CONFRONTO TRA INSERTION-SORT E COUNTING-SORT

> *Autore: Ivan Necerini*
>
> > *Matricola: 7049380*
>
> *Università degli studi di Firenze*
>
> > *Laboratorio di Algoritmi, Esercizio B*
>
> > *Docente: Simone Marinai*

Indice

Introduzione

Il presente notebook Jupyter si focalizza sul confronto delle prestazioni di due algoritmi di ordinamento: insertion sort e counting sort. L'obiettivo è comprendere le differenze di efficienza tra questi due approcci e valutare le situazioni in cui ciascun algoritmo si dimostra più vantaggioso.

Per condurre i test e ottenere una migliore comprensione dei due algoritmi di ordinamento, sono stati impiegati i seguenti moduli Python:

- `numpy` → utilizzato per generare array con valori casuali utilizzando la funzione `numpy.random.randint()`.
- `matplotlib` → utilizzato per creare grafici e tabelle al fine di analizzare visivamente le prestazioni dei due algoritmi. Le funzioni `matplotlib.pyplot.plot()` e `matplotlib.pyplot.table()` sono state impiegate a tal scopo.

Eseguire il codice sottostante per installarli:

In [1]:
```
!pip install numpy --user --quiet
!pip install matplotlib --user --quiet
```

```
Note: you may need to restart the kernel to use updated packages.
Note: you may need to restart the kernel to use updated packages.
```

Cenni teorici

Algoritmi di ordinamento

L'ordinamento è un problema fondamentale nell'ambito dell'informatica e dell'analisi degli algoritmi. Consiste nel riorganizzare un insieme di elementi in una sequenza ordinata secondo un determinato criterio, che può essere crescente o decrescente. L'obiettivo principale dell'ordinamento è ottenere una rappresentazione ordinata dei dati, facilitando le operazioni di ricerca, confronto e analisi.

Il problema dell'ordinamento riveste un'importanza cruciale in molti contesti applicativi, come ad esempio l'elaborazione di database, la gestione di grandi quantità di dati, la risoluzione di problemi di ottimizzazione e molto altro ancora. Un algoritmo di ordinamento

efficiente può contribuire significativamente a migliorare le prestazioni di altre operazioni e algoritmi che dipendono dalla corretta disposizione degli elementi.

Esistono numerosi algoritmi di ordinamento, ognuno dei quali presenta differenti approcci e strategie per organizzare gli elementi. Questi algoritmi possono differire notevolmente per complessità computazionale, velocità di esecuzione e adattabilità a specifici tipi di dati o situazioni. La scelta dell'algoritmo di ordinamento più appropriato dipende dal contesto e dai requisiti specifici dell'applicazione.

Insertion sort

L'insertion sort è un semplice algoritmo di ordinamento che si basa sul principio dell'inserimento progressivo degli elementi all'interno di una lista. È un algoritmo intuitivo e può essere applicato efficacemente ad array di dimensioni ridotte o a situazioni in cui l'array è già parzialmente ordinato.

Il funzionamento dell'insertion sort è basato su un approccio incrementale. L'algoritmo analizza gli elementi dell'array uno per uno e, ad ogni passo, inserisce l'elemento corrente nella sua giusta posizione all'interno della porzione già ordinata dell'array. Inizialmente, si considera il primo elemento dell'array come una sequenza ordinata di un solo elemento. Successivamente, si prende l'elemento successivo e lo si inserisce nella posizione corretta nella sequenza ordinata finora, spostando gli elementi più grandi di quello corrente verso destra.

Per comprendere meglio il funzionamento dell'insertion sort, si può immaginare di organizzare una mano di carte in ordine crescente. Si inizia con una carta nella mano (rappresentando il primo elemento dell'array), quindi si prende una nuova carta (il secondo elemento dell'array) e la si inserisce nella posizione corretta nella mano, spostando le carte più grandi verso destra. Si ripete questo processo per ogni carta successiva fino a quando tutte le carte sono state inserite nella posizione corretta.

In qualsiasi momento, le carte che teniamo nella mano sinistra sono ordinate; originariamente queste carte erano le prime della pila di carte che erano sul tavolo. Questa frase, a livello di codice (come si vede successivamente), si traduce nel fatto che all'i-esima iterazione, i primi $i-1$ elementi sono già ordinati tra di loro.

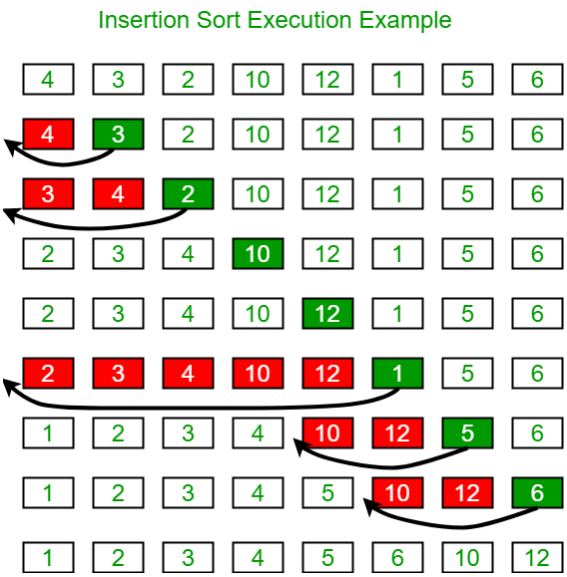


Figura 1: esempio di applicazione di insertion sort

Fonte: Geeksforgeeks

Come si vede dall'implementazione che segue, i numeri di input vengono ordinati sul posto: essi sono risistemati all'interno dell'array A, con al più un numero costante di essi memorizzati all'esterno dell'array in qualsiasi istante.

