# ISTITUTO NAZIONALE DI FISICA NUCLEARE

Sezione di Catania

---

**INFN/TC-87/5**

21 Maggio 1987

C. Ripepi:

**PROGETTAZIONE FORMALIZZAZIONE ED ANALISI COMPARATIVA DI ALGORITMI DI ORDINAMENTO INTERNO DI COMPLESSITA' OTTIMALE**

**Servizio Documentazione**
dei Laboratori Nazionali di Frascati

ISTITUTO NAZIONALE DI FISICA NUCLEARE
Sezione di Catania

INFN/TC-87/5
21 Maggio 1987

# PROGETTAZIONE FORMALIZZAZIONE ED ANALISI COMPARATIVA DI ALGORITMI DI ORDINAMENTO INTERNO DI COMPLESSITÀ OTTIMALE.

Clara Ripepi
INFN - Sezione di Catania, Dipartimento di Fisica - Università di Catania.

## SOMMARIO

Lo studio del problema dell'ordinamento interno e la descrizione di tre algoritmi di complessità ottimale basati su confronti, dei quali si forniscono le codifiche in PASCAL ed in FORTRAN e si analizza il comportamento nel caso pessimo e nel caso medio, offrono lo spunto per un'introduzione critica alla vasta problematica connessa alla progettazione di algoritmi ed alla valutazione della loro efficienza. In particolare sono illustrati a grandi linee il metodo del *divide et impera*, le tecniche di analisi della complessità computazionale asintotica, l'uso di alberi di confronto per la determinazione di un limite inferiore di complessità nel problema dell'ordinamento e per l'indicazione dei possibili comportamenti di un algoritmo in relazione alle diverse configurazioni di input, la tecnica di trasformazione di un algoritmo ricorsivo in uno equivalente che usi solo l'iterazione.

# DESIGN AND COMPARATIVE ANALYSIS OF EFFICIENT SORTING ALGORITHMS.

## ABSTRACT

The internal sorting problem is studied. Three efficient algorithmic solutions, using element comparisons, are presented and coded in PASCAL and FORTRAN languages. The analysis of their expected behavior both in the worst and in the average case gives the opportunity for a critical introduction to the problems related with the design of algorithms and with the evaluation of their efficiency. The subjects considered include the *divide-and-conquer* strategy, the asymptotic computational complexity, the use of comparison trees to get a lower bound on the sorting problem and for modeling the behavior of sorting algorithms, the translation of a recursive algorithm into an iterative one.

## 1. - INTRODUZIONE.

Partendo dall'analisi di un problema particolare ma di importanza intrinseca fondamentale, l'ordinamento di dati, ci proponiamo da una parte di presentare alcuni tra i più efficienti metodi di risoluzione dall'altra di accennare alla vasta problematica connessa alla costruzione di algoritmi ed alla valutazione della loro efficienza.

Risolvere un problema mediante un elaboratore elettronico comporta la progettazione o comunque la ricerca di un algoritmo, cioè di una successione finita di azioni elementari, univocamente definite ed "effettivamente" eseguibili da una macchina in tempo finito, che conducano ad una soluzione "corretta" del problema. Lo spazio di memoria occupato ed il tempo impiegato per eseguire i passi che costituiscono l'algoritmo ne determinano il grado di "efficienza" poiché da essi direttamente dipende il costo di esecuzione di ogni programma di calcolo.

L'analisi di una procedura di calcolo è sovente considerata un inutile fastidio. Perfino nell'ambiente scientifico, in cui i problemi affrontati hanno un preponderante aspetto algoritmico, l'utente medio del calcolatore, che spesso ha imparato a programmare più per necessità che per vocazione, non ha il tempo nè forse la voglia di confrontare diverse procedure per scegliere la più efficiente, o comunque la più adatta alle sue esigenze, e si accontenta generalmente della prima soluzione corretta che riesce a determinare.

Ma esistono regole generali per la costruzione di un buon algoritmo e per la dimostrazione della sua correttezza? È possibile analizzare matematicamente la qualità intrinseca di un algoritmo, studiarne il comportamento al crescere della dimensione del problema o al variare delle configurazioni iniziali dei dati di input? In che senso si può provare che una certa procedura è più efficiente di un'altra o che è la più efficiente possibile? È preferibile un algoritmo ricorsivo o un algoritmo iterativo?

Nel descrivere e valutare le procedure di ordinamento interno che presenteremo avremo modo di accennare a tali questioni, che rientrano nel campo di studio di una nuova affascinante branca dell'informatica, sviluppatasi nell'ultimo decennio, dapprima in modo informale e frammentario, in seguito secondo linee evolutive più sistematiche. Affrontando i problemi legati alla progettazione di algoritmi ed alla valutazione delle loro prestazioni la neodisciplina, tuttora in piena evoluzione, elabora tecniche generali di risoluzione di famiglie di problemi riconducibili a determinati modelli logici, cercando di formulare principi base utilizzabili nella dimostrazione di correttezza di un dato algoritmo e nell'analisi matematica delle sue prestazioni e di fornire altresí criteri di scelta fra soluzioni algoritmiche differenti di uno stesso problema.

## 2. - IL PROBLEMA DELL'ORDINAMENTO.

Definiamo anzitutto il problema da risolvere. Sia dato un insieme $\{X\}$, di cardinalità finita $n$, ai cui elementi $x_1, x_2, \ldots, x_n$, di natura qualsiasi ma tra loro omogenei, sono associati i valori memorizzati nelle componenti $x[1], x[2], \ldots, x[n]$ di un array $X[1:n]$ e detti genericamente chiavi. Definita fra le chiavi una relazione di ordinamento totale "$<$", per ogni terna di valori delle chiavi $x[i], x[j], x[k]$ saranno soddisfatte la legge di tricotomia (vale una ed una sola delle tre possibilità $x[i] < x[j]$, $x[i] = x[j]$, $x[j] < x[i]$) e la legge di transitività (se $x[i] < x[j]$ ed $x[j] < x[k]$ allora $x[i] < x[k]$). Il problema dell'ordinamento consiste nel permutare gli elementi di $\{X\}$ in modo che

si abbia $x[1] < x[2] < \ldots < x[n]$. Se le chiavi non sono tutte distinte può verificarsi che $x[i] = x[j]$ per qualche $i$ e $j$; il processo di ordinamento si dice stabile quando elementi corrispondenti a chiavi identiche non variano il loro iniziale ordine relativo. Ci occuperemo solo di algoritmi di "ordinamento interno", in cui tutti i dati da ordinare sono memorizzati nella memoria centrale; ove ciò non sia fisicamente realizzabile occorrerà associare a tali algoritmi opportune procedure di input/output che trasferendo nella/dalla memoria centrale sottoinsiemi di $\{X\}$ ne consentano l'ordinamento per fasi successive ("ordinamento esterno").

Proviamo dunque a costruire un algoritmo che risolva correttamente il nostro problema.

Consideriamo preliminarmente una procedura $MIN$ che determini il minimo elemento di $\{X\}$. Confrontando la chiave di $x[1]$ (cioè il valore correntemente allocato nella prima componente dell'array $X[1:n]$) con ciascuna delle successive $n-1$ chiavi $x[i]$, $i = 2, \ldots, n$, ogniqualvolta si abbia $x[i] < x[1]$ la procedura scambierà le relative chiavi, invertirà cioè i contenuti di memoria di $x[1]$ e di $x[i]$ avvalendosi di una variabile ausiliaria. Dopo $n-1$ confronti il minimo elemento di $\{X\}$ si troverà memorizzato in $x[1]$.

Si osservi che $n-1$ confronti sono "sufficienti" per la determinazione del minimo ma sono anche "necessari" perché ognuno degli $n-1$ elementi diversi dal minimo dovrà essere confrontato con esso almeno una volta.

Analogamente considerando l'insieme ottenuto eliminando da $\{X\}$ l'elemento memorizzato in $x[1]$, potremo determinarne il minimo applicando la procedura $MIN$ all'array $X[2:n]$; dopo $n-2$ confronti tale minimo si troverà allocato in $x[2]$. Per ordinare l'intero insieme $\{X\}$ basterà allora costruire un algoritmo (lo chiameremo $SELECTMIN$), che richiami $n-1$ volte la procedura $MIN$ su sottoinsiemi di $\{X\}$ di cardinalità via via decrescenti di una unità, disponendo in ordine crescente tutti gli elementi dopo aver eseguito, indipendentemente dalla configurazione iniziale dei dati, $C(n)$ confronti tra essi con

$$C(n) = \sum_{j=1}^{n-1}(n-j) = \frac{n(n-1)}{2} = \frac{n^2-n}{2} \qquad (1)$$

**Tab.A** **Esempio di funzionamento di SELECTMIN**

| x[1] | x[2] | x[3] | x[4] | x[5] | |
|---|---|---|---|---|---|
| 7 | 12 | 5 | 8 | 2 | MIN(1) |
| 5 | 12 | 7 | 8 | 2 | 4 confronti |
| 2 | 12 | 7 | 8 | 5 | |
| | 12 | 7 | 8 | 5 | MIN(2) |
| | 7 | 12 | 8 | 5 | 3 confronti |
| | 5 | 12 | 8 | 7 | |
| | | 12 | 8 | 7 | MIN(3) |
| | | 8 | 12 | 7 | 2 confronti |
| | | 7 | 12 | 8 | |
| | | | 12 | 8 | MIN(4) |
| | | | 8 | 12 | 1 confronto |
| 2 | 5 | 7 | 8 | 12 | |

Nella **Tab.A** è riportato un esempio di funzionamento di $SELECTMIN$. $MIN(j)$ determina il minimo di $X[j:n]$, $j = 1, \ldots, n-1$. Gli elementi che vengono scambiati, producendo una nuova configurazione dei dati, sono sottolineati.

Anticipiamo subito che il numero $C(n)$ di confronti fra elementi, eseguiti da un algoritmo di ordinamento su un insieme di dati di cardinalità $n$, gioca un ruolo essenziale nell'analisi a priori della sua efficienza, perché alla valutazione di $C(n)$ si riconduce sostanzialmente la stima del tempo di esecuzione. Tale criterio trova un'immediata giustificazione ove si consideri che se gli $n$ elementi di $\{X\}$ sono polinomi, vettori, numeri molto grandi o stringhe di caratteri il tempo impiegato per confrontare due elementi è molto maggiore di quello richiesto da qualunque altra operazione eseguita dall'algoritmo, per esempio da un'istruzione di assegnazione, dall'incremento di un indice contatore di un ciclo, ecc. E comunque, anche nel caso in cui gli elementi di $\{X\}$ siano interi (come supporremo per semplicità nelle codifiche degli algoritmi che presenteremo) e dunque il costo di un confronto risulti confrontabile con quello di un'altra operazione, basterà notare che il numero delle altre operazioni ha lo stesso ordine di grandezza di $C(n)$ per convincersi facilmente che il tempo $t(n)$ occorrente per l'ordinamento di $n$ dati risulterà proporzionale a $C(n)$.

Al crescere di $n$ prevale nella (1) il termine in $n^2$, dunque quando $n$ raddoppia si quadruplica il tempo di computazione:

$$t(n) = kn^2 \quad \Longrightarrow \quad t(2n) = k(2n)^2 = 4t(n)$$

$SELECTMIN$ non appare molto efficiente; non sappiamo ancora se sia possibile far meglio, certo è che finché il metodo di ordinamento si baserà sul confronto di tutte le possibili coppie di chiavi non sarà possibile liberarsi dalla dipendenza quadratica da n.

Occorre escogitare un metodo che consenta di ridurre il numero dei confronti, inferendo la relazione di ordinamento tra alcuni degli elementi da confronti già effettuati con altri elementi.

Vedremo che ciò è in effetti possibile e che l'unico vantaggio di $SELECTMIN$ è la sua semplicità, non certo la sua efficienza. La relativa codifica è riportata in **Cod.Ps.1** al solo scopo di introdurre brevemente il linguaggio e lo stile con cui descriveremo algoritmi più complessi ma più efficienti.

Per formalizzare gli algoritmi presentati avremmo potuto scegliere, per non far torto ai fautori dei diversi linguaggi di programmazione oggi in uso e per snellire la trattazione, un semplice linguaggio descrittivo, per cui cioè non esista un compilatore ma che possa agevolmente essere tradotto da ciascuno nel linguaggio preferito. Tuttavia, poiché intendiamo rivolgerci anche a coloro che, pur di evitare tale traduzione, rinuncerebbero ai benefici di un ordinamento efficiente, abbiamo scelto la codifica in $PASCAL$, linguaggio standardizzato, strutturato, che permette la ricorsività e risulta facilmente comprensibile anche a chi non abbia esperienza di programmazione. Gli algoritmi più efficienti saranno codificati anche in $FORTRAN$, per la sua diffusione nell'ambiente scientifico. (Tutti i programmi presentati sono stati provati su $VAX$ 11/780 $DIGITAL$).

Seguendo i canoni della programmazione strutturata che, scomponendo il problema dato in sottoproblemi più semplici, si propone di rendere l'espressione di un algoritmo chiara e concisa, facilmente documentabile e comunicabile, la formalizzazione sarà di tipo modulare, evitando sia inutili ripetizioni di istruzioni, mediante la definizione di opportune procedure, sia le istruzioni di salto, sempre sostituibili con strutture condizionali. Un programma si presenterà dunque come un insieme

di una o più procedure, una delle quali è assunta come programma principale (*main program*). A questo sono affidati l'input e l'output dei dati e la chiamata della procedura di ordinamento (nel nostro caso *SELECTMIN*), la quale a sua volta potrà chiamare altre procedure. Gli elementi da ordinare saranno sempre, per semplicità, supposti interi e l'insieme $\{X\}$ e la sua cardinalità $n$ verranno definiti globalmente, cioè nel programma principale, risultando quindi noti a tutte le procedure senza figurare tra i loro argomenti. Per esempio in **Cod.Ps.1** *CONST nmax* = 1250 e *TYPE elem* = *integer* dichiarano rispettivamente la massima dimensione ed il tipo degli elementi di $X$ definito come $x : array[1..nmax]\ of\ elem$. Per ordinare elementi di tipo diverso o per aumentare la cardinalità basterà perciò cambiare esclusivamente le dichiarazioni *CONST* e *TYPE* del programma principale.

```
PROGRAM SELECTSORT (INPUT, OUTPUT);
TYPE elem=integer; CONST nmax=1250;
VAR x:array[1..nmax] of elem;
    n,i:integer;
PROCEDURE CHANGE (VAR a,b:integer);
VAR c:integer;
(* scambia le chiavi di a e b *)
BEGIN
  c:=a; a:=b; b:=c;
END;
PROCEDURE MIN(j:integer);
VAR k:integer;
(* individua il minimo di X[j:n]  *)
BEGIN
  k:=j+1;
  WHILE k<=n DO
    BEGIN
      IF x[k]<x[j] THEN CHANGE(x[k],x[j]);
      k:=k+1;
    END;
END;
PROCEDURE SELECTMIN;
VAR j:integer;
(* ordina gli elementi di X[1:n]  *)
(* X ed n sono definiti globalmente *)
BEGIN
  FOR j:=1 TO n-1 DO MIN(j);
END;
(* main program *)
BEGIN
  writeln(' enter data number n');
  read(n);
  writeln(' enter data');
  FOR i := 1 TO n DO read (x[i]);
  SELECTMIN;
  writeln;
  FOR i := 1 TO n DO writeln (x[i],' ');
END.
```

**Cod.Ps.1** **SELECTMIN**

Ogni procedura in *PASCAL* è definita prima del suo uso nel programma o in un'altra pro-

cedura che la richiami (ambiente della procedura) e le sue istruzioni agiscono su variabili globali (accessibili in ogni punto del programma) e/o su variabili locali (significative solo nel corpo della procedura) e/o su dati e risultati generici detti parametri formali. All'atto della chiamata il programma principale o la procedura chiamante trasmette e riceve dati relativi al calcolo in atto, detti parametri attuali o effettivi, biunivocamente corrispondenti in ordine e tipo ai parametri formali. La trasmissione può avvenire per valore o per referenza. Nel primo caso la comunicazione è a senso unico; il parametro effettivo, che può anche essere un'espressione, viene valutato ed il suo valore è sostituito al corrispondente parametro formale, il quale non potrà essere usato come output della procedura (si consideri per esempio la chiamata $MIN(j)$ in $SELECTMIN$). Nel secondo caso il parametro effettivo è una variabile e come tale è sostituito al rispettivo parametro formale, che deve essere preceduto dalla parola $VAR$ e fornisce in uscita un risultato (parametro di ritorno della procedura). Per esempio lo scambio di due elementi è affidato alla procedura $CHANGE$; $c$ è una variabile locale che serve per effettuare lo scambio fra i valori dei parametri formali $a$ e $b$. All'atto della chiamata $CHANGE(x[k], x[j])$ i parametri effettivi $x[k]$ ed $x[j]$ vengono sostituiti ad $a$ e $b$ rispettivamente; all'uscita della procedura le chiavi di $x[k]$ ed $x[j]$ risulteranno scambiate.

**Tab.B Parole chiave ed istruzioni PASCAL**

- `(INPUT,OUTPUT)` : files standard di lettura e scrittura.
- `BEGIN··END` : parole chiave che racchiudono un blocco di istruzioni eseguibili, separate dal delimitatore ";".
- `CONST,VAR,TYPE` : parole chiave di dichiarazione di costanti, variabili, tipi.
- `variabile := espressione` : istruzione di assegnazione. (Es: $j := 1$ alla variabile $j$ e' assegnato il valore 1).
- `(*...*)` : commenti, ignorati dal compilatore.
- Operatori aritmetici: + (addizione), − (sottrazione), * (moltiplicazione, / (divisione), `DIV` (divisione intera), `MOD` (resto della divisione), ** (potenza). Operatori booleani: `AND,OR,NOT`. Operatori relazionali: $=, <>, <, <=, >, >=$ (rispettivamente $=, \neq, <, \leq, >, \geq$).
- `MAXINT` : massimo intero disponibile nel calcolatore usato; `TRUNC(x)` : effettua la conversione reale-intero, calcolando $\lfloor x \rfloor$, cioè il più grande intero $\leq x$; `LN(x)` logaritmo neperiano di $x$.
- `read, readln, write, writeln` : istruzioni di input/output.
- `IF ''condizione'' THEN ''istruzione A'' ELSE ''istruzione B''` : Se la "condizione" è soddisfatta allora viene eseguita l' "istruzione (o il blocco di istruzioni) A" altrimenti è eseguita l'"istruzione (o il blocco di istruzioni) B".
- `WHILE ''condizione'' DO ''istruzione A''` : Finché la "condizione" è soddisfatta allora viene eseguita l' "istruzione (o il blocco di istruzioni) A".
- `FOR ''indice'':= ''espr1'' TO (DOWNTO)''espr2'' DO ''istruzione''` : l'istruzione è eseguita per i valori interi e consecutivi, crescenti (`TO`) o decrescenti (`DOWNTO`) dell'indice, a partire da `espr1` fino ad `espr2`.
- `GOTO ''label''` : istruzione di salto ad una etichetta predichiarata, permette tra l'altro di uscire da una struttura prima della sua normale conclusione (per esempio da una procedura, restituendo il controllo alla procedura chiamante).

