

Algoritmi di ordinamento

Marco Liverani*

23 Luglio 2005

1 Introduzione

Il problema dell'ordinamento di un insieme è un problema classico dell'informatica che, oltre ad avere una indiscutibile valenza in ambito applicativo, tanto che spesso si ritrova il problema dell'ordinamento all'interno di problemi ben più complicati, è anche un utilissimo strumento didattico. Infatti il problema in sé è molto semplice e chiunque è in grado di comprenderne i termini essenziali e per la sua risoluzione sono stati proposti numerosissimi algoritmi molto eleganti che, se presentati in successione in modo opportuno, consentono di evidenziare gli aspetti fondamentali della progettazione e della costruzione di un algoritmo efficiente, di bassa complessità computazionale.

Il problema dell'ordinamento (*sort*) può dunque essere posto nei seguenti termini: dato un insieme di elementi qualsiasi $A = \{a_1, a_2, \dots, a_n\}$ su cui sia possibile definire una *relazione di ordine totale* (ossia una relazione riflessiva, antisimmetrica e transitiva definita su ogni coppia di elementi dell'insieme), che indicheremo come di consueto con il simbolo " \leq ", si richiede di produrre una permutazione degli elementi dell'insieme, in modo tale che $a_{i_h} \leq a_{i_k}$ per ogni $h \leq k$, $h, k = 1, 2, \dots, n$. Ad esempio, se consideriamo l'insieme costituito da $n = 4$ numeri naturali $A = \{a_1 = 7, a_2 = 3, a_3 = 15, a_4 = 6\}$, allora la soluzione del problema dell'ordinamento è data dalla permutazione (a_2, a_4, a_1, a_3) . La soluzione del problema è unica a meno di elementi uguali. Ad esempio nel caso dell'insieme $A = \{a_1 = 7, a_2 = 3, a_3 = 15, a_4 = 7\}$, il problema ammette due soluzioni equivalenti date dalle permutazioni (a_2, a_1, a_4, a_3) e (a_2, a_4, a_1, a_3) .

Nelle pagine seguenti esamineremo una serie di algoritmi via via sempre più sofisticati (e dunque di complessità computazionale sempre più bassa) che risolvono il problema in modo efficiente. La maggior parte degli algoritmi presentati operano esclusivamente sulla base del confronto dei valori dell'insieme da ordinare, mentre altri, proposti al termine della rassegna, risolvono il problema in modo ancora più efficiente utilizzando alcune informazioni aggiuntive sull'insieme da ordinare (ad esempio sulla presenza di elementi duplicati, sul valore minimo e il valore massimo all'interno dell'insieme, o altre informazioni che potrebbero consentire di introdurre delle ottimizzazioni nell'algoritmo, in grado di ridurre in modo significativo la complessità dell'algoritmo stesso).

*liverani@mat.uniroma3.it – <http://www.mat.uniroma3.it/users/liverani>

Algoritmo	Caso migliore	Caso medio	Caso peggiore
SELECTIONSORT	n^2	–	n^2
INSERTIONSORT	n	–	n^2
BUBBLESORT	n	–	n^2
QUICKSORT	$n \log_2 n$	$n \log_2 n$	n^2
MERGESORT	$n \log_2 n$	$n \log_2 n$	$n \log_2 n$
HEAPSORT	$n \log_2 n$	$n \log_2 n$	$n \log_2 n$
COUNTINGSORT	n	n	n
BUCKETSORT	–	–	n

Tabella 1: Confronto della complessità computazionale degli algoritmi esaminati

I primi tre algoritmi analizzati (SELECTIONSORT, INSERTIONSORT e BUBBLESORT) sono estremamente elementari e consentono un'implementazione assai semplice; il costo da pagare alla semplicità di questi algoritmi sta ovviamente nell'elevata complessità computazionale, che in questi casi è $O(n^2)$. L'algoritmo QUICKSORT consente di raggiungere una complessità di $O(n \log_2 n)$ nel caso medio, mentre nel caso più sfavorevole ritorna ad una complessità di $O(n^2)$. Gli algoritmi MERGESORT e HEAPSORT consentono di raggiungere una complessità di $O(n \log_2 n)$ anche nel caso peggiore e, dal momento che è possibile dimostrare che il limite inferiore alla complessità computazionale del problema dell'ordinamento mediante confronti (senza dunque poter sfruttare altre informazioni sull'insieme da ordinare) è proprio pari a $n \log_2 n$, possiamo concludere che tali algoritmi sono *ottimi*. Gli algoritmi COUNTINGSORT e BUCKETSORT sono invece basati su altri criteri e strategie, diverse dal confronto fra i valori degli elementi dell'insieme da ordinare, e sfruttano pertanto altre informazioni sui dati in input; grazie a questo consentono di ridurre ulteriormente la complessità nel caso peggiore, arrivando ad una complessità lineare di $O(n)$. Sotto a tale soglia è impossibile scendere, dal momento che per ordinare un insieme di n elementi è necessario esaminarli tutti almeno una volta. Nella Tabella 1 è riportata una sintesi della stima della complessità degli algoritmi analizzati in queste pagine. Nel seguito ci concentreremo soltanto nello studio della complessità nel caso peggiore, dal momento che è il parametro più conservativo per la valutazione dell'efficienza di un algoritmo.

Gli algoritmi sono presentati in pseudo-codice, ma è molto semplice tradurre tale codifica in uno specifico linguaggio di programmazione, come C, Java, Pascal, Perl o altri ancora. Per la rappresentazione delle informazioni la via più semplice è quella di utilizzare degli array, ma in alcuni casi anche una semplice lista concatenata renderà possibile una codifica molto efficace.

2 Selection sort

Il primo algoritmo preso in esame è il SELECTIONSORT, un algoritmo decisamente intuitivo ed estremamente semplice. Nella pratica è utile quando l'insieme da ordinare è composto da pochi elementi e dunque anche un algoritmo non molto

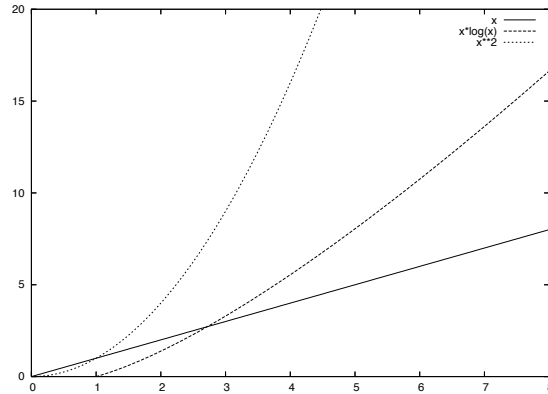


Figura 1: Grafico delle funzioni n , $n \log_2 n$ e n^2

efficiente può essere utilizzato con il vantaggio di non rendere troppo sofisticata la codifica del programma che lo implementa.

L'idea di fondo su cui si basa l'algoritmo è quella di ripetere per n volte una procedura in grado di selezionare alla i -esima iterazione l'elemento più piccolo nell'insieme e di scambiarlo con l'elemento che in quel momento occupa la posizione i . In altre parole alla prima iterazione dell'algoritmo verrà selezionato l'elemento più piccolo dell'intero insieme e sarà scambiato con quello che occupa la prima posizione; alla seconda iterazione sarà selezionato il secondo elemento più piccolo dell'insieme (ossia l'elemento più piccolo dell'insieme "ridotto" costituito dagli elementi $\{a_2, a_3, \dots, a_n\}$) e sarà scambiato con l'elemento che occupa la seconda posizione, e così via fino ad aver collocato nella posizione corretta tutti gli n elementi.

