

Corso di laurea in Informatica

Introduzione agli Algoritmi

Il problema dell'ordinamento:
algoritmo Merge Sort

Angelo Monti



SAPIENZA
UNIVERSITÀ DI ROMA

Nella precedente lezione...

...abbiamo dimostrato il seguente:

Teorema. Il costo computazionale di qualunque algoritmo di ordinamento **basato su confronti** è $\Omega(n \log n)$.

Riusciamo a progettare degli algoritmi basati sul confronto che richiedono costo computazionale $O(n \log n)$ e sono, quindi, **ottimi**?

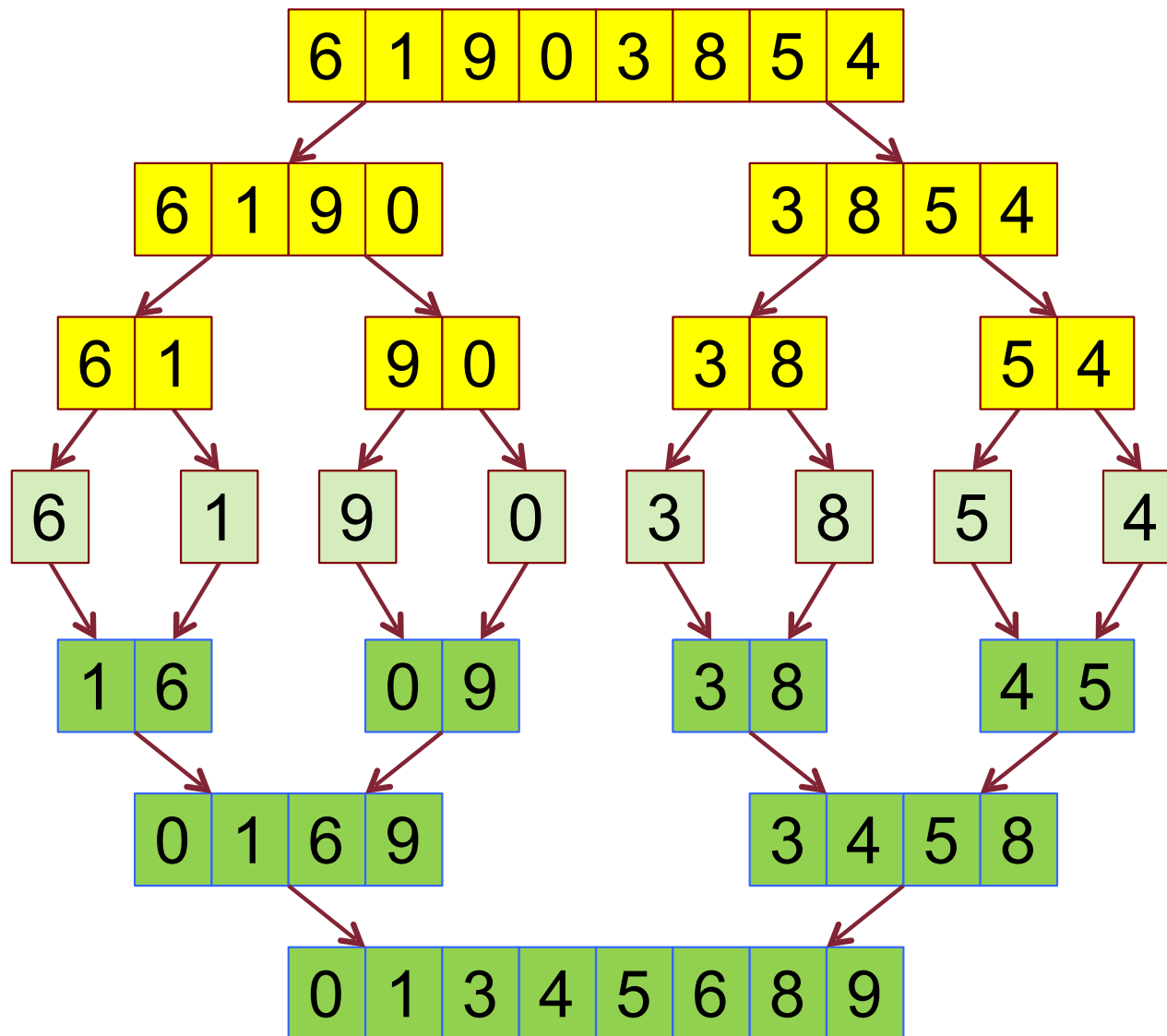
Merge Sort

L'algoritmo **merge sort** (**ordinamento per fusione**) è un algoritmo ricorsivo che adotta una tecnica algoritmica nota come **divide et impera** che può essere descritta come segue:

