` con input `Paradosso` termina allora `Termina(Paradosso, Paradosso)` restituisce FALSE, ovvero `Paradosso` con input `Paradosso` non termina.

- Se `Paradosso` con input `Paradosso` diverge allora `Termina(Paradosso, Paradosso)` restituisce TRUE, ovvero `Paradosso` con input `Paradosso` termina.

Concludendo, abbiamo trovato un input per il programma `Paradosso` che fa divergere il programma quando questo termina oppure lo fa terminare quando questo diverge. Ovviamente un programma che si comporta in modo cosìbizzarro non può esistere.

Il paradosso è stato innescato dall'aver assunto l'esistenza del programma `Termina`, quindi dobbiamo concludere che questo non può esistere, ovvero che il problema della fermata non ammette nessun algoritmo. Questo risultato è stato ottenuto dal matematico inglese Alan Turing nel 1937 ispirato dal lavoro del logico tedesco Kurt Gödel sull'incompletezza dei sistemi formali.

Efficienza degli algoritmi Ora si consideri il problema di determinare se un numero intero $n > 2$ sia primo o meno. La definizione di primalità ci dice che il numero n è primo se e solo se è divisibile solo per 1 e per se stesso. Questa induce un algoritmo che possiamo descrivere nel seguente modo.

```
1 d = 2;
2 WHILE ( d < n && n % d != 0) {
3   d = d+1;
4 }
5 IF ( d == n ) {
6   RETURN TRUE;
7 } ELSE {
8   RETURN FALSE;
9 }
```

Assegniamo ad una variabile chiamata d il valore 2 (riga 1) e, fintanto che essa non è uguale a n e (&&) non divide n (riga 2), il valore di questa viene incrementato di 1 (riga 3). La condizione del WHILE è composta da due sotto-condizioni legate dall'operatore && (si legge *and*): la condizione è vera se e solo se entrambe le condizioni che la compongono sono vere, vale a dire se $d < n$ e, allo stesso tempo, d non divide n ovvero il resto della divisione di n per d è diverso da (!=) 0. Quando la condizione del WHILE diventa falsa ($d = n$ oppure d divide n) usciamo dal ciclo WHILE ed eseguiamo la riga 5 dove viene eseguita l'istruzione IF che funziona nel modo che segue: se la condizione tra parentesi tonde è vera viene eseguita l'istruzione all'interno della prima coppia di graffe che segue (riga 6) altrimenti viene eseguita l'istruzione all'interno della seconda coppia di graffe, quelle dopo ELSE, (riga 8). Infatti, se all'uscita del WHILE il valore della variabile d è n vuol dire che non abbiamo trovato divisori di n e quindi concludiamo dicendo TRUE, ovvero n è primo. Altrimenti il ciclo WHILE è stato interrotto perché l'ultimo valore di d divide n pertanto concludiamo dicendo FALSE, ovvero n è non primo.

Questo algoritmo nel peggiore dei casi esegue $n-2$ divisioni. Si osservi che questo numero potrebbe essere abbassato drasticamente in quanto almeno uno degli eventuali divisori di n deve essere al più $\sqrt{n}$ (in caso contrario il prodotto risultante sarebbe certamente maggiore di n). Quindi, n è primo se e solo se non ha divisori compresi tra 2 e $\sqrt{n}$. Quindi possiamo migliorare il nostro algoritmo riducendo notevolmente il numero di operazioni eseguite nel caso peggiore.

Stiamo toccando un altro importante aspetto dell'informatica ovvero l'efficienza degli algoritmi: visto che più sono le operazioni eseguite maggiore è il tempo di esecuzione dell'algoritmo, si richiede che gli algoritmi, oltre ad esser corretti, eseguano il minor numero di operazioni, ovvero siano efficienti.

1.1 La macchina di Von Neumann

Gli algoritmi come quelli appena descritti, affinché possano essere eseguiti in modo automatico, devono essere opportunamente codificati in modo che l'esecutore auto-

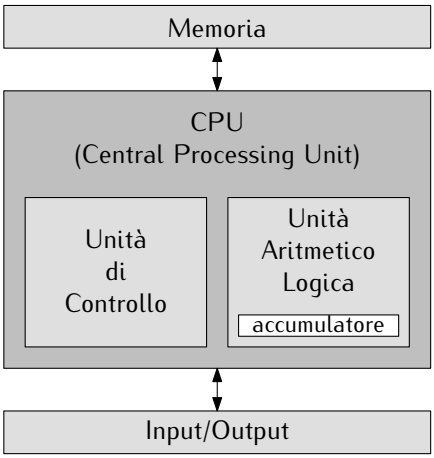


Figura 1.1 La macchina di Von Neumann

matico sia in grado di comprenderli. La codifica dell'algoritmo prende il nome di *programma*. Infine nel codificare gli algoritmi dobbiamo stabilire quantomeno un modello astratto di calcolatore in grado di eseguire il programma.

Tale modello è la macchina di Von Neumann ideata dal matematico John Von Neumann negli anni '40 e poi implementata negli anni successivi. Lo schema della macchina di Von Neumann è illustrato in Figura 1.1. Questa si compone delle seguenti parti.

- La *memoria* che conserva sia il programma (ovvero l'algoritmo opportunamente codificato) che i dati su cui deve lavorare il programma. Può essere vista come un vettore di celle ognuna delle quali contiene una istruzione del programma oppure un dato. Ogni cella è identificabile attraverso un indirizzo che non è altro che un indice del vettore. Una visione più suggestiva della memoria[1] è quella di accomunarla ad un grosso foglio a quadretti dove ogni quadretto contiene una informazione che può essere un dato oppure un frammento di programma.
- La *CPU* ovvero l'*unità di elaborazione* composta da tre elementi principali:
  - L'*unità aritmetico logica* (*ALU*) esegue le istruzioni elementari come quelle aritmetiche e logiche;

[1] Si veda "Informatica, la mela si può mordere" di Pierluigi Crescenzi su Alias, inserto de "il Manifesto del 24 Giugno 2012".

  - L'*unità di controllo* recupera le istruzioni in memoria secondo l'ordine logico stabilito dall'algoritmo e permette la loro esecuzione;
  - L'*accumulatore* è una memoria interna della CPU che viene utilizzata per contenere gli operandi delle istruzioni eseguite dalla ALU.

- L'*input/output* (I/O) costituisce l'interfacciamento del calcolatore verso l'esterno (tastiera, dischi, video...).
- Il *bus di comunicazione* è il canale che permette la comunicazione tra le unità appena descritte.

Torniamo all'algoritmo per la verifica della primalità descritto all'inizio di questo capitolo. Codificato in modo opportuno, l'algoritmo è memorizzato in una porzione di memoria della macchina e l'intero in ingresso (o *input*) in un "quadratino" del foglio a quadretti che indicheremo con `n`. Al momento in cui il programma viene eseguito, l'unità di controllo preleva la prima istruzione (riga 1) del programma che dovrà essere eseguita dalla CPU: nel nostro caso è l'istruzione di assegnamento `d = 2` (assegna il valore `2` alla variabile `d`), quindi la CPU scriverà 2 in un "quadratino" del foglio che convenzionalmente chiameremo `d`. Normalmente l'unità di controllo esegue l'istruzione successiva all'ultima già eseguita. In questo caso viene eseguita l'istruzione nella riga 2. Questa è una *istruzione di controllo* che permette di alterare il flusso del programma in funzione del verificarsi o meno di una condizione. In particolare, se il valore contenuto in `d` è minore o uguale del valore contenuto in `n` e il valore contenuto in `d` non divide quello contenuto in `n` (operazioni eseguite dall'unità aritmetico logica all'interno della CPU) allora l'unità di controllo eseguirà l'istruzione nella riga 3 altrimenti quella in riga 5. La riga 3 aggiunge uno al valore contenuto in `d` e lo sovrascrive in `d`. Infine nella riga 5 troviamo un'altra istruzione di controllo che in base al risultato di una condizione dirotta il flusso del programma sulla riga 6 o sulla riga 8. L'operazione RETURN può esser vista come "scrivi in output il risultato della computazione".

1.2 Al di là dell'astrazione

La memoria di un calcolatore è un vettore di celle contenenti informazioni codificate in modo opportuno. Poiché vengono utilizzati circuiti integrati l'unica informazione possibile è data dallo stato elettrico dei componenti (semplificando, c'è corrente - non c'è corrente). Questo tipo di informazione può essere rappresentata in astratto usando due simboli `0` e `1` per i due stati: ogni cella di memoria contiene una sequenza fissata di questi valori denominati *bit*. Questi due simboli rappresenteranno l'alfabeto del

calcolatore per mezzo del quale esprimere i programmi e i dati (*sistema binario*). Col termine *byte* è indicata una sequenza di 8 bit. Un byte rappresenta il taglio minimo di informazione che un calcolatore è in grado di gestire. Quindi una cella di memoria contiene un numero intero di byte (per i modelli attuali di personal computer questi sono 4 o 8). Una cella di memoria è anche indicata come *parola* o *word*.

Il *linguaggio macchina* descrive come sono rappresentate le istruzioni e i dati che costituiscono i programmi che possono essere eseguiti direttamente dalla CPU. Supponiamo per semplicità che ogni istruzione occupi soltanto una cella di memoria (in generale possono essere più grandi). Ogni istruzione sarà composta di due parti principali: una parte iniziale chiamata *codice operativo* che indica il tipo di azione che si deve eseguire ed una seconda parte costituita dagli operandi. Per esempio l'istruzione

```
1 IF ( valore <= 0 ) {
2   istruzione1;
3 } ELSE {
4   istruzione2;
5 }
```

può avere la seguente codifica

100110	*valore*	*ind1*	*ind2*

dove i primi 6 bit rappresentano il codice operativo, i restanti bit rappresentano gli operandi che in questo caso sono tre anche questi espressi in binario. Il primo operando è il valore confrontato con `0`, il secondo è indirizzo in cui si trova `istruzione1` e l'ultimo è l'indirizzo di memoria in cui è memorizzato `istruzione2`. Si osservi che ogni operando deve occupare un numero predeterminato di bit affinché sia possibile distinguere ogni singolo operando.

Nel linguaggio macchina puro gli indirizzi utilizzati come operandi nelle istruzioni (e che si riferiscono a dati o posizioni all'interno del programma stesso) devono essere indirizzi veri. Quindi il programma cambia a seconda della posizione a partire dalla quale questo viene memorizzato. Tuttavia, al momento della progettazione del programma è più comodo utilizzare delle etichette per indicare sia gli indirizzi di memoria utilizzati per i dati che quelli all'interno del programma. Inoltre anche i codici operativi delle istruzioni "sulla carta" possono essere rappresentati da dei codici più semplici da ricordare. Queste rappresentazioni mnemoniche possono essere formalizzate così da costituire un'astrazione del linguaggio macchina chiamato *linguaggio assembly*. Soltanto al momento della effettiva memorizzazione del programma in memoria per la sua esecuzione sarà necessario sostituire i codici mnemonici e le etichette con i codici operativi e gli indirizzi reali. Questa operazione viene eseguita meccanicamente da un programma chiamato *assembler*.

1.2.1 Codificare i dati

Sia che il programma venga scritto utilizzando direttamente il codice macchina, sia che venga scritto in assembly e poi tradotto con un assembler, sia - come vedremo più avanti - che venga scritto utilizzando un più comodo *linguaggio ad alto livello* il risultato è lo stesso: una sequenza di `0` e `1` che definisce sia i programmi che i dati. In questa sezione descriveremo brevemente come vengono codificati i dati.

Gli interi Siamo abituati a rappresentare i numeri interi positivi utilizzando il *sistema decimale*, ovvero attraverso una sequenza di cifre da `0` a `9`. Il valore rappresentato da una cifra all'interno della sequenza dipende dalla posizione che assume la cifra nella sequenza stessa. Per esempio si consideri il numero `1972`, la cifra `9` vale `900` ovvero 9×10^2 dove `2` è proprio la posizione della cifra `9` all'interno della sequenza (si assume che la cifra più a destra sia in posizione `0`). Quindi

$$1972 = 1 \times 10^3 + 9 \times 10^2 + 7 \times 10^1 + 2 \times 10^0.$$

In definitiva nel sistema decimale il valore dalla cifra in posizione `k` equivale al valore associato alla cifra moltiplicato per `10` elevato alla `k`. L'intero `10` ovvero il numero di cifre di cui disponiamo rappresenta la base del sistema. Tuttavia questo numero non ha nulla di magico e quindi può essere sostituito con qualsiasi altra base.

Nel sistema binario abbiamo a disposizione solo due cifre quindi la cifra in posizione `k` dalla destra ha un valore che vale $b \times 2^k$, dove `b` vale `0` o `1`. Ad esempio

$$\begin{aligned} 11110110100 &= 1\times2^{10}+1\times2^9+1\times2^8+1\times2^7+0\times2^6+1\times2^5+ \\ &\quad 1\times2^4+0\times2^3+1\times2^2+0\times2^1+0\times2^0 \\ &= 1024+512+256+128+32+16+4=1972. \end{aligned}$$

Quanto appena detto ci fornisce un metodo per convertire numeri binari in numeri decimali, ma come avviene il processo inverso?

D'ora in poi adotteremo la convenzione di indicare la base di un numero come pedice del numero stesso quando questa non sarà chiara dal contesto. Ad esempio dalla precedente conversione deduciamo che $11110110100_2 \equiv 1972_{10}$. Ora il nostro problema sarà il seguente: dato un intero `N` in base `10`, qual è la sua rappresentazioni in base `2`, ovvero qual è la sequenza binaria `B` tale che $N_{10} \equiv B_2$? Supponiamo di conoscere la rappresentazione binaria di $\lfloor N/2 \rfloor$, sia questa $B' = b'_k b'_{k-1} \ldots b'_0$ allora se b_0 è il resto della divisione di `N` per `2` (ovvero $b_0 \in \{0,1\}$) si ha

$$\begin{aligned} N &= 2\lfloor N/2 \rfloor + b_0 \\ &= 2\left(b'_k \times 2^k + \ldots + b'_0 \times 2^0\right) + b_0 \\ &= b'_k \times 2^{k+1} + \ldots + b'_0 \times 2^1 + b_0 \times 2^0. \end{aligned}$$

Quindi $N_{10} = (b'_k b'_{k-1} \ldots b'_0 b_0)_2$ ovvero la rappresentazione binaria di N è data dalla rappresentazione binaria di $\lfloor N/2 \rfloor$ seguita - a destra - dal bit che rappresenta il resto della divisione di N per 2. Queste osservazioni inducono un algoritmo per la conversione di qualunque intero N nella sequenza binaria corrispondente.

```
1 k = 0;
2 WHILE ( N > 0 ) {
3   b_k = N % 2;
4   N = N/2;
5   k = k+1;
6 }
7 RETURN b_k b_{k-1} ... b_0
```

Nella tabella che segue troviamo i valori di N, $\lfloor N/2 \rfloor$ e b_k ottenuti con l'algoritmo se $N = 13$.

N	$\lfloor N/2 \rfloor$	b_k
13	6	$b_0 = 1$
6	3	$b_1 = 0$
3	1	$b_2 = 1$
1	0	$b_3 = 1$

Quindi $13_{10} = 1101_2$.

La codifica degli interi negativi può essere fatta in diversi modi, quello più semplice prevede di utilizzare uno dei bit che si ha a disposizione come *bit di segno*. Per esempio se il bit più significativo di una sequenza è lo 0 allora il numero che segue è positivo altrimenti è negativo. Altri modi per rappresentare interi negativi sono la rappresentazione *complemento a 2* e la rappresentazione *in eccesso*.

I razionali Analogamente a quanto avviene per la rappresentazione in base 10, la cifra b_{-i} in posizione i a destra della virgola vale $b \times 2^{-i}$. I numeri razionali possono essere rappresentati in modo molto semplice utilizzando la notazione a *virgola fissa*. Ovvero, se si hanno a disposizione n bit per rappresentare tali numeri, una parte di essi (diciamo k meno significativi) vengono utilizzati per la parte razionale mentre i restanti $n-k$ vengono utilizzati per la parte intera. Se ad esempio $n = 8$ e $k = 2$ la sequenza 10010101 rappresenta il numero 100101.01_2 che corrisponde a $37 + 2^{-2} = 37.25_{10}$. Questa rappresentazione ha lo svantaggio di "sprecare" bit nel caso in cui, per esempio, si rappresentano interi oppure nel caso in cui i numeri da rappresentare sono molto piccoli (tra 0 e 1). Questo problema viene risolto usando una codifica della rappresentazione scientifica del numero, ovvero il numero x viene scritto come $m \times 2^e$ e le singole parti vengono codificate usando interi. Si ottiene una rappresentazione in *virgola mobile* del numero. Lo standard adottato (IEEE 754)

rappresenta un numero in virgola mobile utilizzando di `32` bit. Il primo di questi è il bit di segno; i successivi `8` vengono utilizzati per l'esponente `e`[2], i restanti per rappresentare la mantissa `m`[3].

I caratteri ed il codice ASCII I caratteri (lettere, cifre, punteggiatura, spaziatura) vengono rappresentati attraverso un sistema di codifica a `7` bit denominato *codice ASCII*[4] (American Standard Code for Information Interchange.). Poiché con `7` bit si possono ottenere 2^7 diverse sequenza binarie, ne consegue che i caratteri rappresentabili col codice ASCII sono `128`. Inoltre ogni sequenza binaria del codice ASCII può essere vista come un numero intero da `0` a `127` questo implica che esiste una corrispondenza uno-a-uno tra gli interi da `0` a `127` ed i caratteri. Nella tabella che segue sono mostrati i codici ASCII dei caratteri stampabili, i caratteri mancanti sono spazi, tabulazioni, ritorno-carrello e altri caratteri di controllo. I codici sono rappresentati in forma decimale.

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

[2]L'esponente può essere negativo: la rappresentazione utilizzate è quella in eccesso.
[3]La mantissa è della forma `1.y`, si rappresenta solo la parte dopo la virgola.
[4]pronuncia 'aschi' o 'asci'

Si osservi che i caratteri alfabetici minuscoli hanno codici consecutivi (il carattere 'a' ha codice `97`, 'b' ha codice `98` e via dicendo). La stessa cosa vale per i caratteri maiuscoli e per i caratteri rappresentanti i numeri da `0` a `9`.

1.3 I linguaggi ad alto livello

Il linguaggio macchina ed il linguaggio assembly presentano diversi inconvenienti. Sono dipendenti dall'architettura: ogni processore ha un suo insieme di istruzioni specifico quindi il codice scritto non è portabile. I programmi sono di difficile lettura quindi non adatti alla descrizione di algoritmi complessi.

Per semplificare il compito ai programmatori sono stati introdotti i linguaggi di programmazione ad *alto livello*. Un linguaggio di programmazione ad alto livello è definito in modo da essere svincolato dall'architettura del calcolatore; inoltre può utilizzare elementi del linguaggio naturale rendendo il linguaggio più semplice da utilizzare. Infine, grazie alla sua astrazione da una specifica architettura, i suoi programmi possono essere facilmente trasportati da una macchina ad un altra. Tuttavia il linguaggio deve avere regole precise e non ambigue così che i programmi possano essere tradotti in linguaggio macchina e quindi essere eseguiti da un calcolatore.

