Andrea Peretti mat. 718024

Relazione

Confronto dei tempi di esecuzione di algoritmi di ordinamento

Algoritmi esaminati

- quicksort – versione del libro di testo
- quicksort – versione alla Hoare
- quicksort – una ulteriore versione
- quicksort – versione ottimizzata con insertionsort al di sotto di una soglia
- mergesort – versione ottimizzata con insertionsort al di sotto di una soglia
- mergesort – versione ottimizzata senza chiamate inutli del merge
- mergesort – versione a passo alternato

Descrizione delle modalità di test

Sono stati scritte le procedure degli algoritmi richiesti in linguaggio C e Java,e si ha effettuato il test per la ricerca della soglia per gli algoritmi quicksort e mergesort. Successivamente sono state testate su array di interi generati casualmente di dimensione sempre maggiore e sempre uguali per ogni test tutti gli algoritmi elencati. Per ogni esecuzione è stato misurato il tempo impiegato. I risultati ottenuti dalle misurazioni verranno utilizzati per confrontare l'efficienza di ogni algoritmo in pratica con la sua efficienza teorica. Tutti gli algoritmi ordinano in ordine crescente. I calcoli, le tabelle ed i grafici sono contenuti nel foglio elettronico presente insieme a questo documento.

I test sono stati effettuati su di un Mac Book Pro con Intel Core 2 Duo 2,4 Ghz, 8GB ram DDR3, SO Mac OS X 10.8.2 Mountain Lion, java versione 7. I programmi sono stati avviati da terminale e non dall'IDE e nel sistema non erano aperti altri programmi, inoltre il sistema non è stato usato per tutta la durata dei test.

Ricerca della soglia

La ricerca della soglia è un test che serve a determinare a quale dimensione *i* di un segmento di array gli algoritmi di ordinamento con insertionsort su quel segmento sono più performanti degli stessi senza l'insertionsort. A seguito dei test si sono individuate come soglie il valore di 64 per il mergesort e 69 per il quicksort. Questo significa che, ad esempio per il quicksort, quando l'algoritmo dovrà ordinare una porzione di elemento <= a 69, la funzione richiamerà l'inseritonsort su quel segmento.
Il risultato trovato è poi stato utilizzato per i test successivi.

