Università degli Studi di Udine - Informatica

Relazione Ricerca
Mediana Pesata Inferiore

Dicembre 2019

Simone Not 139032

Not Simone 139032

Introduzione

Questa relazione si occuperà di esporre la risoluzione del progetto di laboratorio di algoritmi e strutture dati.

Il Progetto chiedeva di trovare un algoritmo per calcolare la mediana pesata inferiore nel modo più efficiente possibile, calcolarne la complessità e valutare in modo empirico, tramite l'analisi dei tempi di esecuzione, se il valore teorico corrisponde a quello reale.

La risoluzione è stata eseguita con diversi algoritmi le cui complessità sono $\Theta(n^2)$, $\Theta(n \log n)$ ed $\Theta(n)$

Il problema

Trovare la mediana pesata inferiore, che viene calcolata con il seguente metodo:

Si considerino n valori razionali positivi (pesi) $\omega_1, \dots, \omega_n$ e si indichi con W la loro somma: $W = \sum_{i=1}^{n} \omega_i$

Chiamiamo mediana inferiore pesata di $\omega_1, \dots, \omega_n$, il peso ω_k tale che:

$$\sum_{\omega_i < \omega_k} \omega_i < \frac{W}{2} \leq \sum_{\omega_i \leq \omega_k} \omega_i$$

Input: n valori razionali positivi distinti $\omega_1, \dots, \omega_n \in \mathbb{Q}^+$.

Output: La mediana inferiore pesata di $\omega_1, \dots, \omega_n$.

La soluzione

Questo problema può essere risolto in vari modi ed infatti in classe si è discusso di diverse soluzioni, io ne presenterò 3, che sono, a mio avviso, le più corrette per risolvere questo problema.

Soluzione Naive $\Theta(n^2)$ e $\Theta(n \log n)$

(Le seguenti soluzioni sono implementate nel file del progetto in core/lwm/lwm_naive.py)

Il modo più rapido sarebbe ordinare con un algoritmo di ordinamento e successivamente scorrere l'array ordinato sommando elemento per elemento fino quando la condizione non è soddisfatta.
Entrambe le soluzioni proposte in questo paragrafo utilizzano la stessa modalità di verifica della condizione e differiscono solo per l'algoritmo di ordinamento.

Di seguito viene subito analizzata la funzione di verifica, in quanto è identica per entrambe le soluzioni naive.
La funzione di verifica è composta da un for che somma i valori man mano che scorre l'array ed ogni ciclo controlla se $sum1 < \frac{W}{2} <= sum2$
Ogni ciclo quindi viene svolto in $O(1)$ che per la lunghezza n dell'array si deduce quindi che la complessità della funzione di verifica è $\Theta(n)$

Implementazione con Insertion-Sort

L'Insertion-Sort è l'algoritmo più veloce da implementare, si tratta di un algoritmo di ordinamento in place che parte dal primo elemento e si sposta lungo tutto l'array da ordinare, dove a sinistra dell'indice di scorrimento si trovano tutti gli elementi ordinati ed a destra quelli da ordinare. Ad ogni iterazione, rimuove un elemento dalla sottosequenza non ordinata e lo inserisce nella posizione corretta della sottosequenza ordinata, estendendola così di un elemento.

Il caso ottimo per l'algoritmo è quello in cui la sequenza di partenza sia già ordinata, l'algoritmo in questo caso ha tempo di esecuzione lineare $\Theta(n)$.

Il caso pessimo è invece quello in cui la sequenza di partenza sia ordinata al contrario, in questo caso l'algoritmo di Insertion-Sort ha complessità temporale quadratica $\Theta(n^2)$.
Anche il caso medio ha complessità quadratica, il che lo rende impraticabile per ordinare sequenze grandi.

Si conclude che implementando il progetto con questo algoritmo porterà ad avere una complessità:

$$\Theta(n^2) + \Theta(n) = \Theta(n^2).$$

Implementazione con Merge-Sort

Il Merge-Sort è un algoritmo di ordinamento basato su confronti che utilizza un processo di risoluzione ricorsivo, sfruttando la tecnica del Divide et Impera, che consiste nella suddivisione del problema in sotto problemi della stessa natura di dimensione via via più piccola.

