MERGESORT

a cura di Alessandro Lucchetta e Luca Gagliano
Realizzato nell'ambito del ***progetto Archimede***
con la supervisione dei Proff. Fabio Breda, Rizzo Gabriele, Valentina Fabbro e Francesco Zampieri
I.S.I.S.S. M.Casagrande, Pieve di Soligo, a.s. 2014/15

Abstract. *In questo articolo ci proponiamo di analizzare le due tipologie ricorsive mergesort e quicksort.*

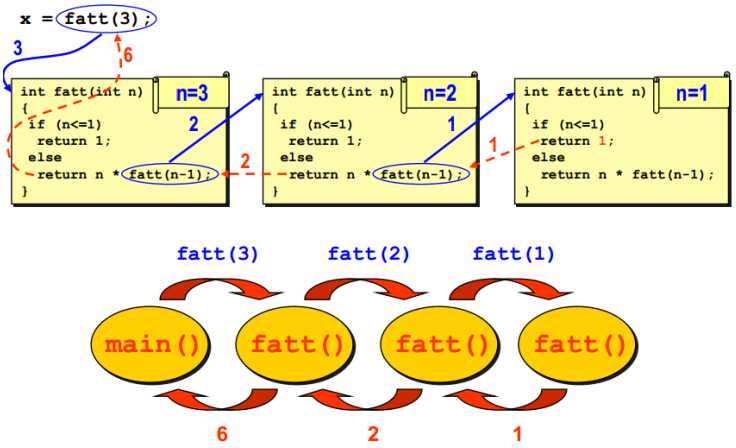
Una funzione è detta ricorsiva se chiama se stessa . Se due funzioni si chiamano l'un l'altra, sono dette mutuamente ricorsive. La funzione ricorsiva sa risolvere direttamente solo casi particolari di un problema detti casi di base se viene invocata passandole dei dati che costituiscono uno dei casi di base, allora restituisce un risultato. Se invece viene chiamata passandole dei dati che NON costituiscono uno dei casi di base, allora chiama se stessa (passo ricorsivo) passando dei DATI semplificati. Ad ogni chiamata si semplificano i dati. Quando la funzione chiama se stessa, sospende la sua esecuzione per eseguire la nuova chiamata. L'esecuzione riprende quando la chiamata interna a se stessa termina. La sequenza di chiamate ricorsive termina quando quella interna (annidata) incontra uno dei casi di base.

PRO

Spesso la ricorsione permette di risolvere un problema anche molto complesso con poche linee di codice.

CONTRO

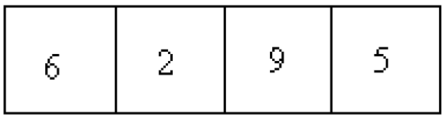
La ricorsione è poco efficiente perchè richiama molte volte una funzione e questo: richiede tempo per la gestione dello stack (allocare e passare i parametri, salvare l'indirizzo di ritorno, e i valori di alcuni registri della CPU) consuma molta memoria (alloca un nuovo stack frame ad ogni chiamata, definendo una nuova ulteriore istanza delle variabili locali non static e dei parametri ogni volta).



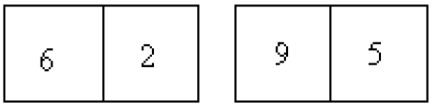
MERGESORT

Il Mergesort e' un algoritmo di ordinamento ricorsivo; Alla base di questo metodo c'e' il procedimento Divide et Impera, che consiste nella divisione di un problema in problemi via via piu' piccoli. Il merge sort opera quindi dividendo l'insieme da ordinare in 2 meta'. Una volta ottenute le meta' si provede alla loro fusione ordinandole. Procedimento:

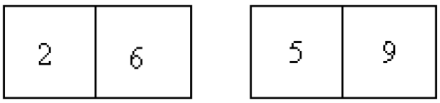
Il vettore viene diviso in due meta', se e' formato da un numero pari di elementi la prima parte ha un elemento in piu'. Si ordinano le parti separatamente e per unirle l'algoritmo estrae il minimo dalle due parti e li fonde in un'unica sequenza. Esempio pratico: Supponiamo di avere un array:



dividiamo l'array in 2 meta' e ripetiamo il processo ricorsivamente fino ad ottenere coppie di 2 elementi:



ordiniamo le coppie ottenute:



e fondiamo in un unico vettore estraendo dagli array ordinati:

L'intero procedimento e' il seguente:

Pseudocodice:
Pseudocodice per l'ordinamento di un vettore di n elementi.
Se ci sono almeno 2 elementi da ordinare:
1. dividi la sequenza in 2 meta';
2. chiamata ricorsiva di mergesort per la prima meta';
3. chiamata ricorsiva di mergesort per la seconda meta';
4. fusione delle due meta'.

