Università degli studi di Udine

Progetto di Algoritmi e Strutture Dati

Analisi Algoritmi di Sorting

Luglio 2025

Componenti del gruppo

Ken Andrew Del Mundo, ..., ...@spes.uniud.it
... ., ..., ...@spes.uniud.it
...., ..., ...@spes.uniud.it

Introduzione

Nel seguente progetto verranno analizzati e confrontati quattro algoritmi di ordinamento: CountingSort, QuickSort, 3-way QuickSort e CycleSort.
Ciascun algoritmo verrà descritto in termini di funzionamento, implementazione, complessità computazionale e caratteristiche (in-place, stabilità, efficienza), evidenziandone i pro e i contro.
Per valutare le prestazioni di ogni singolo algoritmo, saranno presenti grafici rappresentati del tempo di esecuzione rispetto alla lunghezza di un Vettore dato in input.

Grafici

La lunghezza dei vettori varia da 100 a 100000, mentre il range dei valori utilizzati va da 0 a 100000.
Per i grafici che analizzano il comportamento in base alla lunghezza dell'input si è usato un range di 100000.
Per i grafici che analizzano il comportamento in base al range di valori contenuti nel vettore dell'input si è usato una lunghezza 10000.

Note Bene

La misurazione dei tempi in base alla lunghezza potrebbe richiedere molto tempo per la complessità di CycleSort.

CountingSort

Descrizione

CountingSort è un algoritmo non basato su confronti che opera contando il numero di occorrenze di ogni valore presente nel vettore.
È un algoritmo che necessità la conoscenza del range k (attraverso un min_val e max_val) di valori contenuti nel vettore, per poter definire un vettore (di lunghezza k) di occorrenze che conta il numero di occorrenze di un determinato valore $i + min\_val$ (i indice del vettore).
Per garantire la stabilità, si trasforma il vettore *count* in un array cumulativo, in modo da determinare la posizione finale di ciascun elemento nel vettore ordinato. Si itera l'array originale da destra verso sinistra, per mantenere l'ordine relativo degli elementi con lo stesso valore. Siccome richiede un vettore di occorrenze, l'algoritmo non è in-place.
Nota Bene: l'implementazione è diversa da quella vista a lezione. La nuova implementazione è adattata per gestire anche i numeri negativi.

Implementazione

```
def CountingSort(A):
    min_val = min(A)
    max_val = max(A)
    offset = -min_val

    count = [0] * (max_val - min_val + 1)
    copy = [0] * len(A)

    #conta le occorrenze
    i = 0
    for num in A:
        count[num + offset] += 1
        copy[i] = num
        i += 1

    # Calcolo indirizzo piu' a destra di
    # un elemento in posizione i in count
    for i in range(1, len(count)):
        count[i] += count[i - 1]

    # posiziona gli elementi mantenendo l'ordine
    for i in range(len(A) - 1, -1, -1):
        num = copy[i]
        count[num + offset] -= 1
        A[count[num + offset]] = num

    return A
```

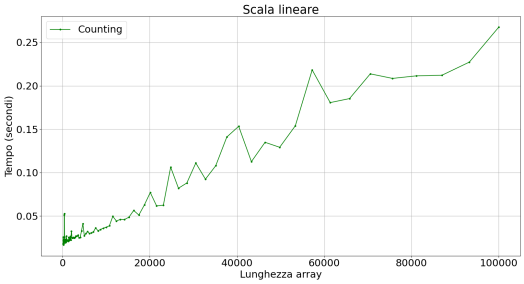
Complessità

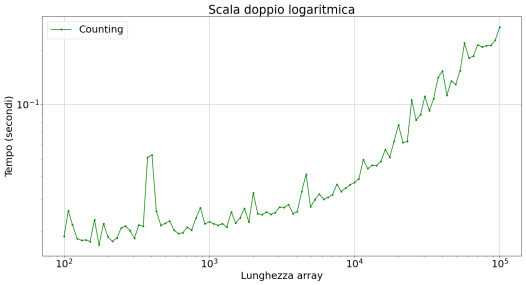
Si scorre il vettore per trovare il minimo, il massimo, per contare le occorrenze e per trovare la posizione finale di ogni elemento.
Si scorre poi l'array delle occorrenze per trovare la posizione finale di ogni elemento i.
Se k è il range di valori con si sta lavorando e n è la lunghezza del vettore, la Complessità è:

$$O(k + n)$$

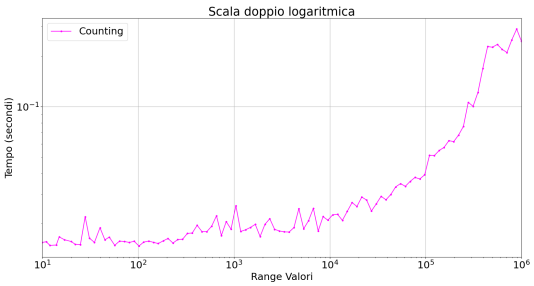
l'algoritmo è lineare, ma perde di efficienza per k molto più grande di n.