Andrea Peretti mat. 718024

Grafico della ricerca della soglia nel quicksort

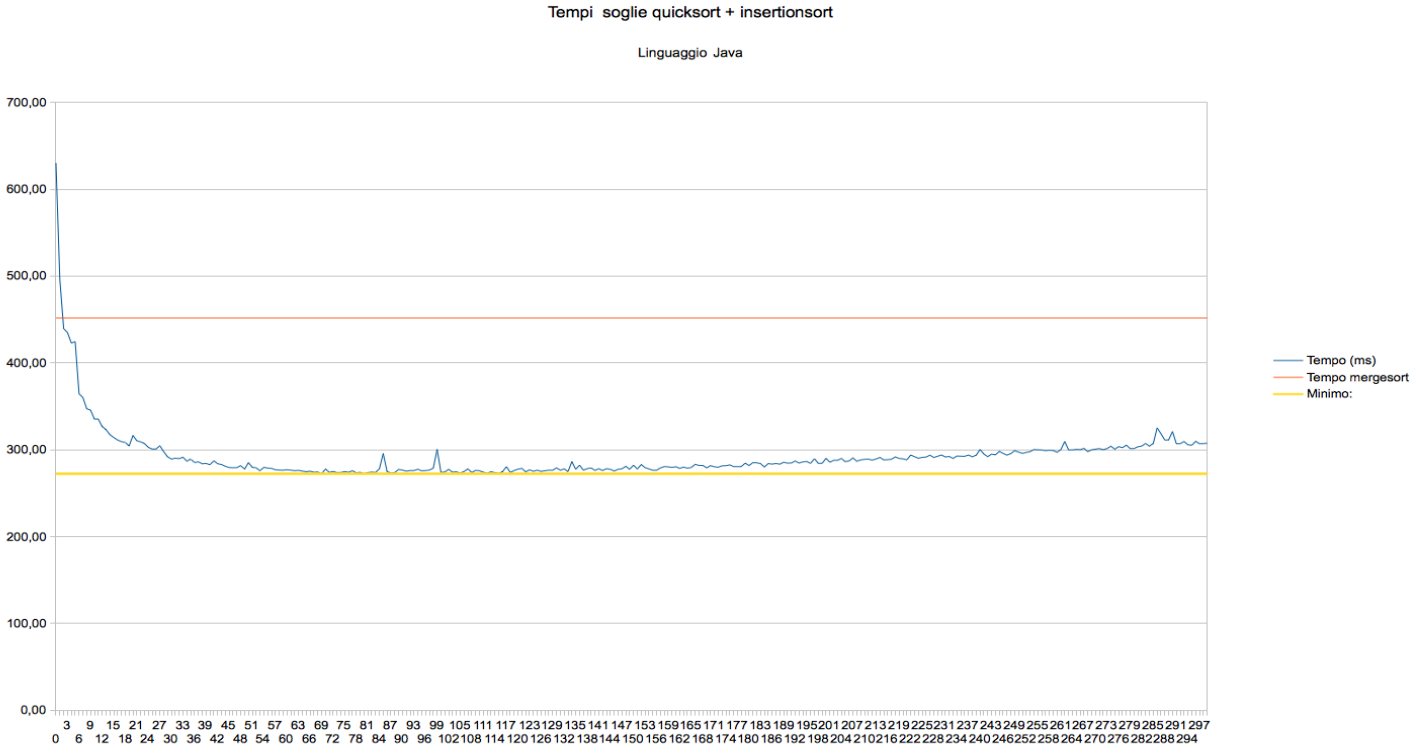
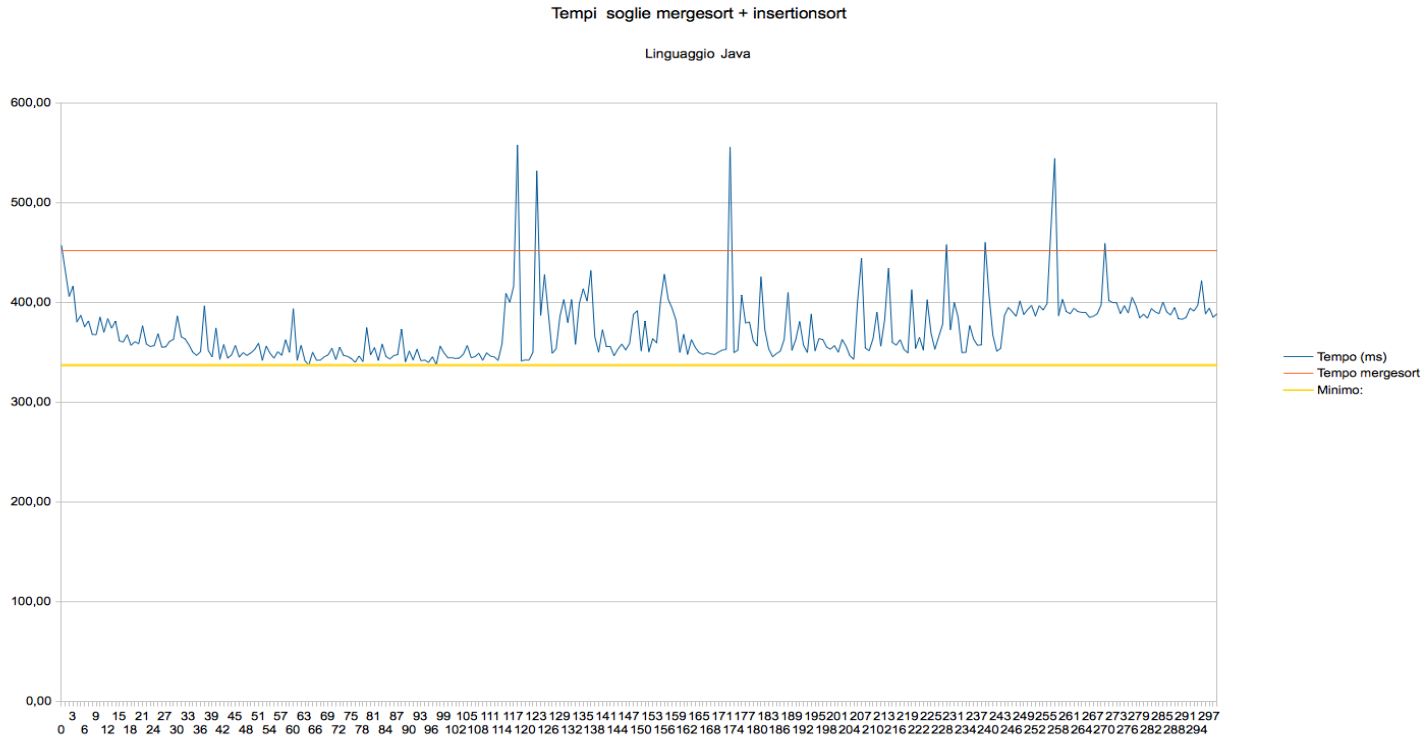