Come è stato detto precedentemente, un programma può essere visto come un dato; inoltre un programma produce in output dati, quindi non deve sorprendere l'esistenza di un programma che prende in input un programma scritto in un linguaggio e restituisce in output un altro programma equivalente al primo codificato in un altro linguaggio. Questo significa che possiamo utilizzare un programma per tradurre un programma scritto in un linguaggio ad alto livello nel suo equivalente in linguaggio macchina. Questo processo di traduzione automatica è detto *compilazione* ed il programma che esegue il processo è detto *compilatore*. È questo il processo che instanzia un programma astratto scritto con un linguaggio ad alto livello su una specifica architettura. Attraverso questo processo siamo in grado di far eseguire un programma $\mathtt{P}$ scritto nel linguaggio ad alto livello $\mathtt{L}$ su le architetture $\mathtt{A}_1, \mathtt{A}_2, \ldots, \mathtt{A}_k$ tutte diverse tra di loro. Per far questo è sufficiente avere un compilatore $\mathtt{C}_i$ che trasforma programmi scritti nel linguaggio $\mathtt{L}$ in codice macchina per l'architettura $\mathtt{A}_i$.

$$\mathtt{P} \rightarrow \mathtt{C}_i \rightarrow \mathtt{P}_i$$

Ovvero il compilatore $\mathtt{C}_i$ prende in input il programma $\mathtt{P}$ scritto nel linguaggio $\mathtt{L}$ e produce in output il programma $\mathtt{P}_i$ scritto in codice macchina per l'architettura $\mathtt{A}_i$. Ovviamente i programmi $\mathtt{P}$ e $\mathtt{P}_i$ devo essere equivalenti!

Nella descrizione degli algoritmi fin qui presentati abbiamo utilizzato un formalismo ad alto livello la cui sintassi ricorda i linguaggi di programmazione della famiglia

del C, C++ o Java. Tuttavia i linguaggi di programmazione ad alto livello più comuni condividono gli stessi concetti di base che descriveremo nei capitoli che seguono.

Capitolo 2

Concetti basilari di un linguaggio di programmazione

Nel suo lavoro sul problema della fermata Alan Turing introdusse un formalismo – successivamente chiamato *Macchina di Turing* – che serviva per descrivere tutto ciò che è considerato computabile, ovvero un linguaggio per descrivere gli algoritmi. Quindi ogni linguaggio di programmazione deve essere in grado di descrivere tutti gli algoritmi descrivibili con una Macchina di Turing. Il formalismo introdotto da Turing è tanto potente quanto semplice infatti per simulare una qualsiasi Macchina di Turing sono sufficienti sei tipi di operazioni.

In queste note descriveremo gli algoritmi utilizzando un formalismo molto vicino al linguaggio C. Tuttavia il nostro scopo non è introdurre un linguaggio di programmazione specifico ma piuttosto introdurre dei metodi che permettano la risoluzione di problemi computazionali. Per tale motivo il formalismo adottato è soltanto uno strumento di lavoro rimpiazzabile con un altro altrettanto valido.

A prescindere da quale sia il linguaggio di programmazione ad alto livello (d'ora in poi diremo soltanto linguaggio di programmazione o linguaggio) adottato, affinché questo risulti sufficientemente espressivo esso deve avere delle caratteristiche che lo accomunano agli altri linguaggi. Un linguaggio di programmazione permette di scrivere programmi che elaborano dati di ingresso (*input*) e producono risultati o dati di *output*; i dati sono memorizzati in memoria e vengono riferiti attraverso le *variabili* che sono dei nomi con i quali viene indicato il contenuto di tali porzioni di memoria; l'operatore di *assegnamento* memorizza un determinato valore in una porzione di memoria individuata da una variabile. Inoltre un linguaggio di programmazione deve poter permettere la valutazione di *espressioni* aritmetiche e logiche e deve fornire delle *strutture per il controllo del flusso* del programma che permettono di eseguire un insieme di istruzioni fino al verificarsi di una data condizione oppure di eseguire un insieme di istruzioni piuttosto che un altro in base al verificarsi o meno di una condizione. Nel resto del capitolo descriveremo in modo più dettagliato queste

componenti dei linguaggi di programmazione definendo le regole che definiscono il nostro linguaggio di riferimento.

2.1 Dati e tipi

Ogni dato in un linguaggio di programmazione ha associato un *tipo* che può essere intero (o *int*), virgola mobile (o *float*), carattere (o *char*), stringa, ... I dati possono essere combinati insieme agli *operandi* per formare *espressioni*. In base al tipo del dato su questo possono essere effettuate alcune operazioni piuttosto che altre: ha senso sommare due interi ma non due stringhe.

I dati si dividono in atomici e strutturati. Quelli atomici non sono divisibili (interi, float, caratteri) mentre quelli strutturati sono tipi di dati composti da più dati atomici. Torneremo più avanti a parlare di dati strutturati, mentre nella tabella che segue mostriamo alcuni esempi di tipi di dati atomici.

INT	16	0	-8	2	
FLOAT	3.1416	-0.1	2.0		
CHAR	'a'	'W'	'0'	'+'	'<'	'$'

Si osservi che `2` è considerato un intero e `2.0` un float, questa osservazione è importante perché se in una espressione compare un float il risultato di questa sarà un float, invece se nell'espressione compaiono solo interi il risultato è di tipo intero. Ad esempio

$$5/2 = 2 \quad 5/2.0 = 2.5$$

la prima divisione è tra interi pertanto il risultato di questa sarà di tipo intero ed in particolare la parte intera inferiore di `5/2`; nella seconda divisione compare come operando il float `2.0` quindi il risultato è di tipo float. Questo è un primo esempio di quello che viene comunemente chiamato *overloading* dell'operatore che permette allo stesso operatore di comportarsi in maniera diversa in base ai suoi operandi: nel caso di / con lo stesso simbolo si può eseguire una divisione oppure una divisione intera.

I char vengono racchiusi da apici per distinguerli dagli altri elementi: il carattere '3' è diverso dall'intero `3`; il carattere '+' è diverso dall'operatore + utilizzato per la somma.

Una stringa è una sequenza di caratteri e quindi un dato composto o strutturato: a volte indicheremo le stringhe come caratteri racchiuse da doppi apici: "programmazione", "algoritmo", "123.45", "R5-j+0".

Alcuni linguaggi di programmazione forniscono il tipo di dato booleano per i valori di verità vero (TRUE) o falso (FALSE). Per esempio $4 > 5$ è FALSE mentre $3-$

$4+1 \leqslant 0$ è TRUE. Tuttavia in molti linguaggi (come il C) l'informazione booleana è gestita attraverso gli interi: 0 equivale a FALSE, diverso da 0 equivale a TRUE.

2.2 Variabili e assegnamento

Il concetto di *variabile* è un elemento chiave in quasi tutti i linguaggi di programmazione. È un modo per dare un nome ad una porzione di memoria utilizzata per contenere un determinato tipo di dato. Con una variabile viene creato un legame (*binding*) tra un nome simbolico e la memoria. In base al tipo di dato la quantità di memoria a cui si riferisce una variabile può essere più o meno grande ed è per questo che alcuni linguaggi (C, C++, Java) richiedono di specificare il tipo di dato a cui la variabile fa rifermento nel momento in cui la variabile viene definita (*dichiarazione di variabile*). La sintassi comunemente usata per definire una variabile è

```
TIPO variabile;
```

Nell'esempio che segue vengono definite due variabili `a` e `b` di tipo intero ed una variabile `x` in virgola mobile.

```
INT a, b;
FLOAT x;
```

L'operatore di *assegnamento* permette di scrivere un determinato valore nell'area di memoria associata ad una variabile: il valore deve essere dello stesso tipo della variabile. Utilizzeremo il simbolo '=' per l'operatore di assegnamento[1].

```
variabile = valore;
```

Con questa istruzione assegniamo alla variabile alla sinistra di '=' il valore contenuto alla destra di '='. Ecco un esempio.

```
INT a, b = 6;
FLOAT x, z;
a = 9;
x = 3.1416;
z = a*b*x;
```

Alla variabile intera `b` viene assegnato il valore `6` al momento della sua dichiarazione, alla variabile intera `a` il valore `9` ed a `x` il valore `3.1415`. Infine alla variabile float `z` è assegnato il risultato di una espressione: il valore assegnato a `z` è il valore

[1] L'utilizzo del simbolo $=$ per indicare l'operatore di assegnamento potrebbe essere fonte di equivoci in quanto potrebbe essere confuso col simbolo matematico che indica l'uguaglianza. Per questo motivo, d'ora in poi, indicheremo l'uguaglianza matematica col simbolo $\equiv$

ottenuto moltiplicando (operatore $*$, si legge *star*) il valore di `a`, `b` e `x`. Chi ha qualche reminiscenza di geometria euclidea dovrebbe aver capito che a `z` viene assegnato il valore dell'area dell'ellisse i cui semiassi hanno lunghezza `a` e `b`.

Alcuni linguaggi di programmazione (come Python), che non richiedono la dichiarazione di variabile, deducono il tipo della variabile dal tipo del dato che gli si sta assegnando.

```
a = -9;
x = 3.1416;
y = a + 1.1;
```

In questo caso vengono create le variabili `a`, `x` e `y`, il loro tipo è dedotto dal dato alla destra del simbolo di assegnamento ($=$). Quindi `a` è intera mentre `x` e `y` sono float. In particolare il tipo di `y` è dato dal tipo del valore restituito dall'espressione `a` $+$ `1.1` ovvero il valore contenuto in `a` più `1.1`; il risultato dell'espressione è $-$ `7.9` che è di tipo float. Spesso per risparmiare spazio e se non crea malintesi ricorreremo anche in queste note alla dichiarazione implicita delle variabili.

2.3 Operatori

Gli *operatori* di un linguaggio di programmazione permettono di manipolare i dati eseguendo su di essi operazioni aritmetiche e logiche oltre all'operazione di assegnamento che abbiamo già visto.

Negli esempi finora incontrati abbiamo già utilizzato diversi operatori aritmetici: somma ($+$); differenza ($-$); divisione ($/$); prodotto ($*$) e resto della divisione intera ($\%$).

Gli operatori logici o booleani permettono di combinare espressioni logiche. Queste ultime, come abbiamo già visto nel Capitolo 1 e come vedremo più in dettaglio in seguito, sono utili per definire delle condizioni nelle istruzioni di controllo del flusso. Nella tabella che segue sono descritti i principali operatori logici.

Nome	Simbolo	Utilizzo	Descrizione del risultato
and	&&	*espr1* && *espr2*	TRUE se *espr1* e *espr2* sono entrambe TRUE altrimenti è FALSE
or	||	*espr1* || *espr2*	FALSE se *espr1* e *espr2* sono entrambe FALSE altrimenti è TRUE
not	!	!*espr1*	TRUE se *espr1* è FALSE altrimenti è TRUE

Gli operatori *relazionali* agiscono su due valori numerici e restituiscono un valore booleano. Nella prossima tabella sono elencati gli operatori relazionali ed il loro comportamento in base ai valori degli argomenti.

Nome	Simbolo	Utilizzo	Descrizione del risultato
Minore	$<$	*val1* $<$ *val2*	TRUE se e solo se *val1* $<$ *val2*
Minore-uguale	$<=$	*val1* $<=$ *val2*	TRUE se e solo se *val1* $\leqslant$ *val2*
Maggiore	$>$	*val1* $>$ *val2*	TRUE se e solo se *val1* $>$ *val2*
Maggiore-uguale	$>=$	*val1* $>=$ *val2*	TRUE se e solo se *val1* $\geqslant$ *val2*
Uguale	$==$	*val1* $==$ *val2*	TRUE se e solo se *val1* $\equiv$ *val2*
Diverso	$!=$	*val1* $!=$ *val2*	TRUE se e solo se *val1* $\neq$ *val2*

Infine il risultato dei prossimi operatori è definito bit per bit a partire dalla rappresentazione binaria degli operandi. Per questo motivo i seguenti operatori sono detti *bit-a-bit*. Sia $\mathtt{x}_0\mathtt{x}_1\ldots\mathtt{x}_k$ e sia $\mathtt{y}_0\mathtt{y}_1\ldots\mathtt{y}_k$ la rappresentazione binaria di *val1* e *val2* rispettivamente, sia *val3* il risultato dell'operatore e $\mathtt{b}_0\mathtt{b}_1\ldots\mathtt{b}_k$ la sua rappresentazione binaria allora

Nome	Simbolo	Utilizzo	Descrizione del risultato
And	&	*val1* & *val2*	per ogni $\mathtt{p}$, $\mathtt{b}_\mathtt{p} = \mathtt{x}_\mathtt{p}$**and** $\mathtt{y}_\mathtt{p}$
Or	|	*val1* | *val2*	per ogni $\mathtt{p}$, $\mathtt{b}_\mathtt{p} = \mathtt{x}_\mathtt{p}$**or** $\mathtt{y}_\mathtt{p}$
Not	$\sim$	$\sim$ *val1*	per ogni $\mathtt{p}$, $\mathtt{b}_\mathtt{p} =$ **not** $\mathtt{x}_\mathtt{p}$
Shift sinistro	$<<$	*val1* $<<$ v	per ogni $\mathtt{p} \geqslant \mathtt{v}$, $\mathtt{x}_\mathtt{p} = \mathtt{x}_{\mathtt{p}-\mathtt{v}}$, per ogni $\mathtt{p} < \mathtt{v}$, $\mathtt{x}_\mathtt{p} = 0$
Shift destro	$>>$	*val1* $>>$ v	per ogni $\mathtt{p} \leqslant \mathtt{k} - \mathtt{v}$, $\mathtt{x}_\mathtt{p} = \mathtt{x}_{\mathtt{p}+\mathtt{v}}$, per ogni $\mathtt{p} > \mathtt{k} - \mathtt{v}$, $\mathtt{x}_\mathtt{p} = 0$

Ad esempio se le variabili a, b, c, d ed e vengono definite nel seguente modo

```
a = 6&3;
b = 6|3;
c =~ 6;
d = 6 << 2;
e = 6 >> 2;
```

per ricavare il valore ad esse assegnato dobbiamo considerare la rappresentazione binaria degli operandi. Assumendo che gli interi vangano rappresentati con 7 bit (più uno per il segno) si ha che la rappresentazione binaria di 6 e 3 è rispettivamente 0000110 e 0000011 allora a = 0000010 ovvero 2, b = 0000111 ovvero 7, c = 1111001 ovvero 121, d = 0011000 ovvero 24 e e = 0000001 ovvero 1.

2.4 Controllo condizionato del flusso

Nell'esempio del calcolo dell'area dell'ellisse visto nel Paragrafo 2.2 abbiamo ottenuto un risultato, il valore nella variabile `z`, eseguendo una serie di istruzioni (separate dal carattere '`;`') secondo un ordine sequenziale. Però, in generale, nella soluzione dei problemi è necessario poter disporre di meccanismi che permettano di alterare il flusso sequenziale delle istruzioni in base al verificarsi di un dato evento o condizione. Già nel Capitolo 1 abbiamo incontrato due costrutti che ci hanno permesso di descrivere algoritmi più interessanti come quello per il calcolo di una approssimazione della radice quadrata di un numero e quello per il test di primalià. In questo paragrafo descriveremo in modo più dettagliato le tipiche strutture dei linguaggi di programmazione che permettono il controllo condizionato del flusso del programma.

Il costrutto WHILE ha la seguente sintassi:

```
WHILE (condizione) {
  sequenza di istruzioni;
}
```

dove *condizione* è una espressione booleana. La sequenza di istruzioni all'interno delle parentesi graffe è separata da un '`;`' e nel caso in cui la sequenza contenga una unica istruzione le parentesi graffe sono facoltative[2]. Come già anticipato, il WHILE continua ad eseguire *sequenza di istruzioni* fintanto che *condizione* è TRUE. Sperabilmente *sequenza di istruzioni* deve rendere FALSE *condizione* affinché si eviti un ciclo infinito.

Un costrutto largamente utilizzato è il FOR che ha la seguente sintassi

```
FOR (istruzione1; condizione; istruzione2) {
  sequenza di istruzioni;
}
```

che può essere descritto utilizzando il WHILE nel seguente modo.

```
istruzione1;
WHILE (condizione) {
  sequenza di istruzioni;
  istruzione2;
}
```

[2]Questo è vero in generale: le parentesi graffe sono utilizzate per delimitare un blocco di istruzioni, se il blocco è composto da una sola istruzione le parentesi non necessitano.

Esempio 2.1. Nel programma che segue è mostrato un esempio di utilizzo del ciclo FOR nel calcolo della somma parziale n-esima della serie geometrica ovvero $s_n = \sum_{i \equiv 0}^{n} x^i$.

```
1 s = 1.0; p = 1.0;
2 FOR ( k = 0;  k < n; k = k+1) {
3   p = p*x;
4   s = s+p;
5 }
```

Il programma mostrato per prima cosa inizializza le variabili s e p quindi non fa altro che eseguire n volte le istruzioni nelle righe 3 e 4. All'inizio del passo k (quando viene eseguito il test $k < n$), per $0 \leqslant k \leqslant n$, la variabile s contiene s_k e la variabile p contiene x^k. Questo fatto può essere dimostrato per induzione sui valori che assume la variabile k. Se $k \equiv 0$, $s \equiv 1 \equiv s_0$ e $p \equiv 1 \equiv x^0$. Supponiamo che all'inizio del passo $k-1$ la proprietà sia vera, allora $s \equiv s_{k-1}$ e $p \equiv x^{k-1}$, dopo le righe 3 e 4 $p \equiv x \cdot x^{k-1} \equiv x^k$ e $s \equiv s_{k-1} + p \equiv s_{k-1} + x^k \equiv s_k$ che sono i valore di p ed s all'inizio del passo k. Il programma termina quando $k \equiv n$, ovvero all'inizio del passo n, questo comporta che il valore finale di $s \equiv s_n$. Questo ragionamento dimostra in modo rigoroso che l'algoritmo mostrato è corretto.

Torniamo alle strutture per il controllo del flusso considerando il costrutto IF. Questo permette di eseguire delle istruzioni in base al risultato di una condizione senza creare cicli nel flusso del programma come avviene per il FOR ed il WHILE.

```
IF (condizione) {
  sequenza di istruzioni1;
} ELSE {
  sequenza di istruzioni2;
}
```

Se la condizione è vera viene eseguito il ramo del programma contenente la prima sequenza di istruzioni altrimenti viene eseguito il ramo ELSE. Esiste una versione del costrutto che non prevede il ramo ELSE, in tal caso se la condizione è vera viene eseguita l'opportuna sequenza di istruzioni altrimenti il flusso del programma riprende dall'istruzione dopo l'IF.

Un esempio di utilizzo del IF è nel calcolo del massimo tra due numeri a e b come nell'esempio che segue.

```
IF ( a > b) {
  max = a;
} ELSE {
  max = b;
}
```

La variabile `max` assume il valore di `a` o di `b` in funzione del risultato del test `a` > `b`.

2.5 Decomposizione e astrazione

All'interno di un programma potremmo aver bisogno più volte di calcolare la radice quadrata oppure il massimo tra due numeri. Finora abbiamo visto come fare, tuttavia risulta essere piuttosto noioso riscrivere i pezzi di codice per il calcolo della radice quadrata o del massimo ogni volta che ne abbiamo bisogno.

Pressoché tutti i linguaggi di programmazione forniscono dei meccanismi che permettono di definire *moduli* o *funzioni* per il trattamento di compiti specifici che quindi risultano essere indipendenti e riutilizzabili.

Per il nostro esempio è possibile definire una funzione che prende in input due numeri e ne calcola il massimo.

La sintassi che utilizzeremo per definire una funzione è la seguente.

```
F( lista parametri ) {
  sequenza istruzioni;
  RETURN risultato;
}
```

`F` è il *nome* assegnato alla funzione, *lista parametri* sono i valori di ingresso ovvero l'input della funzione. Segue il *corpo* della funzione composto da *sequenza istruzioni* ovvero le istruzioni eseguite dalla funzione che presumibilmente coinvolgono i parametri di ingresso e, infine, RETURN *risultato* definisce il risultato della computazione, si dice che il valore *risultato* viene *restituito* dalla funzione. Talvolta l'istruzione RETURN non è presente oppure potrebbe essere posizionata in qualsiasi punto all'interno del blocco della funzione: l'effetto è quello di interrompere l'esecuzione della funzione e restituire in output il valore specificato.