La **Tab.B**, che elenca le parole chiave e le istruzioni $PASCAL$ che useremo, ne agevolerà la comprensione a chi non ne abbia pratica.

Prima di presentare altri algoritmi di ordinamento introdurremo alcuni concetti base di complessità computazionale.

## 3. - COMPLESSITÀ COMPUTAZIONALE.

Definito il problema e determinato un algoritmo che lo risolva correttamente, se riusciamo a valutarne il tempo di esecuzione e lo spazio di memoria occupato durante l'esecuzione stessa, che si dicono rispettivamente "complessità in tempo" e "complessità in spazio" dell'algoritmo, possiamo asserire che essi sono "sufficienti" per la risoluzione del problema studiato; non sappiamo però se siano anche "necessari". In altri termini tali valori fissano un limite superiore di complessità ma la loro conoscenza non ci fornisce alcuna informazione sulla bontà dell'algoritmo finché non siamo in grado di stabilire se sia o no possibile, almeno in linea di principio, abbassarli, elaborando quindi un'altra procedura di calcolo che impieghi meno tempo o invada un minor spazio di memoria. La tecnica comunemente usata a tale scopo consiste nella ricerca di una funzione che costituisca un limite inferiore alla complessità in tempo o in spazio. Se l'algoritmo in studio risolve il problema in un tempo eguale al limite inferiore di complessità in tempo oppure occupa uno spazio di memoria esprimibile mediante una funzione esattamente eguale al limite inferiore di complessità in spazio allora l'algoritmo è ottimo e sarà vano ricercarne uno più efficiente. In pratica tale condizione è raramente dimostrabile ma spesso si può provare che un algoritmo è ottimo in ordine di grandezza.

Poiché oggi sono reperibili sul mercato memorie grandissime a costi relativamente contenuti e tendenzialmente decrescenti, ci si preoccupa della complessità in spazio solo quando essa cresce esponenzialmente al crescere della dimensione del problema; le nostre considerazioni saranno perciò prevalentemente rivolte allo studio della complessità in tempo.

Determinare un parametro significativo che caratterizzi la dimensione del problema è di solito abbastanza semplice: per esempio, se l'algoritmo opera su matrici tale parametro potrà essere il numero di elementi oppure, se si tratta di matrici quadrate, il numero di righe o di colonne; se la procedura agisce invece su un insieme allora il parametro di dimensione, che nel seguito indicheremo con $n$, sarà il numero di dati in ingresso o in uscita o la loro somma o il modulo di uno di essi, e così via. Nel problema dell'ordinamento il parametro che ne caratterizza la dimensione è il numero degli elementi da ordinare.

Per analizzare il comportamento dell'algoritmo per un generico $n$ occorrerà preliminarmente determinare il tipo ed il numero di operazioni impiegate. Potrà trattarsi di azioni elementari che richiedono ciascuna una fissata quantità di tempo, o di operazioni costituite da una serie più o meno lunga di passi elementari. Per esprimere il tempo totale di elaborazione in secondi occorrerebbe conoscere, oltre alla frequenza di esecuzione, anche il tempo effettivo richiesto dalla singola azione elementare, ma quest'ultimo dipende significativamente dal linguaggio di programmazione, dal compilatore, dal processore che vengono usati e dai quali invece si deve prescindere in un'analisi a priori dell'algoritmo. Ipotizzeremo solo che il computer su cui verrà implementato l'algoritmo sia dotato di una *RAM* (memoria ad accesso casuale che permette di accedere ad un elemento o di allocarlo in memoria in una fissata quantità di tempo, indipendente dalla posizione del dato in memoria) e sia di tipo sequenziale (esegua cioè una alla volta e sequenzialmente le istruzioni del programma); l'impiego di un elaboratore dell'ultima generazione, a computazione parallela, richiederebbe naturalmente considerazioni diverse.

Indicando con $n$ il numero di posizioni successive occupate dai dati di ingresso (nella macchina di Turing $n$ denoterebbe la lunghezza della parte di nastro contenente l'input) ed esprimendo la complessità in tempo come funzione $t(n)$ del parametro di dimensione $n$, ci si può limitare a

studiare la complessità asintotica, cioè il tipo e l'ordine di grandezza della funzione di complessità quando $n$ cresce indefinitamente.

In prima approssimazione è sufficiente stimare il numero delle operazioni preminenti, che richiedono cioè maggior tempo di esecuzione rispetto alle altre azioni eseguite dall'algoritmo, e valutare i termini di ordine più alto in $n$, trascurando in molti casi anche le costanti moltiplicative ad essi associate. Essendo infatti il problema formulato per un generico $n$, due algoritmi diversi possono fornire prestazioni confrontabili per valori limitati del parametro di dimensione, ma per $n$ sufficientemente elevato risulta più efficiente l'algoritmo la cui funzione di complessità presenta il termine prevalente in $n$ di grado più basso, pur tenendo nel debito conto che valutazioni particolari, in relazione alla proprie esigenze applicative ed al tipo di calcolatore e di linguaggio usati, potranno far preferire l'algoritmo intrinsecamente meno efficiente. Per esempio un algoritmo lineare che impieghi $1000n$ unità di tempo è asintoticamente più efficiente di un algoritmo quadratico con $t(n) = 5n^2$ unità di tempo ma risulta meno veloce per $n < 200$.

Per valutare la complessità asintotica degli algoritmi di ordinamento di cui ci occuperemo ci basterà una buona stima del numero $C(n)$ di confronti fra chiavi occorrenti per ordinare $n$ elementi in quanto, come si è già accennato, la funzione di complessità in tempo presenta un termine prevalente proporzionale a $C(n)$.

Il numero delle operazioni eseguite da un algoritmo e quindi il suo tempo di esecuzione possono o no variare in relazione alle diverse possibili configurazioni iniziali dei dati. Se non variano l'algoritmo si dice robusto, non risente cioè del modo in cui i dati sono immessi. (Sia $MIN$ sia $SELECTMIN$ sono per esempio robusti.) Se invece diverse configurazioni iniziali comportano elaborazioni distinte è importante, ed in genere è relativamente semplice, individuare la configurazione che determina il tempo di esecuzione maggiore, fornendo un limite superiore alla complessità dell'algoritmo (complessità nel caso pessimo). Più difficile da valutare è il valore della complessità mediato su tutte le possibili configurazioni di input (complessità nel caso medio).

Molto utili nello studio dei limiti superiori ed inferiori di complessità risultano la $O$-notazione e la $\Omega$-notazione rispettivamente.

Diremo che una funzione $t(n)$ è di ordine $f(n)$ e scriveremo

$$t(n) = O(f(n))$$

se e solo se esistono due costanti positive $c_0$ ed $n_0$ tali che

$$|t(n)| \leq c_0|f(n)| \qquad \text{per tutti gli} \quad n \geq n_0$$

Ciò significa che asintoticamente, cioè quando $n$ cresce indefinitamente, $t(n)$ cresce al più come $f(n)$. La definizione data implica che se $t(n) = O(f(n))$ è anche $t(n) = O(f'(n))$ per ogni $f'(n) > f(n)$; nel determinare l'ordine di grandezza di $t(n)$ si cercherà di trovare il più basso livello di complessità entro cui il corretto funzionamento dell'algoritmo è garantito, cioè la più piccola $f(n)$ tale che $t(n) = O(f(n))$.

Analogamente per esprimere matematicamente che una funzione $g(n)$ limita inferiormente la crescita asintotica della complessità si usa la $\Omega$-notazione.

Diremo che $t(n)$ è di ordine $\Omega(g(n))$ e scriveremo

$$t(n) = \Omega(g(n))$$

se e solo se esistono due costanti positive $c_1$ ed $n_1$ tali che

$$|t(n)| \geq c_1|g(n)| \qquad \text{per tutti gli} \quad n \geq n_1$$

Asintoticamente cioè $t(n)$ non cresce meno di $g(n)$. Se $t(n) = \Omega(g(n))$ è anche $t(n) = \Omega(g'(n))$ per ogni $g'(n) < g(n)$.

Supponiamo di aver provato che la complessità asintotica di un qualunque algoritmo che risolva un dato problema è limitata inferiormente da $\Omega(g(n))$. Se si riesce a costruire un algoritmo di complessità $O(g(n))$ allora si può affermare che tale algoritmo è il più efficiente possibile. Per esempio la procedura $MIN$ per la determinazione del minimo di un insieme di $n$ elementi è ottima perché ha tempo di computazione $t(n) = O(n) = \Omega(n)$; (si ricordi che occorrono e bastano $n - 1$ confronti per individuare il minimo).

Non esiste un metodo generale per la determinazione dei limiti inferiori. È spesso molto semplice individuare dei limiti inferiori troppo bassi e quindi non interessanti perché non raggiungibili dalla complessità di procedure effettivamente realizzabili. Per esempio la complessità di una qualunque procedura di ordinamento di $n$ dati è limitata inferiormente da $\Omega(n)$ in quanto se non si esaminano tutti gli $n$ dati il loro confronto non è eseguibile, a meno che i dati non siano stati in qualche modo preordinati. Vedremo subito che è possibile fissare un limite inferiore più alto e quindi più significativo utilizzando una struttura di dati detta albero di decisione.

## 4. - L'ALBERO DI DECISIONE E LA RICORSIVITÀ.

Per descrivere un algoritmo che conduca alla soluzione di un dato problema mediante decisioni successive, al cui numero risulti proporzionale la funzione di complessità dell'algoritmo, si può utilizzare una struttura di dati chiamata *albero di decisione.*

*L'albero* è una struttura informativa di fondamentale importanza in quanto, formalizzando il procedimento di partizioni successive di un insieme, è applicabile in tutte quelle situazioni in cui si richieda un'organizzazione dei dati a ramificazioni successive, per esempio negli schemi di classificazione, nei problemi di suddivisione gerarchica di più elementi, nell'indicazione della priorità di esecuzione delle operazioni di un'espressione algebrica, nei procedimenti enumerativi, logici o numerici, che determinino la soluzione di un problema attraverso l'esame di tutti i casi possibili.

Se ne può dare una definizione rigorosa in forma ricorsiva:

*Un albero è un insieme finito non vuoto X di n elementi* $x_1, x_2, \ldots, x_n$ *detti nodi, uno dei quali è detto radice e se* $n > 1$ *i rimanenti nodi sono ripartiti in m insiemi disgiunti* $X_1, X_2, \ldots, X_m$, *che risultano a loro volta alberi e si dicono sottoalberi della radice.*

I nodi di un albero, graficamente collegati da segmenti detti rami, sono raggruppabili in livelli, definendo livello del generico nodo $x_i$ il numero di nodi da percorrere, partendo dalla radice, che ha livello 0, per giungere sino ad $x_i$. Le radici dei sottoalberi di un nodo $x_i$ sono detti figli o progenie di $x_i$ che è il loro padre; i nodi con figli si dicono interni o non terminali, quelli senza figli sono chiamati nodi terminali o foglie.

Se l'albero è usato per schematizzare il funzionamento di un algoritmo ogni suo nodo non terminale rappresenta una decisione da prendere nel corso dell'esecuzione, le foglie rappresentano le possibili soluzioni del problema per le diverse permutazioni iniziali dei dati. Poiché ogni percorso che dalla radice, attraverso nodi interni, conduce ad una foglia rappresenta la successione di

decisioni da prendere, in una particolare esecuzione dell'algoritmo, per determinare la soluzione indicata dalla foglia, il caso ottimo ed il caso pessimo nel funzionamento dell'algoritmo corrispondono ai percorsi radice-foglia di minima e di massima lunghezza rispettivamente; le relative soluzioni saranno individuate dalle foglie rispettivamente a livello minimo e massimo. Mediando le lunghezze di tutti i possibili percorsi radice-foglia in un dato albero otteniamo il valor medio del numero di decisioni successive che il corrispondente algoritmo comporta.

L'albero che, nella classe di tutti i possibili alberi di decisione relativi ad un dato problema, minimizza la lunghezza massima (o la lunghezza media) dei percorsi radice-foglia individua con tale lunghezza un limite inferiore al numero di decisioni occorrenti per la risoluzione del problema nel caso più sfavorevole (o nel caso medio).

Negli algoritmi di cui ci stiamo occupando le decisioni successive, il cui numero o costo superi quello di ogni altra azione e sia perciò indicativo della complessità, sono i confronti fra gli elementi da ordinare. Se, come supporremo preliminarmente, gli elementi sono tutti distinti allora ogni confronto fra $x_i$ ed $x_j$, $(i, j = 1, \ldots, n)$ ha solo due possibili risultati: $x_i < x_j$ ed $x_j < x_i$. L'albero di decisione è dunque un albero binario, definibile in modo informale come un albero in cui ogni nodo non terminale ha al più due figli. (Se gli elementi non fossero tutti distinti l'albero sarebbe ternario; ogni nodo interno rappresenterebbe ancora un confronto fra elementi e sarebbe radice di tre sottoalberi corrispondenti ai tre possibili risultati del confronto: $x_i < x_j, x_i = x_j, x_j < x_i$).

Più rigorosamente: *Un albero binario è un insieme finito di elementi detti nodi tale che o è vuoto o è formato da un nodo particolare designato come radice e da due alberi binari detti sottoalberi sinistro e destro della radice.*

Dunque un albero binario non è un caso particolare di albero, infatti può essere vuoto (mentre un albero ha sempre almeno un nodo) ed ognuno dei suoi sottoalberi mantiene la propria identità di sottoalbero sinistro o destro anche se l'altro sottoalbero è vuoto.

Un albero ordinato, nel quale cioè si considera significativo l'ordine in cui figurano i nodi, è sempre trasformabile in un albero binario, collegando con rami i nodi aventi lo stesso padre ed eliminando quindi tutti i rami tra un nodo ed i suoi figli tranne quello più a sinistra (**Fig.1**).

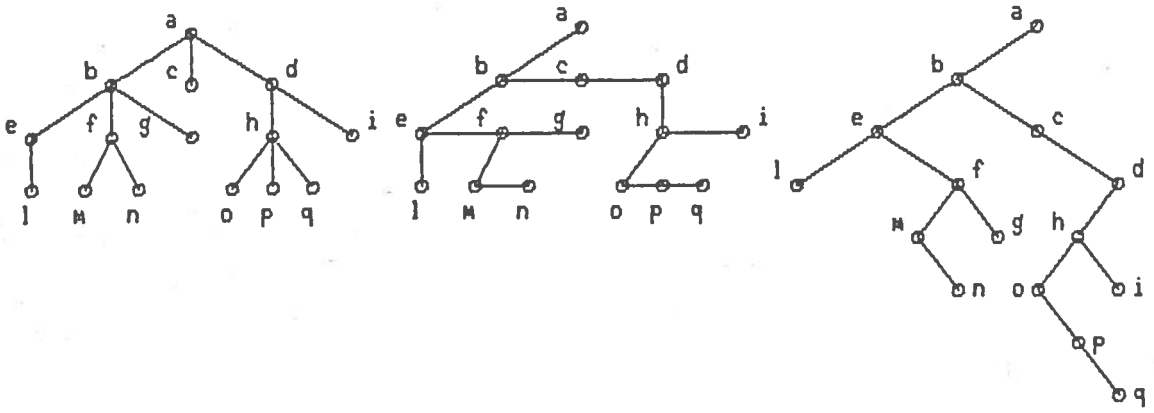


**Fig.1** *Un albero ordinato e la sua trasformazione in albero binario.*

Esistono diverse tecniche di attraversamento di un albero che consentono di visitare tutti i suoi nodi esaminando ciascun nodo una ed una sola volta. L'attraversamento per livello, "top-down"

o "bottom-up", viene eseguito scorrendo da sinistra a destra tutti i nodi, livello per livello, procedendo dall'alto verso il basso o dal basso verso l'alto. Si dice che un albero binario è attraversato in ordine simmetrico (invisita), anticipato (previsita), o posticipato (postvisita) se l'esame della radice è intermedio, precede o segue, rispettivamente, la visita ricorsiva del sottoalbero sinistro e del sottoalbero destro. In **Fig.2** è rappresentato l'albero binario associato ad un'espressione algebrica; l'invisita, la previsita e la postvisita corrispondono rispettivamente alla usuale notazione infissa $(a+b+c*d-e/f)$, alla notazione polacca prefissa in cui l'operatore precede i suoi operandi $(++ab-*cd/ef)$ ed alla notazione polacca postfissa in cui l'operatore segue i suoi operandi $(ab+cd*ef/-+)$.

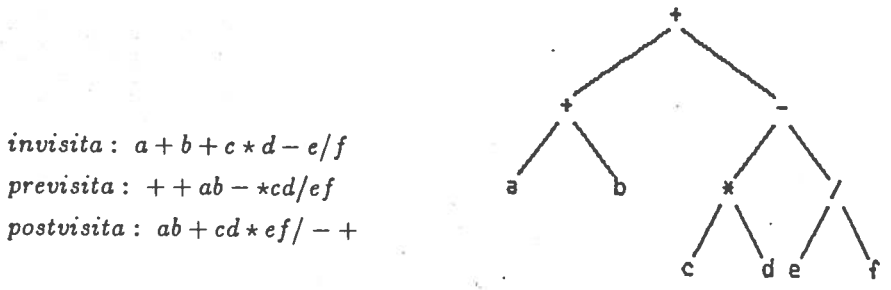


**Fig.2** *Un'espressione algebrica e l'albero binario ad essa associato.*

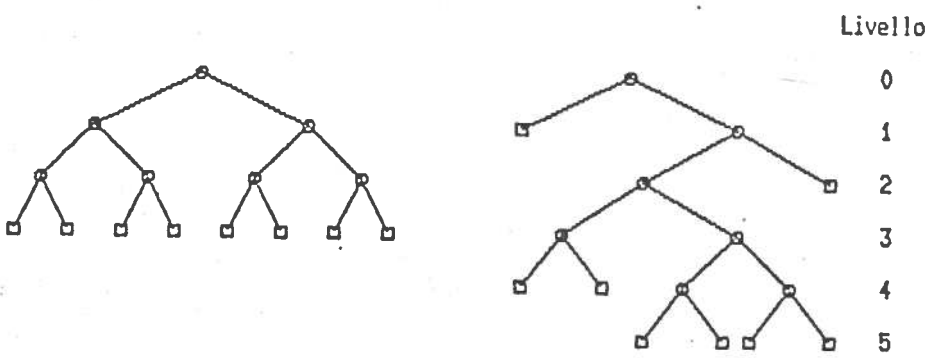


**Fig.3** *Albero binario perfettamente bilanciato (k=3) ed albero sbilanciato con lo stesso numero di foglie.*

In un albero binario il numero massimo dei nodi di un livello varia in progressione geometrica di ragione 2 al crescere del livello poiché da ogni nodo padre a livello $\ell$ possono dipartirsi al più due nodi figli a livello $\ell+1$. Dunque, poiché il generico livello $\ell$ ha $2^\ell$ nodi, un albero binario completo con massimo livello $k$, che si dice perfettamente bilanciato, avrà in totale $\sum_{\ell=0}^{k} 2^\ell = 2^{k+1}-1$ nodi di cui $2^k$ foglie e $2^k-1$ nodi interni; esso minimizza sia il numero massimo $k$ di nodi di decisione incontrati nei percorsi radice-foglia sia la lunghezza media di tali percorsi, fornendo quindi un limite inferiore alla complessità nel caso pessimo e nel caso medio. Si osservi infatti che un albero binario non perfettamente bilanciato con lo stesso numero $2^k$ di foglie avrà sia il livello massimo sia il livello medio maggiori di $k$ in quanto, per ogni foglia situata a livello $\ell < k$, presenterà $2^{k-\ell}$

foglie a livelli $\ell' > k$. Per esempio un albero binario con $2^3 = 8$ foglie di cui una al primo livello ed una al secondo avrà $2^{3-1} + 2^{3-2} = 6$ foglie a livelli maggiori di 3 (**Fig.3**).

