Andrea Peretti mat. 718024

Relazione

Confronto dei tempi di esecuzione di algoritmi di ordinamento

Algoritmi esaminati

- selection sort
- selection sort max, simile al selection sort ma ordinando l'array dal valore più alto
- insertion sort
- merge sort

Descrizione delle modalità di test

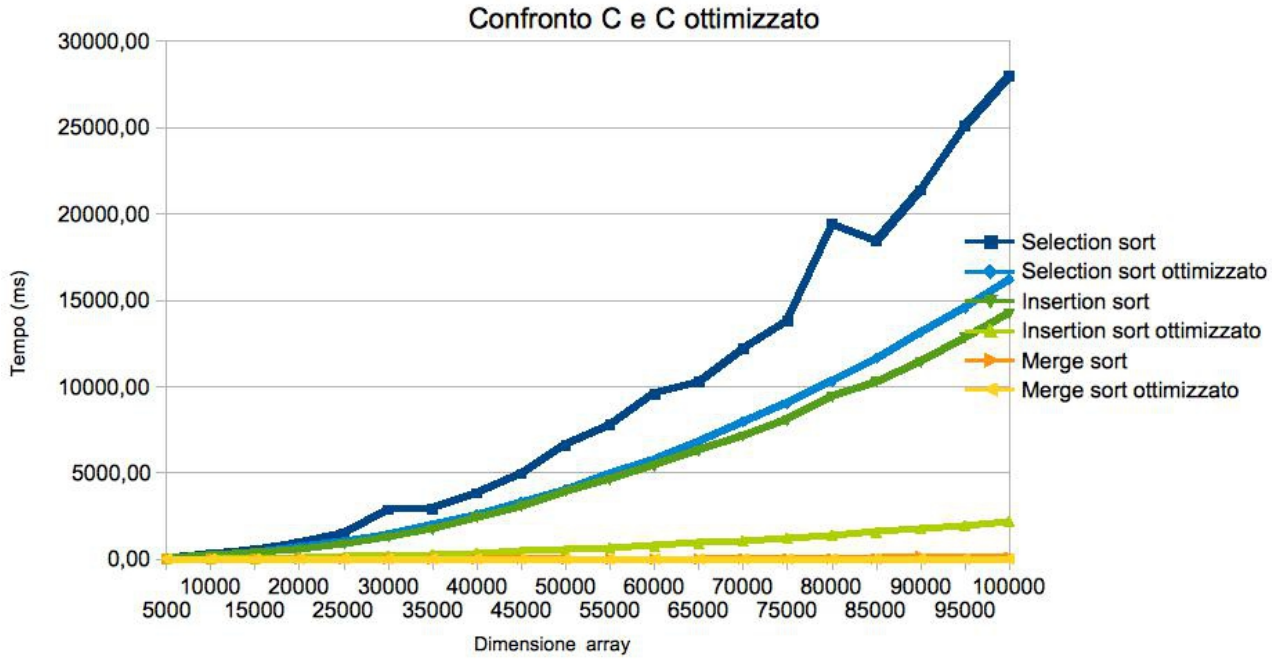
Sono stati scritte le procedure degli algoritmi richiesti in linguaggio C e Java, successivamente sono state testate su array di interi generati casualmente di dimensione sempre maggiore ma diversi per ogni test. Per ogni esecuzione è stato misurato il tempo impiegato. I risultati ottenuti dalle misurazioni 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.

Confronto delle prestazioni di C e Java

Prima di confrontare C e java, apriamo una parentesi sulle prestazioni del C. Il compilatore accetta il parametro -On dove n è il grado di ottimizzazione del codice che può andare da 1 a 3. Nel grafico si può vedere la differenza tra una compilazione con l'ottimizzazione standard ed una con l'ottimizzazione di livello 3:

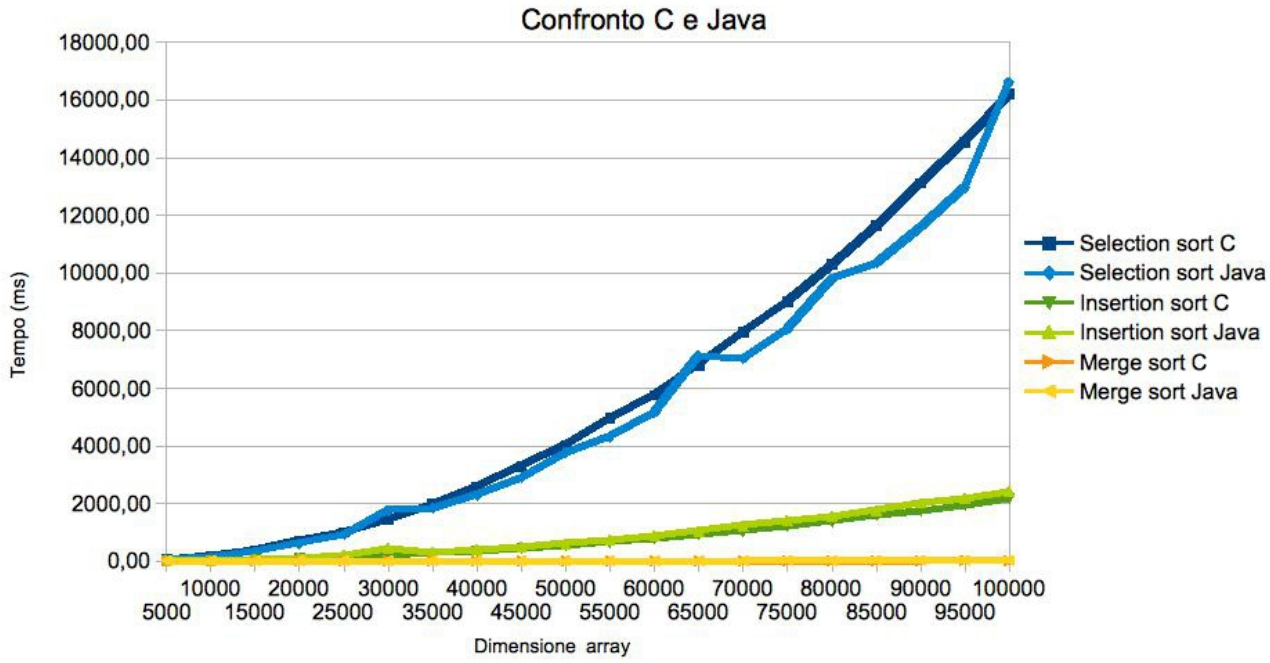
Grafico di confronto C compilazione normale ed ottimizzata



Andrea Peretti mat. 718024

Viste le prestazioni maggiori del codice ottimizzato, per i test è stata utilizzata l'ottimizzazione -O3.

Grafico di confronto tra le prestazioni di C e Java per i vari algoritmi esaminati



Nel grafico le linee di colore più chiaro rappresentano il linguaggio C mentre quelle più scure Java. Facendo la media dei tempi per ogni algoritmo, e poi il rapporto della media di un algoritmo in C e quella di quello in Java, otteniamo che:

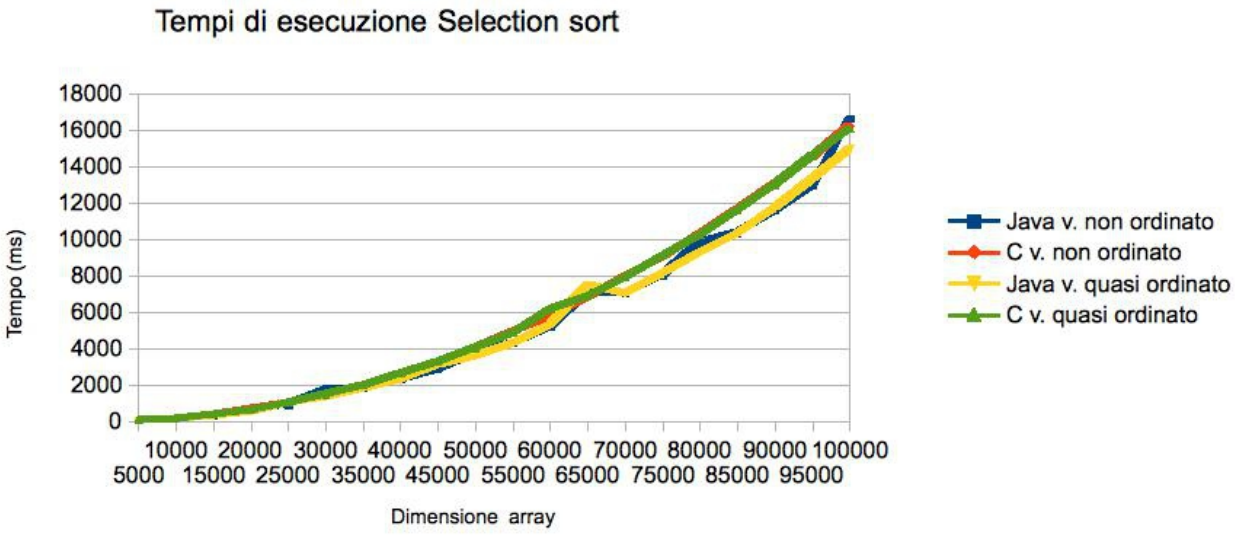
- il Selection sort C è più lento di Java per il 7,9%
- l'Insertion sort C è più veloce del 12,8%
- il Mergesort C è più veloce del 42%.

