Algoritmi di ordinamento

Grossi Mara

Maggio 2025

1 Introduzione

Per questo progetto è stata richiesta l'implementazione di quattro algoritmi di ordinamento e lo studio dei loro tempi di esecuzione, con conseguente confronto fra di essi mediante grafici singoli e comparativi. Gli algoritmi proposti sono quelli di `countingSort`, `quickSort`, `quickSort 3 Ways` e uno a scelta, in questo caso `insertionSort`. La scelta di `insertionSort` è dovuta al mero interesse personale di vedere quanto varino effettivamente i tempi di esecuzione tra algoritmi con complessità temporale lineare e logaritmica rispetto a un algoritmo con complessità quadratica. Gli algoritmi vengono approfonditi nella sezione 3. Lo studio è avvenuto generando campioni al variare dei parametri n ed m che, rispettivamente, indicano la lunghezza dell'array e il range dei valori di input. Nella sezione 2 vengono descritti i criteri con cui sono stati scelti i parametri.

2 Generazione dei parametri

Per l'analisi degli algoritmi sono stati generati degli array sulla base di due parametri, n ed m. Il parametro n descrive la lunghezza che deve avere l'array e varia in un range di valori tra 100 e 100000. Il parametro m, invece, rappresenta in quale range di interi fanno parte gli elementi dell'array e varia tra 10 e 1000000. È stato necessario generare l'input con una certa variabilità al fine di analizzare il comportamento dei vari algoritmi al variare dei parametri. Per questo scopo, quindi, sono stati generati 100 campioni dove vengono rispettivamente fissati n ed m. Per i campioni con n variabile è stato fissato $m = 100000$, mentre per m variabile è stato fissato $n = 10000$. Gli array generati sono distribuiti secondo due funzioni esponenziali rispetto ad i, una per $n \in [100, 100000]$ e l'altra per $m \in [10, 1000000]$:

$$n_i = \lfloor AB^i \rfloor, i \in [0, 99], A = 100, B = e^{\frac{\log(100000)-\log(100)}{99}}$$

$$m_i = \lfloor AB^i \rfloor, i \in [0, 99], A = 10, B = e^{\frac{\log(1000000)-\log(10)}{99}}$$

Per la funzione `generateArray` si è ritenuto opportuno l'uso di un generatore con un valore di inizializzazione non deterministico al fine di garantire maggiore casualità e non-predicibilità nella sequenza dei numeri generabili.

3 Presentazione degli algoritmi

3.1 CountingSort

Implementazione

Dato in input l'array `toOrderArray` da ordinare, introduciamo altri due vettori di supporto `count` e `aux`. In `count` vengono salvate il numero di occorrenze degli elementi di `toOrderArray` nella posizione corrispondente a tale valore. Nonostante `count` indichi la frequenza di ogni elemento in ordine, non è possibile riscrivere tali valori nell'array di input perché andremmo semplicemente a sovrascrivere tale array senza effettuare un vero ordinamento, rischiando inoltre di perdere informazioni che tali elementi potrebbero portare con sé. Modifichiamo quindi `count` in modo che in ogni cella venga salvata la somma cumulativa degli elementi precedenti, andando così a calcolare la posizione che avrebbe l'ultima occorrenza del valore della cella corrispondente. In `aux` vengono salvati gli elementi di `toOrderArray` in ordine crescente attraverso l'ausilio di `count`: partendo dall'ultimo elemento di `toOrderArray`, si trova la posizione in cui quell'elemento andrebbe inserito in `aux` (ovvero `count[toOrderArray[i]]`), decrementando poi tale posizione in `count` in modo da aggiornare l'ultima occorrenza (`count[toOrderArray[i]]--`). Infine `aux` viene copiato in `toOrderArray`.

Complessità

`Countingsort` è un algoritmo di ordinamento che lavora in tempo lineare, ma necessita di ipotesi sull'input. La sua complessità, $\Theta(k+n)$ per $k \in \mathbb{N}$, dipende dal parametro k, il quale rappresenta la dimensione dell'array `count` e, per valori di k di ordine maggiore a n, può portare a complessità elevate. Per ovviare almeno in parte a questo problema sono state definite le funzioni `real_max` e `real_min` che calcolano, rispettivamente, il massimo e il minimo valore in `toOrderArray` in modo da ridurre, seppur in minima parte, la dimensione di `count`.

