

# Analisi empirica degli algoritmi di ordinamento

Graziano Francesco <sup>1</sup> , Ongaro Michele <sup>2</sup> , Petri Riccardo <sup>3</sup> e Ungaro  
Marco <sup>4</sup>

Università degli Studi di Udine, Dipartimento di Matematica e  
Informatica

A.A. 2024-2025

<sup>1</sup>Email: graziano.francesco@spes.uniud.it, Matricola: 166680

<sup>2</sup>Email: ongaro.michele@spes.uniud.it, Matricola: 168049

<sup>3</sup>Email: petri.riccardo@spes.uniud.it, Matricola: 167623

<sup>4</sup>Email: ungaro.marco@spes.uniud.it, Matricola: 168934

# Indice

Introduzione	2
Counting Sort	3
Quick Sort	5
Quick Sort 3 Way	7
Radix Sort	8
Colusioni	9

# Introduzione

Il progetto richiede l'implementazione di quattro algoritmi di ordinamento per array interi di dimensioni variabili. Gli algoritmi che andremo ad analizzare sono il Counting Sort, il Quick Sort, il Quick Sort 3 way e il Radix Sort (algoritmo a scelta). Oltre alla corretta implementazione viene richiesto di effettuare un'analisi empirica dei tempi medi di esecuzione degli algoritmi al variare della dimensione dell'array e del range dei valori interi. Per stimare i tempi di esecuzione di questi algoritmi garantendo un errore relativo massimo pari a 0.001 adotteremo le seguenti metodologie:

- Utilizzeremo un clock di sistema monotono per garantire precisione nelle misurazioni (ad esempio, `perf_counter()` del modulo *time* in Python);
- Andremo a generare almeno 100 campioni per ciascun grafico, con i valori dei parametri (dimensione dell'array  $n$  e intervallo dei valori  $m$ ) distribuiti secondo una progressione geometrica;
- Effettueremo più esecuzioni per ogni campione, per stimare in modo affidabile il tempo medio di esecuzione e, eventualmente, il relativo errore.

Dopo aver stimato i tempi di esecuzione per ciascun algoritmo, risulterà interessante confrontare i grafici ottenuti per analizzare il comportamento degli algoritmi in diverse situazioni, come il caso peggiore, quello migliore o in quello medio.

Da tutto questo potremmo ottenere una verifica empirica dell'andamento asintotico dei tempi di esecuzione di ogni algoritmo.

# Counting Sort

Counting Sort è un algoritmo di ordinamento non comparativo, cioè che non ordina gli elementi confrontandoli tra loro come fanno invece altri algoritmi classici come QuickSort, Merge Sort o Bubble Sort. Invece, conta quante occorrenze di ciascun valore sono presenti nell'array da ordinare e utilizza queste informazioni per posizionare gli elementi nell'array ordinato. Passaggi principali che effettua l'algoritmo:

1. Determinazione del range: Identifica il valore massimo nell'array in input per determinare la dimensione necessaria dell'array di conteggio.
2. Conteggio delle occorrenze: Crea un array ausiliario (**count**) in cui ciascun indice rappresenta un valore possibile dell'array originale, e si conta quante volte ciascun valore appare.
3. Costruzione dell'array cumulativo: Trasforma l'array di conteggio in un array cumulativo, dove ogni elemento indica la posizione finale di un dato valore nell'array ordinato.
4. Costruzione dell'array ordinato: Itera sull'array originale (in genere in ordine inverso per mantenere la stabilità), e si posiziona ogni elemento nella posizione corretta dell'array di output, decrementando il valore corrispondente nell'array di conteggio.

L'algoritmo Counting Sort è particolarmente efficiente quando il range dei valori da ordinare è limitato rispetto alla dimensione dell'array.

## Analisi della complessità

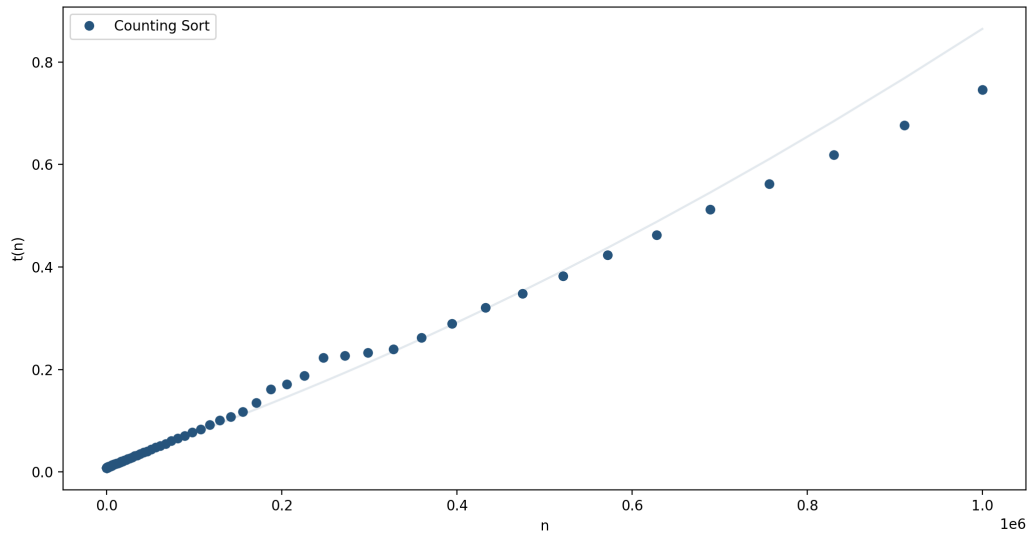
La complessità temporale di Counting Sort è  $O(n + k)$ , dove:

