Insertion sort Vs Quick sort performance

Indice

1 Introduzione

La seguente relazione vuole descrivere sperimentalmente e analizzare alla luce della teoria il comportamento di due noti algoritmi di ordinamento: insertion sort e merge sort, ponendosi quindi l'obiettivo di prendere in considerazione le tre principali casistiche di array: ordinato, random e ordinato al contrario.

2 Descrizione algoritmi

Un algoritmo di ordinamento consiste nel permutare un insieme di numeri dati in input in modo che ogni elemento sia minori di quello che lo segue.

2.1 Isertion Sort

Insertion Sort è un algoritmo che risolve il problema dell'ordinamento scambiando sul posto gli elementi di un array. Può risultare utile per ordinare un piccolo numero di elementi poiché le due iterazioni annidate comportano un tempo di esecuzione quadratico: $O(n^2)$, perciò al crescere del numero di elementi può diventare molto costoso.

2.2 Merge sort

Merge Sort a differenza di insertion Sort utilizza il metodo divide et impera per l'ordinamento di un array. Può tornare utile per l'ordinamento di una quantità considerevole di elementi grazie al suo tempo di esecuzione: $\Theta(nlogn)$. Come vedremo il caso peggiore e medio per quest'algoritmo coincidono.

3 Esperimenti

Ho fatto l'esperimento sul tempo di esecuzione di Insertion Sort nel caso migliore, peggiore e medio.
Successivamente ho fatto l'esperimento sul tempo di esecuzione dei due algoritmi nel caso medio mettendoli a confronto.

4 Prove sperimentali

Nelle prove sperimentali prenderemo in considerazione tre classi di array:

1. Array ordinato - Caso migliore
2. Array random - Caso medio
3. Array ordinato al contrario - Caso peggiore

Per ogni tipologia vengono generati array di dimensione crescente da 2 a 1000, al fine di evitare che eventuali anomalie nell'esecuzione possano compromettere i risultati sperimentali e l'affidabilità dei test per ogni dimensione degli array di classe 2 e 3 i test vengono eseguiti 10 volte e ne viene effettuata una media.

Insertionsort	10	100	10000
Migliore	$2 \times 10^{-6} s$	$2 \times 10^{-5} s$	$2,2 \times 10^{-4} s$
Medio	$6 \times 10^{-6} s$	$4,4 \times 10^{-4} s$	$4,5 \times 10^{-2} s$
Peggiore	$8 \times 10^{-6} s$	$8,2 \times 10^{-4} s$	$8,8 \times 10^{-2} s$

Tabella 1: Tempo di esecuzione di InsertionSort per le tre classi di array di 10, 100 e 1000 elementi

MergeSort	10	100	10000
Migliore	$2,7 \times 10^{-5} s$	$3,8 \times 10^{-4} s$	$5,3 \times 10^{-3} s$
Medio	$2,8 \times 10^{-5} s$	$3,9 \times 10^{-4} s$	$5,1 \times 10^{-3} s$
Peggiore	$2,5 \times 10^{-5} s$	$3,8 \times 10^{-4} s$	$5,1 \times 10^{-3} s$

Tabella 2: Tempo di esecuzione di MergeSort per le tre classi di array di 10, 100 e 1000 elementi

4.1 Prove su Insertion Sort

Nella prima prova vediamo come risponde l'algoritmo insertion sort nell'ordinare le tre classi di array sopraelencate.
Il grafico in Figura 1 è costituito dalla variazione dei tempi di esecuzione all'incremento del numero di elementi da ordinare in ogni array, É evidente come i tempi cambiano drasticamente a seconda della classe di array che viene fornita dall'algoritmo, ciò si può notare anche nei dati raccolti nella Tabella 1.
Dalla teoria sappiamo che nel caso migliore (array ordinato) l'insertion sort impiega un tempo di esecuzione lineare rispetto al numero di elementi inseriti, mentre nel caso peggiore (array ordinato al contrario) impiega un tempo quadratico. Il caso medio (array random) rappresenta la situazione più comune nelle applicazioni reali dell'algoritmo e corrisponde ad una via di mezzo tra i due casi.

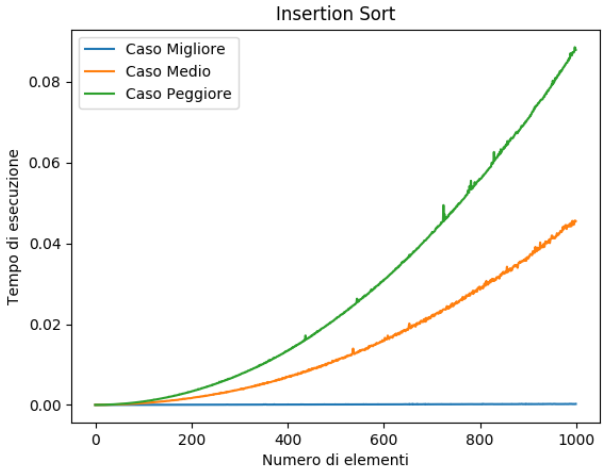

Figura 1: Tempo di esecuzione di Insetion Sort all'aumentare del numero di elementi da ordinare per tre diverse classi di array.

