Il Problema dell'Ordinamento 2/2

Algoritmi 1 – Lezione 8

A. Monti
Sapienza Università di Roma

Corso: Algoritmi 1

Indice

1. Algoritmi di ordinamento in tempo $O(n \log n)$
   - Introduzione
   - Mergesort
     - Implementazione ricorsiva
     - Implementazione iterativa
   - Quicksort
     - Analisi del caso medio

Ordinamento in $O(n \log n)$

Abbiamo visto che gli algoritmi di ordinamento basati sui confronti hanno una complessità asintotica inferiore limitata da $\Omega(n \log n)$.

In questa presentazione illustreremo due algoritmi che raggiungono tale limite ottimale:

- Mergesort;
- Quicksort.

Per ciascuno illustreremo il funzionamento, forniremo un esempio di implementazione, analizzeremo la complessità, e valuteremo i vantaggi e gli svantaggi.

Mergesort: Funzionamento Generale

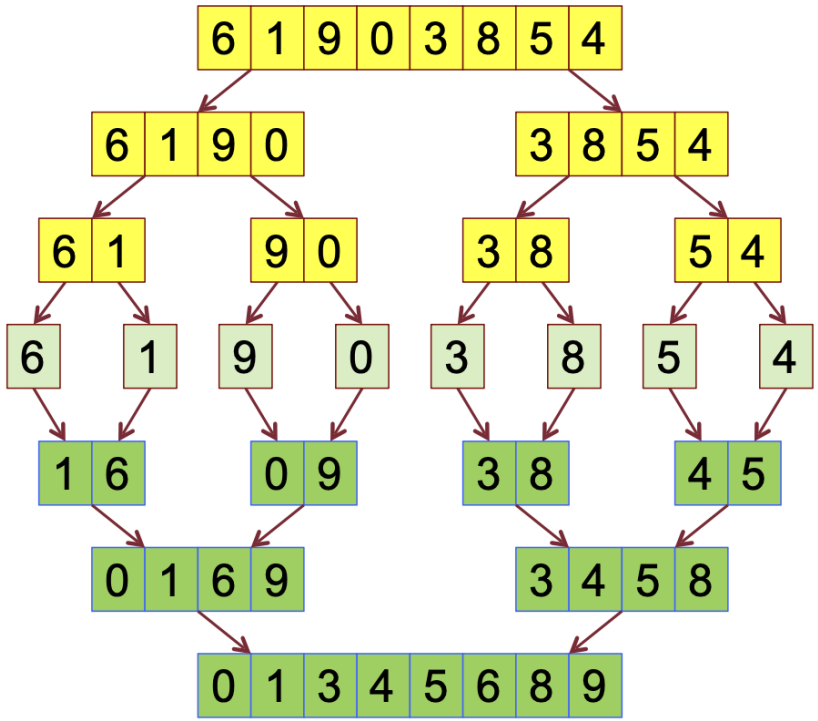
Il **Mergesort** (ordinamento per fusione) è un algoritmo che adotta il paradigma *divide et impera*.

Formalmente, dato un array di n elementi:

- esso viene suddiviso in due sottosequenze di circa $n/2$ elementi;
- le due sottosequenze vengono ordinate ricorsivamente;
- infine, le sottosequenze ordinate vengono fuse per ottenere la sequenza finale ordinata.

Mergesort: Esempio di Esecuzione

- L'array viene suddiviso ricorsivamente fino ad arrivare a sequenze di un solo elemento;
- Poi le sottosequenze vengono fuse progressivamente in sequenze ordinate.



Mergesort: Codice Principale con intervalli

Funzione principale:

```
def merge_sort(A):
    if len(A) > 1:
        m = len(A) // 2
        As = A[:m]
        Ad = A[m:]
        merge_sort(As)
        merge_sort(Ad)
        return  fondi(As, Ad)
```

Osservazioni:

- La lista viene divisa in due metà;
- Si ordinano ricorsivamente le due sottoliste;
- la funzione **fondi** combina le sottoliste ordinate.

Mergesort: Codice della Fusione con intervalli

Idea: dato che `As` e `Ad` sono ordinate:

- Il minimo globale è il più piccolo tra i minimi delle due sottoliste;
- Rimuovendo il minimo da una lista, si conserva la proprietà;
- Quando una lista termina, si copia l'altra così com'è.

```
def fondi(As, Ad):
    i = j = 0
    B = [ ]
    while i < len(As) and j < len(Ad):
        if As[i] <= Ad[j]:
            B.append(As[i])
            i += 1
        else:
            B.append(Ad[j])
            j += 1
    while i < len(As):
        B.append(As[i])
        i += 1
    while j < len(Ad):
        B.append(Ad[j])
        j += 1
    return B
```

Mergesort: Analisi della Complessità

Complessità temporale:

- Ogni chiamata divide l'array in due e costa $\Theta(n)$ per la fusione;
- Ricorrenza:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ 2T(n/2) + \Theta(n) & \text{altrimenti} \end{cases}$$

- Soluzione:

$$T(n) = \Theta(n \log n)$$

Mergesort: Implementazione con Intervalli

Nell'implementazione precedente, per semplicità didattica, abbiamo creato nuove sottoliste da ordinare e poi fondere.

