Java e gli algoritmi di ordinamento

Continua la serie di articoli sugli algoritmi di ordinamento in Java. In questo secondo articolo focalizzeremo l'attenzione sugli algoritmi ricorsivi.

Seconda puntata

di Fabrizio Ciacchi

Nella prima puntata abbiamo accennato alla realizzazione di un semplice algoritmo di ordinamento (Insertion Sort), per introdurre, da una parte i concetti basilari e le motivazioni dello studio degli algoritmi, dall'altra i costrutti in Java che permettono di creare un'applet che mostra il funzionamento dell'algoritmo stesso.

In questo secondo articolo ci concentreremo maggiormente sulla parte teorica che sta dietro all'algoritmo che vogliamo presentare, il Merge Sort, che differisce dal precedente algoritmo in quanto è progettato per funzionare ricorsivamente (ovvero su se stesso).

Quando sviluppiamo un programma, non facciamo altro che utilizzare degli algoritmi che cooperano per raggiungere il risultato finale che ci siamo preposti. Si può ben capire, quindi, che imparare a progettare ed ottimizzare algoritmi sia molto importante per la realizzazione di programmi di una certa complessità o che devono avere determinati requisiti di prestazioni/affidabilità.

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

Lo studio degli algoritmi, o meglio, lo studio della complessità degli algoritmi è lo studio della complessità "asintotica" cui un algoritmo tende. Per capirsi, quello che andiamo a studiare non è come l'algoritmo funziona nel mondo reale (se è più o meno veloce, per intendersi), ma piuttosto come questo, teoricamente, si comporterà.

Gli algoritmi di ordinamento rappresentano un caso particolare nello studio degli algoritmi e mostrano come sia possibile, partendo da un singolo problema (l'ordinamento, appunto), raggiungere lo stesso risultato in modo diverso, andando a diminuire la complessità con ottimizzazioni ed "escamotage" sempre più efficienti (a volte, ad esempio, a scapito dell'utilizzo di memoria).

Calcolare la complessità degli algoritmi è abbastanza semplice (anche se imparare a farlo bene non è così immediato). Posto che un'istruzione (un assegnamento, ad esempio) abbia complessità O(1), sia cioè risolvibile in un tempo finito che non dipende dalla quantità di dati su cui si opera, e che per ogni "ciclo", la complessità aumenti n volte (dove n rappresenta la dimensione dei dati su cui si opera), allora si effettua un semplice calcolo e si stabilisce se la complessità è O(n), O(n**log*n), O(n2) e così via.

Per fare un esempio, è dimostrato che l'ordinamento ha, teoricamente, come complessità massima O(n2), ciò significa che se progettiamo un algoritmo che ha complessità maggiore, sicuramente stiamo sbagliando qualcosa. Questo

metodo di paragone è molto utile durante la progettazione di qualsiasi algoritmo, in quanto ci dice a priori l'efficienza dell'algoritmo stesso.

Gli algoritmi ricorsivi

Generalmente siamo abituati a scrivere o studiare porzioni di codice che tentano di raggiungere il risultato in modo abbastanza lineare, ovvero preso in input l'insieme dei dati su cui lavorare, l'algoritmo esamina ogni singolo valore e poi effettua i calcoli per ottenere il risultato. Facciamo un esempio, immaginiamo di dover cercare un nominativo all'interno dell'elenco telefonico. Il metodo più immediato che ci viene in mente è quello di scorrere l'elenco dalla A alla Z fino a quando non si trova il nominativo cercato, tuttavia si può facilmente intuire che questo modo di procedere è poco efficiente, soprattutto se siamo sfortunati e il nome della persona è, ad esempio, **Rossi Mario**, cosa che ci costringerebbe a scorrere troppi nomi prima di raggiungere il risultato.

Un metodo sicuramente più efficiente sarebbe quello aprire l'elenco telefonico a metà e chiedersi se Rossi Mario sta nella prima metà (dalla A alla L) o nella seconda (dalla M alla Z); stabilito che si trova nella seconda metà, andrebbe applicato il medesimo algoritmo al sottoinsieme in questione, fino a quando, ovviamente, non si trova il numero di telefono della persona cercata.

Volendo schematizzare, posto S l'insieme di riferimento e X l'elemento cercato, in pseudocodice, si avrebbe una funzione di questo tipo.

```
CercaElenco(insieme S) {
   <fai una valutazione A>
   <dividi S in due sottoinsiemi S' e S''>
   <L'elemento X è nell'insieme S'?>
     SI – CercaElenco(insieme S')
     NO – CercaElenco(insieme S'')
}
```

Il che riassume sostanzialmente quanto ci siamo detti, ovvero si esegue lo stesso algoritmo ogni volta in un sottoinsieme più piccolo per trovare la soluzione. Questo modo di procedere è chiamato **ricorsione**, e, a differenza dei modelli iterativi in cui sono presenti uno o più cicli, consiste proprio nel richiamare lo stesso algoritmo (ricorsione diretta) ogni volta su una porzione di dati più piccola, fino a quando non viene trovata la soluzione.

