Relazione: Implementazione e Analisi degli Algoritmi di Ordinamento in Python
Corso: Algoritmi e Strutture Dati

Gruppo:
Gatti Giuseppe 159718 gatti.giuseppe@spes.uniud.it
Lattanzio Fabio M. 159263 lattanzio.fabiomassimo@spes.uniud.it

Anno accademico: 2024/2025
Data di consegna: [Inserisci data di consegna]

INDICE

Introduzione 1

Obiettivi e Specifiche del Progetto 2

Metodologia di Implementazione 3

Metodologia di Misurazione delle Prestazioni 4

Risultati e Analisi 5

Discussione Critica 6

Conclusioni 7

1. Introduzione

Contesto e motivazione
L'ordinamento è uno dei problemi fondamentali in informatica. La capacità di ordinare efficacemente grandi quantità di dati è alla base di molti algoritmi e applicazioni in vari campi, tra cui la ricerca, l'analisi dei dati e l'ottimizzazione. La velocità degli algoritmi di ordinamento può influire significativamente sulle prestazioni complessive dei sistemi informatici. In questo progetto, sono stati implementati e analizzati diversi algoritmi di ordinamento per confrontarne l'efficienza.

Il principale obiettivo di questo progetto è implementare e confrontare l'efficienza di diversi algoritmi di ordinamento, come Quick Sort, Quick Sort 3-Way, Merge Sort e Counting Sort fornendo una specie di scheda tecnica per ognuno di essi. Attraverso misurazioni precise dei tempi di esecuzione, l'obiettivo è comprendere come questi algoritmi si comportano in relazione alla dimensione dell'array (n) e al range min-max dei valori degli elementi dell'array(m).

2. Obiettivi e Specifiche del Progetto

Descrizione degli algoritmi implementati
Sono stati implementati i quattro algoritmi di ordinamento:

Quick Sort: Un algoritmo di ordinamento basato sulla tecnica di "dividi et impera", che seleziona un elemento pivot e partiziona l'array in due sottosequenze.

Quick Sort 3-Way: una variante del Quick Sort classico che partiziona l'array in tre sezioni:

- elementi minori del pivot,

- elementi uguali al pivot,
- elementi maggiori del pivot.
  È particolarmente efficiente quando l'array contiene molti duplicati, poiché riduce il numero di confronti e chiamate ricorsive.

Merge Sort: Un altro algoritmo "dividi et impera" che divide l'array in due metà e le ordina ricorsivamente, per poi unire.

Counting Sort: algoritmo non comparativo adatto per ordinare array di interi in un range limitato. Conta il numero di occorrenze di ciascun valore e calcola direttamente le posizioni finali. Ha complessità O(n + k), dove *k* è il valore massimo degli interi da ordinare. Funziona solo per interi **non negativi**.

Requisiti per le misurazioni
Parametri n: la dimensione dell'array deve essere compresa tra 100 e 100'000.
Parametri m: la dimensione del range di interi deve essere compresa tra 10 e 1'000'000.
Generazione dei campioni: I campioni vengono generati seguendo una progressione geometrica. La variabile n varia in base a una formula esponenziale.
Misurazione dei tempi: Viene utilizzato il modulo ***time.perf_counter()*** per misurare i tempi di esecuzione. Il massimo errore relativo è garantito entro 0.001.

Scheda tecnica

Algoritmo	Caso Ottimo	Caso Medio	Caso Pessimo	Spazio Ausiliario	Compl. Memoria	Stabilità	Note	Caso peggiore: perché?
Quick Sort	Θ(n log n)	Θ(n log n)	Θ(n^2)	Θ(log n)	Bassa	❌	Molto veloce, sensibile alla scelta del pivot	Quando l'array è già ordinato (pivot sempre minimo o massimo) : l'albero di ricorsione diventa lineare
Quick Sort 3-Way	Θ(n) con duplicati	Θ(n log n)	Θ(n^2)	Θ(log n)	Bassa	❌	Ottimo per dati con molti duplicati	Se tutti gli elementi sono distinti e il pivot è mal scelto: non si beneficia della

								partizione in 3
Merge Sort	Θ(n log n)	Θ(n log n)	Θ(n log n)	Θ(n)	Media	✅	Stabile, non in-place, prevedibile in ogni caso	Non cambia: è stabile in tutti i casi grazie alla divisione sistematica dell'array
Counting Sort	Θ(n + k)	Θ(n + k)	Θ(n + k)	Θ(n)	Alta (dipende da k)	✅	Non comparativo, ottimo su interi in range ristretto	Quando k ≫ n (range molto grande rispetto al numero di elementi) : grande uso di memoria

