Università degli Studi di Udine

DIPARTIMENTO DI SCIENZE MATEMATICHE, INFORMATICHE E FISICHE

Corso di Laurea in Internet Of Things, Big Data, Machine Learning

Laboratorio di Algoritmi e Strutture Dati

-

Verifica della complessità asintotica degli algoritmi di ordinamento

Candidati:
Andrea Gioia
(169484) - 169484@spes.uniud.it
Luca Gamberini
(168712) - 168712@spes.uniud.it
Kent Idrizi
(168711) - 168711@spes.uniud.it

Relatori:
Prof. Gabriele Puppis
Prof. Carla Piazza

Anno Accademico 2024–2025

Indice

1 Introduzione

Il progetto include le misurazioni e la conseguente graficazione di 4 algoritmi di ordinamento:

- QuickSort
- QuickSort 3 Way
- CountingSort
- RadixSort

1.1 Specifiche tecniche

Le specifiche tecniche delle misurazioni sono state:

- Array generato con valori decimali **casuali**
- La lunghezza dell'array è compresa tra 100 e 100 mila valori
- Il valore di ciascun elemento dell'array varia casualmente tra 10 e un milione

Gli algoritmi sono stati implementati in linguaggio **Python**.
Le misurazioni dei tempi sono state acquisite tramite la funzione *`perf_counter`* della libreria *time*.

2 Algoritmi

Gli algoritmi di ordinamento presi in esame presentano caratteristiche differenti e risultano più o meno efficienti in base alle carattestiche dell'array di elementi da ordinare.

2.1 QuickSort

Algoritmo di ordinamento ricorsivo del tipo divide-et-impera, quindi basato sulla suddivisione in n sottoproblemi, risolti ricorsivamente, fino al raggiungimento del caso base. Non presenta la necessità di utilizzo di strutture dati aggiuntive, di conseguenza lo scambio di elementi avviene in-place.

Idea
Divide: partizionando l'array A[p:r] in due sottoarray A[p:q-1] (parte inferiore) e A[q+1:r] (parte superiore) in modo che ciascun elemento della parte inferiore sia minore o uguale al pivot A[q], il quale è a sua volta minore o uguale a ciascun elemento della parte superiore. Calcolare l'indice q del pivot fa parte della procedura di partition.

Impera: richiamando ricorsivamente quicksort su ciascun sottoarray A[p:q-1] e A[q+1:r]. Infine per combinare non serve far nulla dato che i due sottoarray sono già ordinati, per cui avrò che l'intero sottoarray A[p:r] è ordinato. La procedura Partition invece mi permette di stabilire un perno, a quel punto avrò che gli elementi precedenti sono minori del perno mentre quelli a destra saranno maggiori, considerando però che successivamente vanno ordinati.

Le complessità asintotiche temporali di QuickSort sono:

- Caso ottimo: $\Omega(n \log_2 n)$
- Caso medio: $\Theta(n \log_2 n)$
- Caso peggiore: $\mathcal{O}(n^2)$

2.1.1 Analisi delle complessità

Il QuickSort è un esempio di algoritmo che lavora in-place, quindi la complessità in spazio equivale a $\Theta(n)$, ovvero la dimensione dell'array di partenza.

Le operazioni che influiscono sulla complessità di tempo sono:

- Chiamata a partition, complessità $\Theta(n)$
- Le due chiamate ricorsive, complessità rispettivamente di $T(q-1)$ e $T(n-q)$.

L'equazione di ricorrenza di QuickSort è:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ T(q-1) + T(n-q) + \Theta(n) & \textit{se } n > 1 \end{cases}$$

Spiegazione delle complessità:

- **Caso migliore:** il pivot divide sempre l'array in due parti, ovvero partition produce due sottoproblemi di dimensione al massimo $\frac{n}{2}$, dato che uno è di dimensione $\lfloor \frac{n-1}{2} \rfloor \leq \frac{n}{2}$ e l'altro di dimensione $\lceil \frac{n-1}{2} \rceil - 1 \leq \frac{n}{2}$. Un limite superiore al tempo di esecuzione è descritto da: $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$
- **Caso medio:** ogni possibile posizione del pivot nel caso medio ha probabilità $\frac{1}{n}$ di essere scelta. Per cui tutti i casi sono equiprobabili. Per un input casuale la complessità media è $T(n) = \frac{1}{n} \sum_{i=0}^{n-1} [T(i) + T(n-i-1)] + \Theta(n)$ che si risolve in $\mathbb{E}[T(n)] = \Theta(n \log n)$.
- **Caso peggiore:** quando la partizione produce un sottoproblema con $n-1$ elementi e uno con 0 elementi. Si può assumere che questa partizione sbilanciata avvenga ad ogni chiamata ricorsiva. La partizione costa $\Theta(n)$. Dato che la chiamata ricorsiva su un array di dimensione 0 ritorna senza fare nulla, $T(0) = \Theta(1)$, e l'occorrenza

del tempo di esecuzione è $T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n)$ con soluzione finale $T(n) = \Theta(n^2)$.
Dunque, se la partizione è massimamente sbilanciata in ogni livello ricorsivo dell'algoritmo, il tempo di esecuzione è $\Theta(n^2)$.

Quicksort non è stabile: funziona dividendo l'array in sottosequenze basate su un pivot, e poi riordinando ricorsivamente. Durante questo processo gli elementi uguali al pivot possono finire in posizioni diverse rispetto all'ordine originale; questo accade perché lo scambio degli elementi non tiene conto della loro posizione iniziale, ma solo del confronto con il pivot.

2.1.2 Codice

```
def QuickSort( A, p, q ):
    if( p < q ):
        r = Partition( A, p, q)
        QuickSort( A, p, r-1 )
        QuickSort( A, r+1, q )
    return A

def Partition(A, p, q):
    x = A[q]
    i = p - 1
    for j in range(p, q):
        if A[j] <= x:   # (corretto A[j], non A[q])
            i += 1
            Scambia(A, i, j)
    Scambia(A, i + 1, q)    # posiziona il pivot al centro
    return i + 1

def Scambia(A, i, j):
    temp = A[i]
    A[i] = A[j]
    A[j] = temp
```

Funzionamento del codice

1. **Funzione Scambia(A, i, j):** Lo scopo è scambiare due elementi in un array. A è l'array degli elementi, i e j gli indici degli elementi da scambiare. Memorizza temporaneamente il valore di A[i] in temp; Assegna A[j] alla posizione i; Ripristina il valore originale di A[i] (ora in temp) nella posizione j. Costo $\mathcal{O}(1)$.