3.2 QuickSort

Implementazione

Dato in input l'array `toOrderArray` da ordinare, a seguito di un controllo iniziale, viene richiamata la funzione `partition_1` su quest'ultimo agli indici iniziale e finale. Dopo la scelta dell'elemento pivotale, che in questo caso ricade sull'ultimo elemento dell'array, la funzione opera con lo scopo di individuare l'indice in cui tale elemento si posizionerebbe se l'array fosse ordinato. Quindi, mediante un ciclo *for*, vengono spostati gli elementi dell'array in modo che, una volta inserito il pivot, gli elementi minori di questo si trovino alla sua sinistra e i restanti alla sua destra. Ovviamente il riposizionamento di tali elementi non garantisce che siano ordinati. Viene dunque restituita la posizione del pivot e

`quickSort` viene richiamato sui due sottoarray individuati a sinistra e destra del pivot.

Complessità

`quickSort` opera in tempo $O(n^2)$, ma mediamente la sua complessità è $\Theta(n \log n)$. `partition_1` ha invece complessità lineare. Il caso peggiore si verifica quando l'array è ordinato in quanto il pivot si trova già nella posizione corretta, ma verrebbe confrontato con tutti i restanti elementi e l'algoritmo verrebbe richiamato ogni volta su un array vuoto e uno di lunghezza $|toOrderArray| - 1$. Questo porta a una complessità quadratica. Attraverso un algoritmo di selezione, quale `median of medians`, è possibile ricercare il pivot migliore in modo da migliorare la complessità di `quickSort`.

3.3 QuickSort 3 Ways

Implementazione

Dato in input l'array `toOrderArray` da ordinare, a seguito di un controllo iniziale, viene richiamata la funzione `partition_2` su quest'ultimo agli indici iniziale e finale. Dopo aver scelto il pivot, in questo caso l'elemento in posizione $\frac{high+low}{2}$, l'array viene partizionato in tre parti: gli elementi minori, uguali e maggiori del pivot. Il partizionamento avviene con l'ausilio di tre variabili che tengono traccia della fine delle varie sezioni. A questo scopo vengono introdotte `less` e `greater` le quali, rispettivamente, tengono traccia delle sezioni con elementi minori e maggiori del pivot. La variabile `equal` viene invece introdotta per scorrere l'array. Tramite un ciclo *while* avvengono gli scambi che portano a identificare le tre sezioni. Alla fine vengono restituiti gli indici `l` e `g` su cui richiamare la ricorsione, ovvero `toOrderArray[low, .., l - 1]` e `toOrderArray[g + 1, .., high]`.

Complessità

La complessità di `quickSort 3 Ways`, come per `quickSort`, è $\Theta(n \log n)$ nel caso medio. La differenza tra i due algoritmi sta nel modo in cui vengono gestiti i duplicati nell'array, come spiegato nella sezione 6. Nel caso peggiore la complessità è dell'ordine quadratico e lo si ritrova ogni qualvolta l'array non viene partizionato correttamente e le chiamate ricorsive avvengono su una sezione di un singolo elemento e l'altra sui restanti. L'algoritmo rimane comunque più efficiente del precedente grazie al partizionamento in tre sezioni.

3.4 InsertionSort

Implementazione

Dato in input l'array `toOrderArray` da ordinare, applichiamo l'algoritmo. Partendo dall'indice 1, `InsertionSort` procede con l'ipotesi che all'*i-esimo* passo

l'array `toOrderArray[0 .. i - 1]` sia ordinato. A ogni passo `toOrderArray[i]` viene confrontato con i suoi elementi precedenti, i quali vengono riposizionati verso destra. Ciò si ripete finché non si trova la sua posizione corretta. Al termine dell'iterazione corrente, `toOrderArray[0 .. i]` risulta ordinato.

Complessità

`insertionSort` opera in tempo $O(n^2)$. La sua complessità può diventare lineare qualora l'array fosse già ordinato in quanto il controllo del ciclo *while* risulterebbe sempre negativo e quindi non verrebbe eseguito.