3. Metodologia di Implementazione

Ambiente di sviluppo

Il progetto è stato sviluppato in Python, utilizzando le librerie standard per la gestione degli array e la misurazione del tempo. In particolare, è stato utilizzato il modulo ***time*** per il calcolo dei tempi di esecuzione. I grafici sono stati creati utilizzando la libreria ***matplotlib***, mentre i dati sono stati salvati in formato ***CSV***.

Il codice è suddiviso in vari moduli e funzioni:

- Funzioni di ordinamento (per ogni algoritmo).
- Funzione per la generazione di array casuali.
- Funzione per la misurazione dei tempi di esecuzione.
- Funzione per il salvataggio dei dati in CSV.
- Funzione per la creazione dei grafici comparativi.
- Funzioni per il testing

4. Metodologia di Misurazione delle Prestazioni

Stima della risoluzione del clock

Per garantire misurazioni precise, è stato utilizzato il metodo di stima della risoluzione del clock. Il tempo minimo misurabile (*Tmin*) è stato calcolato come segue:

import time

def clock_resolution():
start = time.perf_counter()
while time.perf_counter() == start:
pass
stop = time.perf_counter()
return stop – start

```python
def clock_resolution():
    """Stima la risoluzione del clock usando time.perf_counter()."""
    start = time.perf_counter()
    while time.perf_counter() == start:
        pass
    stop = time.perf_counter()
    return stop - start
```

Procedura di misurazione
Un array di dimensione n viene inizializzato con valori casuali nell'intervallo [1, m].
L'algoritmo di ordinamento viene eseguito più volte fino a superare il tempo minimo Tmin. Questo processo viene eseguito con questa funzione :

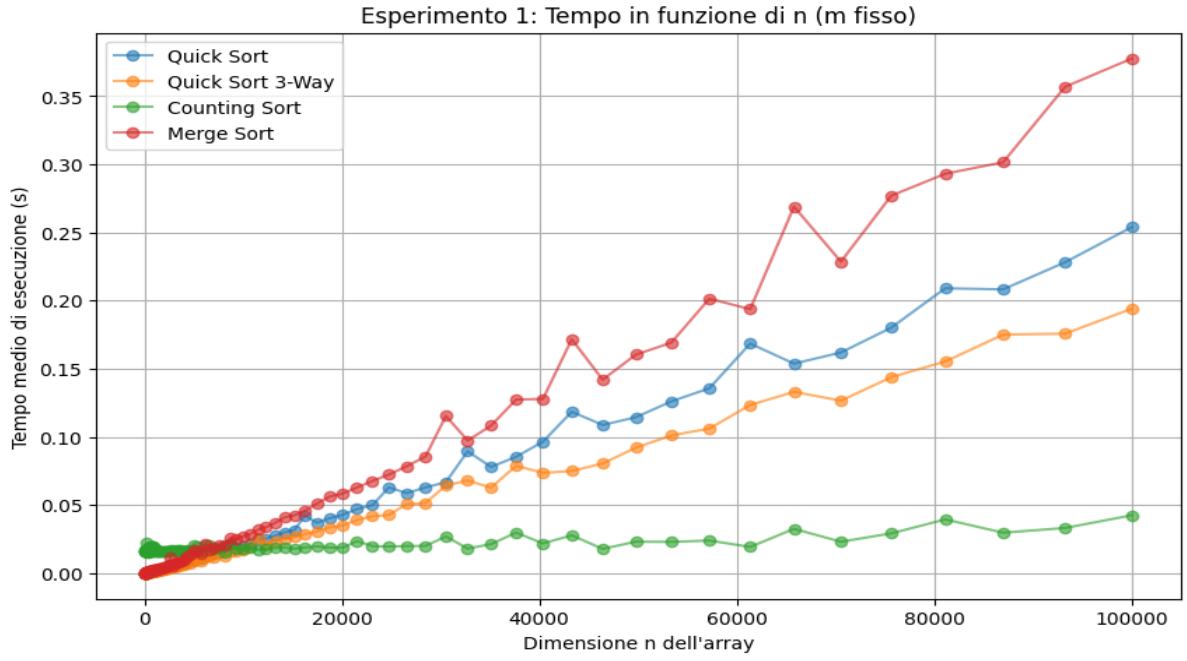
def measure_sorting_time(sort_func, n, m, T_min, num_trials=10, subtract_init=False, init_time=0.0)

