Analisi empirica degli algoritmi di ordinamento

Graziano Francesco [1] , Ongaro Michele [2] , Petri Riccardo [3] e Ungaro Marco [4]

Università degli Studi di Udine, Dipartimento di Scienze matematiche, informatiche e fisiche

A.A. 2024-2025

[1]Email: graziano.francesco@spes.uniud.it, Matricola: 166680
[2]Email: ongaro.michele@spes.uniud.it, Matricola: 168049
[3]Email: petri.riccardo@spes.uniud.it, Matricola: 167623
[4]Email: ungaro.marco@spes.uniud.it, Matricola: 168934

Indice

Capitolo 1

Introduzione

Con la seguente relazione si vuole documentare i risultati empirici ottenuti dalla realizzazione e implementazione di quattro algoritmi di ordinamento per vettori di interi di dimensione variabile. Gli algoritmi che andremo ad analizzare sono il Counting Sort, il Quick Sort, il Quick Sort 3 way e il Radix Sort. Oltre alla corretta implementazione viene richiesto di effettuare un'analisi empirica dei tempi di esecuzione degli algorimti al variare della dimensione del vettore e del range dei valori interi. Per stimare i tempi di esecuzione di questi algoritmi garanantendo 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 100 campioni per ciascun grafico, con i valori dei parametri (dimensione del vettore$n$ 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.

Dopo aver stimato i tempi di esecuzione per ciascun algoritmo, risulterà interessante confrontare i grafici ottenuti per analizzarne il comportamento.

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 nel vettore da ordinare. È particolarmente efficiente quando gli elementi da ordinare sono numeri interi compresi in un intervallo limitato (intervallo $[0, m]$). L'implementazione adottata si articola in tre fasi principali:

- Conteggio delle occorrenze di ciascun elemento; si costruisce un vettore ausiliario C di lunghezza $m+1$, 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 nel vettore ordinato, si trasforma C in un vettore cumulativo. In questa versione, $C[0]$ viene inizialmente decrementando di uno, così che $c[i]$ rapprsenti l'indice massimo in cui si può inserire l'elemento i nel vettore ordinato.
- Costruzione del vettore ordinato; si scorre il vettore originale da destra a sinistra, e si inserisce ciascun elemento x nella posizione $C[x]$ del vettore 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 nel vettore originale.

Risulta interessante notare che il Counting Sort, a differenza degli altri algoritmi, non andrà a modficare direttamente il vettore fornito come parametro, bensì per motivi legati alla logica dell'algoritmo farà uso di un vettore ausiliario che rappresentare una copia ordinata del vettore di partenza.

2.1 Complessità teorica

Per quanto riguarda l'algoritmo Couting Sort, ci aspettiamo una complessità lineare in tempo. Siano quindi n la dimensione del vettore A e m il valore massimo contenuto in A, la complessità sarà:

$$\Theta(n + m)$$

2.2 Analisi empirica

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

- Nel primo esperimento, la dimensione del vettore n varia, mentre il valore massimo m è mantenuto fisso a 100000;
- Nel secondo esperimento, n è fissato a 10000, e viene fatto variare m;
- Nel terzo esperimento, n varia tra 0 e 10000 e m è posto come il quadrato di n.

Ogni esperimento è stato analizzato empiricamente considerando 100 campionamenti distinti nei quali i valori n o m venivano fatti variare seguendo l'andamento di una progressione geometrica. Ogni campionamento inoltre è stato eseguito più volte per stimare in modo affidabile il tempo medio di esecuzione, mantenendo un errore relativo massimo minore di 0.001s. Le misurazioni sono state effettuate utilizzando un clock monotono ad alta precisione (`perf_counter()` in Python).

2.2.1 Primo esperimento (n varia)

In primo luogo, è stato analizzato l'andamento temporale del Couting Sort al variare della dimensione del vettore n, mantenendo costante l'intervallo dei valori a $m = 100000$. Il grafico seguente mostra in ascissa la dimensione n del vettore e in ordinata il tempo di esecuzione per l'ordinamento espresso in secondi.

