UNIVERSITÀ DEGLI STUDI DI UDINE
hic sunt futura

Corso di
Laboratorio di Algoritmi e Strutture Dati (a.a. 2024/25)

Prestazioni ed Analisi di Algoritmi di Ordinamento:

Un Approccio Basato su Quantità ed Entropia

GIUGNO 2025

Collaboratori	Matricola	Indirizzo	Contatti
Londero Lorenzo	167371	IBML	londero.lorenzo001@spes.uniud.it
Peruzza Andrea	168233	IBML	peruzza.andrea@spes.uniud.it
Fabro Giovanni	149182	IBML	fabro.giovanni@spes.uniud.it

Sommario

Introduzione

Obiettivi

Rappresentare e analizzare singolarmente, tramite confronti grafici, i tempi di esecuzione medi di quattro algoritmi di ordinamento.

Descrizione del progetto

Il progetto consiste nell'implementazione di quattro algoritmi di ordinamento per array di interi. Di questi algoritmi si esegue una stima e un'analisi dei tempi medi di esecuzione in funzione della dimensione dell'array e del range di interi su cui gli elementi dell'array possono variare.

Per raggiungere lo scopo ci avvalleremo di alcuni servizi informatici:

- Linguaggio di programmazione **python** utilizzato per l'implementazione dei quattro algoritmi e del programma per l'analisi dei tempi di esecuzione medi degli stessi. La scelta è ricaduta su python per la leggibilità elevata per la comprensione di concetti algoritmici, oltre alla disposizione di librerie utili a questo progetto, come **time** per la misurazione delle prestazioni e **plotly** per la visualizzazione grafica dei dati.
- L'editor di codice sorgente **Visual Studio Code** come ambiente nel quale implementare in python i quattro algoritmi di ordinamento.
- L'applicazione web **Jupyter Notebook** per l'analisi grafica dei tempi di esecuzione. Jupyter permette di combinare codice eseguibile alle visualizzazioni grafiche, oltre alla possibilità di eseguire codici in blocchi separati, limitando così i tempi di processamento nel caso si ripeta l'esecuzione di determinati blocchi.

Algoritmi

QuickSort

QuickSort è spesso la migliore scelta pratica di ordinamento poiché è notevolmente efficiente nel caso medio con costo Θ(nlogn).

QuickSort, similmente ad altri algoritmi di ordinamento come MergeSort, utilizza l'approccio *divide-et-impera,* ovvero l'array viene inizialmente diviso per poi trasmettere ricorsivamente il problema dell'ordinamento ai sotto-array.

Funzionamento dell'algoritmo per l'ordinamento di un array A[p...r]:

- **Divide**: L'array A[p...r] viene ripartito in due sotto-array A[p...q] e A[q+1...r] tali che ogni elemento di A[p...q] sia minore o uguale ad ogni elemento di A[q+1...r].
  L'indice q e il partizionamento in loco dei due sotto-array viene svolto da Partition(A, p, r);
- **Impera**: I due sotto-array sono successivamente ordinati allo stesso modo tramite chiamate ricorsive
- Successivamente tutti i sotto-array ordinati in loco vengono ricombinati a formare l'array A[p...r] ordinato.

La procedura principale di QuickSort è quindi Partition(A, p, r).

Partition(A, p, r) si occupa quindi di decretare un pivot e classificare gli altri elementi dell'array in base al pivot.

Funzionamento di Partition(A, p, r):

- Inizialmente viene scelto l'elemento che farà da pivot (generalmente l'ultimo elemento dell'array)
- Successivamente viene inizializzato l'indice i che fungerà da divisore degli elementi <= del pivot e > del pivot. Per ogni elemento dell'array se esso è <= del pivot viene incrementato i e viene scambiato l'elemento in analisi con l'elemento in posizione i.
- Infine viene posizionato il pivot al suo posto, tra gli elementi <= di lui e gli elementi > di lui, e viene restituito l'indice del pivot.

Vantaggi:

- Procedura **in-place** in quanto Partition classifica gli elementi scambiando gli elementi sullo stesso array di input.
- Generalmente più efficiente della maggior parte degli algoritmi di ordinamento.
- Adattabile a strategie ottimizzate (Esempio: QuickSort3Way).