Grafico della ricerca della soglia nel mergesort

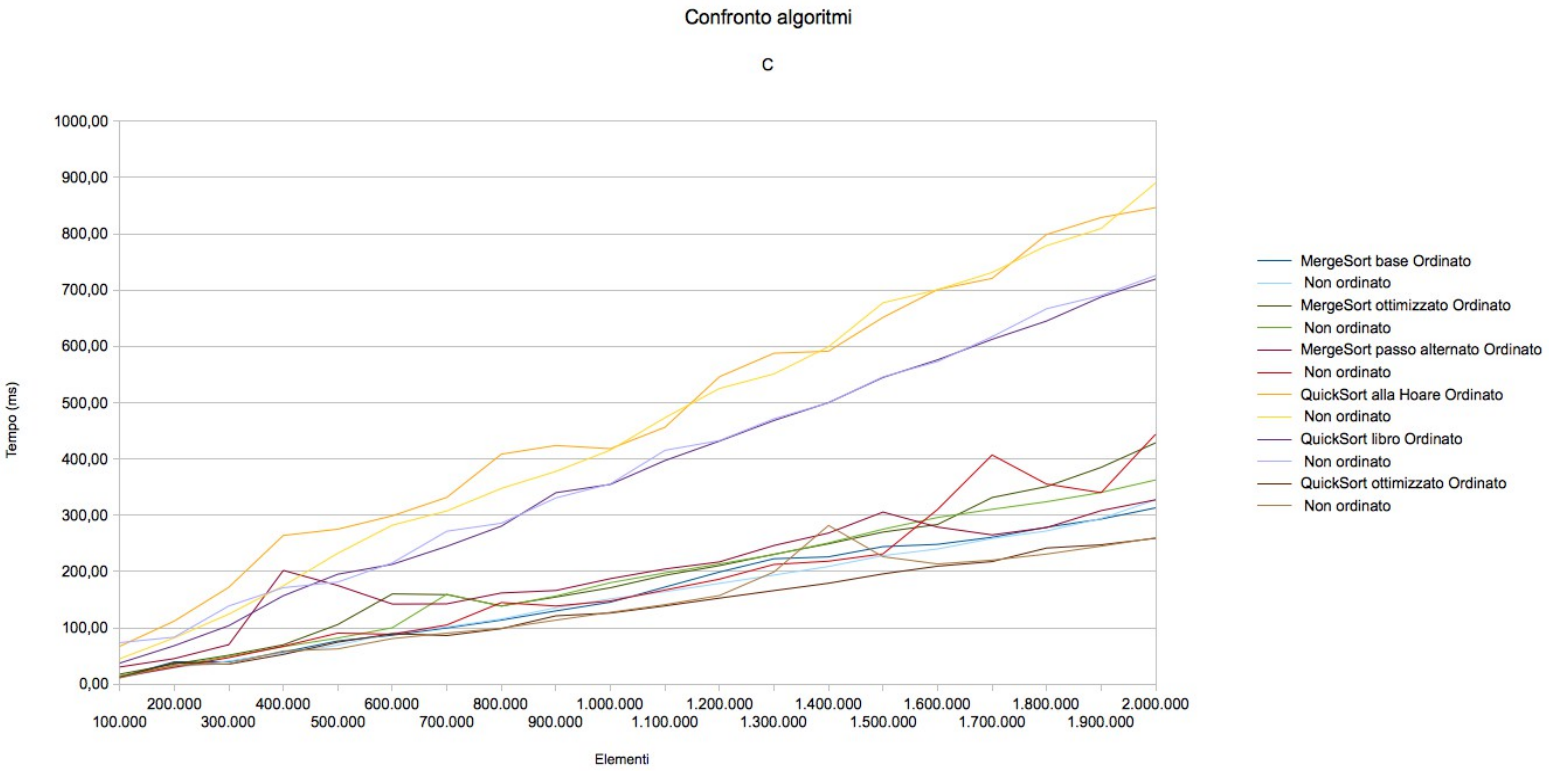


Andrea Peretti mat. 718024

Analisi degli algoritmi

Il confronto degli algoritmi è stato fatto con un programma in linguaggio C. L'algoritmo crea inizialmente un array della dimensione del massimo degli elementi, e per ogni misurazione viene duplicato, in questo modo i test vengono effettuati sempre sullo stesso array. Anche l'array contenente gli elementi quasi ordinati (nei grafici indicato con “ordinato”) è sempre uguale: si crea un array delle stesse dimensioni del primo e si scambiano due elementi a caso tra i primi elementi. Il test viene poi eseguito per ogni algoritmo, facendo partire il timer appena prima della chiamata a funzione e facendolo terminare appena dopo il ritorno.

Grafico con gli algoritmi a confronto. I colori più chiari indicano i tempi degli algoritmi su array quasi ordinati, mentre quelli più scuri di array disordinati.



Mergesort base

La versione base del mergesort è quella ottimizzata con l' insertionsort, il quale algoritmo viene utilizzato quando la parte di array che si deve ordinare è minore di una certa soglia, definita nel paragrafo *Ricerca della soglia*.

La complessità di questo algoritmo è data da quella dell'operazione di merge di due porzioni di array ordinati e dal numero di volte che viene ripetuta.

Nella funzione di merge non vi è distinzione tra caso migliore, medio o peggiore, la complessità temporale è sempre:

$T(n) = \Theta(n)$

dove n è la dimensione dell' array.

