

Laboratorio di Algoritmi e Strutture Dati

A.A. 2024/25

Andrea Gioia - 169484@spes.uniud.it
Luca Gamberini - 168712@spes.uniud.it

20 giugno 2025

Indice

1	Introduzione	2
1.1	Specifiche tecniche	2
2	Algoritmi	2
2.1	QuickSort	2
2.1.1	Codice	3
2.2	QuickSort3Way	3
2.2.1	Codice	3
2.3	CountingSort	4
2.3.1	Codice	4
2.4	RadixSort	5
3	Misurazioni	6
4	Conclusione	6

1 Introduzione

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

- QuickSort
- QuickSort 3 Way
- CountingSort

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, mentre le misurazioni dei tempi sono state prese 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 caratteristiche 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.

Le complessità asintotiche temporali di QuickSort sono:

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

2.1.1 Codice

```
1 def QuickSort( A, p, q ):  
2     if( p < q ):  
3         r = Partition( A, p, q )  
4         QuickSort( A, p, r-1 )  
5         QuickSort( A, r+1, q )  
6     return A  
7  
8 def Partition(A, p, q):  
9     x = A[q]  
10    i = p - 1  
11    for j in range(p, q):  
12        if A[j] <= x:    # (corretto A[j], non A[q])  
13            i += 1  
14            Scambia(A, i, j)  
15    Scambia(A, i + 1, q)    # posiziona il pivot al centro  
16    return i + 1  
17  
18 def Scambia(A, i, j):  
19     temp = A[i]  
20     A[i] = A[j]  
21     A[j] = temp
```

—Commento del codice—

2.2 QuickSort3Way

Versione modificata rispetto al QuickSort classico.

Prevede la suddivisione in 3 parti dell'array di elementi, sulla base di una variabile *pivot*. Gli scambi vengono fatti sulla base di un confronto tra elemento dell'array e la variabile pivot ($>$, $<$, $=$). Le chiamate ricorsive vengono effettuate su porzioni con elementi $<$ pivot e $>$ pivot.

A differenza del QuickSort classico, QuickSort3Way presenta notevole efficienza con array con molti elementi duplicati.

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 Codice

```
1 def QuickSort3Way(arr, l, r):
```

```
2     if l >= r:
3         return
4
5     lt = l
6     i = l
7     gt = r
8     pivot = arr[l]
9
10    while i <= gt:
11        if arr[i] < pivot:
12            arr[lt], arr[i] = arr[i], arr[lt]
13            lt += 1
14            i += 1
15        elif arr[i] > pivot:
16            arr[i], arr[gt] = arr[gt], arr[i]
17            gt -= 1
18        else:
19            i += 1
20
21    QuickSort3Way (arr, l, lt - 1)
22    QuickSort3Way (arr, gt + 1, r)
23
24    return lt, gt
```

2.3 CountingSort

CountingSort è un algoritmo di ordinamento non in-place, in quanto si avvale di un array aggiuntivo che conta il numero di occorrenze di ciascun elemento.

Le complessità asintotiche temporali di CountingSort sono:

- Caso ottimo: $\Omega(n + k)$
- Caso medio: $\Theta(n + k)$
- Caso pessimo: $\mathcal{O}(n + k)$

2.3.1 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
```

2.4 RadixSort

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

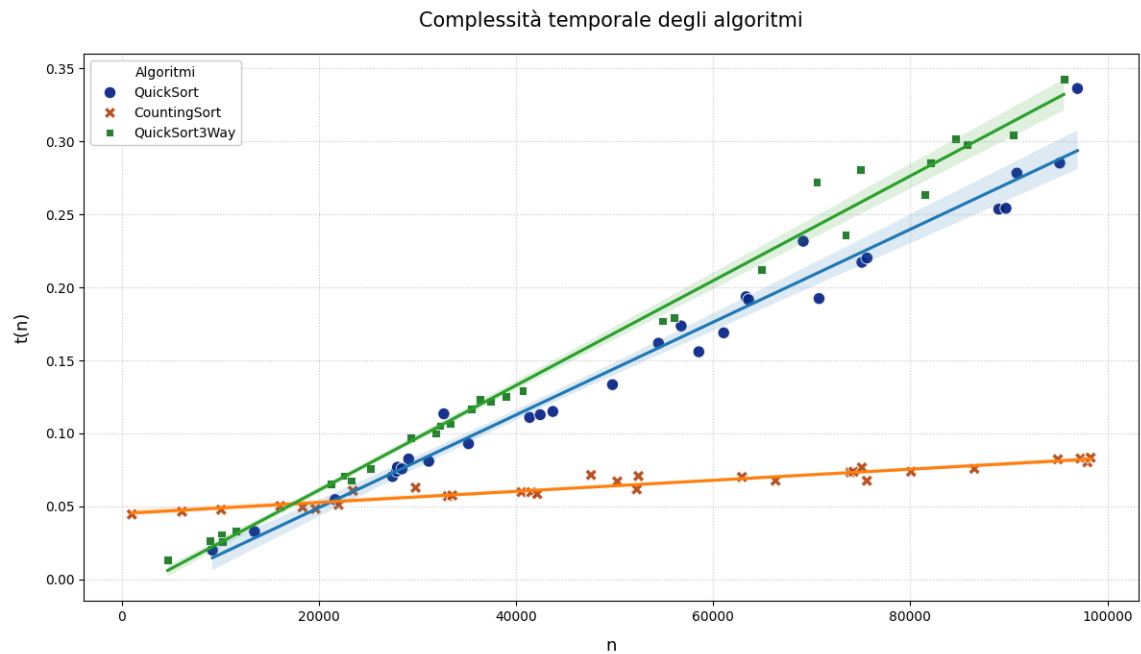
Le complessità asintotiche temporali di RadixSort sono:

- Caso ottimo: $\Omega(n \cdot d)$
- Caso medio: $\Theta(n \cdot d)$
- Caso pessimo: $\mathcal{O}(n \cdot d)$

```
1 def RadixSort(arr):
2     radix_array = [[], [], [], [], [], [], [], [], [], []] #
3     array delle cifre
4     max_val = max(arr) # assegno il valore massimo dell'array
5     exp = 1
6
7     while (max_val // exp) > 0:
8         while len(arr) > 0:
9             val = arr.pop()
10            radix_index = (val // exp) % 10
11            radix_array[radix_index].append(val)
12
13        for bucket in radix_array:
14            while len(bucket) > 0:
15                val = bucket.pop()
16                arr.append(val)
17
18        exp *= 10
```

3 Misurazioni

QuickSort



Parte reale sulle x, parte immaginaria sulle y

4 Conclusione

Massimo Boldi