In [2]:
```
# Implementazione di insertion sort
def insertionSort(A: list):
    for i in range(1, len(A)):
        key = A[i]
        j = i - 1
        # Muovo gli elementi di A[0...i-1], che sono maggiori della chiave,
        #una posizione dopo rispetto a quella dove si trovano
        while j >= 0 and A[j]>key:
            A[j + 1] = A[j]
            j -= 1
        A[j + 1] = key
```

Complessità

Caso peggiore (array ordinato inversamente): l'insertion sort richiede di spostare ogni elemento verso destra di una posizione per ogni elemento precedente. Questo comporta una complessità di tempo quadratica, con una notazione di $O(n^2)$, dove n rappresenta la dimensione dell'array. Quindi, nel caso peggiore, l'insertion sort non è efficiente per grandi quantità di dati.

Caso medio: Nel caso medio, l'insertion sort si comporta meglio rispetto al caso peggiore, ma non raggiunge la complessità del caso migliore. La complessità media dell'insertion sort è ancora $O(n^2)$, ma può essere leggermente migliorata in situazioni in cui l'array è parzialmente ordinato o presenta un certo grado di casualità nella disposizione degli elementi.

Caso migliore (array ordinato): l'insertion sort richiede solo una singola scansione dell'array senza dover effettuare alcuno spostamento. In questo caso, l'algoritmo ha una complessità di $\Omega(n)$, poiché ogni elemento viene confrontato solo con gli elementi precedenti.

Caso peggiore	Caso medio	Caso migliore
$O(n^2)$	$O(n^2)$	$\Omega(n)$

Nonostante sia una soluzione valida per quantità di dati limitate, le caratteristiche dell'insertion sort lo rendono generalmente inefficiente, limitando la sua efficacia su grandi quantità di dati da ordinare.

Correttezza

L'insertion sort è corretto nel senso che garantisce che l'array di input sarà ordinato correttamente dopo l'esecuzione dell'algoritmo. L'idea di base dell'insertion sort è quella di mantenere una sottosequenza di elementi ordinati mentre si attraversa l'array, inserendo ciascun elemento nella posizione corretta all'interno della sottosequenza. Questo processo di inserimento ripetuto garantisce che l'intero array sarà alla fine ordinato correttamente. La sua correttezza può essere dimostrata tramite una invariante di ciclo, dato che l'implentazione mostrata prima è in versione iterativa.

Stabilità

L'insertion sort è un algoritmo di ordinamento stabile, il che significa che preserva l'ordine relativo degli elementi con chiavi uguali. Ciò significa che se due elementi hanno la stessa chiave e sono in posizioni diverse nell'array di input, allora dopo l'applicazione dell'insertion sort, questi due elementi manterranno lo stesso ordine relativo nella sequenza ordinata.

Ad esempio, se si sta ordinando un elenco di studenti per punteggio e due studenti hanno lo stesso punteggio, l'cnsertion sort garantirà che i due studenti mantengano il loro ordine originale nella sequenza ordinata, preservando così la giustizia nella valutazione.

Ecco delle animazioni per comprendere al meglio il funzionamento dell'insertion sort:

Figura 2: esempio animazione insertion sort

Fonte: Wikipedia

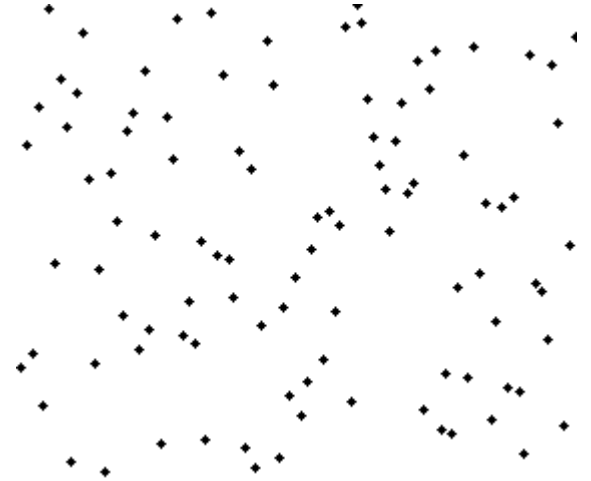


Figura 3: esempio grafico insertion sort

Fonte: Wikipedia

Counting sort

Il counting sort è un algoritmo di ordinamento non comparativo. A differenza di altri algoritmi di ordinamento, il counting sort sfrutta le informazioni sul range dei valori dell'input per ottenere una complessità lineare.