- il problema complessivo si suddivide in due sottoproblemi di dimensione inferiore (**divide**)
- i due sottoproblemi si risolvono ricorsivamente (**impera**)
- le soluzioni dei due sottoproblemi si ricompongono per ottenere la soluzione al problema originario (**combina**).

L'approccio dell'algoritmo Merge Sort è il seguente:

- **divide**: la sequenza di n elementi viene divisa in due sottosequenze di $\frac{n}{2}$ elementi ciascuna;
- **impera**: le due sottosequenze di $\frac{n}{2}$ elementi vengono ordinate ricorsivamente;
- **passo base**: la ricorsione termina quando la sottosequenza è costituita di un solo elemento, per cui è già ordinata;
- **combina**: le due sottosequenze – ormai ordinate – di $\frac{n}{2}$ elementi ciascuna, vengono “fuse” in un'unica sequenza ordinata di n elementi.



•**divide**: la sequenza di n elementi viene divisa in due sotto-sequenze di $n/2$ elementi ciascuna;

•**impera**: le due sotto-sequenze di $n/2$ elementi vengono ordinate ricorsivamente;

•**passo base**: la ricorsione termina quando la sotto-sequenza è costituita di un solo elemento, per cui è già ordinata;

•**combina**: le due sotto-sequenze – ormai ordinate – di $n/2$ elementi ciascuna vengono “fuse” in un’unica sequenza ordinata di n elementi.

Merge Sort

```
Merge_Sort(A, i, j):  
    '''ordina l'array A[i:j+1]'''  
    if i < j:  
        # A[i:j+1] ha più di un elemento  
        # mid è il punto medio per dividere l'array in due metà  
        mid=(i + j)//2  
        # ordina l'array A[i:m+1]  
        Merge_Sort(A, i, m)  
        # ordina l'array A[m+1:j+1]  
        Merge_Sort(A, m+1, j)  
        # fonda i due array ordinati A[i:m+1] e A[m+1,j+1]  
        fondi(A, i, m, j)
```

La chiamata iniziale sarà:

Merge_Sort(A, 0, len(A) - 1)

```

def Merge_Sort(A, i, j):
    if i < j:
        mid=(i + j)//2
        Merge_Sort(A, i, m)
        Merge_Sort(A, m+1, j)
        fondi(A, i, m, j)

```

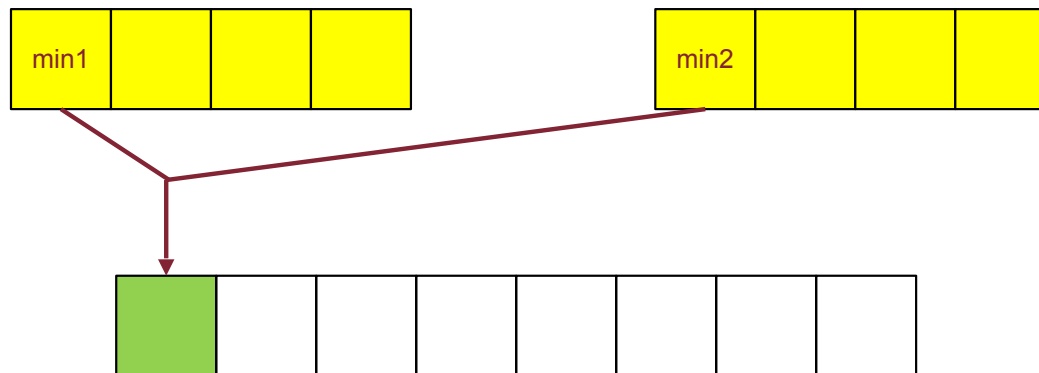
Complessità:

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.