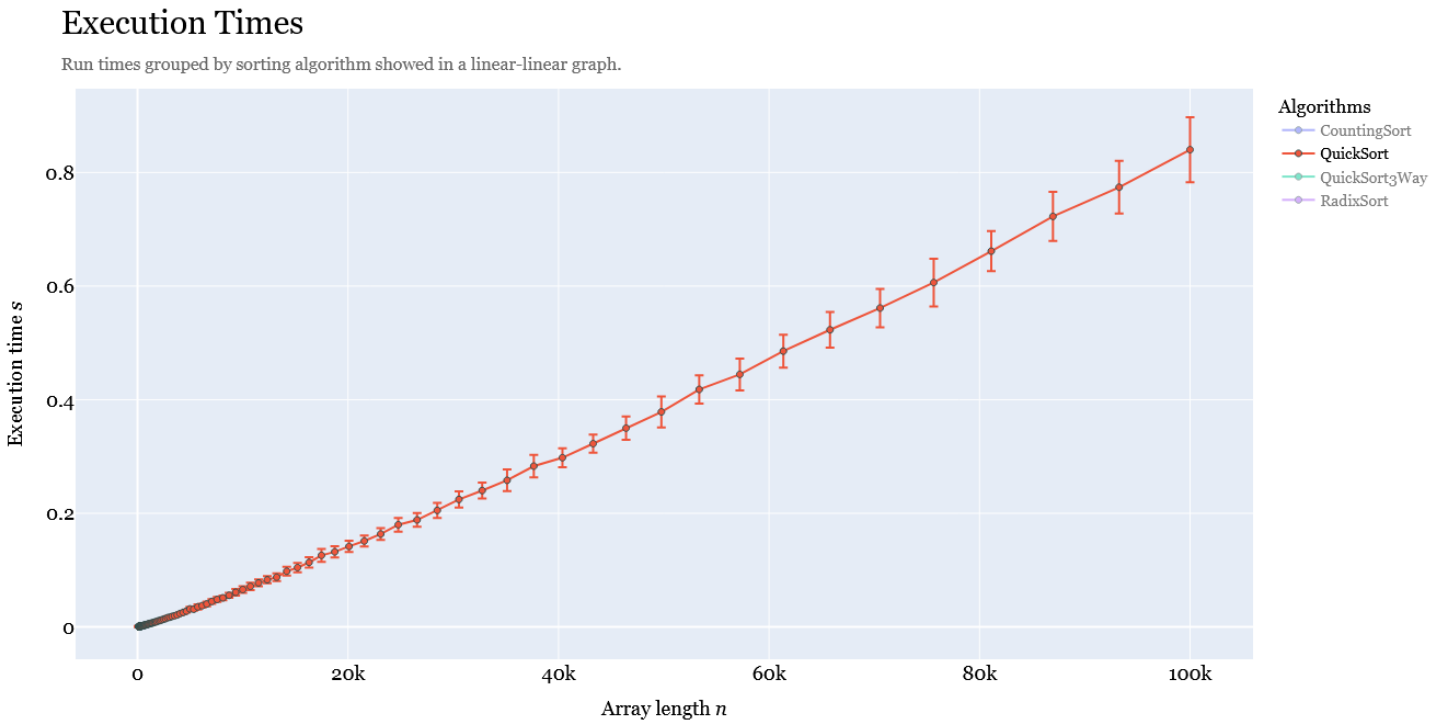
Svantaggi:

- Non è **stabile**, ovvero non conserva l'ordine relativo degli elementi.

Complessità:

- Caso migliore e caso medio: **Θ(nlogn).**
- Caso peggiore (se il pivot scelto è sempre il primo o l'ultimo elemento dell'array ordinato):**Θ(n^2)**

Grafico Lineare:



QuickSort3way

Per far fronte al problema dei duplicati si utilizza questa versione di QuickSort che sfrutta una variante di Partition la quale invece di dividere l'array in due parti (elementi < pivot ed elementi > pivot) lo divide in tre aggiungendo la condizione gli elementi siano uguali al pivot.