Procedimento:

- Se la sequenza da ordinare ha lunghezza 0 oppure 1, è già ordinata. Altrimenti:
- La sequenza viene divisa in due metà, se la sequenza contiene un numero dispari di elementi, viene divisa in due sottosequenze di cui la prima ha un elemento in più della seconda.
- Ognuna di queste sottosequenze viene ordinata, applicando ricorsivamente l'algoritmo
- Le due sottosequenze ordinate vengono fuse. Per fare questo, si estrae ripetutamente il minimo delle due sottosequenze e lo si pone nella sequenza in uscita, che risulterà ordinata.

L'algoritmo Merge Sort, per ordinare una sequenza di n oggetti ha complessità temporale $T(n) = \Theta(n \log n)$ sia nel caso medio che nel caso pessimo. Infatti:

- la funzione merge ha complessità temporale $\Theta(n)$.
- Merge-Sort richiama due volte sé stesso, e ogni volta su circa metà della sequenza in input.

Da questo segue che il tempo di esecuzione dell'algoritmo è dato dalla ricorrenza:

$$\mathrm{T}(n) = 2\mathrm{T}\left(\frac{n}{2}\right) + \Theta(n).$$

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

Si conclude che implementando il progetto con questo algoritmo porterà ad avere una complessità

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

Soluzione Lineare

L'algoritmo di ricerca della mediana pesata inferiore da me implementato fa uso del Select.

Il miglior approccio fino adesso trovato è ordinare l'array e a quel punto cercare l'elemento desiderato. Ma Selezionare un elemento senza ordinare l'array sembra molto più veloce.

Select in O(n) nel peggio caso:

L'algoritmo Select esegue i seguenti passaggi:

1. Dividi gli n elementi dell'array di input in $n/5$ gruppi di 5 elementi ciascuno, e al massimo un gruppo composto dai rimanenti $n \bmod 5$ elementi.
2. Trova la mediana di ciascuno dei gruppi eseguendo l'ordinamento con Insertion-sort degli elementi di ciascun gruppo, dopodiché estrae la mediana dall'array ordinato. Nel caso in cui il gruppo sia di un numero pari di elementi estrae la mediana inferiore.
3. Usa Select in modo ricorsivo per trovare la mediana X delle mediane n/5 trovate nel passaggio 2.
4. Partiziona l'array di input attorno la mediana delle mediane X usando la versione modificata di Partition. Sia key uno in più del numero di elementi sul lato inferiore della partizione, così che X è l'elemento più piccolo e ci sono key elementi sul lato alto della partizione.
5. Se $i = key$, quindi restituire X. Altrimenti, usa Select ricorsivamente per trovare il suo elemento più piccolo nella parte bassa dell'array se $i < key$, o l'elemento ($i - key$) più piccolo nella parte alta se $i > key$.

Complessità del Select

Per analizzare il tempo di esecuzione di Select, determiniamo innanzitutto un limite inferiore sul numero di elementi che sono maggiori dell'elemento di partizionamento x.

Almeno la metà delle mediane trovate il passaggio 2 è maggiore o uguale alla mediana delle mediane X. Pertanto, almeno la metà dei gruppi $\lceil 5/n \rceil$ contribuiscono con almeno 3 elementi che sono maggiori di x, ad eccezione di un gruppo che ha meno di 5 elementi se 5 non divide l'array esattamente, ed il gruppo che contiene x stesso. Rimuovendo questi due gruppi, ne consegue che il numero di elementi maggiore di x è almeno

$$3\left(\left\lceil\frac{1}{2}\left\lceil\frac{n}{5}\right\rceil\right\rceil - 2 \geq \frac{3n}{10} - 6\right)$$

Allo stesso modo, almeno $\frac{3n}{10} - 6$ elementi sono inferiori a x. Pertanto, nel peggiore dei casi, il passaggio 5 chiama SELECT ricorsivamente su al massimo $\frac{7n}{10} + 6$ elementi.

I passaggi 1, 2 e 4 richiedono $O(n)$ tempo.