4 Tempi di esecuzione

Per lo studio dei tempi di esecuzione sono stati generati 100 campioni rispettivamente per n ed m variabili. Inizialmente, attraverso la funzione `getResolution`, viene calcolata la risoluzione del clock di sistema, ovvero la più piccola unità di tempo che il clock riesce a misurare. La stima dei tempi di esecuzione deve garantire un errore relativo minimo E pari a 0.001. In funzione della risoluzione ottenuta R e dell'errore relativo minimo E, viene calcolato il tempo minimo misurabile:

$$T_{min} = R \cdot (\frac{1}{E} + 1)$$

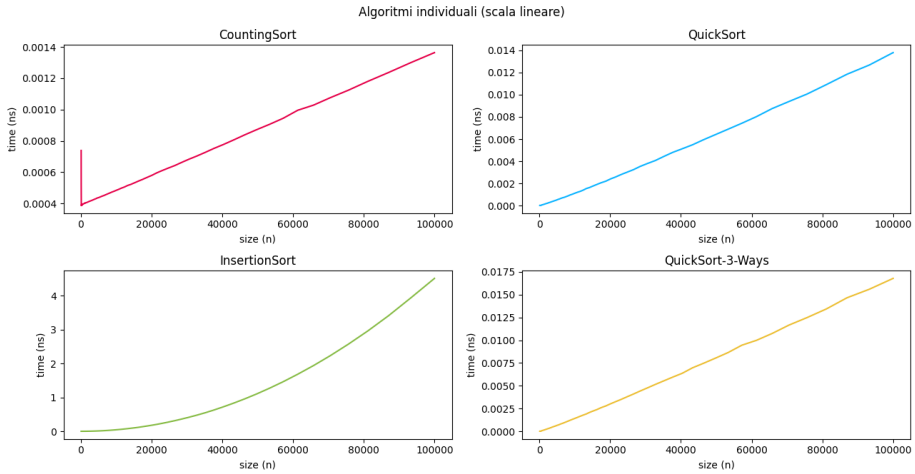
che verrà successivamente usato in `measure`. Tramite questa funzione vengono stimati i tempi medi dei vari algoritmi. Durante il ciclo *while* viene inizializzato un array mediante la funzione `generateArray`, il quale viene poi ordinato dall'algoritmo in questione. Questo ciclo continua a calcolare i tempi di esecuzione fin quando la somma cumulativa di questi non supera T_{min}. Per ottenere una stima più accurata, questo procedimento viene svolto 10 volte tramite un ciclo *for* esterno. Tramite una variabile `count` viene memorizzato il numero di volte che il ciclo *while* viene eseguito. In questo modo possiamo calcolare il tempo medio di un singolo ordinamento, il quale viene salvato in una variabile ausiliaria `times`. La funzione restituisce la mediana dei valori salvati in suddetta variabile. La scelta della mediana piuttosto che della media è avvenuta in seguito all'analisi dei grafici ottenuti dalle varie misurazioni. Inizialmente questi ultimi presentavano picchi nel loro andamento, dovuti probabilmente all'input e agli outliers. Da questo secondo problema è risultato più opportuno il calcolo della mediana. Mediante un secondo clock sono stati misurati anche i tempi medi per la generazione dei singoli array. Questi tempi sono stati sottratti dal runtime totale in modo da ottenere solamente quelli degli algoritmi. I grafici ottenuti da queste misurazioni verranno trattati nelle sezioni 5 e 6.

5 Analisi per n variabile

In questa sezione verranno esposti i grafici riguardanti i tempi di esecuzione dei vari algoritmi per m fissato ed n variabile.