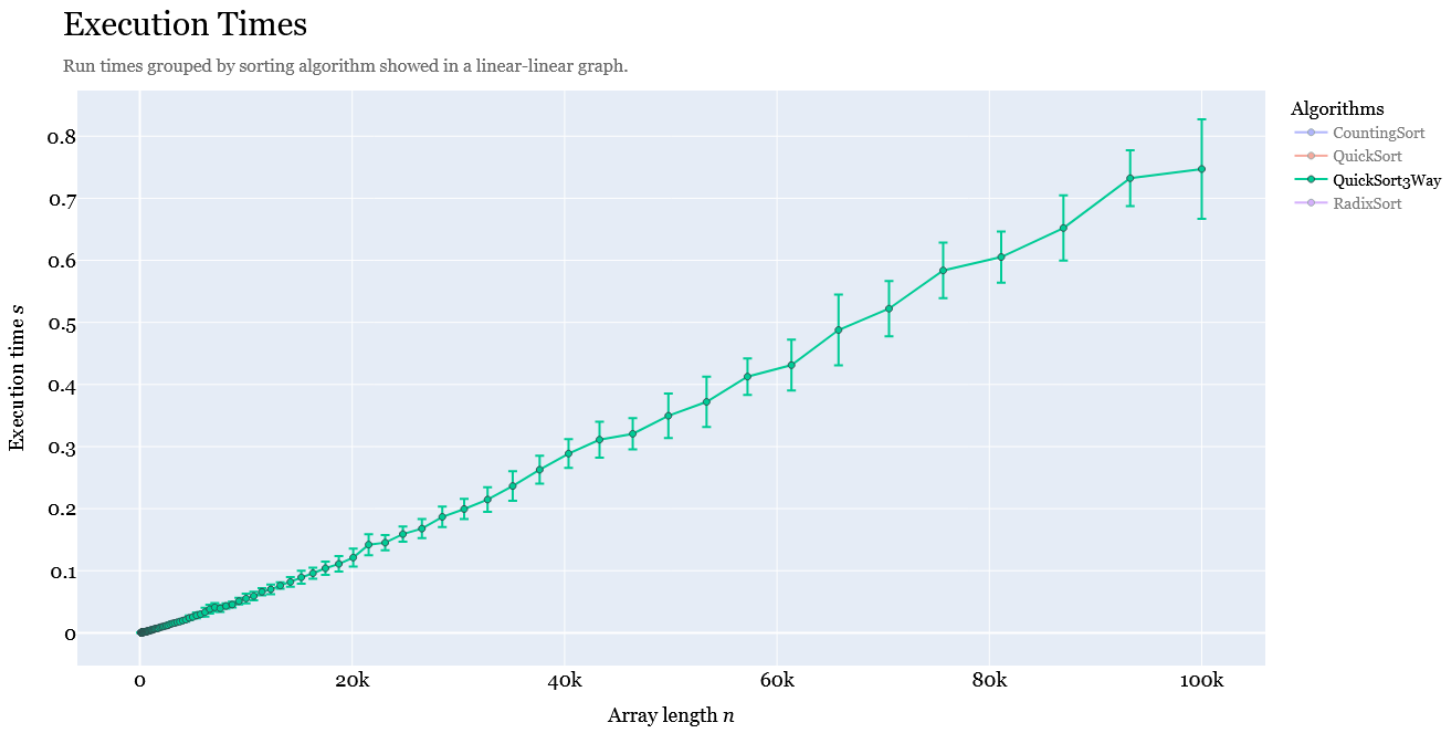
Vantaggi

- Riduce il numero di operazioni su array con tanti duplicati;
- Suddivide meglio l'array evitando squilibri
- Non esegue la ricorsione per i casi uguali al pivot
- Rende l'algoritmo più efficiente con determinati casi

Complessità:

- Con h elementi **distinti:**
  - Caso migliore: $\Theta(n * h)$
  - Caso medio: $\Theta(n\ log\ n)$
- Caso migliore/medio: $O(n\ log\ n)$
- Caso peggiore: $O(n^2)$

Grafico Lineare:



Counting Sort

Si basa sul fatto che ognuno degli n elementi dell'array in input sia un intero compreso tra 1 e k, per qualche intero k. L'idea di base è determinare per ogni elemento x in input, il numero di elementi minori di x. Questa informazione può essere usata per posizionare l'elemento x direttamente nella sua posizione nell'array di output.

Nel codice di Counting Sort si assume che l'input sia un array $A[1 \dots n]$ e si utilizzano altri due array $B[1 \dots n]$ che mantiene l'output ordinato e $C[1 \dots k]$ come array temporaneo. Il vettore C viene popolato incrementando $C[i]$ per ogni valore di un elemento in input, in questo modo $C[i]$ conterrà il numero di input uguali ad i per ciascun elemento intero $i = 1, \dots, k$. Sommando $C[i]$ con $C[i-1]$ si determina quanti elementi dell'input sono minori o uguali ad i. Alla fine, si inserisce ogni elemento dell'array input A nell'array output B nella sua corretta posizione ordinata.

Complessità:

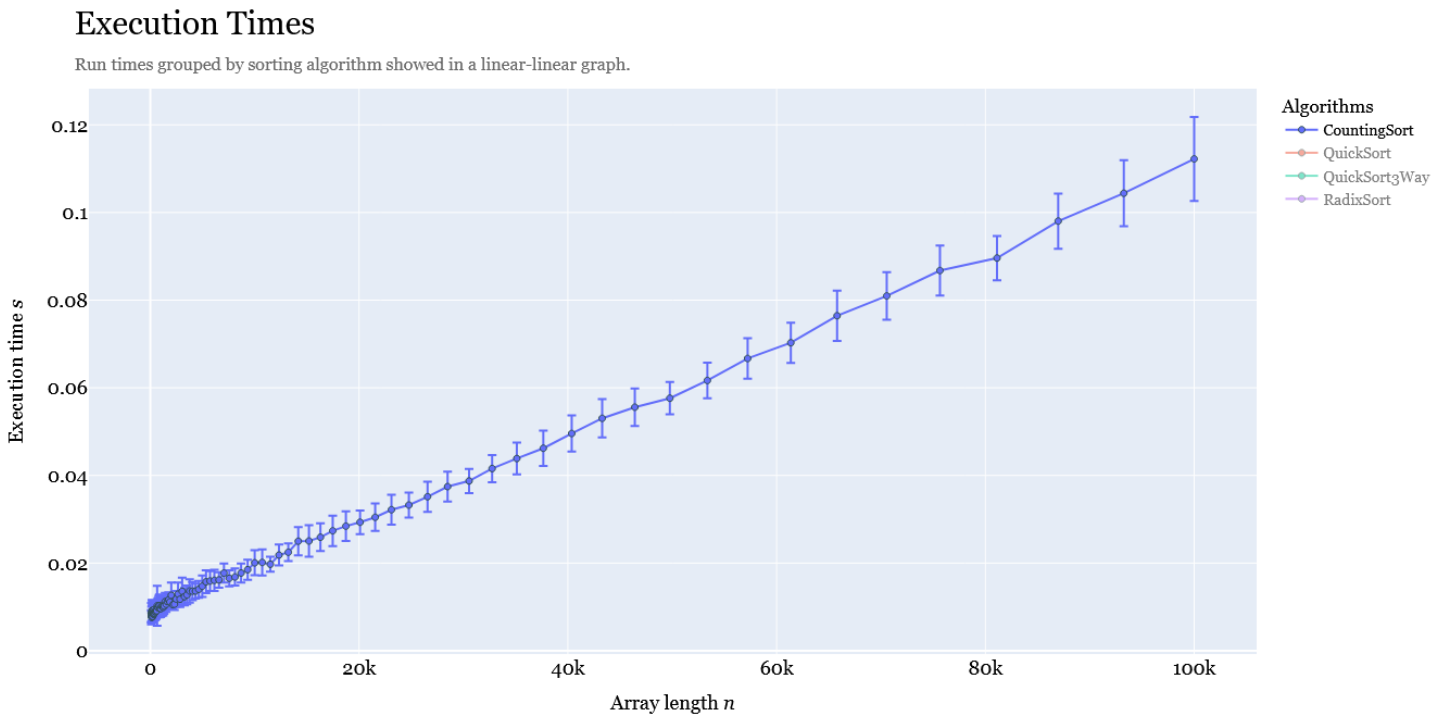
- Il primo ciclo **for** impiega tempo: $O(\mathrm{n})$;
- Il secondo ciclo **for** impiega tempo: $O(k)$;
- Il terzo **for** impiega tempo: $O(n)$;

