Algoritmi di ordinamento

Matteo Ferrara

Dipartimento di Informatica - Scienza e Ingegneria

matteo.ferrara@unibo.it

Algoritmi di ordinamento

Definizione informale

Un algoritmo di ordinamento è una sequenza formale di operazioni per ordinare insiemi di dati.

L'ordinamento dei dati è fondamentale, principalmente perché operare su un insieme di dati ordinato è molto più efficiente che operare sullo stesso insieme di dati non ordinato.

A titolo esemplificativo, dato un insieme di cardinalità n, effettuare una ricerca sull'insieme disordinato richiede una **scansione sequenziale** ($O(n)$); una ricerca sull'insieme ordinato è invece attuabile avvalendosi di **binary search** ($O(\log_2 n)$).

Implementazione in place/non in place

Algoritmo in place

L'implementazione di un algoritmo è detta **in place** se la trasformazione dell'input nell'output è operata senza utilizzare memoria aggiuntiva o al più utilizzando una (piccola) quantità di memoria aggiuntiva costante.

A titolo esemplificativo, un algoritmo in place si limita a sovrascrivere la memoria occupata dall'input, oppure utilizza alcune semplici variabili ausiliarie.

Algoritmo non in place

L'implementazione di un algoritmo è detta **non in place** se la trasformazione dell'input nell'output è operata mediante strutture dati aggiuntive che comportano un considerevole aumento di memoria utilizzata.

Insertion Sort

Insertion sort $O(n^2)$

INPUT: $A = [a_1, \ldots, a_n]$ OUTPUT: $A = [a_1, \ldots, a_n]$

1. **for** $j \leftarrow 2$ **to** n **do**
2. $key \leftarrow A[j]$
3. $i \leftarrow j - 1$
4. **while** $i > 0$ **and** $A[i] > key$ **do**
5. $A[i+1] \leftarrow A[i]$
6. $i \leftarrow i - 1$
7. $A[i+1] \leftarrow key$

Nel **caso ottimo** (array già ordinato), la condizione del while non è mai verificata. La complessità dipende quindi dal solo ciclo esterno ($O(n)$). Nel **caso pessimo** (array ordinato in senso inverso), la condizione del while è verificata per ogni elemento del sottoarray ($O(n^2)$). Anche nel **caso medio**, il ciclo interno ha una complessità lineare in n, per cui la complessità complessiva resta $O(n^2)$.

Insertion Sort - Esempio

Esempio

Ordinamento dell'array A=[3,1,10,9,7] mediante *insertion sort*:

- [3,**1**,10,9,7] (*ciclo esterno* $j = 2$)
- [**1**,**3**,10,9,7] (*ciclo interno*)
- [1,3,**10**,9,7] (*ciclo esterno* $j = 3$)
- [1,3,10,**9**,7] (*ciclo esterno* $j = 4$)
- [1,3,**9**,**10**,7] (*ciclo interno*)
- [1,3,9,10,**7**] (*ciclo esterno* $j = 5$)
- [1,3,9,**7**,**10**] (*ciclo interno*)
- [1,3,**7**,**9**,10] (*ciclo interno*)

Obiettivo

1. Implementare l'algoritmo di ordinamento *Insertion Sort* seguendo lo pseudo-codice visto nelle slide precedenti.
2. Testare l'algoritmo implementato sui due array non ordinati forniti nel materiale dell'esercitazione misurando il tempo richiesto.

N.B.: per velocizzare lo sviluppo si consiglia di utilizzare il codice sorgente presente nel materiale dell'esercitazione.

Merge-Sort (1)

L'algoritmo **merge-sort** fu introdotto da *John Von Neumann* nel 1945 e si basa sul paradigma algoritmico **divide et impera**.

Aspetto di rilievo è il costo computazionale dell'algoritmo, **quasi lineare**, che coincide per caso ottimo, pessimo e medio ($\Theta(n \log n)$).

Il costo computazionale ridotto (rispetto ad altri algoritmi di ordinamento) e soggetto a scarsa variabilità ha portato all'adozione di questo algoritmo come **algoritmo di ordinamento standard** in molte librerie di svariati linguaggi di programmazione come Perl, Java, C. . . .

