# Divide et impera

Luca Becchetti

Presentazione tratta dalle slide che accompagnano il testo Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

# Analisi (alternativa) del Merge Sort

- Sia T(n) il costo (nel caso peggiore) per ordinare un array di dimensione n

$$T(n) \leq \begin{cases} b, \; se\; n \leq 1 \\ 2T(n/2) + cn, \; se\; n > 1 \end{cases}$$

- Consideriamo due passi della ricorsione

$$T(n) \leq 2T(n/2) + cn \leq 2^2 T(n/2^2) + 2(cn/2) + cn = 2^2 T(n/2^2) + 2cn$$

- Dopo i passi

$$T(n) \leq 2^i T(n/2^i) + icn$$

# Analisi del Merge Sort (cont.)

- **Quando fermarsi?**
  - Argomento (n) della funzione $\leq 1$
  - Quindi: $n/2^i \leq 1 \rightarrow$ soddisfatta se $i \geq \log_2 n$
- **Quindi $\rightarrow T(n) \leq 2^{\log n}\, T(1) + cn\log_2 n \leq bn + cn\log_2 n = O(n\log n)$**
- **Abbiamo risolto una (dis)equazione di ricorrenza $\rightarrow T(n) \leq 2T(n/2) + cn$**
- ***Esercizio*: scrivere l'equazione di ricorrenza che dà il costo della ricerca binaria in un array ordinato e risolverla**

# Divide et impera senza ricorsione

- **Per n abbastanza grande, si consideri il seguente algoritmo di ordinamento**

```
SplitAndMerge(array a, k)
    b = []
    for i = 0 to k-1
        temp = sort(a[in/k … (i+1)n/k – 1]) // quadratico
        b = merge(b, temp)
    return b
```