Facciamo un esempio: supponiamo di dover ordinare l'insieme di numeri naturali $A = \{4, 2, 1, 3\}$. Ad ognuno dei passi dell'algoritmo, viene selezionato l'elemento più piccolo nella porzione di insieme ancora da ordinare, scambiandolo con l'elemento che si trova nella prima posizione di tale sottoinsieme. In altri termini al passo i -esimo, viene selezionato l'elemento più piccolo nel sottoinsieme ancora da ordinare costituito dagli elementi a_i, a_{i+1}, \dots, a_n e viene scambiato con l'elemento a_i :

1. $A = (4, 2, \mathbf{1}, 3) \rightarrow A = (1 \mid 2, 4, 3)$
2. $A = (1 \mid \mathbf{2}, 4, 3) \rightarrow A = (1, 2 \mid 4, 3)$
3. $A = (1, 2, \mid 4, \mathbf{3}) \rightarrow A = (1, 2, 3 \mid 4)$
4. $A = (1, 2, 3 \mid \mathbf{4}) \rightarrow A = (1, 2, 3, 4)$

L'Algoritmo 1 riporta una pseudo-codifica di SELECTIONSORT. L'insieme da ordinare è l'insieme A e la variabile n rappresenta il numero di elementi di A : $|A| = n$.

```

SELECTIONSORT( $A, n$ )
1: per  $i = 1, 2, \dots, n$  ripeti
2:   per  $j = i + 1, \dots, n$  ripeti
3:     se  $a_j < a_i$  allora
4:       scambia  $a_i$  e  $a_j$ 
5:     fine-condizione
6:   fine-ciclo
7: fine-ciclo

```

Algoritmo 1: Selection sort

L'algoritmo produce un ordinamento corretto dell'insieme A . Infatti, se per assurdo al termine dell'algoritmo esistessero due elementi a_h e a_k tali che $h < k$ e $a_h > a_k$, allora al passo h del ciclo esterno controllato dalla variabile i , eseguendo il ciclo interno, controllato dalla variabile j , al passo 3 si sarebbe evidenziata questa situazione anomala ($a_{j=k} < a_{i=h}$) e dunque i due elementi sarebbero stati scambiati di posizione; il che contraddice l'ipotesi fatta per assurdo che al termine dell'algoritmo i due elementi si trovassero in ordine reciproco opposto a quello dell'ordinamento.

Dal punto di vista del numero di operazioni elementari svolte dall'algoritmo non esiste un caso particolarmente favorevole o, al contrario, sfavorevole: qualunque sia la disposizione iniziale degli elementi della sequenza da ordinare, se la cardinalità dell'insieme è pari a n allora il numero di operazioni effettuate dall'algoritmo per ordinare la sequenza è $O(n^2)$. Infatti il ciclo esterno (quello controllato dalla variabile i nell'Algoritmo 1) esegue esattamente n iterazioni; ad ogni iterazione del ciclo esterno il ciclo più interno (quello controllato dalla variabile j) esegue $n - i$ iterazioni. Dunque alla prima ripetizione del ciclo esterno vengono compiute $n - 1$ operazioni con il ciclo interno; alla seconda ripetizione del ciclo esterno vengono eseguite $n - 2$ operazioni con il ciclo interno, e così via fino a quando $i = n$. Dunque il numero di operazioni compiute nel complesso dall'algoritmo SELECTIONSORT è pari a

$$(n - 1) + (n - 2) + \dots + (n - n - 1) + (n - n) = \sum_{i=1}^n n - i = \frac{n(n - 1)}{2}$$

Quindi possiamo concludere che la complessità computazionale dell'algoritmo è $O(n^2)$.

È interessante osservare che l'algoritmo SELECTIONSORT opera in modo "cieco", senza sfruttare in alcun modo un eventuale ordinamento parziale degli elementi dell'insieme. Ad esempio, anche nel caso in cui l'insieme da ordinare sia già completamente ordinato, l'algoritmo eseguirà tutte le iterazioni dei due cicli nidificati, con il solo vantaggio di non scambiare mai tra loro gli elementi presi in esame. Tuttavia anche in questo caso apparentemente così favorevole, l'algoritmo esegue un numero di operazioni dell'ordine di $O(n^2)$.

3 Insertion sort

L'idea su cui si fonda l'algoritmo INSERTIONSORT è quella di collocare uno dopo l'altro tutti gli elementi a_i dell'insieme nella posizione corretta del sottoinsieme già ordinato costituito dagli elementi a_1, a_2, \dots, a_{i-1} , inserendolo facendo scorrere a destra gli elementi maggiori.

Ad esempio supponiamo di voler ordinare utilizzando questa strategia l'insieme $A = \{5, 3, 2, 1, 4\}$. Di seguito riportiamo i passi necessari per risolvere il problema:

1. $A = \{5 \mid 2, 3, 1, 4\} \rightarrow A = \{2, 5 \mid 3, 1, 4\}$
2. $A = \{2, 5 \mid 3, 1, 4\} \rightarrow A = \{2, 3, 5 \mid 1, 4\}$
3. $A = \{2, 3, 5 \mid 1, 4\} \rightarrow A = \{1, 2, 3, 5 \mid 4\}$
4. $A = \{1, 2, 3, 5 \mid 4\} \rightarrow A = \{1, 2, 3, 4, 5\}$

Per inserire nella posizione corretta l'elemento a_i , si scorre "all'indietro" il sottoinsieme a_1, a_2, \dots, a_{i-1} , ossia cominciando proprio da a_{i-1} , fino ad arrivare eventualmente ad a_1 . Si itera quindi un ciclo controllato dalla variabile $k = i - 1, i - 2, \dots, 2, 1$ in cui, ad ogni iterazione, se $a_k > a_{k+1}$ si scambiano i due elementi contigui a_k e a_{k+1} . Il ciclo termina quando l'elemento a_i è finito all'inizio del sottoinsieme, oppure quando si individua un elemento $a_k \leq a_{k+1}$. L'Algoritmo 2 riporta la pseudocodifica di INSERTIONSORT; anche in questo caso con A indichiamo l'insieme con n elementi da ordinare (n rappresenta la cardinalità di A).

```
INSERTIONSORT( $A, n$ )
1: per  $i = 2, 3, \dots, n$  ripeti
2:    $k = i - 1$ 
3:   fintanto che  $k \geq 1$  e  $a_k > a_{k+1}$  ripeti
4:     scambia  $a_k$  e  $a_{k+1}$ 
5:      $k = k - 1$ 
6:   fine-ciclo
7: fine-ciclo
```

Algoritmo 2: Insertion sort

Anche in questo caso l'algoritmo produce un ordinamento corretto dell'insieme; infatti, se così non fosse allora dovrebbero esistere due elementi a_t e a_{t+1} tali da non rispettare l'ordinamento: $a_t > a_{t+1}$. Ma questo è impossibile, perché in tal caso i due elementi sarebbero stati scambiati durante l'iterazione del ciclo interno, invertendo la loro posizione reciproca.

Calcoliamo il numero di operazioni eseguite dall'algoritmo a fronte di una istanza del problema di dimensione n . Innanzi tutto osserviamo che, se l'insieme A da ordinare è già ordinato, allora l'algoritmo INSERTIONSORT si dimostra particolarmente efficiente riuscendo a sfruttare questo vantaggio. Infatti, sebbene vengano eseguite tutte le iterazioni del ciclo esterno controllato dalla variabile i , il ciclo interno non viene mai eseguito, perché la seconda delle condizioni che controllano

il ciclo ($a_k > a_{k+1}$) risulta sempre falsa. Dunque in questo caso particolarmente favorevole il numero di operazioni svolte dall'algoritmo è dell'ordine di n .

