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 e Variabilità

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.

Strumenti

- 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, quali:
  - **numpy** libreria fondamentale per il calcolo scientifico con grandi quantità di dati numerici e con alte prestazioni nel lavoro con gli array
  - **time** per la misurazione delle prestazioni
  - **plotly** per la visualizzazione dei dati tramite grafici interattivi
  - **matplotlib** per l'iniziale visualizzazione grezza degli andamenti temporali

- 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.

Alcune specifiche aggiuntive

Tempo minimo misurabile

Durata minima di un'operazione che possiamo misurare con sufficiente accuratezza, in modo che l'errore relativo massimo non superi un certo valore (in questo progetto è stato deciso essere pari a E = 0.001, cioè 0.1%).

Tramite questa formula è possibile determinare il tempo minimo misurabile:

$$T_{min} = R \cdot \left(\frac{1}{E} + 1\right)$$

Dove:

- E = errore relativo massimo ammissibile
- R = risoluzione del clock di sistema

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).

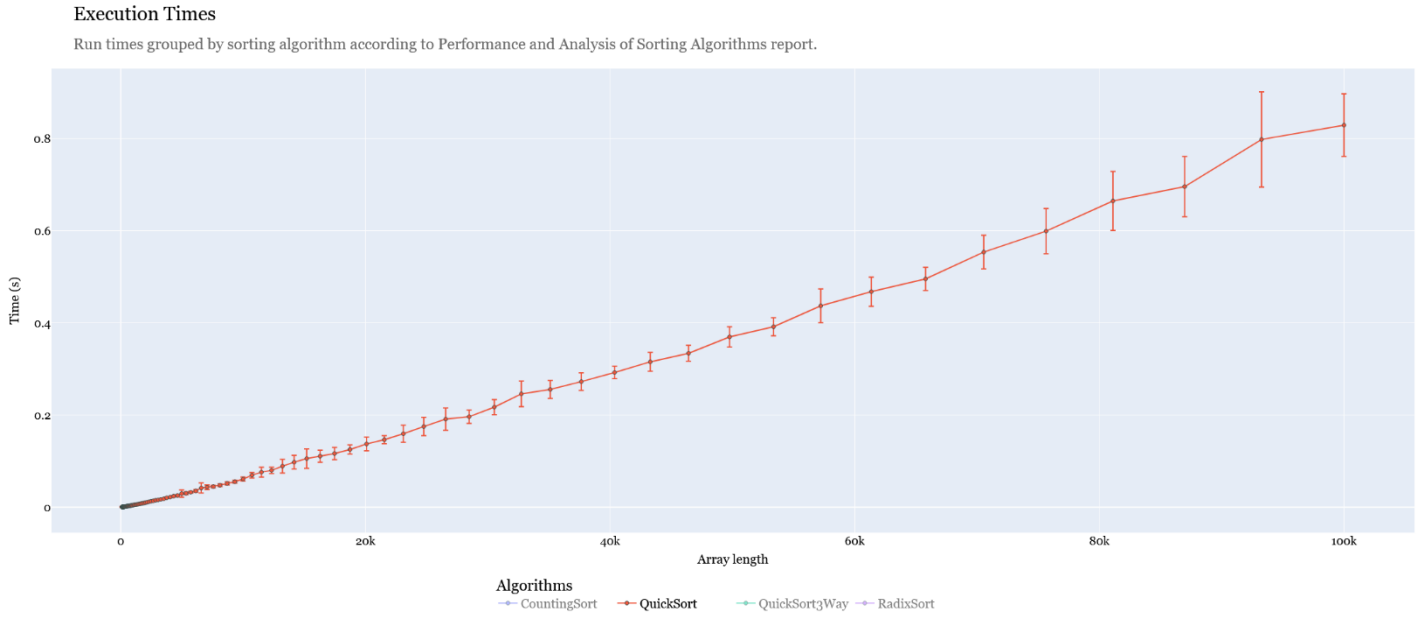
Svantaggi

- 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 >) lo suddivide in tre porzioni: valori minori, valori uguali, valori maggiori rispetto al pivot.

