Tesina di
Algoritmi e Strutture Dati

Studio sugli algoritmi di ordinamento

Cristiano Salerno
Matricola M63000848

Indice

Sommario

In questa tesina studieremo gli algoritmi di ordinamento visti durante il corso; ci concentreremo solamente su tre algoritmi di ordinamento sugli array, ovvero Insertion Sort, Merge Sort e per concludere Heap Sort.

Introdurremo un algoritmo alla volta, discutendo della sua complessità asintotica e illustrando - tramite grafici realizzati in Matlab - il tempo necessario ad ordinare gli array al crescere delle dimensioni degli stessi.

Abbiamo fatto variare il numero degli elementi da 10^4 a 10^6 su 100 array (per ogni algoritmo di ordinamento), quindi con un passo di 10^4 elementi fra un array e l'altro (riferimento alla generazione qui).

È stato inoltre eliminato completamente il rumore dovuto allo scheduling grazie alla classe `java.lang.management.ManagementFactory`, con la quale viene messo a disposizione il metodo `getCurrentThreadCpuTime()`, in grado di restituire - in nanosecondi - il tempo che la CPU ha processato il nostro thread (riferimento qui).

Grazie a questo metodo, abbiamo potuto ottenere il vero tempo di esecuzione di ogni thread (e quindi il vero tempo di esecuzione dell'algoritmo sull'array di dimensioni n), senza fare semplicemente una differenza di timestamp; essa, infatti, non tiene conto né dello scheduling, né di altri fattori come la priorità dei thread assegnata dal SO: basta infatti chiudere a icona l'ambiente Eclipse, o aprire altri processi, per vedere il tempo di esecuzione cambiare notevolmente anche per algoritmi normalmente molto efficienti.

Il processore adoperato per il benchmarking degli algoritmi è un dual-core (quad-core virtuale) clockato a frequenza 2.5 GHz, ma tutti gli algoritmi sono sviluppati con codice sequenziale (mentre l'esecuzione dello stesso algoritmo su più array diversi avviene in maniera parallela), quindi la scelta del numero di core è ininfluente.

Capitolo 1

Insertion Sort

1.1 Introduzione

Iniziamo la trattazione parlando dell'insertion sort: questo è uno degli algoritmi di sorting più lenti, poichè (partendo dal secondo elemento dell'array) confronta ogni elemento i-esimo con i suoi precedenti da 0 a $i-1$ e, se ne trova uno maggiore, lo scambia con esso.

Intuitivamente è molto semplice, ma come spesso accade, più è semplice l'implementazione e più il tempo di esecuzione risulta elevato, poiché l'algoritmo non è efficiente. Come vantaggio, però, abbiamo l'ordinamento sul posto, cioé la necessità di un buffer costante per ordinare (in questo caso un buffer di dimensione uno, cioé una variabile).

1.2 Tempo di esecuzione

Come abbiamo studiato, l'Insertion Sort ha un tempo di esecuzione $T(n) = \Omega(n) = O(n^2)$, quindi compreso fra un andamento lineare e uno quadratico.

Come possiamo effettivamente notare dal grafico 1.1, notiamo chiaramente che il tempo di esecuzione di Insertion Sort cresce quasi quadraticamente con l'aumentare del numero di elementi da ordinare. Per plottare $O(n^2)$ abbiamo usato la funzione $f(n) = an^2$, con $a = 10^{-7}$.

Come possiamo notare, aumentando il numero degli elementi da ordinare (verso i $3 * 10^5$ elementi), aumenta di molto il tempo di esecuzione rispetto all'andamento di $f(n)$: cambiando il fattore moltiplicativo a, ed eventualmente aggiungendo un termine lineare, possiamo tener conto di questi spike temporali e rimanere al di sopra della $T(n)$ misurata.