Ma come è possibile "trovare la soluzione"? I lettori più attenti avranno notato lo pseudocodice <fai una valutazione A>, questa istruzione, che nella realtà può essere composta da più istruzioni, è quella che riesce a risolvere il problema direttamente e ritorna il valore della computazione. Sempre riferendoci all'esempio, l'istruzione in questione potrebbe dapprima valutare se S è un insieme di un solo elemento, e in quel caso vedere se contiene l'elemento X cercato (e restituire il numero di telefono corrispondente) oppure no (e ritornare quindi un errore o comunque un messaggio che l'elemento non è stato trovato).

In questo caso particolare la ricorsione è di tipo lineare, ovvero viene fatta una sola chiamata ricorsiva (o sull'insieme S' o su S"), sarebbe non-lineare altrimenti.

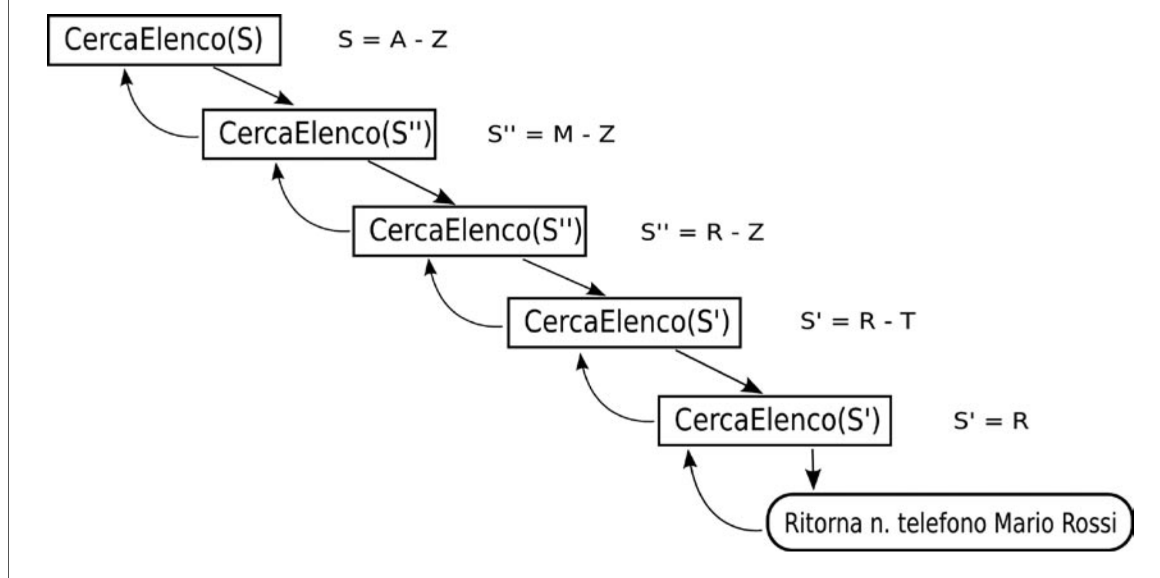
Il Divide et Impera

Un caso particolare nella metodologia ricorsiva si ha quando si vuole risolvere un problema (ad esempio l'ordinamento) su una serie di dati. Il problema è che la ricorsione classica è molto efficiente quando si lavora ogni volta su una porzione più piccola, ma può diventare di utilizzo meno immediato quando invece si vuole che vengano coinvolti tutti i dati nella soluzione del problema. Infatti, la ricorsione classica generalmente inoltra il valore ritornato fino alla prima chiamata; nel nostro esempio quando X viene trovato o non trovato, il risultato viene ritornato dalla n-sima chiamata e, in un effetto a cascata, lo stesso risultato viene ritornato, quindi, dalla prima chiamata del metodo (**Figura 1**).

