# Divide et Impera

(locuzione latina, "dividi e comanda")

Una delle tecniche più note per lo sviluppo di algoritmi efficienti.

Principio: suddividere un problema in due o più sottoproblemi, risolvere i sottoproblemi, combinare le soluzioni dei sottoproblemi opportunamente così da ottenere una soluzione al problema.

I sottoproblemi possono essere risolti applicando lo stesso principio - suddivisione in sotto-sottoproblemi - fino a giungere a "problemi elementari" che possono essere risolti direttamente.

Nota: i sottoproblemi sono istanze dello stesso problema originale ma di dimensioni ridotte.

Elementare significa "risolvibile direttamente".

Formalmente il paradigma divide et impera può essere strutturato in tre passi:

Decomposizione : identifica un numero piccolo di sottoproblemi dello stesso tipo, ciascuno definito su un insieme di dati più piccolo.

Ricorsione : risolvi ricorsivamente ciascun sottoproblema fino a ottenere insiemi di dati di dimensioni tali da essere risolvibili direttamente.

Ricombinazione : combina le soluzioni dei sottoproblemi per fornire una soluzione al problema di partenza.

Esempio : MergeSort (algoritmo di ordinamento per "fusione")
Opera in tempo $O(n \log n)$

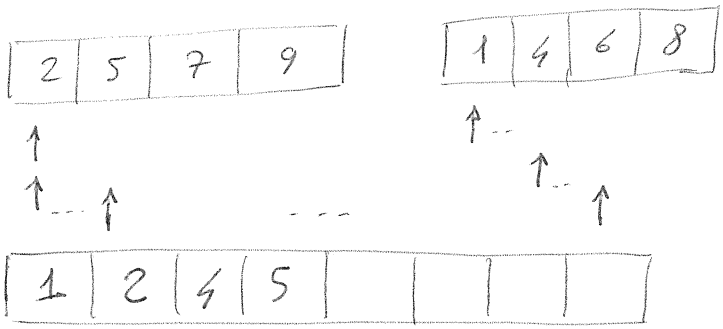
Decomposizione : se la sequenza ha almeno due elementi, dividila in due sottosequenze uguali (o quasi) in lunghezza

Ricorsione : ordina ricorsivamente le sottosequenze

Ricombinazione : fondi le due sottosequenze ordinate in un'unica sequenza ordinata

Osservazioni: per scrivere codice più semplice, nell'implementazione useremo memoria aggiuntiva per effettuare la fusione (esiste una implementazione IN LOCO ma è complessa)

Idea: Date le due metà ordinate, cominciando dall'inizio delle due metà, ad ogni passo mettiamo nella sequenza ordinata fusa, il più piccolo tra i due elementi che stiamo confrontando



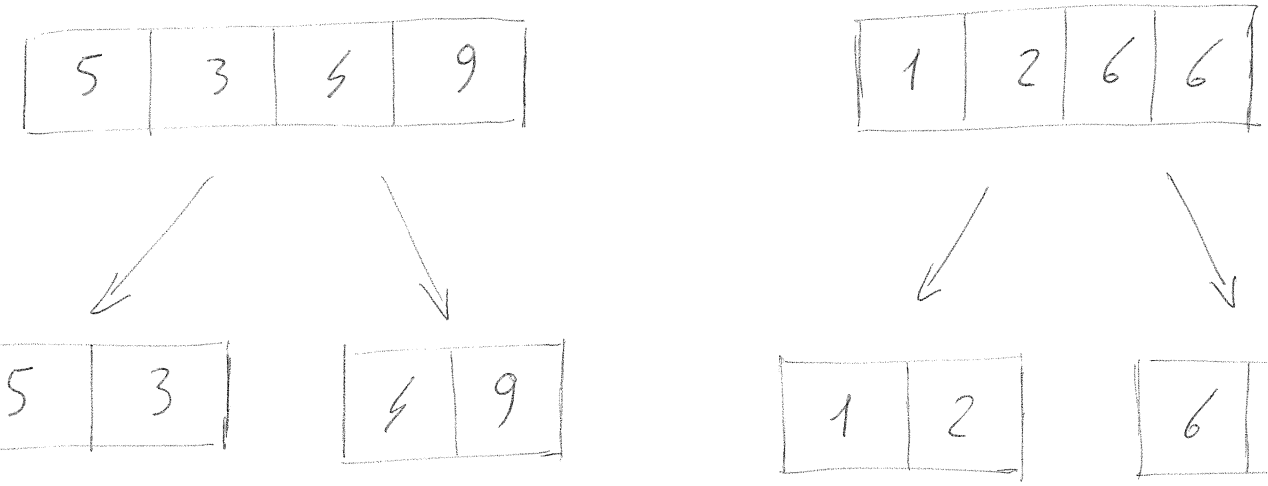

Complessità (tempo): È facile vedere che la procedura di fusione scorre le due metà inserendo un elemento ad ogni passo nell'array di appoggio.