Il funzionamento del counting sort può essere suddiviso in tre fasi:

1. Conta le occorrenze: In questa fase, viene creato un array di conteggio con dimensioni pari all'intervallo massimo dei valori presenti nell'input. Si scandisce l'input e si incrementa il contatore corrispondente al valore dell'elemento. Alla fine di questa fase, l'array di conteggio conterrà il numero di occorrenze per ciascun valore presente nell'input.

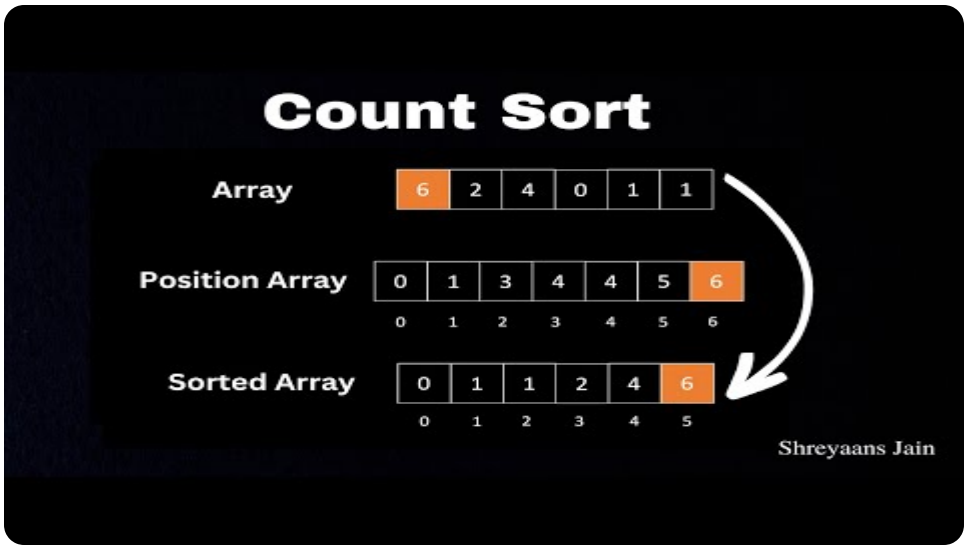
2. Calcola le posizioni: Utilizzando l'array di conteggio, viene calcolato un array ausiliario di posizioni. Questo array indica la posizione in cui ogni valore deve essere posizionato nell'output finale. Viene calcolato sommando i valori precedenti nell'array di conteggio. In altre parole, l'elemento i-esimo nell'array ausiliario di posizioni indica la posizione finale dell'ultimo elemento con valore i nell'output.

3. Costruisci l'output: Durante questa fase, si scorre nuovamente l'input. Per ogni elemento, si determina la sua posizione finale utilizzando l'array ausiliario di posizioni e si inserisce l'elemento in quella posizione nell'output. Successivamente, si decrementa il valore corrispondente nell'array di conteggio per tener traccia delle occorrenze rimanenti.

Tuttavia, il counting sort presenta alcune limitazioni. Richiede un intervallo limitato dei valori di input e può richiedere una quantità significativa di memoria, in particolare quando

l'intervallo è ampio. Pertanto, è efficace in situazioni specifiche in cui queste limitazioni non sono un problema, come l'ordinamento di numeri interi non negativi in un range ristretto.

E' possibile capire a pieno il funzionamento di questo algoritmo con un'animazione (cliccare per aprire il video):



Segue una delle sue possibili implementazioni (supponendo che operi su un insieme di interi non negativi):

In [3]:
```python
# Implementazione del counting sort
def countingSort(A: list):
    # Trova il valore massimo nell'array di input per determinare la dimensione del
    maxValue = max(A)

    # Crea un array di conteggio con dimensioni pari al valore massimo + 1 (ogni va
    counts = [0] * (maxValue + 1)

    # Conta il numero di occorrenze di ciascun elemento nell'array
    for num in A:
        counts[num] += 1

    # Conta il numero di elementi minori o uguali a i
    # Questo array indica la posizione in cui ogni valore deve essere posizionato n
    for i in range(1, maxValue + 1):
        counts[i] += counts[i - 1]

    # Crea l'array ordinato B
    B = [0] * len(A)
    for num in A:
        B[counts[num] - 1] = num
        counts[num] -= 1
    return B
```

Complessità:

La complessità del counting sort dipende sia dalla dimensione dell'array da ordinare (n) che dalla quantità di valori che è possibile inserire nell'array da ordinare (k) (valore massimo, dunque valori da 1 a k).

Il costo dell'algoritmo è $O(n + k)$.

La complessità nel caso peggiore è lineare rispetto alla somma di n e k. Ciò significa che il tempo e lo spazio richiesti per eseguire l'algoritmo crescono in modo lineare all'aumentare di entrambi i valori. La complessità nel caso peggiore si verifica dunque quando i valori nell'array sono distribuiti in modo uniforme su un ampio intervallo.