Quindi il tempo totale è $O(k+n)$. Si utilizza di solito quando si ha $k = O(n)$ e quindi il tempo di esecuzione sarà $O(n)$.

Vantaggi:

- Non è un ordinamento per confronti, usa i valori effettivi degli elementi per indicare direttamente un elemento all'interno di un array;
- È **stabile**, cioè gli elementi con lo stesso valore compaiono nell'array di output nello stesso ordine in cui compaiono in quello di input.

Grafico Lineare:



RadixSort

RadixSort è uno degli algoritmi di ordinamento che non utilizzano confronti.
Agisce ordinando gli elementi di un array analizzando ogni cifra decimale, partendo dalla cifra meno significativa.

Il punto importante di questo algoritmo, che si appoggia ad un secondo algoritmo di ordinamento a scelta, è il mantenimento stabile dell'ordine delle cifre. Per questo motivo l'implementazione di RadixSort deve contenere un algoritmo di ordinamento stabile, come abbiamo visto essere CountingSort.

Funzionamento:

- Dato il numero massimo di cifre egli elementi dell'array, RadixSort applica CountingSort ad ogni colonna partendo dalla cifra meno significativa, ordinando così stabilmente gli elementi

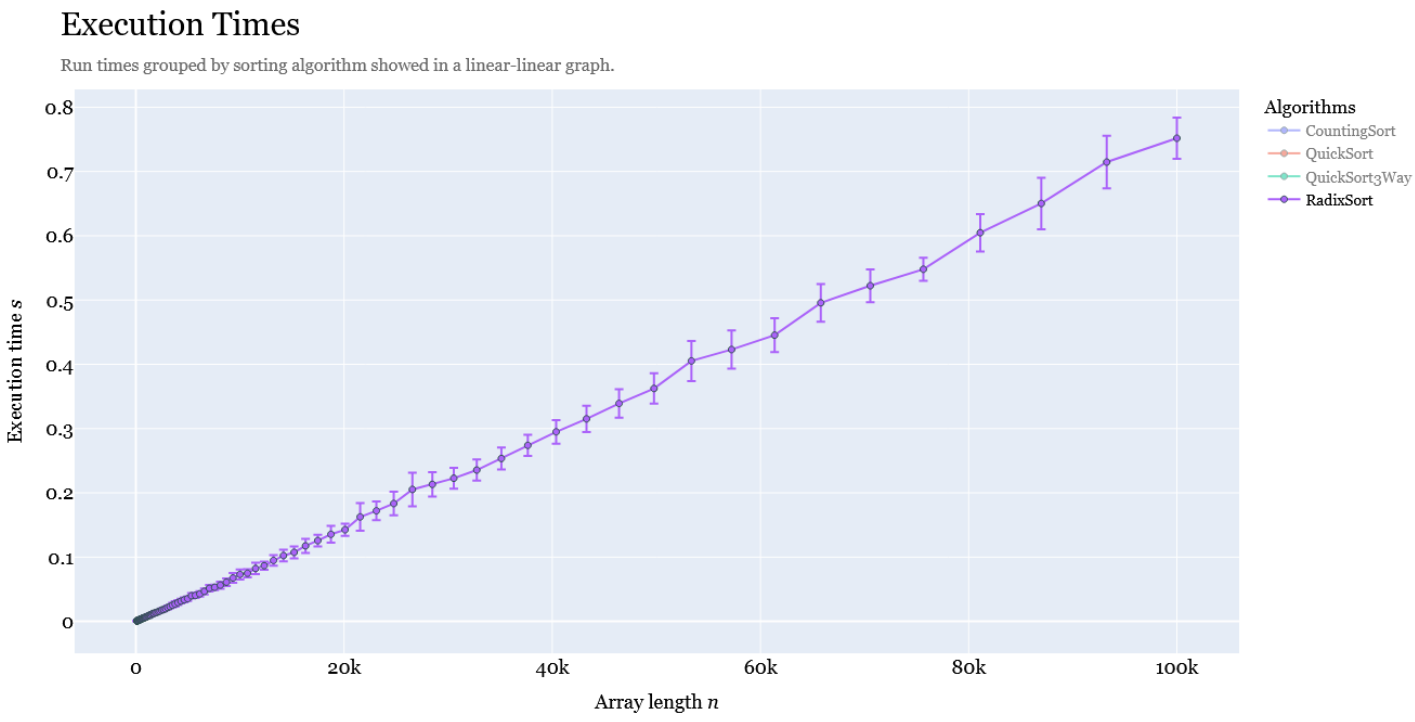
Vantaggi:

- Algoritmo stabile (mantiene la posizione relativa degli elementi).
- Non è basato su confronti, quindi, può superare il limite teorico di complessità Θ(nlogn).
- Molto efficace su dataset o array con numeri interi di lunghezza fissa.

Complessità:

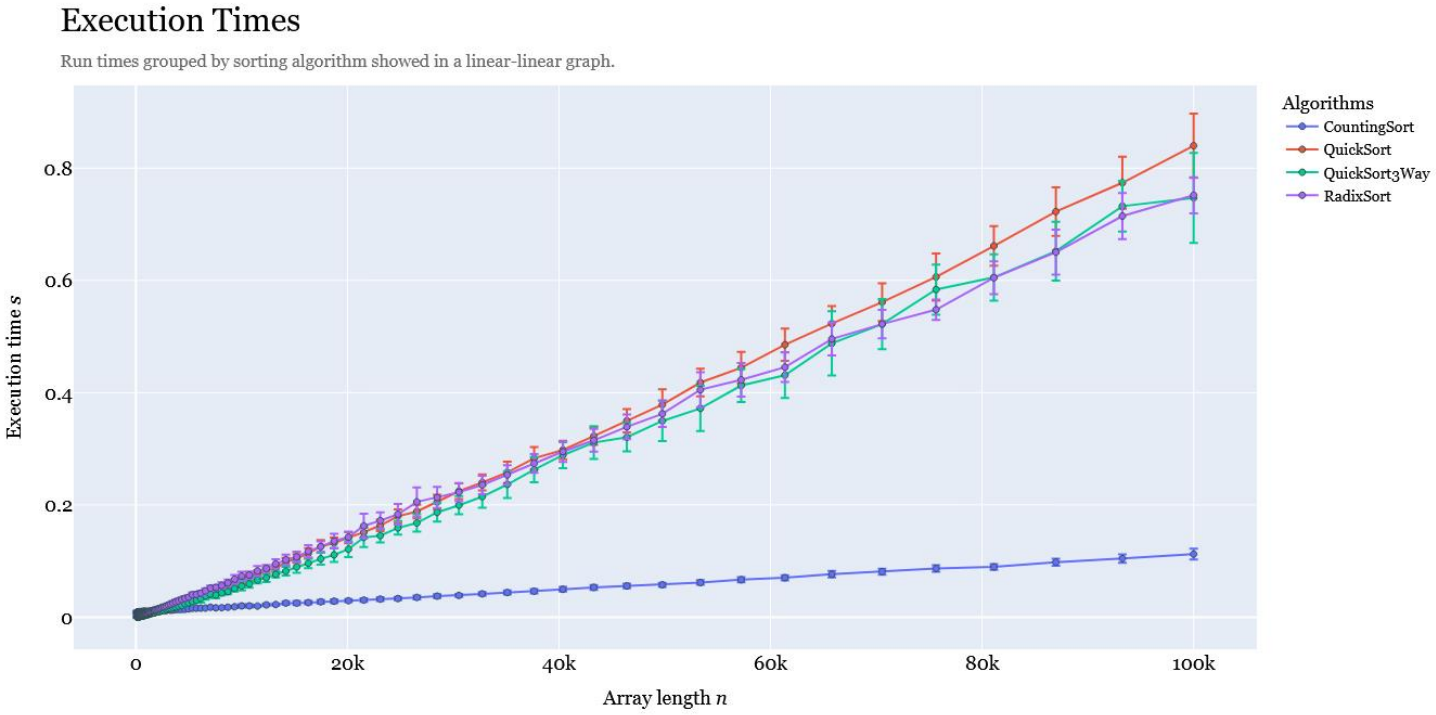
- La complessità dipende da quale algoritmo viene scelto per l'ordinamento intermedio. Se usato CountingSort avrà una complessità di $\Theta(n+k)$ per ordinare tutti gli elementi per ogni cifra fino a d, con k elemento massimo dell'array e d numero massimo di cifre degli elementi dell'array.
  Il costo totale è quindi $\Theta(dn+dk)$ e se d è costante e $k = O(n)$, Il tempo di esecuzione è lineare.

Grafico Lineare:



Analisi dei grafici dei tempi di esecuzione

Per analizzare le tempistiche di risoluzione degli algoritmi descritti in precedenza, sono stati generati 100 campioni per ciascun algoritmo, variando la lunghezza n dell'array in un intervallo compreso tra 100 e 100.000. Ognuno di questi array è popolato da valori interi casuali compresi tra 100 e 1.000.000.