Consideriamo ora il caso meno vantaggioso, quando gli elementi dell'insieme da ordinare sono inizialmente disposti secondo l'ordinamento opposto a quello desiderato (ad esempio $A = \{5, 4, 3, 2, 1\}$). Questa volta il ciclo interno dell'Algoritmo 2 eseguirà ogni volta il massimo numero di iterazioni, perché ogni elemento a_i si trova alla massima distanza dalla posizione corretta in cui deve essere inserito. Dunque per collocare l'elemento a_i il ciclo interno dovrà eseguire $i - 1$ iterazioni. Globalmente quindi il numero di operazioni eseguite dall'algoritmo nel caso peggiore può essere calcolato come segue:

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

La complessità dell'algoritmo nel caso peggiore è quindi $O(n^2)$, ossia la stessa dell'algoritmo SELECTIONSORT, anche se rispetto a quest'ultimo INSERTIONSORT nella pratica effettua un numero inferiore di operazioni dal momento che riesce a sfruttare a proprio vantaggio eventuali sottoinsiemi già ordinati.

4 Bubble sort

L'algoritmo BUBBLESORT (ordinamento "a bolle") si basa sull'idea di far "emergere" man mano gli elementi più piccoli verso l'inizio dell'insieme da ordinare, facendo "sprofondare" al tempo stesso gli elementi più grandi verso il fondo dell'insieme, un po' come avviene con le bollicine gassose in un bicchiere di acqua minerale.

La strategia adottata è quella di scorrere più volte la sequenza da ordinare, verificando ad ogni passo l'ordinamento reciproco degli elementi contigui, a_i e a_{i+1} , ed eventualmente scambiando la posizione di coppie di elementi non ordinate. Procedendo dall'inizio alla fine della sequenza, al termine di ogni scansione si è ottenuto che l'elemento massimo è finito in fondo alla sequenza stessa, mentre gli elementi più piccoli hanno cominciato a spostarsi verso l'inizio della sequenza stessa. Dunque alla fine della prima scansione possiamo essere certi che l'elemento massimo ha raggiunto la sua posizione corretta nell'ultima posizione della sequenza; pertanto, la scansione successiva potrà fermarsi senza considerare l'ultimo elemento dell'insieme e riuscendo così a collocare nella posizione corretta (la penultima) il secondo elemento più grande dell'insieme; e così via fino ad aver completato l'ordinamento dell'intera sequenza. Riportiamo di seguito un esempio in cui è riportata l'evoluzione dell'ordinamento della sequenza durante le quattro scansioni necessarie a risolvere il problema con la strategia che abbiamo appena descritto:

1. $A = \{3, 5, 2, 4, 1\} \rightarrow A = \{3, 5, 2, 4, 1\} \rightarrow A = \{3, 2, 5, 4, 1\} \rightarrow A = \{3, 2, 4, 5, 1\} \rightarrow A = \{3, 2, 4, 1 \mid 5\}$
2. $A = \{3, 2, 4, 1 \mid 5\} \rightarrow A = \{2, 3, 4, 1 \mid 5\} \rightarrow A = \{2, 3, 4, 1 \mid 5\} \rightarrow A = \{2, 3, 1 \mid 4, 5\}$
3. $A = \{2, 3, 1, \mid 4, 5\} \rightarrow A = \{2, 3, 1 \mid 4, 5\} \rightarrow A = \{2, 1 \mid 3, 4, 5\}$

$$4. A = \{2, 1 \mid 3, 4, 5\} \rightarrow A = \{1, 2, 3, 4, 5\}$$

Osserviamo che effettivamente alla fine di ognuna delle quattro scansioni della sequenza, l'elemento più grande del sottoinsieme ancora da ordinare, finisce in fondo alla sequenza, nella sua posizione definitivamente corretta, mentre la sotto-sequenza ancora indisordinata si riduce di volta in volta di un elemento. Ad esempio alla fine della prima scansione l'elemento 5 finisce in ultima posizione; alla fine della seconda scansione l'elemento 4 (il massimo elemento nel sottoinsieme ancora da ordinare) finisce in fondo, in penultima posizione, e così via.

Possiamo formalizzare la strategia risolutiva nell'Algoritmo 3 che fornisce una pseudo-codifica di BUBBLESORT; come al solito n rappresenta il numero di elementi della sequenza A da ordinare.

```

BUBBLESORT( $A, n$ )
1:  $flag = 1$ 
2:  $stop = n - 1$ 
3: fintanto che  $flag = 1$  ripeti
4:    $flag = 0$ 
5:   per  $i = 1, 2, \dots, stop$  ripeti
6:     se  $a_i > a_{i+1}$  allora
7:       scambia  $a_i$  e  $a_{i+1}$ 
8:        $flag = 1$ 
9:     fine-condizione
10:  fine-ciclo
11:   $stop = stop - 1$ 
12: fine-ciclo

```

Algoritmo 3: Bubble sort

Anche in questo caso abbiamo un algoritmo strutturato su due cicli nidificati uno dentro l'altro, ma, a differenza dei casi visti nelle pagine precedenti, questa volta il ciclo esterno non viene eseguito un numero prefissato di volte: l'algoritmo BUBBLESORT sfrutta la condizione che controlla il ciclo più esterno per verificare la possibilità di interrompere l'esecuzione dell'intero procedimento non appena dovesse risultare evidente che l'insieme è completamente ordinato. In questo modo, come vedremo tra breve, l'algoritmo è in grado di sfruttare a proprio vantaggio eventuali ordinamenti già presenti nelle sottosequenze dell'insieme da ordinare.

Dal punto di vista tecnico gioca quindi un ruolo cruciale la variabile $flag$ che assume il significato di indicatore dello stato di ordinamento della sequenza: il valore $flag = 1$ indica che l'insieme probabilmente ancora non è ordinato, mentre il valore $flag = 0$ segnala che la sequenza risulta ordinata. Pertanto all'inizio della procedura, dal momento che non è ancora chiaro se l'insieme è ordinato o meno, la variabile $flag$ viene impostata con il valore 1. Il ciclo esterno dell'algoritmo viene controllato unicamente dal valore di questa variabile e viene ripetuto fino a quando la sequenza non risulterà completamente ordinata.

Per stabilire se l'insieme è ordinato o meno, con il ciclo interno controllato dalla variabile i , viene verificato l'ordinamento reciproco degli elementi contigui a_i e

a_{i+1} , per $i = 1, \dots, n-1$. Se due elementi vengono trovati in posizione reciproca non corretta, allora i due elementi vengono scambiati di posto e, siccome in questo modo l'ordine complessivo degli elementi della sequenza è stato modificato, allora la variabile *flag* viene posta uguale a 1, ad indicare che probabilmente, avendo fallito un test sul controllo dell'ordine degli elementi, allora è possibile che l'intera sequenza non sia ancora completamente ordinata.

Visto che prima di iniziare il ciclo interno, al passo 4 dell'Algoritmo 3, viene posto $flag = 0$, se al termine di una scansione completa della sequenza (ossia, al termine del ciclo più interno) la variabile *flag* è ancora uguale a zero, questo indica che tutti i test sull'ordinamento reciproco tra gli elementi contigui (passo 6) ha dato esito negativo, il che significa che $a_i < a_{i+1}$ per ogni $i = 1, 2, \dots, n-1$ e dunque la sequenza è ordinata.

Una condizione così forte per il controllo del ciclo esterno ci garantisce anche la correttezza dell'algoritmo: BUBBLESORT termina solo se la sequenza è ordinata. D'altra parte che l'algoritmo termini è certo, dal momento che il ciclo interno viene eseguito ogni volta con un numero di iterazioni diminuito di uno rispetto alla volta precedente. Infatti, dal momento che ad ogni scansione della sequenza, viene collocato nella posizione corretta l'elemento massimo del sottoinsieme ancora da ordinare, il valore finale della variabile *i* che controlla il ciclo interno, può essere ridotto ogni volta di un'unità; per questo motivo viene utilizzata la variabile *stop* che inizialmente è posta uguale a $n-1$ e, ad ogni iterazione del ciclo esterno, viene decrementata di 1 (passo 11).

