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 Matematica e Informatica

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

Contenuti

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 algorimti al variare della dimensione dell'array 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 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 $c[i]$ rapprsenti 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 $k >> n$, 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 sperimentale

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 ≤ 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.

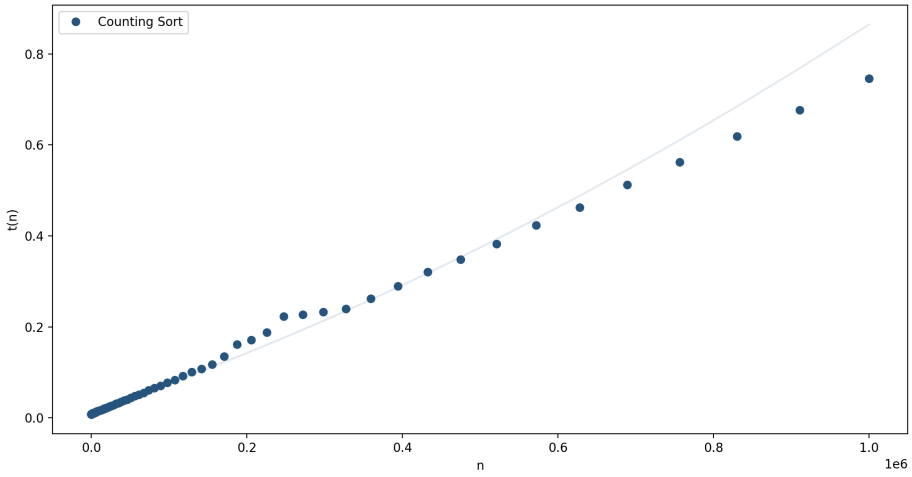


Figura 2.1: Performance del Counting Sort al variare di n.

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

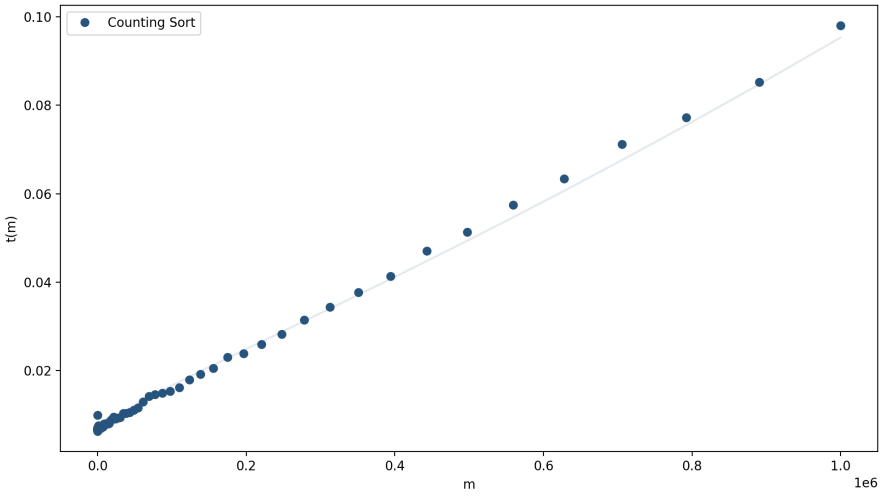


Figura 2.2: Performance del Counting Sort al variare di m.

Capitolo 3

Quick Sort

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.