4.2 Prove su Merge Sort

In questa prova consideriamo come risponde l'algoritmo Merge Sort all'ordinamento delle tre classi di array trattare nella prova precedente. Come possiamo constatare dal grafico in Figura 2 e la Tabella 2 ancora una volta i risultati sperimentali hanno un'evidente corrispondenza con la teoria che è alla base dell'algoritmo considerato, infatti sappiamo che i tempi di esecuzione del Merge Sort nel caso peggiore e nel caso medio coincidono.

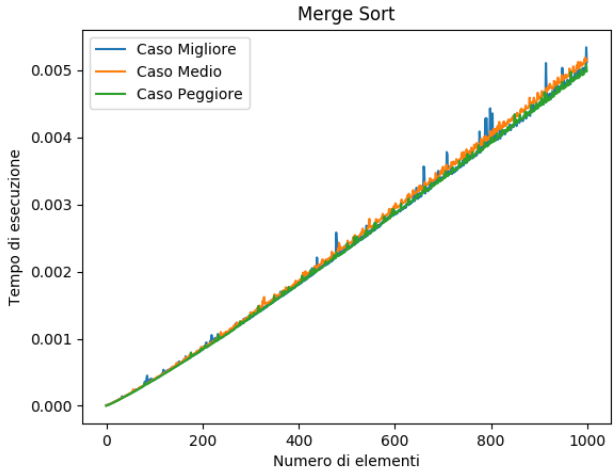

Figura 2: Tempo di esecuzione di Merge Sort all'aumentare del numero di elementi da ordinare per tre diverse classi di array.

4.3 Inseretion Sort VS Merge Sort

Nelle prossime tre prove mettiamo a confronto i comportamenti dei due algoritmi di ordinamento in relazione alle tre classi di array considerate in precedenza.

4.3.1 Caso Migliore

Eseguendo i due algoritmi con degli array ordinati di dimensione crescente, come possiamo vedere in Figura 3 si ottiene che insertion sort è più efficiente, poiché ha costo lineare, mentre merge sort ha lo stesso costo nel caso medio e peggiore.

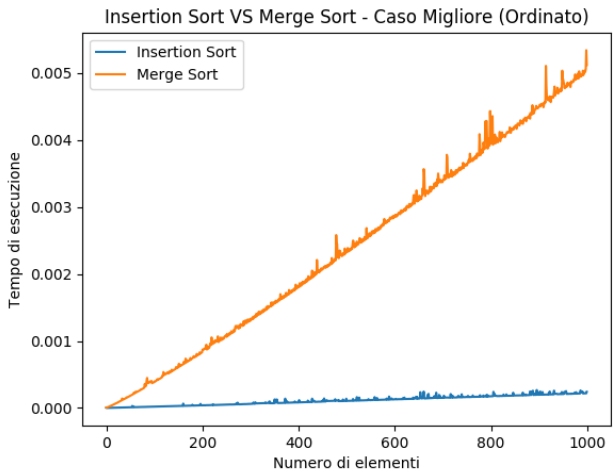

Figura 3: Tempo di esecuzione di InsertionSort e MegeSort all'aumentare del numero di elementi da ordinare per array ordinati.

4.3.2 Caso Medio

Nel caso medio (Figura 4) si applica una randomizzazione dei valori degli array prima di darli "in pasto" agli algoritmi. rispetto al caso precedente si può vedere dal grafico che i ruoli si sono scambiati ed ora insertion sort è più lento.

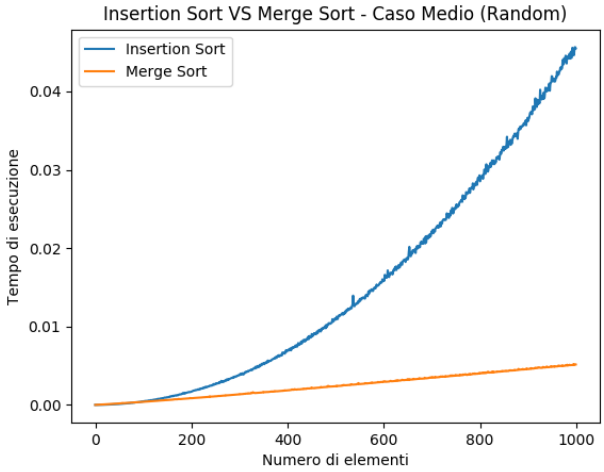

Figura 4: Tempo di esecuzione di InsertionSort e MegeSort all'aumentare del numero di elementi da ordinare per array pseudocasuali.