L'albero perfettamente bilanciato con massimo livello $k$ rappresenta quindi l'algoritmo ideale che, nella classe di tutti gli algoritmi che risolvono un problema con $2^k$ soluzioni, esegue il minor numero di confronti nel caso pessimo e/o nel caso medio. Se un problema ammette un numero di soluzioni $s$ con $2^{k-1} \leq s < 2^k$ l'albero di decisione che minimizza il livello massimo ed il livello medio sarà un albero binario con $s$ foglie quasi perfettamente bilanciato, cioè completo fino al livello $k-1$.

Indicando con $k$ il minimo numero di confronti sufficienti per ordinare $n$ distinti elementi nel caso pessimo, il relativo albero di decisione avrà livello massimo $k$ e quindi al più $2^k$ foglie, che rappresentano le soluzioni del problema; poiché ciascuna delle $n!$ possibili permutazioni semplici di $n$ elementi può essere, in relazione alla sequenza di input, la permutazione ordinata e quindi la soluzione, le foglie dell'albero dovranno essere almeno $n!$. Dunque

$$n! \leq 2^k$$

da cui, utilizzando l'approssimazione di Stirling $n! \approx \sqrt{2\pi n}(n/e)^n$, si ottiene:

$$k \geq \lceil \log n! \rceil \approx \lceil \log \sqrt{2\pi n} + n \log(n/e) \rceil$$

(Il simbolo $\lceil x \rceil$ indica il più piccolo intero maggiore o eguale ad $x$ mentre $\log x$ indica il logaritmo in base 2 di $x$). Perciò:

$$k = \Omega(n \log n)$$

Indicando con $C_p(n)$ il numero di confronti occorrenti ad un algoritmo di ordinamento per ordinare $n$ distinti elementi nel caso pessimo si avrà

$$C_p(n) \geq k = \Omega(n \log n)$$

cioè: *Non è possibile ordinare n elementi con meno di* $\Omega(n \log n)$ *confronti.*
E poiché la funzione di complessità in tempo risulta proporzionale al numero di confronti eseguiti ciò equivale ad affermare che: *Nessun algoritmo di ordinamento basato su confronti può presentare una complessità in tempo inferiore a* $\Omega(n \log n)$ .

La **Fig.4** rappresenta un albero di decisione quasi perfettamente bilanciato per l'ordinamento di quattro elementi distinti $a$, $b$, $c$, $d$. Le foglie contengono le $4! = 24$ possibili soluzioni. In ogni nodo figurano gli elementi che vengono confrontati. Il risultato del confronto "<" o ">" determina il confronto successivo, allocato nel figlio sinistro o nel figlio destro rispettivamente. Alcune delle soluzioni ancora possibili associate al nodo vengono eliminate passando al livello inferiore finché non è raggiunta la foglia con la soluzione corretta. Per $a = 9, b = 5, c = 1, d = 3$ si arriverebbe alla soluzione seguendo il percorso indicato nella seconda parte della figura.

La struttura di albero di decisione, che ci ha permesso di stabilire il limite inferiore di $\Omega(n \log n)$ al minimo numero di confronti occorrenti per l'ordinamento di $n$ dati, indicando quindi i margini di miglioramento esistenti rispetto alla complessità $O(n^2)$ di $SELECTMIN$, può guidarci nella progettazione e nell'analisi di algoritmi che eseguano scelte il più possibile bilanciate e siano perciò rappresentabili mediante alberi di decisione perfettamente o quasi perfettamente bilanciati.

Un notevole guadagno di efficienza può ottenersi scomponendo il problema dato in sottoproblemi, risolvibili separatamente, applicati a sottoinsiemi disgiunti dei dati di cardinalità eguali o comunque molto vicine; se le dimensioni dei sottoproblemi sono ancora troppo grandi si procede ad ulteriori suddivisioni fino ad ottenere dei sottoproblemi direttamente risolvibili. Tale strategia di risoluzione, indicata in letteratura con la locuzione *divide-and-conquer* o *divide et impera*, include naturalmente la determinazione di un metodo che consenta di ricavare la soluzione globale del problema dato ricombinando opportunamente i risultati parziali dei sottoproblemi.

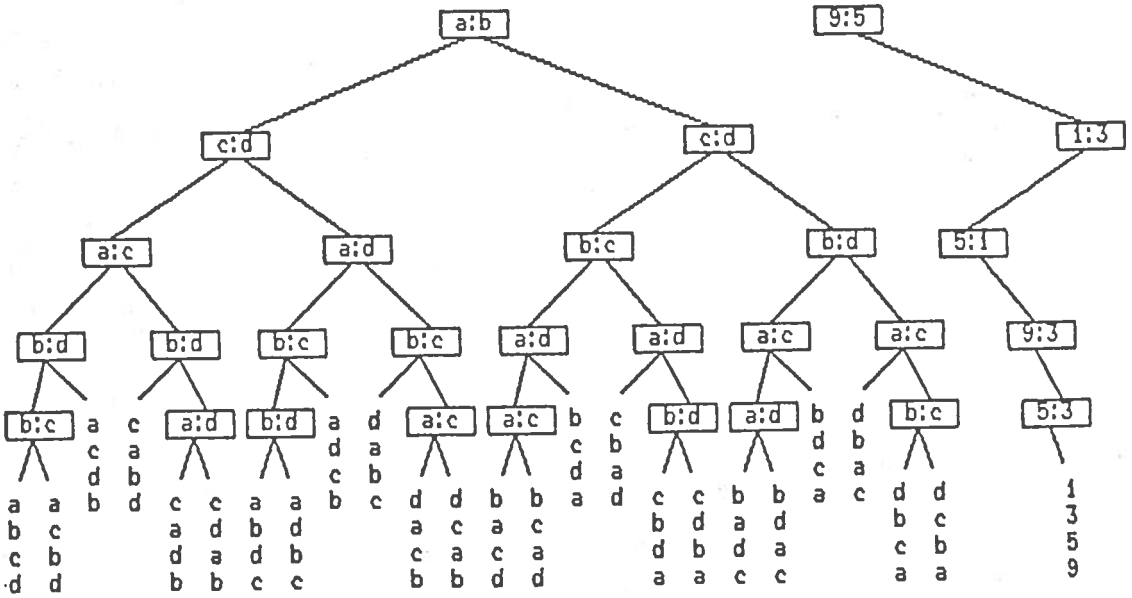


**Fig.4** *Albero di decisione quasi perfettamente bilanciato per l'ordinamento di 4 elementi distinti* $a$, $b$, $c$, $d$ *e percorso radice-foglia determinato dai valori* $a = 9$, $b = 5$, $c = 1$, $d = 3$.

Quando i problemi sono dello stesso tipo del problema originale una procedura che esprima la riapplicazione del principio del *divide et impera* si presta ad una formulazione ricorsiva.

Il termine ricorsione indica la definizione di un oggetto (insieme, relazione, funzione, procedimento,...), che si dice ricorsivo, in termini di se stesso. Ogni definizione ricorsiva deve contenere una definizione esplicita per qualche valore degli argomenti, altrimenti non terminerebbe mai (condizione di chiusura).

L'uso del nome (identificatore) di una procedura nel corpo della procedura stessa ne comporta un'esecuzione direttamente ricorsiva: la procedura chiama se stessa al suo interno. Una procedura è invece indirettamente ricorsiva se ne richiama un'altra che a sua volta richiama la prima.

La ricorsività non è sempre conveniente in programmazione, non solo perché non tutti i linguaggi la ammettono ma anche e soprattutto perché in alcuni compilatori si paga una pesante penalità in termini di tempo di esecuzione, di occupazione di memoria e di comprensibilità delle operazioni effettivamente eseguite dall'elaboratore. Infatti la legge di ricorsività è formulata *top-down*, va cioè dal generale al particolare, dalla globalità dei dati a sottoinsiemi dei dati di cardinalità

decrescenti. Le procedure si succedono una interna all'altra finché non si perviene a sottoinsiemi elementari sui quali ha inizio il calcolo, che dunque è di fatto eseguito dal basso verso l'alto. Il meccanismo di elaborazione, completamente gestito dal calcolatore, risulta talora talmente complesso che è difficile ricostruirlo per correggere eventuali errori del programma.

Tuttavia, nonostante questi innegabili svantaggi, la ricorsività rimane un potente strumento di programmazione. Moltissimi problemi infatti si prestano in modo così naturale ad una formulazione ricorsiva che la stesura del programma risulta molto semplice, con bassissima probabilità di commettere errori; la relativa dimostrazione di correttezza si può in genere agevolmente sviluppare per induzione. La valutazione del numero di operazioni-chiave eseguite dall'algoritmo è inoltre esprimibile mediante semplici relazioni di ricorrenza, risolvibili con tecniche generali.

Quando il problema lo consenta o addirittura, per la sua particolare natura, lo richieda, un primo approccio di risoluzione di tipo ricorsivo è dunque consigliabile per una formulazione compatta dell'algoritmo e per facilitare l'analisi della complessità. Sarà sempre possibile successivamente, ove lo si ritenga opportuno, trasformare l'algoritmo ricorsivo in uno equivalente che usi solo l'iterazione. (Il termine iterazione esprime l'idea di un procedimento ripetitivo, cioè la riutilizzazione di una medesima tecnica di calcolo su diverse serie di dati o su risultati intermedi del calcolo stesso, fino al verificarsi di una determinata condizione di arresto.) La versione iterativa, ottenuta mediante una tecnica di trasformazione di cui indicheremo in seguito le regole principali, appare globalmente più complessa (perché meno compatta) della versione ricorsiva ma, essendo costituita da istruzioni elementari direttamente eseguibili, è facilmente ricontrollabile in caso d'errore.

## 5. - QUICKSORT.

Un famoso algoritmo di ordinamento, mediamente molto efficiente e basato sulla strategia del *divide et impera*, è *QUICKSORT*, che utilizza il risultato di ciascun confronto per decidere quali chiavi debba confrontare successivamente. Tale tecnica, che sarebbe inappropriata in calcolatori a computazione parallela, è invece utilissima nei calcolatori di tipo sequenziale in quanto permette di ridurre notevolmente i confronti diretti tra gli elementi, utilizzando la proprietà transitiva della relazione di ordinamento.

Il principio base di *QUICKSORT* consiste nel confrontare gli elementi da ordinare, che preliminarmente supporremo distinti, con un elemento "pivot" (perno), scelto a caso fra essi, effettuando opportuni scambi in modo da suddividere l'array $X[1:n]$ in due parti $X[1:r-1]$ ed $X[r+1:n]$, contenenti rispettivamente gli elementi minori e maggiori del perno $x[r]$ ed a ciascuna delle quali è applicata ricorsivamente la procedura *QUICKSORT*, fino al completo ordinamento di $X[1:n]$. Seguiamone il funzionamento su un esempio concreto, riportato nella **Tab.C**.

Se la permutazione iniziale dei dati è casuale non è necessario usare una funzione *RANDOM* per la scelta del perno; si può utilizzare come pivot l'elemento mediano della sequenza oppure, come faremo noi, il primo elemento $x[1]$. Due cursori $\ell$ ed $r$, simboleggiati nella tabella rispettivamente da $\uparrow$ e da $\Uparrow$, percorrono $X[2:n]$. Il cursore sinistro $\ell$ si trova inizialmente in $x[2]$ e si muove verso destra finché non incontra un elemento maggiore del perno $x[1]$; quello destro $r$ parte da $x[n]$ e spostandosi verso sinistra si ferma quando punta un elemento minore di $x[1]$. Si scambiano allora le chiavi di $x[\ell]$ ed $x[r]$ ed i cursori avanzano ancora, con eventuali altri scambi, finché il

**Tab.C Esempio di funzionamento di QUICKSORT**

| x[1] | x[2] | x[3] | x[4] | x[5] | x[6] | x[7] | x[8] | x[9] | x[10] | Q.SORT | $\ell$ | $r$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 42 | 52 | 13 | 63 | 26 | 57 | 20 | 18 | 31 | 47 | (1,10) | | |
| | ↑ | | | | | | | | ⇑ | | 2 | 10 |
| | ↑ | | | | | | | ⇑ | | | | 9 |
| 42 | 31 | 13 | 63 | 26 | 57 | 20 | 18 | 52 | 47 | | | |
| | | ↑ | | | | | ⇑ | | | | 3 | 8 |
| | | | ↑ | | | | ⇑ | | | | 4 | |
| 42 | 31 | 13 | 18 | 26 | 57 | 20 | 63 | 52 | 47 | | | |
| | | | | ↑ | | ⇑ | | | | | 5 | 7 |
| | | | | | ↑ | ⇑ | | | | | 6 | |
| 42 | 31 | 13 | 18 | 26 | 20 | 57 | 63 | 52 | 47 | | | |
| | | | | | ⇑ | ↑ | | | | | 7 | 6 |
| 20 | 31 | 13 | 18 | 26 | 42 | 57 | 63 | 52 | 47 | | | |
| 20 | 31 | 13 | 18 | 26 | (42) | | | | | (1,5) | | |
| | ↑ | | | ⇑ | | | | | | | 2 | 5 |
| | ↑ | | ⇑ | | | | | | | | | 4 |
| 20 | 18 | 13 | 31 | 26 | | | | | | | | |
| | | ↑⇑ | | | | | | | | | 3 | 3 |
| | | ⇑ | ↑ | | | | | | | | 4 | |
| 13 | 18 | 20 | 31 | 26 | | | | | | | | |
| 13 | 18 | (20) | | | | | | | | (1,2) | | |
| | ↑⇑ | | | | | | | | | | 2 | 2 |
| ⇑ | ↑ | | | | | | | | | | | 1 |
| 13 | 18 | | | | | | | | | | | |
| | 18 | | | | | | | | | (2,2) | | |
| | | | 31 | 26 | (42) | | | | | (4,5) | | |
| | | | | ↑⇑ | | | | | | | 5 | 5 |
| | | | | ⇑ | ↑ | | | | | | 6 | |
| | | | 26 | 31 | | | | | | | | |
| | | | 26 | | | | | | | (4,4) | | |
| | | | | | | 57 | 63 | 52 | 47 | (7,10) | | |
| | | | | | | | ↑ | | ⇑ | | 8 | 10 |
| | | | | | | 57 | 47 | 52 | 63 | | | |
| | | | | | | | | ↑⇑ | | | 9 | 9 |
| | | | | | | | | ⇑ | ↑ | | 10 | |
| | | | | | | 52 | 47 | 57 | 63 | | | |
| | | | | | | 52 | 47 | (57) | | (7,8) | | |
| | | | | | | | ↑⇑ | | | | 8 | 8 |
| | | | | | | | ⇑ | ↑ | | | 9 | |
| | | | | | | 47 | 52 | | | | | |
| | | | | | | 47 | | | | (7,7) | | |
| | | | | | | | | | 63 | (10,10) | | |
| 13 | 18 | 20 | 26 | 31 | 42 | 47 | 52 | 57 | 63 | | | |

cursore destro non incrocia e supera il cursore sinistro. Scambiando $x[1]$ con $x[r]$, che contiene a questo punto l'elemento più a destra fra tutti quelli minori del perno, si ottiene una permutazione $x[1], \ldots, x[r-1], x[r], x[r+1], \ldots, x[n]$ degli elementi di $X[1:n]$, con $x[i] < x[r]$ per $i = 1, \ldots, r-1$ ed $x[r] < x[j]$ per $j = r+1, \ldots, n$. Il perno $x[r]$ risulta allora elemento di separazione delle due sezioni $X[1:r-1]$ ed $X[r+1:n]$ di $X[1:n]$, ordinabili a loro volta mediante $QUICKSORT$.

Nel nostro caso $X$ ha dieci elementi; il pivot iniziale è $x[1] = \underline{42}$ (nella tabella il pivot è sempre sottolineato); $\ell$ si ferma nella locazione [2] di partenza perché vi trova 52 che è maggiore del perno mentre $r$ parte da $x[10] = 47$ ed avanza fino ad $x[9] = 31 < 42$. Si scambiano di posto 31 e 52 ed i cursori avanzano di un passo; $x[3] = 13 < 42$, $42 < x[4] = 63$, dunque $\ell$ giunge sino alla locazione [4], $r$ rimane fermo in $x[8] = 18 < 42$. Scambiati di posto 63 e 18 si riprende con $\ell$ in $x[5] = 26 < 42$ ed $r$ in $x[7] = 20$; $\ell$ si sposta in $x[6] = 57$, $r$ non si muove. Si scambiano 57 e 20 e si portano $\ell$ in [7] ed $r$ in [6]. Poiché i cursori si sono incrociati (si noti che ciò accade quando $\ell = r+1$) il loro movimento si arresta. Si scambia allora il pivot con $x[6] = 20$ e si ottiene la partizione attorno al nuovo pivot $x[6] = 42$. $QUICKSORT$ viene richiamata ricorsivamente su $X[1:5]$ ed $X[7:10]$.

Formalizziamo in $PASCAL$ l'algoritmo presentato (**Cod.Ps.2**). La codifica più naturale di $QUICKSORT$ è ovviamente ricorsiva. Ai parametri formali *left* e *right* corrisponderanno le locazioni iniziale e finale, rispettivamente, dell'array (o della sezione di array) da ordinare. La partizione della sezione di $X$ compresa fra le locazioni $[j]$ ed $[r]$ attorno al perno $x[j]$ è affidata alla procedura non ricorsiva $PIVOT(j, r)$, chiamata dalla procedura ricorsiva $QUICKSORT(left, right)$ di cui utilizza gli stessi parametri effettivi, riportati nella tabella sotto $Q.SORT$. Per ipotesi si ha $x[i] < x[r+1]$ per ogni $i$ con $j \leq i \leq r$. Per garantire tale assunzione anche per $r = n$ occorre definire $x[n+1]$. Perciò nel programma principale, che al solito esegue l'input e l'output dei dati, è inserita l'istruzione di assegnazione $x[n+1] := MAXINT$, (dove $MAXINT$ è, in $PASCAL$ standard, il massimo intero disponibile nel calcolatore usato); essa deve precedere la prima chiamata $QUICKSORT(1, n)$, che dà inizio al processo di ordinamento dell'intera sequenza $X[1:n]$. Si osservi che il parametro formale $r$, essendo un parametro di ritorno della procedura $PIVOT$, in quanto trasmette in uscita la locazione finale del perno, è preceduto dalla parola chiave $VAR$. Per esempio all'atto della prima chiamata $j = left = 1$ ed $r = right = n$; $PIVOT(1, n)$ effettua la partizione di $X[1:n]$ attorno al perno $x[1]$. All'uscita della procedura, $r$ contiene la posizione finale del perno, cioè, nel nostro esempio, [6]. Quando $left = right$ $QUICKSORT$ non esegue alcun confronto, in quanto un insieme costituito da un solo elemento è già ordinato.