Il tempo medio per esecuzione viene calcolato come la somma dei tempi di tutte le iterazioni divisa per il numero di iterazioni.

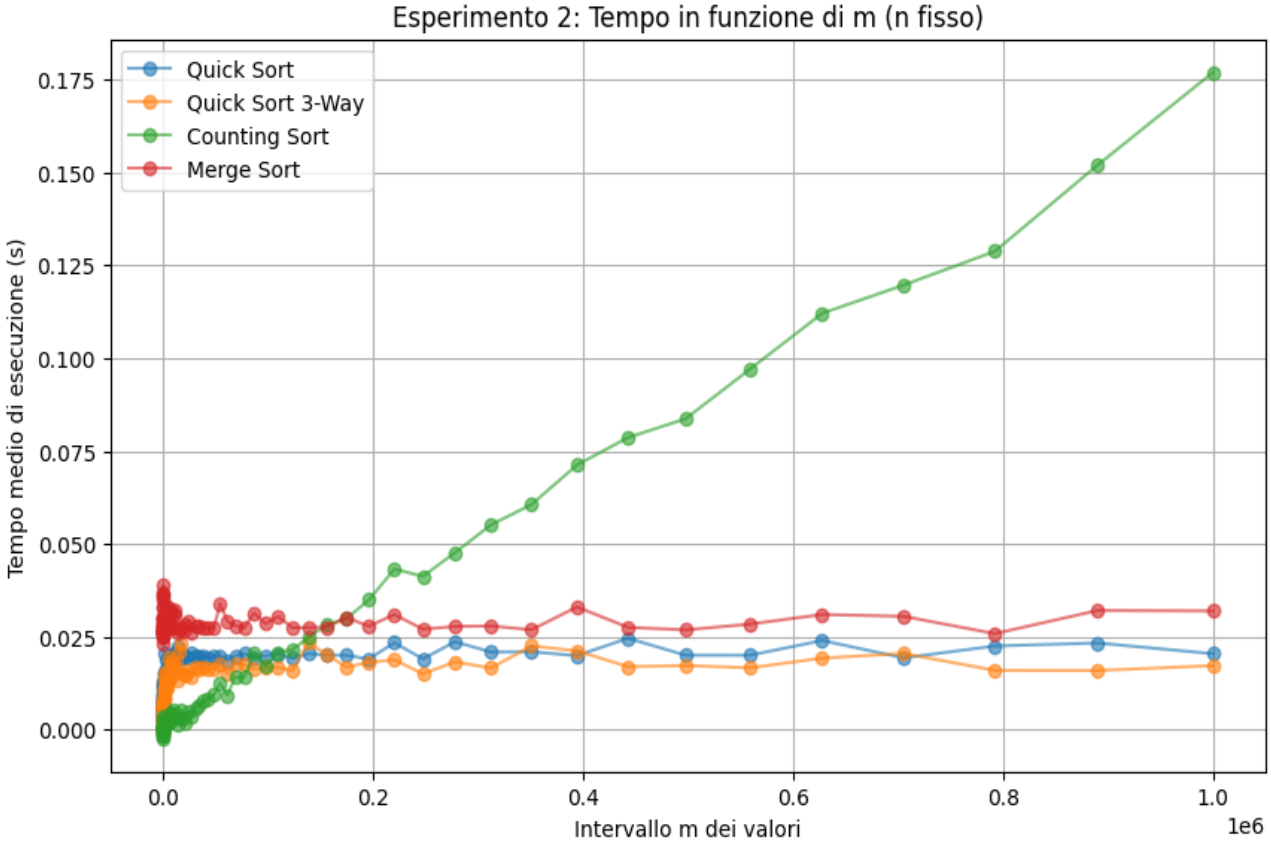
Gestione dell'errore relativo
Per garantire che l'errore relativo nelle misurazioni sia inferiore a 0.001, vengono effettuate misurazioni ripetute e viene calcolato il tempo medio di esecuzione per ciascun set di parametri.