Nel caso più favorevole, ossia quando la sequenza *A* è già ordinata, l'algoritmo esegue un'unica scansione dell'intera sequenza (passi 5-9) e quindi termina, dal momento che la variabile *flag* è rimasta impostata a zero. Dunque in questo caso il numero di operazioni eseguite è dell'ordine di n .

Il caso meno favorevole, anche in questo caso, è costituito da una sequenza inizialmente ordinata al contrario. In tal caso BUBBLESORT ad ogni iterazione del ciclo esterno riesce a collocare nella posizione corretta soltanto un elemento dell'insieme; dunque sono necessarie $n-1$ iterazioni del ciclo esterno per ordinare l'intera sequenza. Per ogni iterazione del ciclo esterno il ciclo nidificato al suo interno esegue ogni volta una ripetizione in meno. Quindi il numero di operazioni svolte nel caso peggiore può essere calcolato ancora una volta con la seguente espressione:

$$(n-1) + (n-2) + \dots + 2 + 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

In conclusione anche l'algoritmo BUBBLESORT, nel caso peggiore, ha una complessità computazionale di $O(n^2)$.

5 Quick sort

L'algoritmo QUICKSORT è forse il più famoso algoritmo di ordinamento; la libreria standard del linguaggio di programmazione C fornisce un'implementazione efficiente di tale algoritmo per l'ordinamento di array denominata `qsort()`. È basato su una strategia di tipo *divide et impera*, ossia sull'idea di poter suddividere il

problema in sottoproblemi di uguale natura, ma via via sempre più semplici da risolvere; ovviamente tale strategia è vantaggiosa solo se lo sforzo necessario successivamente per ricomporre le soluzioni dei sottoproblemi ed ottenere la soluzione del problema iniziale, è inferiore all'impegno necessario per risolvere il problema nel suo complesso con un algoritmo diretto. Questo tipo di strategia si presta molto bene ad essere implementata mediante un algoritmo ricorsivo, che viene applicato ad istanze sempre più piccole, fino ad arrivare ad istanze elementari del problema originale, che possano essere risolte in via diretta con una sola operazione, senza dover innescare altre chiamate ricorsive.

Dunque è necessario ridefinire la strategia risolutiva in modo induttivo; l'approccio adottato da QUICKSORT può essere schematizzato come segue:

- 1: **se** $|A| = 1$ **allora**
- 2: allora A è ordinato
- 3: **altrimenti**
- 4: scegli un elemento *pivot* in A e ridistribuisce gli elementi di A in due sottoinsiemi A_1 e A_2 , disponendo in A_1 gli elementi minori del *pivot* e in A_2 gli elementi maggiori o uguali al *pivot*
- 5: applica nuovamente l'algoritmo sui due sottoinsiemi A_1 e A_2
- 6: **fine-condizione**

In sostanza il passo 2 costituisce la base del procedimento induttivo, ossia la base della ricorsione: si assume, ovviamente, che un insieme costituito da un solo elemento ($|A| = 1$) sia di per sé ordinato. Il passo 4 è la caratteristica specifica del QUICKSORT che illustreremo in dettaglio tra breve, mentre il passo 5 innescava due chiamate ricorsive, spezzando il problema originario (ordinare l'insieme A) in due istanze dello stesso problema più piccole (ordinare il sottoinsieme A_1 e il sottoinsieme A_2).

Nello specifico la suddivisione del problema in due sottoproblemi analoghi operata da QUICKSORT è basata sull'idea di separare ogni volta gli elementi "piccoli" da quelli "grandi", scegliendo in modo arbitrario ogni volta una soglia per definire un metro per distinguere tra elementi grandi ed elementi piccoli. L'elemento che svolge il ruolo di soglia è chiamato *pivot*, perché è proprio una sorta di "perno" attorno al quale si spostano gli elementi della sequenza da riordinare.

L'Algoritmo 4 riporta una pseudo-codifica di QUICKSORT. In questo caso, visto che la stessa procedura viene applicata ricorsivamente a sottosequenze di A sempre più piccole, invece di indicare genericamente con n la cardinalità di A , utilizzeremo la variabile p per indicare l'indice del primo elemento della sottosequenza di A che stiamo considerando e r per indicare l'indice dell'ultimo elemento della stessa sottosequenza. Ovviamente, quando all'inizio viene presa in considerazione l'intero insieme A , si avrà $p = 1$ e $r = n$. La funzione ricorsiva è QUICKSORT, che richiama anche la funzione PARTITION che si occupa di riposizionare gli elementi del sottoinsieme che gli viene passato come argomento utilizzando come *pivot* l'elemento del selezionato dalla procedura FINDPIVOT nello stesso sottoinsieme.

La procedura FINDPIVOT costituisce un accorgimento tecnico per assicurare che l'elemento selezionato consenta di suddividere l'array in due componenti non vuote, contenenti entrambe almeno un elemento: per far questo viene scelto come *pivot*

```

QUICKSORT( $A, p, r$ )
1:  $pivot = \text{FINDPIVOT}(A, p, r)$ 
2: se  $pivot \neq \text{null}$  allora
3:    $q = \text{PARTITION}(A, p, r, pivot)$ 
4:   QUICKSORT( $A, p, q$ )
5:   QUICKSORT( $A, q + 1, r$ )
6: fine-condizione

PARTITION( $A, p, r, pivot$ )
1:  $i = p, j = r$ 
2: ripeti
3:    fintanto che  $a_j \geq pivot$  ripeti
4:      $j = j - 1$ 
5:    fine-ciclo
6:    fintanto che  $a_i < pivot$  ripeti
7:      $i = i + 1$ 
8:    fine-ciclo
9:    se  $i < j$   allora
10:    scambia  $a_i$  e  $a_j$ 
11:    fine-condizione
12:  fino a quando  $i < j$ 
13: restituisci  $j$ 

FINDPIVOT( $A, p, r$ )
1:  per  $k = p + 1, \dots, r$  ripeti
2:    se  $a_k > a_p$   allora
3:     restituisci  $a_k$ 
4:    altrimenti se  $a_k < a_p$   allora
5:     restituisci  $a_p$ 
6:    fine-condizione
7:  fine-ciclo
8: restituisci  $\text{null}$ 

```

Algoritmo 4: Quick sort

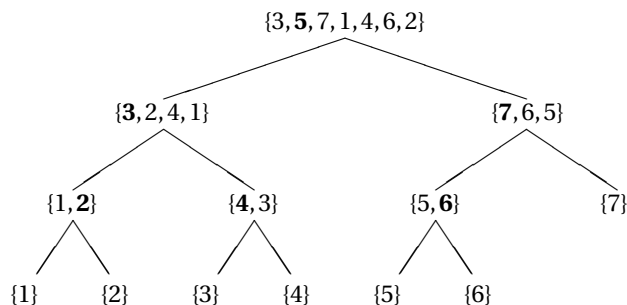


Figura 2: Quick sort dell'insieme A con 7 elementi

il maggiore tra i primi due elementi differenti presenti nell'insieme; se gli elementi sono tutti uguali, allora `FINDPIVOT` restituisce il valore *null* per indicare che l'insieme è ordinato. La scelta del *pivot* avviene in modo piuttosto arbitrario, tanto che, se gli elementi dell'insieme da ordinare sono tutti diversi, si può anche scegliere di volta in volta in modo del tutto casuale il *pivot* nel sottoinsieme preso in considerazione (ad esempio si potrebbe scegliere sempre il primo elemento del sottoinsieme).

La suddivisione ricorsiva della sequenza in sottosequenze può essere rappresentata efficacemente con un albero binario. Consideriamo ad esempio il seguente insieme $A = \{3, 5, 7, 1, 4, 6, 2\}$. Le suddivisioni successive a cui è sottoposto l'insieme A sono rappresentate in Figura 2, dove sono riportati in grassetto gli elementi scelti di volta in volta come *pivot*.