5.1 Grafici individuali

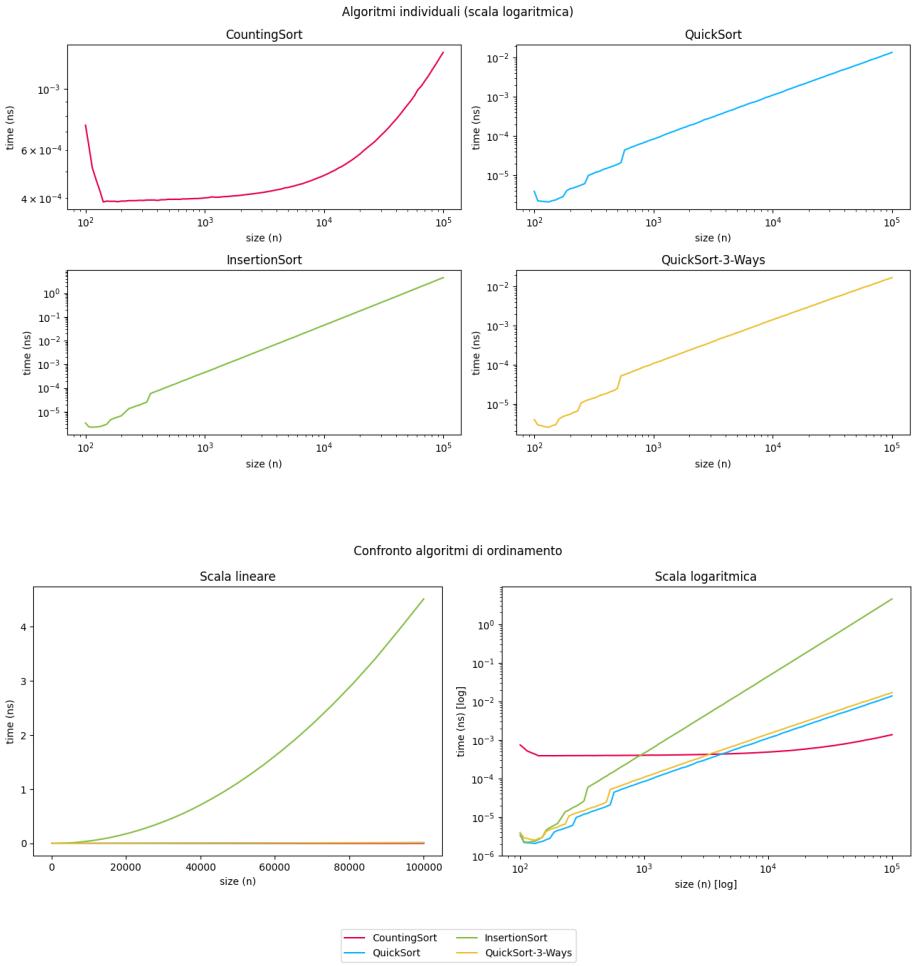
Scala lineare



Come si evince dai grafici, il tempo di esecuzione dei vari algoritmi aumenta all'aumentare della dimensione dell'input. Nonostante attraverso la scala lineare non si riescano a evidenziare le proprietà dei vari algoritmi, si riesce comunque a vedere l'andamento quadratico dell'algoritmo di `insertionSort`. Il picco iniziale mostrato in `countingSort`, invece, potrebbe essere spiegato da altri processi che erano in esecuzione durante la misurazioni dei tempi e potrebbero aver introdotto degli errori. Un altro motivo lo si può trovare nella scala dell'asse delle ordinate usata, in quanto i tempi sono molto vicini tra di loro e anche una leggera variazione di questi risulta molto evidente, nonostante siano in linea con gli altri.

Scala logaritmica

Come per la scala lineare, anche in quella logaritmica è evidente un certo incremento all'aumentare della dimensione dell'input. A differenza della prima, invece, per `countingSort` è più evidenziato l'incremento all'aumentare della dimensione. Per quanto riguarda i restanti algoritmi, l'andamento quadratico di `insertionSort` non risulta più marcato come in precedenza, mentre per `quickSort` e `quickSort 3 Ways` non si riescono ancora a intravedere i loro effettivi andamenti, cosa che sarà invece più visibile per i grafici comparativi nella sezione 5.2.



5.2 Grafici comparativi

A differenza dei grafici individuali, attraverso quelli comparativi sono ben evidenti gli andamenti dei vari algoritmi. Dalla scala lineare è possibile vedere come `insertionSort` operi in tempi molto più lunghi rispetto agli altri algoritmi, confermando difatti che non sia un ottimo algoritmo di ordinamento. Dalla scala logaritmica, invece, si evince che `quickSort` e `quickSort 3 Ways` presentano gli stessi tempi di esecuzione, quindi per n variabile il loro andamento sembra essere simile, cosa che invece non avverrà per n fissato (sezione 6). L'algoritmo di `countingSort` sembra essere quello più efficiente quando m è fissato, presentando un andamento quasi lineare, sebbene con un input abbastanza grande inizia ad accennare anche lui a una minima crescita.

