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 Analisi delle complessità

Il QuickSort è un esempio di algoritmo che lavora in-place, quindi la complessità in spazio equivale a $\Theta(n)$, ovvero la dimensione dell'array di partenza.

Le operazioni che influiscono sulla complessità di tempo sono:

- Chiamata a partition, complessità $\Theta(n)$
- Le due chiamate ricorsive, complessità rispettivamente di $T(q-1)$ e $T(n-q)$.

L'equazione di ricorrenza di QuickSort è:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ T(q-1) + T(n-q) + \Theta(n) & \text{se } n > 1 \end{cases}$$

2.1.1.1 Caso pessimo

Nel caso pessimo, QuickSort ha complessità $\mathcal{O}(n^2)$.
La circostanza in cui QuickSort si trova nel caso pessimo è legata a una scelta sconveniente del pivot, il quale genera uno sbilanciamento nella divisione in parti dell'array. link alla fonte.

2.1.2 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 Analisi delle complessità

Come nel caso di QuickSort, anche il QuickSort3Way lavora in-place, senza bisogno di stutture dati aggiuntive, quindi la complessità in spazio è $\Theta(n)$.

Considerando che l'algoritmo itera gli elementi dell'array e siccome il ciclo for viene eseguito al massimo n volte.
L'algoritmo partiziona l'array in tre sezioni (elementi $<$ pivot, $=$ pivot e $>$ pivot) attraverso un singolo passaggio che opera in tempo lineare $\Theta(n)$. La complessità temporale dipende dall'equilibrio delle partizioni:

- **Caso ottimo** ($\Omega(n \log n)$): Il caso ottimo si verifica quando il pivot divide in partizioni bilanciate l'array, di conseguenza in partizioni ciascuna di dimensioni $n/3$.
- **Caso medio** ($\Theta(n \log n)$): La complessità resta invariata grazie alla gestione efficiente del confronto sugli elementi uguali, non andando a fare ulteriori confronti.
-
- **Caso pessimo** ($\mathcal{O}(n^2)$): Occorre in situazioni in cui il pivot viene sistematicamente posizionato alla prima o all'ultima posizione dell'array. Questo va a creare partizioni sbilanciate.

2.2.2 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 Analisi delle complessità

CountingSort è un algoritmo stabile e senza confronto, il quale richiede la dichiarazione di un array ausiliario di dimensione *max - min + 1*. Tutto ciò fa sì che la complessità in spazio di CountingSort sia $\mathcal{O}(n + k)$, dove n è il numero di elementi dell'array e k è pari all'ampiezza dell'intervallo di valori.

Essendo CountingSort un algoritmo di ordinamento senza confronto, il posizionamento degli elementi nell'array non influisce sulla sua efficienza, di conseguenza la complessità, sia nel caso ottimo, medio e pessimo, sarà sempre $\mathcal{O}(n + k)$; tuttavia, diventa inefficiente per intervalli di valori molto grandi ($k > n$).

2.3.2 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 Analisi delle complessità

L'algoritmo RadixSort richiede due array ausiliari, uno di dimensione pari alla base numerica scelta, e l'altro di dimensione n (numero di elementi). Di conseguenza, la complessità in spazio dell'algoritmo è pari a $\mathcal{O}(n + b)$.

Nel caso ottimo, tutti gli elementi hanno lo stesso numero di cifre. Considerando d come il numero delle cifre, la complessità totale sarà $\mathcal{O}(d(n+b))$.

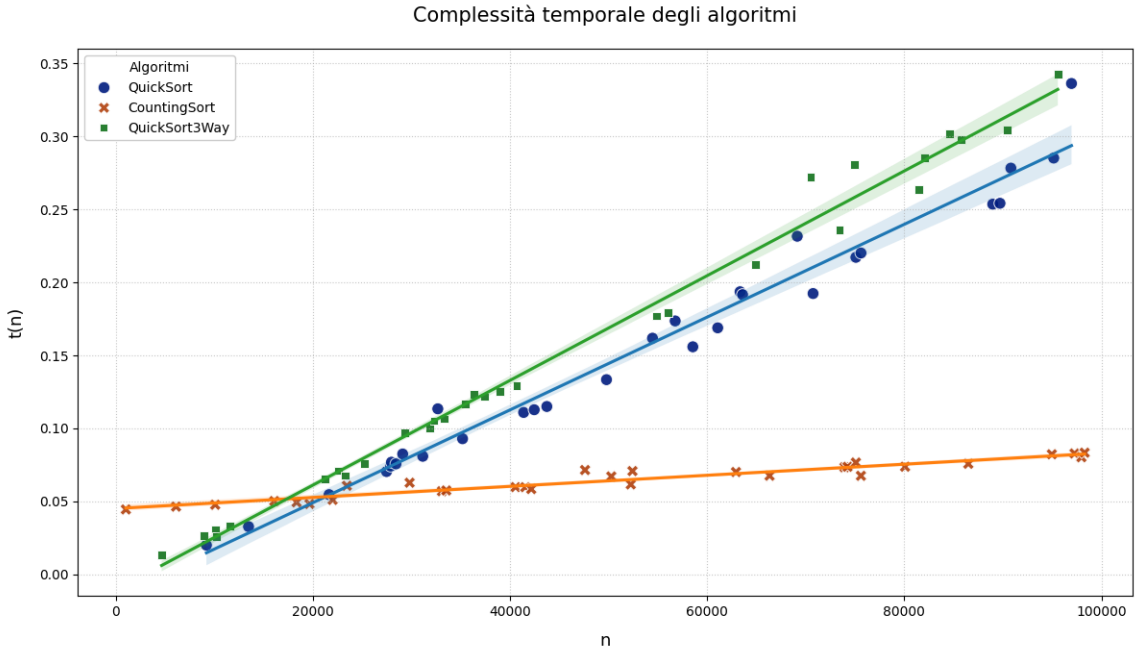
Nel caso pessimo, tutti gli elementi hanno lo stesso numero di cifre, ad eccezione di un elemento che avrà un numero di cifre molto più elevato. Se il numero di cifre dell'elemento più grande è pari a d, la complessità temporale diventa $\mathcal{O}(d(n+b))$. Mediamente, la complessità temporale nel caso medio di RadixSort è di $\mathcal{O}(d(n+b))$, dove n è il numero di elementi e b è la base di rappresentazione.

2.4.3 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)

        for bucket in radix_array:
            while len(bucket) > 0:
                val = bucket.pop()
                arr.append(val)

        exp *= 10
```

3 Misurazioni

QuickSort



Parte reale sulle x, parte immaginaria sulle y

4 Conclusione

Conclusione della relazione.