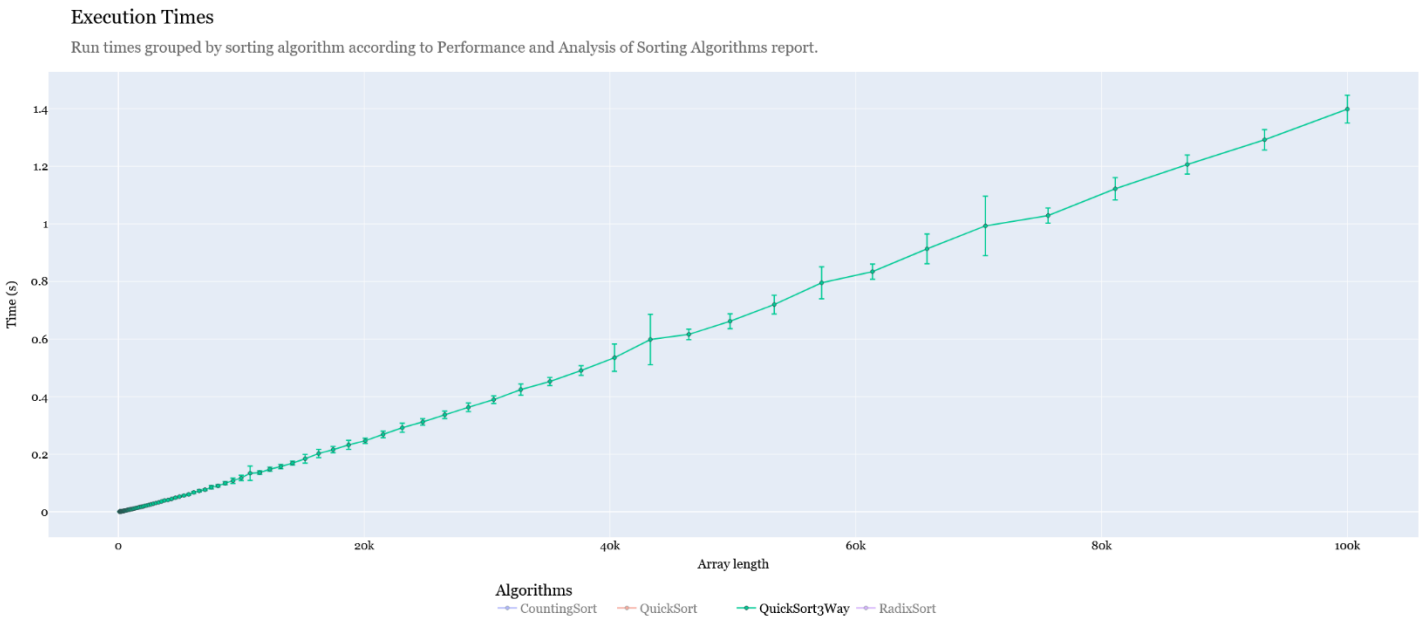
Vantaggi

- 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 $\boldsymbol{h}$ elementi **distinti:**
  - Caso migliore: $\Theta(n * h)$
  - Caso medio: $\Theta(nlogn)$
- Caso migliore/medio: $O(nlogn)$
- 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: $O(n)$;
- 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.

