Corso di laurea in Informatica
Introduzione agli Algoritmi

Il problema dell'ordinamento: algoritmo Quicksort

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

Quicksort

L'algoritmo ***quicksort*** (***ordinamento veloce***) ha costo $O(n^2)$ nel caso peggiore ma nella pratica è spesso la soluzione migliore per grandi valori di n perché:

- il suo tempo di esecuzione atteso è $\Theta(n \log n)$
- i fattori costanti nascosti sono molto piccoli
- permette l'ordinamento "in loco".

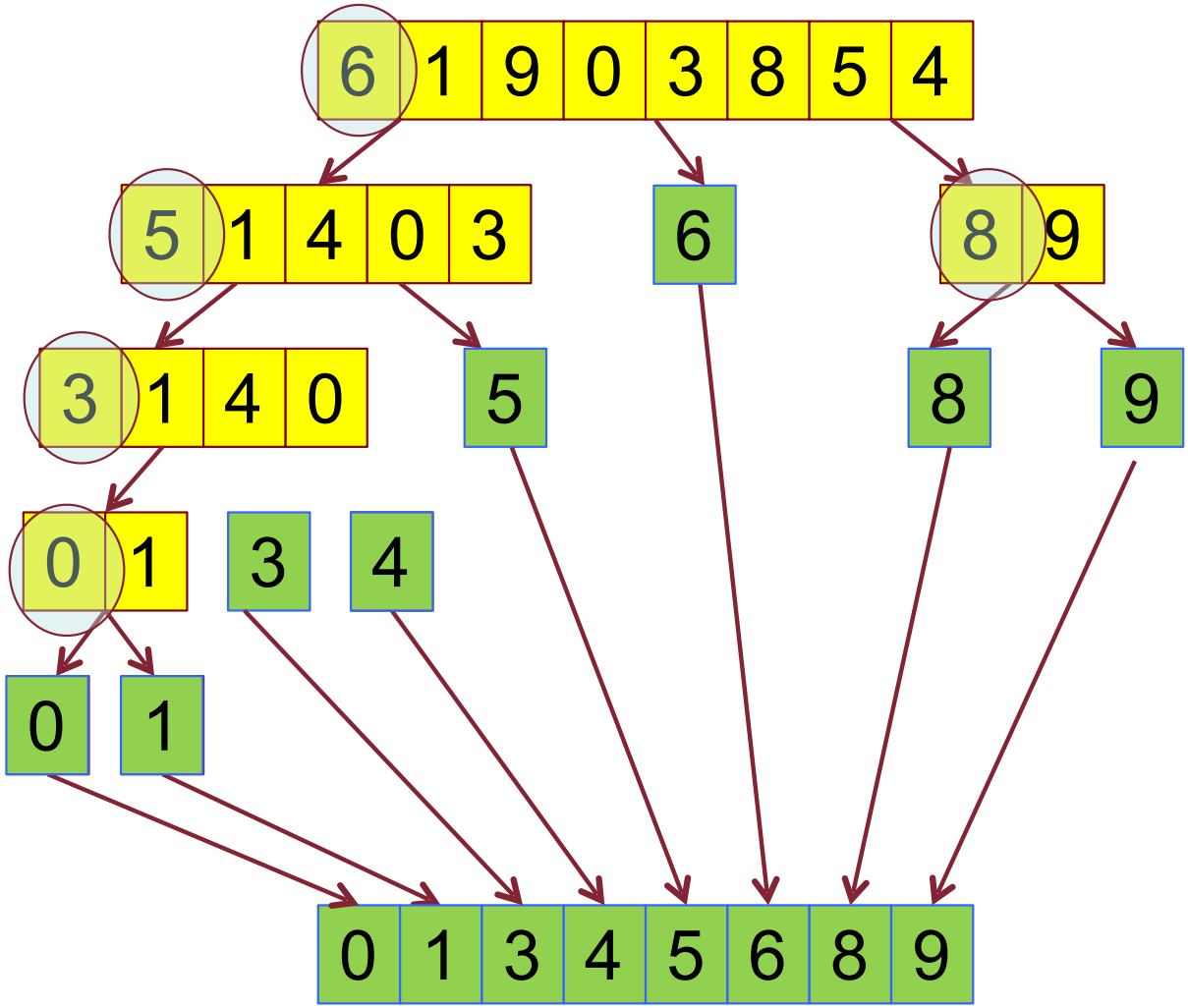
Riunisce i vantaggi del Selection sort (ordinamento in loco) e del Merge sort (ridotto tempo di esecuzione). Ha però lo svantaggio dell'elevato costo computazionale nel caso peggiore.