Pertanto, se le due metà contengono $n$ elementi complessivamente, il costo della procedura è $\Theta(n)$.

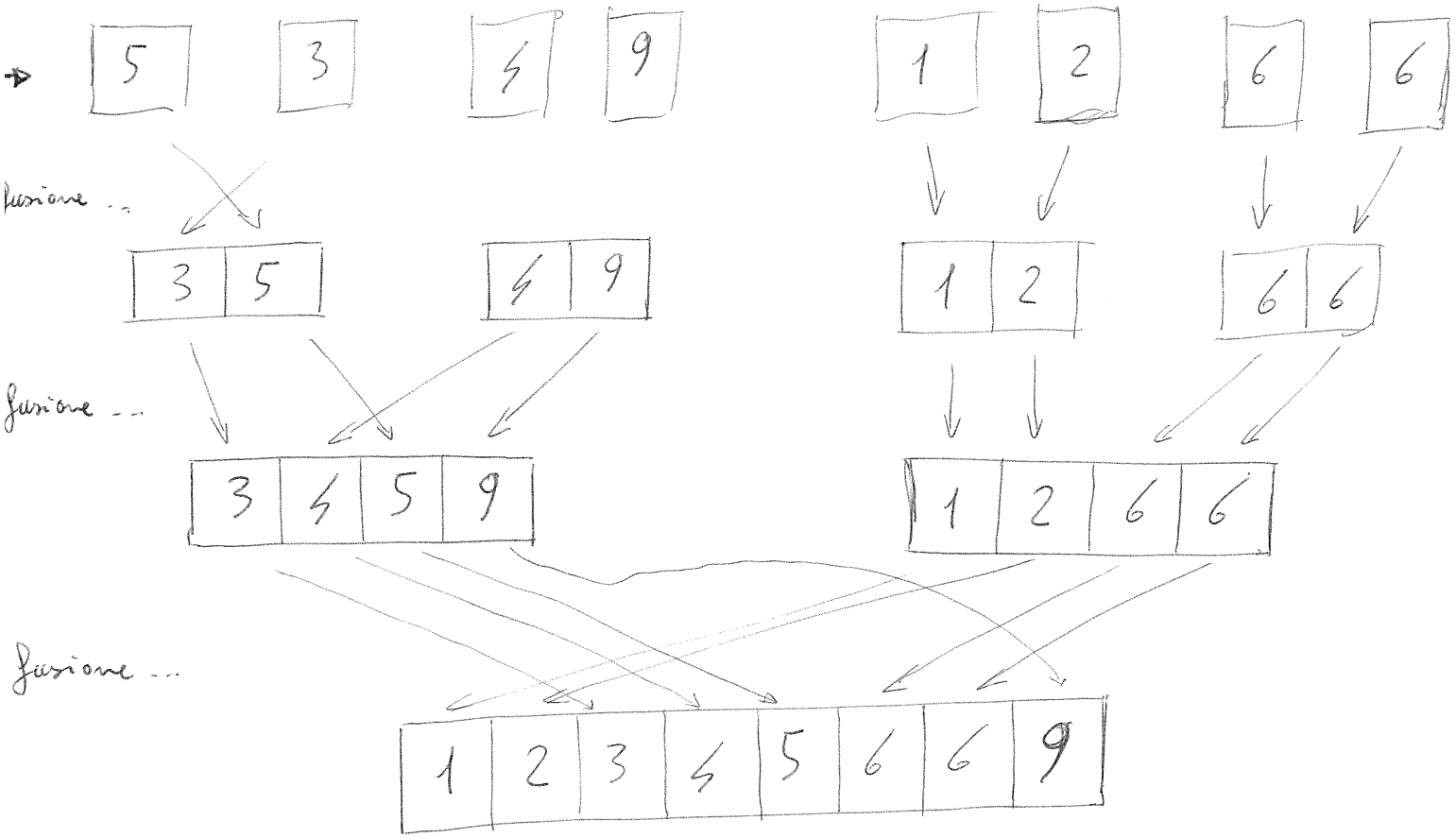
Spazio: Nell'algoritmo MergeSort useremo lo stesso array b per tutte le operazioni di fusione.

Esempio di esecuzione




Problemi elementari: array di 1 elemento — già ordinati!