BenchMark





Grafici CountingSort a variare della lunghezza del vettore in input



Grafici CountingSort a variare del range dei valori del vettore in input

QuickSort

Descrizione

Quicksort è un algoritmo di sorting basato su confronti e sulla strategia divide et impera.
Si parte da un vettore A da ordinare, ad ogni iterazione di *QuickSortApp* se si sta lavorando con una sezione (determinata da *low* e *high*) ha più di un elemento, si sceglie un *pivot* (in questo caso si è scelto il valore indicato dall'indice *high*) e lo si posiziona nella sua posizione finale, posizionando i valori minori/uguali del pivot alla sua sinistra e i valori maggiori al pivot alla sua destra.
Si richiama successivamente la procedura sulla porzione a sinistra e la porzione a destra del pivot.
Anche se si lavora con chiamate ricorsive, esse possono essere sostituite con uno stack minimale, rendendolo in-place.

Implementazione

```
def QuickSort(A):
    return QuickSortApp(A, 0, len(A) - 1)

def QuickSortApp(A, low, high):

    if low < high:
        pi = Partition(A, low, high)
        QuickSortApp(A, low, pi - 1)
        QuickSortApp(A, pi + 1, high)

    return A
```

```
def Partition(A, low, high):

    pivot = A[high]
    pivot_pos = low - 1

    for j in range(low, high):
        if A[j] <= pivot:
            pivot_pos += 1
            A[pivot_pos], A[j] = A[j], A[pivot_pos]

    #posiziona il pivot in posizione finale
    A[pivot_pos + 1], A[high] = A[high], A[pivot_pos + 1]

    return pivot_pos + 1
```

Complessità

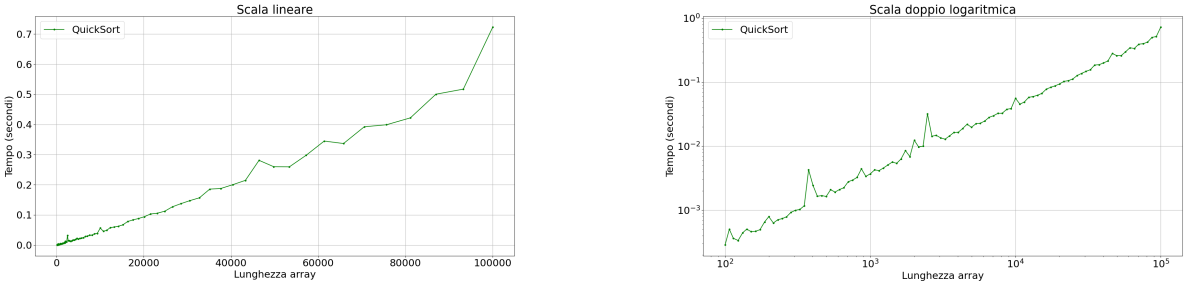
La complessità è altamente legata alla scelta del pivot, poichè determina il modo in cui vado a dividere il problema.
Nel caso medio la complessità è:

$$O(n \log n)$$

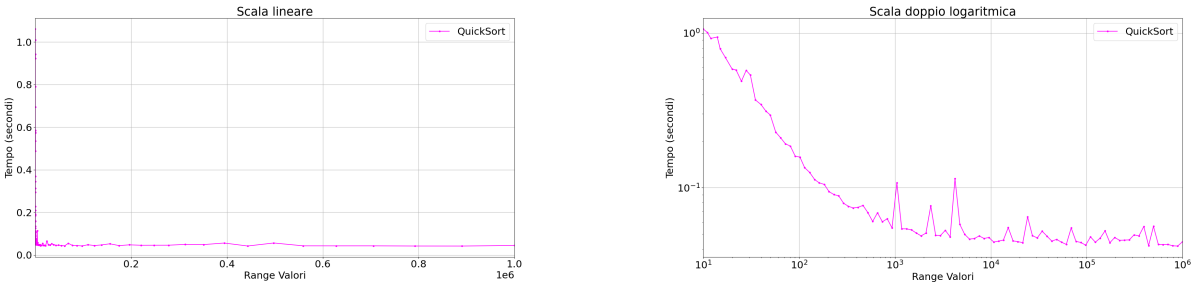
(come visto a lezione).
Il Worst Case Scenario si manifesta quando, scegliendo il pivot, finisco per dividere il problema in 2 array dove uno dei due ha lunghezza 0/1. Ad ogni scelta di pivot vado ad effettuare una singola chiamata ricorsiva, portando la complessità a :