Per la complessità spaziale, l'algoritmo così definito ha bisogno di un array ausiliario per copiarci gli elementi nella fase di merge, quindi:

$S(n) = \Theta(n)$

Considerando l'albero delle chiamate ricorsive. Poichè ad ogni livello si dimezza la dimensione della porzione di array, avremo in tutto log n livelli.

Il numero totale di passi è quindi:

$T(n) = \Theta(n \log n)$

Il numero massimo di record di attivazione presenti contemporaneamente è log n, mentre la merge richiede spazio n al livello 0, n/2 al livello 1 ecc... Quindi la

complessità è n + log n, quindi
$S(n) = \Theta(n)$.

Da quello che si può notare nel grafico, l'algoritmo (rappresentato dalle linee blu ed azzurra) offre una buona efficienza. La crescita riflette le previsioni teoriche, anche se si avvicina molto ai risultati di un algoritmo lineare.

Mergesort ottimizzato

La differenza tra il mergesort base ed il mergesort ottimizzato è che quest'ultimo evita di richiamare la funzione di merge. Inoltre non utilizza l'ottimizzazione con l'insertionsort. Con questa accortezza i tre casi non sono più indifferenziati: esiste infatti un caso migliore, in cui l'array è già ordinato e la chiamata al merge non viene mai fatta. La complessità per il caso migliore è quindi:
$T_{best} = \Theta(n)$
Nel grafico l'algoritmo è rappresentato con le linee verde chiaro e scuro. In questo caso, anche se teoricamente le linea verde scuro sarebbe dovuta crescere meno velocemente di quella verde chiaro (che rappresenta l'algoritmo su un array non ordinato) non segue la previsione teorica.