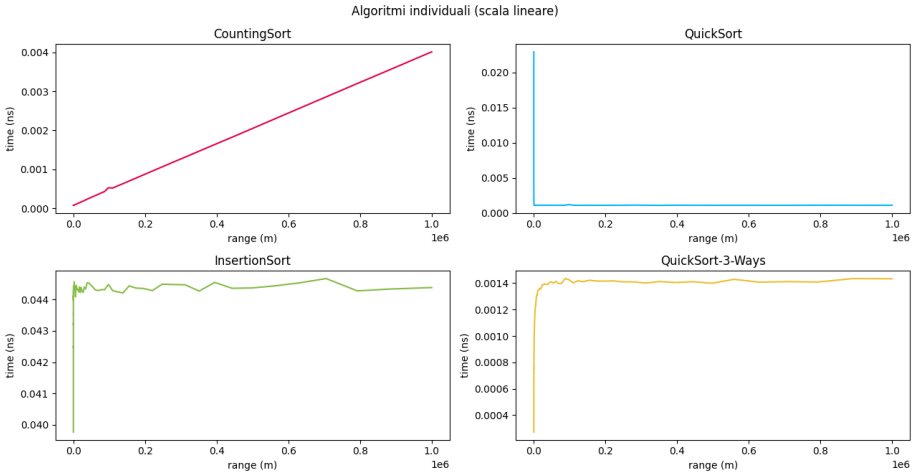
6 Analisi per m variabile

In questa sezione verranno esposti i grafici riguardanti i tempi di esecuzione dei vari algoritmi per n fissato e m variabile.

6.1 Grafici individuali

I grafici per m variabile risultano più significativi rispetto ai precedenti e riescono a mettere in luce le proprietà dei singoli algoritmi.

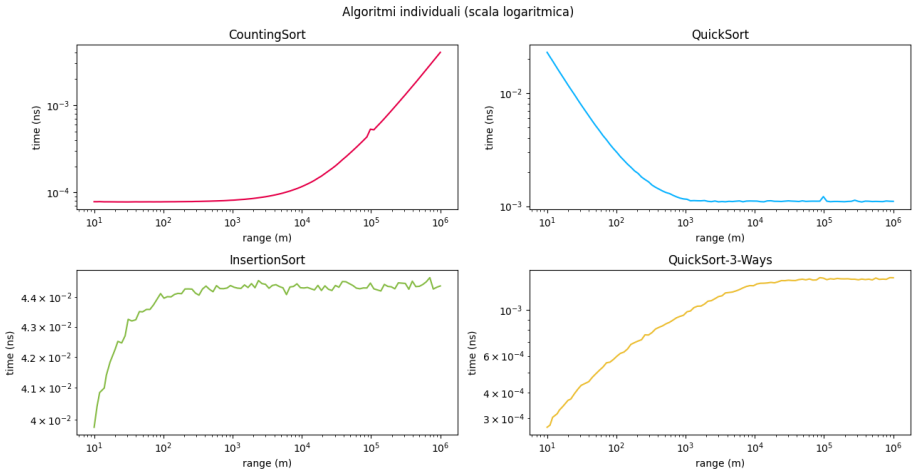
Scala lineare



L'algoritmo di `insertionSort` presenta un andamento quasi lineare, dovuto al fatto che questo algoritmo è più sensibile al crescere della dimensione dell'array, piuttosto che ai valori presenti. Come anche preannunciato nella sezione 3.1, la complessità di `countingSort` dipende dai valori passati in input quindi, al crescere di questi, anche i tempi di esecuzione tendono a crescere. Interessanti sono invece gli andamenti di `quickSort` e `quickSort 3 Ways`, che vengono spiegati nella sezione 6.1.