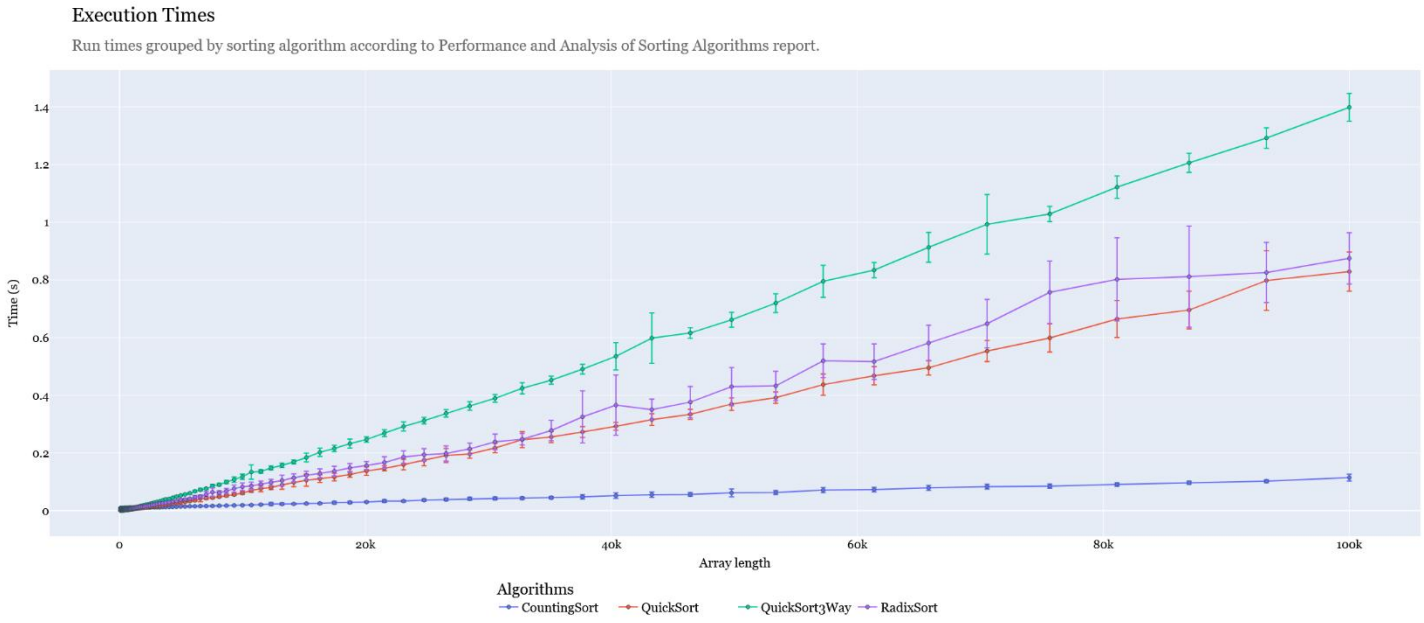
Grafico Lineare:

Analisi dei tempi di esecuzione in funzione della mole di dati

Osservazioni generali (scala lineare)

Per analizzare le tempistiche di risoluzione degli algoritmi descritti in precedenza, in funzione della quantità di dati, sono stati generati, in 100 punti tra 100 e 100.000 sull'asse delle ascisse (che rappresenta la lunghezza n degli array), 60 campioni per ciascun punto per stimare con maggior precisione la vera media temporale di esecuzione. Ognuno di questi array è popolato da valori interi casuali compresi tra 100 e 1.000.000.

Di seguito il grafico con gli andamenti temporali medi di tutti e quattro gli algoritmi:



Andiamo a vedere più nel dettaglio le varie differenze tra gli algoritmi confrontando tra loro quelli con tempi di esecuzione simili.

