Corso di laurea in Informatica
Introduzione agli Algoritmi

Il problema dell'ordinamento: Merge Sort e Quick Sort a confronto

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

```
def Merge_sort(A, primo, ultimo):
    if primo < ultimo:
        # localizza la posizione centrale
        medio=(primo + ultimo)//2
        Merge_sort(A, primo, medio) # Ordina A[primo: medio+1]
        Merge_sort(A, medio+1, ultimo) # Ordina A[medio+1: ultimo+1]
        fondi(A, primo, medio, ultimo) # Fonde le due parti ordinate
```

```
def Quick_Sort(A, primo, ultimo):
    if primo < ultimo:
        # posiziona il pivot nessua sua corretta posizione con alla sua
        # sinistra gli elementi minori e restituisce la posizione del pivot
        k = Partiziona(A, primo, ultimo)
        Quick_Sort(A, primo, k-1)  # Ordina la parte sinistra
        Quick_Sort(A, k+1, ultimo)  # Ordina la parte destra
```

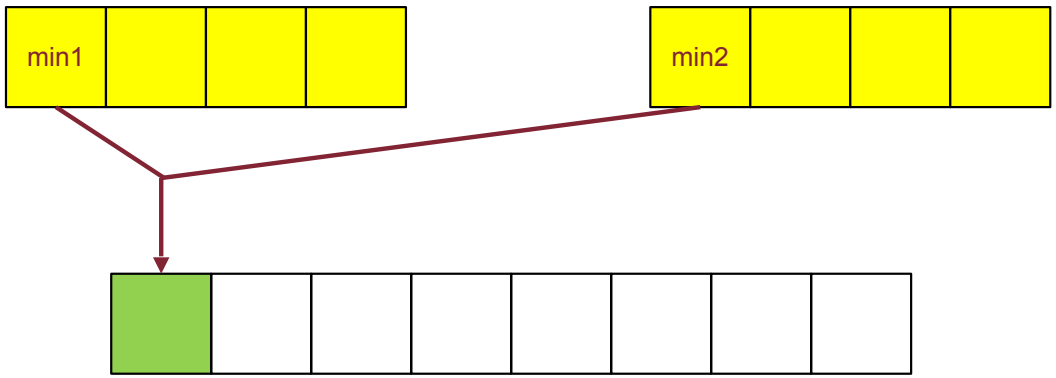
```
def Merge_sort(A, primo, ultimo):
    if primo < ultimo:
        medio=(primo + ultimo)//2
        Merge_sort(A, primo, medio)
        Merge_sort(A, medio+1, ultimo)
        fondi(A, primo, medio, ultimo)
```

Complessità:
Indicando con $S(n)$ il costo della fusione delle due sottoliste con dimensione complessiva n. Si ha la seguente equazione di ricorrenza:

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

Funzionamento della funzione fondi():

- la funzione sfrutta il fatto che le due sottosequenze sono ordinate;
- il minimo della sequenza complessiva non può che essere ***il più piccolo fra i minimi delle due sottosequenze*** (se essi sono uguali, scegliere l'uno o l'altro non fa differenza);
- dopo aver eliminato da una delle due sottosequenze tale minimo, la proprietà rimane: il prossimo minimo non può che essere il più piccolo fra i minimi delle due parti rimanenti delle due sottosequenze.



```python
def fondi(A, primo, medio, ultimo):
    i,j=primo, medio+1
    B=[]
    while i <= medio and j <= ultimo:
        if A[i] <= A[j]:
            B.append(A[i])
            i+=1
        else:
            B.append(A[j])
            j+=1
    while i <= medio:
        B.append(A[i])
        i+=1
    while j <= ultimo:
        B.append(A[j])
        j+=1
    for i in range(len(B)):
        A[i+primo] = B[i]
```

Fonde le due sottoliste ordinate $A[primo : medio + 1]$ e $A[medio + 1 : ultimo + 1]$ in un'unica sottolista lista ordinata $A[primo : ultimo + 1]$

Complessità di tempo: $\Theta(ultimo - primo) = \Theta(n)$
Complessità di spazio: $\Theta(ultimo - primo) = \Theta(n)$

Siamo ora pronti a valutare il costo temporale della funzione Merge_Sort:

```
def Merge_sort(A, primo, ultimo):
    if primo < ultimo:
        medio=(primo + ultimo)//2
        Merge_sort(A, primo, medio)
        Merge_sort(A, medio+1, ultimo)
        fondi(A, primo, medio, ultimo)
```

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

Risolvendo l'equazione di ricorrenza si ottiene:

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

```
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)
```

Complessità:
Indicando con $P(n)$ il costo della partizione della lista con n elementi si ha la seguente equazione di ricorrenza:

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

dove $0 \leq k \leq n-1$.

```python
def Partiziona(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
```

Nella lista $A[primo : ultimo + 1]$ posiziona l'elemento di pivot $A[primo]$ nella sua corretta locazione con alla sua sinistra gli elementi più piccoli ed alla sua destra gli elementi maggiori o uguali.

Complessità di tempo: $\Theta(ultimo - primo) = \Theta(n)$
Complessità di spazio: $\Theta(1)$

```python
def Partiziona(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
```

Nella lista $A[primo : ultimo + 1]$ posiziona l'elemento di pivot $A[primo]$ nella sua corretta locazione con alla sua sinistra gli elementi più piccoli ed alla sua destra gli elementi maggiori o uguali.

Complessità di tempo: $\Theta(ultimo - primo) = \Theta(n)$
Complessità di spazio: $\Theta(1)$

Siamo ora pronti a valutare il costo temporale della funzione Quick_Sort:

```
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)
```

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

Non è difficile dimostrare, ad esempio con il metodo della sostituzione che:

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

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

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:

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

- 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 da ordinare 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]
```

Dall'equazione di ricorrenza:

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

assumiamo dunque che, per una sequenza di lunghezza n, vengano generate due sottosequenze di lunghezza k e $n-1-k$, dove k è un intero tra 0 e $n-1$ scelto con uguale probabilità. La somma dei tempi per calcolare il caso medio può essere espressa come la media sui valori possibili di k, ovvero:

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

Notiamo ora che per ogni valore di k nella sommatoria, i termini $T(k)$ e $T(n-1-k)$ appaiono in modo simmetrico: $T(k)$ appare quando si considera la parte della sequenza di dimensione k e $T(n-1-k)$ appare per la parte della sequenza di dimensione $n-1-k$. Poiché i due termini compaiono con la stessa frequenza nella sommatoria, possiamo riscrivere la sommatoria come segue:

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

A questo punto, l'equazione risulta essere una ricorrenza che esprime il tempo medio in funzione delle dimensioni delle sottosequenze.
La soluzione di questa ricorrenza è $\Theta(n \log n)$, come si può dimostrare ad esempio utilizzando il **metodo di sostituzione**.