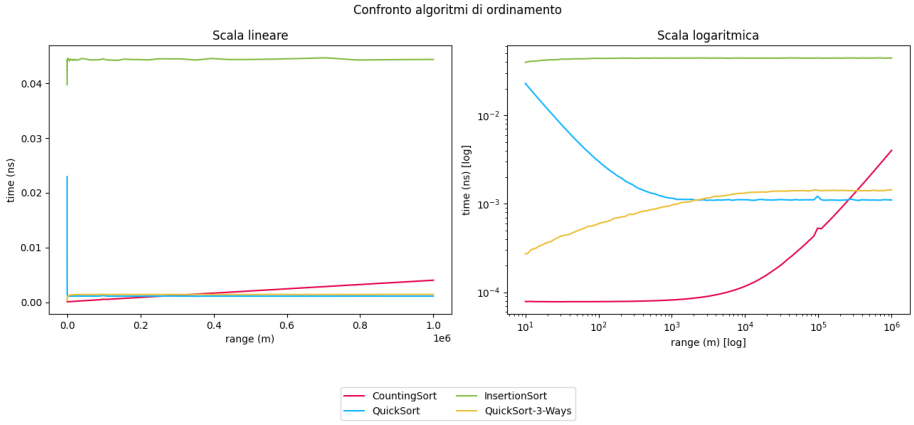
Scala logaritmica

Grazie alla scala logaritmica è possibile notare come gli algoritmi `quickSort` e `quickSort 3 Ways` lavorino in modi differenti in base all'input. Il primo presenta una peggiore performance quando m risulta piccolo in quanto sono presenti più elementi ripetuti e quindi avverranno maggiori scambi, portando al caso peggiore di questo algoritmo, a differenza invece di `quickSort 3 Ways`. Quest'ultimo, difatti, lavora meglio su valori ripetuti in quanto avverrebbero meno scambi e la sezione centrale che si viene a creare durante l'esecuzione



dell'algoritmo (ovvero quella dei valori uguali al pivot) risulta più ricca e le chiamate ricorsive vengono svolte su sottoarray di dimensioni ridotte. Si nota che `insertionSort` impiega meno tempo quando m risulta piccolo. Questo è dovuto al fatto che, con tanti elementi ripetuti, è più veloce ordinare l'array e quindi il ciclo *while* viene eseguito meno volte, portando a tempi di esecuzione minori.

6.2 Grafici comparativi



L'algoritmo `insertionSort` risulta ancora quello meno efficiente. Come discusso anche nella sezione 6.1, tralasciando i tempi per m piccolo, `quickSort` e `quickSort 3 Ways` presentano un andamento simile al crescere dei valori di in-

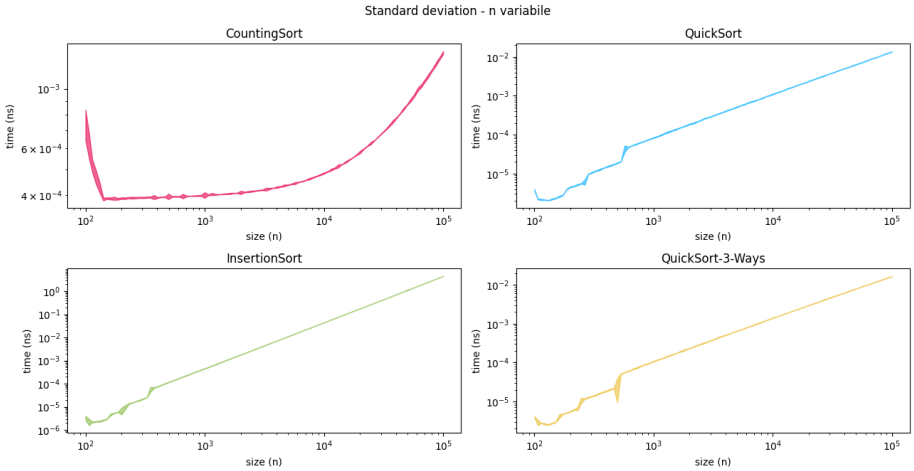
put. Il motivo principale per cui il caso peggiore di `quickSort` si verifica quando nell'array sono presenti tanti valori ripetuti sta nella scelta del pivot. I tempi di esecuzione si allungano in quanto le chiamate ricorsive vengono fatte una su un singolo elemento e l'altra sui restanti. La scelta migliore ricade sul valore medio tra tutti gli elementi presenti, dato che l'array verrebbe partizionato meglio e le chiamate ricorsive avverrebbero su due sezioni della stessa lunghezza, rendendo di fatto l'algoritmo più efficiente. Come anche esposto nella sezione 3.2, esistono degli algoritmi che permettono di selezionare il pivot migliore. Infine `countingSort`, come ci si aspettava, opera bene inizialmente, ma tende a peggiorare notevolmente.

