Java e gli algoritmi di ordinamento

Nelle precedenti puntate abbiamo introdotto i primi concetti di algoritmica e abbiamo presentato due algoritmi di ordinamento. Continuiamo presentando l'Heap Sort e gli alberi binari.

Terza puntata

di Fabrizio Ciacchi

Lo studio degli algoritmi di ordinamento, come già detto nelle precedenti puntate, consente di esplorare differenti aspetti nella progettazione degli algoritmi. Dato infatti un unico problema, l'ordinamento, si può vedere come sia possibile implementare ogni volta un algoritmo sempre più efficiente.

Nelle puntate precedenti abbiamo presentato due algoritmi (Insertion Sort e Merge Sort) la cui logica di funzionamento differiva profondamente ed abbiamo anche notato che l'algoritmo di più difficile comprensione era quello più efficiente. Tuttavia entrambi gli algoritmi manipolavano i dati solo nell'ordinamento. Di seguito presentiamo uno dei migliori algoritmi per l'ordinamento, il quale utilizza una struttura dati particolare, che consente quindi di ridurre notevolmente la complessità e l'utilizzo di memoria.

Fabrizio Ciacchi fabrizio@ciacchi.it

È impiegato come consulente per Vodafone. Tra le sue maggiori aree di interesse ci sono GNU/Linux, il linguaggio di programmazione Java e lo sviluppo di siti web in PHP. Nel tempo libero scrive articoli su GNU/Linux e legge libri di Asimov.
Il suo sito è http://fabrizio.ciacchi.it

Qual è l'algoritmo migliore?

L'algoritmo in questione si chiama Heap Sort ed utilizza una struttura dati chiamata heap per poter poi ordinare gli elementi. Manipolare i dati prima che questi vengano realmente ordinati apporta numerosi vantaggi in termini di efficienza. Infatti "predisporre" tali dati all'ordinamento consente di costruire un algoritmo che ha pressoché la stessa complessità del Merge Sort, ovvero O(n *log*n), ma un minore dispendio di memoria.

Il Merge Sort, infatti, per quanto efficiente, risulta molto dispendioso in termini di memoria, occupandone una quantità in proporzione ai dati n di partenza. In pratica l'utilizzo di memoria si attesta, per il Merge Sort, su una complessità O(n), mentre risulta essere O(1), e quindi costante, per l'Heap Sort.

Il rovescio della medaglia è che l'Heap Sort, contrariamente al Merge Sort e all'Insertion Sort, non è un algoritmo stabile, ovvero non mantiene ordinati eventuali valori che già lo sono, ripetendo su di essi l'ordinamento.

Prima di procedere con l'analisi dell'algoritmo vero e proprio, vogliamo però fare un piccolo sunto delle caratteristiche degli algoritmi presen-

TABELLA 1

NOME	CASO MIGLIORE	CASO MEDIO	CASO PEGGIORE	MEMORIA	STABILE	METODO
Insertion Sort	O(n)	O(n2)	O(n2)	O(1)	Si	Inserimento
Selection Sort	O(n2)	O(n2)	O(n2)	O(1)	No	Selezione
Merge Sort	O(n logn)	O(n logn)	O(n logn)	O(n)	Si	Merge
Heap Sort	**O(n** logn)	**O(n** logn)	**O(n** logn)	**O(1)**	**No**	**Selezione**
Quick Sort	O(n logn)	O(n logn)	O(n logn)	O(logn)	No	Partizionamento

tati fino ad ora e di altri che presenteremo più avanti (**Tabella 1**).

Si nota quindi che l'Heap Sort è uno dei migliori algoritmi esistenti, avendo praticamente una complessità bassa e costante. Esso utilizza la stessa tecnica di selezione del meno efficiente Selection Sort, ovvero i dati vengono scambiati all'interno della stessa struttura dati e non vengono "inseriti" come nell'Insertion Sort.

L'ultimo algoritmo, il Quick Sort, si contende il podio con l'Heap Sort, in quanto più efficiente in alcune situazioni e migliorabile tramite opportune ottimizzazioni. Mentre l'Heap Sort è oggetto di questa puntata, il Quick Sort lo sarà della prossima in modo da fornire un quadro completo degli algoritmi di ordinamento, della progettazione di algoritmi e della loro complessità.