- Il passaggio 2 consiste in (n) chiamate di ordinamento di inserzione su set di dimensioni $O(1)$.
- Il passaggio 3 richiede tempo $T\left(\left\lceil\frac{n}{5}\right\rceil\right)$
- Il passaggio 5 richiede al massimo un tempo di $T\left(\frac{7n}{10} + 6\right)$

Supponendo che T sta aumentando in modo monotono possiamo quindi ottenere la ricorrenza

$$T(n) \leq \begin{cases} O(1) & if\ n < 140 \\ T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\frac{7n}{10} + 6\right) + O(n) & if\ n \geq 140 \end{cases}$$

Dimostrazione che il tempo di esecuzione è lineare per sostituzione.

Supponiamo $T(n) \leq cn$ per una costante opportunamente grande c e $n\ < 140$, (questo presupposto vale se c è abbastanza grande).

Inoltre, viene presa una costante a per descrivere la componente non ricorsiva dell'algoritmo. Questa variabile viene considerata con an con $n > 0$

$$T(n) \leq c\left\lceil \frac{n}{5} \right\rceil + c\left(\frac{7n}{10} + 6\right) + an$$

$$T(n) = \frac{cn}{5} + c + \frac{7cn}{10} + 6c + an$$

$$T(n) = \frac{9cn}{10} + 7c + an$$

$$T(n) = cn + \left(-\frac{cn}{10} + 7c + an\right)$$

Quindi:

$$-\frac{cn}{10} + 7c + an \leq 0$$

La disuguaglianza è equivalente alla disuguaglianza $c \geq 10a\left(\frac{n}{(n-70)}\right)\ quando\ n\ >\ 70$.

Dato che si assume che $n \geq\ 140$ si ha $\frac{n}{(n-70)} \leq 2$ e quindi la scelta $c \leq 20a$ soddisferà la disuguaglianza. Si noti che la costante 140 può essere sostituita con qualsiasi numero intero maggiore di 70 e quindi scegliere c di conseguenza.

Conclusione: Il tempo di esecuzione è quindi lineare $\Theta(n)$.

Considerazioni sulle soluzioni

Propongo come soluzione l'algoritmo $\Theta(n \log n)$.

Ho implementato 2 versioni del programma con il Select ma purtroppo mi sono reso conto tardi di un errore di implementazione. Ho comunque fatto le analisi dei tempi e l'algoritmo di per sé funziona ma non se ci sono ripetizioni del valore della mediana pesata inferiore nell'array. In quel caso va in ricorsione infinita. L'errore si trova in una cattiva gestione dei valori di ritorno nel 3-way-partition.

Le seguenti soluzioni sono implementate nel file del progetto in core/lwm/lwm_linear.py

Le seguenti soluzioni sono implementate nel file del progetto in core/lwm/lwm_median_approx.py

Il programma

Scelta del linguaggio

Per risolvere questo problema ho deciso di utilizzare Python, un linguaggio di programmazione ad alto livello, orientato agli oggetti, adatto, tra gli altri usi, a sviluppare applicazioni distribuite, scripting, computazione numerica e system testing.

La motivazione di questa scelta è dettata dal fatto che è un linguaggio multi-paradigma che ha tra i principali obiettivi: dinamicità, semplicità e flessibilità e si applica bene in ambito didattico, in quanto permette di concentrare l'attenzione più sull'algoritmo che sulla sua implementazione. Infatti, Python è uno dei linguaggi più simili allo pseudo codice che attualmente abbiamo a disposizione. Inoltre, la possibilità di utilizzare un linguaggio di alto livello rende l'implementazione molto più veloce e dinamica, tagliano notevolmente i tempi di sviluppo e test.

Tuttavia, mi rendo conto che per questioni di efficienza, in un ambiente di produzione, scegliere Python per implementare algoritmi di basso livello non sia una scelta corretta, ma essendo un progetto didattico questa considerazione può potenzialmente essere ignorata.

