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
Kent - 168711

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.
Le misurazioni dei tempi sono state acquisite 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

```
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 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

Introduzione algoritmo

Il *Counting Sort* è un algoritmo di ordinamento non basato su confronti, adatto quando i valori da ordinare sono interi non negativi con intervallo di valori relativamente contenuto rispetto al numero di elementi.

Idea
Contare le occorrenze di ciascun valore in un array ausiliario, calcolare una somma cumulativa (prefix sum) e quindi posizionare ogni elemento in output nel posto corretto per ottenere un ordinamento stabile.

Complessità

Counting Sort assume che ciascuno dei n elementi in input sia un intero nell'intervallo $[0, k]$. L'algoritmo non utilizza confronti tra elementi, ma lavora sfruttando il valore numerico degli stessi come indice in un array ausiliario. Questo approccio consente di superare il limite inferiore $\Omega(n \log n)$ valido per gli algoritmi di ordinamento basati su confronti.
La complessità dell'algoritmo è la seguente:

- **Caso migliore:** Quando $k = O(n)$, cioè quando l'intervallo dei valori è proporzionale al numero di elementi, l'intero algoritmo opera in tempo lineare: $\Theta(n)$. In

questo scenario, il Counting Sort è estremamente efficiente e utilizza $\Theta(n)$ spazio aggiuntivo.

- **Caso medio:** In situazioni intermedie, in cui k non è trascurabile ma nemmeno molto più grande di n, l'algoritmo ha una complessità di $\Theta(n+k)$. Il comportamento resta comunque più efficiente rispetto a molti algoritmi basati su confronti (come quicksort o mergesort), specialmente quando l'intervallo k rimane relativamente contenuto.
- **Caso peggiore:** Quando $k \gg n$, cioè l'intervallo dei valori è molto più ampio del numero di elementi, la complessità diventa $\Theta(n+k)$, ma con un impatto significativo in termini di memoria e tempo. L'array ausiliario $C[0 \dots k]$ può occupare molto spazio anche se pochi valori sono effettivamente presenti, portando a inefficienze.

Counting Sort è anche stabile: mantiene l'ordine relativo degli elementi con valore uguale, caratteristica fondamentale quando si lavora con dati associati (satellite data) o quando viene usato come sottoprocedura in algoritmi come il Radix Sort.
Questa stabilità è garantita dallo scorrimento dell'array originale in senso inverso durante la copia finale nell'array di output.

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
```

Funzionamento del codice

Il seguente frammento implementa una versione modificata del Counting Sort che supporta anche numeri negativi. Di seguito si spiega il funzionamento passo dopo passo:

1. **Individuazione del massimo e del minimo:** Si inizializzano due variabili `max` e `min` con il primo elemento dell'array. Successivamente, si scorre l'array per determinare il valore massimo e minimo effettivi. Questo passaggio è fondamentale per calcolare correttamente l'intervallo degli elementi e adattare l'algoritmo anche a numeri negativi.

2. **Inizializzazione dell'array dei conteggi:** Si crea un array ausiliario `C` di dimensione `(max - min + 1)`, inizializzato a zeri. Questo array verrà utilizzato per contare quante volte ogni valore appare nell'array originale. La sottrazione di `min` serve a traslare i valori negativi in indici validi (a partire da 0).

3. **Conteggio delle occorrenze:** Si scorre l'array originale `arr` e, per ogni valore `arr[i]`, si incrementa `C[arr[i] - min]`. In questo modo, ogni posizione dell'array `C` conterrà il numero di occorrenze del valore corrispondente.

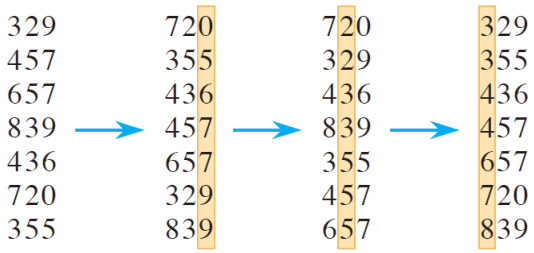
4. **Ricostruzione dell'array ordinato:** Con un doppio ciclo (for + while), si scorrono tutti i valori dell'array `C`. Per ogni indice `i`, finché `C[i]` è maggiore di zero, si inserisce il valore corrispondente (`i + min`) nell'array originale `arr` alla posizione `k`, incrementando `k` e decrementando `C[i]`. Questo processo ricostruisce l'array ordinato in modo diretto, sovrascrivendo l'input.

2.4 RadixSort

Introduzione algoritmo

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.

Idea
RadixSort prevede di prendere in esame le singole cifre dei numeri da ordinare, in base alla loro posizione, andando successivamente a posizionarle in ordine crescente o decrescente.
Questo processo viene svolto ricorsivamente per ciascuna colonna di cifre, partendo dalla meno significativa.