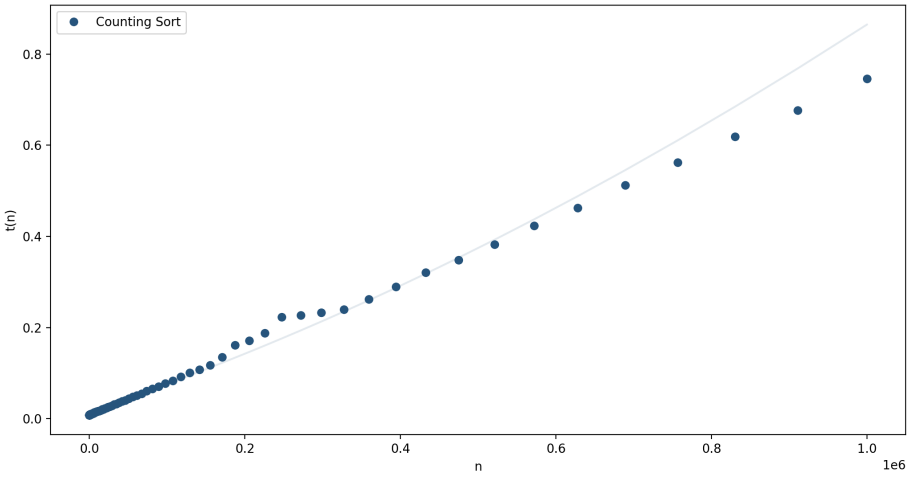

Figura 2.1: Performance del Counting Sort al variare di n.

La crescita, come da aspettative, segue un andamento lineare. Il calcolo del rapporto tra n e tempo infatti assume asintoticamente un valore costante.

2.2.2 Secondo esperimento (m varia)

Successivamente, è stato analizzato l'andamento temporale al variare del parametro m, mantenendo costante la dimensione del vettore a $n = 10000$. Il grafico seguente presenta il parametro m e in ordinata il tempo di esecuzione del Couting Sort espresso in secondi.

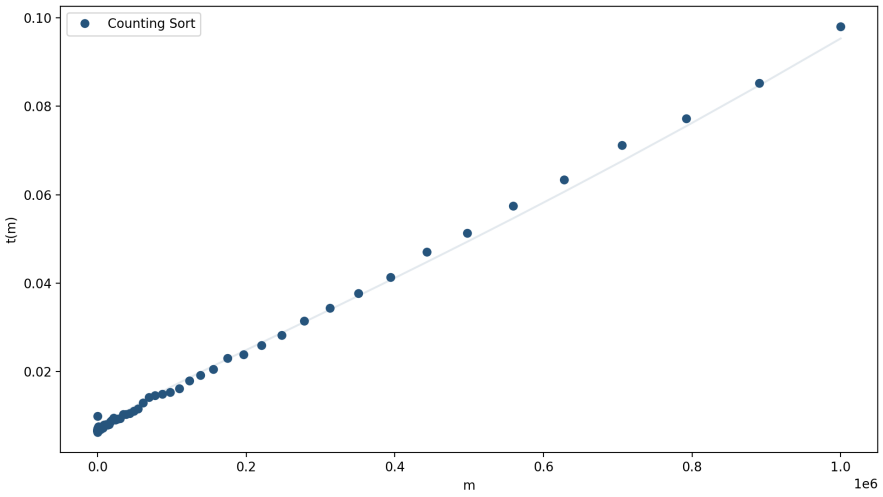

Figura 2.2: Performance del Counting Sort al variare di m.

La crescita, come da aspettative, segue un andamento lineare nonostate la lunghezza del vettore sia costante, questo porta conferma alla complessità teorica $\Theta(n + m)$.

2.2.3 Terzo esperimento (crescita quadratica)

In terza battuta, al variare di n tra 100 e 10000 (rispetto al range 100-1000000, per mantenere contenuto il tempo di esecuzione) è stato posto il massimo m come il quadrato della lunghezza del vettore. Con questa misurazione, mostriamo che se al crescere della dimensione del vettore, l'elemento massimo cresce con un ordine superiore, l'algoritmo diventa molto inefficiente rispetto ai tradizionali algoritmi basati su scambi e confronti, per i quali la crescita di m non comporta problemi. Il grafico seguente presenta in ascissa la lunghezza n dei vettori, in ordinata il tempo di esecuzione del Counting Sort espresso in secondi.