4.3.3 Caso Peggiore

In quest'ultima prova (Figura 5) sono stati considerati degli array al contrario, ciò corrisponde al caso peggiore di insertion sort, infatti possiamo notare come rispetto al caso medio in cui si aveva un picco di 0.04 s per ordinare un array random di 1000 elementi, con l'array inverso della stessa dimensione ci vuole circa il doppio del tempo, infatti supera 0.08s

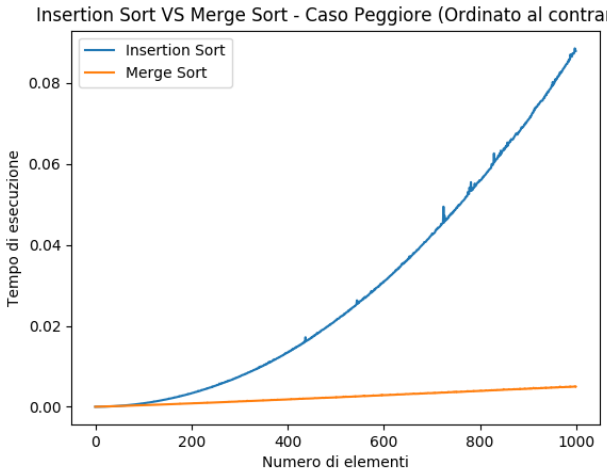

Figura 5: Tempo di esecuzione di InsertionSort e MegeSort all'aumentare del numero di elementi da ordinare per array ordinati al contrario.

4.3.4 Analisi dei Risultati Sperimentali

Alla luce della teoria e dei risultati sperimenti possiamo affermare che tra queste due fonti c'è una netta corrispondenza, infatti a partire da Insertion Sort:
Ordinare un array già ordinato di 1000 elementi è costato un tempo di $2,2 \times 10^{-4}$ s, mentre ordinare un array ordinato al contrario (caso peggiore per insertion sort) ha impiegato un tempo di $8,8 \times 10^{-2}$ s.
Due ordini di grandezza di differenza tra i due casi fanno capire quanto sia ampio il divario tra il caso migliore e il caso peggiore. Il caso medio (array random) si è comportato leggermente meglio nel caso peggiore con un tempo di $4,5 \times 10^{-2}$ s.
Come sappiamo dalla teoria ordinare un array per il merge sort è un'operazione che impiega un tempo che ovviamente cresce al crescere del numero di elementi da ordinare, ma non è influenzato dalla classe di array da ordinare, con le prove sperimentali effettuate infatti è emerso che per ordinare tre array di 1000 elementi di tipo diverso ha impiegato sempre 5×10^{-3}s.

4.4 Documetazione del codice

Il progetto Python utilizzato per le prove sperimentali è composto dai seguenti file .py: Main, InsertionSort, MergeSort, ArrayGen, Test, Plot.

- I File InsertionSort.py e MergeSort.py come si può intuire dal nome implementano i due algoritmi di ordinamento.
- Nel file ArrayGen.py sono contenuti tre metodi adibiti alla generazione di tre classi di array: ordinati, ordinati al contrario e pseudocasuali.
- Il file Test.py contiene i metodi necessari per le prove sperimentali da eseguire, in questo caso sei metodi per il caso migliore, medio e peggiore, sia per insertion sort che per merge sort.
- I file Plot.py e Table.py si occupano rispettivamente di stampare i dati sperimentali su di un grafico .png e di salvarli in una tabella .xlsx.
- Infine il file Main.py ha il compito di coordinare tutte le classi e i file in modo da realizzare le prove sperimentali e dei risultati ottenuti.

5 Conclusione

In seguito al confronto tra dati teorici e sperimentali possiamo concludere che le ipotesi riguardanti i tempi di esecuzione di InserctioSort e MergeSort sono confermate sperimentalmente.
Come visto in Figura 1 insertion sort ha dei tempi diversi a seconda del tipo di array che deve ordinare, mentre come abbiamo visto in Figura 2

merge sort impiega sempre un tempo indipendentemente dal tipo di array da ordinare.

6 Informazioni Sul Calcolatore

6.1 Hardware

CPU: Intel Core i7-8565u @ 1.80Ghz
RAM: 8 GB DDR3
Graphic: Intel HD Graphics 620

6.2 Software

OS: Windows 10 Home
IDE: Pycharm Community Edition 2020.1.1