Mergesort ottimizzato a passo alternato

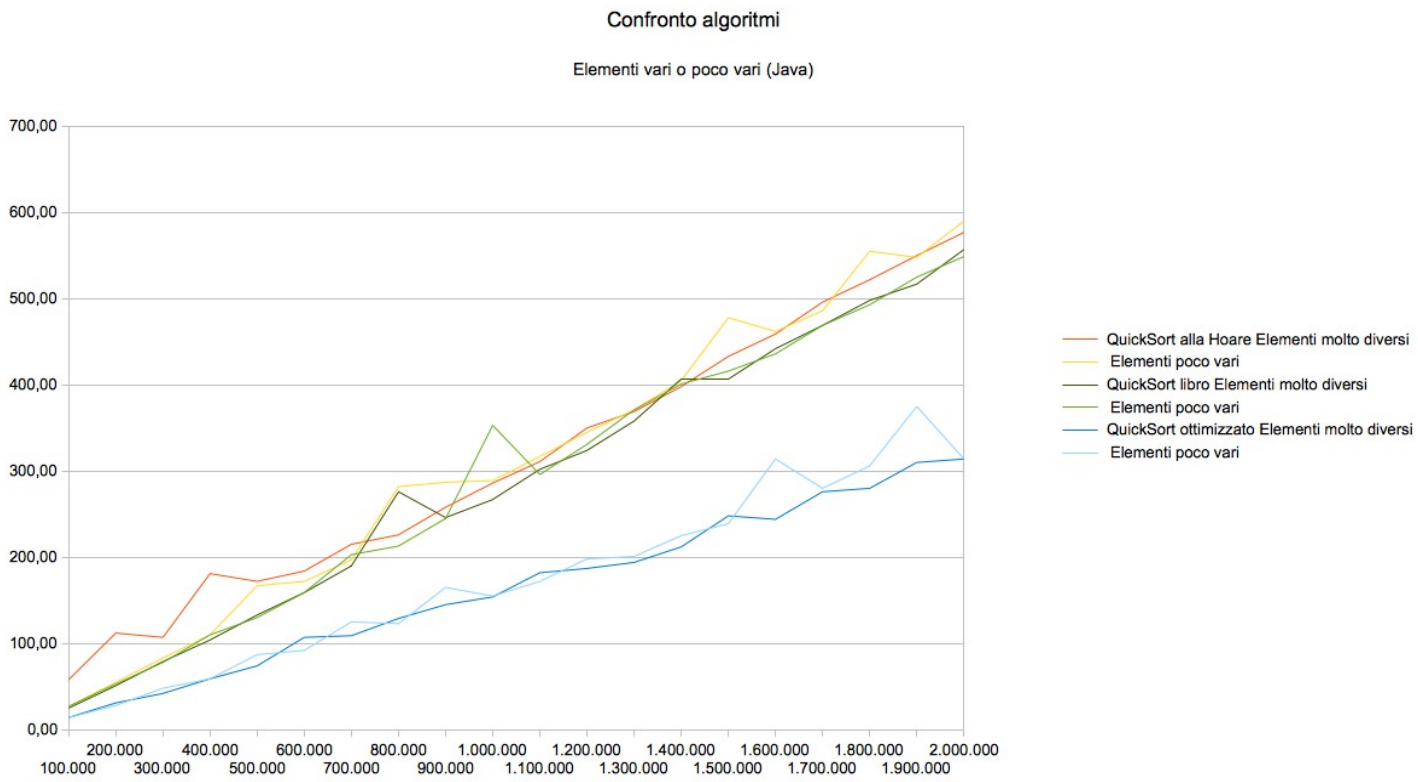
In questa versione dell' algoritmo si utilizza una versione modificata della merge in cui si elimina la ricopiatura finale utilizzando un array ausiliario. Questo velocizza la fase di merge. L'ottimizzazione del mergesort ottimizzato però, se utilizzata, comporta la copia del segmento di array all'altro. Questa operazione è lineare e meno costosa del merge, ma non cambia l'equazione di ricorrenza e non si ha il caso migliore.
Nel grafico si può notare che questa versione produce risultati simili a quelli del mergesort ottimizzato, anche se più vari.