$$O(n^2)$$

BenchMark



Grafici QuickSort a variare della lunghezza del vettore in input



Grafici QuickSort a variare del range dei valori del vettore in input

3-way QuickSort

Descrizione

3-way Quicksort è un algoritmo di sorting basato sul Quicksort.
È una variante del Quicksort che divide il vettore in 3 parti (attraverso *threeWayPartition*):
- gli elementi minori del *pivot*
- gli elementi uguali del *pivot*
- gli elementi maggiori del *pivot*

Isolare la porzione di elementi uguali al pivot permette di diminuire la dimensione dei sottovettori delle due chiamate ricorsive, rendendolo più efficiente del QuickSort Normale.

Implementazione

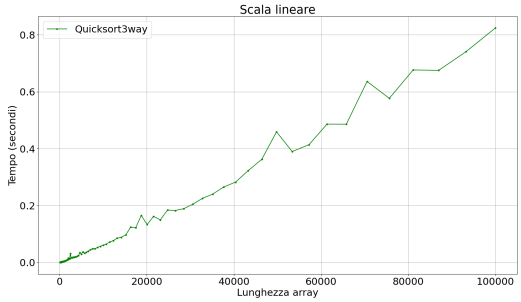
```
def ThreeWayQuickSort(A):
    return threeWaySort(A, 0, len(A) - 1)

def threeWaySort(A, low, high):

    if low < high:
        lt, gt = threeWayPartition(A, low, high)
        threeWaySort(A, low, lt - 1)
        threeWaySort(A, gt + 1, high)

    return A
```

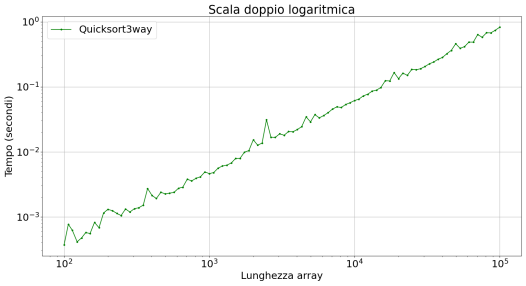
```
def threeWayPartition(A, low, high):

    pivot = A[low]
    lt = low       # A[low..lt-1] < pivot
    gt = high      # A[gt+1..high] > pivot
    i = low        # A[lt..i-1] == pivot

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

    return lt, gt
```

Complessità

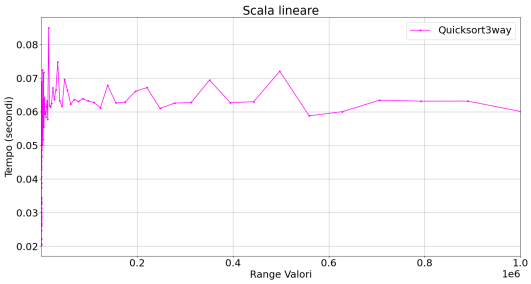
La complessità rimane invariata rispetto a Quicksort.
La differenza si vede in presenza di elementi duplicati, che rende il 3-way QuickSort più efficiente.