Merge Sort

```
def fondi(A, i, m, j):  
    a, b = i, m+1  
    B = []  
    while a <= m and b <= j:  
        if A[a] <= A[b]:  
            B.append( A[a] )  
            a += 1  
        else:  
            B.append( A[b] )  
            b += 1  
    while a <= m: #la prima sottolista non è terminata  
        B.append( A[a] )  
        a+=1  
    while b <= j: #la seconda sottolista non è terminata  
        B.append( A[b] )  
        b+=1  
    for x in range( len(B) ): #ricopio in A gli elementi in B  
        A[ i+x ] = B[ x ]
```

Valutiamo il costo computazionale della funzione `Fondi()` su lista di n elementi:

- Inizializzazione delle variabili: $\Theta(1)$;
- Primo ciclo *while*:
 - ogni iterazione ha costo $\Theta(1)$ e incrementa di 1 l'indice i oppure l'indice j . Quindi il costo del *while* varia da un minimo di $n/2$ a un massimo di n , ossia è $\Theta(n)$.
- Secondo e terzo *while* (mai eseguiti entrambi):
 - si ricopia nel vettore B l'eventuale "coda" di una delle due sottosequenze: $O(n)$;
- Copia del vettore B nell'opportuna porzione del vettore A: $\Theta(n)$.

Dunque il costo $S(n)$ della funzione `Fondi()` è:

$$S(n) = \Theta(1) + \Theta(n) + O(n) + \Theta(n) = \Theta(n)$$

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

```
def Merge_Sort(A, i, j):  
    if i < j:  
        mid=(i + j)//2  
        Merge_Sort(A, i, m)  
        Merge_Sort(A, m+1, j)  
        fondi(A, i, m, j)
```

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

Risolvendo l'equazione di ricorrenza si ottiene:

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

OSSERVAZIONE

L'operazione di fusione non si può fare "in loco", cioè aggiornando direttamente il vettore A , senza incorrere in un aggravio del costo.

Infatti, in A bisognerebbe fare spazio via via al minimo successivo, ma questo costringerebbe a spostare di una posizione tutta la sottosequenza rimanente per ogni nuovo minimo, il che costerebbe $\Theta(n)$ per ciascun elemento da inserire, facendo lievitare quindi il costo computazionale della fusione da $\Theta(n)$ a $\Theta(n^2)$.

Ciò a sua volta risulterebbe nell'equazione di ricorrenza:

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

Che risolta darebbe:

$$T(n) = \Theta(n^2)$$

Analisi della Complessità spaziale Venendo alla complessità di spazio, abbiamo:

- Lo stack di chiamate cresce fino ad una profondità massima di $\log_2 n$
- Ogni chiamata nello stack richiede $O(1)$ spazio per le variabili locali, oltre allo spazio di due sottoliste temporanee.
- Il contributo complessivo delle variabili locali è quindi $O(\log n)$.
- Passiamo ora allo spazio totale richiesto dai vettori temporanei:
 - La prima chiamata (la radice della ricorsione) divide l'array in due metà, creando sottoliste di dimensione $\frac{n}{2}$ ciascuna, che occupano $O(n)$ in totale.
 - La seconda chiamata, a sua volta, divide una delle sottoliste in ulteriori due metà, creando due nuove sottoliste di dimensione $n/4$, che occupano $\frac{1}{2}O(n)$ in totale.
 - Questo processo continua fino alla base della ricorsione, dove i sottovettori sono di lunghezza 1.
 - Se consideriamo tutte le chiamate attive nello stack in un dato momento, la memoria occupata dalle sottoliste temporanee segue la seguente somma:

$$O(n) + \frac{1}{2}O(n) + \frac{1}{4}O(n) + \dots = O(n) \cdot \sum_{i=0}^{\log_2 n} \frac{1}{2^i} = O(n) \cdot O(1) = O(n)$$

Riassumendo, per quanto riguarda la complessità dell'algoritmo abbiamo:

- **Tempo:** $\Theta(n \log n)$
- **Spazio:** $\Theta(n)$ per le sottoliste temporanee utilizzate durante l'operazione di fusione.

L'algoritmo ha dunque lo svantaggio di non ordinare "in loco" e di richiedere memoria aggiuntiva $O(n)$.

Esercizio svolto

Esercizio. Nonostante *MergeSort* funzioni in tempo $\Theta(n \log n)$ mentre *InsertionSort* in $O(n^2)$, i fattori costanti sono tali che l'*InsertionSort* è più veloce del *MergeSort* per valori piccoli di n . Quindi, ha senso usare l'*InsertionSort* dentro il *MergeSort* quando i sottoproblemi diventano sufficientemente piccoli.

