

# 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

# Contenuti

<b>1</b>	<b>Introduzione</b>	<b>2</b>
<b>2</b>	<b>Counting Sort</b>	<b>3</b>
2.1	Analisi della complessità . . . . .	3
2.2	Analisi empirica . . . . .	4
2.3	Grafico dei tempi di esecuzione . . . . .	4
<b>3</b>	<b>Quick Sort</b>	<b>6</b>
3.1	Analisi della complessità . . . . .	6
3.2	Analisi empirica . . . . .	6
3.3	Grafico dei tempi di esecuzione . . . . .	7
<b>4</b>	<b>Quick Sort 3 Way</b>	<b>9</b>
4.1	Analisi della complessità . . . . .	9
4.2	Analisi empirica . . . . .	10
<b>5</b>	<b>Radix Sort</b>	<b>11</b>
5.1	Analisi della complessità . . . . .	11
5.2	Analisi empirica . . . . .	11
<b>6</b>	<b>Conclusioni</b>	<b>13</b>

# Capitolo 1

## 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.

# Capitolo 2

## Counting Sort

Il Counting Sort è un algoritmo di ordinamento non comparativo: anziché effettuare confronti tra gli elementi, si basa sul conteggio delle occorrenze di ciascun elemento presente nell'array da ordinare. È particolarmente efficiente quando gli elementi da ordinare sono numeri interi compresi in un intervallo limitato (intervallo  $[0, k]$ ). L'implementazione adottata si articola in tre fasi principali:

- Conteggio delle occorrenze di ciascun elemento; si costruisce un array ausiliario  $C$  di lunghezza  $k + 1$ , inizializzato a zero, in cui per ogni elemento  $x$  in  $A$ , si incrementa  $C[x]$  di 1.
- Calcolo delle posizioni cumulative; per ottenere la posizione finale di ciascun elemento nell'array ordinato, si trasforma  $C$  in un array cumulativo. In questa versione,  $C[0]$  viene inizialmente decrementando di uno, così che  $c[i]$  rappresenti l'indice massimo in cui si può inserire l'elemento  $i$  nell'array ordinato.
- Costruzione dell'array ordinato; si scorre l'array originale da destra a sinistra, e si inserisce ciascun elemento  $x$  nella posizione  $C[x]$  dell'array ordinato, decrementando  $C[x]$  di 1 dopo ogni inserimento. Questo garantisce la stabilità dell'ordinamento, poiché gli elementi con lo stesso valore vengono inseriti nell'ordine in cui appaiono nell'array originale.

Le due funzioni fornite nel codice sono:

- `countingSort(A, B, k)`: scrive l'output ordinato in  $B$ ;
- `uniformedCountingSort(A, k)`: funzione wrapper che restituisce direttamente una copia ordinata di  $A$ .

### 2.1 Analisi della complessità

Siano  $n$  la dimensione dell'array  $A$  e  $k$  il valore massimo contenuto in  $A$ .

**Tempo:**

- Conteggio:  $O(n)$  per scorrere l'array  $A$  e contare le occorrenze di ciascun elemento.
- Calcolo delle posizioni cumulative:  $O(k)$  per trasformare l'array  $C$  in un array cumulativo.

- Costruzione dell'array ordinato:  $O(n)$  per scorrere l'array  $A$  e inserire gli elementi nell'array ordinato  $B$ .
- Totale:  $O(n + k)$ .

**Spazio:**

- Array ausiliario  $C$ : richiede  $O(k)$  spazio.
- Array ordinato  $B$ : richiede  $O(n)$  spazio.
- Totale:  $O(n + k)$ .

L'algoritmo è efficiente quando  $k = O(n)$ , ovvero quando il range massimo dei valori interi è proporzionale alla dimensione dell'array. In scenari in cui  $k \gg n$ , il costo della fase di conteggio e l'allocazione dell'array ausiliario  $C$  possono rendere l'algoritmo meno competitivo rispetto ad altri metodi di ordinamento.