- $n$  è il numero di elementi nell'array da ordinare;
- $k$  è il valore massimo presente nell'array.

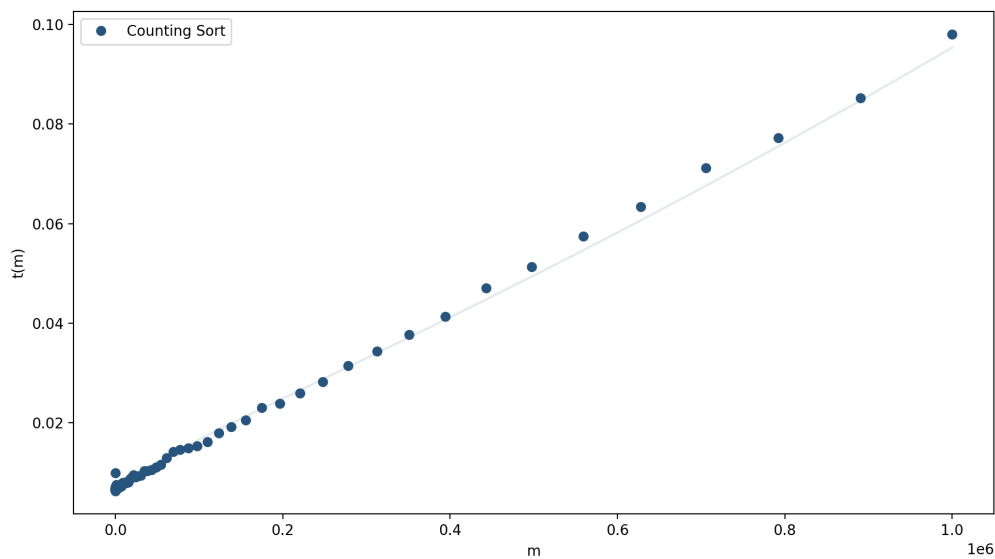
Nel caso in cui si ha  $k = O(n)$  allora la complessità diventa  $O(n)$

## Grafico dei tempi di esecuzione

Come prima cosa abbiamo generato il grafico in cui varia la lunghezza dell'array **n** mentre il range di valori che possono esserci nell'array rimane bloccato a 100000 ( $m = 100000$ ). Nel grafico qui sotto abbiamo come ascissa la variazione di **n** da 100 a 1000000 in scala scientifica dove ogni valore va moltiplicato per  $10^6$ , come ordinata abbiamo il tempo in secondi di riordinamento dell'array.



Poi abbiamo generato il grafico in cui varia il range di valori che possono esserci nell'array  $\mathbf{m}$  mentre la lunghezza dell'array rimane bloccata a 10000 ( $n = 10000$ ). Nel grafico qui sotto abbiamo come ascissa la variazione di  $\mathbf{m}$  da 10 a 1000000 in scala scientifica, dove ogni valore va moltiplicato per  $10^6$ , e come ordinata abbiamo il tempo in secondi di riordinamento dell'array.



# Quick Sort

Quick Sort è un algoritmo di ordinamento basato sulla tecnica del "divide et impera". L'idea principale è quella di selezionare un elemento pivot e partizionare l'array in due sottosequenze: gli elementi minori del pivot e quelli maggiori. Passaggi principali che effettua l'algoritmo:

1. Si sceglie un elemento **pivot** dall'array.
2. Si riordinano gli elementi in modo tale che tutti quelli minori del **pivot** lo precedano e quelli maggiori lo seguano.
3. Si applica **ricorsivamente** Quick Sort alle due sottosequenze.

Quick Sort è molto efficiente nella pratica, anche se nel caso peggiore ha una complessità quadratica.