2. **Funzione Partition(A, p, q):** Lo scopo è partizionare l'array in modo che tutti gli elementi minori uguali al pivot siano a sinistra e quelli maggiori del pivot a destra. A è l'array da partizionare, p l'indice iniziale del sottovettore, q l'indice final (usato come pivot). Si sceglie il pivot x = A[q] (ultimo elemento); l'indice

i tiene traccia della fine della sezione degli elementi minori o uguali al pivot; il ciclo for scorre l'array da p a q-1, Se A[j] è minore uguale al pivot, incrementa i e scambia A[i] con A[j]; infine scambia A[i+1] (primo elemento maggiore del pivot) con A[q] (pivot). Ora, il pivot è nella sua posizione corretta.

3. **Funzione QuickSort(A, p, q):** Lo scopo è ordinare ricorsivamente l'array utilizzando il partizionamento. A è l'array da ordinare, p l'indice iniziale e q l'indice finale. La condizione di base è se $p \geq q$, il sottovettore ha 0 o 1 elemento $\rightarrow$ già ordinato. $r = Partition(A, p, q)$ posiziona il pivot e restituisce la sua posizione finale. QuickSort(A, p, r-1) ordina la parte sinistra (elementi minori o uguali al pivot); QuickSort(A, r+1, q) ordina la parte destra (elementi maggiori del pivot).

2.2 QuickSort3Way

QuickSort 3-Way è un'ottimizzazione del QuickSort classico progettata per gestire efficientemente array con molti elementi duplicati. Mentre il QuickSort standard partiziona l'array in due sottoarray (elementi $\leq$ pivot ed elementi $>$ pivot), la versione 3-Way divide l'array in tre partizioni:

- Elementi minori del pivot (a sinistra).
- Elementi uguali al pivot (al centro).
- Elementi maggiori del pivot (a destra).

A differenza del QuickSort classico, QuickSort 3-Way presenta una notevole efficienza con array contenenti molti elementi duplicati.

Idea
Scelta del pivot: Il primo elemento dell'array (arr[l]) è selezionato come pivot.
Inizializzazione dei puntatori:

- lt (less than): confine sinistro degli elementi minori del pivot.
- gt (greater than): confine destro degli elementi maggiori del pivot.
- i: puntatore corrente per scorrere l'array.

Partizionamento 3-way:

- Se `arr[i]` $<$ `pivot`: scambia arr[i] con arr[lt], incrementa lt e i
- Se `arr[i]` $>$ `pivot`: scambia arr[i] con arr[gt], decrementa gt (senza incrementare i).
- Se `arr[i]` $==$ `pivot`: incrementa solo i.

Ricorsione:

- Applica ricorsivamente l'algoritmo alle partizioni sinistra (l a lt-1) e destra (gt+1 a r).

- La partizione centrale (lt a gt) contiene elementi già ordinati (tutti uguali al pivot).

Le complessità asintotiche temporali di QuickSort3Way sono:

- Caso ottimo: $\Omega(n \log n)$
- Caso medio: $\Theta(n \log n)$
- Caso pessimo: $\mathcal{O}(n^2)$

2.2.1 Analisi delle complessità

Come nel caso di QuickSort, anche il QuickSort3Way lavora in-place, senza bisogno di stutture dati aggiuntive, quindi la complessità in spazio è $\Theta(n)$.

Considerando che l'algoritmo itera gli elementi dell'array e siccome il ciclo for viene eseguito al massimo n volte.
L'algoritmo partiziona l'array in tre sezioni (elementi $<$ pivot, $=$ pivot e $>$ pivot) attraverso un singolo passaggio che opera in tempo lineare $\Theta(n)$. La complessità temporale dipende dall'equilibrio delle partizioni:

- **Caso ottimo** $\Omega(n)$: Tutti gli elementi sono uguali o la partizione centrale contiene tutti gli elementi (nessun elemento $<$ o $>$ del pivot). Dopo un singolo passaggio di partizionamento: partizione sinistra $\rightarrow$ vuota; partizione centrale $\rightarrow$ tutti gli elementi; partizione destra $\rightarrow$ vuota. Nessuna chiamata ricorsiva successiva. Costo singola scansione: $O(n)$.

- **Caso medio** $\Theta(n \log n)$: Partizionamenti bilanciati con pivot scelto casualmente. Il pivot divide l'array in tre partizioni di dimensioni approssimativamente: $\frac{n}{3}$ elementi $<$ pivot; $\frac{n}{3}$ elementi $=$ pivot; $\frac{n}{3}$ elementi $>$ pivot; Altezza albero di ricorsione: $O(log\ n)$. L'equazione di ricorrenza é $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{n}{3}\right) + O(n) = 2T\left(\frac{n}{3}\right) + O(n)$. Soluzione: $T(n) = \Theta(n \log n)$

- **Caso peggiore** $\mathcal{O}(n^2)$: Si verifica quando il pivot è sistematicamente l'elemento minimo o massimo dell'array (es. array già ordinato in senso crescente/decrescente). Ogni partizionamento produce: una partizione sinistra di dimensione 0, una partizione centrale di dimensione 1 (solo il pivot), una partizione destra di dimensione $n - 1$. L'altezza dell'albero di ricorsione diventa $O(n)$. Costo per livello: livello $0 \rightarrow O(n)$; livello $1 \rightarrow O(n-1)$... livello k $\rightarrow O(n-k)$. L'equazione di ricorrenza é $T(n) = T(0) + T(n-1) + O(n) = T(n-1) + O(n)$. Soluzione: $T(n) = O(n^2)$.

2.2.2 Codice

```
def QuickSort3Way(arr, l, r):
if l >= r:
    return

lt = l
i = l
gt = r
pivot = arr[l]

while i <= gt:
    if arr[i] < pivot:
        arr[lt], arr[i] = arr[i], arr[lt]
        lt += 1
        i += 1
    elif arr[i] > pivot:
        arr[i], arr[gt] = arr[gt], arr[i]
        gt -= 1
    else:
        i += 1

QuickSort3Way (arr, l, lt - 1)
QuickSort3Way (arr, gt + 1, r)

return lt, gt
```

Funzionamento del codice

1. **Caso Base (l $\geq$ r):** ferma la ricorsione se il sottoarray ha 0 o 1 elemento.
2. **Puntatori:** lt tiene traccia degli elementi minori del pivot; gt tiene traccia degli elementi maggiori del pivot; i scorre l'array da sinistra a destra.
3. **Scambi:** elemento minore: scambiato con arr[lt], incrementa lt e i; elemento maggiore: scambiato con arr[gt], decrementa gt;
   elemento uguale: incrementa solo i.
4. **Ricorsione:** chiamata ricorsiva sulla partizione sinistra (l a lt-1); chiamata ricorsiva sulla partizione destra (gt+1 a r).