Compilazione ed esecuzione del programma

È necessario Python 3.6 o superiore per eseguire questo programma

- Per eseguire il progetto eseguire main.py
- Per il calcolo tempi eseguire il main_time.py

Su Linux è già definito l'interprete quindi è possibile eseguire il programma banalmente con:
./main.py
./main_time.py

In alternativa è possibile eseguire il programma nel modo standard con:
python3 main.py
python3 main_time.py
Questo metodo funziona su tutti i sistemi operativi.

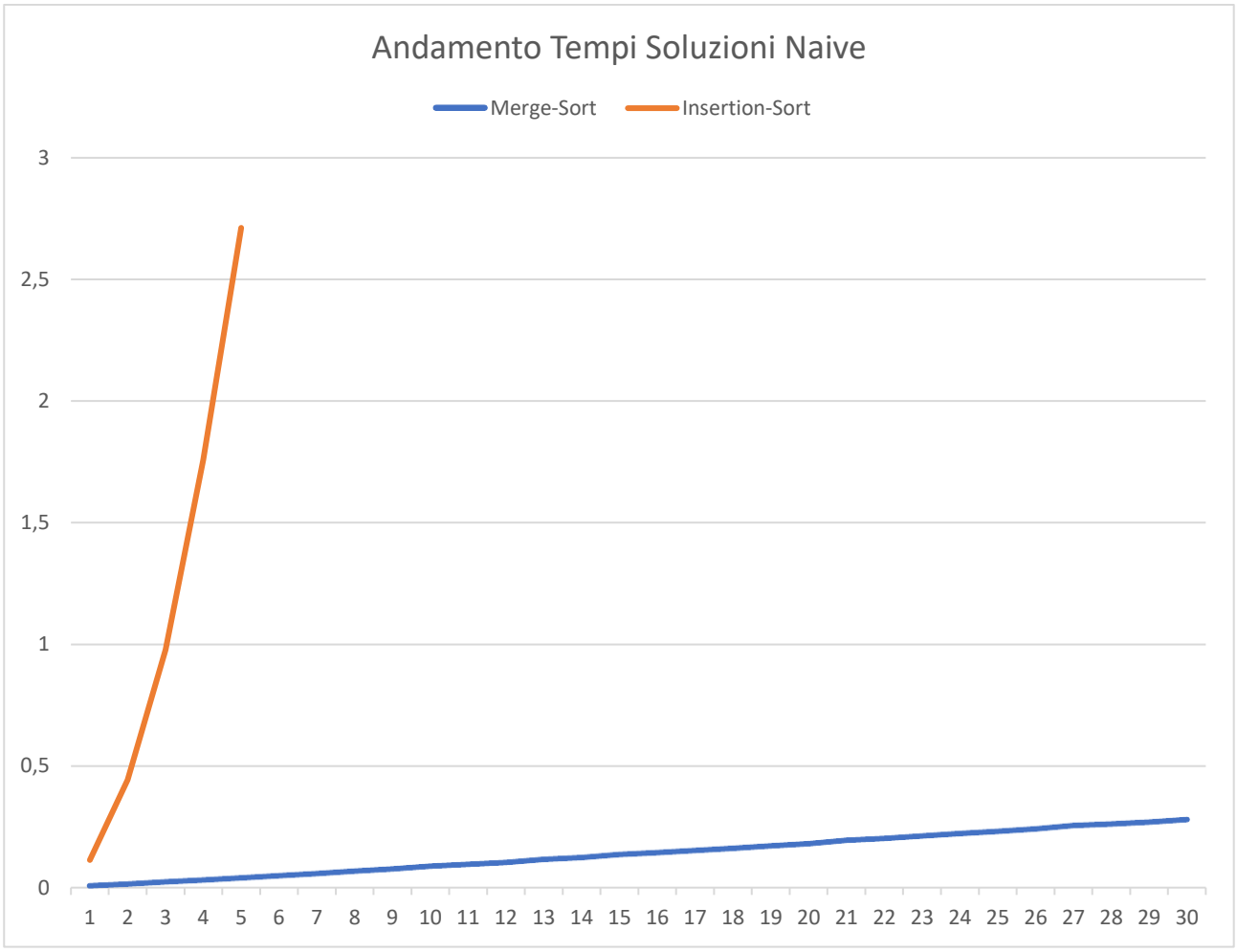
Se si è in un ambiente virtuale (venv) l'esecuzione viene fatta con:
python main.py
python main_time.py