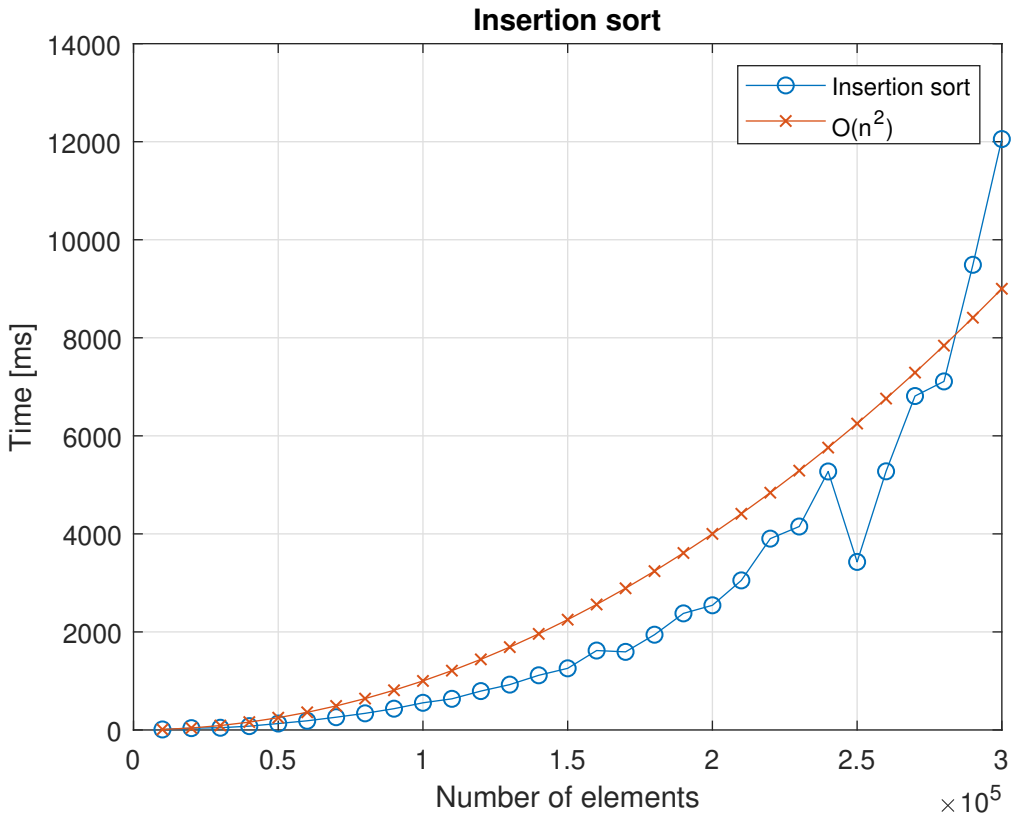

Figura 1.1: Andamento del tempo di esecuzione di Insertion Sort all'aumentare della dimensione dell'array da ordinare.

Capitolo 2

Merge Sort

2.1 Introduzione

Introduciamo il secondo algoritmo studiato durante il corso. Merge Sort è un algoritmo completamente diverso da Insertion Sort, in quanto è ricorsivo e non ordina sul posto (cioé ha bisogno di una quantità di memoria direttamente proporzionale alla lunghezza n dell'array da ordinare).

Il suo funzionamento (così come quello di tutti gli algoritmi ricorsivi) è basato sul divide et impera, cioé scomporre un macroproblema in tanti sottoproblemi di più facile risoluzione, fino ad arrivare ad un caso base banalmente risolubile, per poi tornare indietro sfruttando il caso base raggiunto.

Merge Sort funziona in due fasi diverse: la prima fase consiste nel dividere a metà l'array da ordinare, mentre nella seconda fase è presente la fusione ordinata delle due metà di array.

Bisogna dividere ricorsivamente l'array fin quando non arriviamo al caso base (array di un solo elemento), per poi risalire la catena fondendo in maniera ordinata questi mini-array ottenuti, fino ad ottenere di nuovo l'array originale ma ordinato; per fare ciò, quindi, abbiamo bisogno di copie di questi mini-array (quindi la memoria richiesta è proporzionale alla lunghezza dell'array stesso).

2.2 Tempo di esecuzione

Il Merge Sort ha un tempo di esecuzione $T(n) = \Theta(n \log n)$. È peggiore del caso migliore dell'Insertion Sort, ma è un algoritmo migliore, - per quanto riguarda il tempo di esecuzione, ma non lo spazio occupato - in quanto il suo limite asintotico superiore è inferiore a quello di Insertion Sort, che è quadratico.