L'esempio che segue definisce la funzione `Max` che calcola il massimo tra due numeri.

```
Max( a, b ) {
  IF ( a > b) {
    max = a;
  } ELSE {
    max = b;
  }
  RETURN max;
}
```

Le variabili `a` e `b` sono utilizzate per memorizzare i valori di ingresso della funzione e `max` è una variabile interna della funzione il cui valore viene restituito in output

con l'istruzione RETURN. A questo punto possiamo utilizzare la funzione appena costruita all'interno di un programma come nel caso che segue.

```
tmp = Max( b, c)
max = Max( a, tmp );
```

Assumiamo che `a`, `b` e `c` siano variabili numeriche il cui valore è stato definito precedentemente. La prima riga invoca la funzione `Max` con input i valori contenuti in `b` e `c`, il massimo tra i due valori viene assegnato alla variabile `tmp`. La successiva invocazione di `Max` è con input il valore di `a` e quello di `tmp`. L'effetto è quello di calcolare il massimo tra tre numeri. Lo stesso frammento di codice può essere sintetizzato nel seguente modo.

```
max = Max( a, Max( b, c) );
```

Le funzioni permettono di decomporre problemi in sottoproblemi più semplici consentendo di progettare una soluzione assemblando in modo opportuno i moduli che forniscono le soluzioni dei singoli sottoproblemi. Questa visione modulare permette di creare dei livelli di astrazione che consentono al programmatore o progettista di non aver bisogno di conoscere i dettagli di come è implementato quel determinato modulo: gli basta sapere cosa fa e come si utilizza.

Questo meccanismo permette di scrivere codice più compatto e quindi più semplice da analizzare ed inoltre, decomponendo in sottoproblemi sempre più semplici, viene ridotta la possibilità di commettere errori di programmazione. Infine possono essere utilizzate funzioni precedentemente scritte (anche da terzi) permettendo di estendere le funzionalità del linguaggio stesso.

Abbiamo detto che le variabili utilizzate come parametri nella definizione di una funzione (*parametri formali*) assumono il valore degli argomenti passati alla funzione al momento delle sua invocazione. Consideriamo il codice che segue.

```
F( )  {
  a = 5; b = 10;
  max = Max( a, b );
}
```

Se andiamo a vedere la definizione della funzione `Max` osserviamo che utilizza come parametri formali due variabili denominate `a` e `b` come quelle nel codice sopra. È da osservare che sebbene i simboli adottati siano gli stessi stiamo parlando di due cose diverse: ovvero la variabile `a` in `Max` è un'altra cosa rispetto la variabile `a` in `F`. È lo stesso simbolo che viene utilizzato con due diversi significati, vediamo perché questo non crea ambiguità. Abbiamo detto che una variabile è semplicemente un nome utilizzato per indicare una determinata regione di memoria contenente dati (ma anche

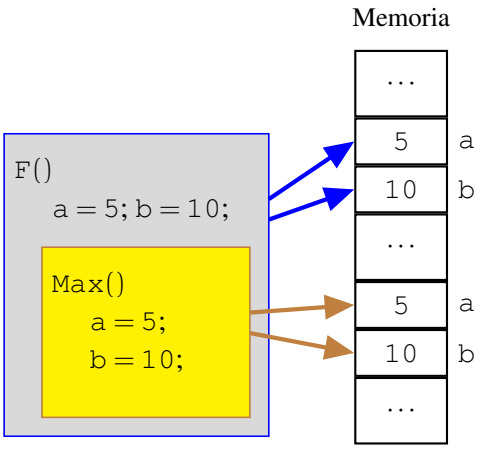


Figura 2.1 Le variabili `a` e `b` nello scope di `F` fanno riferimento a posizioi della memoria diverse rispetto le variabili `a` e `b` nello scope di `Max`.

funzioni), durante l'esecuzione del programma viene creato un abbinamento (mapping) tra variabili – quindi nomi – e memoria, questo mapping è chiamato *ambiente* (scope). Ogni funzione ha il proprio scope: in `F a` è il nome assegnato alla zona di memoria contenente `5` e `b` a quella contenente `10`; poi viene invocata la funzione `Max` e durante la sua esecuzione vige l'ambiente di `Max` in cui con `a` e `b` ci riferiamo ad altre due diverse zone di memoria che in questo caso, perversamente, contengono una copia dei valori che avevano `a` e `b` nello scope di `F`. Terminata l'esecuzione di `Max`, il suo scope viene cancellato e torna in vigore quello di `F`, almeno fin quando `F` è in esecuzione (si veda la Figura 2.1).

Per meglio fissare questo punto vediamo un altro esempio. Consideriamo questa funzione

```
G( a )  {
  a = a+1;
  RETURN a;
}
```

e utilizziamola nel seguente modo

```
P( ) {
  a = 5;
  b = G( a );
}
```

L'esecuzione del programma `P` assegnerà a `b` il valore di $a+1 \equiv 6$ ed il valore di `a` resterà `5` in quanto l'operazione di incremento di `a` nella funzione `G` è all'interno dello scope di `G`: la porzione di memoria a cui si riverisce il nome `a` in `G` non ha nulla a che vedere con quella a cui si riferisce `a` in `P`.

2.6 Gli array

L'*array* è un dato strutturato che permette la memorizzazione di una sequenza ordinata di dati tutti dello stesso tipo. Il fatto che la sequenza sia ordinata implica che ad ogni elemento della sequenza è associato un indice numerico che ne identifica la posizione all'interno della sequenza. Se `a` è un array di `n` elementi allora `a[0]` è il primo elemento della sequenza, `a[1]` il secondo e, in generale, per $0 \leqslant i < n$ `a[i]` è l'$i+1$-esimo elemento.

Nella dichiarazione di una variabile di tipo array, oltre che indicarne il nome ed il tipo degli elementi, bisogna indicarne anche la dimensione, ovvero il numero di elementi di cui è composto. Questo perché in molti linguaggi di programmazione la dimensione degli array è immodificabile. Nel caso in cui l'array viene inizializzato al momento della dichiarazione la sua dimensione è implicita e ricavata dall'inizializzazione.

```
INT a[15];
b[] = { 3, 0, 1, 4, 10, 6 };
```

Nella prima riga viene creato un array `a` di tipo intero composto da `15` elementi, nella seconda viene creato ed inizializzato in modo implicito un array `b` di dimensione `6`.

Esempio 2.2. Vediamo un semplice esempio di utilizzo di array. Assumiamo che `b` sia un array di $n > 0$ cifre binarie, la funzione che segue calcola il valore decimale rappresentato da `b` in forma binaria.

```
Bin2Dec ( b[], n ) {
  decimale = 0;
  p = 1;
  FOR ( k = 0 ;  k < n;  k = k+1) {
    decimale = decimale + b[k]*p;
    p = p*2;
  }
  RETURN decimale;
}
```

La funzione prende come parametri `n` e l'array `b`, si noti che il nome del parametro `b` è seguito dalle parentesi quadre aperte e chiuse per informare la funzione che `b` si riferisce ad un array e non ad un dato atomico. La variabile `p` indica il valore 2^k e questo valore viene sommato a `decimale` nel caso in cui il bit in posizione `k` di `b` è `1`.

Come è stato più volte ribadito, una variabile è un nome assegnato ad una porzione di memoria contente un particolare dato. Questo è vero per i dati atomici: interi, numeri in virgola mobile, caratteri, booleani. Per gli array la situazione cambia. La variabile che identifica un array è un nome assegnato ad un riferimento (o *puntatore* o *indirizzo*) ad una porzione di memoria contenente la sequenza di dati.

```
1 a = 3;
2 b = a;
3 c[] = { 1, 2, 3 };
4 d = c;
5 d[0] = 4; c[1] = 5;
```

La riga 1 crea la variabile `a` assegnandone il valore `3` ovvero viene scritto il valore `3` in un'area dati e questa area viene chiamata `a`. Nella riga 2, in una seconda area dati nominata `b`, viene copiato il valore di `a` (si veda la Figura 2.2 in cui le aree di memoria sono rappresentate con rettangoli). Quando si definisce l'array `c` (riga 3) l'area di memoria denominata `c` contiene un puntatore all'area di memoria contente il dato ovvero l'array. Quindi, quando si esegue la riga 4, nell'area di memoria denominata `d` viene copiato il puntatore presente in `c` creando una situazione come quella illustrata in Figura 2.2 dove risulta che l'array è riferito da due variabili, anche se indirettamente. Quindi assegnare `4` a `d[0]` equivale ad assegnare `4` a `c[0]` e assegnare `5` a `c[1]` è la stessa cosa che assegnare `5` a `d[1]`

Come effetto collaterale di questa interpretazione si ha che quando l'array è parametro di una funzione questa ne può modificare il contenuto in quanto la funzione non ottiene una copia dell'array ma una copia del puntatore originale. Come nel caso dell'esempio, lo stesso array risulterà riferito indirettamente da due variabili: la variabile originale e la variabile parametro nella funzione. Prima di analizzare esempi più complicati che utilizzano questa proprietà presentiamo una funzione se esegue uno scambio tra una coppia di elementi di un array `a`, gli elementi sono individuati dagli indici `i` e `j`

```
Scambia (a[], i, j)  {
  t = a[i];
  a[i] = a[j];
  a[j] = t;
}
```

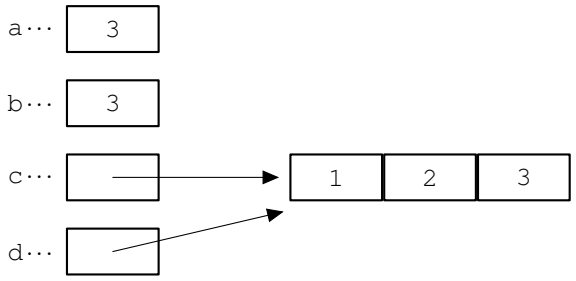


Figura 2.2 Differenza tra dati atomici ed array

La variabile di appoggio `t` viene usata per salvare il valore di `a[i]` che quindi può essere sovrascritto con quello di `a[j]`. Infine ad `a[j]` viene assegnato il vecchio valore di `a[i]` memorizzato in `t`.

Esempio 2.3. Utilizzeremo la proprietà delle funzioni di modificare gli array in input per progettare una funzione che somma uno ad un numero binario memorizzato in un array: una cifra binaria per ogni elemento dell'array.

```
SommaUnoBinaria ( a[], n )  {
  k = 0;
  WHILE (k < n && a[k] == 1) {
    a[k] = 0;
    k = k+1;
  }
  IF (k < n) {
    a[k] = 1;
  }
}
```

L'algoritmo trova il primo zero meno significativo e lo trasforma in uno, tutti gli uno che lo precedono diventano zero. Quindi il ciclo WHILE prosegue scrivendo zeri nelle posizioni incontrate fintanto che l'elemento considerato è un uno e non si è raggiunto il termine dell'array (in tal caso l'array era composto da soli uno). Al termine del ciclo, se non si è raggiunta la fine dell'array – quindi l'uscita dal WHILE è stata causata da un $a[k] \equiv 0$, l'ultimo elemento considerato viene posto a uno. La funzione non necessita di RETURN in quanto il risultato è lo stesso array `a` modificato. Tuttavia potremmo essere più precisi prevedendo un risultato

booleano ed in particolare TRUE se la somma è andata a buon fine, FALSE se l'uscita dal WHILE è stata causata da k == n.

2.7 Stringhe

Le stringhe sono sequenze di caratteri e poiché si adotta la codifica ASCII (Paragrafo 1.2.1) una stringa la rappresenteremo come una sequenza di interi *piccoli*[3] terminata da un da un carattere speciale detto di *fine stringa* e rappresentato con '\0'.

Con l'istruzione c ='a'; assegniamo alla variabile numerica c il codice ASCII del carattere 'a', allo stesso modo avremmo potuto scrivere c = 97; dove 97 è proprio il codice ASCII del carattere 'a'.

I singoli caratteri di una stringa posso o essere letti e modificati utilizzando la sintassi utilizzata per gli array. Invece per quanto riguarda l'inizializzazione possiamo ricorrere alla sintassi seguente.

```
a[] = ''programma''
```

Qui si definisce una stringa a contenente la parola "programma". Quindi a[0] contiene 'p', a[1] contiene 'r' e così via fino a a[9] che contiene il carattere di fine stringa '\0'.

Esempio 2.4. La funzione seguente restituisce la lunghezza di una stringa ricevuta in input.

```
StrLen( a[] ) {
  k = 0;
  WHILE ( a[k] != '\0') {
    k = k+1;
  }
  RETURN k;
}
```

La funzione scandisce l'array contenente la stringa fino ad incontrare il carattere di fine stringa, per ogni nuovo carattere incrementa il valore della variabile k restituita in output a fine scansione.

[3] Piccoli perché la codifica ASCII prevede codici di 8 bit con i quali si codificano al più $2^8 = 256$ simboli diversi, si veda il Paragrafo 1.2.1

2.8 Dati composti

La quasi totalità dei linguaggi di programmazione fornisce dei meccanismi per aggregare più dati possibilmente non omogenei (dello stesso tipo). Queste aggregazioni vengono chiamate *tuple*, *struct* o *record*. Gli elementi delle aggregazioni sono in numero costante e vengono denominati *campi* o *membri*. Ogni membro è identificato da un *nome* (o *etichetta*) al quale è associato un valore. Definendo un nuovo tipo di tupla si definisce un nuovo tipo di dato che, a sua volta, può essere utilizzato come parte di una nuova aggregazione.

La definizione formale di un nuovo tipo di tupla varia da linguaggio a linguaggio. Noi useremo la seguente sintassi:

```
nometupla  {
  TIPO0 nome0;
  TIPO1 nome1;
  ...
  TIPOK nomek;
}
```

Questo definisce un nuovo tipo di dati chiamato *nometupla* composto da `k+1` campi descritti nell'elenco racchiuso tra graffe, nella descrizione è specificato il nome ed il tipo.

```
TUPLAPROVA  {
  FLOAT x, y;
  INT n;
  CHAR s[];
}
```

Nell'esempio di definisce il tipo di dati composto TUPLAPROVA formato da due campi `x` ed `y` di tipo FLOAT, da un campo `n` di tipo intero e un campo `s` di tipo array di caratteri.

TUPLAPROVA può essere utilizzato come un tipo per la creazione delle singole istanze di TUPLAPROVA.

```
TUPLAPROVA t;
```

L'istruzione precedente definisce una variabile `t` di tipo TUPLAPROVA.

L'accesso, sia in lettura che in scrittura, ai campi di una tupla avviene attraverso l'operatore *punto* (.). Ad esempio con `t.x`; si accede al campo `x` della nostra tupla `t`. Si possono assegnare valori ai campi di una tupla racchiudendo i valori tra parentesi graffe come nel seguente esempio

```
t = { 1.2, -0.3, 12, ''stringa'' };
```

dove i campi `x`, `y`, `n` ed `a` sono stati inizializzati rispettivamente a `1.2`, -0.3, `12` e "stringa".

Infine una variabile che identifica una tupla, diversamente dagli array, fa riferimento all'area di memoria contenente la tupla stessa e non ad un riferimento a questa. Pertanto quando le tuple sono argomento di una funzione il parametro formale nella funzione contiene una copia della tupla passata.

Esempio 2.5. Vogliamo scrivere una funzione che prende in input $n \geqslant 2$ punti del piano cartesiano e restituisce il più piccolo rettangoo che li contiene. Iniziamo col definire due tuple utilizzate per descrivere i punti e i rettangoli.

```
PUNTO  {
  FLOAT x, y;
}
RETTANGOLO  {
  PUNTO min, max;
}
```

Per convenzione assumiamo che `min` sia il vertice con coordinate minime e `max` quello con coordinate massime.

La funzione `BoundingBox` prende come parametri di input un array di punti e la dimensione dell'array.

```
1  BoundingBox( p[], n ) {
2    RETTANGOLO r;
3    r.min.x = p[0].x; r.min.y = p[0].y;
4    r.max.x = p[0].x; r.max.y = p[0].y;
5    FOR ( k = 1 ;  k < n ;  k = k+1 ) {
6      IF ( p[k].x < r.min.x) {
7        r.min.x = p[k].x;
8      } ELSE IF (p[k].x > r.max.x) {
9        r.max.x = p[k].x;
10     }
11     IF ( p[k].y < r.min.y) {
12       r.min.y = p[k].y;
13     } ELSE IF (p[k].y > r.max.y) {
14       r.max.y = p[k].y;
15     }
16   }
17   RETURN r;
18 }
```

Nelle righe 2-4 viene definita la variabile `r` di tipo RETTANGOLO i cui valori saranno restituiti in output, questa viene inizializzata utilizzando il primo punto in `p`. Nelle righe 5-16

vengono scanditi i punti rimanenti e se uno di questi ha una coordinata non contenuta nel rettangolo `r` quest'ultimo viene ingrandito fino a coprire il punto. La riga 17 restituisce i valori contenuti in `r`.

Capitolo 3

Efficienza degli algoritmi

Un programma quando è in esecuzione utilizza risorse tra le quali le più importanti sono la memoria del calcolatore ed il tempo di calcolo del processore[1]. È auspicabile che gli algoritmi siano progettati in modo tale che le risorse utilizzate siano economizzate il più possibile.

3.1 Tempo di calcolo

Consideriamo la risorsa tempo di calcolo o semplicemente tempo partendo col definire i metodi di valutazione dell'impiego di questa risorsa. Ovviamente non ha alcun senso contare quanti nanosecondi impiega un programma per terminare in quanto questa grandezza non è funzione soltanto dell'algoritmo implementato e dei dati di input. Quello che si fa è stimare il numero di istruzioni elementari eseguite: infatti, se si assume che il tempo di esecuzione di una istruzione sia costante, il tempo complessivo di esecuzione dell'algoritmo risultetà proporzionale al numero di istruzioni eseguite.

Se il programma non presenta cicli o istruzioni di controllo del flusso, ovvero è un programma sequenziale, il suo tempo di esecuzione dipende soltanto dal numero di istruzioni presenti nel programma, ovvero dalla lunghezza del programma. Invece se sono presenti istruzioni di controllo del flusso il numero di istruzioni eseguite può variare in funzione dei parametri di input.

L'algoritmo che segue cerca il minimo all'interno di un array di `n` interi.

[1] Tra le risorse da economizzare compare anche la quantità di traffico di rete generato dall'applicazione anche se questa risorsa non è propriamente del singolo calcolatore ma di un sistema più ampio che lo include.

```
1 TrovaMinimo(a[], n)  {
2   m = 0;
3   FOR (i = 1;  i < n;  i = i+1) {
4     IF ( a[i] < a[m] ) {
5       m = i;
6     }
7   }
8   RETURN m;
9 }
```

L'algoritmo trova l'indice `m` in cui si trova il minimo elemento dell'array. La variabile `m` viene inizializzata a `0` e, eventualmente, aggiornata nel caso in cui viene trovato un elemento più piccolo di `a[m]`.

Cerchiamo di stimare il numero di operazioni eseguite. Gli assegnamenti nella riga 2 e 3 ($i = 1$) vengono eseguiti una volta sola; il test $i < n$ del FOR viene eseguito n volte ($n-1$ con esito positivo e l'ultimo con esito negativo), l'incremento di i ed il test dell'IF (riga 3) vengono eseguiti $n-1$ volte (solo quando il test $i < n$ ha esito positivo) e il RETURN viene eseguito una volta. Per quanto riguarda l'istruzione in riga 5 questa viene eseguita solo quando viene incontrato un elemento più piccolo del minimo attuale: se il minimo complessivo è in $a[0]$ la riga 5 non verrà mai eseguita; al contrario, se gli elementi dell'array fossero ordinati dal più grande al più piccolo la riga 5 verrebbe eseguita $n-1$ volte. Quindi in totale il numero di operazione varia tra $2+n+2(n-1)+1 \equiv 3n+1$ e $2+n+2(n-1)+1+n-1 \equiv 4n$. Il conto eseguito non è precisissimo in quanto l'istruzione $i = i+1$ non contiene solo un assegnamento ma anche una somma e l'istruzione $a[i] < a[m]$ oltre al test contiene due operazioni di accesso ad un elemento dell'array che è la composizione di tre operazioni elementari (si veda la parte introduttiva del Capitolo 6). Tuttavia questo non rappresenta un problema perché solitamente si è interessati solo all'ordine di grandezza del numero di operazioni eseguite, quindi in questo caso diremo che il numero di operazioni (quindi il tempo di calcolo) è proporzionale ad n, la dimensione dell'array, sia nel caso più benevolo che in quello più malevolo.