Dove "args" rappresenta lo standard input

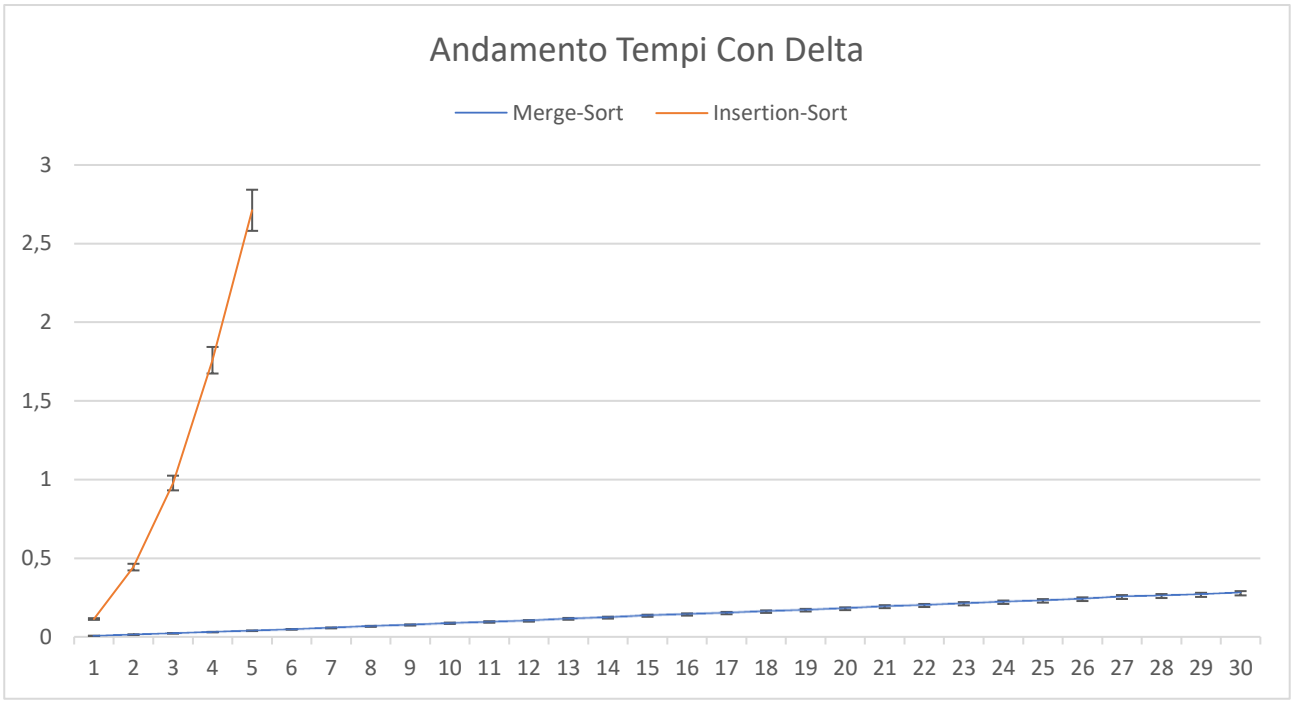
Analisi dei tempi

Si vuole calcolare il Tempo di esecuzione di un programma su un input d, per determinare il tempo che trascorre dall'inizio dell'esecuzione del programma su d sino al momento in cui l'esecuzione termina. Questo calcolo del tempo verrà ripetuto su diversi valori di d in modo da determinare una dipendenza (magari approssimata) funzionale esplicita.