Gli alberi binari e gli Heap

Prima di poter spiegare il funzionamento dell'Heap Sort bisogna introdurre gli alberi. In informatica gli alberi sono strutture dati che consentono di organizzare i dati in un certo modo.

Un albero generico è fondamentalmente composto da uno o più nodi (che contengono le informazioni) connessi, tramite degli archi, ad altri nodi. Si instaura quindi una gerarchia tra un nodo padre connesso tramite arco orientato ad un nodo figlio. Un nodo senza padri è detto **radice**, mentre i nodi senza figli sono detti **foglie**.

La caratteristica degli alberi è che possono essere costruiti secondo alcune proprietà per rendere più semplice la ricerca dei dati al loro interno. Un tipo di albero che rende più agevole la ricerca di determinati elementi è l'albero binario, ovvero un albero in cui ogni nodo ha, al più, due figli. In questo modo mentre un albero è rappresentato generalmente da liste (ovvero oggetti che puntano ad altri oggetti), un albero binario può essere facilmente rappresentato con degli array, di più semplice manipolazione e che richiedono un utilizzo di memoria sicuramente inferiore.

Per fare un esempio, una rappresentazione grafica di un albero binario e della sua rappresentazione come array è riportata nella **Figura 1**.

Dall'immagine si evince che un albero binario rappresentato sotto forma di array è facilmente manipolabile. Infatti è abbastanza semplice accedere un determinato elemento. Ad esempio il figlio sinistro di un nodo padre ha come posizione 2i+1 (e quindi indice dell'array 2k), mentre quello destro 2i+2 (e indice 2k+1).

Un **Heap** non è altro che un albero binario in cui i nodi padre hanno un elemento con valore maggiore dei nodi figli. Questo implica automaticamente che il nodo **radice** sia quello con chiave maggiore, ed è proprio su questa caratteristica dell'heap che si basa l'heap sort. Altra cosa importante è che se si rimuove un elemento dell'heap, l'heap "spezzato" risultante non sarà

FIGURA 1

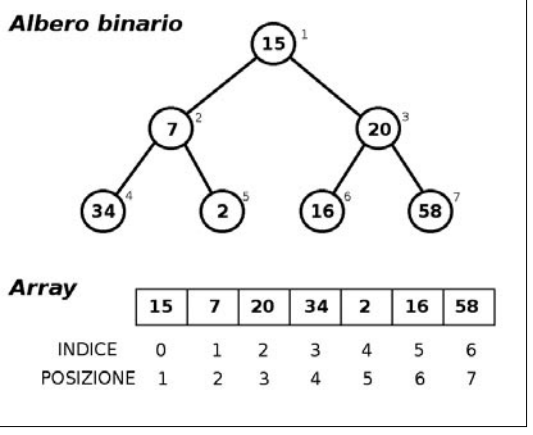


FIGURA 2

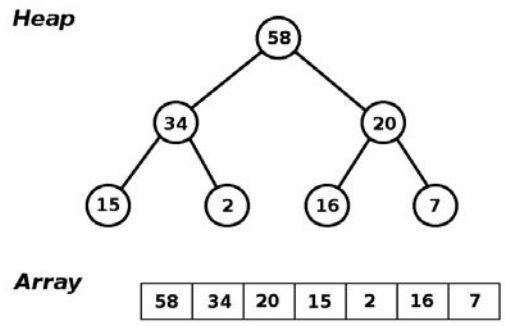
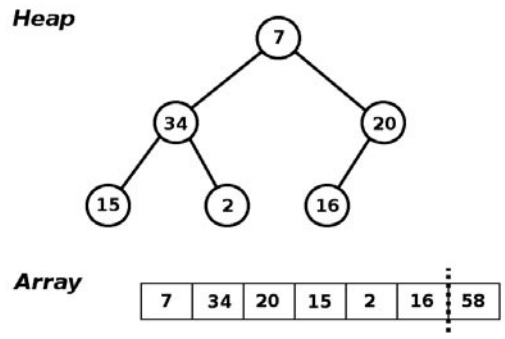


FIGURA 3



uno heap, ma sarà abbastanza facile ripristinare questa proprietà grazie alla stessa procedura con qui l'heap è stato costruito. Questo è possibile scambiando il padre con il figlio più grande e poi ripetendo la medesima procedura su quel figlio fino a quando l'elemento non si trova nella giusta posizione. In pratica si affonda il nodo fino a che la proprietà dell'heap è stata ripristinata.