Con l'analisi appena eseguita abbiamo espresso il tempo di calcolo di un algoritmo in funzione della dimensine dell'input (nel nostro caso n, la dimensione dell'array) e dalla conformazione dell'input stesso isolando un caso peggiore da uno migliore. Per semplificare ulteriormente vogliamo eliminare la dipendenza dalla conformazione dell'input lasciando soltanto come parametro la sua dimensione. Questo obiettivo si raggiunge considerando soltanto l'istanza peggiore, ovvero quella che richiede più tempo di calcolo. Il seguente algoritmo cerca un elemento `e` in un array `a` di `n` elementi, se lo trova restituisce la posizione di `e` in `a` altrimenti -1.

```
1  TrovaElemento(a[], n, e)  {
2    i = 0;
3    WHILE ( i < n && a[i] != e ) {
4      i = i+1;
5    }
6    IF (i == n) {
7      RETURN -1;
8    } ELSE {
9      RETURN i;
10   }
11 }
```

L'algoritmo scandisce l'array fino a trovare l'elemento oppure fino a raggiungere la fine dell'array. Nel caso in cui `e` non faccia parte di `a`, l'array viene scandito tutto e pertanto il numero di operazioni eseguite è proporzionale a `n`. Se `e` è in `a` allora il numero di operazioni è proporzionale a `i` dove `i` è la posizione della prima occorrenza di `e` in `a`. Questo significa che se `e` fosse in `a[0]` il numero complessivo di operazioni eseguite sarebbe costante (non dipendenti dalla dimensione dell'input). Poiché nella valutazione siamo interessati al caso più pessimistico concludiamo che nel caso peggiore il numero di operazioni eseguite è lineare nella dimensione dell'input.

Esempio 3.1. Consideriamo il seguente problema: dati due array `a` e `b` di `n` e `m` elementi, rispettivamente, con $m \leqslant n$, si vuole progettare una funzione che determini se esiste un segmento di `a` uguale a `b`.

La funzione che segue verifica se `b` coincide col segmento di `a` composto da `m` caratteri che comincia da `a[i]`. Restituisce la posizione `i` se tale segmento viene trovato, altrimenti restituisce -1.

```
1  Sottoarray( a[], b[], n, m ) {
2    FOR (i = 0; i <= n-m; i = i+1) {
3      j = 0;
4      WHILE ( j < m && a[i+j] == b[j] ) {
5        j = j+1;
6      }
7      IF ( j == m) {
8        RETURN i;
9      }
10   }
11   RETURN -1;
12 }
```

Ad ogni esecuzione del ciclo for si verifica se `b` coincide con la sequenza `a[i]`, `a[i+1]`,..., `a[i+m-1]` per ogni `i` possibile. La verifica avviene all'interno del ciclo WHILE. Se la

sottosequenza viene trovata la funzione viene interrotta restituendo il valore di `i` trovato (righe 7-9). Se tale sottosequenza non esiste allora per ogni `i` la ricerca fallisce quindi viene restituito -1.

Veniamo al costo della funzione in termini di tempo nel caso peggiore. Il ciclo più esterno (FOR) viene eseguito $n-m$ volte mentre per il WHILE il caso peggiore si ha quando all'uscita del ciclo `j` contiene $m-1$ e $a[i+j] != b[j]$. Il numero complessivo di operazioni è all'incirca il numero complessivo di volte in cui si esegue il corpo del WHILE ovvero

$$(n-m)(m-1) \equiv m(n-m)-(n-m)$$

che come ordine di grandezza è $m(n-m)$.

3.2 Quantità di memoria

Anche la stima dell'utilizzo di memoria di un algoritmo si vuole far dipendere dalla dimensione dell'input considerando, come nel caso del tempo di calcolo, soltanto l'ordine di grandezza di tale quantità. Nel caso delle funzioni descritte nel paragrafo precedente la quantità di memoria utilizzata, trascurando quella dovuta ad un numero costante di variabili, è la memoria utilizzata per memorizzare l'input, ovvero l'array `a` per le funzioni `TrovaMinimo` e `TrovaElemento` e gli array `a` e `b` nel caso della funzione `SottoArray` descritta nell'Esempio 3.1. Poiché non si può fare a meno di memorizzare l'input e l'output, quello che interessa ai fini della stima della memoria utilizzata da un algoritmo è la memoria aggiuntiva utilizzata ovvero la quantità di memoria utilizzata al netto di quella necessaria per memorizzare l'input e l'output. In tutti i tre casi visti nel paragrafo precedente la memoria aggiuntiva è in quantità costante, non dipendente dall'input.

Consideriamo il problema del calcolo degli elementi della successione numerica di Fibonacci. Questa sequenza numerica fu introdotta dal matematico Leonardo Fibonacci – siamo a Pisa nel XIII secolo – allo scopo di prevedere l'evoluzione di una colonia di conigli partendo dall'assunzione che ogni coppia di conigli, a partire dal secondo mese di vita in poi, genera una coppia di conigli al mese. Partiamo da una coppia di conigli appena nati, al secondo mese questo numero resta invariato, al mese `3` viene generata la prima coppia di conigli per un totale di `2` coppie. Al mese `4` la prima coppia di conigli ha generato un'altra coppia, mentre la seconda generazione di conigli ha solo un mese, quindi in totale le coppie di conigli sono `3`. In generale, sia F_n il numero di coppie di conigli presenti al mese `n`, questo numero è dato dal numero di coppie presenti nel mese precedente (F_{n-1}) più le coppie generate dai conigli presenti nel mese $n-2$ (ovvero F_{n-2}). Quindi la successione di Fibonacci è

definita come segue:

$$F_n \equiv \begin{cases} 0 & \text{se } n \equiv 0, \\ 1 & \text{se } n \equiv 1, \\ F_{n-1} + F_{n-2} & \text{altrimenti.} \end{cases} \tag{3.1}$$

La seguente funzione, dato `n`, restituisce F_n.

```
Fibonacci( n ) {
  INT F[n+1];
  F[0] = 0; F[1] = 1;
  FOR (i = 2; i <= n; i = i+1) {
    F[i] = F[i-1] + F[i-2];
  }
  RETURN F[n];
}
```

Questa funzione memorizza nell'array `F` di dimensione $n+1$ tutti i numeri di Fibonacci fino all'$n+1$-esimo e restituisce l'ultimo trovato. Quindi la memoria aggiuntiva utilizzata dalla funzione dipende dall'input ed in particolare è proporzionale al valore di `n`. Si faccia attenzione al fatto che qui `n` non è la dimensione dell'input ma il valore assegnato ad un parametro della funzione. La dimensione dell'input è un qualche cosa che dipende dalla quantità di memoria utilizzata per memorizzare tutti i valori passati alla funzione. In questo caso è la quantità di memoria utilizzata per memorizzare il valore `n`. Poiché questo valore è rappresentato in forma binaria, il numero di bit utilizzati per memorizzarlo è circa $\log_2 n$. Questo significa che la quantità di memoria utilizzata dalla funzione, ed anche il tempo di calcolo, è *esponenziale* nella dimensione dell'input.

Almeno per quanto riguarda la quantità di memoria utilizzata possiamo progettare un algoritmo meno sprecone.

```
Fibonacci( n ) {
  IF (n == 0 || n == 1) {
    RETURN n;
  }
  f2 = 0; f1 = 1;
  FOR (i = 2; i <= n; i = i+1) {
    f0 = f2+f1;
    f2 = f1; f1 = f0;
  }
  RETURN f0;
}
```

Questo algoritmo si basa sull'osservazione che calcolare per l'`i`-esimo numero di Fibonacci abbiamo bisogno dei due numeri di Fibonacci che lo precedono. La variabile

`f0` dentro il ciclo contiene F_i, `f1` contiene F_{i-1} e `f2` contiene F_{i-2}. Dopo aver calcolato `f0` come somma di `f1` e `f2`, si aggiornano i valori di `f1` e `f2` per l'esecuzione successiva del ciclo. In questo modo viene utilizzata soltanto una quantità costante di memoria aggiuntiva. Per quanto riguarda la risorsa tempo nulla cambia rispetto al primo algoritmo.

Capitolo 4

Ricorsione

La definizione di successione di Fibonacci dell'Equazione 3.1 è stata data in forma *ricorsiva*. Una definizione ricorsiva è composta da una parte diretta, chiamata *base* in cui viene data la risposta diretta dei casi più semplici e da una parte *induttiva*, o *ricorsiva*, in cui il problema viene definito in funzione di versioni più semplici dello stesso problema. Una definizione ricorsiva è ben data se la catena di scomposizioni prima o poi termina nei casi base. Ad esempio dall'Equazione 3.1 sappiamo che $\mathtt{F}_4 \equiv \mathtt{F}_3 + \mathtt{F}_2$, $\mathtt{F}_3 \equiv \mathtt{F}_2 + \mathtt{F}_1$, $\mathtt{F}_2 \equiv \mathtt{F}_1 + \mathtt{F}_0$, siamo arrivati al caso base $\mathtt{F}_1 \equiv 1$ e $\mathtt{F}_0 \equiv 0$ e quindi possiamo ricostruire $\mathtt{F}_2 \equiv 1$ quindi $\mathtt{F}_3 \equiv 2$ e $\mathtt{F}_4 \equiv 3$.

La ricorsione non è solo una tecnica per fornire definizioni di problemi, è anche utilizzabile per risolvere problemi. Ovvero è utilizzabile come tecnica di programmazione grazie al fatto che i linguaggi di programmazione permettono chiamate ricorsive delle funzioni. Ovvero una funzione al suo interno può invocare una istanza di se stessa.

Vogliamo scrivere una funzione che stabilisca se una data sequenza è *palindroma* ovvero se resta la stessa indipendentemente dal verso in cui viene letta. Ad esempio la data del 21 febbraio 2012 se scritta in forma numerica risulta essere palindroma: 21022012. Osserviamo che la sequenza vuota e quella composta da un unico elemento sono palindrome; la sequenza $\mathtt{a}_0, \mathtt{a}_1, \ldots, \mathtt{a}_{n-1}\mathtt{a}_n$ è palindroma se e solo se $\mathtt{a}_0 \equiv \mathtt{a}_n$ e la sequenza $\mathtt{a}_1, \ldots, \mathtt{a}_{n-1}$ è palindroma. Questa osservazione induce la seguente soluzione del problema in cui la sequenza è memorizzata in un array di dimensione $\mathtt{n}$.

```
Palindroma( a[], n ) {
  RETURN PalindromaRic( a, 0, n-1 );
}

PalindromaRic( a[], s, d ) {
  IF ( s >= d) {
    RETURN TRUE;
  }
  IF ( a[s] == a[d]) {
    RETURN PalindromaRic( a, s+1, d-1);
  }
  RETURN FALSE;
}
```

L'invocazione della funzione `PalindromaRic(a, s, d)` verifica se la sottosequenza di $a[s], a[s+1], \ldots, a[d-1], a[d]$ è palindroma. Se $s \geqslant d$ allora la sequenza è vuota o composta da un unico elemento, ricadiamo nel caso base e quindi la sequenza risulta palindroma. Altrimenti se gli estremi della sequenza sono uguali andiamo a verificare che sia palindroma la parte interna, altrimenti sono diversi e la sequenza non è palindroma. Per stabilire se tutta la sequenza `a` è palindroma si invoca la funzione `Palindroma` che innesca la ricorsione e riceve il risultato. Ad esempio, se $a[] = \{21022012\}$ questa è la sequenza di invocazioni ricorsive.

```
Palindroma( a, 8 )  = PalindromaRic( a, 0, 7)
                    = PalindromaRic( a, 1, 6)
                    = PalindromaRic( a, 2, 5)
                    = PalindromaRic( a, 3, 4)
                    = PalindromaRic( a, 4, 3)
                    = TRUE.
```

4.1 Tempo di calcolo

Questa tecnica di progettazione di soluzioni è molto potente in quanto è sufficiente scomporre il problema in problemi più semplici e poi combinare in modo opportuno le soluzioni. Ora cerchiamo di capire come valutarne l'efficienza.

Per valutare il tempo di calcolo di un algoritmo descritto in modo ricorsivo possiamo ricorrere ancora alla ricorsione. Essendo l'algoritmo definito per mezzo di se stesso, anche il tempo di calcolo dipenderà dal tempo di calcolo di tutte le chiamate ricorsive. Vediamo di calcolare il numero di istruzioni eseguite dalla funzione `Palindroma( a, n )` nel caso peggiore, questo è una certa funzione di `n` che

chiameremo T. Se $n \equiv 1$ oppure $n \equiv 0$ il numero di istruzioni eseguito è una certa costante c_0, altrimenti $T(n)$ sarà dato da una seconda costante c_2 alla quale va sommato il numero di operazioni richieste per verificare se una sequenza di $n-2$ elementi è palindroma: questo numero è $T(n-2)$. Riassumendo

$$T(n) \equiv \begin{cases} c_0 & \text{se } n \equiv 0,1, \\ c_1 + T(n-2) & \text{altrimenti.} \end{cases} \tag{4.1}$$

Ora è semplice calcolare la soluzione dell'Equazione 4.1 infatti

$$\begin{aligned} T(n) &\equiv c_1 + T(n-2) \\ &\equiv 2c_1 + T(n-4) \\ &\equiv 3c_1 + T(n-6) \\ &\equiv 4c_1 + T(n-8) \\ &\dots \\ &\equiv kc_1/2 + T(n-k) \end{aligned}$$

La sequenza di uguaglianze ha termine quando $k \equiv n$ oppure $k \equiv n+1$, in entrambi i casi $T(n) \approx c_0 + nc_1/2$, ovvero il numero di operazioni eseguite nel caso peggiore è dell'ordine di grandezza di n.

Per quanto riguarda la quantità di spazio utilizzato occorre tener presente che ogni invocazione ricorsiva necessita del proprio ambiente o scope (si veda il Paragrafo 2.5) e che lo spazio di memoria utilizzato dall'ambiente della funzione chiamante non viene rilasciato prima che termini l'esecuzione della funzione chiamata. Nel caso della funzione `Palindroma()` la quantita di ambienti aperti e non ancora rilasciati può arrivare fino a n quindi anche la quantità di memoria addizionale complessiva utilizzata è lineare in n.

Utilizzare in modo troppo disinvolto il meccanismo della ricorsione potrebbe riservare brutte sorprese. Torniamo al problema del calcolo di F_n: potremmo essere tentati di scrivere una funzione ricorsiva che segua la definizione data nell'Equazione 3.1 nel seguente modo.

```
FibonacciRic( n ) {
  IF ( n == 0 || n == 1 ) {
    RETURN n;
  }
  RETURN FibonacciRic( n-1 ) + FibonacciRic( n-2 );
}
```

L'algoritmo è corretto ma è estremamente inefficiente: per calcolare F_n calcoliamo F_{n-2} che viene calcolato di nuovo quando calcoliamo F_{n-1}. Per quantificare l'inefficienza consideriamo il numero di operazioni $T(n)$ eseguite per calcolare F_n per $n > 1$.

Questo lo possiamo esprimere come $\mathtt{T(n)} \equiv \mathtt{T(n-1)+T(n-2)+costante}$ che è un numero maggiore di $\mathtt{F_n}$ che a sua volta è un numero che cresce esponenzialmente in $\mathtt{n}$ ovvero $\mathtt{F_n} \equiv \mathtt{c}^{\mathtt{n}}$ per una qualche costante $\mathtt{c} > 1$.

4.2 Un problema intrattabile

Il numero di operazioni eseguite dall'algoritmo ricorsivo per il calcolo di $\mathtt{F_n}$ cresce esponenzialmente in $\mathtt{n}$ e per questo motivo è considerato molto inefficiente, vediamo il perché. Supponiamo che il numero di operazioni eseguite da `FibonacciRic( n )` sia $2^{\mathtt{n}}$, i calcolatori moderni eseguono meno di 2.0×10^5 milioni di operazioni al secondo, ovvero circa 2^{35} istruzioni al secondo. Se $\mathtt{n}$ fosse 50 avremmo bisogno di 2^{15} secondi ovvero circa 9 ore di calcolo. Con l'aumentare di $\mathtt{n}$ il tempo di attesa per ottenere il risultato diventa presto insostenibile come illustrato nella seguente tabella in cui il tempo di esecuzione è espresso in secondi (s), ore (h), giorni (g) e anni (a).

n	40	45	50	55	60	65	70	75	80
tempo	0.5 s	17 s	9 h	12 g	388 g	34 a	1089 a	34865 a	1115689 a

L'esponenzialità del tempo di esecuzione rende anche limitato l'effetto di miglioramenti nella velocità dei calcolatori, perché, in tal caso, basta aumentare di poco il numero $\mathtt{n}$ per vanificare ogni miglioramento. Ad esempio supponiamo che la tecnologia attuale ci permetta di eseguire $\mathtt{m} \equiv 2^{\mathtt{a+b}}$ operazioni in un secondo invece delle $2^{\mathtt{a}}$ operazioni al secondo consentite dalla generazione precedente[1]. In queste condizioni occorrerebbero $2^{\mathtt{n}}/\mathtt{m} \equiv 2^{\mathtt{n-a-b}}$ secondi per calcolare $\mathtt{F_n}$ invece dei $2^{\mathtt{n-a}}$ impiegati nella generazione precedente. L'effetto di tale miglioramento viene neutralizzato molto rapidamente al crescere del numero $\mathtt{n}$ in quanto è sufficiente portare tale numero a $\mathtt{n+b}$ per ottenere lo stesso tempo complessivo di esecuzione. In altre parole, un miglioramento delle prestazioni per un fattore *moltiplicativo* si traduce in un aumento solo *additivo* del numero casi resi trattabili.

Per fortuna, per quanto riguarda il calcolo di $\mathtt{F_n}$, abbiamo un algoritmo che esegue un numero lineare di operazioni (questo significa che nei casi descritti in tabella il nostro algoritmo impiegherebbe una frazione di secondo). Purtroppo questo non è sempre vero per molti problemi. Ci sono problemi che ammettono solo algoritmi che eseguono un numero esponenziale di operazioni come il problema che stiamo per introdurre.

Il problema delle **Torri di Hanoi** è un gioco del XIX secolo inventato da un matematico francese Edouard Lucas ispirato ad una leggenda indiana sulla divinità

[1] Si noti che stiamo assumendo qualcosa di molto forte, ovvero che tra una generazione e la successiva di calcolatori le prestazioni aumentano di $2^{\mathtt{b}}$ volte.

Brahma e sulla fine del mondo. In un tempio induista dedicato alla divinità, vi sono tre pioli di cui il primo contiene $n \equiv 64$ dischi d'oro impilati in ordine di diametro decrescente, con il disco più ampio in basso e quello più stretto in alto (gli altri due pioli sono vuoti). Dei monaci *sannyasin* spostano i dischi dal primo al terzo piolo usando il secondo come appoggio, con la regola di non poter spostare più di un disco alla volta e con quella di non porre mai un disco di diametro maggiore sopra un disco di diametro inferiore. La legenda dice che quando i monaci avranno terminato di spostare tutti i dischi nel terzo piolo, avverrà la fine del mondo.