3.1 Analisi della complessità

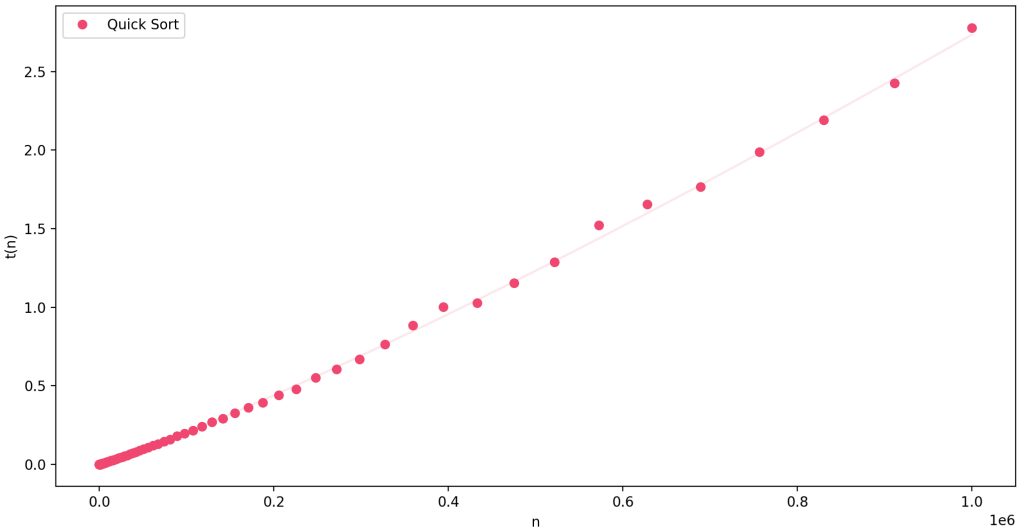
La complessità temporale di Quick Sort è:

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

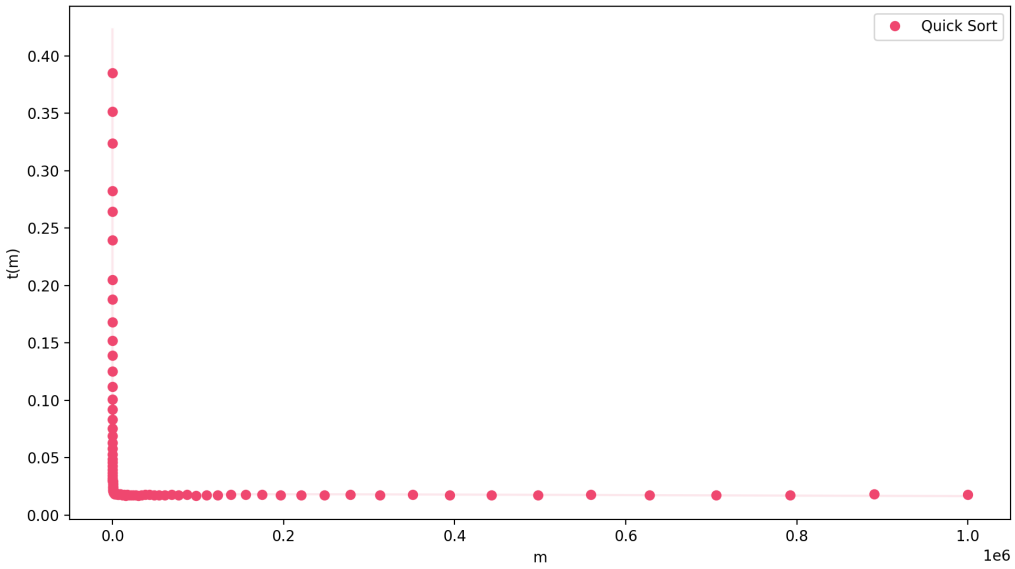
Nella pratica, il caso peggiore si verifica raramente se si utilizza un buon criterio di scelta del pivot (es. pivot casuale o mediana di tre).

3.2 Grafico dei tempi di esecuzione

Abbiamo generato un grafico in cui varia la lunghezza dell'array n da ordinare, mentre il paramentro 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, (m), mantenendo costante la sua lunghezza a 10000 ($n = 10000$). Nel grafico riportato qui sotto, sull'asse delle ascisse è rappresentata la variazione di 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.

Capitolo 4

Quick Sort 3 Way

TODO

Capitolo 5

Radix Sort

TODO

Capitolo 6

Conclusioni

TODO