Come possiamo vedere nella figura 2.1, il tempo di esecuzione (a parte essere molto piccolo anche per dimensioni molto elevate dell'array da ordinare) cresce seguendo rigorosamente la traiettoria di $f(n) = an \log n$, con $a = 1.5 * 10^{-5}$.

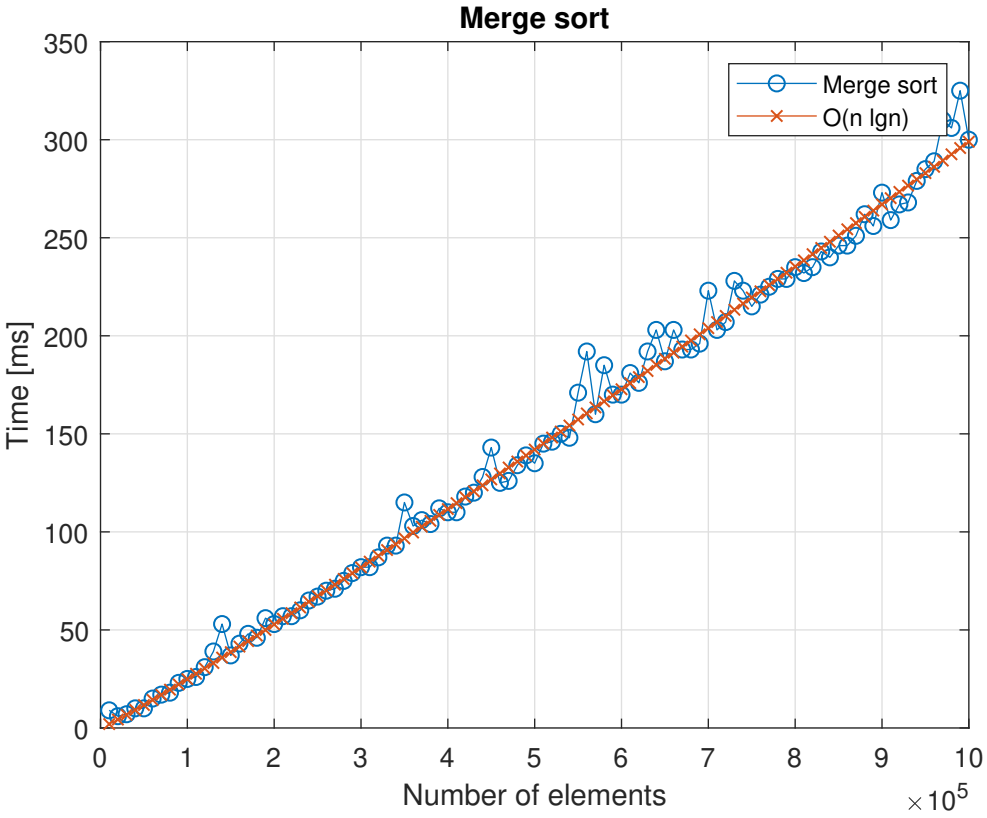

Figura 2.1: Andamento del tempo di esecuzione di Merge Sort all'aumentare della dimensione dell'array da ordinare.

Capitolo 3

Heap Sort

3.1 Introduzione

Introduciamo ora il terzo e ultimo algoritmo studiato, ovvero Heap Sort.

Esso è probabilmente uno dei migliori algoritmi di ordinamento, poiché ordina sul posto e la sua complessità asintotica massima è $T(n) = O(n \log n)$, quindi unisce i pregi di Insertion Sort e Merge Sort.

Il suo funzionamento è basato sugli alberi binari, più precisamente sugli alberi heap (da cui il nome dell'algoritmo). Un albero max-heap (o min-heap) è un albero binario la cui radice è il nodo con chiave massima (o minima) rispetto a tutte le altre chiavi degli altri nodi.