5. Risultati e Analisi
Presentazione dei dati
Nel seguente grafico, vengono mostrati i tempi di esecuzione in funzione di n (con m costante). I grafici sono stati creati per ogni algoritmo implementato.

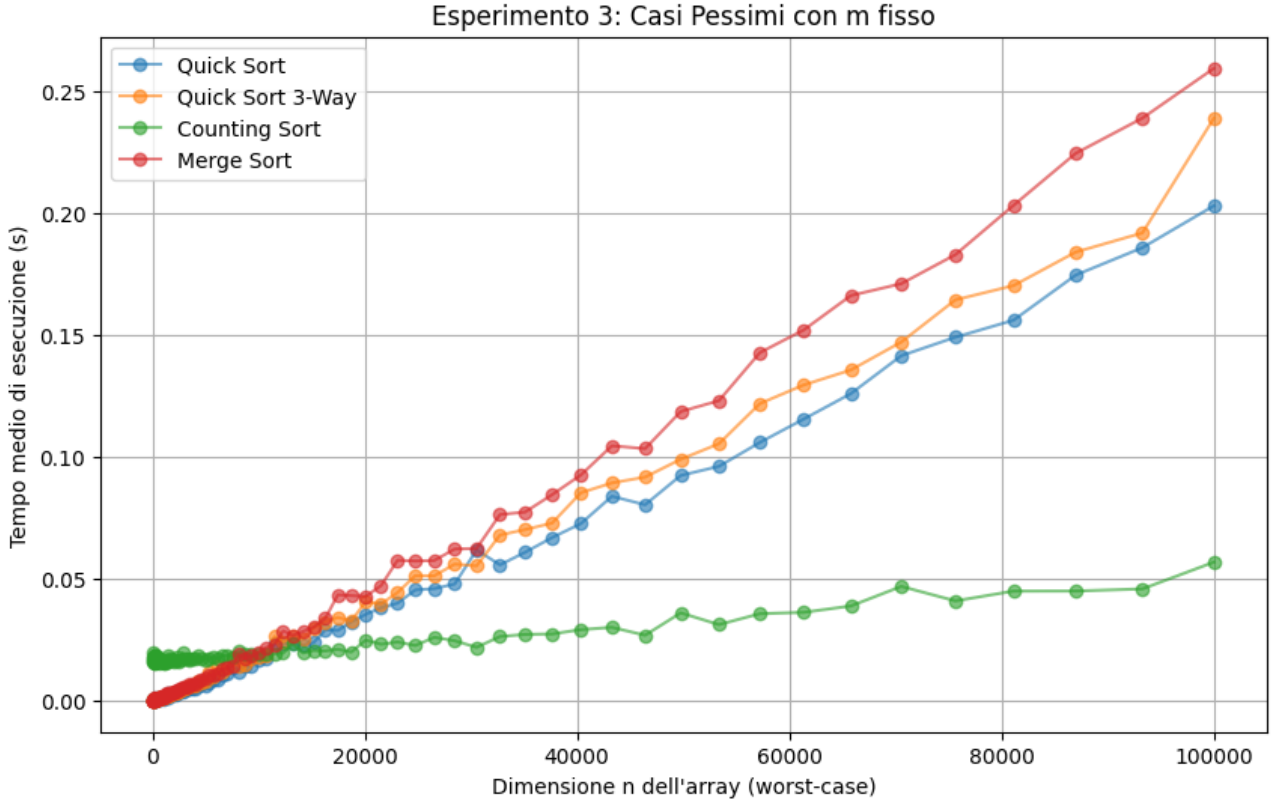
Esperimento 1: Tempo in funzione di n (m fisso)