La soluzione di questo gioco è semplice da descrivere usando la ricorsione. Supponiamo di avere spostato ricorsivamente i primi $n-1$ dischi sul secondo piolo, usando il terzo come appoggio. Possiamo ora spostare il disco più grande dal primo al terzo piolo, e quindi ricorsivamente spostare gli $n-1$ dischi dal secondo al terzo piolo usando il primo come appoggio. L'algoritmo che segue implementa questa idea. Questo prende in input quattro parametri: il primo è il numero di dischi, il secondo è il piolo di origine, il terzo è il piolo di appoggio e il quarto è il piolo destinazione.

```
1 TorriHanoi( n, a, b, c ) {
2   IF (n == 1) {
3     sposta un disco da a a c;
4   } ELSE {
5     TorriHanoi( n - 1, a, c, b );
6     sposta un disco da a a c;
7     TorriHanoi( n - 1, b, a, c );
8   }
9 }
```

Se $n > 1$ i primi $n-1$ pioli in `a` vengono spostati in `b` usando `c` come appoggio (riga 5); il disco rimasto in `a` viene spostato in `c` (riga 6); infine gli $n-1$ dischi in `b` vengono spostati in `c` usando `a` come appoggio (riga 7).

Il numero di operazioni di spostamento eseguite dall'algoritmo `TorriHanoi` per spostare `n` dischi è pari a $2^n - 1$, questo risultato può essere dimostrato per induzione: il caso base $n \equiv 1$ è immediato; nel caso $n > 1$, per ipotesi induttiva, occorrono $2^{n-1} - 1$ mosse per ciascuna delle due chiamate ricorsive a cui aggiungiamo la mossa nella riga 6, per un totale di $2 \times (2^{n-1} - 1) + 1 \equiv 2^n - 1$ mosse.

Inoltre è possibile dimostrare che questo numero di mosse è necessario, ovvero non è possibile risolvere il problema con meno di $2^n - 1$ operazioni di spostamento.

Il problema delle Torri di Hanoi mostra dunque che, anche se un problema è risolubile mediante un algoritmo, non è detto che l'algoritmo stesso possa sempre risolverlo in tempi ragionevoli. Questo rende il problema intrattabile, ovvero non risolvibile efficientemente.

4.3 Alcuni problemi di calcolo combinatorico

La ricorsione si presta molto bene alla risoluzione di problemi di calcolo combinatorico, cominciamo dai più semplici.

Il primo problema affrontato è la generazione tutte le 2^n sequenze binarie di lunghezza n, che possiamo equivalentemente interpretare come tutti i possibili sottoinsiemi di un insieme di n elementi. Per illustrare questa corrispondenza, enumeriamo gli elementi da 0 a $n-1$ e associamo il bit in posizione p della sequenza binaria all'elemento p dell'insieme (dove $0 \leqslant p \leqslant n-1$): se tale bit è pari a 1, l'elemento p è nel sottoinsieme; altrimenti, il bit è pari a 0 e l'elemento non appartiene a tale sottoinsieme. Le 2^n sequenze binarie le memorizzeremo in un array binario a che viene riutilizzato ogni volta sovrascrivendone il contenuto.

I bit della sequenza vengono generati da quello con indice minore a quello con indice maggiore. Se a[p] è il prossimo elemento da generare questo deve valere prima 0 e poi 1, gli elementi successivi saranno generati ricorsivamente a partire da queste due assegnazioni. Vale a dire, l'elemento in posizione p ha lo stesso valore in tutte le sequenze ottenibili variando gli elementi che lo seguono in tutti i modi possibili.

La funzione ricorsiva GenSeqBinRic() genera tutte le sotto-sequenze composte dai caratteri '0' e '1' a partire dalla posizione k della stringa a.

```
1  GenSeqBinRic( a[], k ) {
2    IF ( a[k] == '\0') {
3      mostra la sequenza a;
4    } ELSE {
5      a[k] = '0';
6      GenSeqBinRic( a, k+1 );
7      a[k] = '1';
8      GenSeqBinRic( a, k+1 );
9    }
10 }
11 GeneraSequenzeBinarie( n ) {
12   CHAR a[n+1];
13   a[n] = '\0';
14   GenSeqBinRic(a, 0 );
15 }
```

La funzione GeneraSequenzeBinarie() innesca la ricorsione invocando la funzione ricorsiva su input n.

Esempio 4.1. Considerando il caso `n` $\equiv$ `4`, l'algoritmo di generazione delle sequenze binarie produce prima tutte le sequenze che iniziano con `'0'` per poi passare a produrre tutte quelle che iniziano con `'1'`. Lo stesso principio viene applicato ricorsivamente alle sequenze di lunghezza 3, 2 e 1 ottenendo quindi tutte le sequenze binarie di quattro simboli nel seguente ordine (leggere dall'alto in basso e poi da sinistra a destra):

```
0000    0100    1000    1100
0001    0101    1001    1101
0010    0110    1010    1110
0011    0111    1011    1111
```

Come secondo esempio mostriamo un algoritmo che genera tutte le *combinazioni di* `n` *elementi di classe* `m` ovvero tutti i sottoinsiemi di cardinalità `m` di un insieme di `n` elementi. Come nel caso precedente, descriviamo un sottoinsieme di un insieme di dimensione `n` utilizzando un array binario `a` di dimensione `n` che in posizione `p`$-$`1` vale `1` se e solo se l'elemento `p`-esimo dell'insieme appartiene al sottoinsieme.

L'algoritmo ricorsivo `GeneraCombinazioniRic()` risolve il problema più generale di posizionare in tutti i modi possibili `m` uno nelle prime `k`$+$`1` posizioni dell'array `a`. Se `k` $<$ `0` viene mostrata la sequenza `a`. Altrimenti in posizione `k` di `a` viene posizionato, a turno, `0` (nel caso `k` sia almeno `m` e quindi rimane spazio per gli `m` uno) e `1` (se `m` è maggiore di zero); dopo aver posizionato l'elemento in posizione `k` viene invocata la ricorsione per riempire le restanti `k` posizioni dell'array (quelle che vanno dalla posizione 0 alla posizione `k`$-$`1`). Nel primo caso, avendo messo uno zero in posizione `k`, la ricorsione dovrà posizionare `m` uno; mentre, nel secondo caso, visto che un uno è stato utilizzato in posizione `k`, se ne dovranno posizione altri `m`$-$`1` nelle `k` posizioni rimaste libere.

```
1  GeneraCombinazioniRic( a[], k, m ) {
2    IF (k < 0) {
3      mostra la sequenza a;
4    } ELSE {
5      IF (k >= m) {
6        a[k] = 0;
7        GeneraCombinazioniRic( a, k-1, m );
8      }
9      IF (m > 0) {
10       a[k] = 1;
11       GeneraCombinazioniRic( a, k-1, m-1 );
12     }
13   }
14 }
15 GeneraCombinazioni( n, m ) {
16   INT a[n];
17   GeneraCombinazioniRic( a, n, m );
18 }
```

La funzione `GeneraCombinazioni()` fa partire la ricorsione creando un array di dimensione `n` che verrà utilizzato come primo parametro della chiamata alla funzione ricorsiva insieme con gli interi `n` ed `m`.

Infine come ultimo esempio mostriamo un algoritmo per la generazione di tutte le permutazioni degli `n` elementi contenuti in una sequenza `a`. Anche in questo caso progettiamo una funzione, chiamata `GeneraPermutazioniRic()`, che risolve un problema più generale: permuta in tutti i modi possibili i primi `k` elementi dell'array `a` di dimensione `n`. La funzione è ricorsiva e agisce nel seguente modo: ciascuno dei `k` elementi viene scambiato, a turno, con quello in posizione $k-1$ mentre i precedenti $k-1$ elementi vengono permutati ricorsivamente.

```
1  GeneraPermutazioniRic( a[], k ) {
2    IF (k == 0) {
3      mostra la sequenza a;
4    } ELSE {
5      FOR (i = 0; i < k; i = i+1) {
6        Scambia( a, i, k-1 );
7        GeneraPermutazioni( a, k-1 );
8        Scambia( a, i, k-1 );
9      }
10   }
11 }
12 GeneraPermutazioni( a[] ) {
13   GeneraPermutazioniRic( a, StrLen( a ) );
14 }
```

L'utilizzo della funzione `Scambia`() (descritta nel Paragrafo 2.6) dopo la ricorsione serve a mantenere la proprietà che gli elementi, dopo esser stati permutati, vengano riportati nella loro posizione di partenza. Questo passaggio è fondamentale in quanto, dopo che l'elemento in posizione `i` è stato scambiato con quello in posizione $k-1$ e che sono state generate tutte le permutazione dei primi $k-1$ elementi, il prossimo elemento da mettere in posizione $k-1$ dovrà essere quello che si trovava in posizione $i+1$ *prima* dello scambio di `i` con $k-1$. La funzione `GeneraPermutazioni`() fa partire la ricorsione invocando la funzione `GeneraPermutazioniRic`() con input l'array contente gli elementi da permutare e la lunghezza di questo.

Esempio 4.2. Sia $n \equiv 4$ e gli elementi da permutare siano `a`, `b`, `c` e `d`. L'algoritmo genera prima le permutazioni aventi `a` in ultima posizione (elencate nella prima riga), poi quelle aventi `b` in ultima posizione (elencate nella seconda riga) e così via:

bcda	cbda	cdba	dcba	bdca	dbca
dcab	cdab	cadb	acdb	dacb	adcb
bdac	dbac	dabc	adbc	badc	abdc
bcad	cbad	cabd	acbd	bacd	abcd

Restringendoci alle permutazioni aventi `a` in ultima posizione (prima riga), i rimanenti elementi `b`, `c`, `d` vengono permutati in modo analogo usando la ricorsione su questi tre elementi.

Le tre funzioni mostrate in questo paragrafo eseguono un numero esponenziale di operazioni ma questo è necessario in quanto l'output prodotto è di per se di dimensione esponenziale.

Ringraziamenti

La prima parte della Sezione 4.3 è parzialmente estrapolata dal capitolo aggiuntivo (disponibile solo sul web) del libro “Strutture di dati e algoritmi” di P. Crescenzi, G. Gambosi, R. Grossi e G. Rossi edito da Pearson Italia.

Capitolo 5

Ordinamento e ricerca

Ordinare gli elementi di un insieme secondo qualche criterio oppure ricercare uno o più elementi in un insieme sono tra i problemi di base più ricorrenti in informatica, in quanto, spesso, questi appaiono come sottoproblemi di problemi più complessi.

Nel Paragrafo 3.1 abbiamo già affrontato un primo semplice problema di ricerca: dato un insieme di `n` elementi di tipo intero (in reatà va bene qualsiasi tipo purché sugli elementi si possa definire una relazione d'ordine) memorizzato in un array `a`, progettare una funzione che restituisca l'elemento minimo dell'insieme. Abbiamo proposto la funzione `TrovaMinimo` che restituisce la posizione contenente l'elemento minimo dell'array. Di seguito è mostrata una versione leggermente più generale di quella mostrata nel Paragrafo 3.1 dove la ricerca del minimo è ristretta ad una sottosequenza di `a` delimitata dagli indici $\texttt{sx} \leqslant \texttt{dx}$. Il numero di operazioni eseguite è proporzionale a $\texttt{dx} - \texttt{sx}$.

```
1 TrovaMinimoSottoseq( a[], sx, dx ) {
2   m = sx;
3   FOR ( i = sx+1 ;  i <= dx ;  i = i+1) {
4     IF ( a[i] < a[m]) {
5       m = i;
6     }
7   }
8   RETURN m;
9 }
```

Gli algoritmi che analizzano tutti gli elementi di una sequenza in ordine di apparizione sono detti di *scansione lineare*. Questo tipo di ricerca viene utilizzata quando non è possibile formulare ipotesi su come sono memorizzati gli elementi nella sequenza e, pertanto, l'elemento cercato potrebbe essere in una qualsiasi posizione.

5.1 Ordinamento

Ordinare una sequenza significa ridisporre gli elementi in modo che questi seguano un ordine prestabilito. Per semplicità assumiamo che gli elementi siano di tipo intero, che siano memorizzati in una array `a` di `n` elementi e che si vogliano disporre dal più piccolo al più grande, ovvero in modo non decrescente.

Il primo algoritmo che vedremo, denominato *ordinamento per selezione* o *selection sort* esegue `n` passi ed al passo `i` sostituisce l'elemento in posizione `i` ($0 \leqslant i \leqslant n-1$) con il minimo tra gli elementi nelle posizioni comprese tra `i` e $n-1$. Questo ci assicura che al termine del passo `i` gli elementi dalla posizione `0` alla posizione `i` si trovano nelle loro posizioni finali.

```
1 SelectionSort( a[], n ) {
2   FOR (i = 0; i < n-1; i = i+1) {
3     m = TrovaMinimoSottoseq( a, i, n-1);
4     Scambia(a, i, m);
5   }
6 }
```

Come si vede in questa funzione viene usata massicciamente la decomposizione utilizzando la funzione `TrovaMinimoSottoseq` e la funzione `Scambia` descritta nel Paragrafo 2.6.

Tenendo conto che ogni invocazione della funzione `TrovaMinimoSottoseq` esegue all'incirca $n-i$ operazioni e la funzione `Scambia` ne esegue un numero costante `c`, concludiamo che il numero di operazioni totali eseguite da `SelectionSort` è dell'ordine di

$$\sum_{i\equiv 0}^{n-1} (n-i+c) \equiv cn + \sum_{i\equiv 1}^{n} i \approx cn + n(n+1)/2.$$

Ovvero il numero di operazioni è proporzionale a n^2.

La correttezza dell'algoritmo è assicurata dal fatto che durante ogni iterazione del ciclo FOR viene mantenuta la seguente proprietà: al termine del passo `i`, per $i \equiv 0,\ldots,n-1$, gli elementi dalla posizione `0` alla posizione `i` si trovano nelle loro posizioni finali. Questo tipo di proprietà, che viene mantenuta durante tutta l'esecuzione dell'algoritmo, viene detta *invariante*.

Ora vedremo un altro algoritmo di ordinamento che ha una diversa proprietà invariante. Anche questo algoritmo esegue `n` passi e al passo $i \equiv 1,2\ldots,n-1$ l'elemento in posizione `i` viene inserito al posto giusto tra i primi $i+1$ elementi. In altre parole al passo `i` l'elemento `a[i]` viene posto nella giusta posizione all'interno della sequenza di elementi da `a[0]` ad `a[i]`. Questo algoritmo prende il nome di *insertion sort* o algoritmo di *ordinamento per inserimento*.

```
1  InsertionSort( a[], n ) {
2    FOR (i = 1; i < n; i = i+1) {
3      p = a[i];
4      j = i;
5      WHILE ((j > 0) && (a[j-1] > p)) {
6        a[j] = a[j-1];
7        j = j-1;
8      }
9      a[j] = p;
10   }
11 }
```

Con p indichiamo l'elemento da inserire al passo i, ovvero a[i]. L'algoritmo confronta p con i primi i elementi fino a trovare la posizione corretta in cui inserirlo, ogni volta che trova un elemento più grande lo sposta di una posizione in avanti per far posto a p.

La correttezza deriva dalla proprietà che al termine del ciclo WHILE del passo i tutti gli elementi più grandi di p nella sequenze $a[0], a[1], \ldots, a[i]$ occuperanno posizioni maggiori di j che è la posizione che occuperà p.

Per avere una idea del numero di operazioni eseguite dall'algoritmo stimeremo quante volte viene eseguito il test dell'istruzione WHILE. Questo numero dipende dall'istanza, ovvero dalla disposizione iniziale degli elementi di a: se l'array fosse già ordinato il test verrebbe eseguito una sola volta perché $a[i-1] \leqslant a[i]$. Tuttavia nel caso più sfavorevole il WHILE termina quando $j \equiv 0$. In tal caso il numero complessivo di operazioni è all'incirca i per ogni $i \equiv 1, \ldots, n-1$ quindi in totale, come nal caso del Selection Sort, un numero di operazioni dell'ordine di n^2.

Un altro semplice algoritmo di ordinamento è il *bubble sort*, questo scorre ripetutamente l'array scambiando elementi adiacenti che non sono tra di loro nella posizione corretta. La scansione prosegue fino a quando l'array risulta ordinato.

```
BubbleSort( a[], n ) {
  i = 0;
  WHILE ( i == 0 || ( i < n && finito == FALSE) ) {
    finito = TRUE;
    FOR (j = 0; j < n-i-1; j = j+1) {
      IF (a[j] > a[j+1]) {
        finito = FALSE;
        Scambia(a, j, j+1);
      }
    }
    i = i+1;
  }
}
```

Il ciclo FOR più interno viene eseguito almento una volta (quando `i` vale `0`), se l'array risulta ordinato `finito` resta uguale a TRUE e l'algoritmo esce dal WHILE al passo successivo. Altrimenti le scansioni continuano fino a che `i` prende il valore `n` oppure `finito` resta TRUE dopo l'esecuzione del FOR.

L'algoritmo è corretto perché vale la seguente proprietà: al termine del passo `i`, l'$n-i$-esimo elemento in ordine di grandezza di `a` viene spostato in posizione $n-i-1$ di `a`. Per $i \equiv 0$ prima o poi `a[j]` sarà l'ultima occorrenza del massimo in `a`; essendo l'ultimo massimo, per tutti i valori di `j` seguenti il test IF risulta positivo pertanto questo viene scambiato di volta in volta con l'elemento successivo fino a giungere in posizione $n-1$ (inoltre `finito` viene impostato a FALSE garantendo l'esecuzione di un passo ulteriore). Supponendo l'invariante vero fino al passo $i-1$, al passo `i`, il ciclo FOR sposterà il massimo tra gli elementi $a[0], a[1], \ldots, a[n-i-1]$ in posizione $n-i-1$.

Anche in questo caso il numero di operazioni eseguite nel caso peggiore è quadratico nella dimensione dell'array.

5.1.1 Ordinamento per fusione

Gli algoritmi di ordinamento visti finora hanno tutti costo computazionale quadratico nella dimensione della sequenza da ordinare. Il prossimo algoritmo chiamato *merge sort* o *algoritmo di ordinamento per fusione* risultarà essere più efficiente, almeno dal punto di vista del numero di operazioni eseguite.

Su un array `a` di `n` elementi l'algoritmo può essere descritto informalmente nel seguente modo: consideriamo coppie di elementi adiacenti dell'array `a`, se queste non sono posizionate nell'ordine corretto le scambiamo; al passo successivo partiamo da un array diviso in $n/2$ sequenze ordinare di due elementi ciascuna, da coppie di sequenze consecutive ricaviamo una sequenza ordinata composta da quattro elementi; continuiamo ad ordinare coppie di sequenze consecutive ricavate al passo precedente fino ad ottenere una unica sequenza ordinata. Nella Figura 5.1 è mostrato graficamente il comportamento dell'algoritmo su un array di otto elementi.

Un sottoproblema fondamentale che emerge dalla descrizione di questo algoritmo è quello di ordinare una sequenza composta da due sottosequenze già ordinate, questa operazione è detta di *fusione* o *merge*. Si osservi che non ci possiamo permettere di utilizzare uno degli algoritmi fin qui descritti in quanto troppo costosi.

Supponiamo che l'operazione di fusione sia svolta dalla funzione `Fusione`: questa prende in input l'array `a` e tre interi $sx \leqslant cx \leqslant dx$; assumendo che la sottosequenza $a[sx], a[sx+1], \ldots, a[cx]$ e la sottosequenza $a[cx+1], a[cx+2], \ldots, a[dx]$ siano ordinate, la funzione restituisce la sottosequenza $a[sx], a[sx+1], \ldots, a[dx]$ ordinata.