2.3 CountingSort

Introduzione algoritmo

Il *Counting Sort* è un algoritmo di ordinamento non basato su confronti, adatto quando i valori da ordinare sono interi non negativi con intervallo di valori relativamente conte-

nuto rispetto al numero di elementi.

Idea
Contare le occorrenze di ciascun valore in un array ausiliario, calcolare una somma cumulativa (prefix sum) e quindi posizionare ogni elemento in output nel posto corretto per ottenere un ordinamento stabile.

2.3.1 Analisi delle complessità

Counting Sort assume che ciascuno dei n elementi in input sia un intero nell'intervallo $[0, k]$. L'algoritmo non utilizza confronti tra elementi, ma lavora sfruttando il valore numerico degli stessi come indice in un array ausiliario. Questo approccio consente di superare il limite inferiore $\Omega(n \log n)$ valido per gli algoritmi di ordinamento basati su confronti.
La complessità dell'algoritmo è la seguente:

- **Caso migliore:** Quando $k = O(n)$, cioè quando l'intervallo dei valori è proporzionale al numero di elementi, l'intero algoritmo opera in tempo lineare: $\Theta(n)$. In questo scenario, il Counting Sort è estremamente efficiente e utilizza $\Theta(n)$ spazio aggiuntivo.
- **Caso medio:** In situazioni intermedie, in cui k non è trascurabile ma nemmeno molto più grande di n, l'algoritmo ha una complessità di $\Theta(n+k)$. Il comportamento resta comunque più efficiente rispetto a molti algoritmi basati su confronti (come quicksort o mergesort), specialmente quando l'intervallo k rimane relativamente contenuto.
- **Caso peggiore:** Quando $k \gg n$, cioè l'intervallo dei valori è molto più ampio del numero di elementi, la complessità diventa $\Theta(n+k)$, ma con un impatto significativo in termini di memoria e tempo. L'array ausiliario $C[0 \dots k]$ può occupare molto spazio anche se pochi valori sono effettivamente presenti, portando a inefficienze.

Counting Sort è anche stabile: mantiene l'ordine relativo degli elementi con valore uguale, caratteristica fondamentale quando si lavora con dati associati (satellite data) o quando viene usato come sottoprocedura in algoritmi come il Radix Sort.
Questa stabilità è garantita dallo scorrimento dell'array originale in senso inverso durante la copia finale nell'array di output.

Codice

```
1     def countingSort(arr):
2     max = arr[0]
3     min = arr[0]
4
5     for i in range(1, len(arr)):
```

```
6           if arr[i] > max:
7               max = arr[i]
8           elif arr[i] < min:
9               min = arr[i]
10
11      C = [0] * (max - min + 1)
12      for i in range(len(arr)):
13          C[arr[i] - min] += 1
14
15      k = 0
16      for i in range(len(C)):
17          while C[i] > 0:
18              arr[k] = i + min
19              k += 1
20              C[i] -= 1
```

Funzionamento del codice

Il seguente frammento implementa una versione modificata del Counting Sort che supporta anche numeri negativi. Di seguito si spiega il funzionamento passo dopo passo:

1. **Individuazione del massimo e del minimo:** Si inizializzano due variabili `max` e `min` con il primo elemento dell'array. Successivamente, si scorre l'array per determinare il valore massimo e minimo effettivi. Questo passaggio è fondamentale per calcolare correttamente l'intervallo degli elementi e adattare l'algoritmo anche a numeri negativi.

2. **Inizializzazione dell'array dei conteggi:** Si crea un array ausiliario `C` di dimensione `(max - min + 1)`, inizializzato a zeri. Questo array verrà utilizzato per contare quante volte ogni valore appare nell'array originale. La sottrazione di `min` serve a traslare i valori negativi in indici validi (a partire da 0).

3. **Conteggio delle occorrenze:** Si scorre l'array originale `arr` e, per ogni valore `arr[i]`, si incrementa `C[arr[i] - min]`. In questo modo, ogni posizione dell'array `C` conterrà il numero di occorrenze del valore corrispondente.

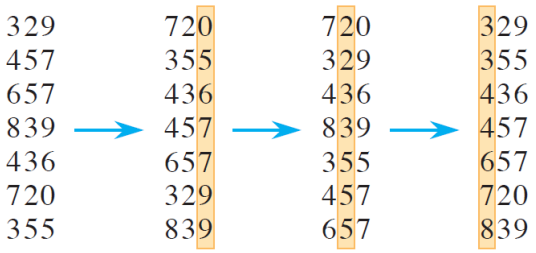
4. **Ricostruzione dell'array ordinato:** Con un doppio ciclo (for + while), si scorrono tutti i valori dell'array `C`. Per ogni indice `i`, finché `C[i]` è maggiore di zero, si inserisce il valore corrispondente (`i + min`) nell'array originale `arr` alla posizione `k`, incrementando `k` e decrementando `C[i]`. Questo processo ricostruisce l'array ordinato in modo diretto, sovrascrivendo l'input.

2.4 RadixSort

Introduzione algoritmo

RadixSort è un algoritmo di ordinamento basato sull'ordinamento cifra per cifra, partendo da quella meno significativa. Risulta particolarmente efficiente in presenta di molti numero con la stessa quantità di cifre.

Idea
RadixSort prevede di prendere in esame le singole cifre dei numeri da ordinare, in base alla loro posizione, andando successivamente a posizionarle in ordine crescente o decrescente.
Questo processo viene svolto ricorsivamente per ciascuna colonna di cifre, partendo dalla meno significativa.



Esempio di ordinamento

Radix Sort, per funzionare correttamente, richiede un algoritmo di ordinamento stabile per l'ordinamento delle singole cifre (o caratteri, a seconda del caso).

Solitamente, come algoritmo sottostante viene utilizzato Counting Sort, il quale, però, nella sua versione tradizionale non è stabile. Infatti, in presenza di chiavi uguali, Counting Sort potrebbe modificare l'ordine relativo degli elementi rispetto all'array di input.