CountingSort

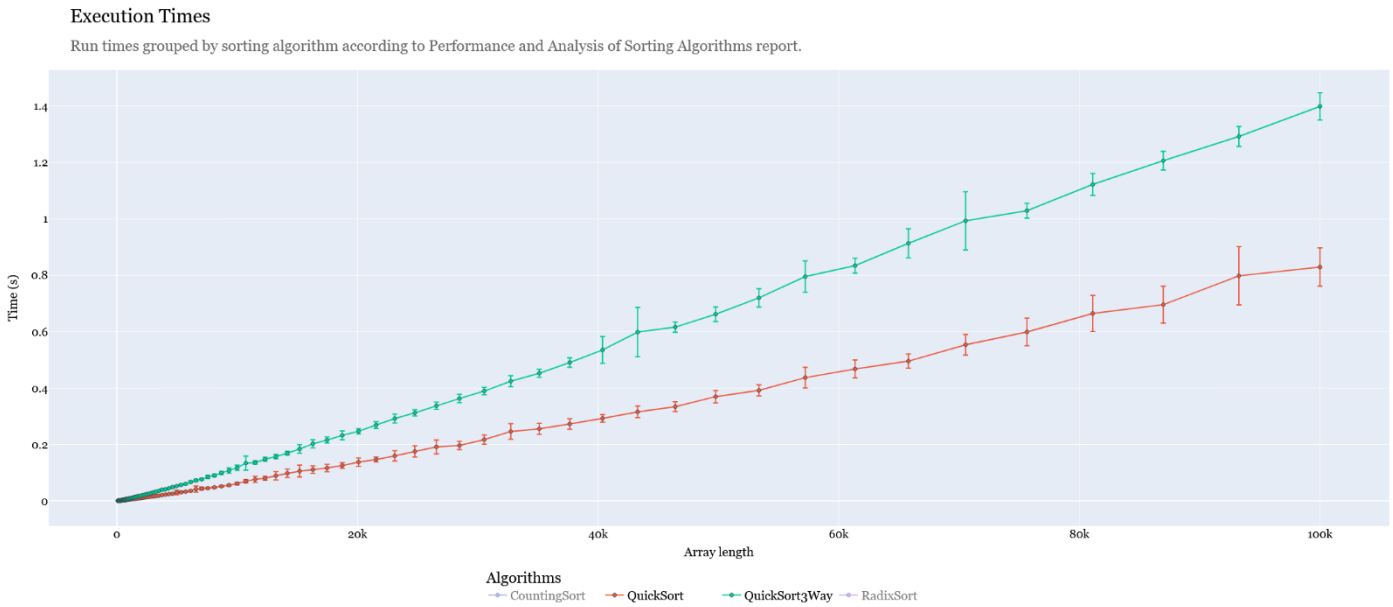
CountingSort è sicuramente l'algoritmo più interessante in questa analisi in quanto si distacca maggiormente dagli altri.
Presenta un andamento quasi costante come ci si poteva aspettare dalle caratteristiche descritte in precedenza. Possedendo infatti una complessità di $O(n+k)$, il distacco dall'andamento 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, che hanno quindi più probabilità di contenere numeri molto grandi (k massimo 1.000.000) aumentino i tempi di esecuzione dell'algoritmo.

Quando k è asintoticamente maggiore di n, ovvero $k \in \omega(n)$, CountingSort non si comporta più come un algoritmo lineare.

QuickSort & QuickSort3Way

Per array di grandi dimensioni si notano significative divergenze tra i due algoritmi, il QuickSort3Way non riesce a gestire array di grosse dimensioni tanto bene quanto la sua versione base QuickSort, presentano però entrambi una crescita abbastanza regolare molto simile nel tempo, coerente con la loro complessità media$O(n \cdot log(n))$.

Il distacco 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, se non rompe, l'ottimizzazione nella partizione introdotta da QuickSort3Way.



RadixSort & QuickSort

RadixSort è forse l'algoritmo che stupisce di più, in quanto ci si aspetterebbe un andamento più simile a quello di CountingSort, dato lo utilizza come subroutine. La complessità teorica risulterebbe infatti 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).