È importante notare che se k è relativamente piccolo rispetto a n, l'algoritmo può essere molto efficiente. Tuttavia, se k è comparabile o addirittura maggiore di n, l'algoritmo diventa meno efficiente.

Correttezza:

Il counting sort è un algoritmo corretto e garantisce che l'array di input sarà ordinato correttamente. Poiché il counting sort utilizza informazioni sull'ordine dei valori e non confronta direttamente gli elementi, l'ordinamento è garantito essere corretto. La sua correttezza può essere dimostrata tramite una invariante di ciclo, dato che l'implentazione mostrata prima è in versione iterativa.

Stabilità:

Il counting sort è un algoritmo di ordinamento stabile. Quando ci sono elementi con la stessa chiave, il counting sort assegna loro posizioni nell'array ordinato in base all'ordine in cui compaiono nell'array di input. Ciò significa che due elementi con la stessa chiave manterranno il loro ordine relativo nella sequenza ordinata, garantendo così la stabilità.

La stabilità di counting sort è importante. Una delle motivazioni è che counting sort viene spesso utilizzato come subroutine di radix sort.

Descrizione dei test

Per valutare le prestazioni degli algoritmi di ordinamento, è necessario generare una lista di valori interi di dimensione n e misurare il tempo di esecuzione di ambedue gli algoritmi.

Per ottenere una valutazione accurata dell'Insertion Sort, verranno creati dati per il caso migliore (dati già ordinati) e il caso peggiore (dati ordinati inversamente). Invece, per il counting sort, ciò non è necessario, in quanto questi particolari dati non influiscono sulle sue prestazioni, poiché la sua complessità rimane costante in tutti i casi. I test di ordinamento verranno eseguiti su un insieme di n valori, con n che varia da 1 a 5000 (dunque ci vorrà qualche secondo per la generazione dei grafici e delle tabelle).

Per ciascun test, verranno eseguite più prove, generando input diversi, e verrà calcolato il valore medio di ogni gruppo di prove. In questo modo, saranno ottenute informazioni significative sulle prestazioni e sul comportamento degli algoritmi di ordinamento confrontati.

Generazione dei dati per l'esecuzione dei test

Di seguito sono presentate alcune funzioni per generare array di valori:

1. La funzione `randomArray(n)` accetta un intero n come parametro e genera un array di n valori casuali compresi tra 0 e maxValue.
2. Le funzioni `sortedArray(n)` e `reversedArray(n)` generano invece array di n elementi, ordinati in modo crescente e decrescente rispettivamente.

In [6]:
```python
import numpy as np

# Valore massimo che può essere inserito nell'array
maxValue = 200

# Array di numeri random
def randomArray(n):
    # converte l'array numpy in una lista utilizzando il metodo .tolist()
    return np.random.randint(0, maxValue + 1, n).tolist()

# Array ordinato (crescente)
def sortedArray(n):
    # da 0 a n-1
    return list(range(n))

# Array ordinato inversamente (decrescente)
def reversedArray(n):
    # da n-1 a 0
    return list(range(n-1, -1, -1))
```

Esecuzione dei test

Il parametro "n" rappresenta il numero totale di iterazioni nell'esecuzione dello script sottostante. Ad ogni iterazione, viene generato un array composto da "i" elementi, che successivamente verrà ordinato utilizzando due algoritmi che saranno confrontati.

La misurazione del tempo di esecuzione dei due algoritmi `insertionSort(A: list)` e `countingSort(A: list)` avviene tramite l'utilizzo del modulo "timeit" e della sua funzione "timeit.timeit()". Per ottenere una misura più precisa, l'ordinamento viene ripetuto un numero di volte pari a "nrTestEachIteration" e successivamente viene calcolata la media dei tempi ottenuti.

È importante sottolineare che viene eseguito prima l'algoritmo counting sort sull'array, seguito dall'insertion sort. Questo perché il counting sort restituisce un nuovo array con gli elementi ordinati, mantenendo invariato l'array originale, mentre l'insertion sort effettua l'ordinamento direttamente sull'array di partenza (sul posto). Invertendo l'ordine delle chiamate, l'insertion sort ordinerà l'array generato, di conseguenza il counting sort riceverà sempre un array già ordinato.

In [7]:
```
import timeit

n = 5000
step = 250
nrTestEachIteration = 250

insertionSortWorst = []
countingSortWorst = []

insertionSortAverage = []
countingSortAverage = []

insertionSortBest = []
countingSortBest = []


def measureTime(function, args):
    # Restituisce la media del tempo di esecuzione (in millisecondi) della funzione
     return timeit.timeit(stmt=lambda: function(args), number=nrTestEachIteration)

# Caso peggiore, elementi ordinati inversamente
for i in range(1, n, step):
    arrayTest = reversedArray(i)
    # Test sul counting sort
    countingSortWorst.append(measureTime(countingSort, arrayTest))
    # Test sull'insertion sort
    insertionSortWorst.append(measureTime(insertionSort, arrayTest))

# Caso medio, elementi casuali
for i in range(1, n, step):
    arrayTest = randomArray(i)
    # Test sul counting sort
    countingSortAverage.append(measureTime(countingSort, arrayTest))
    # Test sull'insertion sort
    insertionSortAverage.append(measureTime(insertionSort, arrayTest))


# Caso migliore, elementi gia' ordinati
for i in range(1, n, step):
    arrayTest = sortedArray(i)
    # Test sul counting sort
    countingSortBest.append(measureTime(countingSort, arrayTest))
    # Test sull'insertion sort
    insertionSortBest.append(measureTime(insertionSort, arrayTest))

print("END, DEBUG")
```