Se, invece, come abbiamo detto, vogliamo che vengano coinvolti tutti i dati, oltre ad applicare ricorsivamente il metodo su ogni porzione dell'insieme, dobbiamo effettuare, alla fine, un lavoro di ricombinazione (e qui sta la differenza) per unire i vari risultati. Questa metodologia è chiamata comunemente "Divide et Impera" (tradotto, Dividi e Conquista) che in pratica consiste nel suddividere in "sottoinsiemi" (chiamate Partizioni) i dati su cui opera un algoritmo e nel risolvere il problema su di essi, per poi ricomporre alla fine i risultati trovati su ogni singolo sottoinsieme (Partizione) tramite, appunto, un lavoro di combinazione. In pseudo-codice:

```
DividEtImpera(S) {
  <se S ha dimensione inferiore a m> {
```

FIGURA 1 Esempio delle chiamate ricorsive di CercaElenco. Ogni volta si cerca in una porzione dell'elenco più piccola e una volta trovato il numero della persona cercata, il risultato viene ritornato fino alla prima chiamata del metodo.




```
    <risolvi direttamente il problema>
  } altrimenti {
    <dividi S in B sottoinsiemi S1, S2,
                                ..., Sb>
    DividEtImpera(S1);
    DividEtImpera(S2);
    ...
    DividEtImpera(Sb);
    <combina i risultati ottenuti>
  }
}
```

Posto che i diversi sottoinsiemi siano bilanciati (cioè equamente ripartiti), supposto che la risoluzione diretta al problema richieda O(1), nel caso in cui si operi un Divide et Impera su tutti i sottoinsiemi, allora la complessità dell'algoritmo è generalmente O(n**logn*).

Il Merge Sort

Il Merge Sort è un algoritmo di ordinamento **ricorsivo** (utilizza, infatti, la metodologia del Divide et Impera) e **stabile**, ovvero mantiene eventualmente ordinati i dati se già lo sono. È considerato uno dei migliori algoritmi perché ha complessità, sia nel caso peggiore che nel caso medio che nel migliore, di O(n**logn*), ovvero una complessità inferiore di quasi un ordine alla O(n2) raggiunta dall'Insertion Sort e considerata asintotica per gli algoritmi di ordinamento.

Il funzionamento del Merge Sort e la sua implementazione tramite il linguaggio Java sono molto semplici. In pratica si suddivide l'insieme di dati di partenza in unità sempre più piccole tramite una funzione di **Split**, e quando si ha ogni singolo elemento, lo si ricompone (**Merge**) ma in maniera ordinata. Si può fare riferimento all'immagine per capire meglio il concetto (**Figura 2**).

Come ogni classe Java, si inizializza la classe, estendendo tuttavia la classe **SortAlgorithm** (vedere paragrafo successivo), che è una classe di ordinamento generica attraverso la quale potremo utilizzare l'algoritmo nell'applet.

```
class MergeSortAlgorithm extends
                          SortAlgorithm {
```

Si ha, quindi, il primo costruttore che prende in input l'array a[], il limite inferiore (lo0) e quello superiore (hi0) della computazione. Le eventuali eccezioni vengono rilanciate.

```
void sort(int a[], int lo0, int hi0)
                     throws Exception {
```

Si assegnano, innanzitutto, alle variabili locali i valori dei due limiti,

```
int lo = lo0;
int hi = hi0;
```

E si richiama il metodo **pause** della superclasse sui due limiti

```
pause(lo, hi);
```

Nel caso in cui il limite inferiore è maggiore o uguale a quello superiore, significa che l'insieme è composto di un solo elemento. Questa condizione è necessaria per poter far iniziare la funzione di **Merge** dopo aver eseguito i vari **Split.**

```
if (lo >= hi) { return; }
```

Si calcola, quindi, l'elemento di mezzo e si applica ricorsivamente il Sort su ogni metà dell'insieme dei dati di partenza. Si tratta a tutti gli effetti dello **Split** citato poco fa.

```
int mid = (lo + hi) / 2;
sort(a, lo, mid);
sort(a, mid + 1, hi);
```

Da qui in poi, invece, effettuiamo l'operazione di **Merge**, ovvero ricomponiamo i risultati ricorsivamente. Fermiamoci un attimo ed analizziamo cosa è successo: nella prima parte dell'algoritmo abbiamo analizzato il caso base in cui l'elemento è composto da un solo elemento, caso che fa ritornare indietro nella ricorsione e fa iniziare il Merge, altrimenti continuiamo ad applicare ricorsivamente il Sort, ogni volta sulle due metà dei dati di partenza (Split).