Queste differenze possono essere causate per i seguenti motivi:
Il grafico Java del selection sort presenta dei punti che non seguono l'andamento del grafico, i quali possono alterare la media, ma è anche possibile che in alcuni casi Java sia più performante di C. I valori registrati per il mergesort invece sono molto piccoli, questo potrebbe risultare in dati non troppo affidabili. Per l'insertion sort invece otteniamo un valore vicino a quello che altri benchmark hanno misurato: il Java è mediamente meno performante del C di circa il 10%, questo perchè Java è un linguaggio orientato agli oggetti, è semi interpretato siccome viene eseguito il bytecode da una macchina virtuale, ed inoltre la memoria è gestita dal Garbage Collector e non dall'utente come capita per il c, andando ad influire sul tempo di esecuzione con un ulteriori procedure per la pulizia della memoria.

Analisi degli algoritmi

Selection sort

Andrea Peretti mat. 718024

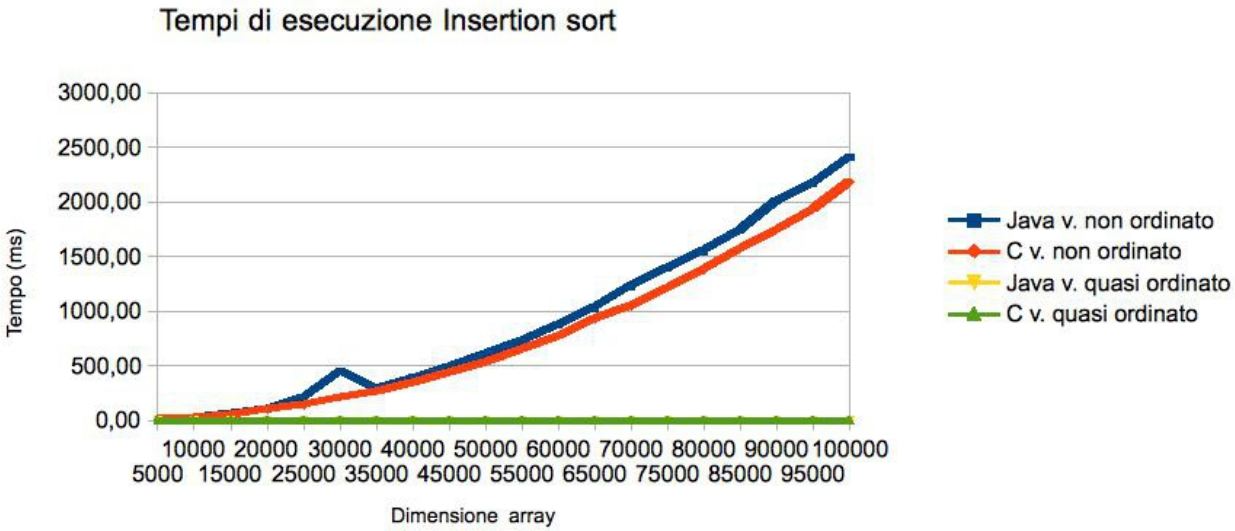


Complessità temporale:
Il selection sort è un algoritmo quadratico, infatti come visibile anche dal grafico presenta una curva che aumenta esponenzialmente. L'algoritmo non presenta casi migliori o casi peggiori, ed il tempo di esecuzione su di un array quasi ordinato è simile a quello su di uno non ordinato:

Con 100.000 elementi su un array non ordinato: 16204,27 ms
Con 100.000 elementi su un array quasi ordinato: 16042,73 ms

Complessità spaziale:
L'algoritmo è ricorsivo, quindi in memoria sarà presente l'array ed altre poche variabili.
(1).