END, DEBUG

Generazione delle tabelle dei tempi di esecuzione

Di seguito sono riportate le tabelle che mostrano i tempi di esecuzione dei due algoritmi nei tre casi considerati, al crescere della quantità dei valori da ordinare:

1. Le tabelle sono state create utilizzando la libreria `numpy` e il metodo `numpy.pyplot.table`

In [8]:
```
import matplotlib.pyplot as plt

# Creazione delle liste di dati
# Ogni lista contiene tre elementi: una lista di valori da 1 a n con un passo speci
# formattati in notazione scientifica (con 3 cifre decimali) per l'algoritmo "inser
# valori formattati in notazione scientifica per l'algoritmo "counting sort"
worstData = [
    [i for i in range(1, n, step)],
    ["{:.3e}".format(val) for val in insertionSortWorst],
    ["{:.3e}".format(val) for val in countingSortWorst]
]

averageData = [
    [i for i in range(1, n, step)],
    ["{:.3e}".format(val) for val in insertionSortAverage],
    ["{:.3e}".format(val) for val in countingSortAverage]
]

bestData = [
    [i for i in range(1, n, step)],
    ["{:.3e}".format(val) for val in insertionSortBest],
    ["{:.3e}".format(val) for val in countingSortBest]
]

# Prende in input una lista di colonne di dati, una tupla di intestazioni e un tito
def traceTables(columns: list, headers: tuple, title: str):
    fig, ax = plt.subplots(figsize=(8, 10))
    plt.title(title)

    # Unisci le liste di dati come colonne al fine di creare un array bidimensional
    data = np.stack(tuple(columns), axis=1)

    # Stile tabella
    ax.axis('off')
    table = ax.table(cellText=data, colLabels=headers, loc='center', cellLoc='cente
    table.auto_set_column_width(col=list(range(len(columns))))
    table.scale(1, 1.5)

    # Colorazione headers e le righe pari
    cell_colors = {
        cell: ("#ffd1d1", {"weight": "bold"})
```

```
        if table[cell].get_text().get_text() in headers
        else ("#ffe4e4", {})
        for cell in table._cells
        if cell[0] % 2 == 0
    }
    for cell, (color, text_props) in cell_colors.items():
        # set the color of the cell
        table[cell].set_facecolor(color)
        # set the text properties of the cell
        table[cell].set_text_props(**text_props)
        # ** used to expand the dictionary
    plt.show()

traceTables(
    worstData,
    ("Nr elementi", "Insertion sort", "Counting sort"),
    "Caso peggiore: tempi di esecuzione con lista ordinata inversamente [ms] (Figur
)

traceTables(
    averageData,
    ("Nr elementi", "Insertion sort", "Counting sort"),
    "Caso medio: tempi di esecuzione con valori random [ms] (Figura 5)"
)

traceTables(
    bestData,
    ("Nr elementi", "Insertion sort", "Counting sort"),
    "Caso migliore: tempi di esecuzione con lista ordinata [ms] (Figura 6)"
)
```

Caso peggiore: tempi di esecuzione con lista ordinata inversamente [ms] (Figura 4)

Nr elementi	Insertion sort	Counting sort
1	3.072e-04	8.852e-04
251	2.690e-02	4.544e-02
501	1.640e-01	1.192e-01
751	1.296e-01	1.829e-01
1001	3.327e-01	1.922e-01
1251	3.846e-01	3.128e-01
1501	6.385e-01	3.568e-01
1751	6.800e-01	3.993e-01
2001	8.963e-01	6.041e-01
2251	1.000e+00	7.064e-01
2501	1.258e+00	5.151e-01
2751	1.473e+00	6.904e-01
3001	1.642e+00	6.003e-01
3251	2.195e+00	7.713e-01
3501	2.488e+00	9.166e-01
3751	2.891e+00	8.590e-01
4001	3.413e+00	8.208e-01
4251	3.488e+00	1.020e+00
4501	4.011e+00	1.076e+00
4751	3.772e+00	1.293e+00

Caso medio: tempi di esecuzione con valori random [ms] (Figura 5)