CODICE

* File: mergesort.cpp
* Author: Alessandro Lucchetta, Luca Gagliano
*
* Created on 30 gennaio 2015, 14.51
*/

include <iostream>
include <time.h>

```
include <cstdlib>

    using namespace std;

    void mergesort(int[], int, int);
void merge(int[], int, int, int);

    int main()

int n=8;
int a[n];
srand(time(NULL));


for ( int i = 0; i < n; i++)

a[i] = (rand()

for ( int i = 0; i<n; i++)

cout « a[i]«;

;
int i;
//chiamata alla funzione di ordinamento
mergesort(a, 0, n-1);
//visita del vettore
for(i=0; i<n; i++)

cout«a[i]«;

return 0;


    void mergesort(int a[], int left, int right)

//indice dell'enemento mediano
int center;
//se ci sono almeno di 2 elementi nel vettore
if(left<right)

//divido il vettore in 2 parti
center = (left+right)/2;
//chiamo la funzione per la prima meta'
mergesort(a, left, center);
//chiamo la funzione di ordinamento per la seconda meta'
mergesort(a, center+1, right);
//chiamo la funzione per la fusione delle 2 meta' ordinate
merge(a, left, center, right);



    void merge(int a[], int left, int center, int right)
```

```
int i, j, k;
//vettore di appoggio
int b[8];
i = left;
j = center+1;
k = 0;
//fusione delle 2 meta'
while ((i<=center)  (j<=right))

if (a[i] <= a[j])

b[k] = a[i];
i++;

else

b[k] = a[j];
j++;

k++;


//se i e' minore di center significa che alcuni elementi
//della prima meta' non sono stati inseriti nel vettore
while (i<=center)

//allora li aggiungo in coda al vettore
b[k] = a[i];
i++;
k++;


//se j a' minore di right significa che alcuni elementi
//della seconda meta' non sono stati inseriti nel vettore
while (j<=right)

//allora li aggiungo in coda al vettore
b[k] = a[j];
k++;


//alla fine copio il vettore di appoggio b nel vettore a
for (k=left; k<=right; k++)

a[k] = b[k-left];
```

Analisi

Il mergesort gira nel caso peggiore con complessità rispetto al tempo O(n log2 n). I fattori costanti non sono buoni, così non si usa mergesort per piccoli array. Siccome la funzione merge fa copie dei sottoarray, non opera in-place. Mergesort si affida pesantemente alla funzione di merge che ha complessità

lineare rispetto al tempo.
Passiamo ora ai dettagli dell'analisi. Il numero di attivazioni della procedura mergesort dipende dal numero di componenti del vettore da ordinare.
Per una analisi del mergesort è conveniente disegnare un albero per rappresentare le chiamate ricorsive.

Per semplicità, consideriamo un vettore iniziale che abbia n elementi, con n potenza di 2 (n=2 elevato k). [k = log2n]. Ecco una tabella riassuntiva delle attivazioni:

Livello	Attivazioni
(1)	1 attivazione su un vettore di n (=2^k) componenti
(2)	2 attivazioni su 2 vettori di n/2 (=2^{k-1}) componenti
(3)	4 attivazioni su 4 vettori di n/4 (=2^{k-2}) componenti
.....	
(i)	2^{i-1} attivazioni su 2^{i-1} vettori di n/(2^{i-1}) (=2^{k-i+1}) elementi;
.....	
$log_2n+1=k+1$	2^k (=n) attivazioni su 2^k vettori di 1 componente.

Attivazioni di mergesort

1 + 2 + 4 + ... + 2k = 2k+1 - 1 = 2*2*(log2n+1) - 1 = 2 x 2*2*(log2n) - 1 = 2n - 1 = O(n)
Operazioni di confronto
Ad ogni livello h (h >1) si fa il merge di 2h-1 vettori di lunghezza n/2h-1 [si veda la tabella sopra] poichè, detta L la lunghezza dei due vettori da fondere, il numero di confronti del merge è (nel caso peggiore) 2L, il merge di due vettori al livello h richiede 2*(n/2h-1) confronti ma poiche' al livello h ci sono 2h-1 vettori da fondere, il numero totale di confronti a tale livello è fisso e vale 2n.
Pertanto, il numero globale di confronti nei k livelli vale:
time(n) = 2*n*k = O(n*log2n)