In generale: Quicksort

Il quicksort opera sul posto, cioè senza bisogno di array aggiuntivi di lunghezza n, inoltre l'algoritmo non è stabile.
Il quicksort effettua una partizione dell'array per posizionare gli elementi minori prima del pivot e quelli maggiori dopo. Questa operazione ha un costo lineare $\Theta(n)$, e viene eseguita log n volte, dove n è la dimensione dell'array, quindi la complessità è
$T_{best} = \Theta(n \log n)$
mentre nel caso peggiore, ossia quando l'algoritmo sceglie sempre il valore minimo (o massimo) come pivot in un array già ordinato, formando una partizione di n-1 elementi e l'altra di 0:
$T_{worst} = \Theta(n^2)$
Essendo un algoritmo ricorsivo, ha bisogno di spazio per lo stack relativi alle chiamate a funzione. L'altezza minima dell'albero delle chiamate è log n, quindi
$S_{best} = \Theta(\log n)$
Ma, nello stesso caso peggiore che causa il Tworst, l'altezza dell'albero delle chiamate ricorsive è proporzionale a n:
$S_{worst} = \Theta(n)$
Per il caso medio:
$T_{medio} = \Theta(n \log n)$
$S_{medio} = \Theta(\log n)$
Nei test il pivot è sempre stato scelto in modo casuale, e nessuno implementa l'ottimizzazione dell'eliminazione della ricorsione di coda.

Grafico che mostra l'esecuzione dell'algoritmo su array con elementi molto simili e su array con elementi molto diversi

Andrea Peretti mat. 718024



Quicksort versione del libro di testo

In questa versione, gli elementi uguali al pivot possono stare sia a sinistra che a destra di esso. L'algoritmo scorre la porzione di array da entrambi i lati verso il centro finchè non trova due elemento dal lato sbagliato, e ne effettua lo scambio per poi continuare finche tutta la porzione è stata esaminata. Rispetto agli altri algoritmi, è poco più performante della versione alla Hoare ma molto meno di quella ottimizzata. Le differenze dei tempi di esecuzione tra erray con elementi ordinati e disordinati sono minime. Questo algoritmo ha un caso peggiore che si verifica quando tutti gli elementi sono uguali:
$T_{worst} = (n^2)$
Nel grafico si può notare un picco nel caso con molti elementi simili nella prova nell'array con 1 milione di elementi.

Quicksort Hoare

Questa versione è la meno performante. Secondo il grafico, l'esecuzione su di un array quasi ordinato è mediamente meno performante dell'esecuzione su un array non ordinato. L'esecuzione su elementi molto simili provoca spesso dei picchi, riflettendo il caso peggiore teorico del quicksort.