Tuttavia, è possibile implementare `merge_sort` senza creare nuove liste, operando direttamente su porzioni dell' array originale, definite da due indici i e j, che delimitano l'intervallo $A[i : j]$.

Nota: anche in questo caso, la funzione di fusione `fondi` richiede comunque un vettore d'appoggio temporaneo per eseguire correttamente la fusione.

Mergesort: Implementazione con indici

```
def merge_sort(A, i, j):
    if i < j:
        m = (i + j) // 2
        merge_sort(A, i, m) # ordina A[i .. m]
        merge_sort(A, m+1, j) # ordina A[m +1 .. j]
        fondi(A, i, m, j) # fondi A[i .. m] e A[m+1 .. j]

def fondi(A, i, m, j):
    B = [ ]              # Lista d'appoggio per la parte ordinata
    p, q = i, m+1        # p scorre su A[i .. m], q su A[m+1 .. j]
    # Fusione dei due intervalli ordinati
    while p <= m and q <= j:
        if A[p] <= A[q]:
            B.append(A[p])
            p += 1
        else:
            B.append(A[q])
            q += 1
    # Copia gli eventuali elementi rimanenti della prima metà
    while p <= m:
        B.append(A[p])
        p += 1
    # Copia gli eventuali elementi rimanenti della seconda metà
    while q <= j:
        B.append(A[q])
        q += 1
    # Copia la lista ordinata B nella posizione corretta di A
    A[i : j +1] = B
```

Mergesort: Considerazioni sull'Efficienza

Vantaggi dell'approccio con indici:

- Non vengono create nuove sottoliste ad ogni chiamata ricorsiva;
- Si lavora direttamente sull'array originale, evitando copia e slicing ripetuti.

Limite: la funzione `fondi` richiede comunque uno spazio aggiuntivo per un vettore temporaneo B, necessario per eseguire la fusione ordinata.

Conclusione: Mergesort non è un algoritmo *in-place*, poiché utilizza memoria extra proporzionale alla dimensione dell'array.

Merge Sort iterativo (bottom-up) 1/2

- L'algoritmo parte da sottosequenze ordinate di lunghezza $k = 1$ e raddoppia progressivamente k ad ogni iterazione:

$$k = 1, 2, 4, 8, \ldots$$

- Ad ogni iterazione, l'array viene attraversato da sinistra a destra fondendo coppie di sottosequenze contigue di lunghezza k.
  - Se l'ultima sottosequenza è più corta di k, non viene fusa (è già ordinata).
- Nella fase k la funzione merge viene applicata circa $\frac{n}{2^k}$ a liste di dimensione al più 2^k. Il costo di ogni iterazione è dunque $\Theta(n)$.
- Poiché la lunghezza delle sottosequenze raddoppia ad ogni iterazione, il numero totale di iterazioni è $\lceil \log n \rceil$.
- Da cui la complessità totale dell'algoritmo è:

$$\Theta(n \log n)$$

Merge Sort iterativo 2/2

Ecco un possibile codice in python

```
def merge_sort_iterativo(A):
    n = len(A)
    k = 1
    while k < n:
        for i = 0 to n-1 step 2*k:
            l = i
            m = min(i + k, n)
            r = min(i + 2*k, n)
            if m < r:
                fondi(A, l, m, r)
        k = 2*k
```

Quicksort: Funzionamento Generale

Il **Quicksort** è un algoritmo di ordinamento basato sul paradigma *divide et impera*.

Più precisamente:

- Si seleziona un **pivot** all'interno della lista;
- Si partiziona la lista attorno al pivot: a sinistra gli elementi minori, a destra quelli maggiori o uguali, con il pivot posizionato correttamente;
- Si applica ricorsivamente l'algoritmo alle due sottoliste risultanti.

L'algoritmo è corretto perché il pivot viene sempre posizionato nella sua posizione definitiva e le sottoliste vengono ordinate separatamente.

Per implementare Quicksort in modo efficiente, è importante evitare la creazione di nuove sottoliste ad ogni passo ricorsivo. Questo riduce il consumo di memoria e permette di ordinare direttamente l'array originale.

