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
Luca Gamberini - 168712

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

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 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.

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

```
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
```

—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

```
    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
```

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

```
    def countingSort(arr):
    max = arr[0]
    min = arr[0]

    for i in range(1, len(arr)):
        if arr[i] > max:
            max = arr[i]
```

```
        elif arr[i] < min:
            min = arr[i]

    C = [0] * (max - min + 1)
    for i in range(len(arr)):
        C[arr[i] - min] += 1

    k = 0
    for i in range(len(C)):
        while C[i] > 0:
            arr[k] = i + min
            k += 1
            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 presenta di molti numero 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: $\mathcal{O}(n \cdot d)$

2.4.1 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.

2.4.2 Codice

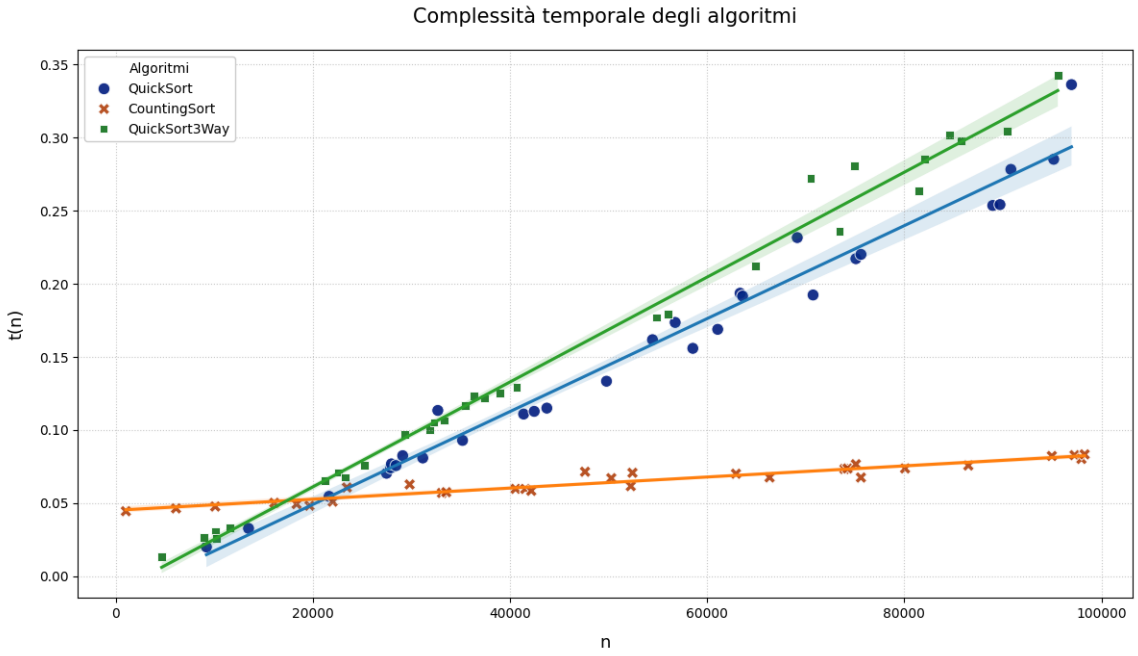
```
def RadixSort(arr):
    radix_array = [[], [], [], [], [], [], [], [], [], []]  # array delle cifre
    max_val = max(arr)  # assegno il valore massimo dell'array
    exp = 1

    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
```

3 Misurazioni

QuickSort



Parte reale sulle x, parte immaginaria sulle y

4 Conclusione

Conclusione della relazione.