Il tempo complessivo $t(n)$ impiegato dall'algoritmo per l'ordinamento di $n$ dati dipende linearmente dal numero $C(n)$ di confronti eseguiti:

$$t(n) = aC(n) + b$$

dove $a$ è una costante e $b$ indica il complesso di termini che crescono con $n$ meno velocemente di $C(n)$. Per valutare la complessità di $QUICKSORT$ sarà dunque sufficiente stimare $C(n)$.

È facile verificare che in ogni chiamata della procedura $PIVOT(j, r)$ si eseguono al più $r-j+2$ confronti fra elementi. Infatti il ciclo più esterno ($WHILE\ \ell <= r$) ha termine quando $\ell = r+1$ cioè quando il puntatore sinistro incrocia e supera il puntatore destro, per cui $x[\ell]$ risulta l'elemento più a sinistra fra tutti quelli maggiori del perno mentre $x[r]$ è l'elemento minore più a destra. Il perno $x[j]$ ha dunque subito $r-j$ confronti con gli elementi compresi fra le posizioni $\ell = j+1$ ed $r$,

più eventualmente altri due confronti con gli elementi corrispondenti ai valori finali dei puntatori. Per esempio nella chiamata $PIVOT(1,5)$ il 13 ed il 31 sono confrontati due volte con il perno 20 nei due cicli $WHILE\ x[\ell] < x[j]$ e $WHILE\ x[j] < x[r]$.

```
PROGRAM QUICKSORTING (.....);
.....................
PROCEDURE CHANGE .................;
.....................
PROCEDURE PIVOT(j:integer; VAR r:integer);
LABEL 10; VAR l:integer;
(* ripartisce gli elementi di X[j : r] attorno *)
(* al pivot x[j]; all'uscita il pivot è x[r] *)
BEGIN
  l:=j+1;
  WHILE l<=r DO
    BEGIN
      WHILE x[l]<x[j] DO l:=l+1;
      WHILE x[j]<x[r] DO r:=r-1;
      IF l<r THEN
        BEGIN
          CHANGE (x[l],x[r]);
          l=l+1; r:=r-1;
        END;
      IF (l=r) AND (x[l]=x[j]) THEN GOTO 10;
    END;
  CHANGE(x[j],x[r]); 10:
END;
PROCEDURE QUICKSORT(left,right:integer);
VAR r:integer;
(* ordina gli elementi di  X[left : right] *)
BEGIN
  IF left<right THEN
    BEGIN
      r:=right;
      PIVOT(left,r);
      QUICKSORT(left,r-1);
      QUICKSORT(r+1,right);
    END;
END;
(* main program *)
.....................
  x[n+1]:=MAXINT;
  QUICKSORT(1,n);
.....................
```

**Cod.Ps.2 QUICKSORT**

Per particolari permutazioni iniziali dei dati uno degli elementi coinvolti negli ultimi due confronti può essere lo stesso perno ovvero $x[r+1]$ corrispondente al perno finale del livello precedente (o a $MAXINT$ se il livello attuale è il primo). Ad esempio in $PIVOT(1,2)$ il pivot 13 è confrontato due volte con il 18 ed una volta con se stesso; in $PIVOT(4,5)$ il pivot è confrontato due

volte con il 26 ed una volta col (42) perno finale del livello precedente.

La Fig.5 rappresenta l'albero delle chiamate ricorsive prodotte da *QUICKSORT* applicata all' insieme della **Tab.C**. In ogni nodo figurano, racchiusi in parentesi, i parametri effettivi delle diverse chiamate di *QUICKSORT* (e di *PIVOT*); il terzo numero fornisce la locazione del perno all'uscita di *PIVOT* e non è presente nei nodi terminali perché, come si è detto, quando $left = right$ *PIVOT* non è chiamata. L'albero va visitato in ordine binario anticipato o previsita (esame della radice, previsita del sottoalbero sinistro, previsita del sottoalbero destro).

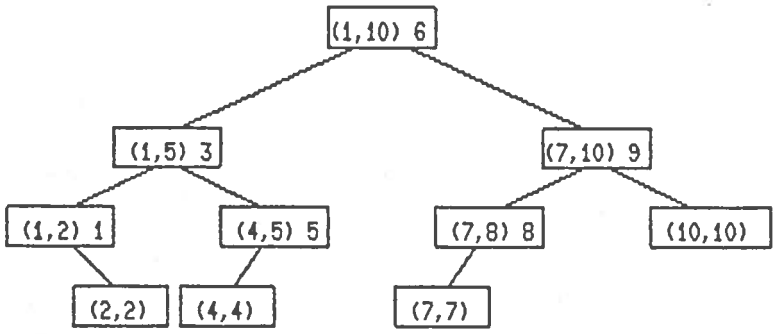


**Fig.5** *Albero binario delle chiamate ricorsive prodotte dalla procedura QUICKSORT applicata all'insieme della* **Tab.C**. *Il terzo numero fornisce la locazione del perno all'uscita della procedura PIVOT.*

I due confronti oltre gli $r - j$ possono anche non verificarsi (per esempio nella chiamata $PIVOT(1, 10)$ nessun elemento è confrontato due volte con il pivot; avvengono in totale 9 confronti), ma ciò non cambia sostanzialmente la valutazione di $C(n)$.

La formulazione ricorsiva di *QUICKSORT* ci permette di valutare abbastanza agevolmente il numero medio di confronti $C_M(n)$ richiesti per ordinare un array di $n$ elementi. Sotto l'ipotesi che le $n!$ possibili configurazioni iniziali dei dati siano equiprobabili, in ogni chiamata $PIVOT(j, r)$ il perno può trovarsi, all'uscita della procedura, in una qualunque posizione $k$ dell'intervallo $[j, r]$, con eguale probabilità $p = 1/(r - j + 1)$. Se $x[k]$ è il perno, $j \leq k \leq r$, *QUICKSORT* è chiamata ricorsivamente sui sottoinsiemi $X[j : k-1]$, di cardinalità $k-j$, ed $X[k+1 : r]$, di cardinalità $r-k$, per l'ordinamento dei quali occorreranno in media $C_M(k-j)$ e $C_M(r-k)$ confronti rispettivamente.

Ricordando che $PIVOT(1, n)$ esegue in genere $n + 1$ confronti (per particolari permutazioni iniziali dei dati ne esegue, come si è detto, $n$ o $n - 1$) possiamo scrivere:

$$C_M(n) = n + 1 + \frac{1}{n}\sum_{k=1}^{n}[C_M(k-1) + C_M(n-k)] = n + 1 + \frac{2}{n}\sum_{k=0}^{n-1} C_M(k) \quad n > 1 \qquad (2)$$

con le condizioni iniziali:

$$C_M(0) = C_M(1) = 0 \qquad (3)$$

Si tratta di una relazione di ricorrenza (la funzione $C_M(n)$ è infatti definita in termini di se stessa su valori decrescenti di $n$), tipica di molti algoritmi in cui si presenta un elemento disposto a caso

su $n$ elementi. Riscritta per $n-1$ invece di $n$ la (2) diviene:

$$C_M(n-1) = n + \frac{2}{n-1}\sum_{k=0}^{n-2} C_M(k)$$

da cui

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

Esplicitiamo $C_M(n)$ ed utilizzando l'espressione ottenuta per $C_M(n)$ valutiamo $C_M(n-1), C_M(n-2), \ldots, C_M(2)$:

$$\begin{aligned}
C_M(n) &= 2 + \frac{n+1}{n}C_M(n-1) = \\
&= 2 + \frac{n+1}{n}\left[2 + \frac{n}{n-1}C_M(n-2)\right] = \\
&= 2(n+1)\left[\frac{1}{n+1} + \frac{1}{n}\right] + \frac{n+1}{n-1}\left[2 + \frac{n-1}{n-2}C_M(n-3)\right] = \\
&\vdots \\
&= 2(n+1)\left[\frac{1}{n+1} + \frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{4}\right] + \frac{n+1}{3}\left[2 + \frac{3}{2}C_M(1)\right] = \\
&= 2(n+1)\sum_{k=3}^{n+1}\frac{1}{k}
\end{aligned}$$

Poiché

$$\sum_{k=3}^{n+1}\frac{1}{k} < \int_2^{n+1}\frac{1}{x}dx = \ln(n+1) - \ln 2 < \ln(n+1)$$

si ottiene

$$C_M(n) = 2(n+1)\sum_{k=3}^{n+1}\frac{1}{k} < 2(n+1)\ln(n+1) = \frac{2}{\log e}(n+1)\log(n+1) = O(n\log n) \qquad (4)$$

(qui e nel seguito i simboli $\ln x$ ed $\log x$ indicano rispettivamente il logaritmo naturale ed il logaritmo in base 2 di $x$). Ciò significa che il numero medio di confronti eseguiti da *QUICKSORT* cresce al più come $n\log n$ e dunque mediamente l'algoritmo è ottimo perché la sua complessità in tempo raggiunge il limite inferiore $\Omega(n\log n)$ stabilito dall'albero di decisione.

Per comprendere come mai *QUICKSORT* funzioni così bene nel caso medio cerchiamo di individuare il caso pessimo ed il caso ottimo.

Abbiamo già accennato al fatto che il bilanciamento nella partizione dei dati migliora l'efficienza degli algoritmi impostati con la tecnica del *divide et impera*, in quanto è l'albero di decisione perfettamente (o quasi perfettamente) bilanciato a minimizzare il numero medio ed il numero massimo di nodi di decisione disposti nei percorsi radice-foglia.

Il caso ottimo di *QUICKSORT* si avrà allora quando in tutte le chiamate il perno si sposta, all'uscita della procedura $PIVOT(j,r)$, nella locazione mediana dell'insieme $X[j:r]$, dividendo quest'ultimo in due insiemi di eguale cardinalità. Supponendo, per semplificare il calcolo, che sia $n = 2^h - 1$, il meccanismo ricorsivo nel caso ottimo presenta $h-1$ livelli, con $h = \log(n+1)$.

Al primo livello di ricorsione $QUICKSORT(1,n)$ chiama $PIVOT(1,n)$, che riloca il perno $x[1]$ nella posizione $[2^{h-1}]$, ripartendo $X[1:n]$ nei due arrays $X_1[1:2^{h-1}-1]$ ed $X_2[2^{h-1}+1:n]$ di $2^{h-1}-1$ elementi. Ad ogni livello $\lambda$ di ricorsione $\lambda = 1,\ldots,h-1$ le $2^{\lambda-1}$ chiamate di $QUICKSORT$ costruiscono complessivamente $2\cdot 2^{\lambda-1} = 2^{\lambda}$ sottoinsiemi di cardinalità $2^{h-\lambda}-1$, eseguendo ciascuna $(n+1)/2^{\lambda-1}$ confronti, per un totale di $n+1$ confronti nel livello. Il procedimento ricorsivo si arresta a livello $h-1$ dove le $2^{h-2}$ chiamate di $QUICKSORT$ costruiscono $2^{h-1}$ insiemi di un elemento, evidentemente già ordinati e sui quali perciò non è chiamata $PIVOT$.

Il numero $C_O(n)$ di confronti eseguiti nel caso ottimo da $QUICKSORT$ è dunque dato da

$$C_O(n) = (n+1)(h-1) = (n+1)[\log(n+1) - 1] \qquad n > 1 \tag{5}$$

e si può ottenere anche dalla relazione di ricorrenza che esprime il numero $C(n)$ di confronti effettuati per ordinare $n$ elementi come somma degli $n+1$ confronti richiesti da $PIVOT(1,n)$ e dei confronti occorrenti per l'ordinamento di $X[1:k-1]$ e di $X[k+1:n]$, se $x[k]$ è il perno all'uscita di $PIVOT(1,n)$:

$$C(n) = n + 1 + C(k-1) + C(n-k) \qquad n > 1 \tag{6}$$

Basta imporre $C(k-1) = C(n-k) = C((n-1)/2)$ e risolvere la ricorrenza con le condizioni iniziali (3).

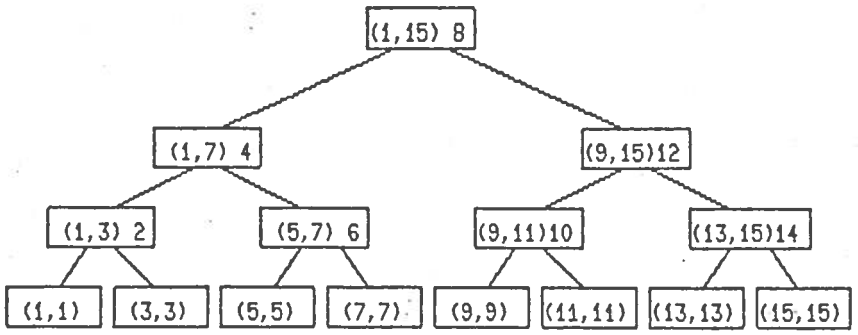


**Fig.6** *Albero binario perfettamente bilanciato delle chiamate ricorsive prodotte da QUICKSORT applicata ad un insieme di 15 elementi nel caso ottimo (per esempio alla sequenza 36 17 16 14 25 21 28 20 51 39 43 40 63 58 70).*

La **Fig.6** rappresenta l'albero delle chiamate ricorsive prodotte nel caso ottimo dalla procedura $QUICKSORT$ applicata ad un insieme di 15 elementi, come la sequenza seguente, che nei quattro livelli di iterazione subisce le trasformazioni indicate (in parentesi quadre sono racchiuse le sezioni di $X$ sulle quali agiscono le varie chiamate; i corrispondenti perni sono sottolineati mentre i perni del livello precedente sono racchiusi in parentesi tonde):

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| [36 | 11 | 16 | 14 | 25 | 21 | 28 | 20 | 51 | 39 | 43 | 40 | 63 | 58 | 70] |
| [20 | 11 | 16 | 14 | 25 | 21 | 28] | (36) | [51 | 39 | 43 | 40 | 63 | 58 | 70] |
| [14 | 11 | 16] | (20) | [25 | 21 | 28] | 36 | [40 | 39 | 43] | (51) | [63 | 58 | 70] |
| [11] | (14) | [16] | 20 | [21] | (25) | [28] | 36 | [39] | (40) | [43] | 51 | [58] | (63) | [70] |

sinistro si porta da $x[2]$ ad $x[n+1] = MAXINT$ mentre quello destro rimane fermo in $x[n]$; il perno $x[1]$, che è il massimo dell'insieme, si sposta nell'ultima posizione dividendo $X[1:n]$ nei due insiemi $X_1[1:n-1]$ ed $X_2$ di cardinalità nulla. Ad ogni successivo livello di ricorsione il perno $x[1]$ si sposta nell'ultima posizione $[r]$ dividendo $X[1:r]$ nei due insiemi $X_1[1:r-1]$ ed $X_2$ vuoto, per $r = n, \ldots, 2$. Indicando con $C_p(n)$ il numero di confronti occorrenti per ordinare $n$ elementi nel caso pessimo avremo allora:

$$C_p(n) = \sum_{k'=3}^{n+1} k' = \frac{3+n+1}{2}(n-1) \simeq \frac{n^2}{2} = O(n^2) \qquad n > 1 \tag{7}$$

risultato ottenibile anche dalla (6) ponendovi $k = 1$ e risolvendo con le condizioni iniziali (3).

*QUICKSORT* è dunque inefficiente nel caso pessimo mentre il suo funzionamento nel caso medio è schematizzabile mediante un albero di decisione quasi perfettamente bilanciato o comunque così vicino a tale forma da mantenere l'andamento $n \log n$ della funzione $C_M(n)$.

Se fosse possibile costruire un algoritmo che operi ricorsivamente su sottoinsiemi dei dati sempre bilanciati esso risulterebbe ottimo per qualunque permutazione di input. Gli ultimi due algoritmi di ordinamento che presenteremo soddisfano appunto a tale condizione. Prima di esaminarli vogliamo però spendere ancora qualche parola sul funzionamento del meccanismo ricorsivo in una procedura.

Quando un linguaggio ammette la ricorsività, ad ogni successiva chiamata ricorsiva, interna alla precedente, diviene attiva una nuova esecuzione della procedura, nel cui ambito vengono generate nuove versioni delle variabili locali che vi figurano, distinte dalle omonime variabili relative alle esecuzioni lasciate in sospeso; ciò consente il salvataggio dei risultati intermedi. Ogni parametro di una procedura ricorsiva (per esempio *left* o *right* in *QUICKSORT*) può quindi assumere contemporaneamente valori distinti, in relazione a chiamate diverse non ancora completate. Quando è raggiunta la condizione di chiusura del meccanismo ricorsivo (nel nostro caso quando $left \geq right$), si completano le esecuzioni interrotte, cominciando dalle chiamate più interne; il valore di una variabile, impiegato in una data esecuzione, al completamento di quest'ultima viene rilasciato, per essere sostituito dal valore utilizzato nella precedente chiamata da completare.

Il meccanismo implicitamente usato dall'elaboratore per la memorizzazione dei valori dei parametri è quello caratteristico della pila (*stack* in inglese), struttura informativa ad una dimensione, dinamicamente variabile, con disciplina di accesso *LIFO* (*last in first out*), in cui cioè l'ultimo elemento inserito è il primo ad essere estratto. Una pila è sostanzialmente costituita da una sequenza ordinata di elementi in cui l'inserzione o l'estrazione di un elemento può essere effettuata ad una sola estremità, detta testa (*top*) della pila, ed è dunque una struttura utilissima per memorizzare dei dati in forma temporanea, per esempio gli indirizzi di ritorno dei sottoprogrammi o i valori degli indici contatori in più cicli nidificati.

Ogniqualvolta la risoluzione di un problema (o l'esecuzione di un calcolo) viene temporaneamente sospesa per affrontare un sottoproblema, che ha origine nell'ambito del primo e che a sua volta può generare altri sottoproblemi, i valori dei dati relativi ai problemi (o ai calcoli) interrotti vengono inseriti in una pila; l'ultimo problema generato si porta in testa alla pila dei problemi da risolvere e dovrà essere completato prima dei precedenti. I linguaggi ad alto livello che ammettono la ricorsività gestiscono una pila per ogni argomento e variabile locale.