## 2.2 Analisi empirica

Sono stati condotti due esperimenti distinti per analizzare le prestazioni empiriche del Counting Sort:

- Nel primo esperimento, la dimensione dell'array  $n$  varia, mentre il valore massimo  $k$  è mantenuto fisso a 100000.
- Nel secondo esperimento,  $n$  è fissato a 10000, e viene fatto variare  $m$ .

In entrambi i casi, i parametri sono stati scelti seguendo una progressione geometrica. Per ogni valore di  $n$  o  $m$  sono stati generati almeno 100 campioni casuali, ciascuno eseguito più volte per stimare in modo affidabile il tempo medio di esecuzione, mantenendo un errore relativo massimo  $\leq 0.001$ . Le misurazioni sono state effettuate utilizzando un clock monotono ad alta precisione (`time.perf_counter()` in Python).

I risultati ottenuti sono in linea con le aspettative teoriche: Nel primo grafico (a  $k$  fisso), il tempo cresce linearmente con  $n$ , coerentemente con la complessità  $O(n + m)$ . Nel secondo grafico (a  $n$  fisso), il tempo cresce linearmente con  $m$ , evidenziando l'impatto della fase di conteggio e dell'allocazione dell'array  $C$ . I due grafici risultanti mostrano visivamente come il Counting Sort si comporti in relazione ai due parametri chiave  $n$  e  $m$ .

## 2.3 Grafico dei tempi di esecuzione

In primo luogo, è stato analizzato l'andamento temporale al variare della dimensione dell'array  $n$ , mantenendo costante l'intervallo dei valori a  $k = 100000$ . Il grafico seguente mostra in:

- **Ascissa:** dimensione  $n$  dell'array (scala logaritmica,  $10^2 \leq n \leq 10^6$ ).
- **Ordinata:** tempo di esecuzione (secondi) per l'ordinamento.

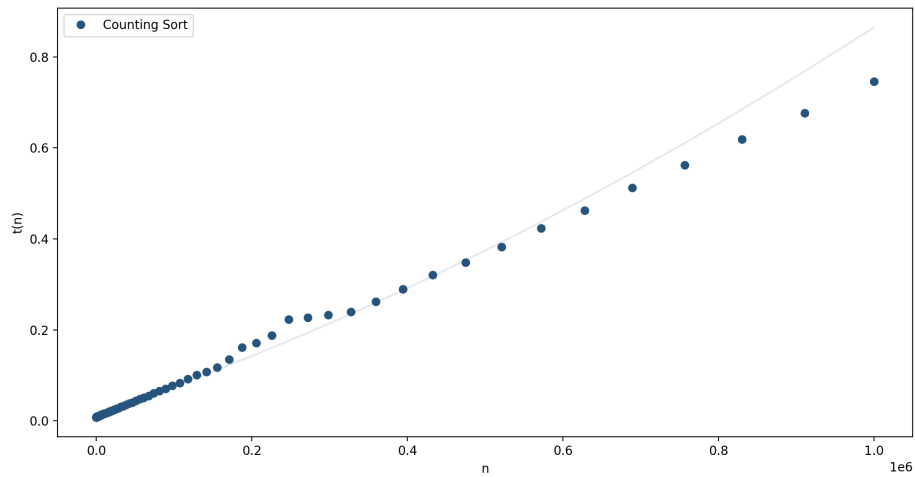


Figura 2.1: Performance del Counting Sort al variare di  $n$ .