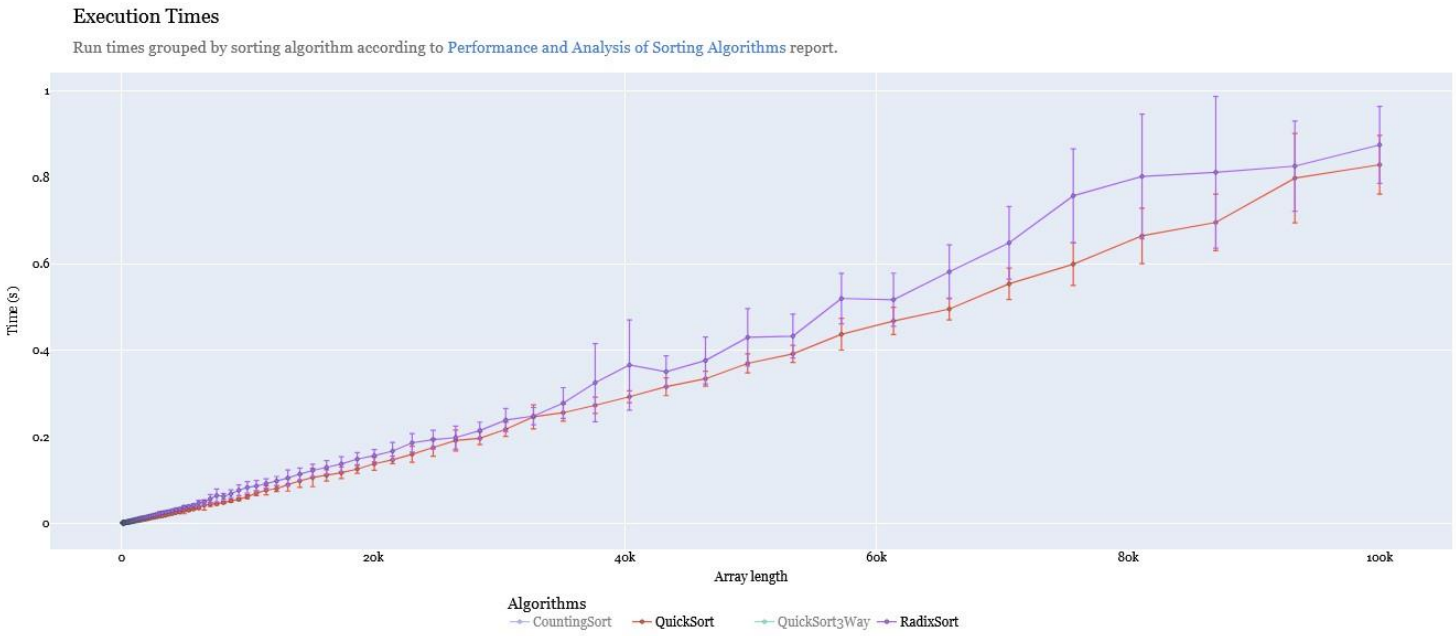
Invece, soprattutto nella prima metà del grafico, con array di lunghezza più moderata, emerge come RadixSort abbia una complessità addirittura peggiore di QuickSort, la quale è $O(n \cdot log(n))$.

Questo apparente paradosso si spiega considerando che i parametri `$d$` e `$k$` non sono trascurabili nel range testato: per array con valori fino a 10^5, il numero di cifre in base 10 è già 5, e ogni passaggio di RadixSort comporta una copia completa dell'array, generando un overhead significativo. Per comprendere meglio quanto detto, confrontiamo le due complessità assumendo $d * k$ trascurabile:

$$\frac{d * k}{n * log_{10}(n)} = \frac{d}{log_{10}(n)}$$

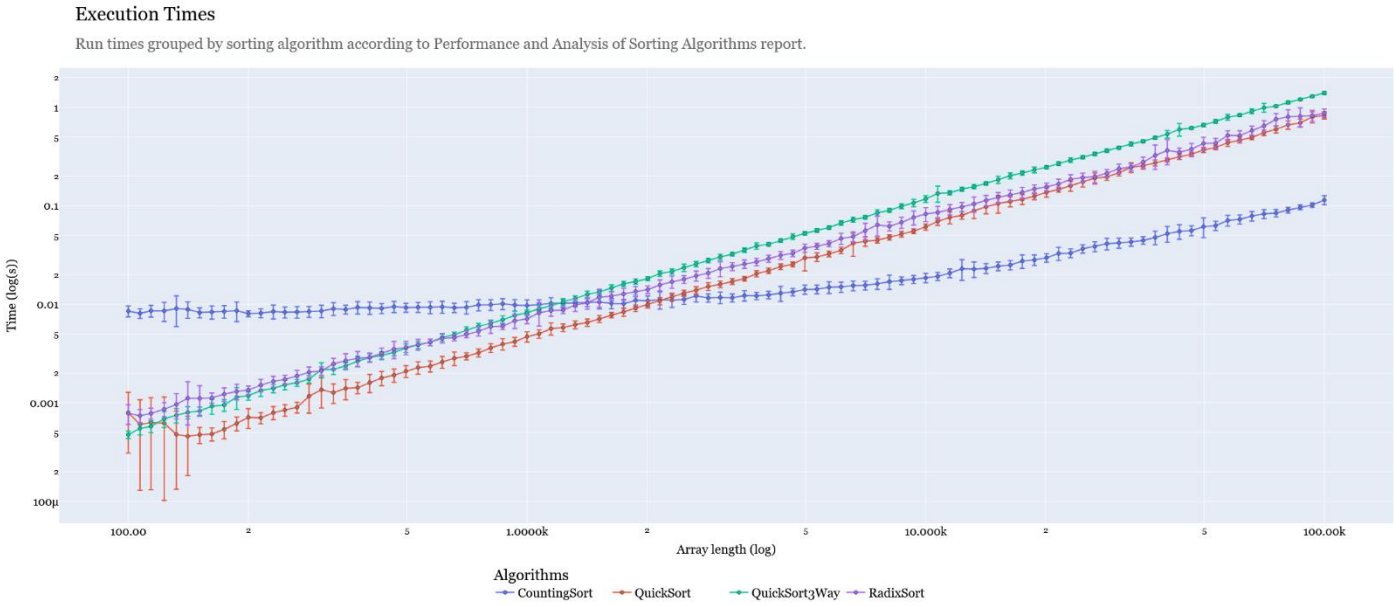
Questo soprattutto mostra che RadixSort inizierà a diventare più efficiente solo per $d \ll log_{10}(n)$. Nel nostro contesto, essendo $d = 5$, tale vantaggio si manifesterà per $n \gg 10^5 = 100{\cdot}000$