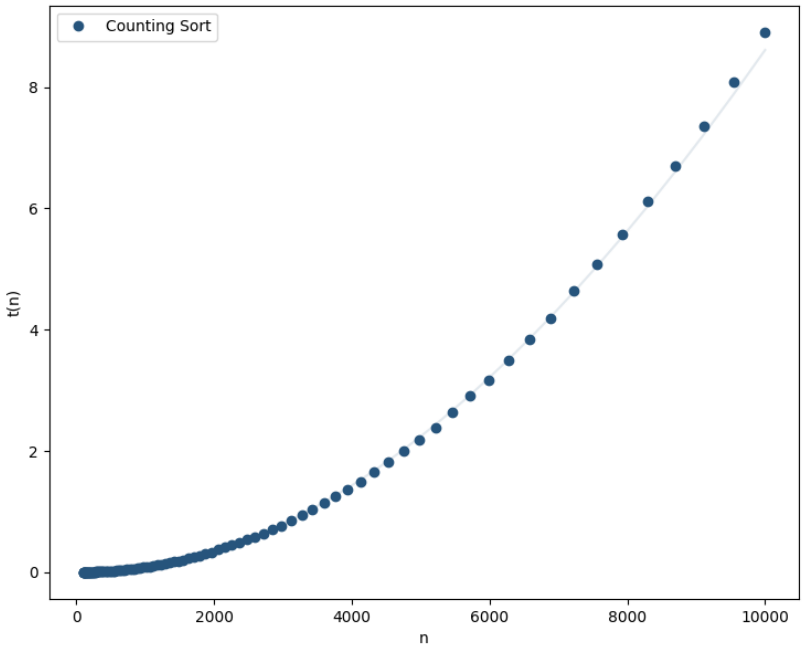

Figura 2.3: Performance del Counting Sort al variare di n e $m = n^2$.

In questo caso, come ci aspettavamo, l'andamento sembra essere proprio quadratico rispetto ad n ma può arrivare ad essere anche peggio nel caso in cui m cresce ancora di più. In ogni caso i tempi anche solo per vettori da 10000 elementi risultano molto alti, superando gli 8 secondi, da qui vediamo anche le alte costanti moltiplicative nascoste dietro la notazione asintotica.

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 vettore A, si seleziona un pivot, in questo caso l'elemento finale del sottovettore $A[p...q]$, 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 del Quick Sort analizzata non è in place a causa della coda delle ricorsioni, e non soddisfa la condizione di stabilità.

3.1 Complessità teorica

Per quanto riguarda l'algoritmo Quick Sort ci aspettiamo di misurare una crescita temporale pari a $O(n \cdot \log(n))$ nel caso medio, ossia quando il vettore da ordinare è generato casualmente e quindi la scelta del pivot partiziona in maniera equa il vettore. La complessità nel caso peggiore, che per il Quick Sort è rappresentato da il vettore ordinato, ci aspettiamo che sia $O(n^2)$.

3.2 Analisi empirica

Per analizzare empiricamente il comportamento del Quick Sort sono stati eseguiti i tre esperimenti seguenti:

- Nel primo esperimento la dimensione del vettore n viene fatta variare, mantenendo fisso $m = 100000$;
- Nel secondo esperimento abbiamo $n = 10000$ costante, con variazione di m;
- Nel terzo esperimento studio il caso peggiore quindi il vettore sarà ordinato, $m = 100000$, n varia tra 100 e 10000.

Ogni esperimento è stato analizzato empiricamente considerando 100 campionamenti distinti nei quali i valori n o m venivano fatti variare seguendo l'andamento di una progressione geometrica. Ogni campionamento inoltre è stato eseguito più volte per stimare in modo affidabile il tempo medio di esecuzione, mantenendo un errore relativo massimo

minore di 0.001s. Le misurazioni sono state effettuate utilizzando un clock monotono ad alta precisione (`perf_counter()` in Python).

3.2.1 Primo esperimento (n varia)

A partire dai dati misurati è stato generato un grafico nel quale la lunghezza del vettore indicata con n varia da 100 a 1000000, mentre il parametro m è mantenuto costante. L'ordinamento è stato eseguito utilizzando l'algoritmo Quick Sort su vettori contenenti numeri casuali.