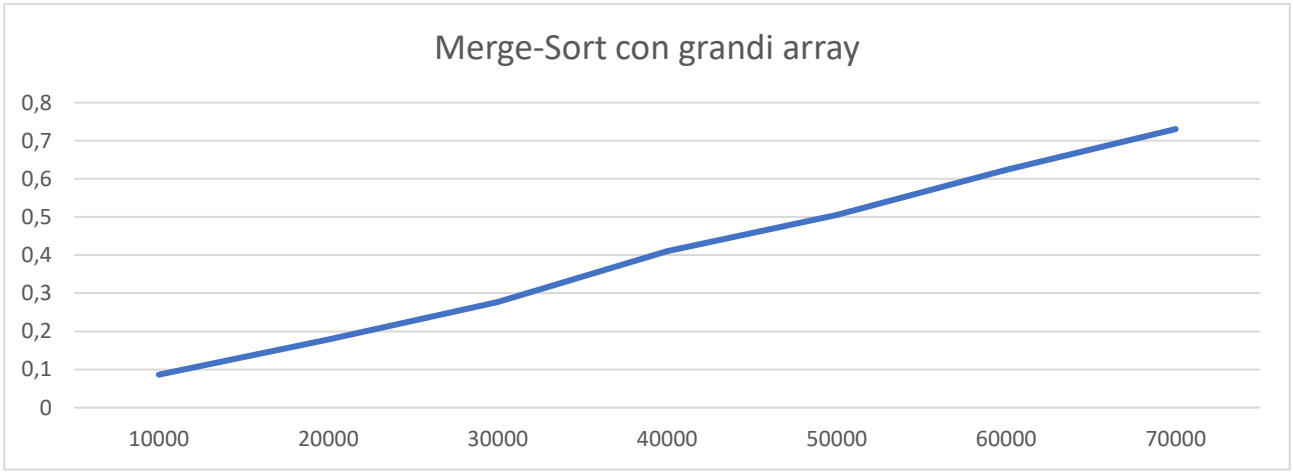
(I dati grezzi si trovano a fine relazione)



(i dati sono rappresentati in migliaia 1 = 1000, i tempi in secondi)



(i dati sono rappresentati in migliaia 1 = 1000, i tempi in secondi)