Se il linguaggio che si intende usare non consente la ricorsione o se il costo di quest'ultima è elevato nel compilatore disponibile, simulando il meccanismo della pila, che in una procedura ricorsiva è completamente gestito dall'elaboratore, è comunque possibile trasformare un algoritmo ricorsivo, di cui siano già state provate la correttezza e l'efficienza, in uno equivalente che usi solo l'iterazione.

Esistono delle regole generali di trasformazione che, applicate fedelmente, garantiscono l'equivalenza semantica delle due versioni, ricorsiva ed iterativa, di una procedura:

a) Al principio della procedura si inserisce un array, inizialmente vuoto, che funge da pila.

Ogni chiamata ricorsiva della procedura è sostituita da un insieme di istruzioni che effettuano:

b) la memorizzazione nell'array pila sia dei valori dei parametri e delle variabili locali, relativi all'esecuzione da sospendere, sia di una etichetta di ritorno che, assegnata alla prima istruzione eseguibile dopo la chiamata ricorsiva, indica il punto in cui riprenderà la computazione interrotta;
c) la valutazione dei parametri effettivi della chiamata ricorsiva e la relativa assegnazione ai parametri formali della procedura;
d) un salto all'inizio della procedura per avviare l'esecuzione interna.

Ogni uscita dalla procedura originale è sostituita da un altro insieme di istruzioni che eseguono:

e) un controllo sullo stato della pila, interrompendo l'esecuzione se la pila è vuota;
f) l'estrazione dalla pila sia della "label" di ritorno (per rientrare nella procedura nel punto di interruzione della chiamata sospesa) sia dei valori memorizzati per le variabili locali ed i parametri, ai quali vengono riassegnati;
g) un salto all'istruzione indicata dalla label di ritorno;

In pratica tali regole sono sovente semplificabili, in concomitanza di particolari situazioni. Per esempio per sostituire una chiamata ricorsiva che figuri come ultima istruzione eseguibile di una procedura la pila non è necessaria, basta valutare i nuovi valori dei parametri ed inserire una istruzione di salto all'inizio della procedura (regole c e d); o ancora se c'è un unico punto di rientro nella procedura si possono evitare le labels di ritorno.

Nel caso specifico di $QUICKSORT$, che nel caso pessimo presenta $n-1$ livelli di ricorsione, per la memorizzazione temporanea dei parametri effettivi di $left$ e $right$ relativi alle successive chiamate ricorsive della procedura occorrerebbe un array pila di dimensione massima $2n-2 = O(n)$. Tale spazio di memoria può tuttavia essere ridotto ad $O(\log n)$ usando una versione iterativa, formalizzata in **Cod.Ps.2a**, nella quale il più piccolo dei due sottoinsiemi in cui il pivot suddivide $X[left : right]$ è sempre ordinato per primo mentre le due locazioni, iniziale e finale, dell'altro sottoinsieme vengono memorizzati in un array ausiliario $st[1 : kmax]$ che funge da pila. Per determinare il valore da assegnare a $kmax$ osserviamo che lo spazio $S(n)$ richiesto dalla pila nell'ordinamento di $n$ elementi risulterà massimo nel caso in cui in tutti i livelli di iterazione la procedura $PIVOT$ divide $X[left : right]$ in sottoinsiemi di eguale cardinalità, allocando il perno nella locazione mediana $[(left + right)/2]$. Si ha allora la ricorrenza

$$S(0) = S(1) = 0; \qquad S(n) \leq 2 + S(\lfloor (n-1)/2 \rfloor) \qquad n > 1$$

dunque, se $2^{h-1} < n \leq 2^h - 1$, sarà

$$S(n) \leq S(2^h - 1) \leq 2 + S(2^{h-1} - 1) \leq \cdots \leq 2j + S(2^{h-j} - 1) \leq \cdots \leq 2(h-1) + S(1) = 2\lfloor \log n \rfloor$$

cioè la pila usata in tale procedura iterativa non conterrà mai più di $2\lfloor \log n \rfloor$ elementi. Se dunque $nmax$ è la massima dimensione di $X$ porremo $kmax$ eguale al doppio della parte intera del logaritmo in base 2 di $nmax$. Al solito $X$ ed $n$ sono dichiarati globalmente come pure le costanti $nmax$ e $kmax$. La variabile $k$, inizializzata a zero, punta al *top* della pila e varia di $\pm 2$ secondo che si inseriscano (o si estraggano) i parametri effettivi di $left$ e $right$ relativi all'esecuzione da sospendere (o da riprendere). Il salto all'inizio della procedura è realizzato con il ciclo di $WHILE\ k >= 0$. All'interno del ciclo $WHILE\ left < right$ nella versione ricorsiva figurano, dopo $PIVOT(left, r)$, le due chiamate di $QUICKSORT$ su $X[left : r-1]$ ed $X[r+1 : right]$. Nella versione iterativa l'$IF\ r - left < right - r$ confronta le cardinalità dei due arrays memorizzando nella pila le locazioni estreme dell'array con più elementi. Dopo l'assegnazione $right := r - 1$ (o $left := r + 1$) e l'incremento di $+2$ del puntatore $k$ si procede all'ordinamento dell'array più piccolo con la stessa tecnica di calcolo. Quando $right <= left$ se nella pila si trovano dei valori memorizzati ($IF\ k > 0$) essi vengono estratti a coppie, consentendo l'ordinamento delle corrispondenti sezioni di $X$. L'algoritmo ha termine quando la pila è trovata vuota dal $WHILE\ k >= 0$ e risulta $right < left$.

(**Dichiarati globalmente:* $X[1 : nmax]$, *n e le costanti nmax e* $kmax = 2\lfloor \log nmax \rfloor$***)

```
PROCEDURE QUICKSORT(left,right:integer);
(* iterative version *)
VAR r,k:integer;
    st:array [1..kmax] of integer;
BEGIN
  k:=0;
  WHILE k>=0 DO
    BEGIN
      WHILE left<right DO
        BEGIN
          r:=right; PIVOT (left,r);
          IF r-left < right-r THEN
            BEGIN
                st[k+1]:=r+1; st[k+2]:=right; right:=r-1;
            END
                    ELSE
            BEGIN
                st[k+1]:=left; st[k+2]:=r-1; left:=r+1;
            END;
          k:=k+2;
        END;
      IF k>0 THEN
        BEGIN
          right:=st[k]; left:=st[k-1];
        END;
      k:=k-2;
    END;
END;
```

**Cod.Ps.2a Versione iterativa di QUICKSORT**

L'insieme della **Tab.C** al quale venga applicata la versione iterativa di $QUICKSORT$ subirà

le seguenti trasformazioni

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| [42 | 52 | 13 | 63 | 26 | 57 | 20 | 18 | 31 | 47] | (1,10) | st: - |
| {20 | 31 | 13 | 18 | 26} | 42 | [57 | 63 | 52 | 47] | (7,10) | st:1,5 |
| {20 | 31 | 13 | 18 | 26} | 42 | {52 | 47} | 57 | [63] | (10,10) | st:1,5,7,8 |
| {20 | 31 | 13 | 18 | 26} | 42 | [52 | 47] | 57 | 63 | (7,8) | st:1,5 |
| {20 | 31 | 13 | 18 | 26} | 42 | [47] | 52 | 57 | 63 | (7,7) | st:1,5 |
| [20 | 31 | 13 | 18 | 26] | 42 | 47 | 52 | 57 | 63 | (1,5) | st: - |
| {13 | 18} | 20 | [31 | 26] | 42 | 47 | 52 | 57 | 63 | (4,5) | st:1,2 |
| {13 | 18} | 20 | [26] | 31 | 42 | 47 | 52 | 57 | 63 | (4,4) | st:1,2 |
| [13 | 18] | 20 | 26 | 31 | 42 | 47 | 52 | 57 | 63 | (1,2) | st: - |
| 13 | [18] | 20 | 26 | 31 | 42 | 47 | 52 | 57 | 63 | (2,2) | st: - |
| 13 | 18 | 20 | 26 | 31 | 42 | 47 | 52 | 57 | 63 | | |

Le parentesi quadre racchiudono l'array in corso di esame, le cui locazioni estreme figurano in parentesi tonde subito dopo gli elementi. Le sezioni di $X$ ancora da ordinare sono racchiuse in parentesi graffe. Dopo st: sono riportati i valori correntemente inseriti nella pila o "-" se la pila è vuota.

## 6. - MERGESORT.

Un altro elegante esempio della tecnica del "divide et impera" è costituito da un algoritmo di ordinamento, chiamato *MERGESORT* (ordinamento per fusione) il quale, operando ricorsivamente su partizioni dell'insieme sempre bilanciate, presenta complessità $O(n \log n)$ anche nel caso pessimo. Ne daremo preliminarmente una descrizione intuitiva.

Si è già detto del guadagno di efficienza ottenibile con un algoritmo (organizzato ricorsivamente) che determini la soluzione globale di un certo problema come ricombinazione delle soluzioni parziali di sottoproblemi, identici al problema dato ma applicati a sottoinsiemi dei dati di cardinalità eguali (o comunque molto vicine). La partizione ricorsiva dei sottoinsiemi prosegue fino al raggiungimento di insiemi elementari, sui quali il problema è direttamente risolvibile. Nel caso dell'ordinamento sono elementari gli insiemi costituiti da un solo elemento, evidentemente già ordinati. Confrontando due di essi si può banalmente generare per fusione un insieme ordinato di cardinalità due, avente come primo elemento quello che precede l'altro nella data relazione di ordinamento, nel nostro caso il più piccolo, essendo "$<$" l'usuale relazione di minore.

Generalizzando il procedimento di ricombinazione, dati due gruppi $X_1$ ed $X_2$ di elementi, che siano già stati separatamente ordinati, si può ottenere dalla loro fusione un unico insieme ordinato $Y$, di cardinalità $|Y| = |X_1| + |X_2|$, confrontando ordinatamente gli elementi correntemente in testa ad $X_1$ ed $X_2$ e ponendo il minore dei due nella prima locazione libera di $Y$, inizialmente vuoto. I confronti hanno termine quando, esaurito uno dei due gruppi dati, si allocano ordinatamente in fondo a $Y$ gli elementi eventualmente rimasti nell'altro gruppo.

Per esempio la fusione di $X_1 = \{6, 9, 13\}$ ed $X_2 = \{2, 8, 15, 19, 25\}$ nell'insieme ordinato $Y = \{2, 6, 8, 9, 13, 15, 19, 25\}$ ha inizio dal confronto fra il 6 ed il 2 e dall'assegnazione $y[1] := 2$. Il 6, rimasto in testa ad $X_1$, è minore di 8, nuovo elemento in testa ad $X_2$, ed è perciò copiato in $y[2]$. Procedendo similmente il confronto fra 9 ed 8 determina $y[3] := 8$, quello fra 9 e 15 implica $y[4] := 9$; l'ultimo confronto fra 13 e 15 trasferisce il 13 in $y[5]$, esaurendo $X_1$; gli elementi 15, 19 e 25, rimasti in $X_2$, sono allocati ordinatamente in $y[6]$, $y[7]$, $y[8]$.

```
PROGRAM MERGESORTING (.....);
TYPE elem=integer; CONST nmax=....;
VAR x,y:array[1..nmax] of elem; n,i:integer;
PROCEDURE COPY (kmin,kmax:integer;VAR j,q:integer);
VAR k:integer;
(* se j = q copia le chiavi di X[kmin : kmax] nell'array di servizio Y; *)
(* se kmin=kmax=q allora y[j] := x[q], q := q + 1.   *)
BEGIN
  FOR k:=kmin TO kmax DO
    BEGIN
      y[j]:=x[k]; q:=q+1;
    END;
END;
PROCEDURE MERGE(left,mid,right:integer);
VAR l,j,k,m:integer;
(* fonde X[left : mid] ed X[mid + 1 : right] in un   *)
(* unico array ordinato costruito in Y e ricopiato in X  *)
BEGIN
  l:=left;j:=left;m:=mid+1;
  WHILE ((l<=mid) AND (m<=right)) DO
    BEGIN
      IF x[l]<x[m] THEN COPY(l,l,j,l)
        ELSE COPY(m,m,j,m);
      j:=j+1;
    END;
  IF l>mid THEN COPY(m,right,j,j)
    ELSE COPY(l,mid,j,j);
  FOR k:=left TO right DO x[k]:=y[k];
END;
PROCEDURE MERGESORT(left,right:integer);
VAR mid:integer;
(* ordina per fusione X[left : right] *)
BEGIN
  IF left<right THEN
    BEGIN
      mid:=(left+right)DIV 2;
      MERGESORT(left,mid);
      MERGESORT(mid+1,right);
      MERGE(left,mid,right);
    END;
END;
(* main program *)
.............................
      MERGESORT(1,n);
.............................
```

**Cod.Ps.3** **MERGESORT**

L'ordinamento di un insieme $X$ per fusione consiste allora nella suddivisione di $X$ in due sottoinsiemi $X_1$ ed $X_2$ con $|X_1| \simeq |X_2|$, ciascuno dei quali è a sua volta ripartito ricorsivamente in due sezioni di cardinalità differenti al più di uno. Durante tale partizione dei dati non sono effettuati confronti e non c'è dunque spostamento di chiavi; i confronti, gli scambi e le fusioni hanno inizio sugli insiemi elementari di un solo elemento, cui si giunge dopo un sufficiente numero di chiamate ricorsive, una interna all'altra, di *MERGESORT*. Il procedimento di ricombinazione

è effettuato dal basso verso l'alto: gli insiemi ordinati di due elementi, generati dalla fusione degli insiemi elementari, sono ricombinati in insiemi ordinati di quattro elementi e così via fino all'ordinamento dell'intero insieme.

In **Cod.Ps.3** è riportata la codifica *PASCAL* di *MERGESORT*.

In $QUICKSORT(left, right)$ la partizione dell'insieme $X[left : right]$ è effettuata in modo tale che i due sottoinsiemi $X[left : r-1]$ ed $X[r+1 : right]$, separati dall'elemento $x[r]$ scelto come pivot, una volta ordinati non debbano essere ricombinati, in quanto tutti gli elementi del primo sottoinsieme risultano minori di tutti quelli del secondo. $MERGESORT(left, right)$ invece divide l'array $X[left : right]$ in due sezioni $X[left : mid]$ ed $X[mid+1 : right]$, con $mid = \lfloor (left + right)/2 \rfloor$, le ordina ricorsivamente, indipendentemente l'una dall'altra, quindi le fonde invocando $MERGE$.

La procedura di ricombinazione $MERGE(left, mid, right)$ opera confrontando iterativamente i due elementi più piccoli dei due arrays già ordinati $X[left : mid]$ ed $X[mid+1 : right]$ ed allocandone il minore dei due in un array di servizio $Y[left : right]$; quando si raggiunge l'ultimo elemento di uno dei due arrays, gli elementi rimasti nell'altro array vengono ordinatamente copiati nelle locazioni ancora libere di $Y$. Infine gli elementi ormai ordinati di $Y[left : right]$ sono ricopiati in $X[left : right]$. (Gli arrays $X$ ed $Y$, la loro cardinalità $n$ e la dimensione massima $nmax$ sono dichiarati globalmente).

La procedura $COPY(kmin, kmax, j, q)$, che effettua la copiatura delle chiavi, serve a snellire la formalizzazione, evitando ripetizioni di istruzioni. Se $j = q$ la procedura esegue $y[j] := x[k]$ per $j = q, ..., q + kmax - kmin$ e $k = kmin, ..., kmax$, cioè copia le chiavi di $X[kmin : kmax]$ nell'array ausiliario $Y$; se invece $kmin = kmax = q$ allora $COPY$ alloca in $y[j]$ la chiave di $x[q]$ ed incrementa $q$ di uno.

La prima chiamata $MERGESORT(1, n)$, inserita nel programma principale, dà inizio all'ordinamento dell'intero insieme.

La **Tab.D** illustra un esempio concreto di funzionamento su un insieme di 9 elementi; le sbarre verticali indicano le successive partizioni e fusioni dei dati. Le trasformazioni successive dell'insieme sono indicate nell'ordine di esecuzione della **Cod.Ps.3**. Sotto `MERGESORT` e `MERGE` sono riportati i parametri effettivi delle rispettive chiamate; le sequenze di tali chiamate sono rappresentate nei due alberi di **Fig.8**, da visitare in ordine binario anticipato e posticipato rispettivamente. La procedura $MERGESORT$, chiamata per la prima volta nel programma principale coi parametri $(1, 9)$, divide $X[1 : 9]$ nei due arrays $X[1 : 5]$ ed $X[6 : 9]$; quindi, sospeso temporaneamente l'ordinamento di $X[6 : 9]$, ripartisce $X[1 : 5]$ in $X[1 : 3]$ ed $X[4 : 5]$. Procedendo analogamente $X[1 : 3]$ è suddiviso in $X[1 : 2]$ ed $X[3 : 3]$ (insieme elementare costituito da $x[3] = 12$) ed ancora $X[1 : 2]$ è ripartito nei due insiemi elementari $X[1 : 1] = x[1] = 29$ ed $X[2 : 2] = x[2] = 16$. A questo punto è chiamata per la prima volta, coi parametri $(1, 1, 2)$, la procedura $MERGE$ che fonde $x[1]$ ed $x[2]$ nell'array ordinato $X[1 : 2] = |16, 29|$. Successivamente $MERGE(1, 2, 3)$ fonde $X[1 : 2]$ con $x[3]$ generando $X[1 : 3] = |12, 16, 29|$. Si passa adesso all'ordinamento (che era stato temporaneamente sospeso) di $X[4 : 5]$, che viene suddiviso nei due insiemi elementari $x[4] = 62$ e $x[5] = 21$, fusi poi da $MERGE(4, 4, 5)$ nell'array ordinato $X[4 : 5] = |21, 62|$. Finalmente, dopo la fusione di $X[1 : 3]$ ed $X[4 : 5]$ prodotta da $MERGE(1, 3, 5)$, si ritorna alla prima invocazione di $MERGESORT$ e si ordina ricorsivamente allo stesso modo $X[6 : 9]$.