Per garantire la stabilità, è sufficiente iterare l'array di input in ordine inverso durante la fase di costruzione dell'array di output. In questo modo, quando più elementi hanno la stessa chiave, essi verranno copiati nell'output nello stesso ordine in cui compaiono nell'input, preservando così la stabilità dell'ordinamento.

2.4.1 Analisi delle complessità

- **Caso ottimo:** Le condizioni del caso ottimo di RadixSort sono:
  - d piccolo;
  - $k < n$.

La complessità rimane $\Theta(d(n+k))$, tuttavia, con valori di d e k costanti si ottiene un $\Theta(n)$.

- **Caso pessimo:** Il caso peggiore si verifica nelle condizioni di:
  - d elevato (numeri molto grandi);
  - $k \geq n$ (rende CountingSort inefficiente).

  La complessità resta $\mathcal{O}(d(n+k))$, ma può degenerare fino a diventare $\mathcal{O}(n \cdot log(n))$

- **Caso medio:** Considerando n elementi da ordinare, ciascuno composto da d cifre, e assumendo che ogni cifra possa assumere al più k valori distinti, Radix Sort ha una complessità asintotica pari a $\Theta(d(n+k))$.
  Tuttavia, ciò è valido solo se l'algoritmo utilizzato per ordinare le singole cifre (come Counting Sort) ha complessità $\Theta(n+k)$, ovvero è lineare rispetto al numero di elementi e all'ampiezza del dominio delle cifre.

2.4.2 Motivazioni della scelta

É stato scelto RadixSort come quarto algoritmo a scelta per il suo approccio innovativo e diverso rispetto agli altri algoritmi selezionati, offrendo una prospettiva di confronto interessante.
Oltre a ciò, si tratta di un algoritmo utile e applicato in vari contesti reali.

RadixSort, inoltre, varia la efficienza in base all'algoritmo di ordinamento sottostante, il quale, nel nostro caso countingSort, deve essere stabile per preservare l'ordinamento parziale ottenuto nelle iterazioni precedenti. La stabilità di CountingSort garantisce che elementi con la stessa cifra nella posizione corrente mantengano l'ordine relativo stabilito nelle cifre precedenti, condizione essenziale per la correttezza dell'algoritmo. L'uso di CountingSort come subroutine conferisce a RadixSort una complessità lineare $\mathcal{O}(d \cdot n)$ poiché $k = 10$ (base fissa) è costante, rendendolo particolarmente efficiente per chiavi numeriche con un numero limitato di cifre d.

2.4.3 Codice

```
def RadixSort(arr):
    radix_array = [[], [], [], [], [], [], [], [], [], []]  # array delle cifre
    max_val = max(arr)  # assegno il valore massimo dell'array
    exp = 1

    while (max_val // exp) > 0:
        while len(arr) > 0:
            val = arr.pop()
            radix_index = (val // exp) % 10
            radix_array[radix_index].append(val)
```

```
11
12          for bucket in radix_array:
13              while len(bucket) > 0:
14                  val = bucket.pop()
15                  arr.append(val)
16
17          exp *= 10
```

Funzionamento del codice

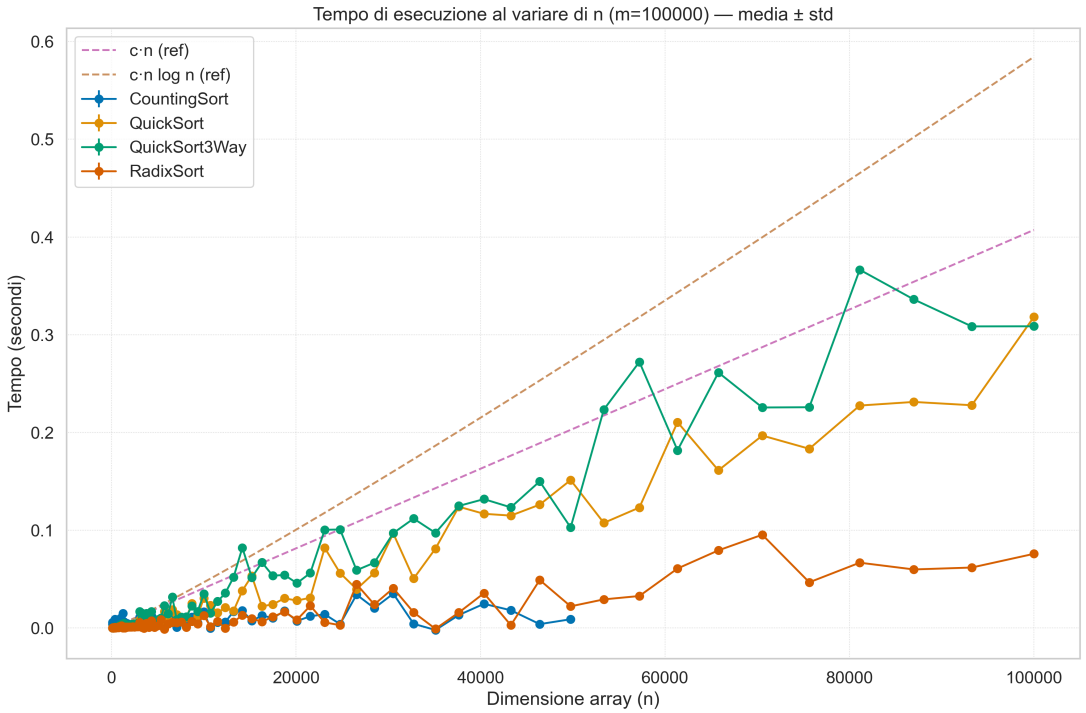
1. **Funzione GetMax(A):** Lo scopo è trovare il valore massimo nell'array. A è l'array di elementi. Inizializza `max` con il primo elemento; scorre tutti gli elementi dall'indice 1 alla fine; se `A[i] > max`, aggiorna `max` con `A[i]`. Restituisce il valore massimo. Costo $\mathcal{O}(n)$.

2. **Funzione CountingSortForRadix(A, exp):** Lo scopo è ordinare l'array in base a una specifica cifra. A è l'array da ordinare, `exp` è l'esponente che definisce la cifra (1, 10, 100...). Inizializza un array di output `B` di dimensione `n`; un array di conteggio `C` di dimensione 10 (base) inizializzato a zero; conta le occorrenze di ogni cifra in `(A[i] / exp) % 10` in `C`; modifica `C` in somme cumulative; costruisce `B` partendo dalla fine di `A` per stabilità; copia `B` in `A`. Costo $\mathcal{O}(n + k)$ con $k = 10$.