Nr elementi	Insertion sort	Counting sort
1	1.944e-04	8.936e-03
251	3.736e-02	4.018e-02
501	8.733e-02	1.631e-01
751	1.995e-01	1.092e-01
1001	2.284e-01	1.572e-01
1251	3.336e-01	2.167e-01
1501	2.858e-01	2.035e-01
1751	3.700e-01	2.357e-01
2001	4.591e-01	2.653e-01
2251	5.645e-01	3.016e-01
2501	9.043e-01	3.433e-01
2751	8.726e-01	3.621e-01
3001	1.268e+00	3.975e-01
3251	1.009e+00	4.360e-01
3501	1.471e+00	4.867e-01
3751	1.777e+00	5.226e-01
4001	2.094e+00	7.450e-01
4251	2.207e+00	6.394e-01
4501	2.545e+00	8.049e-01
4751	2.560e+00	6.583e-01

Caso migliore: tempi di esecuzione con lista ordinata [ms] (Figura 6)

Nr elementi	Insertion sort	Counting sort
1	1.912e-04	6.940e-04
251	4.354e-02	5.032e-02
501	4.174e-02	1.781e-01
751	6.127e-02	1.427e-01
1001	8.472e-02	1.926e-01
1251	1.067e-01	3.661e-01
1501	2.319e-01	4.020e-01
1751	2.153e-01	4.736e-01
2001	1.726e-01	5.559e-01
2251	2.116e-01	6.570e-01
2501	3.395e-01	5.166e-01
2751	3.530e-01	6.249e-01
3001	2.599e-01	7.140e-01
3251	2.881e-01	7.204e-01
3501	3.391e-01	7.175e-01
3751	4.375e-01	1.229e+00
4001	3.510e-01	1.030e+00
4251	3.737e-01	1.139e+00
4501	4.155e-01	9.436e-01
4751	5.377e-01	1.087e+00

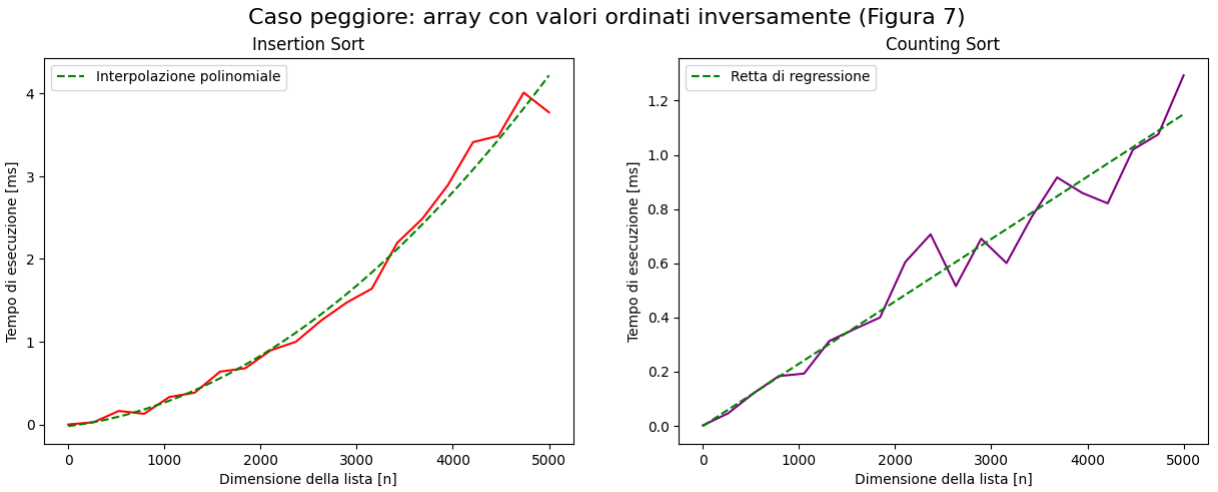
Generazione dei grafici

La generazione dei grafici è stata realizzata utilizzando la libreria matplotlib. I tre set di grafici presentati confrontano i tempi di esecuzione dei due algoritmi in base alla lunghezza dell'array.

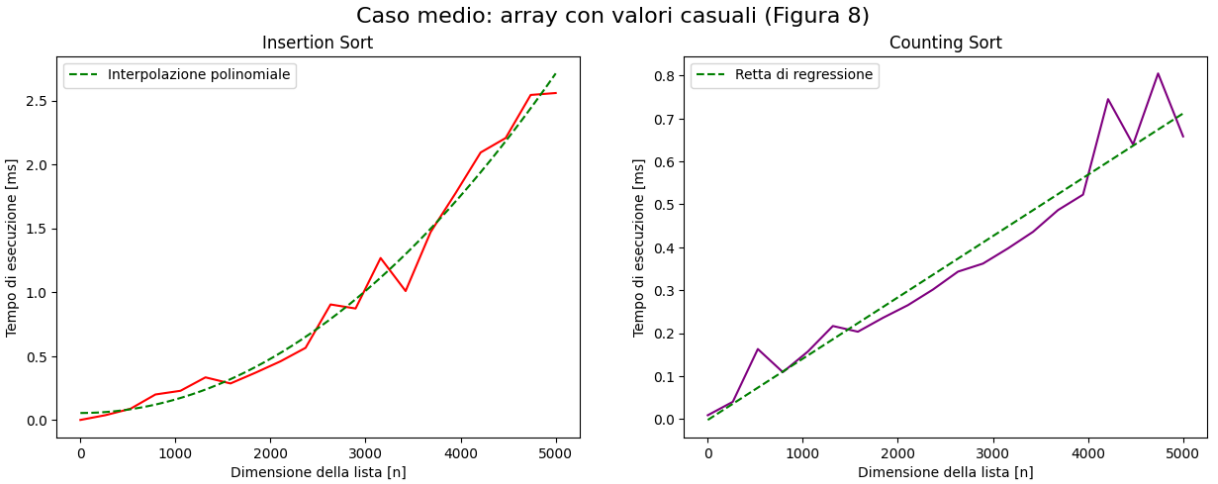
I valori rappresentati nei grafici si riferiscono, rispettivamente, al caso peggiore (valori ordinati inversamente), al caso medio (valori casuali) e al caso migliore (valori ordinati in modo corretto).