Il Quicksort è un algoritmo di tipo *divide-et-impera* che seleziona un elemento, detto *pivot*, e partiziona l'array attorno al pivot.

Più precisamente:

- Nella lista di n elementi si seleziona un pivot.
- Il pivot viene posizionato nella sua corretta posizione, in modo da ottenere due sottosequenze: una contenente gli elementi minori del pivot e l'altra gli elementi maggiori o uguali al pivot.
- L'algoritmo viene quindi applicato ricorsivamente su entrambe le sottosequenze.

Quicksort



- ***divide***: nella sequenza di *n* elementi seleziona un ***pivot***. Viene individuata la giusta posizione del pivot e otteniamo due sottosequenze: quella degli elementi minori o uguali del pivot, e quella degli elementi maggiori del pivot;

- ***impera***: le due sotto-sequenze vengono ordinate ricorsivamente;

- ***passo base***: la ricorsione procede fino a quando le sottosequenze sono costituite da un solo elemento;

- ***combina***: non occorre

Una prima implementazione che non ordina in loco e restituisce una copia ordinata del vettore:

```
def quick_sort(A):
    if len(A) <= 1:
        # L' array di lunghezza 0 o 1 è già ordinato
        return A
    # Scegli il primo elemento come pivot
    pivot = A[0]
    # Partiziona l'array rispetto al pivot
    As = [ ] # Elementi minori del pivot
    Ad = [ ] # Elementi maggiori o uguali al pivot
    for i in range(1, len(A)):
        if A[i] < pivot:
            As.append(A[i])
        else:
            Ad.append(A[i])
    # Chiama ricorsivamente il quicksort su As e Ad
    # e unisci i risultati
    return quick_sort(As) + pivot + quick_sort(Ad)
```

```python
def quick_sort(A):
    if len(A) <= 1:
        return A
    pivot = A[0]
    As = [ ]
    Ad = [ ]
    for i in range(1,len(A):
        if A[i] < pivot:
            As.append(A[i])
        else:
            Ad.append(A[i])
    return quick_sort(As) + pivot + quick_sort(Ad)
```

Per quanto riguarda la complessità temporale, si ottiene la seguente ricorrenza:

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

dove k, con $0 \leq k \leq n-1$, rappresenta la posizione corretta del pivot all'interno della lista ordinata.

Non è difficile dimostrare, ad esempio per sostituzione, che:

$$\Omega(n \log n) \leq T(n) \leq O(n^2).$$

In particolare, per la **complessità temporale** si distinguono due casi:

- **Caso ottimo**:
  - Il pivot divide l'array in due parti bilanciate, ciascuna di dimensione circa $\frac{n}{2}$.
  - Il costo della partizione è $\Theta(n)$, poiché ogni elemento viene confrontato con il pivot, e il costo della ricomposizione rimane $\Theta(n)$.
  - La complessità risulta quindi:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) = \Theta(n \log n).$$

- **Caso pessimo**:
  - Si verifica quando il pivot scelto è sempre l'elemento più piccolo o più grande, come nel caso di un array già ordinato o inversamente ordinato.
  - La divisione è estremamente sbilanciata: una sottolista contiene tutti gli elementi tranne il pivot, l'altra è vuota.
  - La ricorrenza diventa:

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

L' implementazione proposta, sebbene concettualmente chiara, presenta un problema principale:

- Ogni partizionamento crea nuove liste (As e Ad), aumentando l'uso di memoria e il costo di copia.