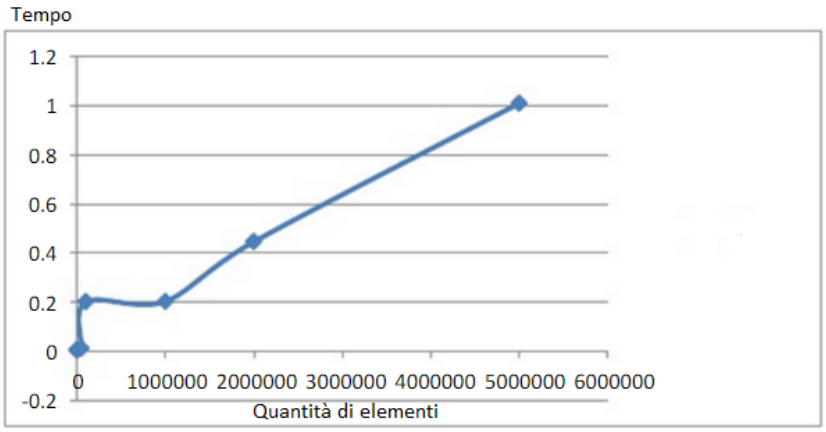
Conclusioni

merge = n + O(n*log2n) = O(n*log2n)
In teoria è il migliore algoritmo di ordinamento (abbiamo assunto però nullo il costo di attivazione di una procedura). Il problema dell'ordinamento di n elementi ha delimitazione inferiore ï(n*log2n).