3. **Funzione RadixSort(A):** Lo scopo è ordinare l'array elaborando le cifre dalla meno alla più significativa. A è l'array da ordinare. Trova `m = GetMax(A)`; per ogni `exp` (da 1, a 10, a 100... finché `m/exp > 0`), applica `CountingSortForRadix(A, exp)`. All'ultima iterazione, l'array è ordinato. Costo $\mathcal{O}(d \cdot (n + k))$ dove d è il numero di cifre.

3 Misurazioni

3.1 Analisi delle Prestazioni al Variare della Dimensione dell'Array (Scala Lineare)



Tempo vs n (scala lineare). Tempo medio di esecuzione al variare di n (con $m = 100\,000$). Punti: medie su k run; barre: deviazione standard. Linee tratteggiate: curve teoriche di riferimento normalizzate su n_0 per confronto pratico.

3.1.0.1 Descrizione e obiettivo

Questo grafico mostra il tempo medio di esecuzione in funzione della dimensione dell'input n con m fissato a $100\,000$. Lo scopo è confrontare i tempi assoluti rilevati e visualizzare l'overhead pratico degli algoritmi.

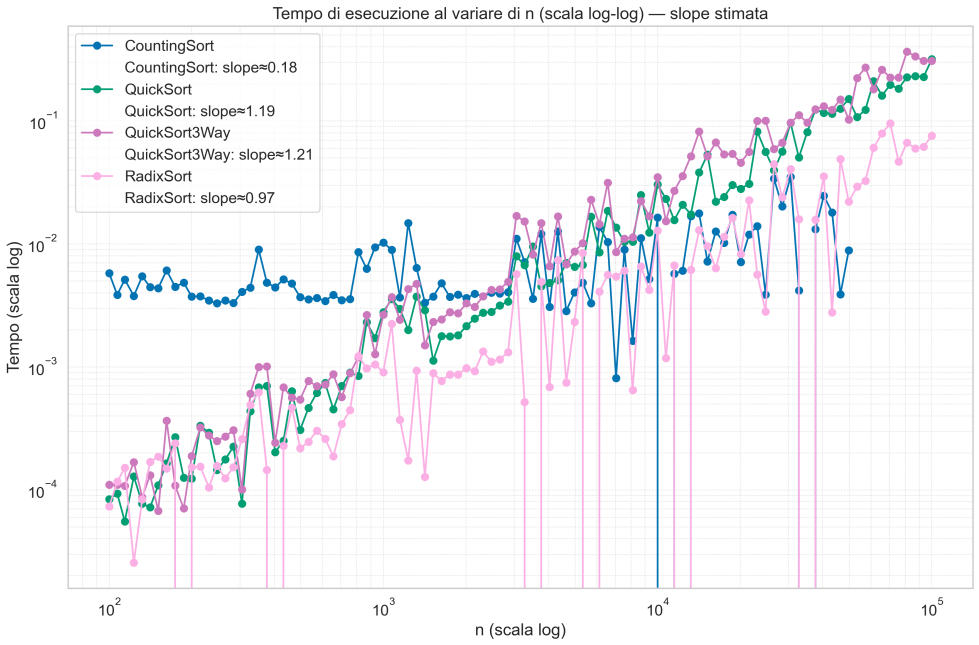
3.1.0.2 Osservazioni chiave

- I valori assoluti del tempo evidenziano chiaramente le differenze pratiche: algoritmi con complessità asintotica più favorevole (es. $O(n \log n)$) si collocano su tempi nettamente inferiori rispetto ad algoritmi quadratici per n sufficientemente grandi.
- Le curve teoriche di riferimento (linee tratteggiate) per $c \cdot n$ e $c \cdot n \log n$ sono state normalizzate su un punto di riferimento n_0 ed evidenziano che, sebbene due algoritmi possano avere lo stesso ordine asintotico, le costanti fanno la differenza nelle dimensioni realistiche dell'esperimento.

- Le barre di deviazione standard, se contenute, confermano la ripetibilità sperimentale; eventuali picchi o rumore debbono essere commentati (carico di sistema, garbage collection, variazioni nella generazione degli input).

3.1.0.3 Caveat interpretativi Non trarre conclusioni sull'ordine di crescita esclusivamente osservando la curvatura in scala lineare: curve asintoticamente diverse possono essere vicine nei range sperimentali limitati. Per questo motivo è utile affiancare il grafico con la versione log-log (Grafico 2).

3.2 Analisi delle Prestazioni al Variare della Dimensione dell'Array (Scala Log-Log)



Tempo vs n (scala log-log). Regressione lineare su log-log per stimare la pendenza empirica a tale che time $\approx Cn^a$. Valori di pendenza stimati riportati in legenda. Nota: per complessità $O(n \log n)$ la pendenza reale tende a 1.

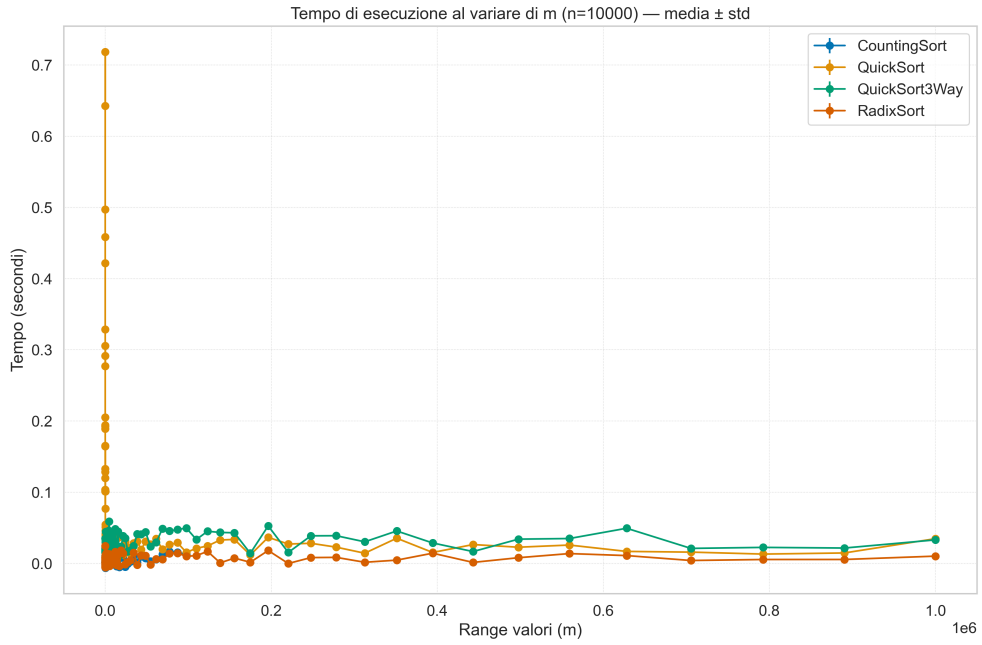
3.2.0.1 Descrizione e obiettivo La scala log-log è utilizzata per stimare la relazione di tipo potenza tra tempo e dimensione dell'input: una regressione lineare tra log(time) e log(n) fornisce una stima empirica dell'esponente effettivo.