L'Heap Sort

Ma come funziona l'Heap Sort? La logica di funzionamento dell'Heap Sort non è immediata, infatti l'Heap Sort esegue i seguenti passi:

1. Si convertono tutti i dati (di dimensione N) in un heap massimo;
2. Si scambia il primo elemento (il più grande) con quello in ultima posizione;
3. Si ricompone l'heap sugli N-1 elementi;
4. Si ripetono i punti 2 e 3 (con l'indice N che decresce);

Per fare un esempio pratico proviamo ad ordinare un array di sette elementi; come mostrato nelle **Figure 2, 3 4 e 5** tale array viene innanzitutto convertito in uno heap massimo.

Viene quindi scambiato il primo elemento con quello in ultima posizione, e viene ripristinata la struttura di heap massimo sugli N-1 elementi.

Ripristinando l'heap si avrà che il secondo elemento più grande è in prima posizione, il più grande è in ultima e gli N-1 elementi dell'array sono un heap. Si procede iterativamente a scambiare il primo elemento con l'ultimo dell'heap, fino a quando i dati non sono ordinati.

Implementazione

Andando ad analizzare il codice possiamo vedere l'implementazione dell'algoritmo sia tutto sommato abbastanza semplice. Apriamo un semplice editor di testo, come il notepad o gedit, e creiamo un file di nome **HeapSortAlgorithm.java**, quindi inseriamo il codice sottostante.

FIGURA 4

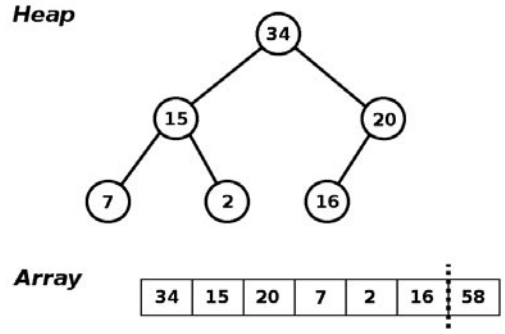
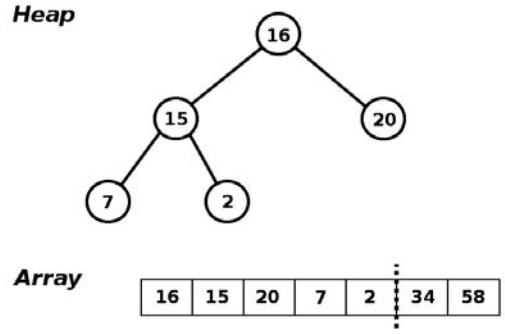


FIGURA 5



```
class HeapSortAlgorithm extends SortAlgorithm {
  void sort(int a[]) throws Exception {
```

Il nome della classe deve essere ovviamente uguale a quello del file che lo contiene. Tale classe estende la classe SoltAlgorithm che è alla base dell'applet ed è stata già presentata nelle scorse puntate (il codice di tale classe e della classe SortItem sono allegati al codice sorgente di questo articolo). Si crea quindi il primo metodo, di tipo void (poiché non restituisce valori di ritorno), che prende in input un array a e che rilancia eventuali eccezioni.

Si prosegue poi estrapolando la lunghezza dell'array a, assegnandola alla variabile N.

```
int N = a.length;
```

Come abbiamo già detto, prima di procedere con l'ordinamento dell'array dobbiamo convertirlo in un heap binario massimo, questo viene fatto agendo iterativamente sui vari nodi con la funzione **downheap**, che ha, appunto, il compito di trasformarlo.

```
for (int k = N/2; k > 0; k--) {
  downheap(a, k, N);
  pause();
}
```

Adesso che si ha un heap binario massimo, si deve sostituire il primo elemento (dell'heap, cioè l'elemento massimo) con l'ultimo (dell'array). In questo modo l'elemento più grande finisce in ultima posizione. Ciò è possibile grazie ad una variabile di appoggio T che consente di scambiare i due valori.

```
do {
  int T = a[0];
  a[0] = a[N - 1];
  a[N - 1] = T;
```

Ovviamente dobbiamo tenere presente che a questo punto l'array è "idealmente" diviso in due parti, in cui la prima parte rappresenta l'heap binario massimo, mentre la parte finale contiene gli elementi dell'array.