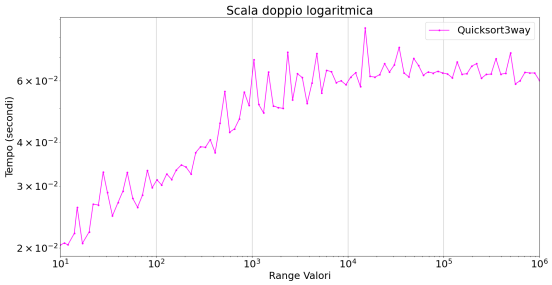
BenchMark





Grafici 3-way QuickSort a variare della lunghezza del vettore in input





Grafici 3-way QuickSort a variare del range dei valori del vettore in input

CycleSort

Descrizione

Cyclesort è un algoritmo di sorting ottimale in termini di scrittura in memoria, utile nei sistemi con scrittura in memoria costosa (es. memorie flash o EEPROM).
Minimizza il numero di scritture poichè evita scambi inutili: ogni elemento viene spostato al massimo una volta nel suo posto corretto.
Ad ogni iterazione, trovo l'elemento che dovrebbe stare nella posizione indicata da *start*.
start punta ad un elemento iniziale *item*. Trovo la posizione finale *pos* di *item*: ponendo inizialmente $pos = start$, incremento *pos* per il numero di numeri minori ad *item* presenti alla destra di *item*. Se rimane che $pos = start$, allora *start* è la posizione finale dell'elemento *item*, passo al controllo dell'elemento puntato da $start + 1$.
Altrimenti, scambio l'elemento *item* con quello puntato da *pos*(che mi diventa il nuovo *item*). Attraverso dei cicli, finchè non trovo l'elemento che deve stare nella posizione *start*, posiziono i vari *item* che ricavo nelle loro posizioni finali.
Una volta trovato, passo all'elemento $start + 1$.
E' un algoritmo in-place poichè non va ad utilizzare altre strutture dati, ma non è stabile.

Implementazione

```python
def cycle_sort(A):
    n = len(A)

    # Scorri ogni elemento tranne l'ultimo
    for start in range(n - 1):
        # Elemento corrente da posizionare
        item = A[start]
        # Posizione dove dovrebbe andare 'item'
        pos = start

        # Conta quanti elementi a destra sono minori di 'item'
        for i in range(start + 1, n):
            if A[i] < item:
                pos += 1

        # Se è già nella posizione giusta, salta
        if pos == start:
            continue

        # Se ci sono duplicati, salta le posizioni occupate
        while item == A[pos]:
            pos += 1

        # Metto elemento in posizione corretta
        A[pos], item = item, A[pos]

        # Continua a ciclare gli elementi
        # finché non si chiude il ciclo
        while pos != start:
            pos = start
            # Ricalcola la posizione corretta
            #dell'item corrente
            for i in range(start + 1, n):
                if A[i] < item:
                    pos += 1

            # Salta eventuali duplicati
            while item == A[pos]:
                pos += 1

            # Scambia nuovamente
            A[pos], item = item, A[pos]

    return A
```

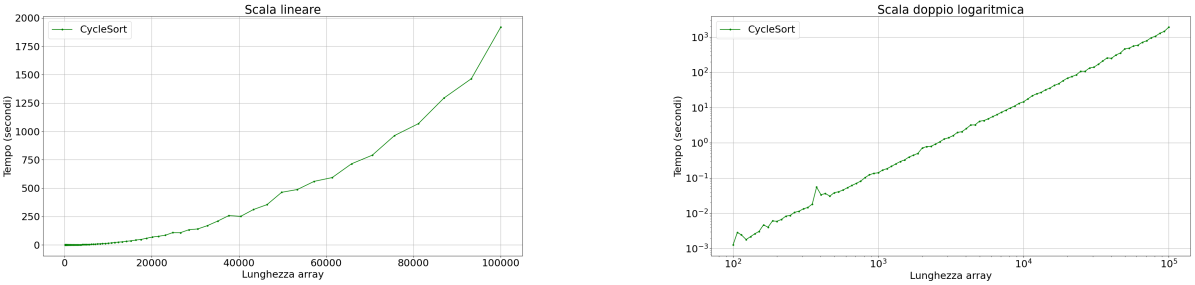
Complessità

Per ogni elemento, si vanno a scorrere $O(n)$ elementi del vettore.
Perciò la complessità è di:

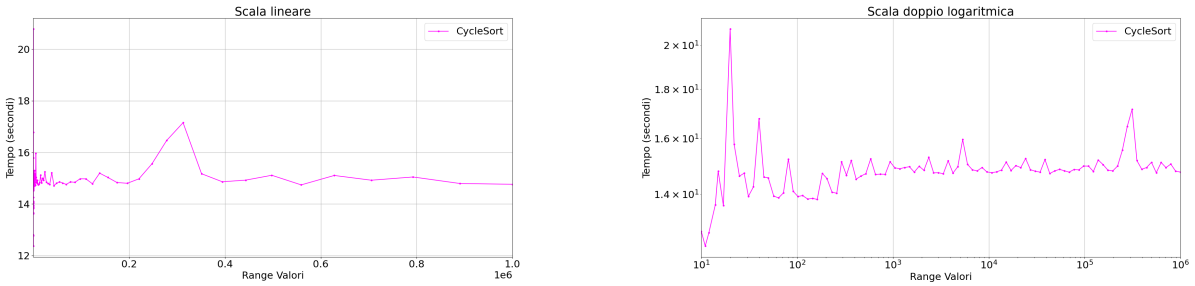
$$O(n^2)$$

L'alta complessità è causata principalmente dall'alto numero di confronti effettuati.