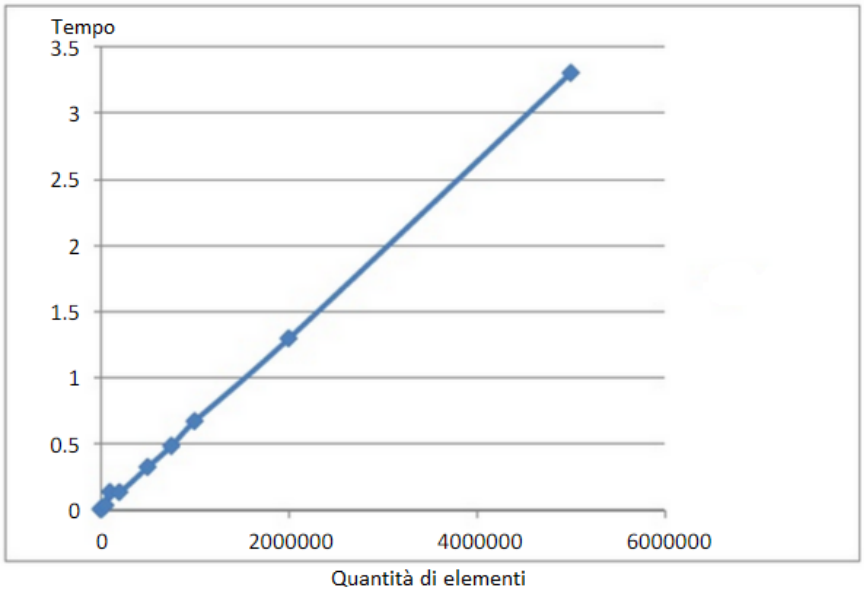
Breve confronto con quicksort

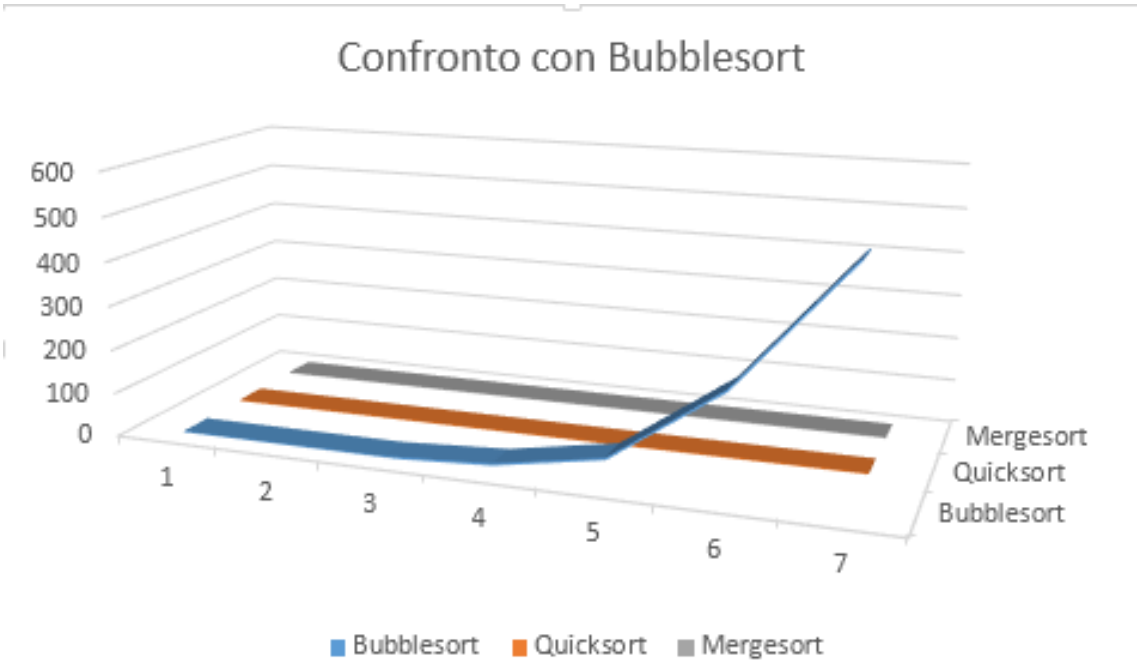
Intuitivamente, il quicksort suddivide a ogni passo il vettore in due sotto-vettori uguali. Come mergesort, anche quicksort suddivide il vettore in due sotto-vettori, delimitati da un elemento limite (pivot). Ma, mentre il mergesort assicura questa proprietà (spezzando il vettore a metà), il quicksort non assicura la riuscita: ottiene lo scopo se il pivot è ben scelto, ma negli altri casi, funzionerà peggio rispetto a mergesort. Queste considerazioni ci inducono a pensare che il quicksort riesca al più a emulare il merge sort, ma senza riuscire a eguagliarlo in tutti i casi.

QUICK SORT						
N	T_{avg} (Seconds)	T_1 (Seconds)	T_2 (Seconds)	T_3 (Seconds)	T_4 (Seconds)	T_5 (Seconds)
1000	0.0050003	0.0040003	0.0050003	0.0050003	0.0070004	0.0040002
2000	0.00560034	0.0040003	0.0050003	0.0070004	0.0050003	0.0070004
5000	0.0056003	0.0050003	0.0050003	0.0050003	0.0050003	0.0080003
10000	0.0072004	0.0070004	0.0090005	0.0090005	0.0060003	0.0050003
50000	0.01240068	0.0100005	0.0140008	0.0130007	0.0130007	0.0120007
100000	0.20201156	0.1770102	0.2070118	0.2060118	0.2160124	0.2040116
1000000	0.20281158	0.1910109	0.2130121	0.1990113	0.1970113	0.2140123
2000000	0.4458255	0.3910224	0.4270245	0.4530259	0.4840276	0.4740271
5000000	1.00885768	0.9810561	1.0190583	1.0050573	1.0010573	1.0380594



MERGE SORT						
N	T_{avg} (Seconds)	T_1 (Seconds)	T_2 (Seconds)	T_3 (Seconds)	T_4 (Seconds)	T_5 (Seconds)
1000	0.00560032	0.0050003	0.0060003	0.0070004	0.0050003	0.0050003
2000	0.00740038	0.0060003	0.0080004	0.0070004	0.0060003	0.0100005
10000	0.01160062	0.0110006	0.0120006	0.0090005	0.0130007	0.0130007
50000	0.03640208	0.0400023	0.035002	0.0410023	0.0330019	0.0330019
100000	0.13700784	0.0580033	0.0850048	0.0730042	0.0650038	0.4040231
200000	0.13520778	0.1320075	0.1280075	0.1290074	0.1460084	0.1410081
500000	0.3250186	0.2950169	0.3440197	0.3180182	0.3160181	0.3520201
750000	0.48342766	0.4470256	0.506029	0.472027	0.4700269	0.5220298
1000000	0.67183842	0.6080347	0.7290417	0.6850393	0.6740385	0.6630379
2000000	1.29567414	1.2650724	1.3530774	1.3010744	1.2910739	1.2680726
5000000	3.30358896	3.3861937	3.2751874	3.2781875	3.2821877	3.2961885




Confronto con Bubblesort
600
500
400
300
200
100
0
1
2
3
4
5
6
7
Mergesort
Quicksort
Bubblesort
Bubblesort
Quicksort
Mergesort