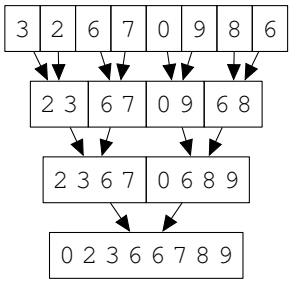


Figura 5.1 Schema di esecuzione dell'algoritmo di ordinamento per funzione

L'algoritmo di ordinamento per fusione si presta bene per essere descritto in forma ricorsiva. Supponiamo di dover ordinare la porzione dell'array `a` delimitata dagli indici $\texttt{sx} \leqslant \texttt{dx}$: se $\texttt{sx} \equiv \texttt{dx}$ la sottosequenza è già ordinata; altrimenti dividiamo la sequenza in due parti uguali, le ordiniamo ricorsivamente e quindi le ordiniamo tra di loro con l'operazione di fusione.

```
1 MergeSort( a[], sx, dx ) {
2   IF (sx < dx) {
3     cx = (sx+dx)/2;
4     MergeSort( a, sx, cx );
5     MergeSort( a, cx+1, dx );
6     Fusione( a, sx, cx, dx );
7   }
8 }
```

La variabile intera `cx` il cui valore è calcolato in riga 3 indica la posizione mediana tra `sx` e `dx`; le righe 4 e 5 eseguono la parte ricorsiva dell'algoritmo mentre la riga 6 ricombina le due sequenze ordinate provenienti dalle due chiamate ricorsive in un'unica sequenza ordinata.

L'operazione di fusione ordina gli elementi delle due sottosequenze di `a` racchiuse dagli indici `sx`, `cx` e $\texttt{cx}+1$, `dx` in un array di appoggio `b`. Partendo da $\texttt{i} \equiv \texttt{sx}$ e $\texttt{j} \equiv \texttt{cx}+1$, memorizziamo il minimo tra `a[i]` e `a[j]` nella prima posizione libera di `b`: se tale minimo è `a[i]`, allora incrementiamo `i` di `1`, altrimenti incrementiamo `j` di `1`. Ripetiamo questo procedimento fino a quando `i` oppure `j` raggiungono il termine delle rispettive sottosequenze. I restanti elementi della sottosequenza non ancora completata (se ve ne sono) vengono riversati nelle successive posizioni libere di `b`. Ora `b` conterrà gli elementi del segmento $\texttt{a[sx]}, \texttt{a[sx}+1\texttt{]}, \ldots, \texttt{a[dx]}$ ordinati in

modo non decrescente, per cui sarà sufficiente copiare b all'interno di tale segmento.

```
1  Fusione( a[], sx, cx, dx ) {
2    i = sx; j = cx+1; k = 0;
3    WHILE ( i <= cx && j <= dx ) {
4      IF (a[i] <= a[j]) {
5        b[k] = a[i]; i = i+1;
6      } ELSE {
7        b[k] = a[j]; j = j+1;
8      }
9      k = k+1;
10   }
11   FOR ( ; i <= cx; i = i+1) {
12     b[k] = a[i]; k = k+1;
13   }
14   FOR ( ; j <= dx; j = j+1) {
15     b[k] = a[j]; k = k+1;
16   }
17   FOR (i = sx; i <= dx; i = i+1) {
18     a[i] = b[i-sx];
19   }
20 }
```

Ad ogni iterazione del ciclo WHILE, l'indice i oppure l'indice j viene incrementato di uno. Il primo parte da sx e arriva al massimo a cx mentre il secondo parte da cx+1 e arriva al massimo a dx. Dunque tale ciclo può essere eseguito al più (cx−sx)+(dx−cx−1) ≡ dx−sx−1 volte[1]. Uno solo dei due cicli FOR viene eseguito, per al più cx−sx o dx−cx−1 iterazioni. Infine, l'ultimo ciclo FOR verrà eseguito dx−sx volte: pertanto, la fusione dei due segmenti richiede un numero di passi proporzionale a dx−sx, ovvero proporzionale alle lunghezze dei due segmenti da fondere. La correttezza dell'algoritmo deriva dall'osservazione che ogni volta che viene aggiunto un nuovo elemento in b, questo è il minimo tra gli elementi restanti

Per stimare il numero di operazioni eseguite dalla funzione MergeSort facciamo riferimento alla Figura 5.1. Osserviamo che questo numero è dato dal numero k di livelli di ricorsione (i livelli dell'albero nella Figura 5.1) moltiplicato per il costo di tutte esecuzioni della funzione Fusione in ogni livello. Se enumeriamo i livelli dal basso verso l'alto, al livello $\ell = 1$ viene eseguita una fusione su n elementi; per $\ell = 2$ vengono eseguite 2 fusioni su n/2 elementi ciascuna e, in generale, al livello ℓ vengono eseguite ℓ fusioni su n/ℓ elementi ciascuna. Pertanto il costo di tutte le

[1] In realtà uno dei due indici, ma non entrambi, arriverà rispettivamente fino a cx+1 o dx+1. Quindi il ciclo WHILE può essere eseguito una volta in più rispetto a quanto affermato ma questo non altera la sostanza perché siamo interessati agli ordini di grandezza.

fusioni su ogni livello risulta sempre essere proporzionale a n. Ora passiamo a valutare quanto vale k rispetto ad n. Osserviamo che si parte da un unico segmento di lunghezza n sul primo livello e, raddoppiando il numero di segmenti da un livello al successivo, si giunge a creare n segmenti di lunghezza 1 sull'ultimo livello k; quindi $2^k = n$, ovvero $k = \log_2 n$. Concludendo, il numero di operazioni complessive risulta proporzionale a $nk = n \log_2 n$.

Questo rappresenta un notevole miglioramento nelle prestazioni rispetto agli algoritmi di costo quadratico in quanto la funzione $n \log_2 n$ ha una crescita *asintoticamente* più lenta della funzione n^2, in particolare la funzione n ha una crescita esponenzialmente più veloce della funzione $\log_2 n$. In Figura 5.2 è mostrato in forma grafica l'andamento delle due funzioni da cui emerge con forza la differenza di crescita tra queste.

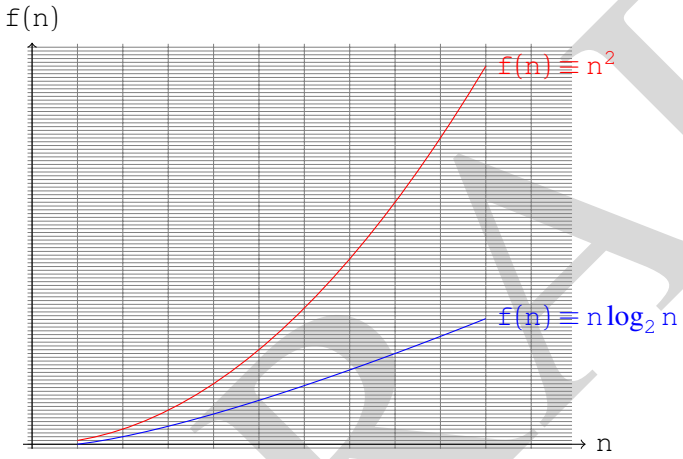


Figura 5.2 Andamento delle funzioni n^2 e $n \log_2 n$.

Sebbene più efficiente dal punto di vista del tempo di calcolo, il merge sort risulta più dispendioso in termini di utilizzo di memoria. Questo perché la funzione `Fusione` utilizza un array di appoggio di dimensione n. Gli altri algoritmi di ordinamento visti finora, invece, ordinavano *in loco*, ovvero senza utilizzo di memoria supplementare.

5.1.2 Ordinare contando

Supponiamo che l'array a da ordinare contenga soltanto due elementi, per esempio 0 e 1, allora è sufficiente scandire gli n elementi dell'array e contare il numero z di occorrenze di 0. L'array ordinato è costruito inserendo z zeri nelle prime z posizioni e completando i restanti elementi dell'array con 1.

Questo algoritmo può essere generalizzato nel caso in cui gli elementi dell'array siano nell'insieme $\{0,1,\ldots,c\}$: contiamo il numero di occorrenze di ogni elemento e, partendo dall'elemento più piccolo, riversiamo nell'array finale a partire dalla prima posizione libera tante copie di ogni elemento quante sono le occorrenze di quell'elemento. L'algoritmo che ne scaturisce è noto come algoritmo di *ordinamento per conteggio* o *counting sort*.

```
1  CountingSort( a[], n, c ) {
2    FOR (e = 0; e <= c; e = e+1) {
3      occ[e] = 0;
4    }
5    FOR (i = 0; i < n; i = i+1) {
6      occ[a[i]] = occ[a[i]]+1;
7    }
8    k = 0;
9    FOR (e = 0; e < c; e = e+1) {
10     WHILE (occ[e] > 0) {
11       a[k] = e; k = k+1; occ[e] = occ[e]-1;
12     }
13   }
14   FOR (; k < n; k = k+1) {
15     a[k] = c;
16   }
17 }
```

Nell'array occ vengono memorizzate il numero di occorrenze degli elementi, più precisamente, occ[e] equivale al numero di volte in cui e compare in a. Dopo aver costruito occ (righe 2-7), tutti i occ[e] elementi e (in ordine crescente) vengono copiati a partire dalla posizione k (righe 8-13), infine, le restanti posizioni libere di a vengono occupate da c (righe 14-16).

Il numero di operazioni eseguite dall'algoritmo e la memoria utilizzata dipendono, oltre che dalla dimensione dell'array, anche da c ovvero il valore del massimo elemento in input. Ma vediamo i dettagli: la costruzione di occ richiede un numero di operazioni proporzionale a $n+c$; per calcolare il numero di volte che vengono eseguite le istruzioni tra le righe 9-13 notiamo che, per un determinato valore di e, il numero di volte che viene eseguito il test del WHILE è $occ[e]+1$, pertanto il costo delle righe in questione è $\sum_{e=0}^{c-1}(occ[e]+1) \equiv n+c$; infine l'ultimo ciclo, tra le

righe 14-16 viene eseguito al più n volte. In totale il numero di operazioni eseguito dall'algoritmo è proporzionale a $n + c$ mentre la memoria aggiuntiva richiesta è proporzionale a c.

5.1.3 Ordinare distribuendo

L'algoritmo di *ordinamento per distribuzione*, detto anche *quick-sort*, è basato sulla seguente idea: selezioniamo un elemento della sequenza chiamato *pivot* e denotiamolo con p; dividiamo la sequenza in tre parti, la prima parte composta dagli elementi minori o uguali a p, la seconda parte composta dal solo p e la terza composta dagli elementi maggiori di p. Si osservi che in questo modo p si troverà nella posizione finale rispetto all'ordinamento voluto (non-decrescente), inoltre gli elementi alla destra di p, essendo tutti maggiori di p, occuperanno, nell'ordinamento finale, le posizioni alla destra di p; analogamente gli elementi alla sinistra di p andranno ad occupare posizioni alla sinistra di p. Pertanto non ci resta che ripetere l'algoritmo ricorsivamente sulla prima parte (elementi minori o uguali a p) e sulla terza parte (elementi maggiori di p).

L'algoritmo che si occupa di suddividere la sequenza nelle tre parti è solitamente denominato algoritmo di *distribuzione*.

```
1  Distribuzione( a[], sx, dx ) {
2    i = sx + 1; j = dx; finale = sx;
3    WHILE ( i <= j) {
4      WHILE ( i <= j && a[i] <= a[sx]) {
5        finale = i; i = i+1;
6      }
7      WHILE ( i <= j && a[j] > a[sx]) {
8        j = j-1;
9      }
10     IF ( i <= j) {
11       Scambia( a, i, j);
12     }
13   }
14   Scambia(a, sx, finale);
15   RETURN finale;
16 }
```

L'algoritmo prende in input la sottosequenza dell'array a delimitata dagli indici $sx \leqslant dx$; il pivot scelto p è l'elemento in posizione sx. La funzione scandisce la sequenza a partire dai due estremi verso il centro utilizzando l'indice i (da $sx + 1$ in poi) e l'indice j (da dx a ritroso), termina quando gli indici i e j si incrociano. Tutti gli elementi da j in poi devono risultare maggiori di p mentre tutti quelli da $sx + 1$

fino a i devo risultare minori o uguali a p. Per questo il ciclo WHILE delimitato dalle righe 4-6 termina, oltre che con l'incrocio di i e j, quando viene trovato un elemento maggiore di a[sx]. Allo stesso modo il ciclo WHILE tra le righe 7-9 termina quando viene trovato un elemento minore o uguale a a[sx]. Al termine dei due cicli WHILE appena descritti, se non è avvenuto l'incrocio, i punta ad un elemento maggiore di a[sx] e j ad uno minore o uguale a a[sx], quindi questi vengono scambiati e si prosegue. Al termine del ciclo WHILE più esterno è avvenuto l'incrocio degli indici i e j, ovvero abbiamo terminato la scansione della sottosequenza di a. L'elemento più a destra minore o uguale a a[sx] è puntato da finale (riga 5) quindi, dopo aver scambiato il pivot con a[finale] viene restituito finale.

Per stimare il numero di operazioni eseguite dall'algoritmo osserviamo che ad ogni passo del ciclo WHILE più esterno viene incrementato i o decrementato j fino a far sovrapporre i due indici. Questo ci dice che il numero di operazioni eseguite è proporzionale alla lunghezza della sequenza presa in considerazione.

L'algoritmo seguente ordina ricorsivamente la sottosequenza di a racchiusa tra gli indici sx e dx.

```
1 OrdinamentoPerDistribuzione( a[], sx, dx ) {
2   IF ( sx < dx) {
3     pos = Distribuzione(a, sx, dx);
4     OrdinamentoPerDistribuzione( a[], sx, pos-1 );
5     OrdinamentoPerDistribuzione( a[], pos+1, dx );
6   }
7 }
```

Se l'array a risulta già ordinato OrdinamentoPerDistribuzione(a, 0, n-1) genera n chiamate ricorsive ognuna delle quali invoca la funzione Distribuzione() di costo lineare in n. Questo comporta che, nel caso peggiore, l'algoritmo di ordinamento per distribuzione richiede l'esecuzione di un numero quadratico in n di operazioni elementari. Se invece si assume che la funzione Distribuzione() divida sempre la sequenza in due parti più o meno della stessa dimensione, con argomenti simili a quelli utilizzati nell'analisi dell'algoritmo di ordinamento per fusione, si dimostra che il tempo di calcolo è proporzionale a $n \log_2 n$. Si può dimostrare che questo stesso tempo di calcolo lo si ottiene in media se si sceglie il pivot a caso. Dal punto di vista pratico l'algoritmo di ordinamento per distribuzione è preferito all'algoritmo di ordinamento per fusione in quanto, sebbene il secondo garantisca prestazioni migliori nel caso peggiore, il primo ha il notevole vantaggio di ordinare *in loco*.

5.2 Ricerca

All'inizio del capitolo abbiamo affrontato il problema della ricerca di un elemento in una sequenza nel caso più generale, in questo paragrafo assumeremo che gli elementi della sequenza siano ordinati in modo non decrescente.

Se l'array `a` ordinato in modo non decrescente possiamo trarre giovamento nella ricerca di una chiave `k` al suo interno: se $k > a[i]$ per un qualche `i` possiamo restringere la ricerca dalla posizione $i+1$ in poi di `a` e, analogamente, se $k < a[i]$ sappiamo che `k` può trovarsi nelle prime `i` posizioni dell'array.

Quindi l'algoritmo confronta la chiave `k` con l'elemento che si trova in posizione centrale nell'array, `a[n/2]`, e se `k` è minore di tale elemento, ricorsivamente ripete il processo nel segmento costituito dagli elementi che precedono `a[n/2]`. Altrimenti, lo ripete in quello costituito dagli elementi che lo seguono. Il processo termina quando `k` coincide con l'elemento centrale del segmento corrente oppure il segmento diventa vuoto, ovvero `k` non è presente nell'array. Questo algoritmo è noto come *ricerca binaria*.

Tale algoritmo, ad ogni chiamata ricorsiva, restringe della metà il campo di ricerca fino, nel caso peggiore, a ridursi ad una sequenza vuota.

```
1  RicercaBinariaRicorsiva( a[], k, sx, dx ) {
2    IF (sx > dx) {
3      RETURN -1;
4    }
5    cx = (sx+dx)/2;
6    IF (k < a[cx]) {
7      RETURN RicercaBinariaRicorsiva( a, k, sx, cx-1 );
8    } ELSE IF ( k > a[cx]) {
9      RETURN RicercaBinariaRicorsiva( a, k, cx+1, dx );
10   } ELSE {
11     RETURN cx;
12   }
13 RicercaBinaria( a[], k, n ) {
14   RETURN RicercaBinariaRicorsiva( a, k, 0, n-1);
15 }
```

La funzione `RicercaBinariaRicorsiva` prende in input oltre l'array `a` e la chiave `k` anche gli indici che ne delimitano il campo di ricerca a sinistra (`sx`) e a destra (`dx`). Nel caso in cui $sx > dx$ la sequenza è vuota e se ne deduce che `k` non è in `a`. Altrimenti viene definito `cx` come l'indice intermedio tra `sx` e `dx` (riga 5), se `k` è minore dell'elemento in posizione `cx` la ricerca prosegue tra gli elementi alla sinistra di `cx` (riga 7), se `k` è maggiore di `a[cx]` la ricerca prosegue tra gli elementi a destra (riga 9), altrimenti abbiamo trovato `k` in posizione `cx` e terminiamo la ri-

cerca (riga 11). La ricorsione viene innescata dalla funzione `RicercaBinaria` che prende in input l'array `a`, la chiave `k` ed `n`, la dimensione di `a`; essa si limita ad invocare la ricorsione sulla sequenza completa (riga 14).

Nella valutazione del numero di operazioni eseguite dall'algoritmo assumiamo il caso peggiore, ovvero `k` non è in `a`. Quando il segmento è vuoto l'algoritmo esegue un numero costante `c` di operazioni, altrimenti il numero di operazioni eseguite è pari a una costante `d` più il numero di operazioni richiesto dalla ricerca della chiave in un segmento di dimensione pari alla metà di quello attuale. Pertanto, il numero totale `T(n)` di passi eseguiti su un array di `n` elementi verifica la seguente equazione:

$$\mathtt{T(n)} \equiv \begin{cases} \mathtt{c} & \text{se } \mathtt{n} \leqslant 1 \\ \mathtt{T(n/2)+d} & \text{altrimenti.} \end{cases}$$

Risolvendo abbiamo

$$\begin{aligned} \mathtt{T(n)} &\equiv \mathtt{T(n/2)+d} \\ &\equiv \mathtt{T(n/4)+2d} \\ &\equiv \mathtt{T(n/8)+3d} \\ &\cdots \\ &\equiv \mathtt{T(n/2^i)+i\cdot d.} \end{aligned}$$

Sia `k` tale che $2^{k-1} \leqslant n \leqslant 2^k$, ovvero $k-1 \leqslant \log_2 n \leqslant k$ abbiamo

$$\mathtt{T(n)} \equiv \mathtt{T(n/2^k)+k\cdot d} \equiv \mathtt{c+k\cdot d} \approx \mathtt{c+d}\log_2 \mathtt{n}.$$

Sfruttando l'ordinamento dell'array, attraverso la ricerca binaria, siamo in grado di progettare un algoritmo dalle prestazioni notevolmente migliori di quelle della ricerca sequenziale che, nel caso peggiore, ha costo proporzionale ad `n`. Si veda la Figura 5.3 dove sono messe a confronto le due funzioni.

L'algoritmo di ricerca binaria appena descritto restituisce la posizione della prima occorrenza trovata della chiave `k`. Con piccole modifiche possiamo fare in modo che l'indice restituito sia quello relativo alla occorrenza più a sinistra.