Alla prima chiamata della procedura `QUICKSORT` la sequenza iniziale composta da n elementi viene suddivisa in due sottosequenze sulla base del valore dell'elemento scelto come *pivot* (5). Si ottengono in questo modo due sottoinsiemi di 4 e 3 elementi: $A^1 = \{3, 2, 4, 1\}$ e $A^2 = \{7, 6, 5\}$. Entrambi hanno più di un elemento e quindi viene invocata nuovamente la procedura `PARTITION` sui due sottoinsiemi. Nel primo caso viene scelto come *pivot* l'elemento 3, nel secondo l'elemento 7. Si ottengono così 4 sottoinsiemi: $A^{11} = \{1, 2\}$, $A^{12} = \{4, 3\}$, $A^{21} = \{5, 6\}$ e $A^{22} = \{7\}$. Su ogni sottosequenza, tranne che su A^{22} che è composta da un solo elemento, viene ancora applicata la procedura `PARTITION` ottenendo quindi sette sottoinsiemi: $A^{111} = \{1\}$, $A^{112} = \{2\}$, $A^{121} = \{3\}$, $A^{122} = \{4\}$, $A^{211} = \{5\}$, $A^{212} = \{6\}$ e $A^{221} = A^{22} = \{7\}$.

In pratica la soluzione finale è già stata raggiunta: la chiusura ricorsiva delle procedure `QUICKSORT` che sono state richiamate durante il procedimento di suddivisione dei sottoinsiemi e redistribuzione degli elementi, completa l'algoritmo restituendo alla fine la sequenza A perfettamente ordinata.

Quante operazioni sono state necessarie per ordinare l'insieme di $n = 7$ elementi? La procedura `FINDPIVOT` è lineare nel numero degli elementi della sequenza, anche se nella pratica spesso esegue solo una o due operazioni di confronto. La procedura `PARTITION` applicata su una sequenza di m elementi impiega esattamente $m = r - p + 1$ operazioni (è lineare nella dimensione dell'input). Ogni coppia di spigoli nell'albero binario riportato in Figura 2 rappresenta una chiamata della proce-

dura PARTITION. Su ogni “livello” dell’albero la procedura opera complessivamente su tutti gli elementi dell’insieme (a prescindere dal numero di volte che viene richiamata), impiegando quindi un tempo $O(n)$ per ogni livello dell’albero binario. Moltiplicando $O(n)$ per la profondità dell’albero otteniamo il numero di operazioni eseguite globalmente dall’algoritmo per ordinare la sequenza di n elementi.

Purtroppo l’albero binario che viene costruito con il procedimento di suddivisione e redistribuzione degli elementi dell’array adottato dall’algoritmo QUICKSORT non è sempre un albero completo e bilanciato come quello riportato in Figura 2. Se fosse completo e bilanciato allora la sua profondità sarebbe $\log_2 n$. Nel caso migliore, dunque, quando la scelta dei *pivot* ricade fortunatamente su elementi che consentono di ripartire i sottoinsiemi in due sequenze con lo stesso numero di elementi, l’algoritmo QUICKSORT impiega un tempo di esecuzione dell’ordine di $n \log_2 n$.

Tuttavia non sempre è così. Ad esempio consideriamo il seguente insieme di $n = 4$ elementi: $A = \{1, 4, 2, 3\}$. Con la procedura FINDPIVOT viene scelto come *pivot* l’elemento 4, ottenendo la suddivisione in due sottosequenze “sbilanciate”: $A^1 = \{1, 3, 2\}$ e $A^2 = \{4\}$. Applicando nuovamente la procedura FINDPIVOT e poi la procedura PARTITION al primo sottoinsieme, si ottiene una ulteriore suddivisione in due sequenze sbilanciate: $A^{1^1} = \{1, 2\}$ e $A^{1^2} = \{3\}$. Infine, applicando ancora una volta la funzione di partizione al primo di questi ultimi sottoinsiemi si ottengono le due sequenze $A^{1^{1^1}} = \{1\}$ e $A^{1^{1^2}} = \{2\}$. La profondità dell’albero binario che rappresenta questa partizione (vedi Figura 3) non è quindi pari a $\log_2 n$, ma è lineare in n , visto che ogni volta che abbiamo effettuato una suddivisione dell’insieme in due parti, siamo riusciti a separare un solo elemento dal resto. Nel caso peggiore quindi l’algoritmo QUICKSORT ha una complessità computazionale di $O(n^2)$

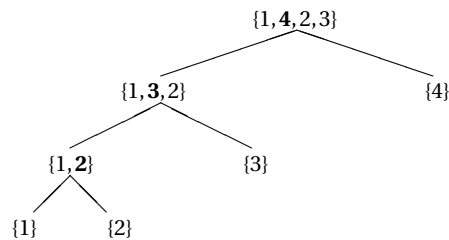


Figura 3: Quick sort dell’insieme A con 4 elementi (albero “sbilanciato”)

6 Merge sort

L’algoritmo di “ordinamento per fusione” adotta una strategia analoga a quella di QUICKSORT, fondata sul principio del *divide et impera*, ma per certi versi la semplifica e al tempo stesso definisce un procedimento più efficiente. L’intuizione su cui si basa questo potente algoritmo di ordinamento è la seguente: potendo disporre di due sequenze già ordinate, è molto facile fonderle in una sola sequenza completamente ordinata, basta scorrerle entrambe estraendo di volta in volta l’elemento

```

MERGESORT( $A, p, r$ )
1: se  $p < r$  allora
2:    $q = \frac{p+r}{2}$ 
3:   MERGESORT( $A, p, q$ )
4:   MERGESORT( $A, q+1, r$ )
5:   MERGE( $A, p, q, r$ )
6: fine-condizione

MERGE( $A, p, q, r$ )
1: siano  $i = p$  e  $j = q+1$ 
2: fintanto che  $i \leq q$  e  $j \leq r$  ripeti
3:   se  $a_i < a_j$  allora
4:      $b_k = a_i$  e  $i = i+1$ 
5:   altrimenti
6:      $b_k = a_j$  e  $j = j+1$ 
7:   fine-condizione
8: fine-ciclo
9: se  $i \leq q$  allora
10:  copia  $(a_i, \dots, a_q)$  in  $B$ 
11: altrimenti
12:  copia  $(a_j, \dots, a_r)$  in  $B$ 
13: fine-condizione
14: copia  $B$  in  $A$ 

```

Algoritmo 5: Merge sort

minimo dalle due sequenze originali, per collocarlo in fondo alla sequenza completamente ordinata che si sta costruendo. Il punto di partenza di questo ragionamento è costituito dal fatto che una sequenza composta da un solo elemento è di fatto una sequenza ordinata.

Partendo da questi presupposti, quindi, è possibile realizzare algoritmo che, dopo aver suddiviso l'insieme iniziale A in n sottoinsiemi costituiti da un solo elemento ciascuno, procede a riaggregare i sottoinsiemi fondendoli ordinatamente fino a ricostruire un'unica sequenza ordinata. L'Algoritmo 5 riporta una pseudo-codifica di MERGESORT.

L'algoritmo è suddiviso in due procedure distinte: la procedura ricorsiva MERGESORT e la procedura iterativa MERGE. In sostanza l'idea di fondo di questo algoritmo è quella di suddividere sempre a metà l'array iniziale, in modo tale che i due sotto-array ottenuti abbiano un ugual numero di elementi (a meno di uno, nel caso di insiemi di cardinalità non rappresentabile come una potenza di 2 o comunque dispari). La suddivisione nel QUICKSORT avveniva spostando a destra e a sinistra del *pivot* gli altri elementi dell'array in modo da ottenere una sorta di ordinamento parziale; nel MERGESORT invece durante il processo di suddivisione l'insieme non viene in alcun modo modificato, i suoi elementi non sono spostati né confrontati fra di loro. Semplicemente si opera una suddivisione dicotomica ricorsiva della sequenza iniziale da ordinare, fino ad ottenere n sequenze di dimensione 1.