Successivamente, è stato analizzato l'andamento temporale al variare dell'intervallo di valori  $m$ , mantenendo costante la dimensione dell'array a  $n = 10000$ . Il grafico seguente presenta:

- Ascissa: intervallo  $m$  dei valori (scala logaritmica,  $10^1 \leq m \leq 10^6$ )
- Ordinata: tempo di esecuzione (secondi) per l'ordinamento

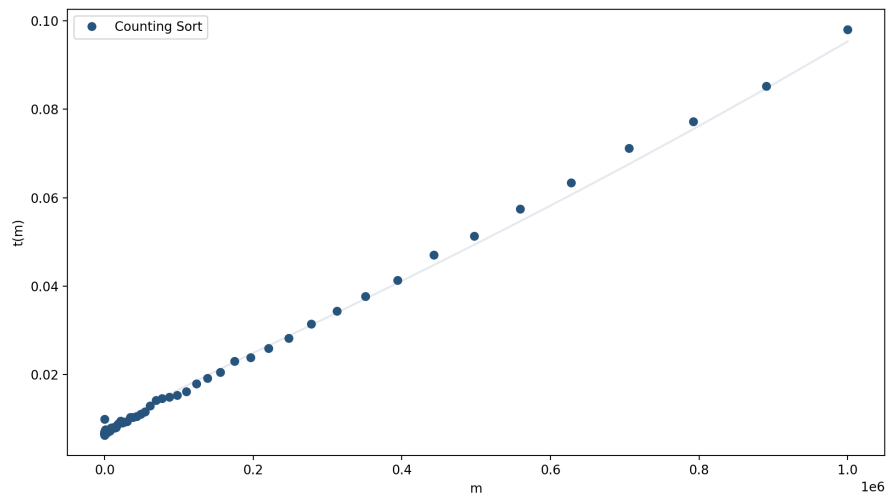


Figura 2.2: Performance del Counting Sort al variare di  $m$ .

# Capitolo 3

## Quick Sort

Il Quick Sort è un algoritmo di ordinamento basato sulla strategia del divide et impera (ovvero: dividere il problema in sottoproblemi più piccoli, risolverli ricorsivamente e combinare i risultati). A partire da un array  $A$ , si seleziona un pivot, in questo caso l'elemento finale del sottoarray  $A[p...q]$ , e si procede alla partizione: tutti gli elementi minori o uguali al pivot vengono spostati a sinistra, quelli maggiori a destra. L'indice finale del pivot è restituito dalla funzione `partition`, e l'algoritmo viene poi richiamato ricorsivamente sulle due sottosequenze risultanti.

L'implementazione utilizzata è in-place (cioè non richiede spazio aggiuntivo proporzionale all'input), ma non stabile (l'ordine relativo tra elementi uguali può cambiare).

La funzione `uniformedQuickSort` è una versione ausiliaria pensata per rendere uniforme l'interfaccia tra gli algoritmi testati: riceve come parametro anche  $k$  (massimo valore degli elementi), che tuttavia non viene utilizzato in questo caso, in quanto Quick Sort non dipende dal dominio dei valori.

### 3.1 Analisi della complessità

Sia  $n$  la dimensione dell'array da ordinare:

- Caso medio:  $O(n \cdot \log(n))$ , si verifica quando il pivot divide l'array in modo bilanciato.
- Caso migliore:  $O(n \cdot \log(n))$ , con partizioni esattamente simmetriche.
- Caso peggiore:  $O(n^2)$ , quando il pivot è sempre il minimo o il massimo elemento (partizione altamente sbilanciata).
- Spazio ausiliario:  $O(\log(n))$  nel caso medio per la profondità dello stack ricorsivo;  $O(n)$  nel caso peggiore.

A differenza di altri algoritmi come Counting Sort o Radix Sort, Quick Sort non richiede conoscenza del range dei valori interi e lavora unicamente tramite confronti tra elementi.

### 3.2 Analisi empirica

Per analizzare empiricamente il comportamento del Quick Sort sono stati eseguiti due esperimenti distinti:

- Esperimento 1: variazione della dimensione dell'array  $n$ , mantenendo fisso  $m = 100000$ .
- Esperimento 2:  $n = 10000$  costante, con variazione di  $m$ .