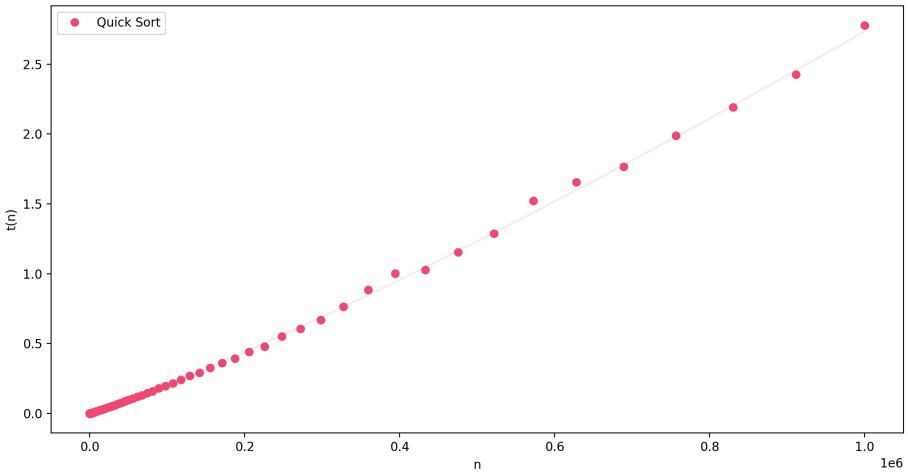

Figura 3.1: Performance del Quick Sort al variare di n.

In questo caso, come ci aspettavamo, l'andamento è $n \log(n)$, quindi poco più che lineare come da aspettative.

3.2.2 Secondo esperimento (m varia)

Successivamente, è stato realizzato un grafico in cui viene fatto variare il parametro m, mantenendo costante la dimensione del vettore a $n = 10000$. I risultati ottenuti sono stati riportati graficamente nel seguente grafico:

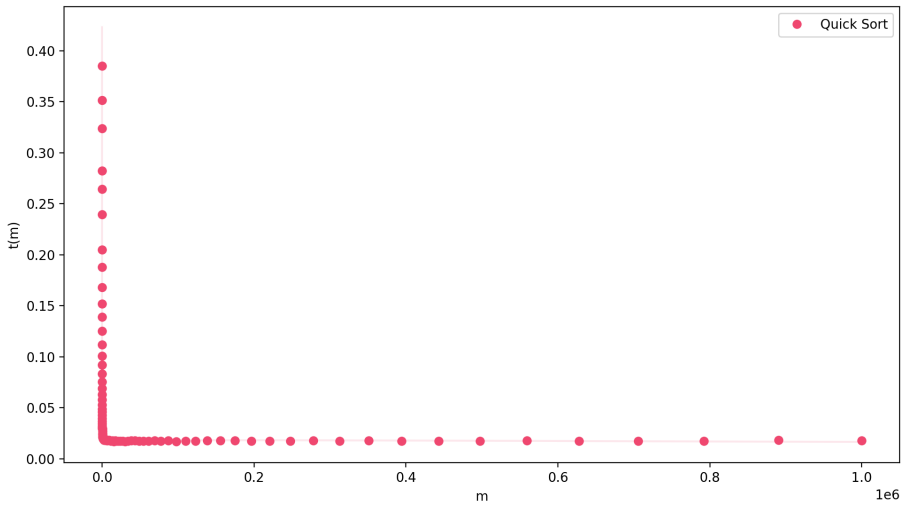

Figura 3.2: Performance del Quick Sort al variare di m.

Si osserva un'anomalia nel grafico quando il numero di valori distinti presenti nel vettore (m) è molto basso. La motivazione è presto detta: Quick Sort rischia di diventare $O(n^2)$ se ci sono tante ripetizioni dello stesso elemento, cosa che, per il principio della piccionaia avviene per valori di m piuttosto bassi (e.g. un vettore da 10000 elementi che possono assumere solamente 10 valori diversi risulta avere molte ripetizioni). Per il resto l'andamento è costante, infatti la complessità del Quick Sort ricordiamo essere $O(n \log n)$, n è costante e quinidi l'algoritmo richiede tempo costante.

3.2.3 Terzo esperimento (studio del caso peggiore)

Per ottenere dei campioni su cui Quick Sort risulta pessimo, le misurazioni sono state effettuate generando vettori già ordinati, in modo da avere una cattiva scelta dell'elemento pivot. I risultati ottenuti empiricamente sono stati rappresentati da un grafico:

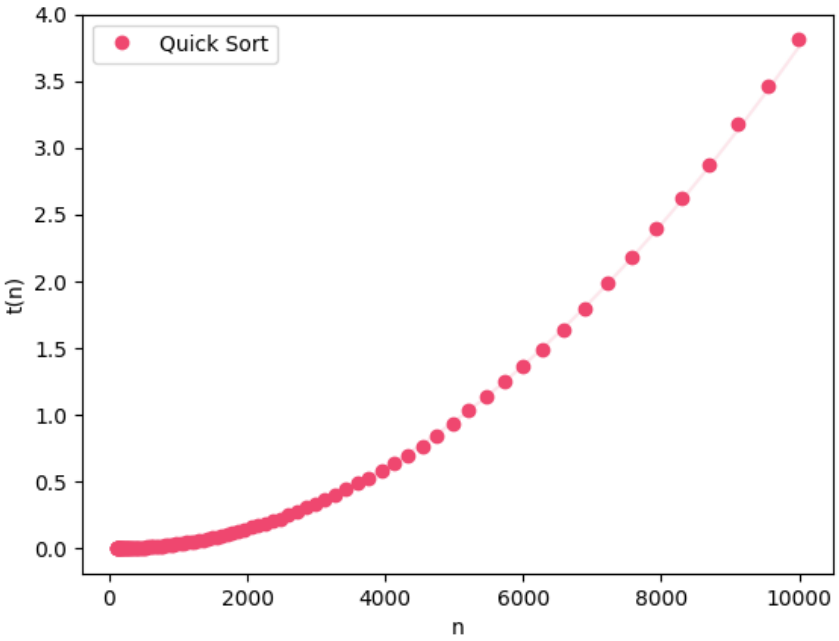

Figura 3.3: Performance del Quick Sort al variare di n su vettori ordinati.

Similmente al grafico del terzo esperimento fatto su Counting Sort, vediamo che l'andamento sembra quadratico, proprio come ci aspettavamo. In ogni caso i tempi sono abbastanza contenuti rispetto al Counting Sort nel suo caso quadratico, dato che con vettori da 10000 elementi rimaniamo comunque sotto i 4 secondi. In questo caso n varia da 100 a 10000, anche con questa limitazione è stato necessario aumentare il limite di ricorsioni imposto di default da python.

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 il vettore 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 il vettore 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 non è in-place per via della coda delle chiamate ricorsive e non è neanche stabile, infatti gli elementi uguali possono cambiare ordine relativo.

4.1 Complessità teorica

Dall'analisi empirica dell'algoritmo Quick Sort 3-Way ci aspettiamo ci osservare un andamento $O(n \cdot \log(n))$ nel caso medio, come il Quick Sort classico, ma con una costante moltiplicativa migliore grazie alla gestione efficiente dei valori duplicati. Nel caso peggiore, ossia nei casi in cui i pivot sono molto sbilanciati e con una quantità ridotta di valori duplicati, ci aspettiamo di osservare da questo algoritmo un costo $O(n^2)$. Il Quick Sort 3-Way nel caso migliore, ossia quando tutti gli elementi sono uguali è lineare rispetto a n.

4.2 Analisi empirica

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

- Nel primo esperimento abbiamo $m = 100000$ fisso, mentre la dimensione del vettore che varia da 100 a 1000000;

- Nel secondo esperimento la dimensione del vettore è $n = 10000$ mentre viene fatto variare il range dei valori m da 10 a 1000000;
- Nel terzo esperimento m viene fissato a 100000 la lunghezza del vettore viene fatta variare da 100 a 100000 e i vettori generati sono tutti generati già ordinati, come per il caso peggiore del Quick Sort.

Ogni esperimento è stato analizzato empiricamente considerando 100 campionamenti distinti nei quali i valori n o m venivano fatti variare seguendo l'andamento di una progressione geometrica. Ogni campionamento inoltre è stato eseguito più volte per stimare in modo affidabile il tempo medio di esecuzione, mantenendo un errore relativo massimo minore di 0.001s. Le misurazioni sono state effettuate utilizzando un clock monotono ad alta precisione (`perf_counter()` in Python).

4.2.1 Primo esperimento (n varia)

Nel primo esperimento, i tempi crescono approssimativamente secondo una curva $O(n \cdot log(n))$, come previsto. Di seguito viene riportato il grafico delle misurazioni che sono state effettuate:

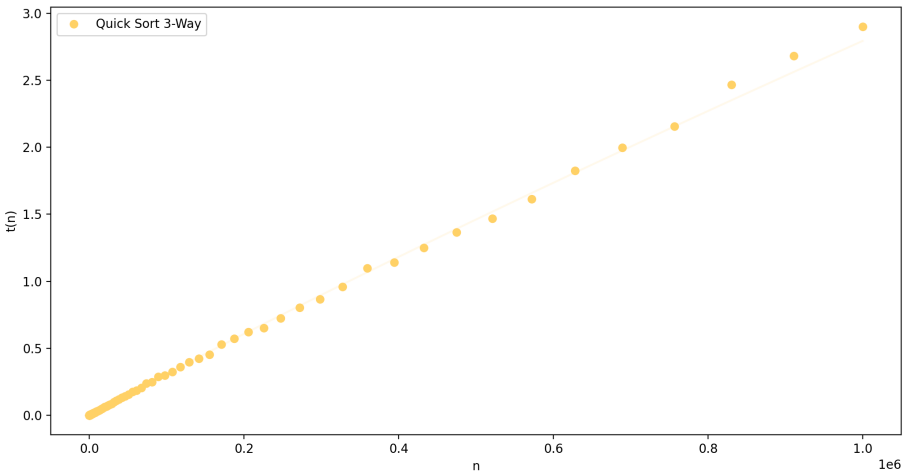

Figura 4.1: Performance del Quick Sort 3 Way al variare di n.

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

4.2.2 Secondo esperimento (m varia)

Nel secondo esperimento, si osserva che per valori m molto bassi aumentano inevitabilmente i duplicati nei dati, e l'algoritmo ne beneficia: i tempi di esecuzione diminuiscono sensibilmente grazie al partizionamento ottimizzato del Quick Sort 3-Way.

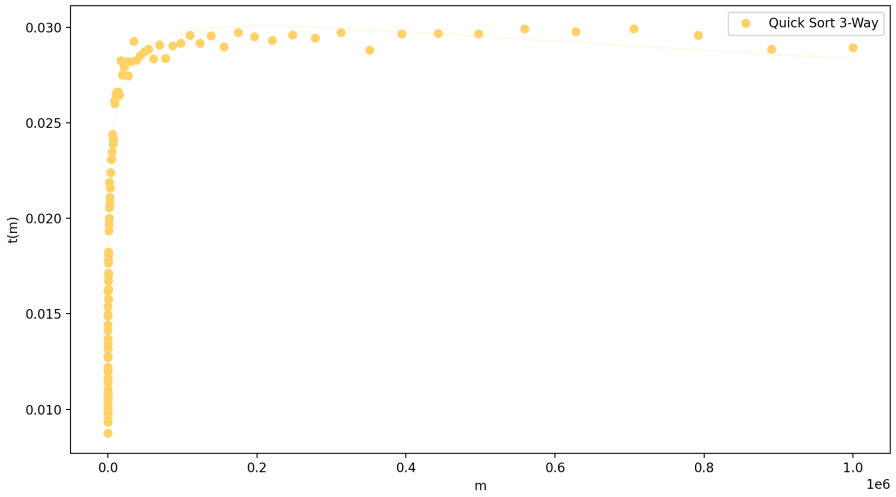

Figura 4.2: Performance del Quick Sort 3 Way al variare di m.

4.2.3 Terzo esperimento (caso peggiore)

Notiamo che anche in questo caso, come per il Quick Sort base, l'andamento peggiora, ma sembra resistere meglio. La performance risulta migliore grazie alla divisione in 3 parti del vettore, che probabilmente permette di evitare ricorsioni nel caso in cui ci siano elementi duplicati. In questo caso il range di n varia tra 100 e 100000 (fermandoci a 1000000 il tempo di misurazione e il numero di ricorsioni arrivava ad essere troppo elevato). Le misurazioni effettuate sono state riportate nel seguente grafico:

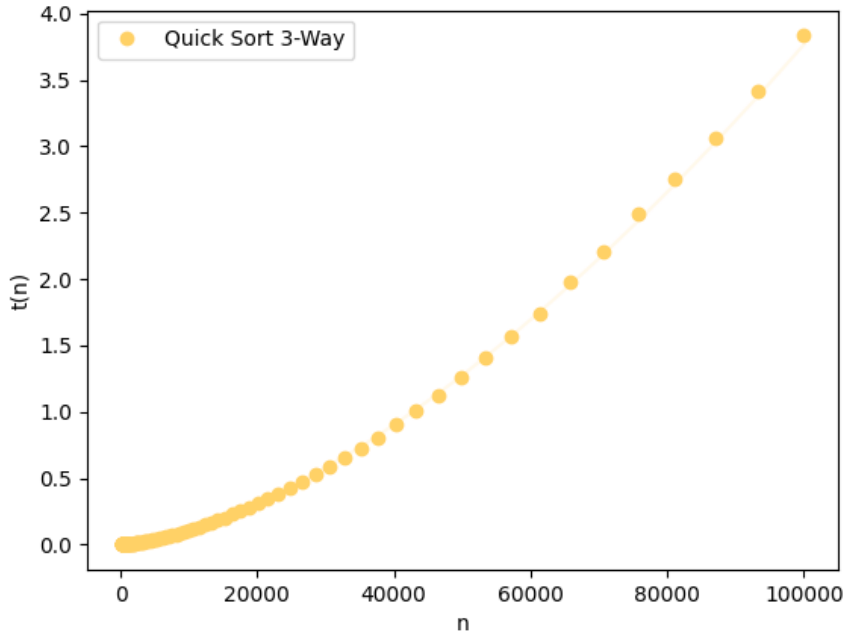

Figura 4.3: Performance del Quick Sort 3 Way su vettori ordinati.

Osserviamo che per vettori di 100000 elementi il tempo resta tra i 3 e 4 secondi, mentre questo avveniva con Quick Sort di base con vettori di appena 10000 elementi. Analizzando il rapporto tra tempo e $n \cdot \log(n)$

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 del vettore e d il numero massimo di cifre (ossia $d = \lceil \log_{10}(k+1) \rceil$):

- Tempo: $\Theta(d \cdot (n + b))$, che nel caso più comune ($b = 10$) diventa $\Theta(n \cdot log_{10}(k))$. Quando $k = O(n^c)$ per una costante c, si ha complessivamente tempo quasi-lineare $\Theta(n)$.
- Spazio: $\Theta(n + b)$ 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 $\Theta(n)$).

5.2 Analisi empirica

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

- Nel primo esperimento $m = 100000$ (valore massimo) fisso, variando la dimensione n del vettore da 100 a 1000000;
- Nel secondo esperimento $n = 10000$ fisso, variando m, cioè il massimo valore presente negli elementi da 10 a 1000000.

- Nel terzo esperimento è stato studiato lo stesso caso ”pessimo” visto per il Counting Sort, ma dal momento che con m che cresce come il quadrato di n la complessità temporale di Radix Sort risulta $\Theta(nlogn)$, abbiamo spinto n fino a 1000000, come negli gli altri casi.

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

I risultati ottenuti confermano l'efficienza teorica:

5.2.1 Primo esperimento (n varia)

Nel primo eseperimento osserviamo che il tempo di esecuzione cresce linearmente con n, confermando il comportamento $O(n \cdot log(m))$, dato m fisso.

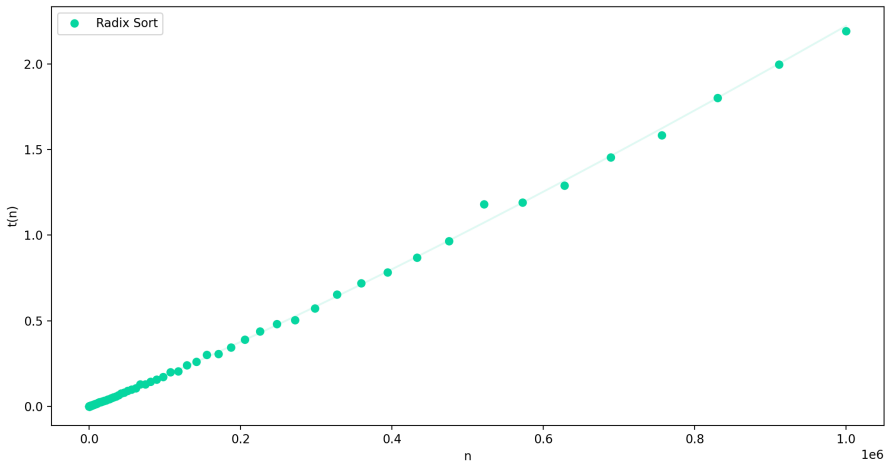

Figura 5.1: Performance del Radix Sort al variare di n.

5.2.2 Secondo esperimento (m varia)

Nel secondo esperimento, aumentando m (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 $O(d \cdot n)$.