**Tab.D** **Esempio di funzionamento di MERGESORT**

| x[1] | x[2] | x[3] | x[4] | x[5] | x[6] | x[7] | x[8] | x[9] | MERGESORT | MERGE |
|---|---|---|---|---|---|---|---|---|---|---|
| \|29 | 16 | 12 | 62 | 21\| | \|56 | 38 | 35 | 49\| | (1,9) | |
| \|29 | 16 | 12\| | \|62 | 21\| | | | | | (1,5) | |
| \|29 | 16\| | \|12\| | | | | | | (1,3) | |
| \|29\| | \|16\| | | | | | | | | (1,2) | |
| \|29\| | | | | | | | | | (1,1) | |
| | \|16\| | | | | | | | | (2,2) | |
| \|16 | 29\| | | | | | | | | | (1,1,2) |
| | | \|12\| | | | | | | | (3,3) | |
| \|12 | 16 | 29\| | | | | | | | | (1,2,3) |
| | | | \|62\| | \|21\| | | | | | (4,5) | |
| | | | \|62\| | | | | | | (4,4) | |
| | | | | \|21\| | | | | | (5,5) | |
| | | | \|21 | 62\| | | | | | | (4,4,5) |
| \|12 | 16 | 21 | 29 | 62\| | | | | | | (1,3,5) |
| | | | | | \|56 | 38\| | \|35 | 49\| | (6,9) | |
| | | | | | \|56\| | \|38\| | | | (6,7) | |
| | | | | | \|56\| | | | | (6,6) | |
| | | | | | | \|38\| | | | (7,7) | |
| | | | | | \|38 | 56\| | | | | (6,6,7) |
| | | | | | | | \|35\| | \|49\| | (8,9) | |
| | | | | | | | \|35\| | | (8,8) | |
| | | | | | | | | \|49\| | (9,9) | |
| | | | | | | | \|35 | 49\| | | (8,8,9) |
| | | | | | \|35 | 38 | 49 | 56\| | | (6,7,9) |
| 12 | 16 | 21 | 29 | 35 | 38 | 49 | 56 | 62 | | (1,5,9) |

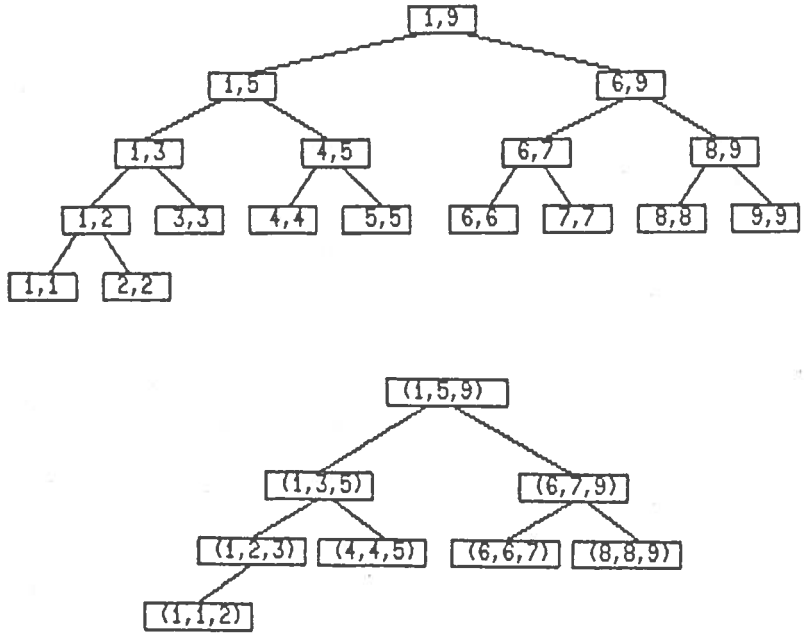


**Fig.8** *Alberi delle chiamate di MERGESORT (ricorsiva) e di MERGE per n=9.*

Ancora una volta il numero di confronti $C(n)$ richiesti da $MERGESORT$ per ordinare $n$ elementi ci fornisce una misura della complessità dell'algoritmo. Per valutarlo osserviamo preliminarmente che il numero $M(n)$ di confronti eseguiti da $MERGE$ per fondere due arrays di cardinalità complessiva $n$ è minore o eguale ad $n-1$. Se $n = 2^h$ allora $C(n)$ è esprimibile mediante la ricorrenza:

$$C(0) = C(1) = 0; \qquad C(n) = 2C\left(\frac{n}{2}\right) + M(n) \qquad n > 1$$

da cui, mediante sostituzioni successive, si ottiene

$$\begin{aligned} C(n) &\leq 2C\left(\frac{n}{2}\right) + n - 1 \leq 2\left[2C\left(\frac{n}{4}\right) + \frac{n}{2} - 1\right] + n - 1 = 4C\left(\frac{n}{4}\right) + 2n - 2 - 1 \leq \\ &\leq 4\left[2C\left(\frac{n}{8}\right) + \frac{n}{4} - 1\right] + 2n - 2 - 1 = 8C\left(\frac{n}{8}\right) + 3n - 4 - 2 - 1 \leq \cdots \leq \\ &\leq 2^h C\left(\frac{n}{2^h}\right) + hn - \sum_{j=0}^{h-1} 2^j = hn - (2^h - 1) = n\log n - (n-1) \end{aligned}$$

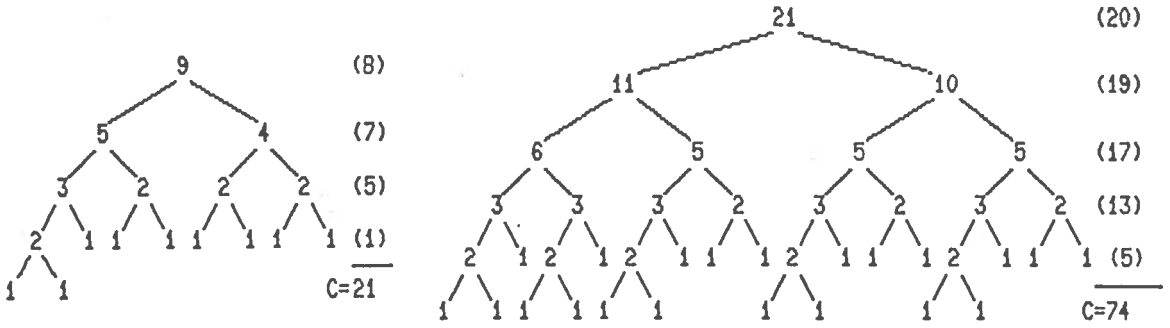


**Fig.9** *Cardinalità delle sezioni di X ordinate da MERGESORT (ricorsiva) per n=9 ed n=21.*

Si verifica agevolmente che ogni elemento in più, oltre i $2^h$ già considerati, comporta al massimo $h+1$ ulteriori confronti. Per convincersene si pensi di disporre nei nodi di un albero binario le cardinalità degli arrays su cui ricorsivamente è chiamata $MERGESORT$ e quindi $MERGE$ (in **Fig.9** sono raffigurati gli alberi relativi ad $n = 9$ ed $n = 21$, fra parentesi sono indicati i confronti eseguiti nel caso pessimo ad ogni livello). Per $n = 2^h$ l'albero risulta perfettamente bilanciato ed ogni suo nodo a livello $j$, con $j = 0, \ldots, h$ avrà il valore $2^{h-j}$. Per ogni elemento aggiunto in $X$ l'albero presenterà una foglia in più e lungo il percorso "radice-nuova foglia" si troveranno $h+1$ nodi con i valori aumentati di una unità.

In generale se $n = 2^h + k$ con $0 \leq k \leq 2^h - 1$ si avrà:

$$C(n) = C(2^h + k) \leq C(2^h) + (h+1)k \leq h2^h - (2^h - 1) + (h+1)k = h(2^h + k) + k - (2^h - 1)$$

cioè

$$C(n) \leq n\lfloor\log n\rfloor + k - (2^h - 1) \leq n\lfloor\log n\rfloor \qquad n = 2^h + k \qquad 0 \leq k \leq 2^h - 1 \qquad (8)$$

Poiché $C(n) \leq n\lfloor\log n\rfloor$ (l'uguaglianza può essere verificata solo se $k = 2^h - 1$ cioè se $n = 2^{h+1} - 1$) la complessità di $MERGESORT$ è $O(n\log n)$.

La versione iterativa che proponiamo in **Cod.Ps.3a** evita il procedimento di memorizzazione in una pila dei parametri delle successive chiamate di $MERGESORT$ ed effettua direttamente il processo di ricombinazione partendo dal basso. Ad ogni livello $j$ di iterazione, $j = 1, 2.., \lfloor \log n \rfloor$, si fondono a due a due gli insiemi generati nel livello precedente; nel caso in cui il numero di tali insiemi risulti dispari, l'ultimo di essi è fuso con l'ultima coppia formata nel livello corrente.

```
PROCEDURE MERGESORT; (* iterative version *)
VAR l,j,k,l2n,p2,pot2j:integer;
(* X ed n globali; l2n=⌊log n⌋; pot2j=2^j; p2=2^(j-1)  *)
BEGIN
  l2n:= trunc(LN(n)/LN(2)); pot2j:=1;
  FOR j:=1 TO l2n DO
    BEGIN
      l:=1;p2:=pot2j;pot2j:=pot2j*2;
      FOR k:=1 TO n DIV pot2j-1 DO
        BEGIN
          MERGE(l,l+p2-1,l+pot2j-1); l:=l+pot2j;
        END;
      IF ((n DIV p2)MOD 2)=1 THEN
        BEGIN
          MERGE(l,l+p2-1,l+pot2j-1);
          MERGE(l,l+pot2j-1,n);
        END
              ELSE MERGE(l,l+p2-1,n);
    END;
END;
```

**Cod.Ps.3a** **Versione iterativa di MERGESORT**

La chiamata $MERGESORT$ attiva la procedura (non ci sono parametri formali). Al primo livello d' iterazione ($j = 1$) gli elementi di $X$ sono fusi a coppie in $\lfloor n/2 \rfloor$ arrays di cardinalità due; se $n$ è dispari l'elemento rimasto è ricombinato con l'ultimo array generato. Al passo successivo si fondono a coppie gli insiemi di due elementi (l'ultimo può averne tre) in $\lfloor n/4 \rfloor$ insiemi di cardinalità quattro, l'ultimo dei quali può avere cardinalità cinque se $\lfloor n/2 \rfloor$ è pari ovvero viene fuso con gli elementi residui se $\lfloor n/2 \rfloor$ è dispari. Così continuando si perviene all'ordinamento dell'intero insieme. Ad ogni livello $j$ di iterazione, $j = 1, 2.., \lfloor \log n \rfloor$, la variabile $l$, inizializzata ad 1, scorrendo l'insieme $X$ con passo $2^j$, individua le locazioni iniziali degli $\lfloor n/2^j \rfloor - 1$ insiemi ordinati $X[l : l + 2^j - 1]$ di cardinalità $2^j$, che la procedura $MERGE(l, l + 2^{j-1} - 1, l + 2^j - 1)$ genera per fusione delle coppie di arrays di $2^{j-1}$ elementi $X[l : l + 2^{j-1} - 1]$ ed $X[l + 2^{j-1} : l + 2^j - 1]$ con $l = 1 + m2^j$ per $m = 0, ..., \lfloor n/2^j \rfloor - 2$. Dopo tali $\lfloor n/2^j \rfloor - 1$ fusioni la variabile cursore ha il valore $l = 1 + 2^j(\lfloor n/2^j \rfloor - 1)$. Se $\lfloor n/2^{j-1} \rfloor$ è dispari si fondono ancora due arrays di cardinalità $2^{j-1}$ ed all'array di $2^j$ elementi così generato si uniscono gli elementi residui con la chiamata $MERGE(l, l + 2^j - 1, n)$; se invece $\lfloor n/2^{j-1} \rfloor$ è pari si fonde l'ultima coppia di insiemi del livello precedente con la chiamata $MERGE(l, l + 2^{j-1} - 1, n)$.

È facile verificare che tale procedura risulta semanticamente equivalente a quella ricorsiva per $n = 2^h$, con $C(n) \leq \sum_{j=1}^{h} 2^{h-j}(2^j - 1) = h2^h - (2^h - 1)$.

Nel caso generale rimane valida la (8); infatti, se $h$ sono i livelli d'iterazione ogni elemento

aggiunto ai primi $2^h$ comporta ancora $h+1$ ulteriori confronti al massimo.

In **Tab.E** sono riportati i parametri effettivi di $MERGE$ e le sezioni di $X$ corrispondentemente ordinate ad ogni livello d'iterazione per $n = 9$. In **Fig.10** sono indicate le cardinalità delle sezioni di $X$ ordinate da $MERGESORT$ iterativa per $n = 9$ ed $n = 21$.

**Tab.E** *Parametri effettivi di MERGE e sezioni di X corrispondentemente ordinate ad ogni livello di iterazione per n=9 (se MERGESORT è iterativa).*

| | | | | |
|---|---|---|---|---|
| $(1,1,2) \mapsto X[1:2]$ | $(3,3,4) \mapsto X[3:4]$ | $(5,5,6) \mapsto X[5:6]$ | $(7,7,8) \mapsto X[7:8]$ | $(7,8,9) \mapsto X[7:9]$ |
| $(1,2,4) \mapsto X[1:4]$ | | $(5,6,9) \mapsto X[5:9]$ | | |
| $(1,4,9) \mapsto X[1:9]$ | | | | |

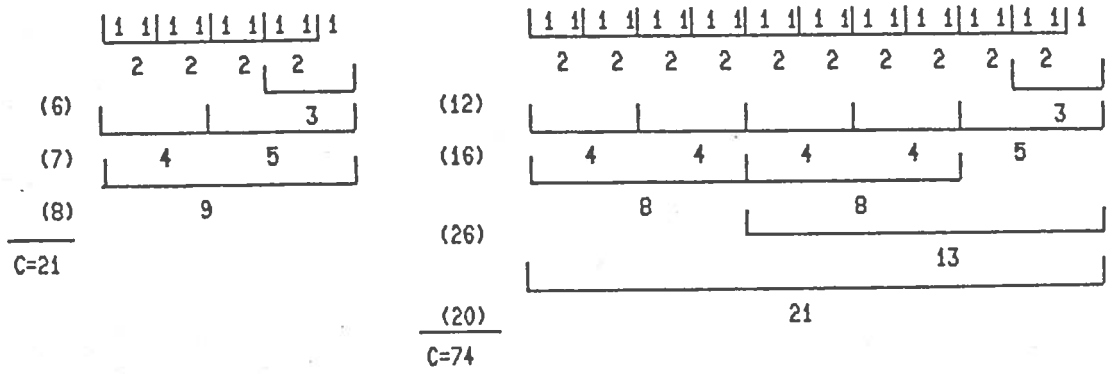


**Fig.10** *Cardinalità delle sezioni di X ordinate da MERGESORT (iterativa) per n=9 ed n=21.*

Si osservi comunque che i parametri effettivi delle chiamate di $MERGE$ (e dunque le cardinalità delle sezioni di $X$ via via ordinate) nelle due versioni ricorsiva ed iterativa sono diversi, pur rimanendo eguale il numero massimo di confronti eseguiti nel caso pessimo (si confrontino la **Fig.8** con la **Tab.E** e la **Fig.9** con la **Fig.10**).

$MERGESORT$ richiede nel caso pessimo e nel caso medio un tempo di computazione superiore al tempo medio speso da $QUICKSORT$ ed occupa maggior spazio di memoria, tuttavia la sua complessità rimane $O(n \log n)$ anche nel caso più sfavorevole. Di questa importante proprietà gode anche un altro algoritmo di ordinamento, oggetto del seguente paragrafo, che rispetto a $MERGESORT$ ha il vantaggio di richiedere poco spazio di memoria.

## 7. - HEAPSORT.

Presentiamo infine un importante algoritmo di ordinamento, basato su una struttura di dati derivata dall'albero binario e chiamata *heap* da J.W.J.Williams (1964) che per primo la introdusse.

*Uno heap è un albero binario completo fino al penultimo livello $k-1$, con le foglie del livello massimo $k$ tutte addossate a sinistra, e tra i suoi nodi è definita una relazione di ordinamento "$<$" tale che il valore allocato in un qualunque nodo risulti $<$ di quello contenuto nel nodo padre.*

Gli $n$ elementi di uno heap possono essere allocati sequenzialmente in un array $X[1:n]$, seguendo l'ordine in cui si presentano i nodi ove si percorra l'albero a partire dalla radice $x[1]$ e scorrendo i livelli successivi da sinistra a destra.

In **Fig.11** è riportato l'esempio di uno heap con nove elementi e la sua allocazione sequenziale.

Si noti che uno heap con $n$ nodi ha $\lfloor n/2 \rfloor$ nodi non terminali ed $\lceil n/2 \rceil$ foglie e che per la definizione di heap si ha:

$$x[i] < x\left[\lfloor i/2^h \rfloor\right] \qquad 1 < i \leq n \quad 1 \leq h \leq \lfloor \log i \rfloor \tag{9}$$

In particolare risulta $x[i] < x[1]$, cioè la radice contiene l'elemento massimo. Fissato il numero di nodi $n$ con $2^k \leq n < 2^{k+1}$ risulta fissata la forma dello heap, con $k+1$ livelli e con massimo livello $k = \lfloor \log n \rfloor$, mentre può variare l'allocazione degli elementi nei nodi, purché rimanga valida la (9). I sottoalberi dei nodi non terminali sono ancora degli heaps.

L'algoritmo di ordinamento *HEAPSORT* agisce sull'allocazione sequenziale di un albero binario con la forma di uno heap ma nei cui nodi, in numero eguale alla cardinalità di $X$, gli elementi da ordinare sono inizialmente disposti in ordine casuale.

Il valore di ciascun nodo non terminale, dall'ultimo $[\lfloor n/2 \rfloor]$ fino alla radice [1], viene confrontato con il maggiore tra gli elementi contenuti nei figli; se ne risulta minore viene prima scambiato con esso e successivamente è confrontato ed eventualmente scambiato con i valori degli ulteriori discendenti, se esistenti, in modo da realizzare uno heap. Una volta costruito lo heap, a partire dalla casuale permutazione iniziale, se ne estrae il massimo allocato nella radice e lo si porta nell'ultima foglia, scambiando le chiavi di $x[1]$ e di $x[n]$. Si ricostruisce quindi lo heap, danneggiato dall'estrazione del massimo, con gli $n-1$ elementi di $X[1 : n-1]$. La procedura, iterata $n-1$ volte sullo heap di volta in volta ridotto di un nodo, genera la permutazione ordinata $x[1] < x[2] < \cdots < x[n-1] < x[n]$.

Nell'esempio della **Fig.12** devono essere ordinati i primi nove interi, inizialmente allocati in ordine casuale nell'albero con la forma di heap a nove nodi (**a**). Poiché $\lfloor 9/2 \rfloor = 4$, l'ultimo nodo non terminale è il [4] che contiene il valore 2, nei nodi figli [8] e [9] sono allocati rispettivamente i valori 7 e 6, il 2 è confrontato e scambiato col 7. Il successivo nodo da testare è il [3] ma il confronto fra $x[3] = 8$ ed $x[7] = 4$ trova i valori in ordine corretto e non dà luogo a scambi. (**a-b**). La chiave 5 di $x[2]$ è confrontata e scambiata con la chiave 9 di $x[5]$ (**b-c**). Si testa infine la radice [1] ed il valore 3 in essa allocato è confrontato e scambiato col 9 del nodo [2]; il successivo confronto è effettuato fra $x[2] = 3$ e la chiave maggiore dei nuovi figli, cioè $x[4] = 7$. Scambiati il 3 ed il 7 il valore 3, che adesso è allocato nel nodo [4], è confrontato e scambiato col 6 del nodo [9] (**c-d**).