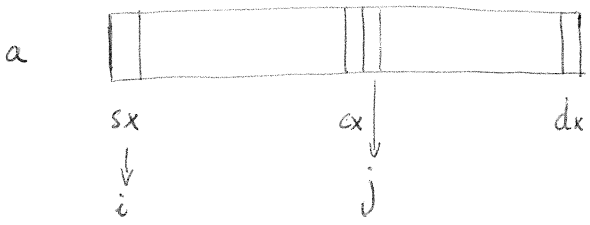

La fusione mette assieme due metà ordinate in un'unica sequenza di lunghezza doppia ordinata.

In codice ...

```
MergeSort (a, sinistra, destra):
  IF (sinistra < destra) {
    centro = (sinistra + destra)/2;
    MergeSort (a, sinistra, centro);
    MergeSort (a, centro+1, destra);
    Fusione (a, sinistra, centro, destra);
  }
```

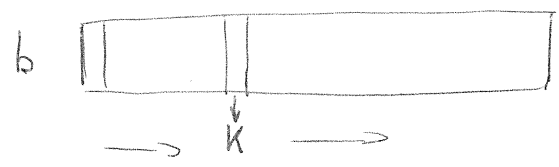
---

```
Fusione (a, sx, cx, dx):
  i = sx;
  j = cx+1;
  k = 0;
  WHILE ((i ≤ cx) && (j ≤ dx)) {
    IF (a[i] ≤ a[j]) {
      b[k] = a[i];
      i = i+1;
    } ELSE {
      b[k] = a[j];
      j = j+1;
    }
    k = k+1;
  }
```




... fino a quando i o j non raggiunge la propria estremità ...

... metti in b[k] il più piccolo e avanza il corrispondente indice ...




... raggiunta una delle estremità ... ⇒

```
FOR ( ; i <= cx ; i = i+1, k = k+1)
    b[k] = a[i];
```

se j è arrivato alla fine ... copia in b gli a[i]...

```
FOR ( ; j <= dx ; j = j+1, k = k+1)
    b[k] = a[j];
```

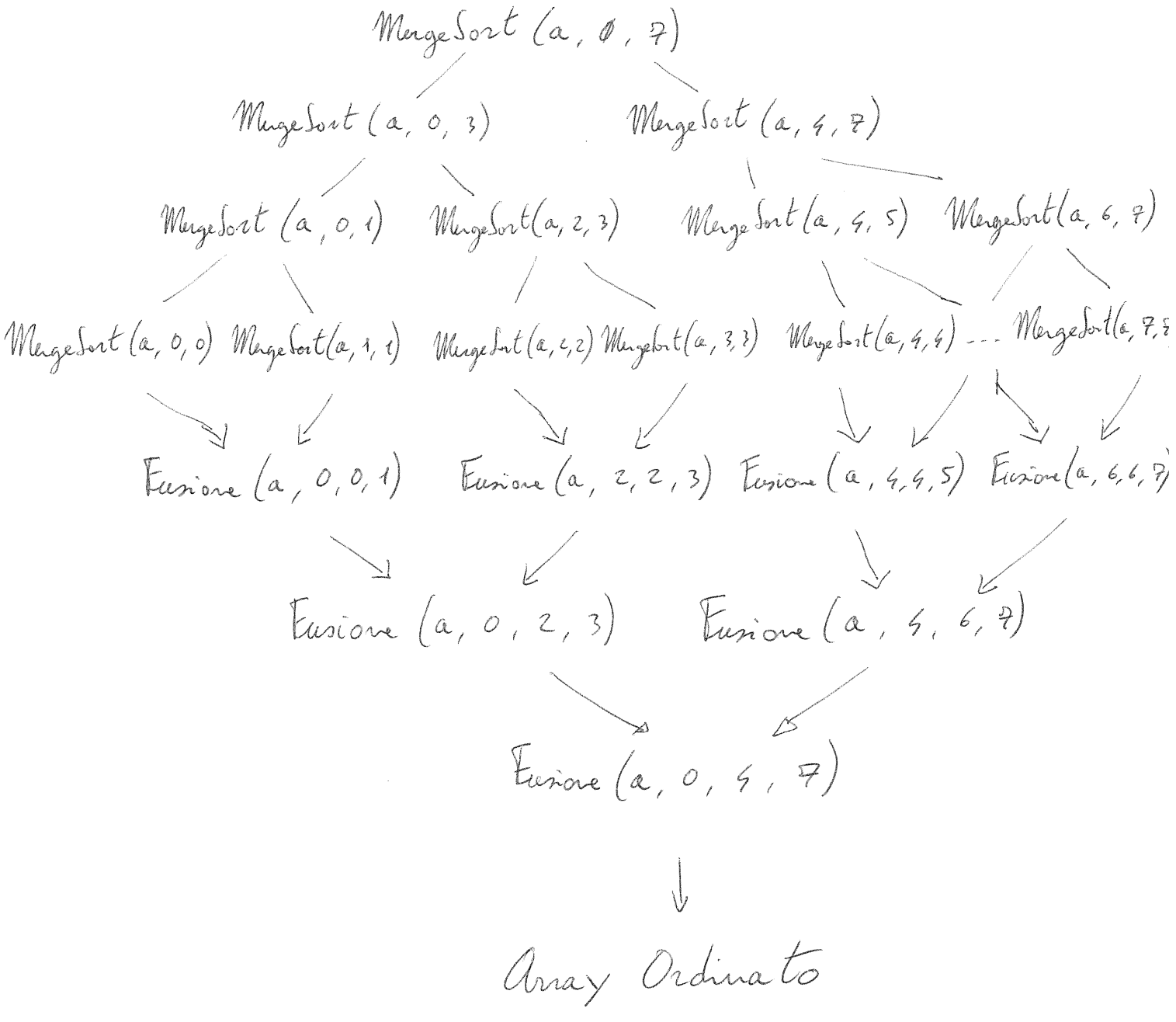
se i è arrivato alla fine ... copia in b gli a[j]

```
FOR (i = sx ; i <= dx ; i = i+1)
    a[i] = b[i - sx];
```