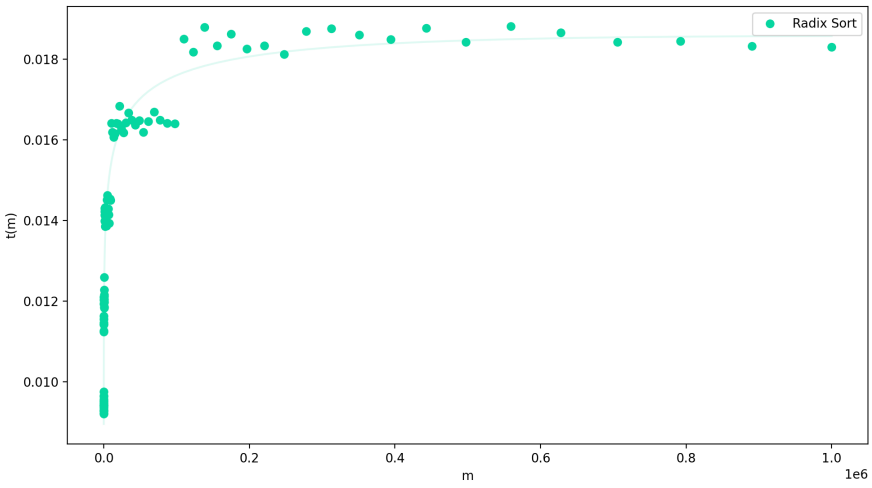

Figura 5.2: Performance del Radix Sort variare di m.

5.2.3 Terzo esperimento ($m = n^2$)

Nel terzo esperimento vediamo che, sebbene Radix Sort impieghi più tempo di Counting Sort quando m è costante, quando facciamo aumentare m come n^2 le parti si invertono drasticamente, visto che abbiamo un $\Theta(nlogn)$ contro un $\Theta(n^2)$. Vediamo che anche con vettori da 1000000 elementi Radix Sort rimane sotto i 6 secondi, mentre appena con vettori da 10000 elementi counting Sort supera gli 8 secondi. Comunque quando m cresce in questo modo, rimangono migliori algoritmi $O(nlogn)$ basati su scambi e confronti, come il Quick Sort.

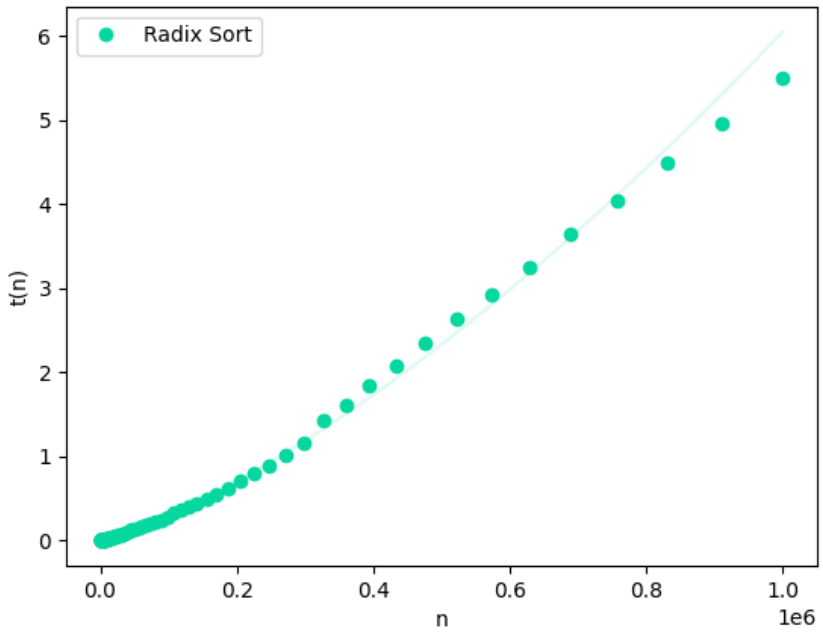

Figura 5.3: Performance del Radix Sort con $m = n^2$.

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 $k \gg n^2$.

Capitolo 6

Conclusioni

- mostriamo che asintoticamente con n fissato e un m che cresce sono meglio gli algoritmi basati su sbambi e confronti (grafici con tutti gli algorimti)
- mostriamo che asintoticamente con m fissato e un n che cresce sono meglio gli algoritmi lineari se le loro ipotesi sono soddisfatte
- (opzionale) cosa succede se facciamo variare m e n conteporaneamente?

- (opzionale) volendo potremmo aggiungere le immagini quicksort$_v s_3 way.png(checonfrontaquicksortequi$
  // grafico complessivo qui