Merge-Sort (2)

Merge-Sort	$O(n \log n)$
INPUT: $A = [a_1, \ldots, a_n]$, $p, r \in \mathbb{N}$	OUTPUT: $A = [a_1, \ldots, a_n]$

1. **if** $p < r$ **then**
2. $\quad q \leftarrow \lfloor (p + r)/2 \rfloor$
3. $\quad$ **Merge-Sort**(A, p, q)
4. $\quad$ **Merge-Sort**$(A, q + 1, r)$
5. $\quad$ **Merge**(A, p, q, r)

La procedura **Merge-Sort** è composta da una serie di chiamate ricorsive atte a dividere (*divide*), ordinare ricorsivamente (*impera*), e infine fondere (*combina*) le sottosequenze generate.
Il processo delle chiamate ricorsive risale quando la sequenza da ordinare ha un solo elemento (implicitamente ordinato).

Merge-Sort (3)

Merge (prima parte)

INPUT: $A = [a_1,\ldots,a_n],\ p,q,r \in \mathbb{N}$ OUTPUT: $A = [a_1,\ldots,a_n]$

1. $i \leftarrow p,\ j \leftarrow q + 1,\ k \leftarrow 1$
2. **while** $i \le q$ **and** $j \le r$ **do** *ordina*
3. **if** $A[i] < A[j]$ **then**
4. $B[k] \leftarrow A[i]$
5. $i \leftarrow i + 1$
6. **else**
7. $B[k] \leftarrow A[j]$
8. $j \leftarrow j + 1$
9. $k \leftarrow k + 1$

…

Merge-Sort (4)

Merge (seconda parte)

```
...
while i ≤ q do                      ricopia leftover sx
    B[k] ← A[i]
    i ← i + 1
    k ← k + 1
while j ≤ r do                      ricopia leftover dx
    B[k] ← A[j]
    j ← j + 1
    k ← k + 1
for k ← p to r do                   ricopia B in A
    A[k] ← B[k-p+1]
```

Merge-Sort (5)

Nonostante **merge-sort** sia tipicamente implementato in versione *ricorsiva*, è possibile costruire una analoga versione *iterativa*.

Lo pseudocodice proposto nelle slide precedenti corrisponde a una versione *ricorsiva non in place*. Questo comporta un dispendio di memoria aggiuntivo, principalmente dovuto all'array di supporto *B*, lungo *n*.
In sostanza, l'utilizzo della memoria aumenta di un fattore $O(n)$.

Una versione *ricorsiva in place* non comporterebbe questo impiego aggiuntivo di memoria.

Si noti infine che, nello pseudocodice proposto, le componenti *ricopia leftover sx* e *ricopia leftover dx* si attivano in modo mutuamente esclusivo.

Merge-Sort (6)

Esempio

Ordinamento dell'array A=[9,3,10,1] mediante *merge-sort*:

$MergeSort(A,1,4)$	[9,3,10,1]
$MergeSort(A,1,2)$	[9,3,10,1]
$MergeSort(A,1,1)$	[9,3,10,1]
$MergeSort(A,2,2)$	[9,3,10,1]
$Merge(A,1,1,2)$	[3,9,10,1]
$MergeSort(A,3,4)$	[3,9,10,1]
$MergeSort(A,3,3)$	[3,9,10,1]
$MergeSort(A,4,4)$	[3,9,10,1]
$Merge(A,3,3,4)$	[3,9,1,10]
$Merge(A,1,2,4)$	[1,3,9,10]

Output A=[1,3,9,10].

Obiettivo

1. Implementare l'algoritmo di ordinamento *Merge-Sort* seguendo lo pseudo-codice visto nelle slide precedenti.
2. Testare l'algoritmo implementato sui due array non ordinati forniti nel materiale dell'esercitazione misurando il tempo richiesto.

N.B.: per velocizzare lo sviluppo si consiglia di utilizzare il codice sorgente presente nel materiale dell'esercitazione.