Altro fattore che aumenta il costo totale è sicuramente l'overhead dell'algoritmo, per via delle operazioni di copiatura.



Osservazioni su array di piccole dimensioni (scala logaritmica)

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 che supera il tempo di ordinamento effettivo.

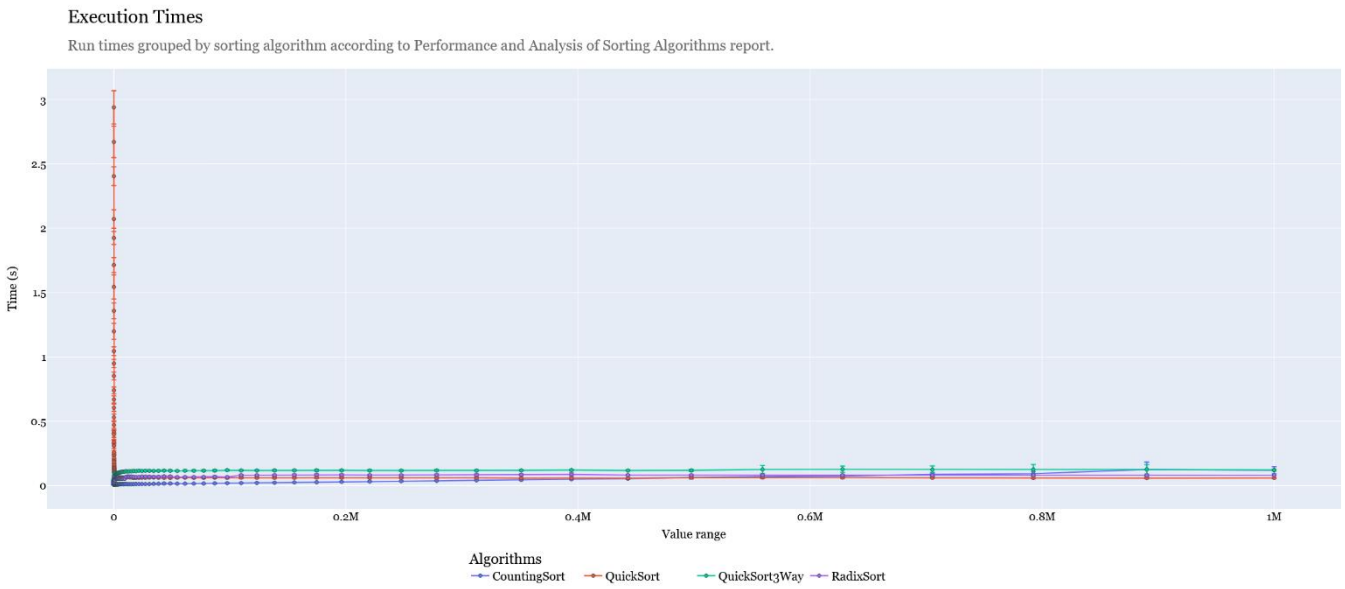
Anche RadixSort con array di piccole dimensioni è più lento rispetto ai due QuickSort questo perché sfrutta sempre l'algoritmo CountingSort quindi in parte eredita la stessa problematica descritta precedentemente, tuttavia appare comunque più veloce di CountingSort in quanto RadixSort lo utilizza più 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 porsi in parallelo a 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.