Insertion sort



Complessità temporale:
L' Insertion sort presenta risultati diversi in base allo stato dell'array su cui deve lavorare, infatti può essere individuato un caso migliore:
Tbest(n) = (n) in cui l'andamento è lineare, l'array è già ordinato o quasi ordinato

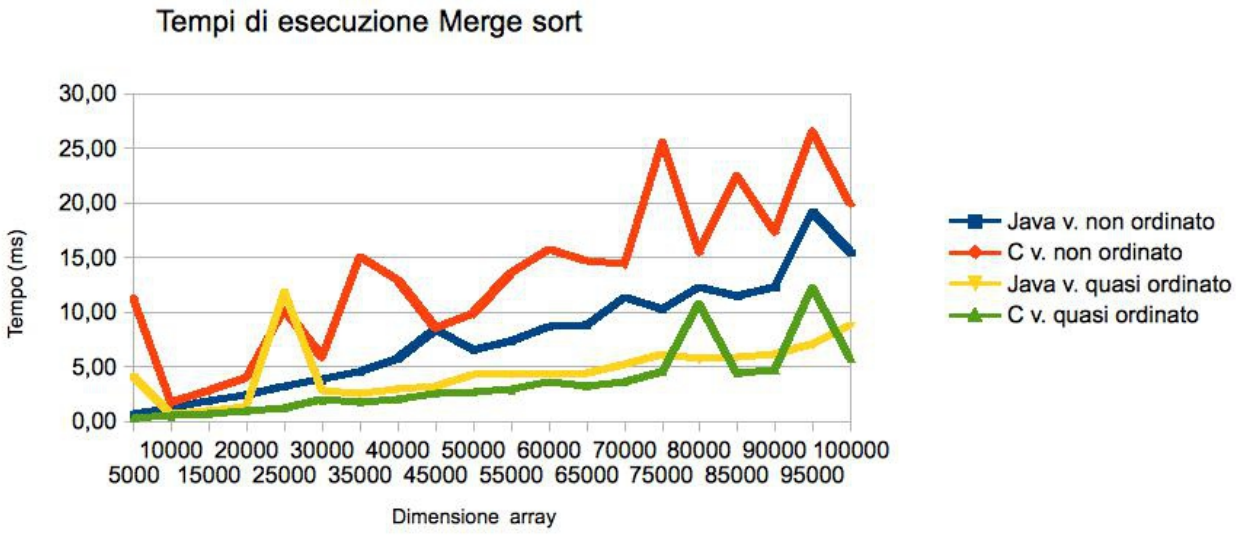
Tmedio(n)= (n^2) in cui l'andamento è quadratico e

Tworst(n) = (n^2) in cui l'array è ordinato in ordine decrescente, può essere identificato come il caso peggiore, dove l'andamento è sempre quadratico.

I risultati ottenuti confermano quanto detto, infatti in un array non ordinato la curva è esponenziale, per 100.000 elementi abbiamo 2177,76 ms, mentre in quello quasi ordinato la curva non presenta più l'incremento esponenziale, per 100.000 elementi solo 0,18ms, un differenza molto evidente.

Complessità spaziale:
L'algortimo è iterativo e non utilizza array ausiliari: (1).

Merge sort



Dettaglio mergesort

L'algoritmo di mergesort implementato è la versione ottimizzata nel merge, ma non a passo alternato.
Complessità temporale:
In questo algoritmo non c'è distinzione tra i vari casi, e la complessità temporale è data dalla complessità della funzione di merge (n) più quella della suddivisione dell'array in sotto-array $(\log n)$: $(n \log n)$.
Da come si può vedere nel grafico l'andamento del mergesort non è ben definito come gli altri esaminati, soprattutto nella versione Java per l'array non ordinato e per C per l'array quasi ordinato. Si può comunque individuare un andamento lineare, ma $(n \log n)$ è molto simile ad un andamento lineare. A differenza della teoria però, un array quasi ordinato permette un esecuzione più veloce di uno non ordinato. È' possibile che si verifichi questo a causa del tempo necessario nella funzione merge a ricopiare nell'array originale i valori ordinati dall'array ausiliario, dimostrando quindi l'importanza dell'ottimizzazione “a passo alternato”.