ricopia in a gli elementi ordinati in b.

---

Relativamente all'esempio ...



# Analisi della complessità

Fusione : richiede un certo numero di operazioni di costo $O(1)$ più le operazioni per la fusione vera e propria che costano $O(n)$.

Manca il costo delle chiamate ricorsive...

Possiamo esprimere formalmente questo costo usando una Relazione di Ricorrenza. Indicando con $T(n)$ il costo della funzione MergeSort su un input di dimensione $n$, risulta:

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq 1 \\ 2\,T(n/2) + c \cdot n & \text{altrimenti} \end{cases}$$

$c_0$ ↙ costo di soluzione del caso elementare

$2\,T(n/2)$ ↓ costo della fusione delle due metà di $n/2$ già ordinate

dove $c_0$ e $c$ sono costanti positive.

La Relazione è risolvibile direttamente esplicitando via via i termini ma appartiene ad una classe più generale di Relazioni per cui si conoscono le soluzioni. Nel nostro caso vale:

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

Esercizio. Risolvere direttamente la relazione di ricorrenza

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq 1 \\ 2T(n/2) + cn & \text{altrimenti} \end{cases}$$

supponendo, per semplicità, che $n$ sia una potenza di 2.

Soluzione. $T(n) \leq 2T(n/2) + c \cdot n$

$$T(n/2) \leq 2T(n/4) + c \cdot n/2$$

$$T(n/4) \leq 2T(n/8) + c \cdot n/4$$

Pertanto, possiamo scrivere:

$$T(n) \leq 2T(n/2) + cn \leq 2\left(2T(n/4) + c \cdot n/2\right) + cn = 4T(n/4) + cn + cn$$

$$\leq 4\left(2T(n/8) + c \cdot n/4\right) + cn + cn$$

$$= 8T(n/8) + c \cdot n + cn + cn$$

Osservando che $T(n/8)$ può ancora essere espresso in termini di $T(n/16)$, e possiamo iterare ... ci rendiamo conto che $T(n/2^i) = T(1)$ quando $n/2^i = 1 \iff i = \log n$. Pertanto:

$$T(n) = 2^{\log_2 n} \cdot T(1) + \underbrace{cn + \dots + cn}_{\log n - 1 \text{ volte}}$$

$$T(n) = 2^{\log n} T(1) + \underbrace{c \cdot n + \dots + cn}_{\log n - 1 \text{ volte}}$$

$\downarrow$ $n$ $\quad$ $\downarrow$ $c_0$

$$\leq c_0 \cdot n + \underbrace{c \cdot n + \dots + cn}_{\log n - 1 \text{ volte}}$$

$$\leq \sum_{j=1}^{\log n} c' \cdot n = c' \cdot n \cdot \log n = O(n \log n)$$

$\downarrow$

$\max\{c_0, c\}$

Come vedrete in altri corsi di algoritmi, esistono tre metodi per risolvere relazioni di ricorrenza:

- Master Theorem (che studieremo tra breve...)
- Metodo "per sostituzione"
  - consiste "nell'indovinare" la soluzione e provare che è corretta per induzione
- Metodo iterativo
  - espande la ricorrenza e la esprime come somma di termini che dipendono solo da $n$ e dalle condizioni iniziali.

L'approccio che abbiamo applicato in questo esempio è il terzo.