Quicksort (1)

Ideato da *Tony Hoare* nel 1960, anche l'algoritmo di ordinamento **quicksort** si basa sul paradigma algoritmico **divide et impera**.

Benché il costo computazionale dell'algoritmo nel caso pessimo sia peggiore di merge sort ($\Theta(n^2)$), l'algoritmo garantisce buone prestazioni nel caso medio ($\Theta(n \log n)$).

Spesso le implementazioni del quicksort prevedono una **componente casuale** (versioni randomizzate) che garantisce che nessuna particolare configurazione dell'input porti al costo computazionale pessimo, permettendo quindi di avvalersi del costo medio come upper bound de facto.

Quicksort (2)

Quicksort $O(n^2)$

INPUT: $A = [a_1,\dots,a_n]$, $p, r \in \mathbb{N}$ OUTPUT: $A = [a_1,\dots,a_n]$

1. **if** $p < r$ **then**
2. $q \leftarrow$ **Partition**(A, p, r)
3. **Quicksort**$(A, p, q-1)$
4. **Quicksort**$(A, q+1, r)$

La procedura **Quicksort** è composta da una serie di chiamate ricorsive. Anzitutto, la sotto procedura **Partition** ripartisce (*divide*) l'input in due sotto array non vuoti tali che il primo contenga esclusivamente elementi minori o uguali a quelli del secondo. Seguono poi due chiamate ricorsive a *Quicksort*, che provvedono a ordinare i sotto array (*impera*).
In questo caso, il merge finale dei risultati parziali (*combina*) è banale, in quanto i sotto array, ordinati in loco, sono già ordinati rispetto all'array complessivo.

Quicksort (3)

Partition

INPUT: $A = [a_1, \dots, a_n]$, $p, r \in \mathbb{N}$

OUTPUT: $A = [a_1, \dots, a_n]$

1. $i \leftarrow p - 1$, $x \leftarrow A[r]$ — *scelta del pivot*
2. **for** $j = p$ **to** $r - 1$ **do** — *ordina su pivot*
3. **if** $A[j] \le x$ **then**
4. $i \leftarrow i + 1$
5. $t \leftarrow A[i]$
6. $A[i] \leftarrow A[j]$
7. $A[j] \leftarrow t$
8. $t \leftarrow A[i + 1]$ — *posiziona il pivot*
9. $A[i + 1] \leftarrow A[r]$
10. $A[r] \leftarrow t$
11. **return** $i + 1$

Quicksort (4)

Esempio

Ordinamento dell'array A=[3,1,10,9,7] mediante *quicksort*:

Quicksort(*A*, 1,5)	[3,1,10,9,7]
Partition(*A*, 1,5)	[3,1,10,9,**7**]
ordina su pivot	[<u>3</u>,1,10,9,**7**]
ordina su pivot	[3,<u>1</u>,10,9,**7**]
posiziona pivot	[3,1,**7**,9,<u>10</u>]
return	*q=3*
Quicksort(*A*, 1,2)	[3,1,7,9,10]
Partition(*A*, 1,2)	[3,**1**,7,9,10]
posiziona pivot	[**1**,<u>3</u>,7,9,10]
return	*q=1*
Quicksort(*A*, 1,0)	
Quicksort(*A*, 2,2)	
…	

Quicksort (5)

Esempio

Ordinamento dell'array A=[3,1,10,9,7] mediante *quicksort*:

…	
Quicksort(*A*,4,5)	[1,3,7,9,10]
Partition(*A*,4,5)	[1,3,7,9,**10**]
ordina su pivot	[1,3,7,9,**10**]
posiziona pivot	[1,3,7,9,**10**]
return	*q=5*
Quicksort(*A*,4,4)	
Quicksort(*A*,6,5)	

Output A=[1,3,7,9,10].

Obiettivo

1. Implementare l'algoritmo di ordinamento *Quicksort* seguendo lo pseudo-codice visto nelle slide precedenti.
2. Testare l'algoritmo implementato sui due array non ordinati forniti nel materiale dell'esercitazione misurando il tempo richiesto.

N.B.: per velocizzare lo sviluppo si consiglia di utilizzare il codice sorgente presente nel materiale dell'esercitazione.