Quicksort in-place

Obiettivo: realizzare un'implementazione del Quicksort che ordini **in loco**, cioè direttamente sulla lista da ordinare, senza strutture ausiliarie.

Ecco una possibile implementazione ricorsiva in-place:

```
def Quick_sort(A, primo, ultimo):
    if primo < ultimo:
        medio = Partiziona(A, primo, ultimo)
        Quick_sort(A, primo, medio - 1)
        Quick_sort(A, medio + 1, ultimo)
```

Resta da definire la funzione `Partiziona`, che:

- sceglie un pivot,
- riordina **in loco** gli elementi, mettendo quelli **minori del pivot** a sinistra e quelli **maggiori o uguali** a destra.

Restituisce infine la posizione finale del pivot.

Funzione di Partizione

La funzione `Partiziona` seleziona un pivot e riorganizza l'array in modo che:

- a sinistra ci siano solo gli elementi minori del pivot;
- a destra quelli maggiori o uguali;
- il pivot venga spostato nella sua posizione finale.

```
def Partiziona(A, inizio, fine):
    pivot = A[inizio]
    i = inizio + 1
    j = fine
    while i <= j:
        if A[i] < pivot:
            i += 1
        elif A[j] >= pivot:
            j -= 1
        else:
            A[i], A[j] = A[j], A[i]
            i += 1
            j -= 1
    A[inizio], A[j] = A[j], A[inizio]
    return j
```

Quicksort: Analisi della Complessità

- **Caso migliore:** partizione bilanciata

$$T(n) = 2T(n/2) + \Theta(n) \Rightarrow \Theta(n \log n)$$

- **Caso peggiore:** partizione sbilanciata

$$T(n) = T(n-1) + \Theta(n) \Rightarrow \Theta(n^2)$$

- **Caso medio:** pivot scelto casualmente

$$\Rightarrow \Theta(n \log n)$$

Conclusione: nonostante il caso pessimo, il quicksort è molto efficiente in pratica. La versione in-place richiede solo $O(\log n)$ spazio di stack ricorsivo.

Scelta casuale del Pivot

Analizziamo la complessità del **quicksort** nel caso medio.
Per semplicità, assumiamo che:

- Tutti gli elementi della lista A siano **distinti**.
- Il pivot venga scelto in modo **casuale**, con probabilità uniforme.

In particolare, vogliamo che **ogni elemento** della sottolista $A[i : j]$ abbia la stessa probabilità di essere scelto come pivot. Per ottenere ciò, basta scambiare un elemento casuale con il primo:

```
import random

k = random.randint(i, j)        # scegli indice casuale
A[i], A[k] = A[k], A[i]         # porta il pivot all'inizio
pivot = A[i]                    # scegli A[i] come pivot
```

Dopo questo scambio, l'elemento $A[i]$ è un pivot scelto *uniformemente a caso* tra gli elementi $A[i], \ldots, A[j]$.

Questa scelta casuale riduce la probabilità di partizioni sbilanciate e permette di analizzare il **caso medio** del quicksort.

Ricorrenza al caso medio

- Sia $T(n)$ il tempo atteso di esecuzione del quicksort su una lista di n elementi distinti.
- Con la scelta casuale del pivot, ogni elemento ha probabilità $\frac{1}{n}$ di essere il pivot.
- Supponiamo che il pivot scelto sia l'elemento in posizione k (con $0 \leq k \leq n-1$). Allora la lista si divide in due sottoliste di dimensione k e $n-1-k$, su cui si applica ricorsivamente il quicksort.
- La ricorrenza per il tempo atteso diventa:

$$T(n) = \frac{1}{n} \sum_{k=0}^{n-1} \left(T(k) + T(n-1-k) \right) + \Theta(n)$$

  dove $\Theta(n)$ rappresenta il costo della partizione e delle operazioni lineari.
- Osservando che per ogni coppia $(k, n-1-k)$, il termine $T(q)$ compare due volte, possiamo riscrivere la ricorrenza in forma più compatta:

$$T(n) = \frac{2}{n} \sum_{q=0}^{n-1} T(q) + \Theta(n)$$

  Questa è la ricorrenza da studiare per il caso medio.
- In generale, la risoluzione di questa ricorrenza porta a:

$$T(n) = O(n \log n)$$

Metodo di sostituzione 1/3

Risolveremo la ricorrenza del caso medio del quicksort:

$$T(n) = \frac{2}{n} \sum_{q=0}^{n-1} T(q) + \Theta(n)$$

utilizzando il **metodo di sostituzione**.