Osservazioni

CountingSort:

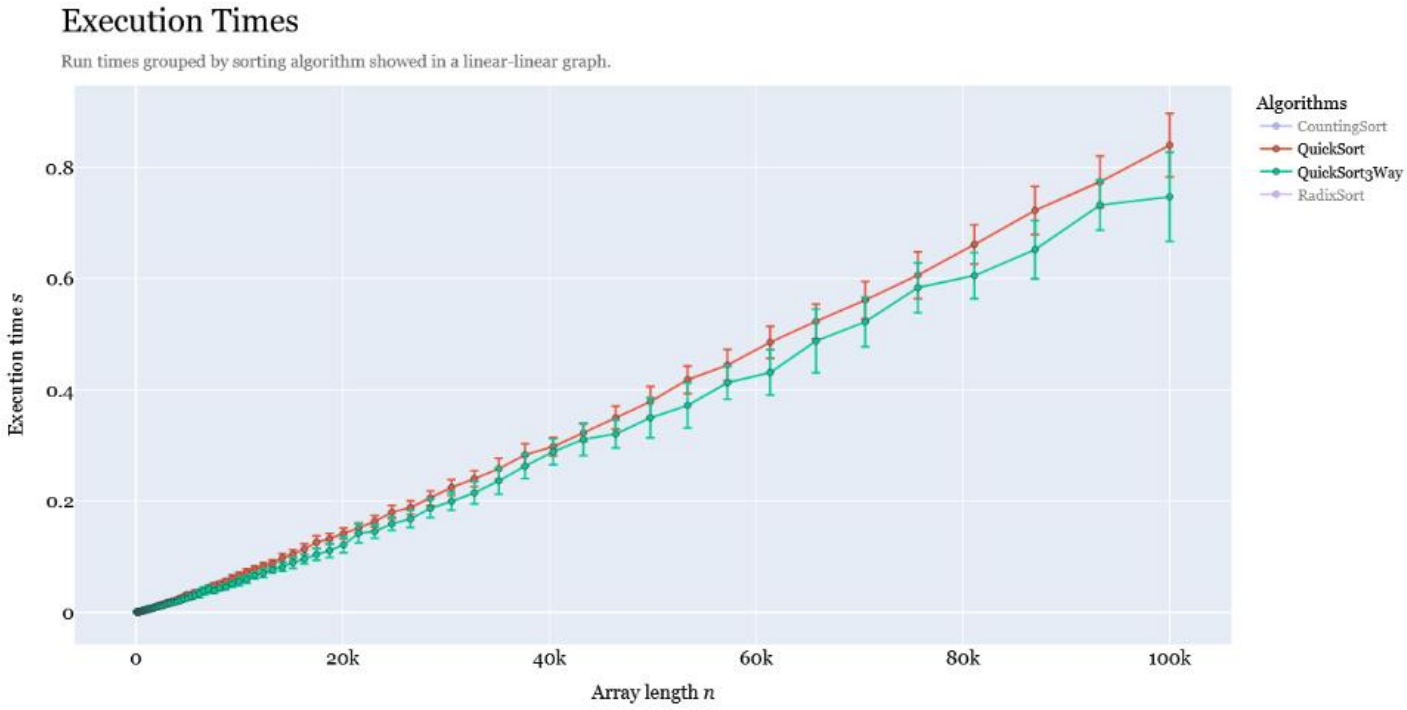
È sicuramente l'algoritmo più interessante in questa analisi in quanto si distacca maggiormente dagli altri.

Presenta un andamento quasi costante come ci si poteva aspettare dalle caratteristiche descritte in precedenza. Possedendo infatti una complessità di O(n+k), il distacco dall'andamento costante è definito proprio da k (che ricordiamo essere l'elemento massimo dell'array su cui è svolta l'analisi), quindi si nota come per array più lunghi, avendo la possibilità randomica di avere valori di k anche molto grandi (massimo 1.000.000), aumenteranno i tempi di esecuzione dell'algoritmo. Sostanzialmente per valori di k maggiori di O(n) l'esecuzione di CountingSort non sarà lineare.

Tuttavia, è importante notare il comportamento dell'algoritmo per array di lunghezza più piccola che verrà analizzata successivamente nel grafico in scala logaritmica.

QuickSort e QuickSort3Way:

Presentano entrambi una crescita abbastanza regolare molto simile nel tempo, coerente con la loro complessità media $O(n \log n)$. Inoltre, per la totalità o quasi degli array la versione QuickSort3Way risulta leggermente più veloce, distacco tra i due algoritmi che si nota maggiormente per array di input di grandi dimensioni, in quanto l'algoritmo riesce a gestire meglio rispetto alla versione base il caso in cui ci siano elementi duplicati, limitando anche i casi peggiori con molti duplicati ad una complessità di O(n*log(n)).
Il distacco non troppo evidente tra i due algoritmi è attribuibile all'ampio range di elementi randomici degli array (da 100 a 1.000.000) che permettono agli array stessi di avere un numero minimo di duplicati, cosa che limita l'ottimizzazione nella partizione introdotta da QuickSort3Way.