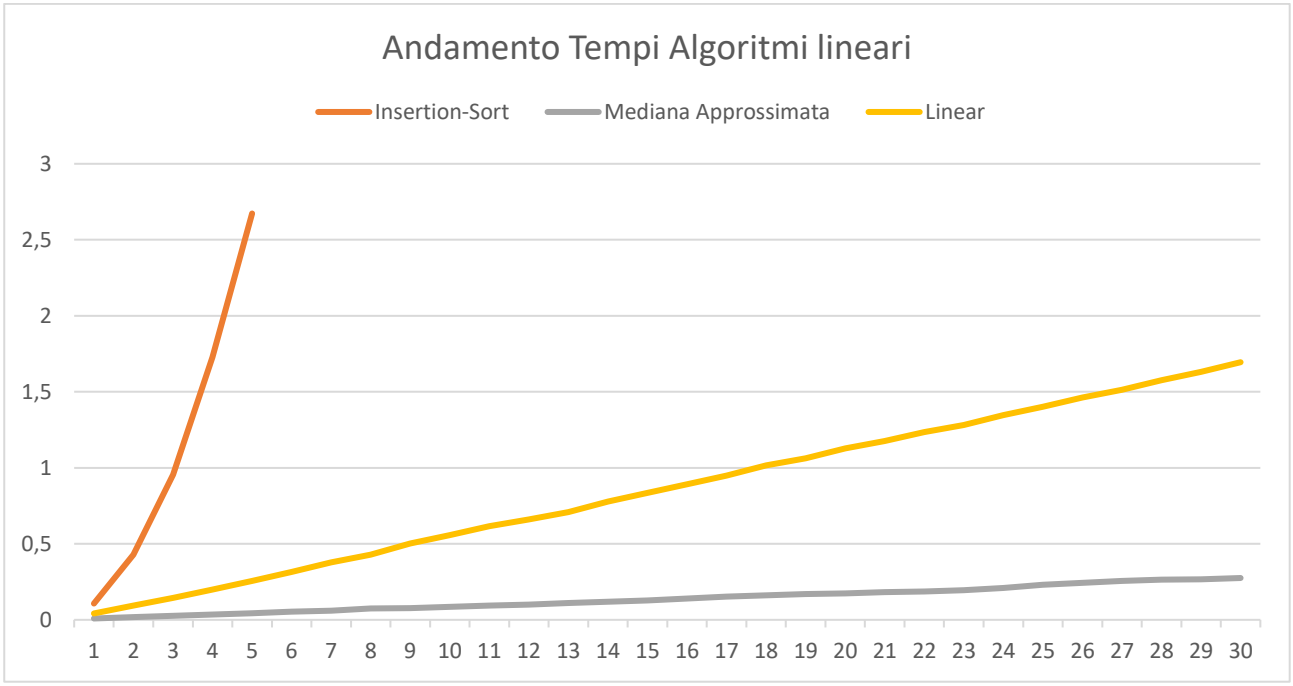


(i tempi dei dati sono rappresentati in secondi)

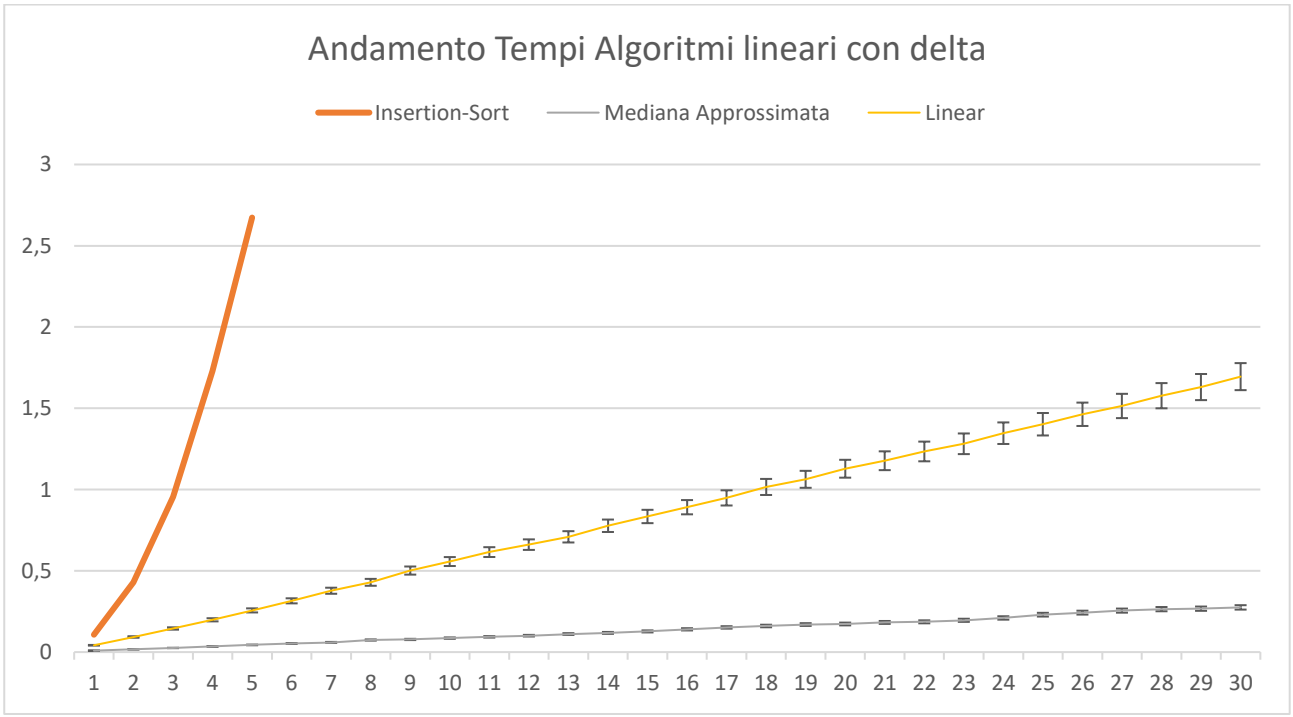
Considerazioni sui grafici naive

- L'algoritmo con l'Insertion-Sort si ha un andamento chiaramente quadratico, nonostante i pochi valori
- Per quanto riguarda l'algoritmo con il Merge-Sort si ha un andamento piuttosto lineare e non si nota molto la curva ed all'apparenza sembra lineare

Ora seguiranno i grafici degli Algoritmi lineari, viene tenuto l'Insertion-Sort per confronto.