Analisi della complessità spaziale

Caso ottimo:

- La profondità della ricorsione è $O(\log n)$, poiché ad ogni passo la dimensione delle sottoliste si dimezza.
- A livello i dello stack è memorizzata una lista di dimensione $\frac{n}{2^i}$.
- Lo spazio totale utilizzato risulta:

$$\sum_{i=0}^{\log_2 n} \frac{n}{2^i} \cdot \Theta(1) = \Theta(n) \cdot \sum_{i=0}^{\log_2 n} \frac{1}{2^i} = \Theta(n) \cdot \Theta(1) = \Theta(n).$$

Caso pessimo:

- La profondità massima della ricorsione è $n-1$, poiché ad ogni passo la dimensione della lista si riduce di 1.
- A livello i dello stack è memorizzata una lista di dimensione $n-i$.
- Lo spazio totale utilizzato risulta:

$$\sum_{i=0}^{n-1} (n-i) \cdot \Theta(1) = \sum_{j=1}^{n} j \cdot \Theta(1) = \frac{n(n+1)}{2} \cdot \Theta(1) = \Theta(n^2).$$

Fortunatamente a differenza di quanto fa la funzione *fondi* del *merge_sort*, la partizione richiesta dal *quick_sort* può essere eseguita "in loco", evitando cosi di creare sottoliste e la successiva concatenazione delle stesse al ritorno della ricorsione.

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

In questa implementazione utilizziamo una funzione *Partition*. La funzione sceglie nel sotto-vettore di A che va dalla posizione *primo* alla posizione *ultimo* un elemento dell'array, detto pivot, e posiziona tutti gli elementi in modo che a sinistra del pivot ci siano solo elementi minori del pivot e a destra elementi maggiori o uguali al pivot. In questo modo, il pivot si trova in posizione corretta nella sequenza ordinata. Tutto questo viene fatto senza utilizzare spazio ausiliario. La funzione infine restituisce la posizione in cui è finito il pivot.

```python
def Partition(A, primo, ultimo):
    # Scegliamo come pivot della sottolista A[primo: ultimo+1]
    # il primo elemento
    pivot = A[primo]
    i ,j = primo + 1, ultimo
    while True:
        while i <= ultimo and A[i] <= pivot:
            i += 1
        while A[j] > pivot:
            j -= 1
        if i < j:
            A[i], A[j] = A[j], A[i]
        else:
            A[primo], A[j] = A[j], A[primo]
            return j
```

Ad ogni iterazione del primo *while* interno l'indice i si incrementa mentre ad ogni iterazione del secondo *while* interno l'indice j si decrementa. Il numero di iterazioni prima che si abbia $i < j$ è

$$\Theta(ultimo - primo) = \Theta(n)$$

dove n è il numero di elementi nel vettore $A[primo : ultimo + 1]$

```
>>> A=[6,7,5,1,2,8,9,4]
>>> partitiona(A,0,len(A)-1)
4
>>> print(A)
[4, 5, 1, 2, 6, 8, 9, 7]
```

L'array A viene tripartito, il pivot 6 finisce in posizione 4 preceduto dagli elementi più piccoli e seguito dagli elementi più grandi.

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

Per quanto riguarda la complessità temporale, continua a valere la seguente ricorrenza:

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

dove k, con $0 \leq k \leq n-1$, rappresenta la posizione corretta del pivot all'interno della lista ordinata. La cui soluzione è:

$$\Omega(n \log n) \leq T(n) \leq O(n^2).$$

Poiché le complessità temporali nei casi migliore e peggiore differiscono significativamente, nella studieremo ora il caso medio. Dimostreremo che la complessità temporale dell'algoritmo in questa situazione è pari a $\Theta((n \log n)$. Dato che il caso medio rappresenta lo scenario più comune, questo risultato conferma il quicksort come un algoritmo altamente efficiente nella maggior parte dei contesti pratici.