3.2.0.2 Interpretazione della pendenza

- Se la regressione restituisce una pendenza a tale che $\log(\text{time}) \approx a \log n + b$, allora time $\approx C \cdot n^a$.

- Per complessità $O(n \log n)$ la pendenza a tende a 1 per n grandi (poiché $\log n$ cresce lentamente rispetto a potenze), dunque una pendenza empirica leggermente maggiore di 1 è coerente con $n \log n$ su intervalli finiti; non implica automaticamente un ordine polinomiale effettivo con esponente > 1.
- Una pendenza prossima a 2 corrisponderebbe a comportamento quadratico $O(n^2)$.

3.3 Impatto del Range dei Valori sulle Prestazioni (Scala Lineare)



Tempo vs m (scala lineare). Effetto del range dei valori m su algoritmi comparativi e non comparativi (con n fissato). Barre: deviazione standard. Nota: Counting Sort richiede memoria proporzionale a m.

3.3.0.1 Descrizione e obiettivo

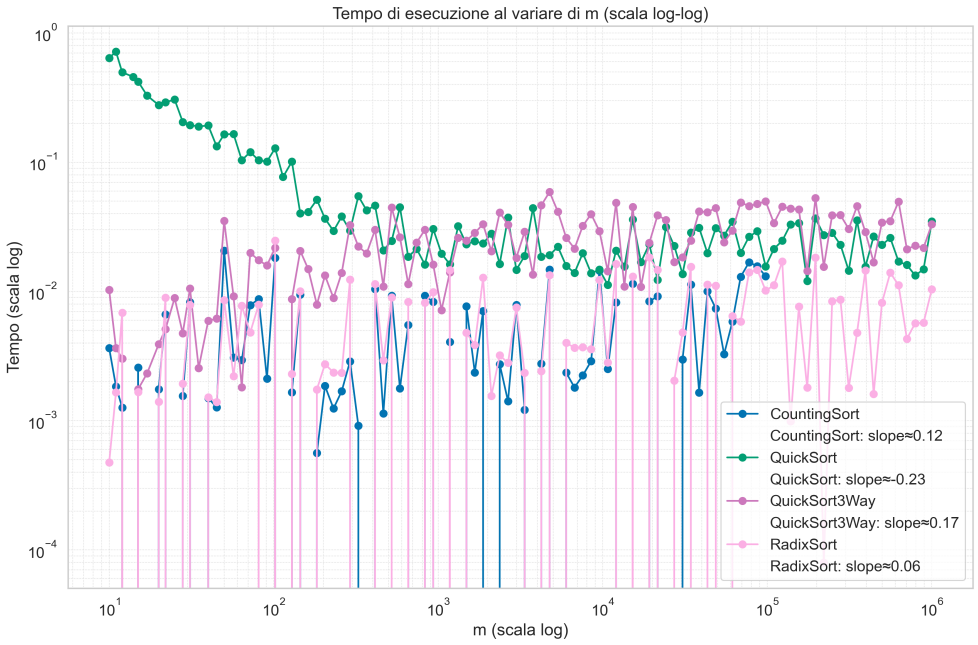
Questo grafico valuta l'impatto del range dei valori m (es. dominio dei numeri generati) sul tempo degli algoritmi, con n fissato. È particolarmente rilevante per algoritmi non comparativi come Counting Sort.

3.3.0.2 Osservazioni chiave

- **Counting Sort:** il tempo cresce sensibilmente con m a causa dell'allocazione e inizializzazione dell'array dei contatori di dimensione m. Anche se il numero di elementi n è piccolo, un m grande comporta costi non trascurabili.

- **Radix Sort:** dipende invece dal numero di cifre d e dalla base b: per chiavi con rappresentazione a lunghezza fissa e d costante, Radix può rimanere efficace anche per m grandi.
- Altri algoritmi comparativi (QuickSort, QuickSort 3-way) risultano relativamente insensibili al valore di m perché la loro complessità dipende principalmente da n.

3.4 Analisi dell'Impatto del Range dei Valori (Scala Log-Log)



Tempo vs m (scala log-log). Regressione log-log per stimare la dipendenza dal range m. Pendenze stimate riportate nella legenda. Counting Sort mostra tipicamente pendenza circa 1 dovuta all'azzeramento dell'array dei contatori.