Analisi dei tempi di esecuzione in funzione della variabilità

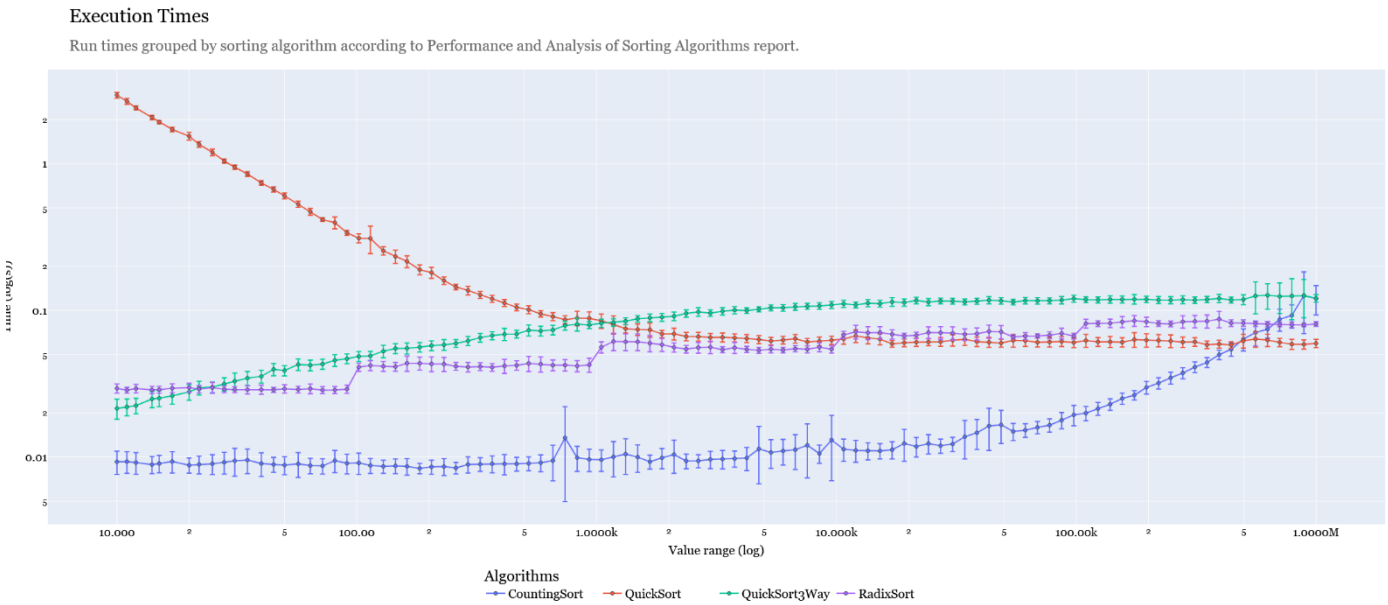
Per studiare il comportamento degli algoritmi rispetto alla ripetitività dei dati, riportiamo un grafico che mostra i tempi di esecuzione al variare del numero di valori distinti presenti negli array. In questa analisi, tutti gli array hanno la stessa lunghezza, mentre l'unica variabile è l'entropia dei dati, ovvero l'ampiezza dell'intervallo di numeri possibili. L'asse delle ascisse rappresenta quindi il numero massimo possibile di valori distinti presenti in ciascun array.

Osservazioni generali (scala lineare)



Da quello che si può vedere, siccome la lunghezza fissa degli array non implica un notevole svantaggio nell'esecuzione degli ordinamenti dopo una certa soglia di numeri distinti nell'array, passiamo alla scala logaritmica con un occhio di riguardo verso l'origine del grafico.