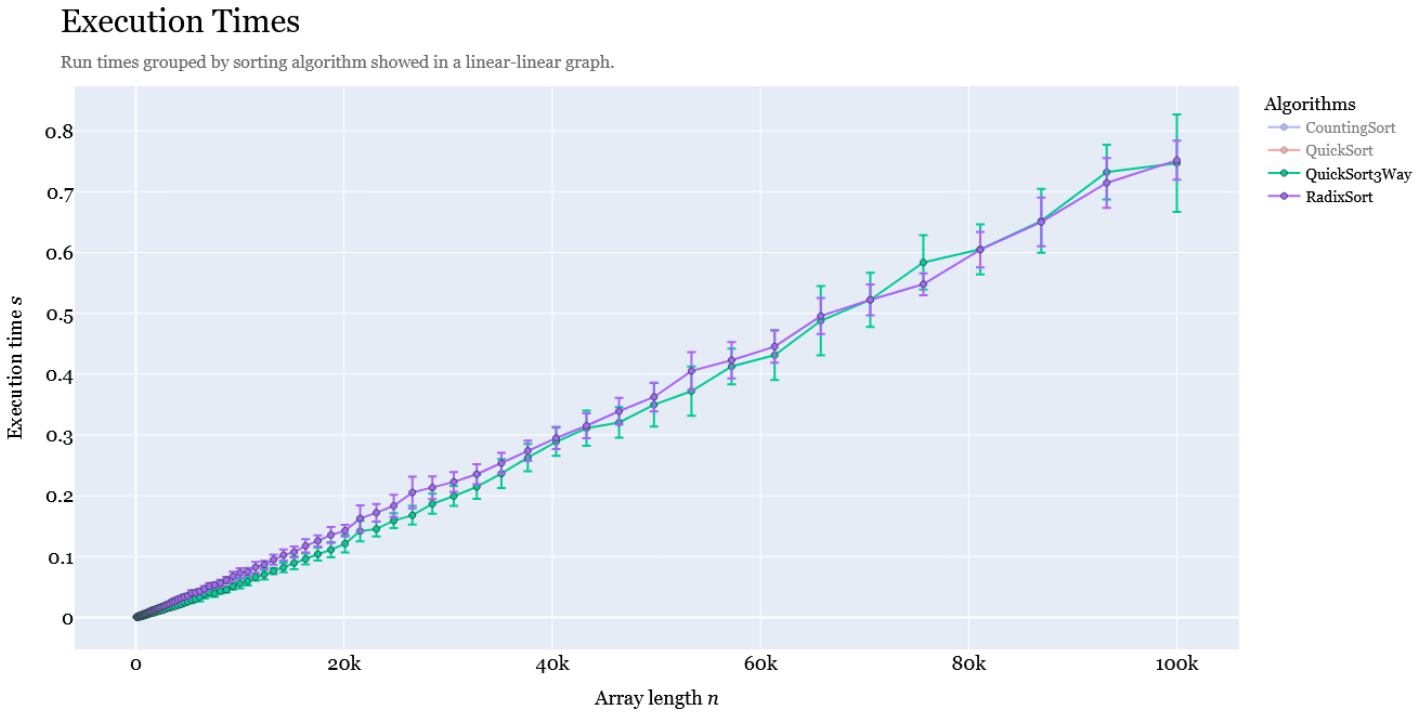
RadixSort:

RadixSort è forse l'algoritmo che stupisce di più, in quanto ci si aspetterebbe un andamento più simile a quello di CountingSort. La complessità teorica risulta però lineare (o quasi) solo nel caso in cui d (numero massimo di cifre degli elementi dell'array) è costante e k (elemento massimo dell'array) è O(n).

Invece soprattutto nella prima metà del grafico, con array di lunghezza più moderata, si nota come RadixSort abbia una complessità addirittura maggiore dei due algoritmi di QuickSort e QuickSort3Way. Questo è dovuto agli alti valori degli elementi massimi k (fino a 1.000.000) rispetto alla lunghezza degli array(da 100 a 100.000), che oltre ad aumentare, come abbiamo visto, la complessità del CountingSort utilizzato all'interno del codice di RadixSort per ordinare ogni cifra, aumenta anche il numero di cifre sulle quali RadixSort itera CountingSort, aumentando così la complessità totale.