Questa divisione logica permette di porre l'attenzione sul fatto che è necessario, spostare l'indice finale dall'ultima posizione alla penultima, in quanto l'ultimo elemento risulta, ovviamente, ordinato.

```
N = N - 1;
pause(N);
```

Non resta che ripristinare la struttura dell'heap sugli N-1 elementi, in modo tale che in prima posizione sia presente il nuovo elemento maggiore (ovvero il secondo maggiore rispetto a quello che abbiamo appena messo in ultima posizione), per poi spostarlo nella nuova ultima posizione (ovvero N-1)

```
downheap(a, 1, N);
```

E si procede così iterativamente (grazie al ciclo do-while), fino a quando l'indice non raggiunge la prima posizione che rappresenterà l'elemento più piccolo.

```
  } while (N > 1);
}
```

Veniamo quindi al vero e proprio fulcro dell'algoritmo di ordinamento Heap Sort, la funzione **downheap**, chiamata in alcuni libri anche maxheapify, ovvero la funzione che consente di creare un heap binario massimo (con l'elemento maggiore in testa all'heap e quindi primo elemento dell'array).

Abbiamo visto che alla funzione viene passato l'array, la sua dimensione N e un intero k. Questo intero vale, nella prima chiamata, N/2, ed è abbastanza facile capirne il perché: come abbiamo detto, infatti, un heap binario massimo ha al più un numero di foglie (cioè figli "finali") pari a quella di tutti i nodi padri più uno, in termini più semplici, prendendo l'elemento a metà dell'array siamo sicuri di aver preso l'ultimo nodo del penultimo "livello", ovvero del livello che ha delle foglie.

```
void downheap(int a[], int k, int N)
                              throws Exception {
```

In quella chiamata il valore di k decresce, così viene creato l'heap sui primi N/2 elementi e automaticamente risulteranno ordinati i figli (cioè gli altri N/2 elementi). Si procede, infatti, ponendo in una variabile temporanea il valore corrispondente all'indice k (-1 perché si tratta di un array).

```
  int T = a[k - 1];
```

E si fa un ciclo fino a quando k è inferiore a N/2 (cioè fino a quando non si trova un figlio con il quale si possa scambiare il valore con il padre).

```
while (k <= N/2) {
  int j = k + k;
```

E si assegna ad una variabile j, il valore di due volte k, ovvero il valore che, nell'array corrisponde al figlio destro dell'elemento. Se quel figlio esiste (cioè il suo valore è inferiore a N) e il valore dell'elemento al suo interno è maggiore di quello dell'elemento sinistro, si aumenta j di un elemento, in modo che nella selezione successiva sarà il figlio con il valore "maggiore" ad essere confrontato con il padre.

```
if ((j < N) && (a[j - 1] < a[j])) { j++; }
```

Si verifica, poi, che il valore dell'elemento nel figlio attuale (cioè il più grande tra destro e sinistro) sia minore o uguale a quello del padre, se questo accade si esce dal ciclo e tra padre e figlio la situazione rimane immutata, preservando la proprietà dell'heap,