In [9]:
```
def approximate(plot, x, y, degree, label):
    # Calcola i coefficienti dell'approssimazione polinomiale
    coefficients = np.polyfit(x, y, degree)
    # Crea una nuova serie di punti x equidistanti
    newX = np.linspace(1, n, n)
    # Calcola i valori approssimati corrispondenti ai nuovi punti x
    newY = np.polyval(coefficients, newX)
    # Disegna la curva approssimata
    plot.plot(newX, newY, '--', label=label, color='green')

def tracePlots(left_data, right_data, plot_title: str = None):
    x = np.linspace(1, n, len(left_data))
    fig, (left, right) = plt.subplots(1, 2, figsize=(15, 5))

    # Insertion sort, grafico a sinistra
    left.plot(x, left_data, color='red')
    left.set_title('Insertion Sort')
    left.set_xlabel('Dimensione della lista [n]')
    left.set_ylabel('Tempo di esecuzione [ms]')

    # Aggiungi l'approssimazione polinomiale al grafico Insertion Sort
    approximate(left, x, left_data, 2, label='Interpolazione polinomiale') # Aggiun
    left.legend()


    # Counting sort, grafico a destra
    right.plot(x, right_data, color='purple')
    right.set_title('Counting Sort')
    right.set_xlabel('Dimensione della lista [n]')
    right.set_ylabel('Tempo di esecuzione [ms]')

    # Aggiungi l'approssimazione polinomiale al grafico Counting Sort
    approximate(right, x, right_data, 1, label='Retta di regressione') # Aggiungo l
    right.legend()

    if plot_title:
        fig.suptitle(plot_title, fontsize=16)
```

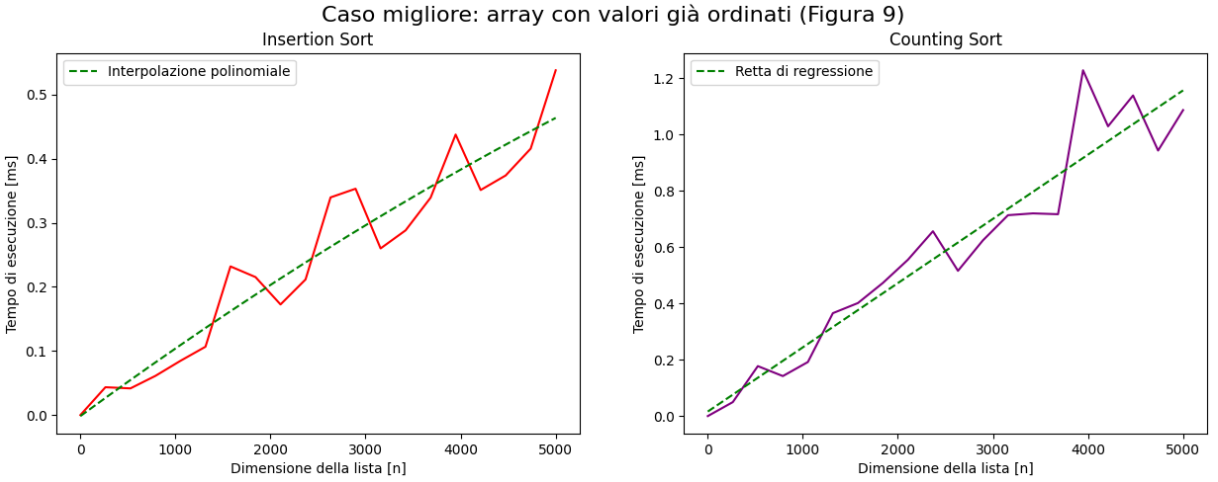
In [10]:
```
tracePlots(insertionSortWorst, countingSortWorst, "Caso peggiore: array con valori
```



In [11]:
```
tracePlots(insertionSortAverage, countingSortAverage, "Caso medio: array con valori
```



In [12]:
```
tracePlots(insertionSortBest, countingSortBest, "Caso migliore: array con valori gi
```



Osservazioni finali

Le prestazioni dei due algoritmi di ordinamento, insertion sort e counting sort, sono state valutate mediante la generazione di grafici utilizzando il modulo matplotlib. I grafici evidenziano chiaramente i vantaggi e gli svantaggi di ciascun algoritmo e il loro comportamento al variare della dimensione dei dati da ordinare e della loro distribuzione.