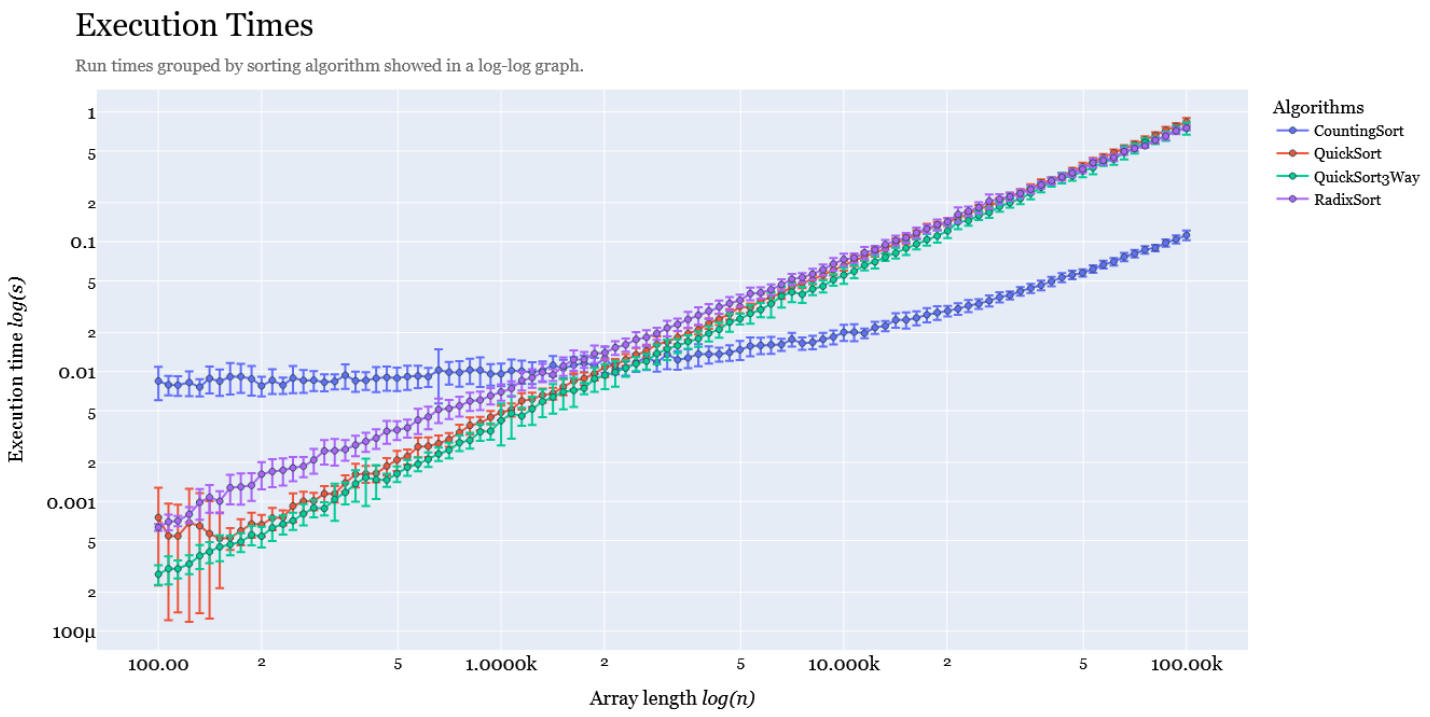
Altro fattore che aumenta il costo totale è sicuramente l'overhead dell'algoritmo, dovuto ad esempio al fatto che ad ogni passaggio RadixSort copia tutto l'array.

Solo nella parte finale del grafico, in cui gli array hanno lunghezza maggiore (circa da n=65k), RadixSort appare a tratti leggermente più efficiente di QuickSort3Way (come è possibile vedere nel grafico sottostante), proprio per l'aumento di n rispetto a k che invece rimane sempre variabile da 100 a 1.000.000.



Osservazioni su array di piccole dimensioni

Interessante per l'analisi risulta essere la prima parte del grafico, quella con array di lunghezza più contenuta. Per analizzare quindi la situazione attingiamo al grafico in scala logaritmica comprendente tutti gli algoritmi.



Nel grafico si può osservare l'overhead iniziale del CountingSort che risulta più lento con array di dimensione più piccola. Questo è dovuto al fatto che CountingSort utilizza un vettore ausiliario di lunghezza $k = max(A)$ per memorizzare le occorrenze degli elementi del vettore da riordinare quindi all'inizio la differenza tra la dimensione dell'array e l'elemento massimo può essere molto grande arrivando a richiedere un tempo di inizializzazione del conteggio tsle che supera il tempo di ordinamento effettivo.

Anche RadixSort con array di piccole dimensioni è più lento rispetto ai due QuickSort questo perché sfrutta sempre l'algoritmo CountingSort quindi in parte eredita la stessa problematica descritta precedentemente, tuttavia appare comunque più veloce di CountingSort in quanto RadixSort lo utilizza più volte ma su porzioni più piccole del dominio dei valori, limitando quindi il costo dell'inizializzazione dell'array di conteggio e relegando l'overhead complessivo ad essere più proporzionato ad n (lunghezza array) rispetto a k (elemento massimo dell'array).

Anche QuickSort nell'analisi sugli array più corti presenta un leggero picco anomalo, per poi allinearsi invece con in QuickSort3Way. Questo lieve picco è causato con molta probabilità dalla scelta di un pivot troppo sbilanciato, cosa che è più facile che accada con array di piccole dimensioni, che porta ad un overhead dell'algoritmo maggiore in quanto saranno necessari più chiamate ricorsive e più swap di posizione. Anche in questo si differenzia da QuickSort3Way che gestisce meglio i casi limite ed evita partizioni non necessarie.
Successivamente i due algoritmi si allineano in quanto con l'aumento della lunghezza degli array le partizioni diventano statisticamente più equilibrate.

SCRIVERE ROBA + GRAFICO SULLE VARIABILITà,

SCRIVERE COME FUNZIONE COUNTINGSORT2 SU RADIXSORT

SCRIVERE CONCLUSIONI

AGGIUNGERE SE NECESSARIO ALTRE DESCRIZIONI DI LIBRERIE USATE