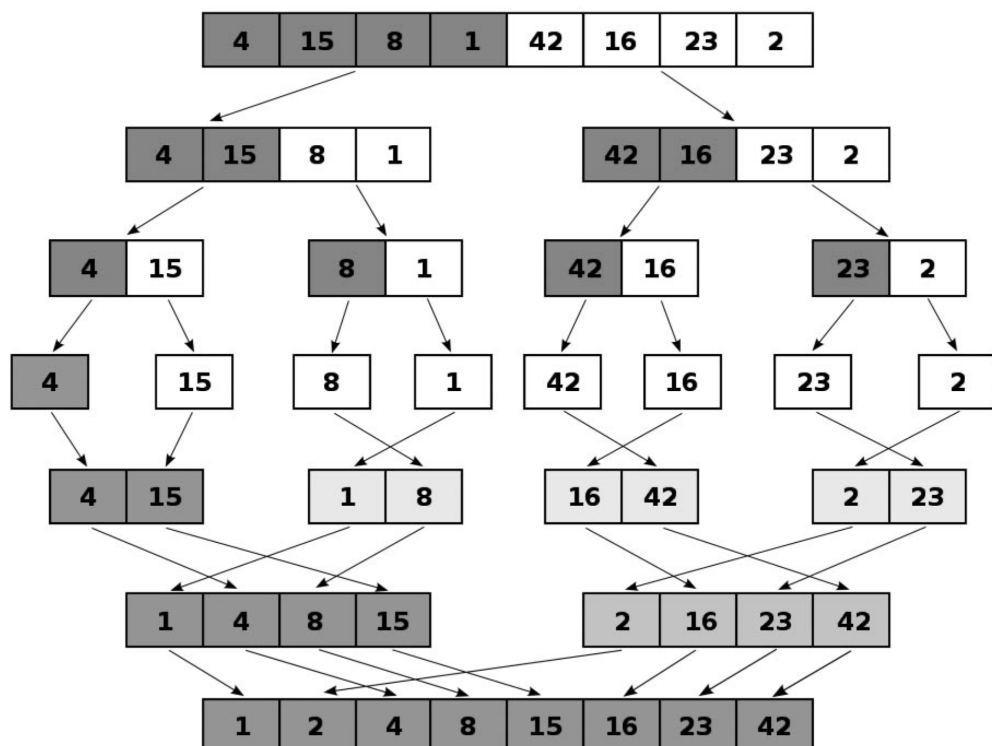
Quando abbiamo applicato il Sort su insiemi composti da un solo elemento ed, in pratica, è finita la ricorsione, iniziamo ad unire i singoli elementi in array di volta in volta più grandi (fino alla ricostruzione dell'insieme), ma nell'unione procediamo anche a metterli in ordine. Posto che il limite inferiore finisca dove è presente l'elemento di mezzo, mentre il limite superiore comincia subito dopo,

```
int end_lo = mid;
int start_hi = mid + 1;
```

Facciamo un ciclo che ha come condizione il fatto che il limite inferiore raggiunga la sua fine e il limite superiore raggiunga il suo inizio.

```
while ((lo <= end_lo) && (start_hi <= hi)) {
```

FIGURA 2 Funzionamento del Merge Sort. I dati vengono suddivisi in porzioni sempre più piccole ed i valori vengono ricombinati in modo ordinato, fino a quando l'ordinamento non è completo.



Il seguente comando setta (graficamente nell'applet) il limite inferiore ed esce se viene richiesto di fermare l'ordinamento

```
pause(lo);
if (stopRequested) {
   return;
}
```

Qui si consuma, invece, il vero e proprio Merge, ovvero se il valore dell'elemento nel limite inferiore è inferiore al valore dell'elemento di mezzo + 1 (ovvero l'elemento iniziale del limite superiore), allora incrementa il limite inferiore,

```
if (a[lo] < a[start_hi]) {
   lo++;
} else {
```

altrimenti si assegna ad una variabile T di appoggio il valore dell'elemento iniziale del limite superiore.

```
int T = a[start_hi];
```

e si fa un ciclo per spostare ogni elemento di una posizione fino al limite indicato.

```
for (int k = start_hi - 1; k >= lo; k--) {
   a[k+1] = a[k];
   pause(lo);
}
```

Assegnando, infine, il valore di start_hi al limite inferiore ed aumentando, di conseguenza, tutti gli indici.

```
a[lo] = T;
lo++;
end_lo++;
start_hi++;

} // fine if
} // fine while
} // fine sort
```

Un po' spaesati? È comprensibile. Cosa abbiamo fatto esattamente? Con la parte di Merge abbiamo in pratica confrontato ogni elemento dei due insiemi, partendo dal primo elemento di ciascuno (lo per il primo insieme, start_hi per il secondo insieme) ed abbiamo spostato in avanti il limite lo, se il valore in esso contenuto era più piccolo del primo elemento del secondo insieme, abbiamo invece spostato l'elemento del secondo insieme (quello in start_hi) al posto di lo, se invece era maggiore (ed abbiamo spostato poi tutti gli indici di conseguenza). Cercando di semplificare, abbiamo unito i due insiemi mettendo gli elementi più piccoli sempre in prima posizione.