In entrambi i casi, i parametri sono distribuiti secondo una progressione geometrica. Per ogni coppia  $(n, m)$  sono stati generati almeno 100 campioni casuali, ciascuno eseguito più volte per stimare in modo accurato il tempo medio di esecuzione, con un errore relativo massimo  $\leq 0.001$ . Le misurazioni sono state effettuate mediante un clock monotono ad alta precisione (`time.perf_counter()` in Python).

Poiché Quick Sort opera esclusivamente tramite confronti, il valore massimo  $m$  non influisce direttamente sulla complessità. Tuttavia, può avere effetti indiretti sulla distribuzione dei dati (ad esempio, la presenza di molti duplicati o valori ripetuti), il che può influenzare l'efficienza della partizione.

I risultati ottenuti mostrano che:

- Nel primo grafico, i tempi di esecuzione crescono come  $O(n \log n)$ , in linea con il comportamento teorico nel caso medio.
- Nel secondo grafico, la variazione di  $m$  non produce cambiamenti significativi nei tempi di esecuzione, a conferma del fatto che l'algoritmo non dipende dal dominio numerico.

### 3.3 Grafico dei tempi di esecuzione

È stato generato un grafico in cui varia la lunghezza dell'array da ordinare, indicata con  $n$ , mentre il parametro  $m$  (range dei valori) è mantenuto costante. L'ordinamento è stato eseguito utilizzando l'algoritmo Quick Sort su array contenenti numeri casuali. Sull'asse delle ascisse è riportato il valore di  $n$ , da 100 a 1.000.000, mentre sull'asse delle ordinate è rappresentato il tempo medio di esecuzione, espresso in secondi.

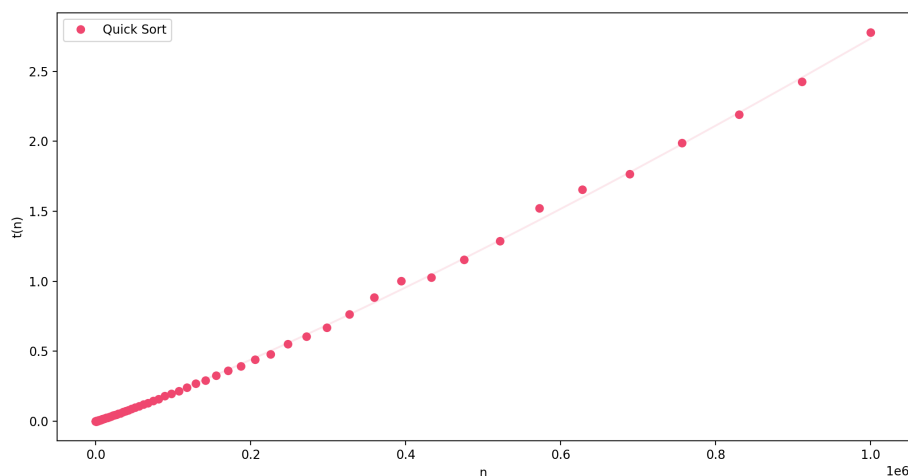


Figura 3.1: Performance del Quick Sort al variare di  $n$ .



Successivamente, è stato realizzato un grafico in cui varia il range dei valori interi presenti nell'array, indicato con  $m$ , mantenendo costante la dimensione dell'array a  $n = 10000$ . Nel grafico riportato di seguito, sull'asse delle ascisse è rappresentata la variazione di  $m$ , da 10 a 1.000.000, in scala scientifica. Sull'asse delle ordinate è riportato il tempo medio di esecuzione dell'ordinamento, espresso in secondi.