- Si consideri una modifica del *MergeSort* in cui il caso base si applica ad una porzione del vettore di lunghezza k , che viene ordinata usando *InsertionSort*.
- Le porzioni vengono combinate usando il meccanismo standard di fusione.
- Si determini il valore di k come funzione di n per cui l'algoritmo modificato ha lo stesso tempo di esecuzione asintotico del *MergeSort*.

Soluzione. Il codice che realizza tale versione è il seguente.

```
def Merge_Insertion(A, i, j, k):  
    dim = j - i + 1  
    if dim > k:  
        m = (i + j) // 2  
        Merge_Insertion (A, i, m, k)  
        Merge_Insertion (A, m + 1, j, k)  
        Fondi(A, i, m, j)  
    else Insertion_Sort(A, i, j)
```

La chiamata iniziale sarà:

Merge_Insertion(A, 0, len(A)-1, k)

L'equazione di ricorrenza che lo caratterizza è la seguente:

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

$$T(k) = \Theta(k^2)$$

Risolviamola col metodo iterativo:

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

$$= 2[2T(n/2^2) + \Theta(n/2^1)] + \Theta(n/2^0)$$

$$= 2[2[2T(n/2^3) + \Theta(n/2^2)] + \Theta(n/2^1)] + \Theta(n/2^0)$$

$$= 2^3T(n/2^3) + 2^2\Theta(n/2^2) + 2^1\Theta(n/2^1) + 2^0\Theta(n/2^0)$$

...

$$= 2^iT\left(\frac{n}{2^i}\right) + \sum_{j=0}^{i-1} 2^j\Theta\left(\frac{n}{2^j}\right)$$

$$= 2^iT\left(\frac{n}{2^i}\right) + \Theta(i \cdot n)$$

Ci fermiamo incontrando il caso base, il che succede quando $\frac{n}{2^i} = k$, ossia $2^i = \frac{n}{k} \implies i = \log_2 \frac{n}{k}$

Sostituendo tale valore nell'espressione precedente otteniamo:

$$\begin{aligned} T(n) &= 2^{\log_2 \frac{n}{k}} T\left(\frac{n}{2^{\log_2 \frac{n}{k}}}\right) + \Theta\left(n \log \frac{n}{k}\right) \\ &= \frac{n}{k} \Theta(k^2) + \Theta\left(n \log \frac{n}{k}\right) \\ &= \Theta(nk) + \Theta\left(n \log \frac{n}{k}\right) \\ &= \Theta(nk) + \Theta(n \log n - n \log k) \end{aligned}$$

Se $k = O(\log n)$ otteniamo:

$$T(n) = O(n \log n) + \Theta(n \log n - n \log \log n) = \Theta(n \log n)$$

L'idea di utilizzare un algoritmo derivato dal MergeSort e dall'InsertionSort è dovuta a Tim Peters che lo ha creato nel 2002. Quest'algoritmo prende il nome di **Timsort** ed è attualmente l'algoritmo di ordinamento standard del linguaggio di programmazione *Python* (a partire dalla versione 2.3) e *Rust*.

L'idea del TimSort è dunque la seguente:

1. Dividi l'array da ordinare in "piccoli" blocchi conosciuti come **Run**.
2. Ordina ciascuna delle Run tramite l'insertionSort
3. Fondi le run ordinate usando la funzione Fondi del MergeSort.

Corso di laurea in Informatica

Introduzione agli Algoritmi

Esercizi per casa



SAPIENZA
UNIVERSITÀ DI ROMA

ESERCIZI

- Scrivere la versione iterativa dell'algoritmo *Merge_Sort* e valutarne la complessità (temporale e spaziale)
- Scrivere la versione ricorsiva dell'algoritmo *Fondi*.
- Si supponga di scrivere una variante del *MergeSort*, chiamata *4MergeSort* che, invece di suddividere il vettore da ordinare in 2 parti (e ordinarle separatamente), lo suddivide in 4 parti, le ordina ognuna riapplicando *4MergeSort*, e le riunifica usando un'opportuna variante *4Fondi* di *Fondi* (che fa la fusione su 4 sottovettori invece che su 2).
 - Come cambia, se cambia, il costo computazionale di *4Merge sort* rispetto a quello di *Merge_Sort*?