L'altro costruttore, invece, prende in input solamente l'array a[] e fa una chiamata al precedente costruttore con limite inferiore zero (il primo elemento dell'array) e come limite superiore la lunghezza (-1, perché è un array). Si chiude, infine la classe MergeSortAlgorithm.

```
void sort(int a[]) throws Exception {
     sort(a, 0, a.length-1);
   } // fine sort
} // fine classe
```

Qual è, quindi, la complessità? Poiché il Merge Sort opera ogni volta su un sottoinsieme che è la metà dell'insieme originale, si andranno a spostare gli elementi solo quanto necessario. Mentre nell'Insertion Sort si processavano comunque tutti gli elementi (e la complessità era O(n2), nel Merge Sort, è facile intuire che quando si lavora nell'insieme di sinistra non si opera in quello di destra e viceversa. La complessità risulta (attraverso calcoli di complessità che qui non riportiamo) essere, almeno a logica, inferiore a quella dell'Insertion Sort, ma sicuramente maggiore di quella lineare (in quanto alcuni elementi vengono "toccati" più di una volta), ed è cioè O(n**logn*), detta anche complessità sovralineare.

> Per poter compilare ed eseguire il programma sono necessari tre strumenti, il compilatore Java (per compilare i file sorgenti), le librerie Java e il plugin Java per il proprio browser (che cambia anche in base al sistema operativo). Fortunatamente per ottenere tutti gli strumenti necessari al corretto funzionamento, sia per Windows che per Linux, è possibile scaricare il JDK (Java Development Kit), il quale contiene al suo interno il JRE (Java Runtime Environment) con i relativi plugin per abilitare la JVM per il proprio browser. Il JDK è scaricabile dal sito http://java.sun.com/javase/downloads/index.jsp.
>
> **RIQUADRO 1** Installare Java

FIGURA 3 Ecco come appare l'applet di ordinamento allegata all'articolo. L'applet genera dei dati (da 1 a N) non ordinati, cliccando su di essa si può vedere il funzionamento dell'algoritmo di ordinamento.

Per chiudere possiamo dire che il Merge Sort non è un algoritmo di tipo "in place" (o "in loco"), ovvero utilizza più memoria di quella che viene allocata per i dati di input; in pratica vengono stanziate (a causa della natura ricorsiva) un numero di variabili maggiori di n (dove n è la dimensione dell'input), ed in questo senso per quanto più efficiente, il Merge Sort consuma più memoria dell'Insertion Sort. In computer moderni questo non è un problema, ma se ci trovassimo a progettare un algoritmo per un dispositivo embedded o per un server di produzione, potrebbe essere una proprietà da prendere in considerazione.

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 contenuta, 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 **MergeSortAlgorithm.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 il "MergeSort".

All'interno del CD allegato alla rivista o sul sito www.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), è possibile compilare i file Java con un unico comando, dalla cartella che contiene i sorgenti delle tre classi proposte:

```
javac MergeSortAlgorithm.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 testi (Notepad o gEdit) e creare un file con

estensione ".htm" (ad esempio MergeSort.htm), inserendo questo tag all'interno del body:

```
<applet codebase="./" code=SortItem.class
                        width=200 height=200>
<param name="alg" value="MergeSort" />
</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 e **MergeSort** (cioè la prima parte del nome della classe) come suo valore. Per visualizzare l'applet aprire il file html creato con il proprio browser internet.

Conclusioni

Con questo secondo articolo, oltre a presentare un algoritmo di ordinamento più efficiente, abbiamo voluto introdurre un concetto nuovo, quello del "divide et impera", che altro non è che la suddivisione di un problema difficile da risolvere in sottoproblemi di più facile soluzione; come si è visto ciò permette di semplificare notevolmente l'attività di studio, progettazione e creazione di un algoritmo, e può essere usato per diminuirne la complessità.

Si lascia al lettore la possibilità di "giocare" con gli algoritmi proposti e lo si invita a provare nuove soluzioni; sul sito web di Computer Programming www.infomedia.it, nella sezione "Listati On Line", è presente il codice sorgente delle due lezioni finora presentate.

Le classi **SortItem** e **SortAlgorithm** sono state create da James Gosling e sono di proprietà della Sun Microsystems. Il codice è 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 **MergeSortAlgorithm** è stata creata da Jason Harrison ed è di proprietà dell'Università di British Columbia.

RIQUADRO 2 Sorgenti e licenze

CODICE ALLEGATO

ftp.infomedia.it **Ordinamento2**