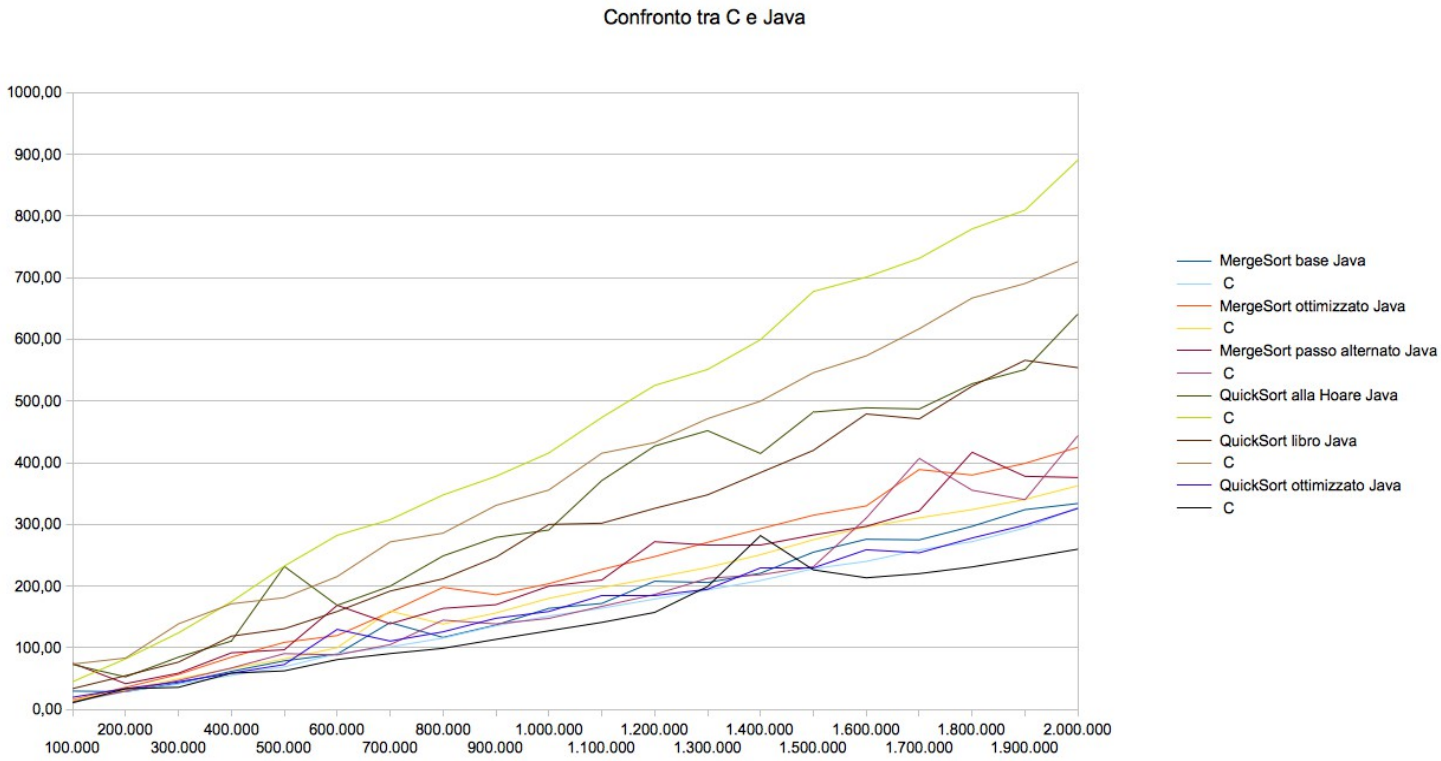
Quicksort versione ottimizzata

Questa versione è uno degli algoritmi più performanti testati, essa implementa il quicksort Hoare ma quando l'array analizzato è minore di una certa soglia si ordina con insertionsort. L'algoritmo presenta un picco nell'esecuzione sull'array da 1,4 milioni di elementi quasi ordinato, probabilmente incappando nel caso peggiore. Lo stesso accade con alcune esecuzioni su array con elementi poco vari.

Confronto tra C e Java

Il grafico mostra i tempi misurati nell'esecuzione degli algoritmi nel linguaggio java e c

Andrea Peretti mat. 718024



Dal grafico si può notare un comportamento strano, infatti nel caso del mergesort il C è più veloce di Java, mentre nel caso del quicksort Java è parecchio più performante, al contrario dei test generali in cui Java è meno performante di C.

Previsione dell'esecuzione di un algoritmo quadratico

Stando ai risultati ottenuti dal compito 4, l'insertion sort è l'algoritmo quadratico più performante tra quelli testati. I risultati ottenuti furono:

5000	11,74
10000	25,15
15000	57,44
20000	103,69
25000	206,15
30000	447,30
35000	293,52
40000	386,57
45000	491,96
50000	604,46
55000	729,54
60000	875,22
65000	1029,53
70000	1230,80
75000	1397,77
80000	1549,84
85000	1748,47
90000	2013,18
95000	2176,00
100000	2407,42

Andrea Peretti mat. 718024

La misurazione per 100000 elementi è $T(100000)=2407,42$, dunque:

$T(2000000)=T(20*100000)=20^2*T(100000)=400 * 2407,42=962968$ ms

Per ordinare un array da 2 milioni di elementi dovrebbe impiegare circa 16 minuti.

Il tempo misurato è di 939474, circa 15 minuti.

Analisi della versione leggermente errata del quicksort Hoare

Questa versione presenta il seguente ciclo:

```
while (true) {
                        while (a[i] < x)
                              i++;
                        while (a[j] > x)
                              j--;
                        if (i <= j) {
                              scambia(a, i, j);
                              i++;
                              j--;
                        } else
                              break;
                  }
```

Nel codice la variabile *x* contiene il pivot, *i* è inizializzata ad *inf* e *j* a *sup.*
In alcuni casi può capitare che gli indici escano fuori dal range di inf...sup:
Se 5 è il pivot e l'array è
5, 6, 9, 10
i si fermerà all'indice 0, *j* pure. L'istruzione condizionale *if(i<=j)* è verificata, quindi si procederà allo scambio (che è inutile) e poi di incrementerà *i* e si decrementerà *j* che varrà -1, senza però terminare il ciclo! Alla sucessiva esecuzione il programma genererà un eccezione di *IndexOutOfBounds*.