A questo punto lo heap è stato costituito ed il massimo di $X[1 : 9]$ è allocato nella radice; per estrarlo e portarlo nell'ultima foglia, in fondo alla sequenza, si scambiano $x[1] = 9$ con $x[9] = 3$. Si riapplica la procedura di ricostruzione dello heap agli elementi di $X[1 : 8]$ (**d-e**). I confronti fra ciascuno dei nodi [4], [3], [2] coi rispettivi figli non comportano scambi, trovando gli elementi correttamente disposti; $x[1] = 3$ è confrontato e scambiato prima con $x[3] = 8$, quindi con $x[7] = 4$ (**e-f**) . Si scambiano $x[1] = 8$ con $x[8] = 2$ e si ricostruisce lo heap con gli elementi di $X[1 : 7]$. Così continuando si perviene alla permutazione ordinata.

Si osservi che la semplicità con cui uno heap può essere ricostituito è essenzialmente dovuta alla sua particolare forma che consente, per ogni nodo $[i]$, di rintracciare immediatamente nell'allocazione sequenziale l'elemento associato al padre $x[\lfloor i/2 \rfloor]$, per $1 < i \leq n$, e gli elementi contenuti nei figli sinistro e destro, $x[2i]$ ed $x[2i+1]$ rispettivamente, se esistenti, cioè se $2i+1 \leq n$; il nodo $[i]$ è infatti terminale se $n < 2i$, ha solo il figlio sinistro se $n = 2i$.

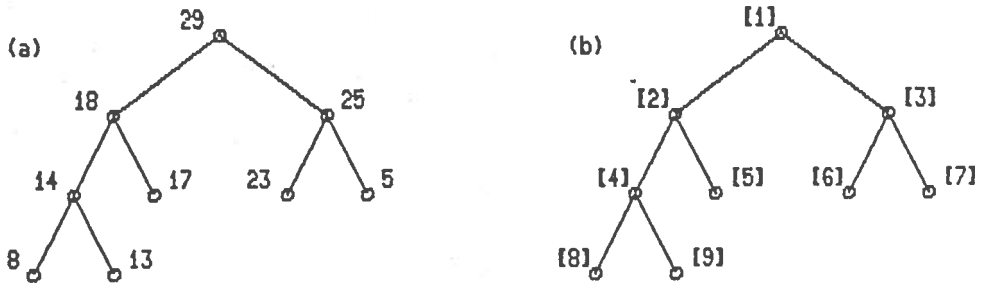


**Fig.11** *Uno heap con 9 nodi e la sua allocazione sequenziale.*

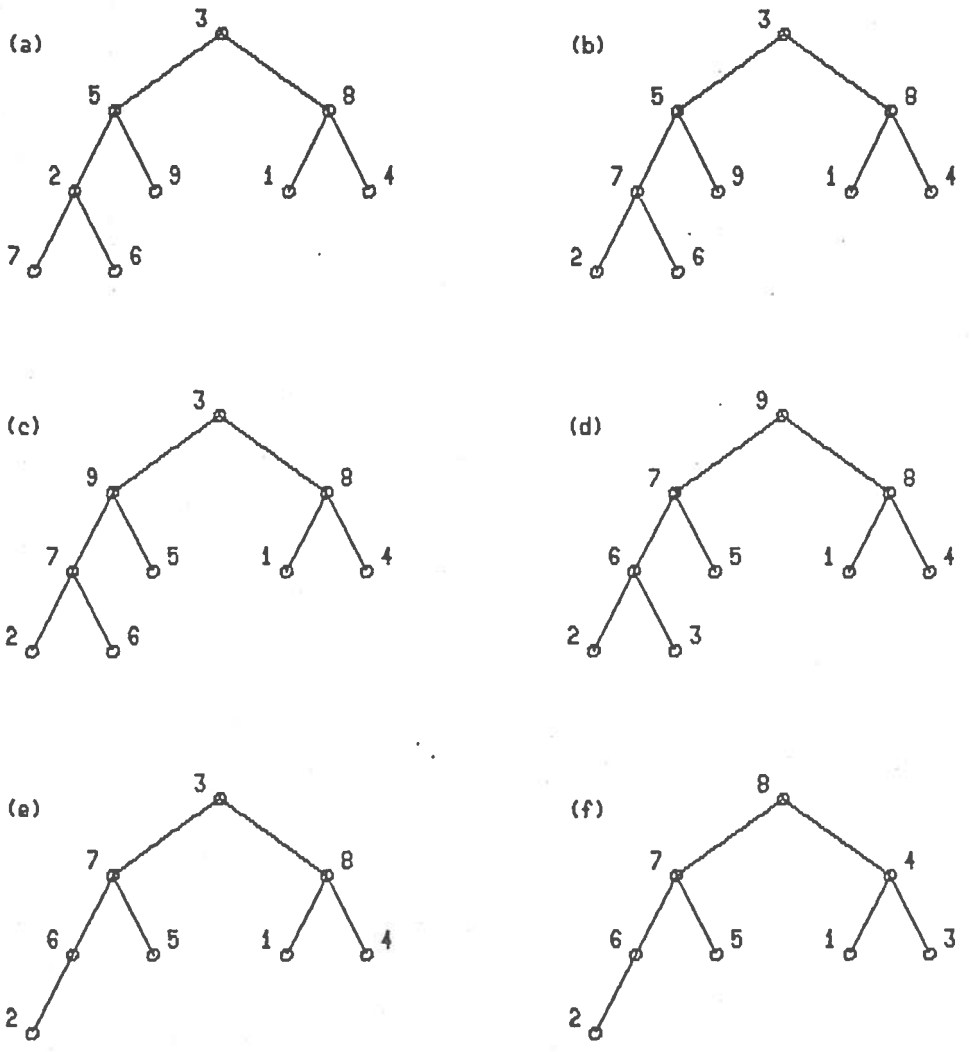


**Fig.12** *Ordinamento mediante heap:*

a-b) $HEAP(4,9)$ $2 \leftrightarrow 7$; $HEAP(3,9)$ *nessuno scambio;*

b-c) $HEAP(2,9)$ $5 \leftrightarrow 9$;

c-d) $HEAP(1,9)$ $3 \leftrightarrow 9$; $HEAP(2,9)$ $3 \leftrightarrow 7$; $HEAP(4,9)$ $3 \leftrightarrow 6$;

d-e) *Estrazione del massimo dallo heap:* $x[1] = 9 \leftrightarrow x[9] = 3$;

e-f) *Ricostituzione dello heap:* $HEAP(4,8)$ $(3,8)$ $(2,8)$ *nessuno scambio;* $HEAP(1,8)$ $3 \leftrightarrow 8$; $HEAP(2,8)$ $3 \leftrightarrow 4$.

In **Cod.Ps.4** è riportata la codifica *PASCAL* di *HEAPSORT*, che non è di per sè ricorsiva ma lo diviene in pratica perché richiama al suo interno la procedura ricorsiva $HEAP(i,j)$, il cui ruolo è quello di ricostruire lo heap tra i nodi $[i]$ e $[j]$.

```
.............................
(* X ed n sono definiti globalmente *)
PROCEDURE HEAP (i,j:  integer);
VAR m:integer;
(* ricostituisce la sezione di heap tra i nodi i e j *)
BEGIN
  IF i<=j/2 THEN
    BEGIN
      m:=2*i;
      IF (m<j) AND (x[m]<x[m+1]) THEN m:=m+1;
      IF x[i]<x[m] THEN
        BEGIN
          CHANGE(x[i],x[m]); HEAP(m,j);
        END;
    END;
END;
PROCEDURE HEAPSORT;
VAR i:integer;
(* ordina gli elementi di X[1:n] *)
BEGIN
  FOR i:=n DIV 2 DOWNTO 1 DO HEAP(i,n);
  FOR i:=n DOWNTO 2 DO
    BEGIN
      CHANGE (x[1],x[i]); HEAP(1,i-1);
    END;
END;
.............................
```

**Cod.Ps.4** **HEAPSORT**

Esaminiamo brevemente il funzionamento di $HEAP(i,j)$ . Se $i > j/2$ il nodo $[i]$ è terminale e si esce dalla procedura. Se $i \leq j/2$ l'$i$-esimo elemento deve essere confrontato con il maggiore dei suoi figli, la cui posizione è contenuta nella variabile locale $m$ : $x[m] = max\{x[2i], x[2i+1]\}$. Se il figlio destro non esiste, cioè se $j < 2i$ allora $x[m] = x[2i]$. Se $x[i] < x[m]$ si scambiano le relative chiavi ed il valore $x[m]$ dev'essere ancora confrontato col maggiore dei valori contenuti nei figli, se esistenti; viene dunque richiamata $HEAP(m,j)$.

I confronti eseguiti da $HEAP(i,j)$ sono tanti quanti i nodi di decisione che si incontrano partendo da $x[i]$ per arrivare sino ad $x[j]$, cioè $\lfloor \log(j/i) \rfloor$.

In *HEAPSORT* il primo ciclo di FOR costruisce lo heap con gli $n$ elementi di input ed ha complessità $O(n)$. Infatti, richiamando $HEAP(i,n)$ per $i = 1, .., \lfloor n/2 \rfloor$, esegue un numero massimo di confronti $C_1(n) = \sum_{i=1}^{\lfloor n/2 \rfloor} \lfloor \log n/i \rfloor$. Supponendo per semplificare i calcoli che lo heap sia completo, cioè $n = 2^{k+1} - 1$ e tenendo conto che per $2^{j-1} \leq i \leq 2^j - 1$ risulta $\lfloor \log(2^{k+1} - 1)/i \rfloor = k + 1 - j$, si deriva facilmente che $C_1(n) = n - \log(n+1) < n$ essendo

$$C_1(n) = \sum_{i=1}^{2^k-1} \left\lfloor \log \frac{(2^{k+1}-1)}{i} \right\rfloor = \sum_{j=1}^{k} \sum_{i=2^{j-1}}^{2^j-1} (k+1-j) = \sum_{j=1}^{k} 2^{j-1}(k+1-j) = 2^{k+1} - k - 2$$

(l'ultima eguaglianza si può facilmente provare per induzione).

Il risultato $C_1(n) < n$ può giustificarsi anche intuitivamente osservando che il generico livello $\ell$, con $0 \leq \ell \leq k$ ha $2^\ell$ nodi, ciascuno dei quali richiede al più $k - \ell$ confronti. (Le foglie del massimo livello $k$ non subiscono alcun confronto, i nodi del livello $k-1$ sono confrontati una volta se non terminali, i $2^{k-2}$ del livello $k-2$ subiscono due confronti e così via fino alla radice che è confrontata $k$ volte.) Dunque

$$C_1(n) = \sum_{\ell=0}^{k-1} 2^\ell (k-\ell) = \sum_{\ell'=1}^{k} \ell' 2^{k-\ell'} = \sum_{\ell'=1}^{k} \ell' \frac{2^k}{2^{\ell'}} \leq n \sum_{\ell'=1}^{k} \frac{\ell'}{2^{\ell'}} < n = O(n)$$

Il secondo ciclo di FOR chiama $n-1$ volte la procedura $HEAP(1, i-1)$ per $i = 2, .., n$ per ricostruire lo heap dopo l'estrazione del massimo ed ha dunque complessità $O(n \log n)$, eseguendo un numero massimo $C_2(n)$ di confronti dato da:

$$C_2(n) = \sum_{j=2}^{n} \lfloor \log j \rfloor < (n-1) \log n = O(n \log n)$$

Anche tenendo conto che ad ogni confronto fra le chiavi di un nodo e di un suo discendente è associato il confronto che stabilisce quale dei valori contenuti nei figli sia il maggiore la complessità di $HEAPSORT$ rimane comunque $O(n \log n)$ e ciò anche nel caso pessimo.

Per $n$ molto grande $QUICKSORT$ è mediamente più efficiente di $HEAPSORT$, impiegando circa metà tempo. Occorre però tener presente che il caso pessimo di $HEAPSORT$ non è peggiore del caso medio, mentre il caso pessimo di $QUICKSORT$ ha, come si è visto, complessità $O(n^2)$.

In **Cod.Ps.4a** è riportata la versione iterativa della procedura $HEAP$, facilmente ottenibile sostituendo l' $IF$ iniziale con un ciclo di $WHILE$ e la chiamata $HEAP(1, i-1)$ con l'assegnamento $i := m$ e con un'istruzione di salto, che restituisce il controllo alla procedura chiamante $HEAPSORT$ nel caso in cui la condizione $x[i] < x[m]$ non sia verificata.

```
PROCEDURE HEAP (i,j:  integer);
(* versione iterativa *)
LABEL 10;
VAR m:integer;
BEGIN
  WHILE i<=j/2 DO
    BEGIN
      m:=2*i;
      IF (m<j) AND (x[m]<x[m+1]) THEN m:=m+1;
      IF x[i]<x[m] THEN
        BEGIN
          CHANGE(x[i],x[m]); i:=m;
        END;
              ELSE GOTO 10;
    END;
  10:
END;
```

**Cod.Ps.4a** **Versione iterativa di HEAPSORT**

## 8. - ORDINAMENTO DI MULTINSIEMI O DI INSIEMI IN PARTE PREORDINATI.

Nella descrizione degli algoritmi presentati abbiamo supposto distinti gli elementi da ordinare. Le relative codifiche funzionano comunque anche con elementi non tutti distinti.

In $PIVOT(j,r)$ (v.**Cod.Ps.2**) l'istruzione condizionata di salto alla fine della procedura, con conseguente restituzione del controllo alla procedura chiamante $QUICKSORT$

$$IF\ (\ell = r)\ AND\ (x[\ell] = x[j])\ THEN\ GOTO\ 10;$$

è stata inserita per evitare il *loop* infinito del $WHILE\ \ell <= r$ nel caso in cui i due cursori $\ell$ ed $r$ si fermino entrambi su uno stesso elemento eguale al perno. Per esempio nella chiamata $PIVOT(1,4)$, applicata alla sequenza $12, 4, 12, 6$, i due cursori si fermano entrambi su $x[3] = 12$ ed ivi resterebbero indefinitamente senza l'istruzione sopra indicata.

Si noti che dei tre algoritmi presentati solo $MERGESORT$ risulta stabile mentre negli altri chiavi identiche possono variare, nel corso dell'ordinamento, il loro iniziale ordine relativo.

La stabilità è comunque solo una delle proprietà auspicabili in un algoritmo di ordinamento di dati non tutti distinti.

L'ordinamento di multinsiemi di $N$ elementi, con $n < N$ valori distinti, è stato particolarmente studiato in questi ultimi anni. Un notevole guadagno di efficienza potrebbe ottenersi se l'algoritmo fosse concepito in modo da riconoscere la particolare struttura dell'input, effettuando l'ordinamento in tempo $O(N \log n)$ invece di $O(N \log N)$, purché naturalmente ciò non comporti un significativo peggioramento nel comportamento dell'algoritmo ove tutti gli elementi siano distinti.

Un algoritmo che ordini $N$ chiavi distinte in tempo $O(N \log N)$, mediamente, ed $N$ chiavi eguali in tempo $O(N)$ è detto da L.M.Wegner (1985) *smooth* (termine già usato da E.W.Dijkstra con riferimento a metodi di ordinamento su liste preordinate).

Un'altra proprietà desiderabile riguarda lo spazio di memoria occupato. Si dice che l'ordinamento avviene *in situ* se occorrono solo $O((\log N)^2)$ celle di memoria, oltre allo spazio occorrente per l'allocazione delle $N$ chiavi. $MERGESORT$ non ha tale proprietà, mentre $HEAPSORT$ e la versione iterativa di $QUICKSORT$, in cui il più piccolo dei sottoinsiemi generati dalla partizione è sempre ordinato per primo, sono *in situ*.

Il nome $QUICKSORT$ dato da C.A.R.Hoare al metodo di ordinamento da lui ideato, basato sulla partizione dei dati attorno ad un elemento pivot (*partition-exchange sorting*), è pienamente giustificato poiché, come già si è detto, esso impiega mediamente un tempo di esecuzione sensibilmente inferiore a quello richiesto da altri algoritmi di complessità ottimale, quali $MERGESORT$ e $HEAPSORT$, ed occupa poco spazio di memoria. Per questo sono state elaborate diverse varianti di $QUICKSORT$, allo scopo o di controllarne il caso pessimo o di migliorarne l'efficienza su insiemi in parte già ordinati o su multinsiemi.

Una variante di $QUICKSORT$ proposta da R.Sedgewick (1975) rende estremamente improbabile il comportamento più sfavorevole ed inoltre migliora sensibilmente il tempo medio di esecuzione. Recentemente L.M.Wegner (1985) ha proposto un nuovo algoritmo derivato da $QUICKSORT$ e denominato $RUNSORT$, non stabile, capace di riconoscere *runs*, cioè sequenze di chiavi in ordine crescente o decrescente e dunque particolarmente efficiente su insiemi preordinati; il suo tempo d'esecuzione tuttavia aumenta sensibilmente nel caso medio.

D.Motzkin (1981) e L.M.Wegner (1982, 1983) hanno indipendentemente ideato ed analizzato due algoritmi derivati da *QUICKSORT* (*stable linked list QUICKSORT*) che risultano stabili, *smooth* ed *in situ*.

Il primo algoritmo, chiamato *TRISORT*, elimina tutte le chiavi eguali all'elemento scelto come pivot durante la procedura di partizione attorno a questo (ed è perciò detto *three-way split algorithm*), richiedendo mediamente $3(N+M)H_{N/M} - 4N$ confronti fra chiavi per ordinare un multinsieme con $N$ elementi ed $M$ occorrenze di ciascuno degli $N/M$ elementi distinti ($H_k$ denota il $k$-esimo numero armonico, $H_k \leq \ln k + 1$ per tutti i $k \geq 1$). È stato anche provato che nel caso generale, in cui ciascuno degli $n$ valori distinti si presenta $x_i$ volte con $1 \leq x_i \leq n$ e $\sum_{i=1}^{n} x_i = N$, la complessità dell'algoritmo approssima il limite inferiore, a meno di una costante moltiplicativa.

Il secondo algoritmo, chiamato *LINKSORT*, esamina da sinistra a destra il sottoinsieme su cui è applicata la procedura di partizione attorno al perno, fino ad incontrare un elemento diverso dal perno. Se non ne trova alcuno il sottoinsieme è considerato unario e non deve essere ulteriormente trattato; altrimenti tutte le chiavi precedentemente esaminate vengono anteposte al sottoinsieme sinistro contenente tutti gli elementi $\leq$ all'elemento pivot e la scansione è continuata normalmente (*two way split*). Si dimostra che il numero medio di confronti richiesti da *LINKSORT* per ordinare $N$ elementi, dei quali ogni elemento distinto si presenti $M$ volte, è limitato superiormente da $2NH_{N/M} + N$.

Altri algoritmi di ordinamento di arrays, derivati da *QUICKSORT*, proposti da L.M.Wegner (1985) riducono significativamente il loro tempo di esecuzione se applicati a multinsiemi ma risultano non stabili e non particolarmente efficienti nel caso medio.