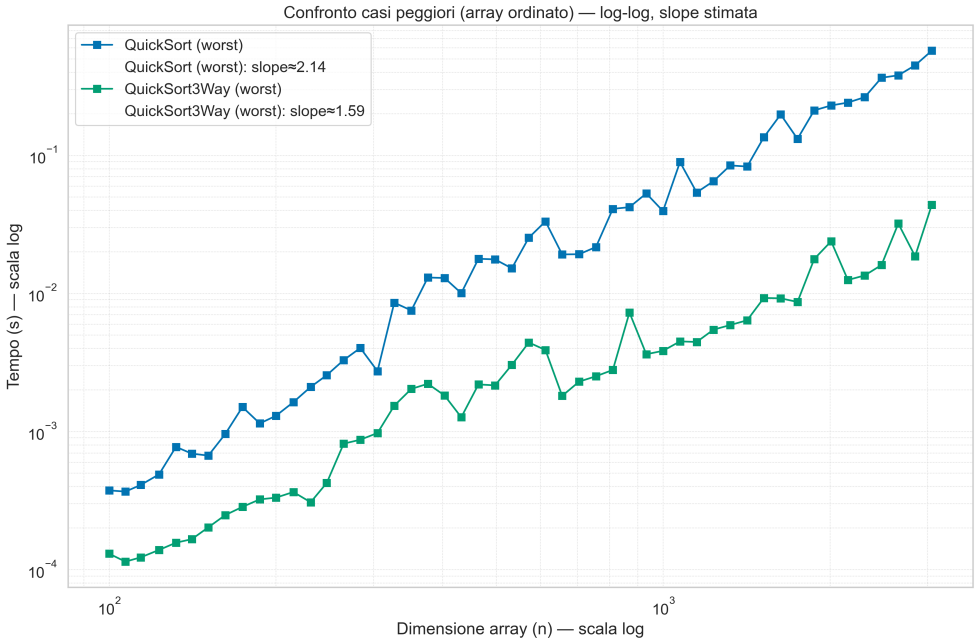
3.4.0.1 Descrizione e obiettivo

Versione log-log del Grafico 3: utile a stimare il comportamento asintotico rispetto a m (quando ha senso interpretare come potenza).

3.4.0.2 Interpretazione

- Una pendenza $a \approx 1$ per Counting Sort indica comportamento lineare rispetto a m (atteso per certe fasi dell'algoritmo come azzeramento dell'array dei contatori).
- Se Radix presenta pendenza molto più bassa o platea, ciò indica che il costo non cresce direttamente con m ma è legato a parametri diversi (numero di cifre d, base b).

3.5 Analisi dei Casi Peggiori per gli Algoritmi QuickSort



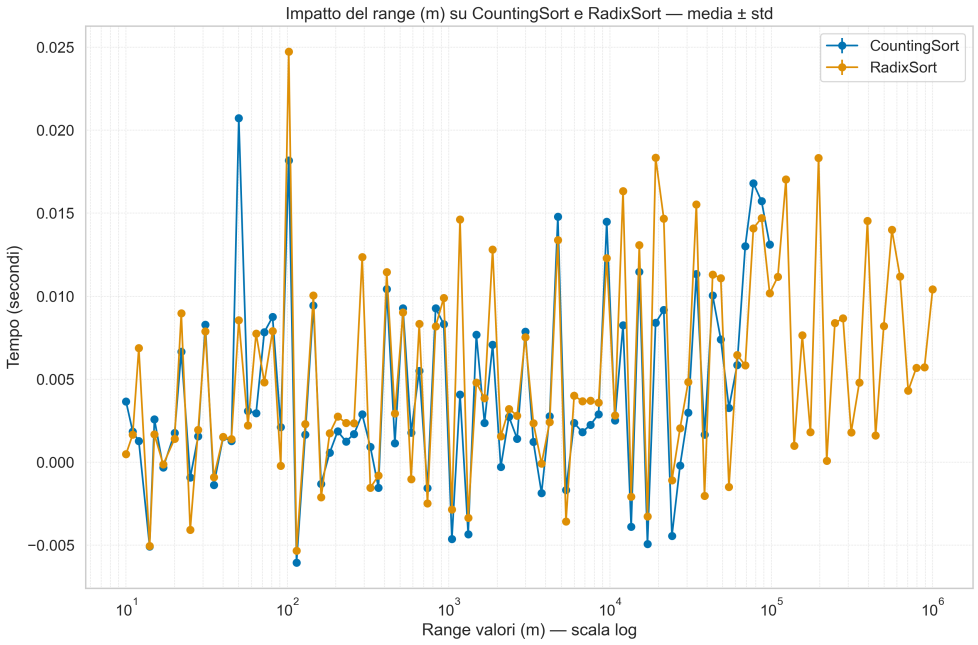
Confronto casi peggiori (log-log). Prestazioni su input ordinato; regressioni log-log usate per stimare l'esponente empirico. QuickSort classico mostra il peggior comportamento se non protetto da scelte pivot robuste.

3.5.0.1 Descrizione e obiettivo Questo grafico confronta le prestazioni degli algoritmi sul caso avverso (ad esempio array già ordinato), con l'obiettivo di evidenziare degradazioni asintotiche, in particolare del QuickSort classico.

3.5.0.2 Osservazioni e interpretazioni

- **QuickSort classico:** il pivot mal scelto può causare la ricorrenza degenerata $T(n) = T(n-1) + \Theta(n)$ e quindi $T(n) = \Theta(n^2)$; su log-log ciò si riflette con una pendenza vicino a 2.
- **QuickSort 3-way o pivot randomizzato:** mostrano pendenze più vicine a 1 (o lievemente sopra) anche sullo stesso set di input, a dimostrazione della robustezza della scelta pivot o della tecnica 3-way in presenza di molteplici valori uguali.

3.6 Analisi Comparativa dell'Impatto del Range su Algoritmi Non Comparativi



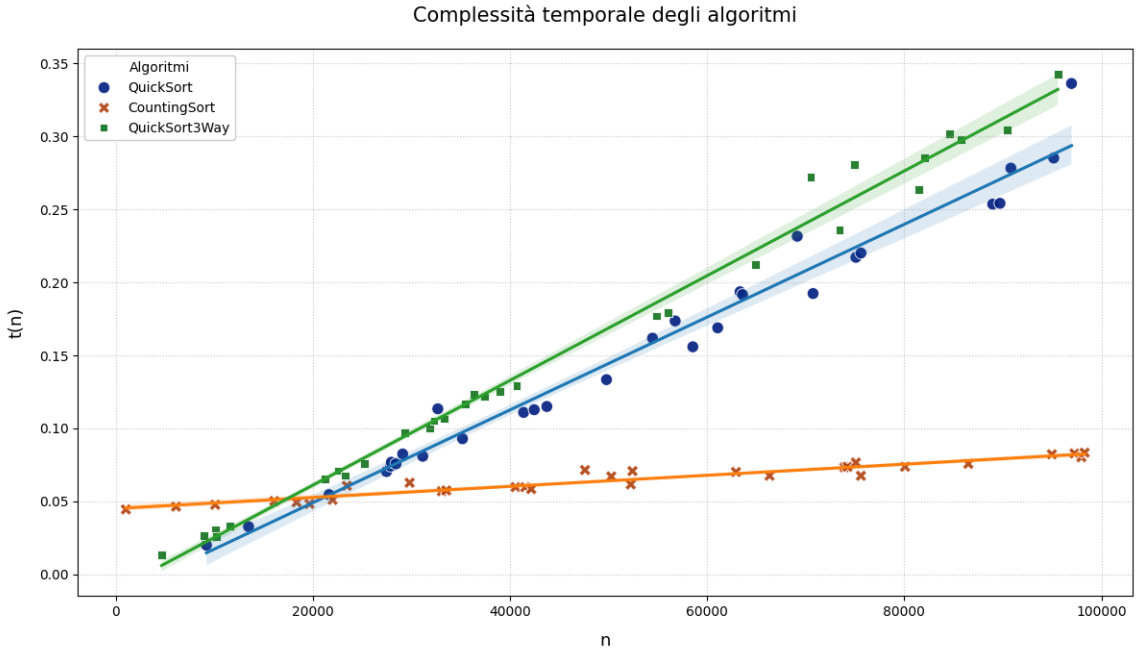
Impatto di m su CountingSort e RadixSort (asse m logaritmico). Confronto diretto che mostra il passaggio di regime: Counting Sort è competitivo per m piccoli, ma diventa inefficiente per m grandi per motivi di memoria e overhead di inizializzazione. Radix mostra migliore scalabilità nel caso di chiavi con numero di cifre limitato.