(i dati sono rappresentati in migliaia 1 = 1000, i tempi in secondi)



(i dati sono rappresentati in migliaia 1 = 1000, i tempi in secondi)

Considerazioni sui grafici lineari

- L'algoritmo Lineare risulta rispettare l'andamento aspettato. Tuttavia, come tempi di esecuzione risulta più lento di quello con il Merge-Sort, nel caso di un'implementazione in linguaggio C questo non sarebbe successo.

Conclusione

I risultati dei tempi empirici non sono stati quelli che mi aspettavo, ma questo è dovuto principalmente al linguaggio ed al fatto che python utilizza una macchina virtuale per funzionare. Ci sono 2 soluzioni che mi vengono in mente per migliorare i risultati e queste sarebbero cambiare interprete di default ed utilizzare PyPy (comune alternativa) che è stato scritto appositamente per rendere più veloce l'esecuzione, e la seconda, molto più drastica è usare CPython per compilare il programma in C puro per ridurre i tempi di esecuzione lanciandolo senza macchina virtuale.

Tempi

Riporto la tabella tempi completa degli algoritmi sopra analizzati. I tempi sono formattati nel seguente modo:
NUMERO DI ELEMENTI, TEMPO IN SECONDI, DELTA

Insertion-Sort

1000	0,107025246	0,005305988	4000	1,728470948	0,084516084
2000	0,429504755	0,021153237	5000	2,6731492	0,13075631
3000	0,955650818	0,046703464			

Merge-Sort

1000	0,006426627	0,000318737	19000	0,16960294	0,008293687
2000	0,013913669	0,000685574	20000	0,178769469	0,008750629
3000	0,02235715	0,001092066	21000	0,191942798	0,009484025
4000	0,030145252	0,001479852	22000	0,199362058	0,009743829
5000	0,03880655	0,001920832	23000	0,210569183	0,01033493
6000	0,047606525	0,002364716	24000	0,2200243	0,010771219
7000	0,056734325	0,002789297	25000	0,228838478	0,011202036
8000	0,066666518	0,00325068	26000	0,239303988	0,011717866
9000	0,075486455	0,003745276	27000	0,253112093	0,012436171
10000	0,086401298	0,004235379	28000	0,259427863	0,012695016
11000	0,094644061	0,004697623	29000	0,266967478	0,013033437
12000	0,101636942	0,00497555	30000	0,277216542	0,013560452
13000	0,114524729	0,005701409	40000	0,410197031	0,02039951
14000	0,121988162	0,006020158	50000	0,504956297	0,02496783
15000	0,134195512	0,006687191	60000	0,624066637	0,030665399
16000	0,142084005	0,007017532	70000	0,730696224	0,035956666
17000	0,151172793	0,007428585	80000	0,837646053	0,041276663
18000	0,16022545	0,007867999			

Linear

1000	0,041967056	0,002094072	19000	1,062948396	0,052246024
2000	0,092988409	0,004626455	20000	1,128107284	0,055051995
3000	0,145202171	0,007147862	21000	1,177266172	0,057857965
4000	0,198659536	0,009736893	22000	1,23442506	0,060663936
5000	0,256542009	0,012557385	23000	1,281583948	0,063469906
6000	0,315331058	0,015608049	24000	1,346742836	0,066275876
7000	0,377514889	0,018650698	25000	1,401901724	0,069081847
8000	0,429227165	0,021206451	26000	1,463060612	0,071887817
9000	0,501596335	0,024957513	27000	1,51421953	0,074693788
10000	0,557033893	0,027544937	280005	1,577378388	0,077498758
11000	0,615218045	0,030225122	29000	1,630537276	0,080305729
12000	0,661101804	0,032417383	30000	1,694696164	0,083111643
13000	0,709160268	0,034694098			
14000	0,777153957	0,038216172			
15000	0,834312844	0,041022142			
16000	0,891471732	0,043828113			
17000	0,94863062	0,046634083			
18000	1,015989508	0,049440054			