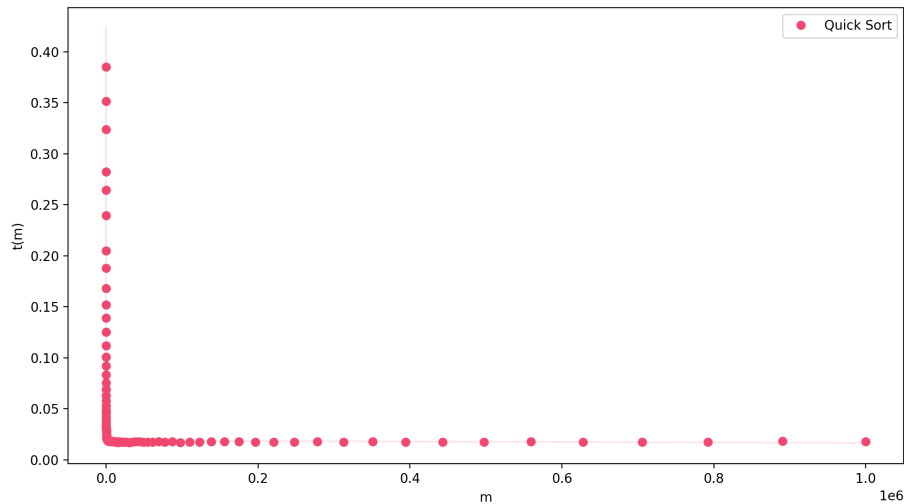


Figura 3.2: Performance del Quick Sort al variare di  $m$ .

Si osserva un'anomalia nel grafico quando il numero di valori distinti presenti nell'array ( $m$ ) è molto basso. Sarà opportuno approfondire l'origine di questo comportamento, che potrebbe essere legato alla scarsa diversità dei dati e al conseguente degrado delle prestazioni dell'algoritmo.

# Capitolo 4

## Quick Sort 3 Way

Il Quick Sort 3-Way è una variante del Quick Sort classico, progettata per migliorare l'efficienza in presenza di molti elementi duplicati. Come il Quick Sort tradizionale, segue la strategia del divide et impera (cioè suddividere l'array in parti più semplici da ordinare ricorsivamente). A differenza della versione classica, che suddivide in due partizioni (elementi minori o maggiori del pivot), il Quick Sort 3-Way suddivide l'array in tre sezioni distinte:

- Elementi minori del pivot;
- Elementi uguali al pivot;
- Elementi maggiori del pivot.

Questa suddivisione viene effettuata durante la fase di partizionamento, evitando confronti e ricorsioni inutili sugli elementi uguali al pivot, che sono già in posizione corretta. La ricorsione viene infatti applicata solo alle sottosequenze strettamente minori o maggiori del pivot. L'algoritmo è in-place (non richiede spazio aggiuntivo significativo) ma non stabile (gli elementi uguali possono cambiare ordine relativo).

### 4.1 Analisi della complessità

Sia  $n$  la dimensione dell'array:

- Caso migliore:  $O(n)$ , quando tutti gli elementi sono uguali, e la partizione centrale include l'intero array.
- Caso medio:  $O(n \cdot \log(n))$ , come il Quick Sort classico, ma con una costante migliore grazie alla gestione efficiente dei duplicati.
- Caso peggiore:  $O(n^2)$ , in caso di pivot molto sbilanciati e assenza di duplicati.
- Spazio ausiliario:  $O(\log(n))$  per la profondità della ricorsione.

Nel complesso, Quick Sort 3-Way si comporta meglio del Quick Sort classico in presenza di dati con molti valori ripetuti, mantenendo lo stesso ordine di complessità.

## 4.2 Analisi empirica

Per verificare empiricamente l'efficienza dell'algoritmo, sono stati condotti due esperimenti separati:

- Esperimento 1:  $m = 100000$  fisso, e variazione della dimensione dell'array  $n$ .
- Esperimento 2:  $n = 10000$  fisso, e variazione dell'intervallo dei valori  $m$ .

I valori di  $n$  e  $m$  sono distribuiti secondo una progressione geometrica. Per ogni coppia di parametri sono stati generati almeno 100 campioni casuali, ciascuno eseguito più volte. I tempi medi di esecuzione sono stati stimati con un clock monotono ad alta precisione (`time.perf_counter()`), garantendo un errore relativo massimo  $\leq 0.001$ .