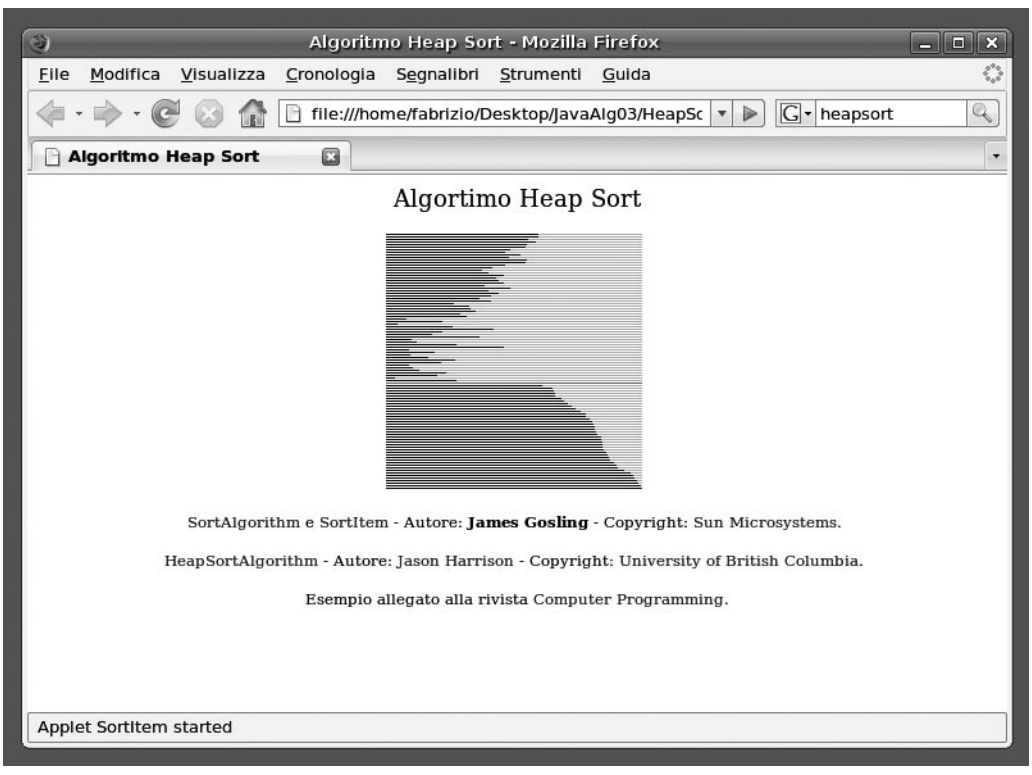
```
if (T >= a[j - 1]) { break; }
```

altrimenti si assegna al padre il valore del figlio e si prova a ripristinare la proprietà dell'heap sul sottoalbero del figlio, scambiando le due chiavi k e j, ovviamente se per esso esiste qualche figlio con il quale confrontare l'elemento (ricordate quando nei paragrafi successivi dicevamo che avremmo "affondato" il valore fino a ripristinare la proprietà dell'heap? È quello che stiamo facendo).

```
 else {
    a[k - 1] = a[j - 1];
    k = j;
    pause();
   }
}
```

Infine si assegna al k-simo elemento il valore di T, ovvero si riassegna "ufficialmente" il valore che avevamo messo nella variabile temporanea alla chiave corrente. In pratica se anche abbiamo fatto uno (o cento) scambi tra padre figlio, in realtà l'elemento in k-sima posizione nell'array (cioè l'elemento a[k-1]) ha ancora il valore vec-

FIGURA 6



chio, mentre noi sappiamo che se siamo usciti dal ciclo il valore corretto è quello iniziale del padre che cercavamo di affondare.

```
    a[k - 1] = T;
    pause();
  }
}
```

Il motivo per cui, invece, la chiamata al downheap nella seconda parte del metodo principale viene fatta con k=1 è abbastanza chiara, avendo spostato la radice in fondo all'array, si deve procedere ad affondare il valore dei figli fino a quando l'heap non è ripristinato.

La complessità dell'Heap Sort

Adesso che abbiamo implementato l'algoritmo, procediamo a calcolarne la complessità. Come abbiamo visto viene utilizzata la funzione **downheap** per costruire l'heap massimo, e tale funzione impiega un tempo pari a T(2n/3) + O(1), questo perché un sottoalbero di un figlio ha al più dimensione 2n/3, e questo è vero nel caso "peggiore" in cui uno dei due sottoalberi è completo.

Senza fare calcoli troppo noiosi e complicati è facile comunque intuire che questa funzione ha una complessità inferiore ad O(n), in quanto lavora ogni volta su una porzione dei dati, ovvero ha complessità pari a O(*logn*).

Visto che l'algoritmo richiede l'esecuzione di almeno n volte del metodo **downheap**, si evince che la complessità totale è nell'ordine di O(n *logn*), e questo indipendentemente dal caso peggiore, medio o migliore, in quanto la procedura non tiene conto di dati già ordinati (o porzioni di essi).