Di questa suddivisione ricorsiva si occupa la procedura ricorsiva MERGESORT, che riceve in input l'intera sequenza A insieme con i due indici p e r che consentono di identificare l'elemento a_p e l'elemento a_r , rispettivamente il primo e l'ultimo elemento della sottosequenza su cui deve operare la procedura. La procedura individua l'indice q dell'elemento che si trova a metà della sottosequenza delimitata dagli elementi a_p e a_r (passo 2 della procedura MERGESORT) ed innesca due chiamate ricorsive della stessa procedura sulla prima metà della sequenza (dall'elemento di indice p fino all'elemento di indice q) e sulla seconda metà (dall'elemento di indice $q + 1$ fino all'elemento di indice r). La procedura ricorsiva si ripete fino a quando la sottosequenza da suddividere non è composta da un solo elemento: quando $p = r$ (passo 1) la procedura termina senza compiere alcuna operazione.

Nel *backtracking* della ricorsione sta il vero e proprio punto di forza di questo algoritmo. Durante il *backtracking* infatti vengono fusi a due a due i sottoinsiemi ottenuti con il processo di suddivisione ricorsiva. Durante la fusione gli elementi vengono posizionati in modo da ottenere da due sottosequenze ordinate un'unica sequenza ordinata. Infatti le sequenze composte da un solo elemento da cui inizia il procedimento di fusione sono di per sé ordinate.

Della fusione ordinata di due sottosequenze contigue, $A' = \{a_p, a_{p+1}, \dots, a_q\}$ e $A'' = \{a_{q+1}, a_{q+2}, \dots, a_r\}$, si occupa la procedura iterativa MERGE. Tale procedura esegue una scansione "in parallelo" di entrambe le sequenze (ciclo principale, righe 2–8, controllato dalle variabili i e j), scegliendo e copiando nell'insieme di appoggio B di volta in volta l'elemento minore tra a_i e a_j , rispettivamente appartenenti alla prima e alla seconda sequenza da fondere insieme. Quando una delle due sequenze è stata copiata completamente in B , il ciclo principale termina e la procedura prosegue copiando in B gli elementi rimanenti dell'altra sottosequenza. Infine gli elementi ordinati dell'insieme B vengono copiati nuovamente in A .

Ad esempio supponiamo di dover fondere tra loro le due sequenze ordinate $A' = \{3, 5, 15\}$ e $A'' = \{2, 4, 6\}$. I passi effettuati dalla procedura MERGE possono essere schematizzati come segue:

1. $A' = \{3, 7, 15\}$ $A'' = \{2, 4, 6\} \rightarrow B = \{2\}$
2. $A' = \{3, 7, 15\}$ $A'' = \{4, 6\} \rightarrow B = \{2, 3\}$
3. $A' = \{7, 15\}$ $A'' = \{4, 6\} \rightarrow B = \{2, 3, 4\}$
4. $A' = \{7, 15\}$ $A'' = \{6\} \rightarrow B = \{2, 3, 4, 6\}$
5. $A' = \{7, 15\}$ $A'' = \emptyset \rightarrow B = \{2, 3, 4, 6, 7, 15\}$

Complessivamente potremmo schematizzare l'intero procedimento di ordinamento di una sequenza di n elementi come nel diagramma esemplificativo rappresentato in Figura 4, in cui viene tracciato il procedimento di scomposizione ricorsiva e di ricomposizione nella fase di *backtracking* della sequenza $A = \{3, 15, 6, 2, 7, 4, 12, 8\}$.

Il procedimento di fusione di due sequenze di k elementi ciascuna richiede un tempo di esecuzione dell'ordine di $2k$, ossia lineare nel numero complessivo di elementi. In questo modo, per aggregare h sequenze ordinate costituite da n/h elementi ciascuna, ottenendo $h/2$ sequenze ordinate, si impiega un tempo dell'ordine

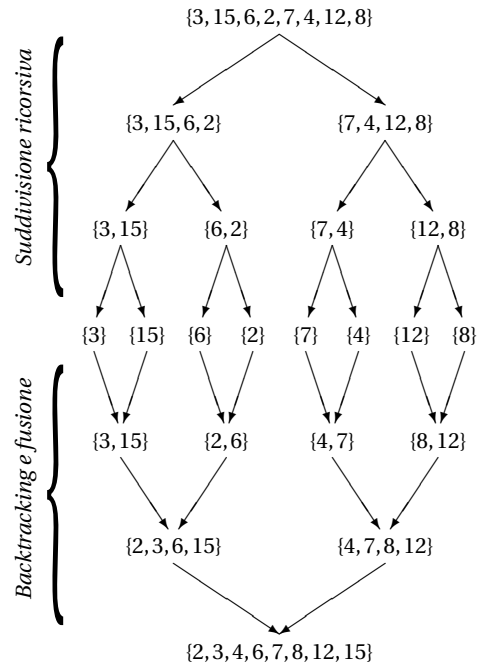


Figura 4: Merge sort dell'insieme A con 8 elementi

di $h(n/h) = n$. Siccome suddividendo sempre a metà l'insieme iniziale si ottiene un albero binario perfettamente bilanciato, la profondità dell'albero con cui possiamo rappresentare le suddivisioni dell'insieme da ordinare (vedi ad esempio la parte superiore della Figura 4) saranno esattamente $\log_2 n$.

Possiamo quindi concludere che la complessità computazionale di MERGESORT è $O(n \log_2 n)$, quindi decisamente più bassa della complessità degli algoritmi visti fino ad ora. Non esistono casi più favorevoli di altri: l'algoritmo adotta lo stesso comportamento a prescindere dalla disposizione iniziale degli elementi nella sequenza da ordinare.

7 Heap sort

Un *heap* è una delle strutture dati con cui è possibile rappresentare e gestire in modo efficiente le *code di priorità*; queste sono delle strutture dati di tipo *FIFO* (*first in first out*) in cui il criterio con cui vengono individuati gli elementi da estrarre dalla coda non si basa esclusivamente sull'ordine di ingresso nella coda stessa, ma su un valore di priorità attribuito ad ogni singolo elemento dell'insieme.

In sostanza gli heap consentono di gestire in modo assai efficiente l'inserimento di un nuovo elemento nella struttura dati e l'estrazione dell'elemento massimo. In

una struttura dati sequenziale, come una lista o un array, l'inserimento di un nuovo elemento richiede un tempo costante se si è memorizzato un puntatore alla prima posizione libera nella sequenza; viceversa la ricerca del massimo richiede un tempo lineare nel numero di elementi presenti nell'insieme. Con la struttura di heap entrambe le operazioni richiedono un tempo logaritmico nel numero di elementi.

Un heap è un albero binario completo; se il numero di elementi non è pari a 2^k (per qualche $k \geq 0$) gli elementi dell'ultimo livello dell'albero dovranno essere raggruppati sulla sinistra, senza lasciare posizioni vuote fra le foglie (vedi Figura 5).

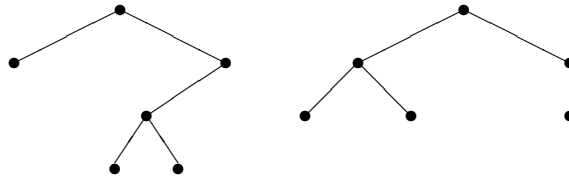


Figura 5: Un albero binario (a sinistra) e un albero binario completo (a destra)

Gli elementi di un heap devono inoltre rispettare una specifica regola di ordinamento reciproco: il vertice padre deve essere sempre maggiore o uguale ai vertici figli. In questo modo nella radice dell'albero è collocato il vertice il cui valore (la priorità) è massimo. L'elemento minimo si trova invece in una delle foglie dell'albero. Non c'è alcuna relazione di ordine specifica tra vertici che non sono tra loro discendenti (ad esempio non è richiesto che sia rispettato uno specifico ordine tra i figli di uno stesso vertice).