Esperimento 2: Tempo in funzione di m (n fisso)



Esperimento 3: Casi pessimi con m fisso



Discussione dei risultati

I risultati mostrano come la performance degli algoritmi cambi al variare di n. Si può notare come **Counting Sort** rispetti l'andamento lineare Θ(n + k). Notiamo che per un n intorno ai 10'000 risulti essere il più efficiente dei quattro algoritmi. Gli altri algoritmi "dividi et impera" crescono in egual modo. Nell'esperimento 2, notiamo però che **Counting Sort** non è il più efficiente tra i quattro, in quanto all'aumentare del range (m), il tempo non è più Θ(n + k), ma più precisamente sarà Θ(k) in quanto k >> n (nell' esperimento 2 abbiamo fissato n = 10'000 ed m varia da 10 a 1'000'000).
Per l'appunto notiamo che quando k = n, **Counting Sort** continua ad essere tra i migliori. Già con una variazione di 200'000 tra il minimo ed il massimo, notiamo che gli altri tre algoritmi sono più veloci ed efficienti.

6. Discussione Critica

Valutazione dell'implementazione

L'implementazione è stata generalmente soddisfacente. Le principali difficoltà incontrate riguardano la gestione del tempo di esecuzione nei casi di dimensioni molto piccole o molto grandi degli array. Ci sono stati problemi nel capire come mai ci venissero tempi negativi in alcune simulazioni e test; abbiamo scoperto poi in seguito che i tempi negativi erano dovuti alla variabile t0 che erroneamente era all'interno del ciclo della funzione di misurazione dei tempi. Una volta tolta dal ciclo *while*, i tempi sono stati calcolati correttamente