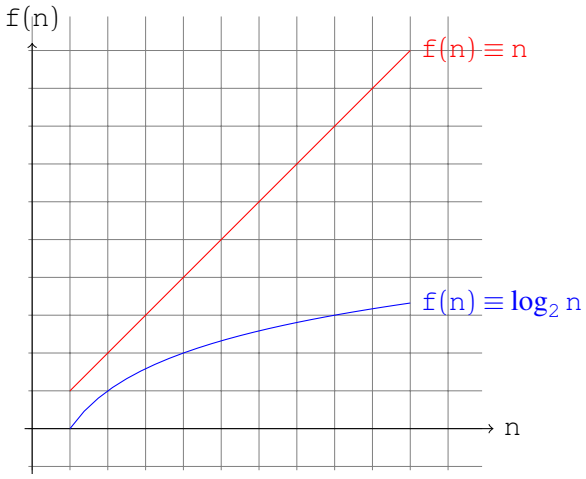


Figura 5.3 Andamento delle funzioni n e $\log_2 n$

.

```
1  RicercaBinariaRicorsiva( a[], k, sx, dx ) {
2    IF (sx > dx) {
3      RETURN -1;
4    }
5    cx = (sx+dx)/2;
6    IF (k == a[cx] && ( cx == 0 || k != a[cx-1] )) {
7      RETURN cx;
8    }
9    IF (k <= a[cx]) {
10     RETURN RicercaBinariaRicorsiva( a, k, sx, cx-1 );
11   } ELSE {
12     RETURN RicercaBinariaRicorsiva( a, k, cx+1, dx );
13   }
14 }
```

La funzione restituisce l'indice cx in cui si trova k solo se questo è quello più a sinistra (ovvero 0) oppure nella posizione che precede cx non si trova una istanza di k (riga 6), negli altri casi la ricorsione continua (riga 9). Simmetricamente possiamo fare in modo che l'indice restituito sia quello della occorrenza più a destra. Questo ci permette, usando le due varianti della ricerca binaria, di ottenere il numero complessivo di occorrenza della chiave k semplicemente sottraendo all'indice del-

la occorrenza più a destra quello della occorrenza più a sinistra. Il tutto può essere svolto eseguendo un numero di istruzioni proporzionale a $\log_2 n$.

Per quanto riguarda la stima della quantità di memoria aggiuntiva utilizzata si deve tener conto del numero massimo di istanze della funzione che sono state lanciate ma non ancora terminate. Utilizzando un argomento simile a quello utilizzato per il calcolo del numero di istruzioni si dimostra che questo numero è al più $\log_2 n$. Poiché ogni istanza della funzione utilizza una quantità costante di memoria, concludiamo che la quantità di memoria aggiuntiva è dell'ordine di $\log_2 n$. Questa quantità può diventare costante se si trasforma l'algoritmo ricorsivo nell'equivalente iterativo.

```
1  RicercaBinariaIterativa( a[], k, n ) {
2    sx = 0; dx = n-1;
3    WHILE ( sx <= dx ) {
4      cx = (sx+dx)/2;
5      IF ( k == a[cx] && ( cx == 0 || k != a[cx-1] )) {
6        RETURN cx;
7      }
8      IF (k <= a[cx]) {
9        dx = cx-1;
10     } ELSE {
11       sx = cx+1;
12     }
13   }
14   RETURN -1;
15 }
```

Quando conviene ordinare per cercare? La ricerca su sequenze ordinate si può fare in modo molto efficiente tuttavia una sequenza ordinata deve essere costruita e mantenuta e questo, come abbiamo visto, ha un costo. Ordinare una sequenza di n elementi richiede l'esecuzione di almeno n operazioni in quanto ogni elemento deve essere quantomeno letto. Ricercare un elemento in una sequenza non ordinata richiede un numero lineare di operazioni, quindi ordinare e poi cercare non è così conveniente a meno che il numero di ricerche eseguite non sia elevato. Assumiamo che l'algoritmo di ordinamento richieda $c_0 \cdot n \log_2 n$ operazioni, che la ricerca sequenziale ne richieda $c_1 \cdot n$ e la ricerca binaria $c_2 \log_2 n$, dove c_0, c_1 e c_2 sono tre costanti. Eseguire k ricerche su un array non ordinato richiede $k \cdot c_1 \cdot n$ mentre se ordinassimo l'array allora tutto costerebbe $c_0 \cdot n \log_2 n + k \cdot c_2 \log_2 n$. Il costo medio di una ricerca sarebbe

$$\frac{c_0 \cdot n \log_2 n}{k} + \cdot c_2 \log_2 n.$$

Pertanto basterebbe che $\mathtt{k}$ fosse $\mathtt{d} \cdot \log_2 \mathtt{n}$ per una opportuna costante $\mathtt{d}$ per avere un costo medio lineare in $\mathtt{n}$. Più $\mathtt{k}$ aumenta più il costo medio diminuisce, quando $\mathtt{k} \geqslant \mathtt{n}$ il costo medio diventa logaritmico.

Capitolo 6

Sequenze dinamiche

Gli array garantiscono un modo efficiente per memorizzare sequenze lineari di elementi omogenei: ovvero sequenze $a_0, a_1, \ldots, a_{n-1}$ di elementi dello stesso tipo in cui ogni elemento occupa una posizione identificata da un intero a partire da 0. Infatti, non importa quanto sia grande l'array, accedere ad un elemento sia in lettura che in scrittura richiede un numero costante di operazioni. Per convincerci di questo si deve tener conto di quanto detto nel Paragrafo 2.6: la variabile utilizzata per indicare un array contiene il riferimento a questo, ovvero contiene la posizione del primo elemento dell'array; inoltre l'array è memorizzato in posizioni contigue di memoria e i suoi elementi occupano uno spazio fissato di memoria d determinato dal tipo. Questo implica che l'elemento a[i] è memorizzato nella posizione $a + i \cdot d$. Quindi l'accesso ad un qualsiasi elemento di un array richiede una somma ed una moltiplicazione.

Nei linguaggi di programmazione tradizionali, come ad esempio il C, la dimensione dell'array deve essere stabilita una volte per tutte al momento della definizione dell'array quindi, a meno di utilizzare escamotage, non è possibile eliminare o inserire elementi nella sequenza. Al contrario, i linguaggi di programmazione di più recente concezione, come Python, mettono a disposizioni alcune strutture dati dinamiche che permettono una notevole libertà con poco sforzo.

In questo capitolo vedremo come rimediare alla mancanza dei linguaggi tradizionali mostrando come costruire strutture dati che hanno forme di dinamismo analoghe a quelle fornite dai linguaggi di programmazione più moderni. Questo ci permetterà anche di capire cosa c'è dietro alcuni costrutti forniti da questi ultimi linguaggi.

6.1 Array dinamici

Il punto debole degli array tradizionali è la loro staticità: la loro dimensione deve essere decisa a priori. Quando questa non è nota si rimedia sovradimensionando l'array,

in questo caso si corre il rischio di sovrastimare troppo e sprecare troppa memoria. Infine, anche quando la stima è corretta, può succedere che il numero di elementi effettivamente presenti nell'array abbia una grande variabilità e quindi la maggior parte del tempo la memoria allocata all'array potrebbe essere sotto-utilizzata.

Alcuni linguaggi di programmazione permettono di aggiungere e cancellare elementi dalla coda dell'array. Lo stratagemma utilizzato è quello di partire da un array di una certa *capacità* iniziale: questa definisce il numero massimo di elementi che l'array può contenere. Quando si aggiunge un nuovo elemento lo si inserisce nella prima posizione libera dell'array. Se non ci sono posizioni libere, quindi l'array risulta essere pieno, esso viene sostituito da un altro array più grande sul quale verranno copiati tutti gli elementi del primo. Viceversa, potendo eliminare l'ultimo elemento inserito nell'array, questo potrebbe diventare sotto-utilizzato; in tal caso l'array verrà rimpiazzato da uno più piccolo.

Il ridimensionamento dell'array è una operazione che comporta, oltre la creazione di un nuovo array, anche la copia di tutti gli elementi dal vecchio: il costo di questa operazione è proporzionale al numero `n` di elementi del vecchio array da copiare nel nuovo. Quindi non ci possiamo permettere di eseguire un ridimensionamento ad ogni inserimento o cancellazione. Però se avessimo la garanzia che questa operazione venga eseguita almeno dopo `n` inserimenti o cancellazioni senza ridimensionamento (che richiedono un numero costante di operazioni) allora il costo dell'operazione di ridimensionamento, se viene visto come spalmato tra le `n` operazioni che separano due ridimensionamenti successivi, ha costo virtualmente costante.

Un *array dinamico* è definito da una tupla composta da tre elementi: un array `a`, un intero `c` che ne definisce la capacità (è la dimensione di `a`) ed un intero $\texttt{n} \leqslant \texttt{c}$ che indica il numero di elementi effettivamente contenuti nell'array dinamico.

```
VECTOR  {
  a[];
  c;
  n;
}
```

Dato un array dinamico `v` di capacità `c` e dimensione `n`, se $\texttt{0} \leqslant \texttt{i} < \texttt{n} \leqslant \texttt{c}$ allora `v.a[i]` è l'elemento in posizione `i` di `v`. Oltre a poter accedere agli elementi di `v`, possiamo aggiungere un nuovo elemento in fondo a `v` e rimuovere l'ultimo elemento di `v`. Se `v` ha almeno una posizione libera ($\texttt{n} < \texttt{c}$), l'operazione `Aggiungi` inserisce un nuovo elemento come ultimo elemento dell'array dinamico; altrimenti, se l'array dinamico è pieno ($\texttt{n} = \texttt{c}$), viene creato un nuovo array di dimensione $\texttt{c}' \equiv \texttt{2n}$, tutti gli elementi del vecchio array vengono copiati nel nuovo e, infine, viene aggiunto il nuovo elemento come ultimo elemento del nuovo array. Viceversa, l'operazione `Elimina`, elimina l'ultimo elemento dell'array dinamico, ovvero `n` viene decre-

mentato di uno facendo si che l'ultimo elemento venga ignorato. Se n è uguale a c/4 (c ≡ 4n), viene creato un nuovo array di dimensione c′ ≡ c/2 (ovvero c′ ≡ 2n) che andrà a contenere tutti gli elementi dell'array dinamico. Questa operazione ha lo scopo di rendere limitata la quantità di memoria non utilizzata.

```
1  Aggiungi( v, k ) {
2    IF ( v.n < v.c) {
3      v.a[v.n] = k; v.n = v.n+1;
4    } ELSE {
5      b = NuovoArray( 2*v.c );
6      FOR ( i = 0 ;  i < v.n ;  i = i+1) {
7        b[i] = v.a[i];
8      }
9      b[v.n] = k;
10     v.a = b; v.c = 2*v.c; v.n = v.n+1;
11   }
12   RETURN v;
13 }
```

La funzione Aggiungi aggiunge l'elemento k in coda all'array dinamico v. Se l'array v.a ha ancora delle posizioni libere k viene inserito nella prima posizione libera indicata proprio da v.n (righe 2-4). Altrimenti viene creato un nuovo array b di dimensione doppia rispetto a v.a (riga 5), tutti gli elementi di v.a vengono copiati nelle prime posizioni di b (righe 6-8); il nuovo elemento k viene copiato nella prima posizione libera (riga 9) e i campi di v vengono aggiornati di conseguenza (riga 10).

La funzione Elimina cancella l'ultimo elemento dall'array dinamico in input che si assume non vuoto.

```
1  Elimina( v ) {
2    v.n = v.n-1;
3    IF ( v.n <= v.c/4) {
4      b = NuovoArray( v.c/2 );
5      FOR ( i = 0 ;  i < v.n ;  i = i+1) {
6        b[i] = v.a[i];
7      }
8      v.a = b; v.c = v.c/2;
9    }
10   RETURN v;
11 }
```

L'ultimo elemento di v viene eliminato semplicemente ignorandolo (riga 2). Se l'occupazione di v scende al di sotto di un quarto della sua capacità (riga 3) quest'ultima viene dimezzata. Quindi viene creato un nuovo array b di dimensione v.c/2 (ri-

	Lettura	Inserimento	Cancellazione
Array	costante	-	-
Array dinamici	costante	costante in media (solo da un estremo)	
Liste	lineare		

Tabella 6.1 Costi delle operazioni su sequenze lineari in base all'implementazione.

ga 4) sul quale vengono riversati tutti gli elementi in `v.a` (righe 5-7). Infine vengono aggiornati i campi di `v` (riga 8).

Una operazione `Aggiungi` o `Elimina` può richiedere l'esecuzione di un numero proporzionale a `v.n` operazioni, diciamo $\mathtt{d} \cdot \mathtt{n}$ dove `d` è una costante. Tuttavia dopo ogni operazione "costosa", per esempio dopo una `Aggiungi` che raddoppia la capacità di `v`, devono seguire un numero proporzionale a `v.n` di operazioni "non costose" che non ne alterano la capacità e che quindi richiedono un numero $\mathtt{d}'$ costante di operazioni. Infatti dopo la `Aggiungi` "costosa" $\mathtt{v.c} \equiv 2\mathtt{v.n}$, quindi se la prossima operazione "costosa" fosse una `Aggiungi` dovrebbero seguire almeno `n` operazioni `Aggiungi` "non costose"; invece se fosse una `Elimina` dovrebbero essere seguite $\mathtt{n}/2$ operazioni `Elimina` "non costose". Quindi `n` operazioni consecutive hanno un costo al più $\mathtt{d} \cdot \mathtt{n} + (\mathtt{n}-1)\mathtt{d}'$, ovvero il costo medio di ogni operazione è al massimo $\mathtt{d} + \mathtt{d}'$ cioè costante.

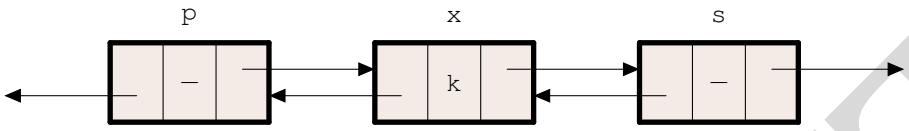
6.2 Liste

Le liste permettono di creare sequenze lineari con un maggiore livello di dinamicità rispetto ai `vector`. Infatti sono state introdotte per permettere l'inserimento e la cancellazione di elementi da qualsiasi posizione della sequenza. Come vedremo, questo implicherà un maggiore costo computazionale. In particolare il costo per inserire o cancellare un elemento nell'`i`-esima posizione della sequenza ha costo proporzionale ad `i`. La Tabella 6.1 mette a confronto i costi computazionali delle operazioni di lettura, inserimento e cancellazione su sequenze dinamiche in base alla loro implementazione. Risulta evidente che la maggior flessibilità viene pagata in termini di maggior costo computazione.

Come è stato già osservato, i punti deboli degli array ma anche i punti di forza derivano dal fatto che i suoi elementi sono memorizzati in posizioni di memoria contigue. Quindi l'inserimento o la cancellazione di un elemento richiederebbe la modifica di una grossa parte dell'array stesso. Con le liste il problema viene risolto permettendo ai sui elementi di risiedere in posizione qualsiasi della memoria. Ogni

elemento della lista, detto *nodo*, a prescindere da dove esso sia memorizzato, contiene informazioni sulla posizione che esso occupa nella lista, più precisamente contiene un riferimento all'elemento che lo segue e, facoltativamente, all'elemento che lo precede. Conoscendo la posizione in memoria del primo elemento della lista possiamo, grazie ai riferimenti contenuti nei nodi, accede a tutti gli altri elementi.

Nella figura che segue è mostrato schematicamente come il nodo `x` della lista contenente la chiave `k` sia collegato al nodo che lo precede `p` e quello che lo segue `s`. A loro volta i nodi `p` e `s` sono collegati a `x` e ad altri due nodi.



Ogni nodo è identificato con la posizione che occupa in memoria, ovvero dal suo indirizzo di memoria o puntatore. Quindi i riferimenti, mostrati nella figura con le frecce, sono a loro volta indirizzi di memoria di altri nodi. Un nodo della lista viene rappresentato da una tupla composta da tre campi, uno per la chiave e due per i riferimenti alle tuple che identificano i nodi collegati.

```
NODO  {
  chiave;
  prec;
  succ;
}
```

Il campo `chiave` contiene l'informazione vera e propria mentre i campi `succ` e `prec` contengono rispettivamente il riferimento al nodo successivo e quello precedente. Nella figura precedente il campo `prec` è rappresentato dal rettangolo a sinistra, il campo `chiave` da quello centrale ed il campo `succ` da quello a destra.

Una lista rappresenta una sequenza lineare che ha un primo elemento ed un ultimo elemento. Se la lista non è vuota il campo *prec* del nodo corrispondente al primo elemento della è vuoto (`NULL`) e lo stesso dicasi per il campo `succ` dell'ultimo elemento della lista. Ogni lista viene identificata con il riferimento al primo elemento della sequenza, quindi una lista `a` viene identificata con un puntatore al primo nodo della lista, se la lista `a` è vuota il puntatore è `NULL`. Se `p` è un puntatore ad un nodo, con `p->chiave`, `p->succ` e `p->prec` indichiamo rispettivamente il campo `chiave`, `succ` e `prec` della tupla a cui fa riferimento (o *puntata da*) `p`. In particolare, se `p` è un puntatore ad una tupla, l'operatore `->` permette di accedere ai campi della tupla puntata da `p`.

Esempio 6.1. La funzione che segue calcola e restituisce la dimensione della lista `a`. Il parametro della funzione è il riferimento al primo elemento della lista.

```
1  Lunghezza( a ) {
2    n = 0;
3    FOR ( p = a ;  p != NULL ;  p= p->succ ) {
4      n = n+1;
5    }
6    RETURN n;
7  }
```

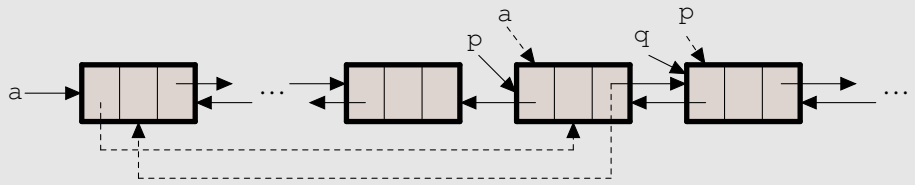
Partendo dal primo nodo della lista si raggiunge l'ultimo passando da un nodo a quello successivo e nel contempo viene incrementata una variabile intera `n` inizializzata a `0`. Quando il puntatore `p` equivale a `NULL` si è raggiunta la fine della lista ed il valore contenuto nella variabile `n` è proprio la lunghezza della lista. Si osservi che se la lista è vuota `a` vale `NULL` e viene restituito `0`.

Esempio 6.2. La funzione `InvertiLista()` inverte l'ordine degli elementi che appaiono nella sequenza di input. La funzione prende in input la lista `a` attraverso il puntatore al suo primo elemento e inverte la lista utilizzando soltanto la manipolazione dei puntatori.

```
1  InvertiLista( a ) {
2    IF ( a == NULL || a->succ == NULL ) {
3      RETURN a;
4    }
5    p = a->succ; a->succ = NULL;
6    WHILE ( p != NULL ) {
7      q = p->succ;
8      p->prec = NULL; p->succ = a;
9      a->prec = p;
10     a = p;
11     p = q;
12   }
13   RETURN a;
14 }
```

Facciamo riferimento alla figura in basso. Supponiamo che i primi `k` elementi di `a` siano stati già invertiti, quindi la lista identificata da `a` contiene i primi `k` elementi della lista originale

in ordine inverso. La variabile `p` punta al `k + 1`-esimo elemento della lista originale. Le istruzioni del ciclo WHILE della funzione hanno lo scopo di posizionare in testa alla lista l'elemento puntato da `p`. Le modifiche apportate ai puntatori dall'esecuzione del ciclo WHILE sono rappresentate con le linee tratteggiate.



La parte iniziale della funzione tratta i casi in cui l'inversione di `a` è banale, ovvero quando la lista è vuota o composta da un solo elemento. Se invece la lista contiene almeno due elementi `p` viene inizializzato al secondo elemento e il campo `succ` del primo elemento attuale, poiché diventerà l'ultimo, viene aggiornato di conseguenza.