## Analisi della complessità

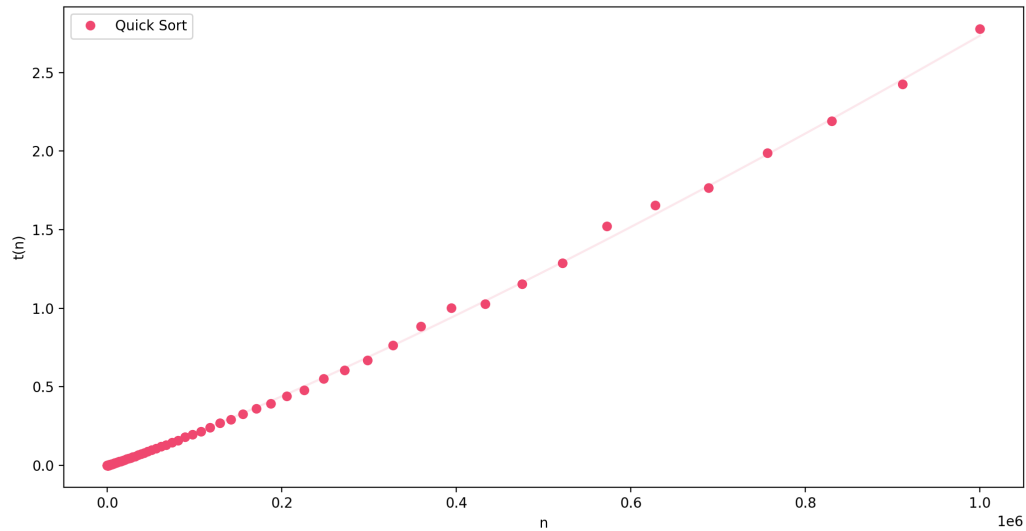
La complessità temporale di Quick Sort è:

- Caso medio:  $O(n \log n)$
- Caso peggiore:  $O(n^2)$
- Caso migliore:  $O(n \log n)$

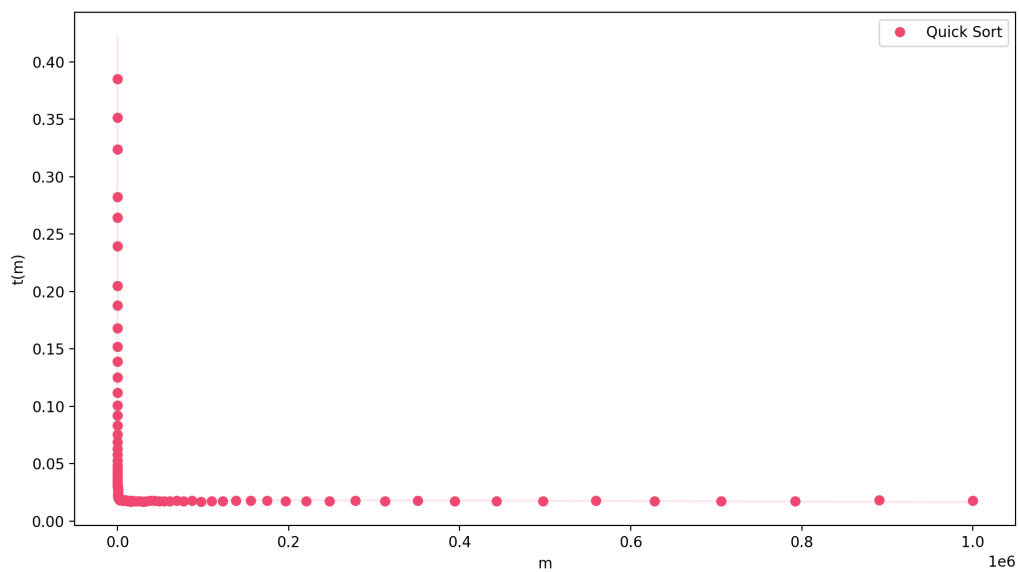
Nella pratica, il caso peggiore si verifica raramente se si utilizza un buon criterio di scelta del pivot (es. pivot casuale o mediana di tre).

## Grafico dei tempi di esecuzione

Abbiamo generato un grafico in cui varia la lunghezza dell'array **n** da ordinare, mentre il parametro **m** resta fisso. L'ordinamento viene eseguito con Quick Sort su array contenenti numeri casuali. Come ascissa abbiamo **n** da 100 a 1000000, e come ordinata il tempo di esecuzione in secondi.



Successivamente, abbiamo realizzato un grafico in cui varia l'intervallo dei valori possibili presenti nell'array, ( $\mathbf{m}$ ), mantenendo costante la sua lunghezza a 10000 ( $n = 10000$ ). Nel grafico riportato qui sotto, sull'asse delle ascisse è rappresentata la variazione di  $\mathbf{m}$  da 10 a 1000000 in scala scientifica (ogni valore va moltiplicato per  $10^6$ ), mentre sull'asse delle ordinate è riportato il tempo di ordinamento dell'array in secondi.



Qui si riscontra una problema nel grafico quando il numero di elementi possibile dell'array è molto basso, bisognerà indagare il motivo.

# Quick Sort 3 Way



# Radix Sort

# Conclusioni