Usando questa importante proprietà, dopo aver reso l'array un albero max-heap con il metodo `BuildMaxHeap`, possiamo estrarre un nodo alla volta dalla radice, richiamare il metodo `MaxHeapify` sull'albero, estrarre la radice che sarà diventata di nuovo il nodo con chiave massima, e iterare fin tanto che l'albero non si svuota.

3.2 Tempo di esecuzione

Come abbiamo accennato nel paragrafo precedente, il tempo di esecuzione dell'algoritmo Heap Sort è $T(n) = O(n \log n)$, poiché sommiamo il tempo impiegato per eseguire n volte `MaxHeapify`, che è $O(n \log n)$, e il tempo impiegato per eseguire `BuildMaxHeap`, che si dimostra essere $O(n)$.

Sommando i due tempi, otteniamo un limite asintotico superiore $O(n \log n)$, che è quindi il tempo migliore rispetto a quelli di Insertion Sort e Merge Sort; a differenza di quest'ultimo, però, c'è una somma di due tempi, $n \log n$ e n, quindi all'atto pratico Heap Sort impiega un tempo quasi doppio di Merge Sort.

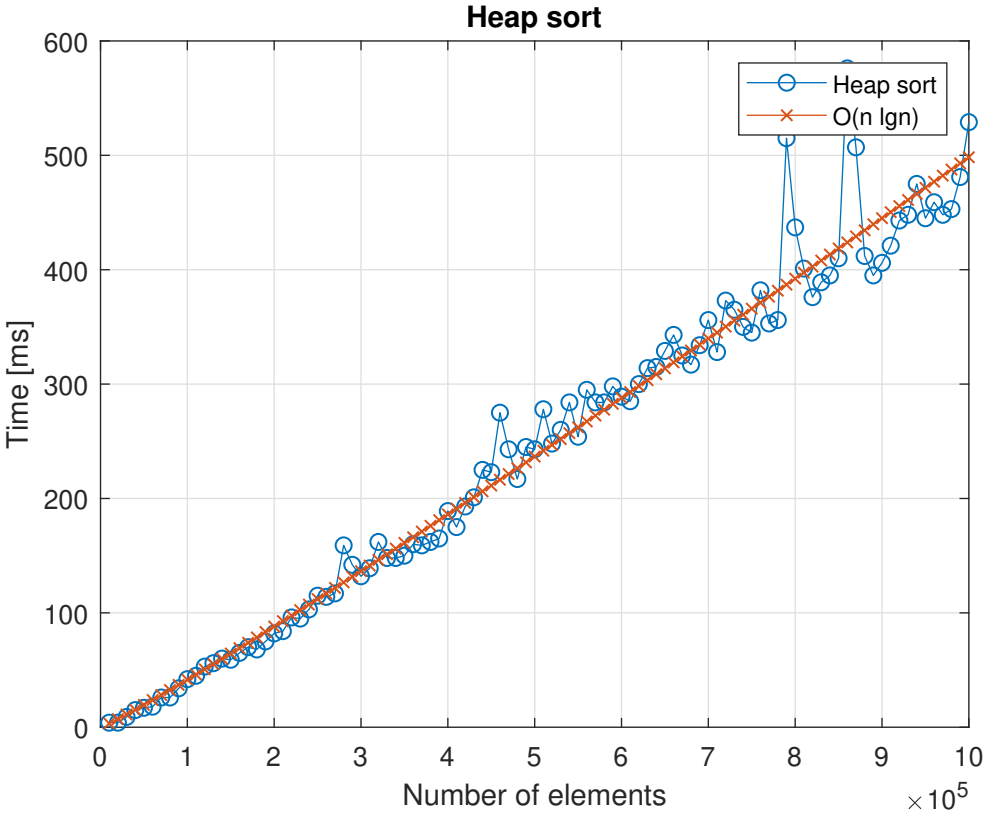

Figura 3.1: Andamento del tempo di esecuzione di Heap Sort all'aumentare della dimensione dell'array da ordinare.

Come possiamo vedere nel grafico 3.1, il tempo di esecuzione cresce come $f(n) = an \log n$, con $a = 2.5 * 10^{-5}$, coefficiente di crescita poco più piccolo del doppio del coefficiente $a = 1.5 * 10^{-5}$ trovato empiricamente nel Merge Sort.