6.2.1 Ricerca

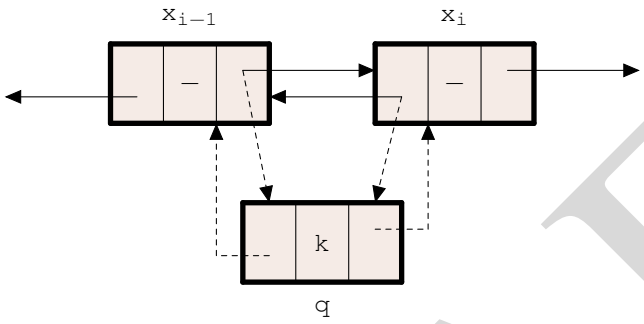
L'operazione di ricerca in una lista permette di accedere all'elemento in una data posizione nella sequenza: in particolare, data la lista `a` ed un intero `i`, la funzione `Ricerca(a, i)` restituisce il nodo in posizione `i` della lista, ovvero quello contenente l'elemento della sequenza in posizione `i`. Se l'elemento in posizione `i` non esiste (la lista è più breve), la funzione restituisce `NULL`.

```
1 Ricerca( a, i ) {
2   p = a; j = 0;
3   WHILE ( j < i && p != NULL ) {
4     p = p->succ; j = j+1;
5   }
6   RETURN p;
7 }
```

La funzione somiglia a quella mostrata nell'Esempio 6.1. Essa scorre la lista `a` dal suo primo elemento utilizzando il puntatore `p` inizializzato al valore di `a`. Ogni volta che il puntatore viene spostato di una posizione (`p = p->succ`) viene incrementata la variabile `j` inizializzata a `0`. Questo processo termina quando `j` equivale a `i` oppure quando il puntatore `p` raggiunge il termine della lista (`p` ≡ `NULL`). In ogni caso si restituisce il valore di `p`. Il costo computazionale è proporzionale al parametro `i` pertanto, nel caso peggiore, lineare nella lunghezza della sequenza.

6.2.2 Inserimento

L'inserimento di un nuovo elemento nella posizione `i` di una sequenza rappresentata da una lista richiede la modifica dei riferimenti del nodo in posizione $\texttt{i}-1$ e, se esiste, del nodo in posizione `i`.



Facendo riferimento alla figura, vogliamo inserire un nuovo nodo `q` contenente la chiave `k` nella posizione `i` della lista. Con x_{i-1} e x_i indichiamo i nodi in posizione $\texttt{i}-1$ e `i` della lista. Dopo l'inserimento i collegamenti tra x_{i-1} e x_i saranno sostituiti da quelli rappresentati dalle linee tratteggiate della figura. In particolare

```
q->prec = x_{i-1};
q->succ = x_i;
x_i->prec = q;
x_{i-1}->succ = q;
```

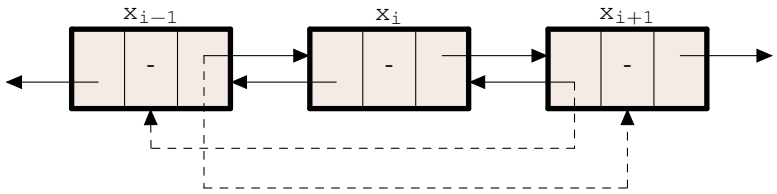
Le osservazioni appena fatte ci consentono di progettare la funzione `Inserisci()` che esegue l'inserimento in posizione `i` di un nuovo nodo con chiave `k` nella lista `a`. La funzione restituisce il puntatore che identifica la lista modificata.

```
1  Inserisci( a, i, k ) {
2    IF ( i == 0 ) {
3      q = NuovoNodo();
4      q->chiave = k; q->succ = a; q->prec = NULL;
5      a->prec = q; a = q;
6    } ELSE {
7      p = Ricerca( a, i-1);
8      IF ( p != NULL ) {
9        q = NuovoNodo();
10       q->chiave = k; q->succ = p->succ; q->prec = p;
11       IF ( p->succ != NULL ) {
12         p->succ->prec = q;
13       }
14       p->succ = q;
15     }
16   }
17   RETURN a;
18 }
```

La funzione invoca la funzione `NuovoNodo()` che crea una tupla tipo `NODO` e ne restituisce l'indirizzo. Se l'elemento deve essere inserito in una posizione diversa dalla prima (righe 7-15) si raggiunge con il puntatore `p` il nodo in posizione $i-1$ e, se questo esiste, viene inserito il nuovo nodo `q` tenendo conto che x_{i-1} è `p` e x_i è `p->succ`. Se invece il nodo deve essere inserito in posizione `0` allora vengono eseguite le righe 3-5. In questo caso il campo `prec` di `q` è `NULL` in quanto `q` diventerà primo nodo della lista. Invece il vecchio primo nodo della lista (puntato da `a`) diventerà quello in posizione `1`, pertanto dovrà essere puntato da `q->succ` ed il suo campo `prec` dovrà puntare a `q` (`a->prec = q`). La funzione restituisce `a` ovvero la lista modificata attraverso il puntatore al suo primo elemento. Si osservi che il valore di `a` è modificato soltanto se l'inserimento viene eseguito in posizione `0`.

6.2.3 Cancellazione

Nel caso in cui si richiede la cancellazione dell'elemento in posizione $i > 0$ dalla lista `a` devono essere modificati i puntatori `succ` e `prec` dei nodi rispettivamente in posizione $i-1$ e $i+1$. La configurazione risultante è mostra nella figura dove i nuovi valori dei puntatori sono mostrati da linee tratteggiate.



La funzione `Cancella()` oltre a gestire il caso più generale mostrato in figura deve accertare che la lista non sia vuota. Inoltre, come nel caso della funzione `Inserisci()`, il caso $\texttt{i} \equiv 0$ deve essere trattato in modo separato in quanto richiede la modifica del puntatore `a` al primo elemento della lista.

```
1  Cancella( a, i ) {
2    IF ( a == NULL ) {
3      RETURN a;
4    }
5    IF ( i == 0 ) {
6      q = a;
7      a = q->succ;
8      a->prec = NULL;
9    } ELSE {
10     p = Ricerca( a, i-1);
11     IF ( p == NULL || p->succ == NULL ) {
12       RETURN a;
13     }
14     q = p->succ;
15     p->succ = q->succ;
16     IF ( p->succ != NULL ) {
17       q->succ->prec = p;
18     }
19   }
20   free( q );
21   RETURN a;
22 }
```

Il puntatore `q` viene assegnato all'elemento da eliminare, quindi se `i` è `0` viene assegnato ad `a` altrimenti a `p->succ` dove `p` punta al nodo in posizione $\texttt{i}-1$ (riga 10). Dopo aver accertato che l'elemento da cancellare sia presente (righe 11-13) si passa alla ri-definizione dei puntatori in `p`, in `q` e, eventualmente, in `q->succ` (righe 14-18). Prima di terminare restituendo la lista modificata (riga 21) viene invocata la funzione `free()`: questa operazione libera la memoria occupata dal nodo puntato da `q`, ovvero fa in modo che la memoria occupata dal nodo torni a far parte della memoria libera che potrà essere riutilizzata nelle successive operazioni. In alcuni

casi questa operazione di riciclo è lasciata ad un meccanismo automatico chiamato *garbage collector* che solleva il programmatore da questo compito.

Si osservi infine che sia nell'operazione di inserimento che in quella di cancellazione si distinguono sempre due casi: inserimento o cancellazione in testa ed inserimento o cancellazione altrove. Quest'ultima operazione si scompone in due operazioni: ricerca del nodo precedente a quello su cui agire; inserire in posizione uno a partire dal nodo cercato o cancellazione del nodo in posizione uno a partire dal nodo trovato.

Esercizio svolto 6.1. Si progettino le funzioni `InserisciZero()` e `InserisciUno()` che, data una lista `a` ed una chiave `k` inseriscono in `a`, in posizione 0 ed 1 rispettivamente, un nuovo nodo con chiave `k`.

Analogamente si progettino le funzioni `CancellaZero()` e `CancellaUno()` che eliminano dalla lista di input, rispettivamente, il primo ed il secondo elemento.

Infine si riscrivano le funzioni `Inserisci()` e `Cancella()` facendo uso delle nuove funzioni e della funzione `Ricerca()`.

Soluzione. È sufficiente estrapolare dalle funzioni `Inserisci()` e `Cancella()` le parti che definiscono le funzioni da progettare. Si tratta di un semplice esercizio di riscrittura.

Esempio 6.3. La funzione `ListaPalindroma()` verifica se la sequenza codificata nella lista `a` in input sia palindroma o meno. In particolare se è palindroma restituisce TRUE altrimenti restituisce FALSE.

```
1  ListaPalindroma( a ) {
2    IF ( a == NULL) {
3      RETURN TRUE;
4    }
5    q = a;
6    WHILE ( q->succ != NULL ) {
7      q = q->succ;
8    }
9    p = a;
10   WHILE ( p->chiave == q->chiave && q != p && p->prec !=
     q ) {
11     p = p->succ; q = q->prec;
12   }
13   IF ( p->chiave != q->chiave ) {
14     RETURN FALSE;
15   } ELSE {
16     RETURN TRUE;
17   }
18 }
```

La lista vuota è palindroma per definizione, questo spiega le righe 2-4. Se questa non è vuota, il puntatore `q` viene spostato sull'ultimo elemento della lista ed il puntatore `p` fa riferimento al primo elemento (righe 5-9). Tra le righe 10 e 12 viene verificato che le chiavi contenute nei nodi puntati da `p` e `q` siano identiche, se così fosse `p` viene spostato verso gli elementi successivi della lista mentre `q` verso quelli precedenti. Il procedimento termina quando le chiavi nei nodi puntati da `p` e `q` divergono oppure quando `p` e `q` raggiungono lo stesso nodo (se la lista ha lunghezza dispari) oppure quando i puntatori `p` e `q` si incrociano (se la lista ha lunghezza pari). Nel primo caso si restituisce FALSE, negli altri due casi TRUE (righe 13-17)

Capitolo 7

Array associativi

Un *array associativo* oppure *mappa* oppure *dizionario* è una struttura dati astratta che contiene un insieme di coppie $(\mathtt{k},\mathtt{v})$ dove il primo elemento della coppia è detto *chiave* ed il secondo *valore*. La chiave è univoca, vale a dire che non esistono due coppie con la stessa chiave. Se l'array associativo contiene la coppia $(\mathtt{k},\mathtt{v})$ si dice che alla chiave $\mathtt{k}$ è associato il valore $\mathtt{v}$. Su un array associativo sono possibili le operazioni di inserimento di una coppia, ricerca di una coppia in base alla chiave e cancellazione della coppia contenente una data chiave.

7.1 Le operazioni di base

Dato un array associativo $\mathtt{d}$ ed una coppia $(\mathtt{k},\mathtt{v})$, l'operazione di *inserimento* verifica se è presente in $\mathtt{d}$ una coppia $(\mathtt{k},\mathtt{v}')$ con la stessa chiave $\mathtt{k}$: nel caso in cui non sia presente viene aggiunto a $\mathtt{d}$ il nuovo elemento $(\mathtt{k},\mathtt{v})$; altrimenti la coppia $(\mathtt{k},\mathtt{v}')$ viene sostituita con la coppia $(\mathtt{k},\mathtt{v})$.

L'operazione di *ricerca* restituisce il valore associato ad una determinata chiave. In particolare questa prende in input un dizionario $\mathtt{d}$ ed una chiave $\mathtt{k}$, se $\mathtt{k}$ è associato al valore $\mathtt{v}$, ovvero se la coppia $(\mathtt{k},\mathtt{v})$ è in $\mathtt{d}$, viene restituito $\mathtt{v}$.

Infine, data una chiave $\mathtt{k}$, l'operazione di *cancellazione* elimina da $\mathtt{d}$ l'eventuale coppia $(\mathtt{k},\mathtt{v})$.

7.2 Funzioni hash

Dalla descrizione astratta delle operazioni si individua una certa affinità tra le chiavi degli array associativi e gli indici degli array tradizionali. Anzi, gli array associativi possono essere visti come una generalizzazione degli array tradizionali in cui l'indice non è altro che la chiave. Alcuni linguaggi di programmazione, come Python, inclu-

dono gli array associativi tra i loro tipi di dato e su questi si opera in maniera simile agli array: per esempio l'istruzione

```
d[k] = v
```

è l'operazione in Python di inserimento della coppia `(k,v)` in un dizionario `d`.

Nel caso in cui il linguaggio di programmazione non includa gli array associativi potremmo essere tentati dall'usare gli array standard trasformando la chiave `k` in un numero intero, questo è sempre possibile basta considerare la rappresentazione binaria di `k`. Tuttavia, poiché la chiave `k` può essere qualsiasi dovremmo prevedere array di dimensione enorme rispetto all'effettivo utilizzo.

Questo sistema può essere corretto trasformando la chiave `k` in un intero più piccolo: se imponiamo per l'array una dimensione `m` la chiave `k` deve essere trasformata in un intero da `0` a $\texttt{m}-1$. Le funzioni *hash* si occupano di questo ovvero data una chiave `k` ed un intero `m` restituiscono un intero in $\{\texttt{0},\ldots,\texttt{m}-1\}$.

Un esempio di semplice funzione hash è il modulo che restituisce il resto della divisione della rappresentazione intera di `k` per `m`. Un altro esempio è la funzione di or esclusivo (o *xor*) bit-a-bit che dividere la chiave `k` in blocchi di lunghezza fissata, ne calcola lo xor bit-a-bit e del valore ottenuto restituisce il resto della divisione per `m`.

Una funzione hash `h` trasforma interi potenzialmente molto grandi in interi racchiusi all'interno di un intervallo limitato, per questo può capitare che per due chiavi diverse tra loro $\texttt{k}_1$ e $\texttt{k}_2$ si abbia $\texttt{h}(\texttt{k}_1) \equiv \texttt{h}(\texttt{k}_2)$; in questo caso siamo in presenza di una *collisione*. Tuttavia, per la natura deterministica delle funzioni hash, se $\texttt{h}(\texttt{k}_1) \neq \texttt{h}(\texttt{k}_2)$ allora necessariamente $\texttt{k}_1$ e $\texttt{k}_2$ sono diverse.

Quest'ultima proprietà viene utilizza per testare l'integrità dei documenti scaricati da internet. Per dare il modo all'utente di verificare che non ci siano stati errori di download, il gestore del documento `f` pubblica l'*impronta digitale* di `f` che non è altro che un intero calcolato con una funzione hash chiamata MD5. L'utente scarica la sua copia di `f` e ne calcola l'MD5, se questo non corrisponde a quello pubblicato dal gestore c'è stato un problema nel download e la copia è corrotta. Nel caso contrario, ovvero se gli MD5 coincidono, in linea teorica non potremmo concludere nulla in quanto potremmo essere alle prese con una collisione. Tuttavia l'MD5 è una "buona" funzione hash e pertanto se gli MD5 coincidono allora possiamo essere abbastanza fiduciosi che anche i documenti coincidono.

Una buona funzione hash è in grado di "amplificare" le differenze tra due chiavi nel senso che gli interi restituiti sono molto diversi anche se le chiavi differiscono di pochi bit. Inoltre una buona funzione hash distribuisce uniformemente le chiavi nel suo codominio. Quindi una buona funzione hash è quello che serve per trasformare chiavi in indici di un array in modo da permetterci di implementare gli array associativi.

Anche la funzione modulo gode di buone proprietà di distribuzione delle chiavi tra gli indici a patto che m sia un numero primo. Se non lo fosse, ovvero se $m \equiv ab$, per tutte le chiavi $k \equiv ax$ si avrebbe $k\%\, m \equiv x\%a$. Ovvero tutte queste chiavi sarebbero mappate in $\{0,\ldots,a-1\}$ piuttosto che in $\{0,\ldots,m-1\}$.

7.3 Implementazione

L'implementazione che mostreremo gestisce le collisioni tra chiavi utilizzando le *liste di trabocco*. L'array associativo, che chiameremo *mappa*, è una coppia così definita:

```
mappa {
  tabella[];
  dim;
}
```

dove dim è la dimensione di tabella e tabella[i] è la lista concatenata contenente le coppie (k,v) tali che Hash(k) ≡ i. Ovvero tutte le coppie che collidono si trovano sulla stessa lista. Le coppie (k,v) sono memorizzate nel campo chiave dei nodi delle liste. Quindi se p è un puntatore ad un nodo di una lista di trabocco allora k e v sono memorizzate rispettivamente in p−>chiave.k e p−>chiave.v.

Passiamo alle implementazioni delle singole funzioni di gestione della mappe partendo dalla funzione che crea un nuovo array associativo contente una tabella capace di ospitare m liste di trabocco vuote.

```
CreaMappa( m ) {
  mappa d;
  d.tabella = NuovoArray( m );
  d.dim = m;
  FOR (i = 0; i < m; i = i+1) {
    d.tabella[i] = NULL;
  }
  RETURN d;
}
```

La funzione di ricerca della coppia con chiave k scorre la lista d.tabella[Hash(k)] e se trova la coppia (k,v) restituisce v altrimenti restituisce NULL.

```
RicercaInMappa( d, k ) {
  h = Hash( k, d.dim );
  p = RicercaChiave(d.tabella[h], k);
  IF ( p != NULL ) {
    RETURN p->chiave.k;
  } ELSE {
    RETURN NULL;
  }
}
```

Questa funzione utilizza la funzione `RicercaChiave()` che data una lista `a` ed un valore `k` restituisce il nodo di `a` in cui compare la coppia `(k,v)`. Se tale nodo non esiste viene restituito `NULL`.

```
RicercaChiave(a, k) {
  p = a;
  WHILE ( p != NULL && k != p->chiave.k ) {
    p = p->succ;
  }
  RETURN p;
}
```

Passiamo alla descrivzione dell'operazione di inserimento di una coppia `(k,v)` nella mappa `d`.

```
InserisciInMappa( d, (k,v) ) {
  h = Hash( e.chiave, d.dim );
  p = RicercaChiave( d.tabella[h], k );
  IF ( p == NULL ) {
    d.tabella[h] = Inserisci(d.tabella[h], 0, (k,v));
  } ELSE {
    p->chiave.v = v;
  }
  RETURN d;
}
```

Innanzi tutto si ricerca in `d.tabella[Hash(k)]` un nodo contenente la chiave `k`, se questo non esiste viene inserito un nuovo nodo contenete la coppia `(k,v)` in testa alla lista `d.tabella[Hash(k)]`; altrimenti al campo `chiave.v` di tale nodo viene assegnato il nuovo valore `v`.

Infine la funzione `CancellaDaMappa()` elimina da `d.tabella[Hash(k)]` l'eventuale nodo contenente una coppia con chiave `k`.

```
CancellaDaMappa( d, k ) {
  h = Hash( k, d.dim );
  p = RicercaChiave( d.tabella[h], k );
  IF ( p == NULL ) {
    RETURN d;
  }
  IF ( p == d.tabella[h] ) {
    d.tabella[h] = Cancella( d.tabella[h], 0);
  } ELSE {
    p->prec = Cancella( p->prec, 1);
  }
  RETURN d;
}
```

Si esegue la ricerca nella lista `a = d.tabella[Hash(k)]` del nodo da eliminare, se questo viene trovato si distinguono due casi: il nodo da eliminare `p` è il primo della lista oppure è in una posizione successiva. Nel primo caso si elimina il nodo in posizione 0 della lista `a` altrimenti viene eliminato il nodo in posizione 1 della lista che inizia con `p->prec`.

7.4 Costo delle operazioni su array associativi

Il numero di operazioni eseguite dalle funzioni descritte nel paragrafo precedente, tranne che per la funzione `CreaMappa()`, è lineare nella lunghezza della lista in cui si interviene. Se il numero complessivo di coppie è `n` e se la dimensione della tabella è `m`, la lunghezza media di ogni lista è `n/m`. Facendo in modo che `m` sia circa `2n` (questo è possibile ricorrendo agli array a dinamici descritti nel Paragrafo 6.1) avremmo liste la cui lunghezza media è costante e tale sarebbe anche il costo medio delle funzioni di inserimento, cancellazione e ricerca su array associativi. La funzione `CreaMappa()` ha costo lineare in `m` e quindi in `n`.