Per semplicità, eliminiamo i termini asintotici e scriviamo:

$$T(n) = \begin{cases} a & \text{se } n \leq 1, \\ \frac{2}{n} \sum_{q=0}^{n-1} T(q) + bn & \text{se } n \geq 2, \end{cases}$$

dove a e b sono costanti positive.

Ipotesi induttiva: proponiamo

$$T(n) \leq c(n+1)\log_2(n+2)$$

per una costante c da determinare.
Nota: ipotizzando più semplicemente $T(n) \leq cn \log_2 n$ il caso base non sarebbe valido ecco perché aggiungiamo $+1$ e $+2$.

Metodo di sostituzione 2/3

Casi base: per $n = 0$ e $n = 1$, basta scegliere $c \geq a$.
Passo induttivo: applichiamo l'ipotesi alla ricorrenza:

$$\begin{aligned} T(n) &\leq \frac{2}{n} \sum_{q=0}^{n-1} T(q) + bn \\ &\leq \frac{2}{n} \sum_{q=0}^{n-1} c(q+1)\log_2(q+2) + bn \\ &= \frac{2c}{n} \sum_{q=0}^{n-1} (q+1)\log_2(q+2) + bn \end{aligned}$$

Stimiamo la sommatoria:

$$\sum_{q=0}^{n-1} (q+1)\log_2(q+2) \leq \frac{n(n+1)}{2}\log_2(n+2) - \frac{n^2}{8}$$

Questa stima è standard per ottenere un termine dominante $\sim n^2 \log n$ e isolare il termine lineare.
Sostituendo:

$$\begin{aligned} T(n) &\leq \frac{2c}{n}\left(\frac{n(n+1)}{2}\log_2(n+2) - \frac{n^2}{8}\right) + bn \\ &= c(n+1)\log_2(n+2) - \frac{cn}{4} + bn \end{aligned}$$

Notiamo che se scegliamo $c \geq 4b$, allora $-\frac{cn}{4} + bn \leq 0$, e quindi $T(n) \leq c(n+1)\log_2(n+2)$

Conclusione: per $c = 4 \cdot \max\{a, b\}$, lipotesi induttiva è verificata per tutti $n \geq 2$, e quindi $T(n) = O(n \log n)$ dimostrando che il quicksort ha complessità media $O(n \log n)$.

Metodo di sostituzione - Stima della sommatoria 1/2

Per completare la dimostrazione della complessità media, dobbiamo giustificare:

$$\sum_{q=0}^{n-1}(q+1)\log_2(q+2) \leq \frac{n(n+1)}{2}\log_2(n+2) - \frac{n^2}{8}$$

1. Controllo rapido tramite integrale (solo per intuizione): Osserviamo che $(q+1)\log_2(q+2)$ è crescente. Quindi:

$$\sum_{q=0}^{n-1}(q+1)\log_2(q+2) \leq \int_0^n (x+1)\log_2(x+2)\,dx$$

L'integrale può essere calcolato esattamente e conferma la stima dell'ordine di grandezza $\sim n^2 \log n$, giustificando la scelta della maggiorazione.

Metodo di sostituzione - Stima della sommatoria 1/2

2. Dimostrazione discreta completa (senza integrali): Dividiamo la sommatoria in due parti nel punto

$$r = \left\lfloor \frac{n-2}{2} \right\rfloor :$$

$$\sum_{q=0}^{n-1}(q+1)\log_2(q+2) = \sum_{q=0}^{r}(q+1)\log_2(q+2) + \sum_{q=r+1}^{n-1}(q+1)\log_2(q+2)$$

Maggioriamo ciascun termine:

$$\sum_{q=0}^{r}(q+1)\log_2(q+2) \leq \sum_{q=0}^{r}(q+1)\log_2(r+2) \leq \sum_{q=0}^{r}(q+1)(\log_2(n+2)-1)$$

$$\sum_{q=r+1}^{n-1}(q+1)\log_2(q+2) \leq \sum_{q=r+1}^{n-1}(q+1)\log_2(n+2)$$

Ricombinando:

$$\sum_{q=0}^{n-1}(q+1)\log_2(q+2) \leq \sum_{q=0}^{n-1}(q+1)\log_2(n+2) - \sum_{q=0}^{r}(q+1)$$

Poiché

$$\sum_{q=0}^{n-1}(q+1) = \frac{n(n+1)}{2}, \quad \sum_{q=0}^{r}(q+1) \geq \frac{n^2}{8},$$

ottieniamo la stima finale:

$$\sum_{q=0}^{n-1}(q+1)\log_2(q+2) \leq \frac{n(n+1)}{2}\log_2(n+2) - \frac{n^2}{8}$$

Questa disuguaglianza è esattamente quella che abbiamo usato nel metodo di sostituzione.