QuickSort nel caso medio

- Valutiamo ora il costo computazionale nel caso medio quando gli $n$elementi da ordinare sono tutti distinti.
- Lavoreremo sotto l'ipotesi che uno qualunque degli n numeri possa essere scelto come pivot. Possiamo fare quest'assunzione nel caso in cui il pivot venga scelto in modo equiprobabile tra gli elementi della lista.
- In questo caso si parla di quicksort randomizzato e per ottenere questo basterà nel codice visto prima far precedere l'istruzione $pivot = lista[primo]$ per la scelta del pivot al primo posto da queste semplici istruzioni di costo $\Theta(1)$

```
k = random.randint(primo,ultimo)
lista[primo], lista[k] = lista[k], lista[primo]
```

In questo modo dalla sottosequenza di dimensione n con uguale probabilità $\frac{1}{n}$ vengono generate da ordinare due sottosequenze di dimensione k e $n-1-k$ rispettivamente con $0 \leq k \leq n-1$.

Abbiamo dunque:

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

Da

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

notiamo che per ogni valore di q e $0 \leq q < n$ il termine $T(q)$ compare due volte nella sommatoria, la prima quando $k = q$ e la seconda quando $k = n - 1 - q$.
Possiamo dunque scrivere:

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

Questa è la ricorrenza da studiare per ottenere la complessità al caso medio. Dimostreremo che la soluzione a questa ricorrenza è $O(n \log n)$.

Per risolvere la ricorrenza utilizzeremo il **metodo di sostituzione**.
Per prima cosa eliminiamo i termini asintotici dall'equazione:

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

dove a e b sono costanti sconosciute.

Proponiamo come ipotesi induttiva:

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

dove c è una costante da determinare.

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

Proponiamo come ipotesi induttiva:

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

Casi base: per $n \leq 1$, assumendo $c \geq a$, abbiamo

- $T(0) = a \leq c \cdot 1 \log_2 2 = c$
- $T(1) = a \leq c \cdot 2 \log_2 3$

Passo induttivo:
Applichiamo ora l'ipotesi induttiva:

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

Per la sommatoria utiliziamo la seguente limitazione:

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

Sostituendo questa stima otteniamo:

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

Per garantire che l'ultima diseguaglianza sia soddisfatta, basta scegliere $c = 4\max\{a, b\}$. Infatti con questa scelta il termine $-\frac{cn}{4} + bn$ è non positivo e quindi:

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

Questo conclude la dimostrazione, mostrando che $T(n) = O(n \log n)$.

Per completare la dimostrazione in ogni dettaglio, resta solo da verificare la limitazione 1.
Per farlo è possibile utilizzare il metodo dell'integrale per approssimare superiormente la sommatoria:

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

Tuttavia proponiamo una dimostrazione che non usa l'integrazione. Limitiamo la sommatoria $\sum_{q=0}^{n-1}(q+1)\log_2(q+2)$ spezzandola in due nel punto $\left\lfloor \frac{n-2}{2} \right\rfloor$:

$$\sum_{q=0}^{n-1}(q+1)\log_2(q+2) = \sum_{q=0}^{\left\lfloor \frac{n-2}{2} \right\rfloor}(q+1)\log_2(q+2) + \sum_{q=\left\lfloor \frac{n-2}{2} \right\rfloor+1}^{n-1}(q+1)\log_2(q+2)$$

$$\leq \log_2\left(\frac{n-2}{2}+2\right)$$
$$\leq \log_2\left(\frac{n+2}{2}\right)$$
$$\leq \log_2(n+2)-1$$

$$\leq \log_2(n-1)$$
$$\leq \log_2(n+2)$$

Abbiamo dunque

$$\sum_{q=0}^{n-1} q \cdot \log_2 q \leq \sum_{q=0}^{\left\lfloor \frac{n-2}{2} \right\rfloor} (q+1) \cdot (\log_2(n+2) - 1) + \sum_{q=\left\lfloor \frac{n-2}{2} \right\rfloor + 1}^{n-1} (q+1) \cdot \log_2(n+2)$$