BenchMark



Grafici CycleSort a variare della lunghezza del vettore in input

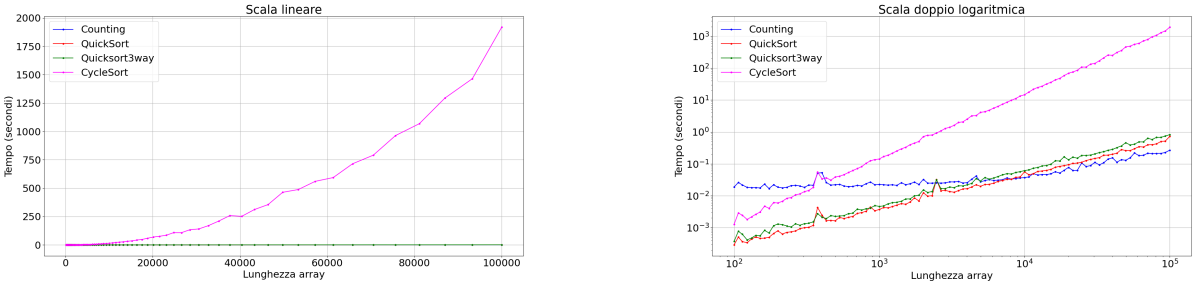


Grafici CycleSort a variare del range dei valori del vettore in input

Analisi Comparazione Algoritmi

Input lunghezza variabile

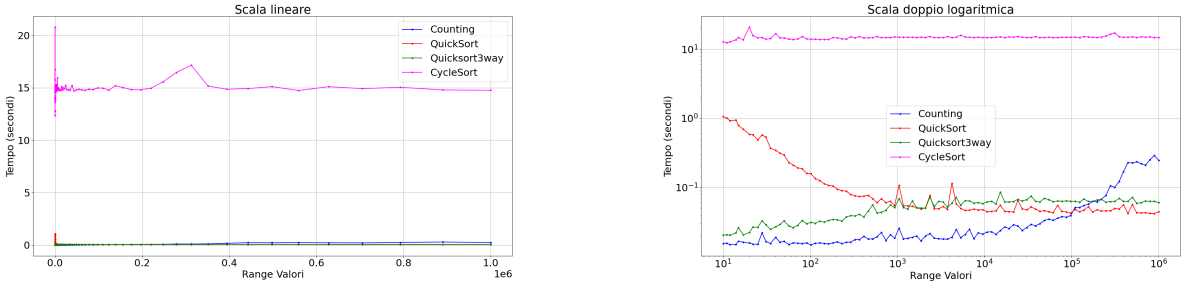
È evidente l'inefficienza temporale di CycleSort rispetto agli altri tre algoritmi, e come il tempo di esecuzione sia altamente legato alla lunghezza del vettore.
Il comportamento degli altri algoritmi risulta più chiaro nel grafico a doppia scala logaritmica.
Si nota come CountingSort diventi la scelta migliore quando il range di valori si avvicina alla lunghezza dell'input.
QuickSort e 3-way QuickSort mostrano andamenti simili, con QuickSort leggermente più performante.
Il motivo per cui 3-way QuickSort è più lento in questo contesto è che la condizione per cui il range è molto più piccolo della lunghezza dell'input (che porta a molte ripetizioni di valori) non si verifica, dato che la lunghezza massima è 100.000 e il range è da 0 a 100.000.



Grafici tutti gli algoritmi al variare della lunghezza del vettore in input con range 0-100000

Input Range Variabile

E' evidente anche qui l'inefficienza temporale di CycleSort rispetto agli altri 3 algoritmi.
Si può notare anche come l'efficienza di CountingSort sia impattata dal fatto che il range di valori con cui si sta lavorando è molto maggiore rispetto alla dimensione dell'array.
Confrontando invece QuickSort e 3-way QuickSort, si nota come al diminuire del range (e quindi all'aumentare delle ripetizioni degli stessi valori) 3-way QuickSort sia molto più efficiente di QuickSort Normale.



Grafici tutti gli algoritmi al variare del range dei valori nel vettore in input con lunghezza=10000