Osservazioni generali (scala logaritmica)



QuickSort & QuickSort3Way

Dal grafico in scala logaritmica si può osservare più dettagliatamente una caratteristica del QuickSort: risulta essere sensibile alla bassa variabilità dei numeri presenti negli array, questo sicuramente dovuto al fatto che una tale scarsità di elementi distinti aumenti la probabilità di scegliere un pivot svantaggioso.
Questo problema viene risolto nel QuickSort3Way tramite l'implementazione di un Partition modificato appositamente per gestire gli elementi duplicati, riducendo numero di confronti e scambi. Utilissimo se ci si trova a dover ordinare array con molte ripetizioni mantenendo le stesse capacità di Quicksort ma con tempi di esecuzione nettamente migliori.

CountingSort & RadixSort

A prima vista sembrerebbe che il RadixSort sia più lento del CountingSort anche se riutilizza il suo stesso codice, analizziamo il motivo:

Overhead di memoria e di inizializzazione

- **CountingSort** alloca due strutture ausiliarie ad ogni chiamata:
  - Un array c di lunghezza k, dove k è il valore massimo intero presente nell'array (può arrivare fino a 10^6).
  - Un array B di output di lunghezza pari a quella dell'array da ordinare (n=10^4).
- **RadixSort** invoca internamente una versione modificata di CountingSort, il quale alloca ogni volta:
  - Un array c di lunghezza fissa k=10 (per il digit corrente, da 0 a 9).
  - Un array B di output di lunghezza n.

Utilizzando RadixSort, ad ogni passo di ordinamento per cifra vengono allocati nuovamente quei due array. Se l'intervallo massimo dei valori è 10^6, bastano 6 cifre per coprire tutti i numeri da 0 a 999 999; pertanto RadixSort esegue al massimo 6 passaggi di CountingSort (una volta per ogni cifra), riempiendo e azzerando l'array `c` di lunghezza 10 e riallocando l'array `B` di lunghezza `n`.

Complessità aggiuntiva per passaggi non necessari

- **CountingSort**, l'array dei conteggi (c) deve sempre essere inizializzato a tutti zero per tutti i valori da 0 a k−1, anche se nel dataset effettivamente appaiono pochissimi valori distinti. Di conseguenza, già solo con l'inizializzazione di un array di dimensione k, tanto maggiore di n, richiederebbe un tempo complessivo $\Theta(k)$, indipendentemente dalla reale variabilità dei dati, perdendo così ogni beneficio di questo algoritmo. (Vedasi ultimi punti del tracciato)
- **RadixSort**, l'algoritmo esegue comunque tutti i d passaggi (dove d = numero di cifre di max(A)) anche se l'array contiene molti valori ripetitivi.

Conclusioni

Dalle analisi effettuate possiamo constatare che l'algoritmo più opportuno da utilizzare varia a seconda delle caratteristiche dell'input.

Resoconto

- QuickSort è un algoritmo estremamente pratico per array medio-grandi soprattutto con dati numerici distribuiti in modo uniforme. Tuttavia, soffre pesantemente quando gli elementi presenti nell'array sono altamente ripetitivi.
- QuickSort3Way è particolarmente efficace quando i dati contengono estremamente tanti duplicati (soprattutto sotto i 20), tuttavia, in presenza di alta variabilità dei valori, il suo vantaggio si annulla, e il QuickSort classico risulta più veloce e performante grazie alla sua minore complessità strutturale.
- CountingSort offre prestazioni eccellenti per array grandi in cui il massimo elemento `k` resta contenuto rispetto a `n` (cioè $k \in O(n)$), comportandosi quasi linearmente. Tuttavia, se `k` cresce più rapidamente di `n` (cioè $k \in \omega(n)$), come avviene per array piccoli ma con ampio range di valori, l'inizializzazione dell'array ausiliario di dimensione `k` diventa un collo di bottiglia, annullando i vantaggi dell'algoritmo.
- RadixSort, che condivide parte della logica di CountingSort, è vantaggioso solo quando il numero massimo di cifre `d` è costante e contenuto rispetto a $log(n)$. Nei casi in cui `d` diventa non trascurabile (come con numeri fino a 1.000.000, dove d = 6), RadixSort accumula overhead (soprattutto per l'allocazione e la copia di array in ogni passaggio), risultando più lento persino di QuickSort.

Riferimento al progetto
Una versione interattiva dei risultati sperimentali, insieme al codice e alla documentazione completa, è consultabile all'indirizzo:
https://londero-lorenzo.github.io/SortingAlgorithms/