7. Conclusioni

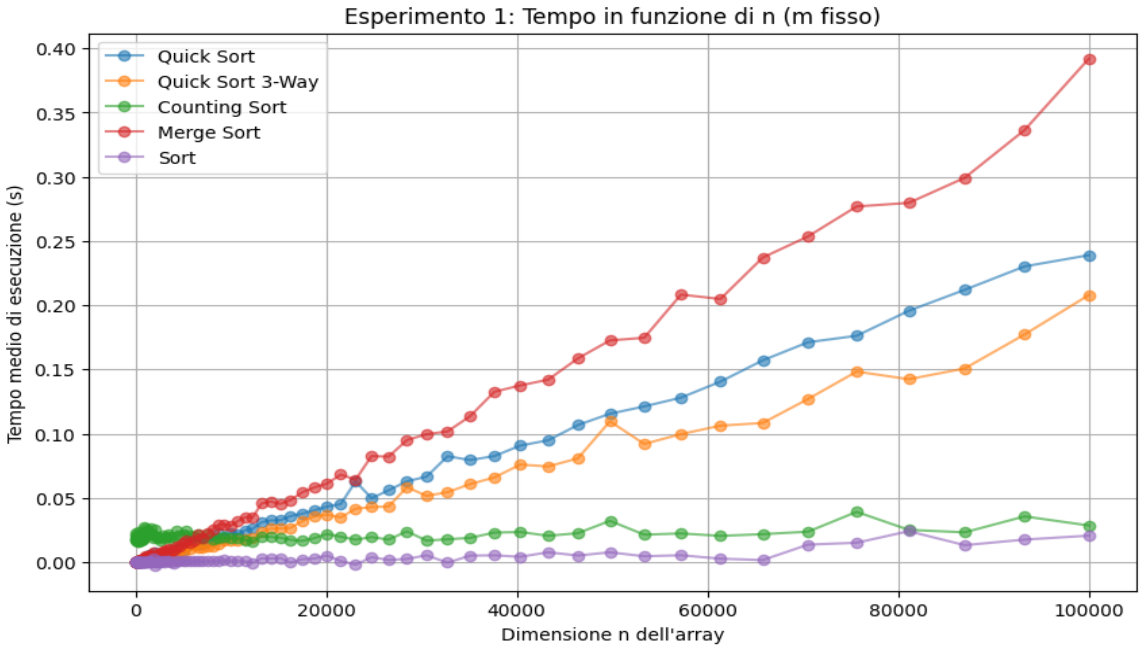
Sintesi dei risultati ottenuti

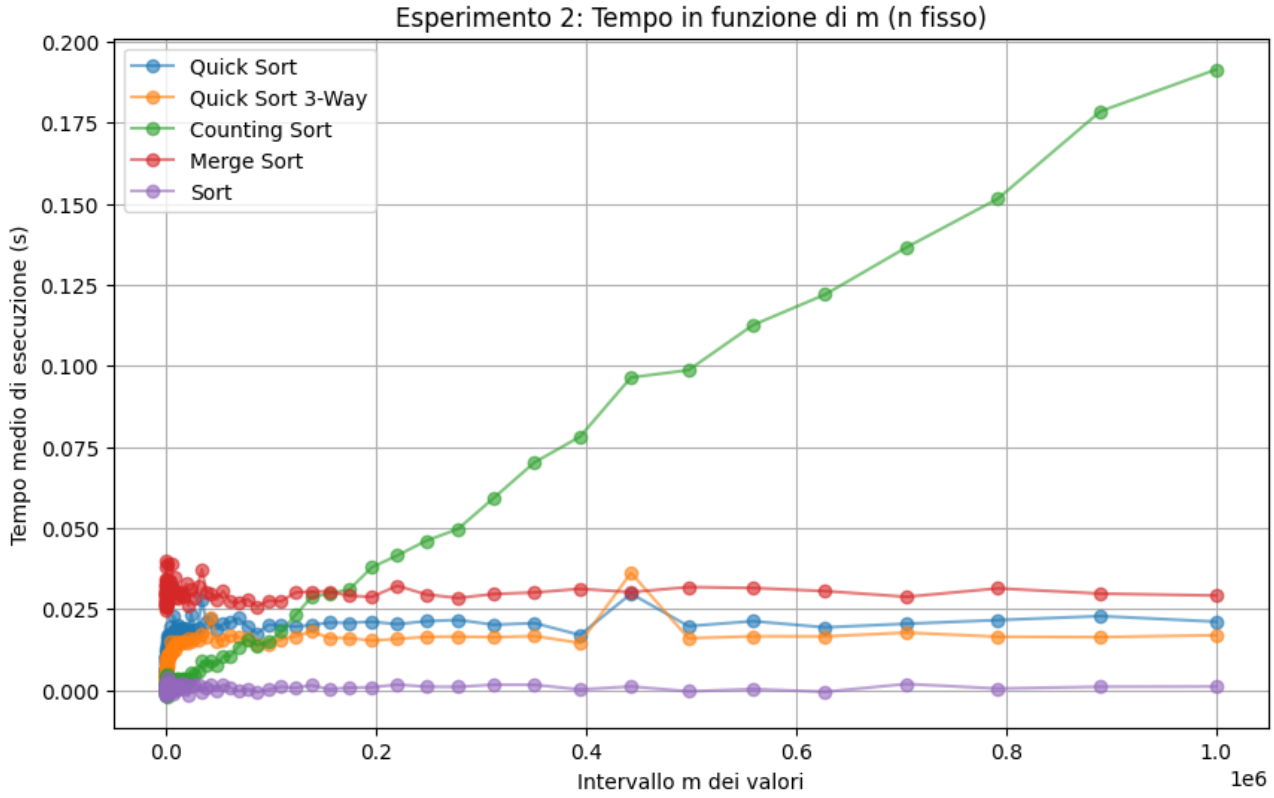
Gli algoritmi di ordinamento implementati sono stati confrontati efficacemente in base ai tempi di esecuzione. **Quick Sort 3-Way** si è dimostrato più veloci rispetto a **Quick Sort**, specialmente con array di grandi dimensioni.