Esempio di ordinamento (non riesco a centrare la caption)

Radix Sort, per funzionare correttamente, richiede un algoritmo di ordinamento stabile per l'ordinamento delle singole cifre (o caratteri, a seconda del caso).

Solitamente, come algoritmo sottostante viene utilizzato Counting Sort, il quale, però, nella sua versione tradizionale non è stabile. Infatti, in presenza di chiavi uguali, Counting Sort potrebbe modificare l'ordine relativo degli elementi rispetto all'array di input.

Per garantire la stabilità, è sufficiente iterare l'array di input in ordine inverso durante la fase di costruzione dell'array di output. In questo modo, quando più elementi hanno la stessa chiave, essi verranno copiati nell'output nell'ordine in cui compaiono nell'input, preservando così la stabilità dell'ordinamento.

Complessità

Caso medio Considerando n elementi da ordinare, ciascuno composto da d cifre, e assumendo che ogni cifra possa assumere al più k valori distinti, Radix Sort ha una complessità asintotica pari a $\Theta(d(n+k))$.

Tuttavia, ciò è valido solo se l'algoritmo utilizzato per ordinare le singole cifre (come Counting Sort) ha complessità $\Theta(n+k)$, ovvero è lineare rispetto al numero di elementi e all'ampiezza del dominio delle cifre.

Caso ottimo Le condizioni del caso ottimo di RadixSort sono:

- d piccolo;
- $k < n$.

La complessità rimane $\Theta(d(n+k))$, tuttavia, con valori di d e k costanti si ottiene un $\Theta(n)$.

Caso pessimo Il caso peggiore si verifica nelle condizioni di:

- d elevato (numeri molto grandi);
- $k \geq n$ (rende CountingSort inefficiente).

La complessità resta $\mathcal{O}(d(n+k))$, ma può degenerare fino a diventare $\mathcal{O}(n \cdot log(n))$

2.4.1 Motivazioni della scelta

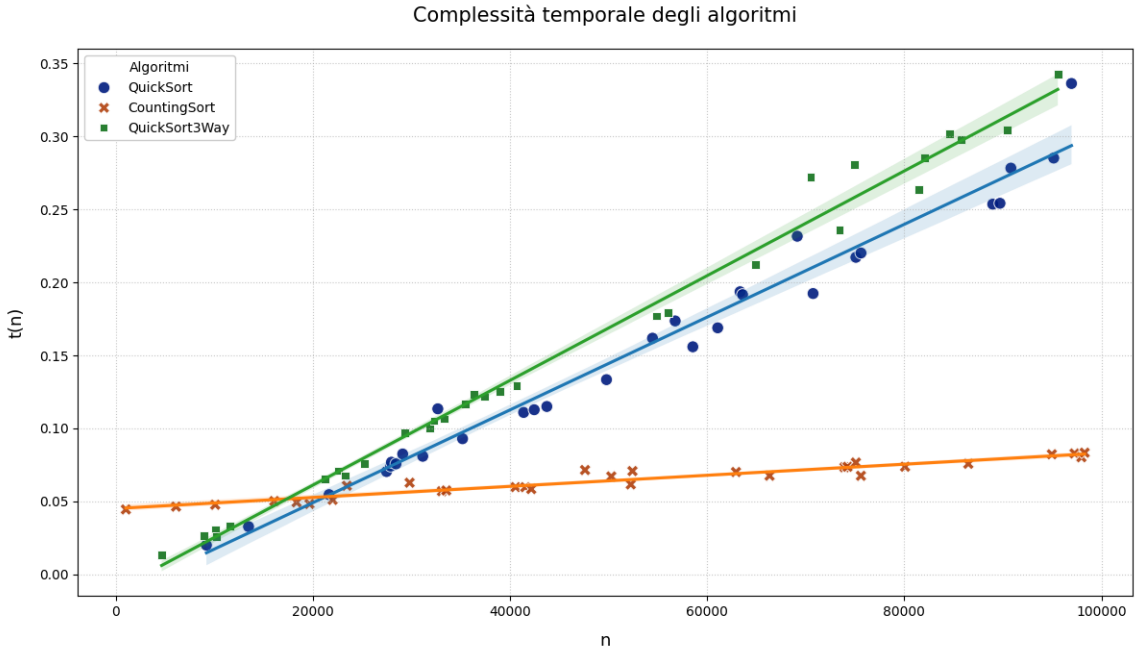
É stato scelto RadixSort come quarto algoritmo a scelta per il suo approccio innovativo e diverso rispetto agli altri algoritmi selezionati, offrendo una prospettiva di confronto interessante.

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)

        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.