Tra gli algoritmi di ordinamento esistenti *QUICKSORT* appare dunque il più adatto per un impiego generalizzato, anche se algoritmi di complessità maggiore possono mostrarsi più convenienti in circostanze particolari.

## 9. - Codifica FORTRAN di QUICKSORT, MERGESORT, HEAPSORT.

Completiamo la trattazione formalizzando in FORTRAN i tre algoritmi presentati, naturalmente in versione iterativa.

Nei limiti consentiti dalla diversità dei due linguaggi la traduzione è quasi letterale.

I commenti, ignorati dal compilatore, sono preceduti da un ! se seguono istruzioni eseguibili, ovvero si trovano in schede commento con una `C` in colonna 1.

Poichè il *FORTRAN* è prevalentemente usato in ambiente scientifico i dati da ordinare sono supposti di tipo reale (REAL*4). Vengono mantenuti gli identificatori dei sottoprogrammi, delle costanti e delle variabili già usati nella codifica *PASCAL*; tuttavia i nomi delle variabili di tipo intero con iniziale diversa da `I,J,K,L,M,N` (che in *FORTRAN* risultebbero di tipo reale senza una esplicita dichiarazione) sono preceduti da una `I`. Per esempio le variabili `r`, `right`, `st` della procedura *QUICKSORT* divengono `IR`, `IRIGHT` ed `IST` nella subroutine *QUICKSORT*.

Non esistendo in *FORTRAN* la distinzione fra variabili locali e globali, tutte le quantità che in *PASCAL* sono state dichiarate globalmente dovranno essere trasmesse come parametri alle subroutines. In particolare l'array $X$ e la sua cardinalità compaiono sempre tra gli argomenti dei sottoprogrammi; grazie all'uso delle dimensioni aggiustabili eventuali modifiche di dimensione

riguarderanno solo il programma principale.

Nella codifica *PASCAL* abbiamo usato per semplicità i files standard di ingresso-uscita. Qui la subroutine *READWRITE* consente l'input-output interattivo e/o da files specificati dall'utente. La variabile intera INP associata all'*input mode* vale 1 se l'input avviene da terminale, 2 se i dati da ordinare sono memorizzati in un file, la cui specificazione (nome ed eventualmente numero e versione), fornita dall'utente, viene assegnata alla variabile di caratteri *FLI*. Se non è noto il numero $N$ dei dati si digiterà $-1$ e *READWRITE* assumerà preliminarmente $N$ eguale alla massima dimensione dichiarata per l'array $X$, quindi leggerà i dati fino all'*ENDFILE* e determinerà l'effettiva cardinalità di $X$, restituendo infine il controllo al programma principale. Completato l'ordinamento si rientrerà in tale subroutine (nell'*entry-point* ENTRY READWRITE1) per procedere all'output sul video e/o su file. La variabile INP1 è associata alla scelta dell'*output mode*, FLO contiene la specificazione del file di output.

Nel programma *QUICKSORTING* l'istruzione di assegnazione $X(N+1) = 1.E37$ garantisce che in ogni chiamata della subroutine $PIVOT(X, N, J, IR)$ risulti $X(I) < X(IR-1)$, per ogni $I$ con $J \leq I \leq IR$, consentendo così il corretto funzionamento al limite della procedura di partizione dei dati attorno al perno. (Naturalmente se i dati da ordinare sono in doppia precisione si varierà opportunamente tale assegnamento).

Nella subroutine *PIVOT* l'istruzione di salto *GOTO* usata nella codifica *PASCAL* è sostituita da un RETURN, che restituisce il controllo alla procedura chiamante nel caso in cui i due cursori L ed IR si fermino entrambi su un elemento eguale al perno X(J). Analogamente l' *ELSE GOTO* della procedura *HEAP* è stato sostituito da un *ELSE RETURN* nell'omonima subroutine.

Nella procedura iterativa *MERGESORT* della **Cod.Ps.3a** le chiamate di *MERGE* contengono come parametri delle espressioni aritmetiche (chiamata per valore). In *FORTRAN* invece tutti gli argomenti di una subroutine sono anche parametri di ritorno della stessa. La corrispondente subroutine *MERGESORT* appare perciò lievemente modificata; le chiamate di *MERGE* sono precedute dal calcolo dei relativi parametri, effettuato utilizzando la FUNCTION NEXTL. Le stesse considerazioni valgono per la subroutine *HEAPSORT*.

Ringraziamenti.

L'interesse per questa ricerca è stato sollecitato dal corso sulla struttura degli algoritmi tenuto alla Scuola Estiva d'Informatica 1983, presso l'Università di Lecce, dal Prof. Fabrizio Luccio, al quale mi è gradito rivolgere un sentito ringraziamento. Ringrazio anche il Prof. Attilio Agodi per utili discussioni durante la stesura del lavoro.

```
      PROGRAM QUICKSORTING          !Sistema in ordine non decrescente gli
      DIMENSION X(1501)             !elementi di X utilizzando QUICKSORT.
      CALL READWRITE(X,N)           !Lettura dei dati.
      KMAX=2*INT(ALOG(FLOAT(N))/ALOG(2.))
      X(N+1)=1.E37  !Garantisce il corretto funzionamento al limite della subroutine PIVOT.
      CALL QUICKSORT(X,N,IST,KMAX)  !Ordinamento dei dati.
      NP1=N+1
      CALL READWRITE1(X,N,NP1)      !Scrittura dei dati.
      STOP
      END
C
      SUBROUTINE READWRITE(X,N)     !Subroutine di lettura e scrittura dei dati.
      CHARACTER FLI*15, FLO*15
      DIMENSION X(1)
      TYPE *,' CHOOSE INPUT MODE: TYPE 1(FROM KEYBOARD); 2(FROM FILE)'
      ACCEPT*, INP
      IF (INP.EQ.1) THEN            !Input da terminale.
         TYPE*, ' ENTER DATA NUMBER N'
         ACCEPT*, N
         TYPE*, ' ENTER DATA '
         ACCEPT*, (X(I),I=1,N)
           ELSE                     !Input da file.
         TYPE*,' ENTER INPUT FILENAME (MAX 9+6 CHARACTERS)'
         ACCEPT 10,FLI
10       FORMAT(A15)
         OPEN(UNIT=1,NAME=FLI,TYPE='OLD')     !Apertura del file di input.
         TYPE*,' ENTER DATA NUMBER N (TYPE -1 IF YOU DO NOT KNOW IT)'
         ACCEPT*,N
         IF (N.LT.0) N=1500
         DO I=1,N
           READ (1,*,END=20) X(I)
         END DO
20       N=I-1
         TYPE*, 'DATA NUMBER =',N
         CLOSE(1)                           !Chiusura del file di input.
      END IF
      RETURN
      ENTRY READWRITE1(X,N,NP1)
C     In MERGESORTING ed in HEAPSORTING si porrà ENTRY READWRITE1(X,N).
      TYPE *,' CHOOSE OUTPUT MODE: TYPE 1 (TO DISPLAY); 2(TO FILE)'
      TYPE *,'                          3 (TO DISPLAY AND FILE)'
      ACCEPT*, INP1
      IF ((INP1.EQ.1).OR.(INP1.EQ.3)) THEN
         TYPE*, (X(I),I=1,N)
      END IF
      IF ((INP1.EQ.2).OR.(INP1.EQ.3)) THEN
         TYPE*,' ENTER OUTPUT FILENAME (MAX 9+6 CHARACTERS)'
         ACCEPT 30,FLO
30       FORMAT(A15)
         OPEN(UNIT=2,NAME=FLO,TYPE='NEW')     !Apertura del file di output.
         WRITE (2,*) (X(I),I=1,N)
         CLOSE(2)                           !Chiusura del file di output.
      END IF
      RETURN
      END
C
```

```
      SUBROUTINE QUICKSORT(X,N,IST,KMAX)    !Ordina gli elementi di X(1:N)
      DIMENSION X(1),IST(KMAX) !KMAX=2⌊log N⌋ : dimensione massima dell'array pila IST.
      K=0                !Azzeramento iniziale dell'indice di pila.
      LEFT=1
      IRIGHT=N
      DO WHILE (K.GE.0)
         DO WHILE (LEFT.LT.IRIGHT)
           IR=IRIGHT
           CALL PIVOT (X,N,LEFT,IR) !Partizione dei dati attorno al perno X(LEFT).
           IF ((IR-LEFT).LT.(IRIGHT-IR)) THEN
             IST(K+1)=IR+1          !Memorizzazione nella pila delle locazioni iniziale
             IST(K+2)=IRIGHT        !e finale del sottoinsieme di cardinalità maggiore;
             IRIGHT=IR-1            !il sottoinsieme più piccolo verrà ordinato per primo.
                ELSE
             IST(K+1)=LEFT
             IST(K+2)=IR-1
             LEFT=IR+1
           END IF
           K=K+2
         END DO
         IF (K.GT.0) THEN           !Se la pila non è vuota se ne estraggono le locazioni
           IRIGHT=IST(K)            !delle sezioni di X che devono ancora essere ordinate.
           LEFT=IST(K-1)
         END IF
         K=K-2
      END DO
      RETURN
      END
C
      SUBROUTINE PIVOT(X,N,J,IR)    !Ripartisce gli elementi di X(J:IR) attorno
      DIMENSION X(1)                !al pivot X(J); all'uscita il pivot è X(IR).
      L=J+1                         !Inizializzazione del cursore sinistro.
      DO WHILE (L.LE.IR)            !Finché i due cursori L ed IR non s'incontrano:
         DO WHILE (X(L).LT.X(J))    !Il cursore sinistro avanza finquando non punta
           L=L+1                    !un elemento maggiore del perno.
         END DO
         DO WHILE (X(J).LT.X(IR))   !Il cursore destro avanza verso sinistra finché non
           IR=IR-1                  !incontra un elemento minore del perno.
         END DO
         IF (L.LT.IR) THEN          !Se i cursori non si sono ancora incontrati gli
           CALL CHANGE (X(L),X(IR)) !elementi che essi puntano vengono scambiati.
           L=L+1
           IR=IR-1
         END IF
         IF ((L.EQ.IR).AND.(X(L).EQ.X(J))) RETURN
      END DO
      CALL CHANGE(X(J),X(IR))       !Il perno si sposta nella locazione (IR).
      RETURN
      END
C
      SUBROUTINE CHANGE(A,B)        !Scambia le chiavi di A e B
      C=A
      A=B
      B=C
      RETURN
      END
```

```
      PROGRAM MERGESORTING          !Sistema in ordine non decrescente gli
      DIMENSION X(500),Y(500)       !elementi di X utilizzando MERGESORT.
      CALL READWRITE(X,N)           !Lettura dei dati.
      CALL MERGESORT(X,Y,N)         !Ordinamento dei dati.
      CALL READWRITE1(X,N)          !Scrittura dei dati.
      STOP
      END
C
      SUBROUTINE MERGESORT(X,Y,N)   !Ordina per fusione, mediante la Subroutine
      DIMENSION X(N),Y(N)           !MERGE, gli N elementi di X(1:N).
C     Per calcolare i parametri effettivi di MERGE utilizza la Funzione NEXTL(L,LH)=L+LH-1.
C     L2N=⌊log N⌋; JPOT2=2^J; JP2=2^(J-1)   J = 1,..,⌊log N⌋
      L2N=INT(ALOG(FLOAT(N))/ALOG(2.))
      JPOT2=1
      DO J=1,L2N                    !Ciclo sul J-esimo livello di iterazione.
         L=1                        !Inizializzazione della variabile cursore.
         JP2=JPOT2
         JPOT2=JPOT2*2
         DO K=1, N/JPOT2-1 !Si generano per fusione ⌊N/2^J⌋ - 1 sezioni di X con 2^J elementi.
           MID=NEXTL(L,JP2)             !MID=L+2^(J-1)-1
           IRIGHT=NEXTL(L,JPOT2)        !IRIGHT=L+2^J - 1
           CALL MERGE(X,Y,N,L,MID,IRIGHT) !Fusione di X(L:MID) ed X(MID+1:IRIGHT)
           L=IRIGHT+1                   !L=1+2^J(⌊N/2^J⌋ - 1)
         END DO
         MID=NEXTL(L,JP2)
         IF (MOD(N/JP2,2).EQ.1) THEN    !Se ⌊N/2^(J-1)⌋ (numero di insiemi generati
           IRIGHT=NEXTL(L,JPOT2)        !al livello precedente) è dispari allora
           CALL MERGE(X,Y,N,L,MID,IRIGHT) !si genera ancora un insieme di 2^J elementi
           CALL MERGE(X,Y,N,L,IRIGHT,N) !e ad esso si uniscono gli elementi residui.
             ELSE                       !Se invece ⌊N/2^(J-1)⌋ è pari allora si fonde
           CALL MERGE(X,Y,N,L,MID,N)    !l'ultima coppia di insiemi del livello J-1.
         END IF
      END DO
      RETURN
      END
C
      FUNCTION NEXTL(L,LH)      ! Calcola i parametri effettivi di MERGE
      NEXTL=L+LH-1
      RETURN
      END
C
      SUBROUTINE MERGE(X,Y,N,LEFT,MID,IRIGHT) !Fonde X(L:MID) ed X(MID+1:IRIGHT)
      DIMENSION X(N),Y(N)                     !in un unico array ordinato X(L:IRIGHT)
      L=LEFT
      J=LEFT
      M=MID+1
      DO WHILE ((L.LE.MID).AND.(M.LE.IRIGHT)) !Finché entrambi gli insiemi non sono
         IF (X(L).LT.X(M)) THEN               !esauriti si confrontano gli elementi
           CALL COPY(X,Y,N,L,L,J,L)           !in testa, copiandone il più piccolo
             ELSE                             !nella prima locazione libera di Y.
           CALL COPY(X,Y,N,M,M,J,M)
         END IF
         J=J+1
      END DO
      IF (L.GT.MID) THEN                !Si copiano in Y gli eventuali elementi rimasti.
         CALL COPY(X,Y,N,M,IRIGHT,J,J)
```

```
            ELSE
         CALL COPY(X,Y,N,L,MID,J,J)
      END IF
      DO K=LEFT,IRIGHT !Si ricopiano in X gli elementi ordinati in Y.
         X(K)=Y(K)
      END DO
      RETURN
      END
C
      SUBROUTINE COPY (X,Y,N,KMIN,KMAX,J,IQ)
      DIMENSION X(N),Y(N)
C     Se J=Q copia le chiavi di X(KMIN:KMAX) nell'array di servizio Y;
C     se KMIN=KMAX=IQ allora Y(J)=X(IQ), Q=Q+1.
      DO K=KMIN,KMAX
         Y(J)=X(K)
         IQ=IQ+1
      END DO
      RETURN
      END
```

---

```
      PROGRAM HEAPSORTING     !Sistema in ordine non decrescente gli
      DIMENSION X(500)        !elementi di X utilizzando HEAPSORT.
      CALL READWRITE(X,N)     !Lettura dei dati.
      CALL HEAPSORT(X,N)      !Ordinamento dei dati.
      CALL READWRITE1(X,N)    !Scrittura dei dati.
      STOP
      END
C
      SUBROUTINE HEAPSORT(X,N)    !Ordinamento mediante heap.
      DIMENSION X(N)
      DO I=N/2,1,-1               !Dall'ultimo nodo non terminale (I=⌊N/2⌋) fino
         L=I                      !alla radice (I=1) si ricostruisce la sezione di
         CALL HEAP(X,N,L,N)       !heap compresa fra il nodo I ed il nodo N.
      END DO
      DO I=N,2,-1                 !Per valori consecutivi e decrescenti di I da N a 2
         CALL CHANGE (X(1),X(I))  !si estrae il massimo dalla sezione di heap compresa
         IM1=I-1                  !fra la radice ed il nodo I e si ricostruisce l'heap,
         L=1                      !eventualmente danneggiato, ridotto di un elemento.
         CALL HEAP(X,N,L,IM1)
      END DO
      RETURN
      END
C
      SUBROUTINE HEAP(X,N,I,J)    !Ricostituisce la sezione di heap tra i nodi I e J.
      DIMENSION X(N)
      DO WHILE (I.LE.J/2) !Il valore del nodo I (se I è non terminale) è confrontato
         M=2*I            !con il maggiore tra gli elementi contenuti nei figli;
         IF ((M.LT.J).AND.(X(M).LT.X(M+1))) M=M+1
         IF (X(I).LT.X(M)) THEN
           CALL CHANGE(X(I),X(M)) !se ne risulta minore viene scambiato con esso
           I=M   !e successivamente è confrontato ed eventualmente scambiato con i
               ELSE RETURN        !valori degli ulteriori discendenti, se esistenti.
         END IF
      END DO
      RETURN
      END
```

BIBLIOGRAFIA.


- Horowitz E. and Sahni S., *Fundamentals of Data Structures*, Computer Science Press, Potomac 1976.
- Horowitz E. and Sahni S., *Fundamentals of Computer Algorithms*, Computer Science Press, Potomac 1978.
- Knuth D.E., *The Art of Computer Programming: Fundamental Algorithms*, Addison-Wesley, Reading, Mass. 1968.
- Knuth D.E., *The Art of Computer Programming: Sorting and Searching*, Addison-Wesley, Reading, Mass. 1968.
- Luccio F., *Strutture, Linguaggi, Sintassi*, Boringhieri, Torino 1978.
- Luccio F., *La Struttura degli Algoritmi*, Boringhieri, Torino 1982.
- Motzkin D. "A stable Quicksort", *Software Pract. Exper.*, vol.11, June 1981, pp.606-611.
- Sedgewick R., "Quicksort", Computer science dept. Stanford technical report STAN-CS-75-492, May 1975.
- Sedgewick R., "Quicksort with equal keys", *SIAM J. Computing*, vol.6, no.2, June 1977, pp.240-267.
- Wegner L.M., "Sorting a linked list with equal keys", *Inform. Processing Lett.*, vol.15, no.5, Dec.1982, pp.205-208.
- Wegner L.M., "The Linksort family - Design and analysis of fast, stable Quicksort derivatives", dissertation, Inst. Angewandte Inform., Univ. Karlsruhe, West Germany, Rep.123, Feb.1983.
- Wegner L.M., "Quicksort for equal keys", *IEEE Transactions on Computer*, vol.C-34, no.4, April 1985, pp.362-367.
- Williams J.W.J., "Algorithm 232: Heapsort", *Communs ACM*, vol.7, 1964, pp.347 sg.
- Wirth N., *Principi di Programmazione Strutturata*, Isedi, Milano 1971.
- Wirth N., *Algorithms+Data Structures=Programs*, Prentice Hall, Englewood Cliffs, N.J. 1976.
- Vax-11 FORTRAN Language Reference Manual, Digital Equipment Corporation. Maynard, Massachusetts 1982.
- Vax-11 FORTRAN User's Guide, Digital Equipment Corporation. Maynard, Massachusetts 1982.
- Vax-11 PASCAL Language Reference Manual, Digital Equipment Corporation. Maynard, Massachusetts 1982.
- Vax-11 PASCAL User's Guide, Digital Equipment Corporation. Maynard, Massachusetts 1982.