Grazie alle sue proprietà (l'essere un albero binario completo) un heap può essere rappresentato utilizzando un array, utilizzando la seguente regola estremamente semplice: il vertice dell'albero che nell'array occupa la posizione i ha come figli (rispettivamente sinistro e destro) gli elementi che occupano le posizioni $2i$ e $2i + 1$, e come padre il vertice che occupa la posizione $\lfloor \frac{i}{2} \rfloor$. La radice dell'albero viene collocata nella posizione di indice 1. L'heap rappresentato in Figura 6 può quindi essere riportato in un array ponendo $A = (a_1 = 60, a_2 = 30, a_3 = 50, a_4 = 20, a_5 = 10, a_6 = 40, a_7 = 45, a_8 = 15, a_9 = 4)$: il padre dell'elemento $a_5 = 10$ è $a_{\lfloor 5/2 \rfloor} = a_2 = 30$ e i figli di $a_3 = 50$ sono rispettivamente $a_{3 \times 2} = a_6 = 40$ e $a_{3 \times 2 + 1} = a_7 = 45$.

Le due operazioni fondamentali da implementare sulla struttura dati che abbiamo così definito sono l'inserimento di un elemento e l'estrazione dell'elemento massimo (l'elemento di massima priorità).

Per l'inserimento di un nuovo elemento in un heap si procede come segue: si aggiunge il nuovo elemento come fogli adell'albero nella prima posizione disponibile, nel rispetto delle caratteristiche dell'heap che non devono essere violate: deve essere un albero binario e completo. Dopo aver collocato come foglia il nuovo elemento si deve garantire che anche la proprietà relativa al valore dei vertici venga rispettata: ogni vertice dell'albero deve essere maggiore o uguale dei propri figli e deve essere minore o uguale al padre. Dunque, per garantire il rispetto di questa regola, si con-

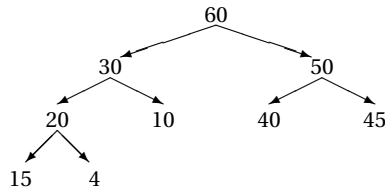


Figura 6: Un esempio di heap

fronterà il nuovo elemento con il padre, eventualmente scambiandoli di posizione e ripetendo questo procedimento fino a quando il nuovo elemento non avrà raggiunto una posizione “stabile” oppure non sarà risalito fino alla radice dell’albero (in tal caso il nuovo elemento è maggiore o uguale ad ogni altro elemento già presente nell’heap). Ovviamente il processo di ricollocazione del nuovo elemento che abbiamo appena descritto non richiede di confrontare l’elemento con ogni altro vertice dell’albero, ma soltanto con quei vertici che si trovano nel cammino unico che porta dalla nuova foglia fino alla radice. Siccome l’heap è un albero binario perfettamente bilanciato, la profondità di un albero con n vertici sarà pari a $\log_2 n$ e tale è il numero di confronti che dovremo compiere, al massimo, per sistemare nella posizione corretta un vertice aggiunto all’heap. L’Algoritmo 6 riporta una pseudo-codifica della procedura di inserimento di un nuovo elemento x nell’heap A rappresentato utilizzando un array.

```

INSERISCI( $A, x$ )
1: sia  $i$  l’indice della prima posizione disponibile in  $A$ 
2:  $a_i = x$ 
3: fintanto che  $i > 1$  e  $a_i > a_{\lfloor i/2 \rfloor}$  ripeti
4:   scambia  $a_i$  e  $a_{\lfloor i/2 \rfloor}$ 
5:    $i = \lfloor i/2 \rfloor$ 
6: fine-ciclo
  
```

Algoritmo 6: Inserimento di un nuovo elemento in un heap

La complessità dell’algoritmo INSERISCI è, come abbiamo detto, $O(\log_2 n)$ nel caso peggiore, ossia quando aggiungendo l’ n -esimo elemento all’heap, tale elemento risulta essere maggiore di ogni altro elemento già presente nell’albero.

L’estrazione dell’elemento massimo dall’heap è immediata, dal momento che tale elemento occupa sicuramente la posizione di radice dell’albero. Tuttavia è necessario ricostruire immediatamente la struttura dell’heap, dal momento che questa risulta compromessa dall’eliminazione della radice. Per ricostruire la struttura si procede come segue: si sposta nella radice l’elemento collocato sull’ultima foglia a destra nell’ultimo livello dell’albero (l’ultimo elemento, nella rappresentazione dell’heap con un array). Tale posizione è probabilmente non corretta e dunque, con un procedimento simile a quello visto in precedenza per l’inserimento di un nuovo ele-

mento, si confronterà la nuova radice con entrambi i suoi figli e si scambierà la posizione reciproca della nuova radice con il suo figlio maggiore. Questo procedimento viene ripetuto fino a quando l'elemento spostato nella radice dell'heap non avrà trovato una sua posizione "stabile" (minore del padre e maggiore o uguale ai due figli), che non comprometta l'ordinamento parziale dei vertici dell'albero. L'Algoritmo 7 propone una pseudo-codifica della procedura per l'estrazione dell'elemento massimo dall'heap A .

ESTRAIMASSIMO(A)

```

1:  $max = a_1$ 
2: sia  $n$  l'indice della posizione dell'ultimo elemento presente in  $A$ 
3:  $a_1 = a_n$ 
4:  $i = 1$ 
5: fintanto che  $i < n$  e  $(a_i > a_{2i}$  o  $a_i > a_{2i+1})$  ripeti
6:   se  $a_{2i} > a_{2i+1}$  allora
7:     scambia  $a_i$  e  $a_{2i}$ 
8:      $i = 2i$ 
9:   altrimenti
10:    scambia  $a_i$  e  $a_{2i+1}$ 
11:     $i = 2i + 1$ 
12:   fine-condizione
13: fine-ciclo
14: restituisci  $max$ 

```

Algoritmo 7: Estrazione dell'elemento massimo e ricostruzione dell'heap

Anche in questo caso la complessità nel caso meno favorevole per l'algoritmo per estrarre il massimo e ridistribuire gli elementi dell'heap, è $O(\log_2 n)$; infatti il ciclo dell'Algoritmo 7 consente di confrontare l'elemento da ricollocare (inizialmente posto nella radice dell'albero – passo 3) con tutti gli elementi che si trovano lungo un cammino che dalla radice porta verso una determinata foglia dell'albero.

L'algoritmo HEAPSORT consente di ordinare in modo efficiente gli elementi di una sequenza sfruttando a proprio vantaggio la struttura dati di heap e le operazioni che abbiamo appena descritto. L'idea su cui si basa l'algoritmo è molto semplice e può essere riassunta nei seguenti termini: al primo passo si "smonta" un elemento dopo l'altro la sequenza da ordinare, inserendo i suoi elementi in un heap inizialmente vuoto; quindi, al secondo passo dell'algoritmo, si "smonta" l'heap, estraendo di volta in volta l'elemento massimo e collocandolo in fondo alla sequenza che così viene riordinata. L'Algoritmo 8 riporta una pseudo-codifica di questo procedimento.