3.6.0.1 Descrizione e obiettivo

Grafico comparativo per mettere a confronto direttamente Counting Sort e Radix Sort su un ampio intervallo di m, con asse m in scala logaritmica per coprire molti ordini di grandezza.

3.6.0.2 Regimi osservati

- **Regime a piccoli m:** Counting Sort è spesso più veloce (costanti favorevoli e semplicità). Radix può avere overhead maggiore a causa delle iterazioni per cifra.
- **Regime a grandi m:** Counting Sort penalizza fortemente per la memoria e i costi di inizializzazione; Radix tende a scalare meglio quando d (numero di cifre) rimane moderato.
- **Soglia pratica:** nel grafico è visibile (ad es. attorno a $m \approx 10^5$) il punto in cui Counting Sort diventa meno pratico; questo valore dipende dall'implementazione e dalla memoria disponibile.



Parte reale sulle x, parte immaginaria sulle y

4 Conclusioni

4.0.0.1 Sintesi dei risultati. Gli esperimenti condotti confermano le attese teoriche sulle famiglie di algoritmi considerate. Tra gli algoritmi *comparativi*, `QuickSort` e `QuickSort 3-way` presentano tempi che, al crescere della dimensione dell'input n, sono compatibili con una complessità $O(n \log n)$, mentre un algoritmo quadratico di riferimento (se considerato) degrada rapidamente. Tra gli algoritmi *non comparativi*, `Counting Sort` e `Radix Sort` evidenziano i loro punti di forza: il primo è lineare in n quando il range dei valori m è moderato (complessità $O(n + m)$ e spazio $O(m)$), il secondo è efficace quando le chiavi hanno lunghezza fissa (complessità $\Theta(d(n + k))$ con d numero di cifre e k la base/bucket), risultando poco sensibile a m se d rimane contenuto.

4.0.0.2 Validazione empirica. I grafici in scala log–log (tempo vs n e tempo vs m) hanno fornito una verifica quantitativa: le pendenze stimate per `QuickSort`/`QuickSort 3-way` sono prossime a 1 (coerenti con $n \log n$ su intervalli finiti), quelle di `Counting Sort` rispetto a m sono vicine a 1 (costo di inizializzazione dell'array dei contatori), e il *caso peggiore* di `QuickSort` su input ordinato mostra un comportamento compatibile con $O(n^2)$ (pendenza prossima a 2). Le barre d'errore ridotte indicano stabilità delle misure; eventuali variazioni localizzate sono attribuibili al rumore di sistema e alla generazione degli input.

4.0.0.3 Trade-off pratici. Oltre all'ordine asintotico, emergono differenze concrete dovute a costanti e gestione della memoria. `QuickSort` è tipicamente il più rapido in media e in-place (stack atteso $O(\log n)$), ma non è stabile e può degradare se la scelta del pivot è sfavorevole; tecniche come pivot randomizzato, median-of-three e partizionamento 3-way riducono drasticamente tale rischio. `QuickSort 3-way` funziona su qualsiasi tipo di dati ordinabili però meno efficiente su dati interi limitati. `Counting Sort` è stabilmente lineare quando $m = O(n)$, ma lo spazio $O(m)$ ne limita l'uso per range molto ampi. `Radix Sort`, usando una routine stabile (ad es. `Counting Sort`) come subprocedura, scala bene su chiavi a lunghezza fissa, con costi che dipendono da d più che da m.

4.0.0.4 Linee guida operative.

- **Uso generale su chiavi generiche:** preferire `QuickSort` con pivot robusto (random/median-of-three) o `MergeSort` quando serve stabilità.
- **Molte duplicazioni:** `QuickSort 3-way` riduce il lavoro sui segmenti di uguali.
- **Chiavi intere in range moderato:** `Counting Sort` è ideale (tempo $O(n + m)$, stabile), purché la memoria $O(m)$ sia sostenibile.
- **Chiavi a lunghezza fissa/byte-string:** `Radix Sort` è competitivo e poco sensibile a m, specialmente con base e d scelti con cura.

4.0.0.5 Limiti e minacce alla validità. Le misure dipendono dall'hardware, dal sistema operativo, dalla gestione del *runtime* e dalla distribuzione degli input. Sebbene si siano effettuate più ripetizioni e si siano usate scale logaritmiche per mitigare l'effetto del rumore e stimare gli esponenti, restano possibili bias dovuti a cache, branch prediction, allocazioni di memoria e dettagli implementativi. I risultati vanno letti come conferme sperimentali *coerenti* con la teoria, non come stime assolute universalmente trasferibili.

4.0.0.6 Conclusione finale. In sintesi, l'analisi congiunta teorico–sperimentale mostra che: (i) gli algoritmi comparativi ben progettati (`QuickSort` con scelte di pivot robuste e `MergeSort`) sono la scelta predefinita su input generici grazie a prestazioni $O(n \log n)$ affidabili; (ii) in scenari strutturati, gli algoritmi non comparativi (`Counting`/`Radix`) offrono vantaggi decisivi, purché le ipotesi (m moderato, chiavi a lunghezza fissa, stabilità della subroutine) siano soddisfatte; (iii) la selezione dell'algoritmo deve considerare non solo l'asintotica, ma anche stabilità, consumo di memoria e caratteristiche del dominio dei dati. Questi risultati forniscono un criterio pratico per scegliere consapevolmente l'algoritmo più adatto al contesto applicativo.