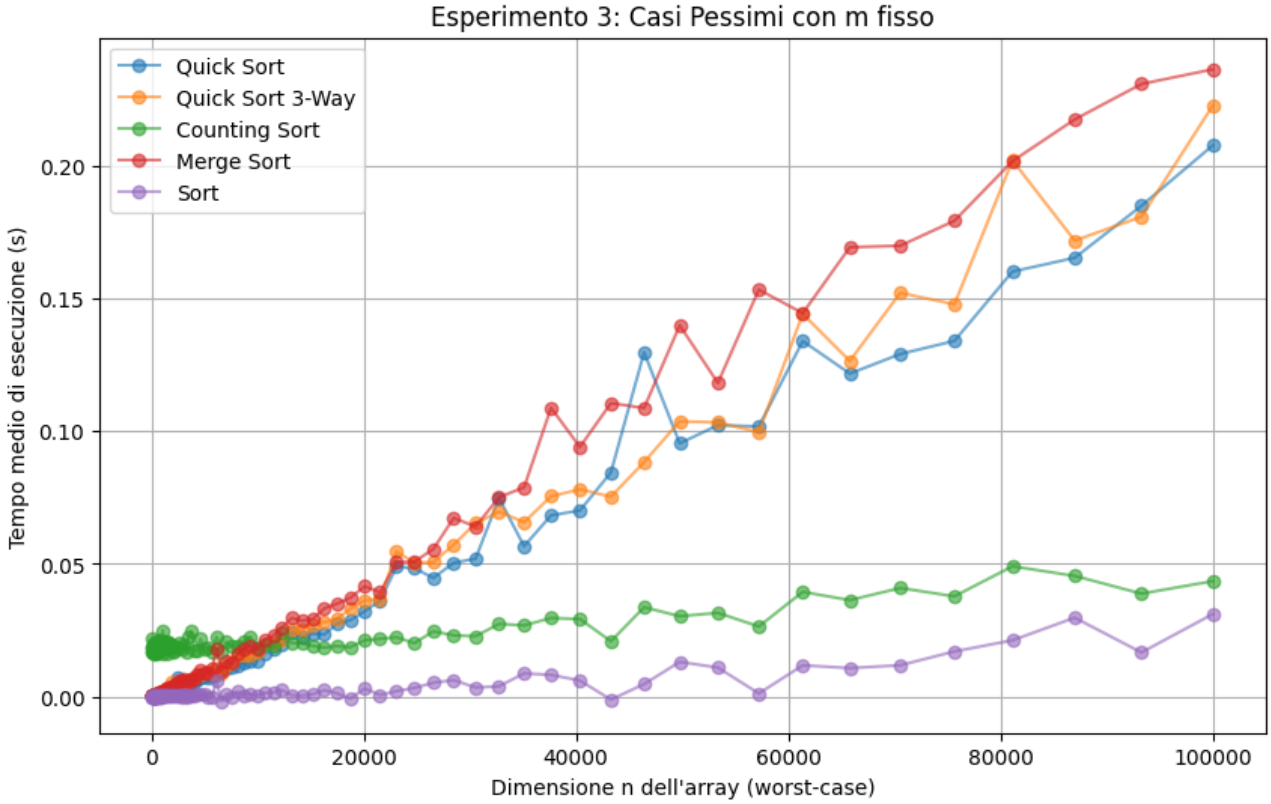
Considerazioni finali

Questo studio ha permesso di comprendere in dettaglio le performance di vari algoritmi di ordinamento e ha fornito una base per ulteriori approfondimenti nel campo dell'analisi delle prestazioni degli algoritmi.

C'è però da dire che in Python è implementato un metodo di sorting che è ottimizzato per il linguaggio stesso. Qui sono riportati i grafici che dimostrano la sua efficienza rispetto gli altri algoritmi analizzati.




Esperimento 2: Tempo in funzione di m (n fisso)
Quick Sort
Quick Sort 3-Way
Counting Sort
Merge Sort
Sort
Tempo medio di esecuzione (s)
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
0.0
0.2
0.4
0.6
0.8
1.0
1e6
Intervallo m dei valori


Esperimento 3: Casi Pessimi con m fisso
Quick Sort
Quick Sort 3-Way
Counting Sort
Merge Sort
Sort
Tempo medio di esecuzione (s)
0.20
0.15
0.10
0.05
0.00
0
20000
40000
60000
80000
100000
Dimensione n dell'array (worst-case)