$$= \sum_{q=0}^{\left\lfloor \frac{n-2}{2} \right\rfloor} (q+1) \log_2(n+2) - \sum_{q=0}^{\left\lfloor \frac{n-2}{2} \right\rfloor} (q+1) + \sum_{q=\left\lfloor \frac{n-2}{2} \right\rfloor + 1}^{n-1} (q+1) \log_2(n+2)$$

$$= \sum_{q=0}^{n-1} (q+1) \log_2(n+2) - \sum_{q=0}^{\left\lfloor \frac{n-2}{2} \right\rfloor} (q+1)$$

$$= \sum_{i=1}^{n} i \log_2(n+2) - \sum_{i=1}^{\left\lfloor \frac{n-2}{2} \right\rfloor + 1} i$$

$$= \frac{n(n+1)}{2} \log_2(n+2) - \frac{(\left\lfloor \frac{n-2}{2} \right\rfloor + 1)(\left\lfloor \frac{n-2}{2} \right\rfloor + 2)}{2} \qquad (2)$$

Ora stimiamo il termine:

$$\begin{aligned}
\left(\left\lfloor\frac{n-2}{2}\right\rfloor+1\right)\left(\left\lfloor\frac{n-2}{2}\right\rfloor+2\right) &= \left\lfloor\frac{n-2}{2}\right\rfloor^2+3\left\lfloor\frac{n-2}{2}\right\rfloor+2 \\
&\geq \left(\frac{n-2}{2}-1\right)^2+3\left(\frac{n-2}{2}-1\right)+2 \\
&= \left(\frac{n}{2}-2\right)^2+3\left(\frac{n}{2}-2\right)+2 \\
&= \frac{n^2}{4}+\frac{n}{2} \\
&\geq \frac{n^2}{4} \qquad (3)
\end{aligned}$$

Infine, riprendendo dalla disequazione (2) e utilizzando la limitazione (3) otteniamo:

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

- Abbiamo appena dimostrato che nel caso medio si ha $T(n) = O(n \log n)$
- Inoltre in media non si può avere meglio dell'ottimo che sappiamo essere $\Theta(n \log n)$.

Deduciamo dunque che per il caso medio del quicksort vale la complessità $\Theta(n \log n)$.

Corso di laurea in Informatica
Introduzione agli Algoritmi

Esercizi per casa

SAPIENZA
UNIVERSITÀ DI ROMA

Esercizi

1. Scrivere la versione iterativa dell'algoritmo di quick sort se ne calcoli la complessità temporale e spaziale al caso pessimo ed al caso ottimo.

2. Si considerino i valori 0, 1, 2, 3, 4, 5, 6, 7. Si determini una permutazione di questi valori che generi il caso peggiore per l'algoritmo Quick sort.

3. Calcolare il costo computazionale del Quick sort nel caso in cui il vettore contenga tutti elementi uguali e poi nel caso in cui sia già ordinato da destra a sinistra.

Esercizi

Dimostrare che la seguente equazione di ricorrenza:

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

con $0 \leq k \leq n-1$, soddisfa la relazione:

$$\Omega(n \log n) \leq T(n) \leq O(n^2).$$

Suggerimento: utilizzare il metodo di sostituzione

- Si progetti un algoritmo il più efficiente possibile per i seguenti problemi:
  - Data una matrice $m \times n$, si vogliono rimescolare i suoi elementi in modo che tutti i vettori riga e tutti i vettori colonna siano ordinati in senso non decrescente.
  - Data una matrice $n \times n$, si vogliono rimescolare i suoi elementi in modo che tutti gli elementi posizionati al di sopra della diagonale principale siano minori o uguali di tutti gli elementi che giacciono sulla diagonale principale che, a loro volta, siano minori o uguali di tutti gli elementi posizionati al di sotto della diagonale principale.