A dire la verità si dovrebbe aggiungere la chiamata iniziale per costruire l'heap, ma poiché questa ha una complessità pari a O(n), e quindi inferiore a quella dell'algoritmo di ordinamento, non viene considerata influente da quel punto di vista. Come si nota dal codice, sembra che la costruzione dell'heap possa avere al più complessità O(n *logn*), ma poiché ogni volta l'insieme dei dati da ordinare decresce, O(n) rappresenta la complessità reale di questa chiamata.

L'applet Java

L'applet si compone di tre elementi, di cui due fissi, il file **SortItem.java**, adibito alla creazione dell'applet ed il file **SortAlgorithm.java** che opera sugli oggetti di tipo SortItem per ordinarli; il terzo file può invece cambiare, poiché definisce l'algoritmo specifico da usare per ordinare i numeri (la classe Java dell'algoritmo di ordinamento, infatti, generalmente estende la classe SortAlgorithm), ed il suo nome viene passato come parametro quando si richiama l'applet. In questo modo è abbastanza facile implementare diversi algoritmi di ordinamento con nomi diversi, e poi darli in pasto al codice dell'applet passandogli il nome dell'algoritmo.

Nell'esempio è stato creato un file di nome **HeapSortAlgorithm.java**, la cui classe estende la classe SoltAlgorithm. Quando poi dovremo richiamare l'applet dalla pagina web, richiameremo il file **SortItem.class** (la versione compilata in bytecode di SortItem.java), dicendogli che l'algoritmo da usare sarà l'HeapSort.

Sul sito ftp.infomedia.it è possibile trovare i sorgenti dell'algoritmo di ordinamento. Per far funzionare correttamente l'applet di ordinamento è necessario compilare i file sorgenti e poi creare una pagina html che permetta il caricamento dell'applet. Una volta scaricato ed installato il JDK (vedere box di approfondimento pubblicato nella prima puntata), è possibile compilare i file Java con un unico comando, dalla cartella che contiene i sorgenti delle tre classi proposte:

```
javac HeapSortAlgorith.java
```

A questo punto è necessario creare una pagina html con il codice per richiamare l'applet in questione; basta aprire un comunissimo editor di testo e creare un file con estensione ".htm" (ad esempio HeapSort.htm), inserendo questo tag all'interno del body:

```
<applet codebase="./" code=SortItem.class
                          width=200 height=200>
<param name="alg" value="HeapSort">
</applet>
```

dove **codebase** identifica la cartella che contiene le classi Java (se ad esempio si carica l'applet da un percorso differente), **code** deve contenere il nome della classe SortItem, e **width** ed **height** rappresentano la dimensione a video. Per richiamare l'algoritmo specifico bisogna inserire il tag <param> passandogli il parametro **alg** come

nome del parametro ed HeapSort (cioè la prima parte del nome della classe) come suo valore. Per visualizzare l'applet, infine, aprire il file html con il proprio browser internet

Conclusioni

In questa puntata abbiamo mostrato come sia importane non solo progettare un algoritmo efficiente, ma anche utilizzare un'opportuna struttura dati. L'Heap Sort ha infatti una complessità molto simile a quella del Merge Sort, tuttavia l'Heap consente di risparmiare una notevole quantità di memoria ausiliaria.

E sono proprio le peculiarità di ogni algoritmo che apportano vantaggi e svantaggi: se si guadagna qualcosa in termini di efficienza, si perde in termini di memoria o di stabilità.

Esiste quindi un algoritmo di ordinamento "migliore"? Nella prossima lezione presenteremo il Quick Sort che rappresenta forse il miglior compromesso ad oggi esistente.

L'applet SortItem e la classe SortAlgorithm sono state create da James Gosling e sono di proprietà della Sun Microsystems.
Il codice è disponibile e modificabile gratuitamente per scopo non commerciale o commerciale senza fini di lucro, senza nessun supporto e a patto di non modificare il copyright originale.
La classe Insert Sort è stata creata da Lars Marius Garshol ed è stata modificata dall'autore dell'articolo.

RIQUADRO 1 Codice e Licenze

CODICE ALLEGATO

ftp.infomedia.it **Ordinamento3**