> NOTA: fare riferimento alle osservazioni già fatte precedentemente riguardo l'insertion sort (Insertion sort, Complessità) e il counting sort (Counting sort, Complessità:).

Caso peggiore: valori ordinati inversamente

Nel caso peggiore dell'insertion sort, quando i dati sono ordinati inversamente, si osserva una complessità quadratica, come confermato dal grafico a sinistra in Figura 7. Il grafico confronta i tempi di esecuzione dell'insertion sort (in rosso) con un'interpolazione polinomiale di secondo grado (parabola verde). Inoltre, i tempi di esecuzione in confronto al caso medio sono peggiori, a parità della quantità dei valori da ordinare, come evidenziato dal massimo nell'asse y nei grafici delle figure 7 e 8, nonché dalle tabelle riportate nelle Figure 4 e 5. Il counting sort, invece, mantiene una complessità lineare, con tempi di esecuzione simili al caso migliore.

Caso medio: valori casuali

Nel caso di ordinamento di una lista composta da valori casuali, il grafico delle prestazioni dell'insertion sort mostra un andamento simile a n^2, come previsto dalla complessità dell'algoritmo (parabola verde rappresentata in verde in figura 8). Al contrario, il counting sort mantiene un comportamento lineare. Questa differenza di complessità si riflette anche nei tempi di esecuzione, come indicato nella tabella riportata in precedenza (figura 5). All'aumentare dei dati, le prestazioni del counting sort sono notevolmente superiori a quelle dell'insertion sort. Tuttavia, è importante notare che quando la quantità di dati da ordinare è limitata, l'insertion sort può avere prestazioni migliori rispetto al counting sort, come già evidenziato in precedenza.

Caso migliore: valori già ordinati correttamente

Nel caso migliore dell'insertion sort, quando i dati sono già ordinati correttamente, si osserva una complessità lineare, come evidenziato dal grafico a sinistra in figura 9. Questa conferma le ipotesi teoriche fatte all'inizio, in cui è stata affermata una complessità nel caso migliore di $\Omega(n)$. L'interpolazione polinomiale di secondo grado nel grafico (rappresentata in verde) mostra chiaramente che l'andamento della curva è lineare, e non quadratico. Il counting sort mantiene una complessità lineare, come previsto.

Inoltre, osservando la tabella in figura 6, si nota che i tempi di esecuzione dell'insertion sort sono inferiori a quelli del counting sort, nonostante entrambi abbiano un comportamento

lineare. Questo è dovuto al fatto che l'insertion sort è un algoritmo più semplice del counting sort, in quanto ordina gli elementi direttamente sulla lista senza l'uso di strutture esterne. L'insertion sort richiede solo un ciclo esterno, mentre il counting sort richiede cicli esterni multipli e quindi esegue più operazioni con costo lineare.

Osservazioni su insertion sort

Le osservazioni sull'insertion sort confermano la corrispondenza tra la complessità teorica e le misurazioni effettuate nella pratica. Nel caso migliore, la complessità risulta effettivamente lineare, mentre nel caso medio e nel caso peggiore diventa quadratica.

Osservazioni sul counting sort

Come previsto, il counting sort mantiene una complessità lineare indipendentemente dai dati in ingresso. È interessante notare che quando l'array è già correttamente ordinato in base alla scelta (crescente o decrescente), le prestazioni del counting sort sono inferiori rispetto a quando i valori sono casuali. Questa osservazione è supportata sia dai tempi di esecuzione riportati in tabella (Figure 4, 5, 6), sia dall'asse verticale dei grafici (Figure 7, 8, 9). Ciò è dovuto al fatto che quando i dati sono ordinati, i valori nella lista vanno da 1 a n (o viceversa) senza ripetizioni, mentre nel caso medio i valori variano da 1 a `maxValue`, quindi quando la lunghezza della lista supera `maxValue`, sono presenti valori duplicati. Questa caratteristica migliora le prestazioni del counting sort, poiché la sua complessità ($O(n + k)$) dipende anche dal range dei dati presenti nella lista (k), e non solo dalla dimensione della lista stessa (n).

Altre osservazioni generali

Per concludere, ecco tre osservazioni generali di confronto tra i due algoritmi, non di natura tempistica:

- Mentre l'insertion sort richiede solo spazio per memorizzare l'array da ordinare, il counting sort richiede spazio aggiuntivo per la creazione di un array ausiliario per conteggiare le occorrenze dei valori.
- Il counting sort è particolarmente efficace quando il range dei valori è relativamente piccolo rispetto alla dimensione della lista. Al contrario, l'insertion sort non è influenzato dal range dei valori.
- L'insertion sort ha una complessità spaziale di $O(1)$ poiché opera sull'array di input senza richiedere spazio aggiuntivo. Il counting sort, invece, ha una complessità spaziale di $O(k)$, dove k rappresenta il range dei valori.

Bibliografia

> Figura 1: Geeksforgeeks

> Figura2: Wikipedia

> Figura3: Wikipedia