I risultati ottenuti sono coerenti con le aspettative teoriche:

- Nel primo esperimento, i tempi crescono approssimativamente secondo una curva  $O(n \cdot \log(n))$ , come previsto.
- Nel secondo esperimento, si osserva che, riducendo  $m$ , aumentano i duplicati nei dati, e l'algoritmo ne beneficia: i tempi di esecuzione diminuiscono sensibilmente grazie al partizionamento ottimizzato del Quick Sort 3-Way.

Nel grafico allegato si evidenziano chiaramente i vantaggi di questa variante rispetto alla versione classica, soprattutto in presenza di dati ridondanti.

# Capitolo 5

## Radix Sort

Il Radix Sort è un algoritmo di ordinamento non comparativo che sfrutta le rappresentazioni numeriche degli elementi. Ordina i numeri interi cifra per cifra, a partire dalla cifra meno significativa (LSD - Least Significant Digit) fino a quella più significativa. L'ordinamento delle singole cifre è effettuato tramite un algoritmo stabile, come il Counting Sort, applicato in sequenza per ogni posizione. Radix Sort lavora bene quando si conosce il numero massimo di cifre degli elementi (ovvero  $\log_{10}(k)$  se  $k$  è il valore massimo presente). Questo algoritmo è in-place (usa spazio aggiuntivo proporzionale a  $n + b$ , con  $b$  la base usata, tipicamente 10), ma non stabile per natura a meno che il sorting delle cifre lo sia (in questo caso è garantita la stabilità tramite Counting Sort).

### 5.1 Analisi della complessità

Sia  $n$  la lunghezza dell'array e  $d$  il numero massimo di cifre (ossia  $d = \lceil \log_{10}(k + 1) \rceil$ ):

- Tempo:  $O(d \cdot (n + b))$ , che nel caso più comune ( $b = 10$ ) diventa  $O(n \cdot \log_{10}(k))$ . Quando  $k = O(n^c)$  per una costante  $c$ , si ha complessivamente tempo quasi-lineare  $O(n)$ .
- Spazio:  $O(n + b)$  per ogni cifra, dove  $b$  è la base (es. 10 cifre decimali).
- Stabile: sì, se il counting sort interno è stabile.
- In-place: no, nel senso stretto (usa memoria ausiliaria  $O(n)$ ).

### 5.2 Analisi empirica

L'algoritmo è stato testato su array di interi generati casualmente, secondo due configurazioni sperimentali:

- Esperimento 1:  $m = 100000$  (valore massimo) fisso, variando la dimensione  $n$  dell'array.
- Esperimento 2:  $n = 10000$  fisso, variando  $m$ , cioè il massimo valore presente negli elementi.

In entrambi i casi, i parametri sono scelti su una scala geometrica. Ogni configurazione  $(n, m)$  è stata eseguita su almeno 100 campioni, ciascuno ripetuto più volte. I tempi di esecuzione sono stati misurati tramite un clock monotono con errore relativo massimo  $\leq 0.001$ .

I risultati ottenuti confermano l'efficienza teorica:

- In Esperimento 1, il tempo di esecuzione cresce linearmente con  $n$ , confermando il comportamento  $O(n \cdot \log_{10}(k))$ , dato  $k$  fisso.
- In Esperimento 2, aumentando  $k$  (cioè il numero massimo di cifre  $d$ ), si osserva un aumento graduale dei tempi dovuto alla crescita del numero di passaggi di counting sort, coerente con la complessità  $O(d \cdot n)$ .

Radix Sort si dimostra particolarmente adatto per dataset numerici con intervallo limitato, risultando spesso più veloce di algoritmi comparativi come Quick Sort nei casi in cui  $k = O(n)$  o  $k \ll n^2$ .

# Capitolo 6

## Conclusioni

TODO (?)