A questo punto è molto semplice calcolare la complessità computazionale dell'algoritmo: con ognuno dei due cicli si ripete per n volte (n indica la cardinalità dell'insieme A che deve essere ordinato) una operazione che ha complessità $O(\log_2 n)$. Dunque $O(n \log_2 n)$ è la complessità globale dell'algoritmo.

```

HEAPSORT( $A, n$ )
1: sia  $H$  un heap vuoto
2: per  $i = 1, 2, \dots, n$  ripeti
3:   INSERISCI( $H, a_i$ )
4: fine-ciclo
5: per  $i = n, n - 1, \dots, 2, 1$  ripeti
6:    $a_i =$  ESTRAIMASSIMO( $H$ )
7: fine-ciclo

```

Algoritmo 8: Heap sort

8 Counting sort

Questo algoritmo è sorprendentemente semplice e potente: sfruttando l'ipotesi aggiuntiva che tutti gli elementi dell'insieme A da ordinare siano naturali minori di una certa soglia $k > 0$ prefissata, riesce a risolvere il problema con una complessità lineare in n , se k è un valore dello stesso ordine di grandezza di n ($k \in O(n)$). È bene osservare che negli algoritmi visti nelle pagine precedenti non avevamo mai posto un vincolo così forte sul valore degli elementi dell'insieme.

L'idea di fondo su cui si basa la strategia risolutiva è la seguente: se nell'insieme esistono p elementi minori o uguali a un certo elemento x , allora x dovrà essere collocato nella posizione $(p + 1)$ -esima nell'ordinamento che si intende costruire. Dunque l'algoritmo opera semplicemente contando il numero di elementi minori di ogni elemento presente nell'insieme, sapendo che i possibili valori di tali elementi sono tutti contenuti nell'insieme $\{0, 1, 2, \dots, k\}$, con k fissato e noto a priori.

Effettuando una prima scansione dell'insieme A da ordinare, viene costruito un array C i cui elementi c_i indicano inizialmente il numero di elementi uguali ad i ($0 \leq i \leq k$). Successivamente eseguendo una scansione di C si modifica il valore dei suoi elementi, assegnando ad ogni c_i il numero di elementi di A minori o uguali ad i . Infine, con una terza ed ultima scansione dell'array C si costruisce un nuovo array B ordinato. L'Algoritmo 9 presenta una pseudo-codifica di questo procedimento.

Il primo ciclo (righe 1–3) consente di inizializzare a zero i contatori rappresentati dagli elementi dell'array C . Questo ciclo esegue k iterazioni. Il secondo ciclo (righe 4–6) consente di contare il numero di elementi uguali presenti in A , memorizzando in c_i il numero di elementi di A di valore i ; quindi il secondo ciclo esegue n iterazioni (tante quanti sono gli elementi di A). Il terzo ciclo (righe 7–9) modifica il valore degli elementi di C : al termine del ciclo ogni elemento c_i contiene il numero di elementi di A minori o uguali ad i ; vengono compiute k iterazioni. Con il quarto ciclo (righe 10–13) viene costruito l'array B con gli stessi elementi di A , ma collocati nell'ordine crescente desiderato; per far questo vengono effettuate n iterazioni. Infine con il quinto ciclo (righe 14–16) gli n elementi di B vengono copiati in A .

L'algoritmo è composto da una sequenza di cicli privi di nidificazione; dunque il numero di operazioni svolte è dato dal numero di operazioni eseguite in ognuno dei cinque cicli e pertanto si può concludere che la complessità dell'algoritmo è $O(n)$, se k è dello stesso ordine di grandezza (o più piccolo) di n .

```

COUNTINGSORT( $A, n, k$ )
1: per  $i = 0, 1, \dots, k$  ripeti
2:    $c_i = 0$ 
3: fine-ciclo
4: per  $i = 1, 2, \dots, n$  ripeti
5:    $c_{a_i} = c_{a_i} + 1$ 
6: fine-ciclo
7: per  $i = 1, 2, \dots, k$  ripeti
8:    $c_i = c_i + c_{i-1}$ 
9: fine-ciclo
10: per  $i = n, n-1, \dots, 2, 1$  ripeti
11:   $b_{c_{a_i}} = a_i$ 
12:   $c_{a_i} = c_{a_i} - 1$ 
13: fine-ciclo
14: per  $i = 1, 2, \dots, n$  ripeti
15:   $a_i = b_i$ 
16: fine-ciclo

```

Algoritmo 9: Counting sort

9 Bucket sort

Questo algoritmo presuppone che i valori degli n elementi dell'insieme A da ordinare siano distribuiti uniformemente nell'intervallo $[s, t)$. La strategia risolutiva si basa quindi sull'idea di suddividere l'intervallo $[s, t)$ in n sottointervalli $[s_i, t_i)$ ($i = 1, 2, \dots, n$; $t_i = s_{i+1}$ per ogni $i < n$) di uguale dimensione che chiameremo *bucket* e di inserire in ogni bucket gli elementi $a_k \in A$ che rientrano nell'intervallo corrispondente ($s_i \leq a_k < t_i$). L'ipotesi iniziale sulla distribuzione uniforme degli elementi di A nell'intervallo $[s, t)$ ci garantisce che ogni bucket conterrà solo pochi elementi. Dunque l'ordinamento dell'insieme originale si ottiene ordinando con un algoritmo elementare ogni singolo bucket e giustappoendo gli elementi dei bucket nell'ordine naturale. I bucket B_i possono essere rappresentati nella pratica con delle liste o degli array. L'Algoritmo 10 propone una pseudo-codifica di BUCKETSORT.

La correttezza dell'algoritmo può essere provata molto facilmente. Supponiamo che in A esistano due elementi distinti $a_h < a_k$; durante l'esecuzione del primo ciclo dell'algoritmo (righe 2–4) i due elementi possono finire in due bucket distinti (B_i e B_j) oppure nello stesso bucket (B_i). Nel primo caso con il terzo ciclo (righe 9–11) i due elementi risulteranno disposti nell'ordine corretto in A ; nel secondo caso sarà la chiamata dell'algoritmo INSERTIONSORT (riga 7) sul bucket B_i a collocare nell'ordine reciproco corretto i due elementi.

Utilizzando strumenti di calcolo delle probabilità è abbastanza facile provare che la complessità dell'algoritmo è $O(n)$, anche se al passo 7 viene invocato un algoritmo di complessità più elevata. Infatti l'insieme su cui deve operare INSERTIONSORT è ridotto a pochi elementi e, per ogni bucket dotato di più di un elemento, esisteranno altrettanti bucket completamente vuoti. Questo consente di bilanciare

```

BUCKETSORT( $A, n, s, t$ )
1:  $\delta = (t - s) / n$ 
2: per  $i = 1, 2, \dots, n$  ripeti
3:   inserisci  $a_i$  nel bucket  $B_k$ , dove  $k = (a_i - s) \bmod \delta$ 
4: fine-ciclo
5: per  $i = 1, 2, \dots, n$  ripeti
6:   sia  $m$  la cardinalità del bucket  $B_i$ 
7:   INSERTIONSORT( $B_i, m$ )
8: fine-ciclo
9: per  $i = 1, 2, \dots, n$  ripeti
10:  concatena il bucket  $B_i$  all'array ordinato  $A$ 
11: fine-ciclo

```

Algoritmo 10: Bucket sort

il numero di operazioni svolte dall'algoritmo, che si mantiene quindi lineare. Per la dimostrazione completa si rimanda a [2].

Riferimenti bibliografici

- [1] A.V. Aho, J.E. Hopcroft, J.D. Ullman, *Data structures and algorithms*, Addison-Wesley, 1987.
- [2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduzione agli algoritmi e strutture dati*, seconda edizione, McGraw-Hill, 2005.
- [3] C. Demetrescu, I. Finocchi, G.F. Italiano, *Algoritmi e strutture dati*, McGraw-Hill, 2004.
- [4] M. Liverani, *Programmare in C – Guida al linguaggio attraverso esercizi svolti e commentati*, Società Editrice Esculapio, 2001.
- [5] M. Liverani, *Qual è il problema? Metodi, strategie risolutive, algoritmi*, Mimesis, 2005.