7 Standard deviation

In questa sezione verranno illustrate la standard deviation dei dati per n e m variabile. La deviazione standard è una misura di quanto i dati varino rispetto alla media in un certo data set. Questa misurazione è avvenuta durante lo studio dei tempi, nella funzione `measure`, mediante la formula

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2}$$

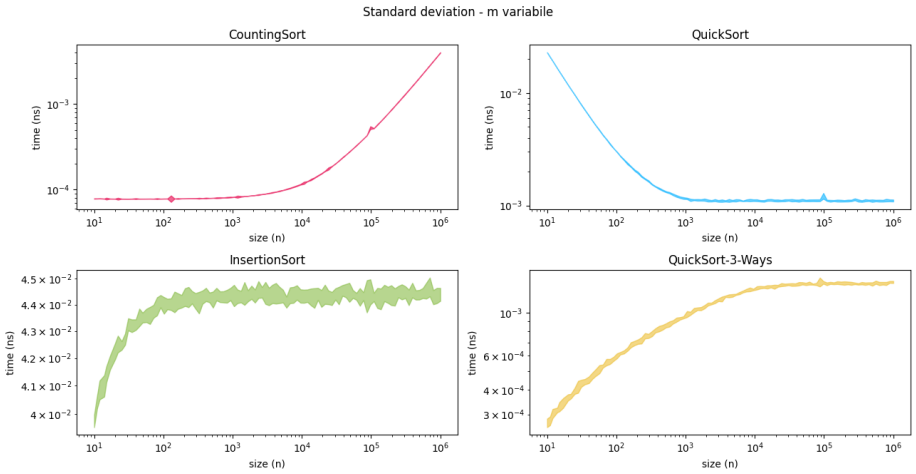
7.1 SD per n variabile



Nel caso di n variabile, in tutti e quattro i casi non è presente una significativa variabilità dei dati in quanto in nessuno dei grafici è presente un grande discostamento dalla media. Il motivo della scarsa variabilità può essere spiegato

dal fatto che il range del valori di input è fissato e questo porta ad avere tempi di esecuzione simili dei singoli algoritmi.

7.2 SD per m variabile



Per m variabile, invece, in `insertionSort` e `quickSort 3 Ways` si può notare la presenza di variabilità. Questo è dovuto da m che, crescendo per i vari campioni, porta a performance variabili dei vari algoritmi. Alcuni di questi algoritmi risultano più sensibili all'input, come mostrato nella sezione 6, portando quindi a deviazioni standard più elevate.

8 Conclusioni

In conclusione è possibile dire che, per n variabile, l'algoritmo che sembra lavorare meglio su array di grandi dimensioni risulta essere `countingSort`, che tende a non crescere come gli altri. D'altra parte, però, tale algoritmo tende a perdere performance con range di valori molto grandi a causa dell'array ausiliario che risulta sensibile alla natura dell'input e la sua dimensione può crescere considerevolmente. `insertionSort`, sia per n che per m variabili, risulta essere quello meno efficiente di tutti, con tempi di esecuzione molto elevati. Per quanto riguarda `quickSort` e `quickSort 3 Ways`, il loro andamento per n variabile risulta pressoché identico. Entrambi sono algoritmi che tendono a operare bene, ma è utile conoscere il contesto in cui vengono applicati. Per valori di m piccoli è più opportuno l'uso di `quickSort 3 Ways` in quanto riesce a gestire meglio array con un elevato numero di duplicati, rendendolo di fatto più efficace. In generale è sempre conveniente conoscere il contesto in cui questi algoritmi vengono applicati per poter sfruttare al meglio le loro potenzialità.