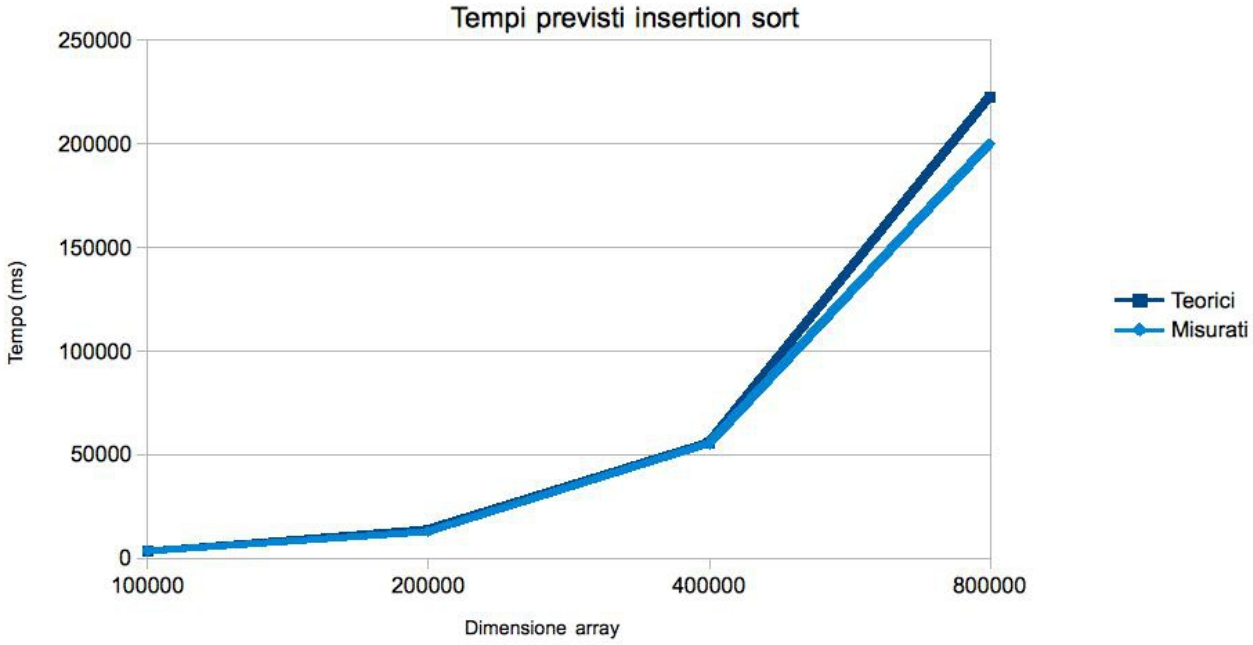
Complessità spaziale:
Essendo una funzione ricorsiva, la memoria necessaria aumenta all'aumentare degli elementi dell'array, ed è pari alla profondità dell'albero di ricorsione: $(\log n)$. L'algoritmo fa inoltre uso di un array ausiliario, quindi si aggiunge (n), con:
S(n) = (n) + $(n \log n)$ = (n).

Previsione dei tempi di esecuzione:

Andrea Peretti mat. 718024

In questo paragrafo si prenderà in considerazione l'Insertion sort.
Dai dati ottenuti effettuando un nuovo test su array più grandi, per l'algoritmo di Insertion sort il tempo necessario ad ordinare 100.000 elementi in java su di un array non ordinato è di 3479,07 ms. Come spiegato al corso, ogni volta che la dimensione raddoppia il tempo impiegato quadruplica, quindi per 200.000 elementi dovremmo avere un tempo di 13916,28 ms, e così via.
Da nuovi test sono stati individuati questi risultati:

Confronto tra incremento teorico e pratico del tempo di esecuzione dell'insertion sort



I risultati ovviamente non sono esattamente quelli previsti, ma è palese che sono molto simili.

Confronto tra gli algoritmi esaminati

Dei tre algoritmi esaminati è possibile individuare pro e contro:

Selection sort:
pro:
- semplice

contro:
- *Estremamente inefficiente*

Insertion sort:
pro:
- *caso migliore lineare*

contro:
- *caso peggiore e medio quadratico*

Merge sort:
pro:
- lineare

contro:
- Ricorsivo, più memoria utilizzata

Andrea Peretti mat. 718024

Dai punti precedenti si può arrivare alla conclusione che il selection sort può essere comodo perchè facile da implementare, ma per l'inefficienza non vale la pena il suo utilizzo. L'insertion sort se la cava meglio del selection sort in ogni caso, ma il divario tra il caso migliore e gli altri è molto grande, per questo può essere utile se per elenchi di elementi quasi ordinati. Il merge sort è quello più performante tra quelli esaminati, ma essendo ricorsivo utilizza più memoria di quello che utilizzarebbero gli altri due.