- **Supponiamo n divisibile per k per semplicità**

# Analisi di MergeAndSplit

- **Sia T(n) il costo nel caso peggiore**

# Merge Sort

**Algorithm** *mergeSort*($S$)

**Input** sequence $S$ with $n$ elements

**Output** sequence $S$ sorted according to $C$

**if** $S.size() > 1$

$(S_1, S_2) \leftarrow partition(S, n/2)$

$mergeSort(S_1)$

$mergeSort(S_2)$

$S \leftarrow merge(S_1, S_2)$

Divide

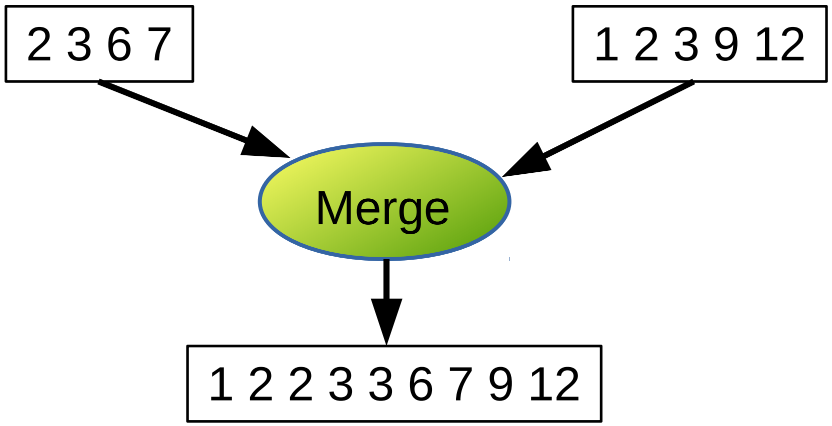
Ricorsione

Impera

# Merge

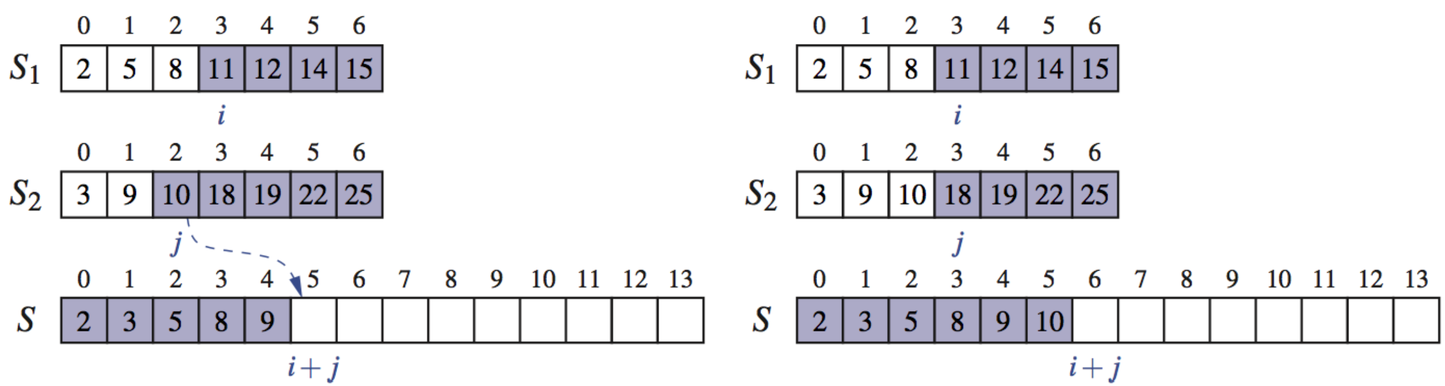
**Date due liste ordinate con $n_1$ e $n_2$ elementi → lista ordinata di $n_1 + n_2$ elementi**

- **Operazione fondamentale**
- **Sue varianti usate nei motori di ricerca**



# Merge → algoritmo (in Java)

```java
/** Merge contents of arrays S1 and S2 into properly sized array S. */
public static <K> void merge(K[ ] S1, K[ ] S2, K[ ] S, Comparator<K> comp) {
  int i = 0, j = 0;
  while (i + j < S.length) {
    if (j == S2.length || (i < S1.length && comp.compare(S1[i], S2[j]) < 0))
      S[i+j] = S1[i++];                    // copy ith element of S1 and increment i
    else
      S[i+j] = S2[j++];                    // copy jth element of S2 and increment j
  }
}
```



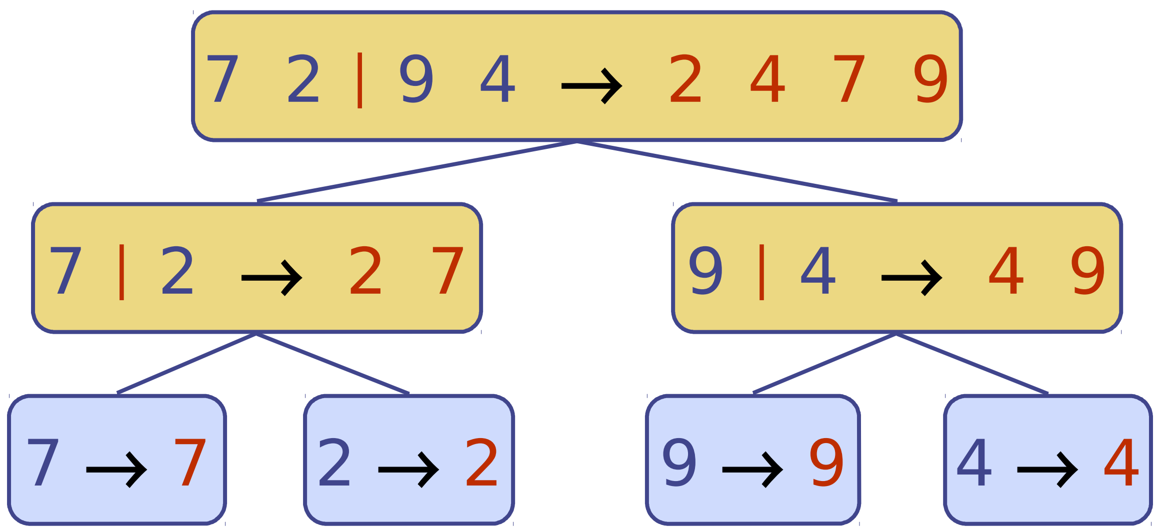
# Algoritmo Merge Sort (in Java)

```
/** Merge-sort contents of array S. */
public static <K> void mergeSort(K[ ] S, Comparator<K> comp) {
  int n = S.length;
  if (n < 2) return;                                    // array is trivially sorted
  // divide
  int mid = n/2;
  K[ ] S1 = Arrays.copyOfRange(S, 0, mid);              // copy of first half
  K[ ] S2 = Arrays.copyOfRange(S, mid, n);              // copy of second half
  // conquer (with recursion)
  mergeSort(S1, comp);                                  // sort copy of first half
  mergeSort(S2, comp);                                  // sort copy of second half
  // merge results
  merge(S1, S2, S, comp);                  // merge sorted halves back into original
}
```

# Analisi dell’algoritmo Merge Sort

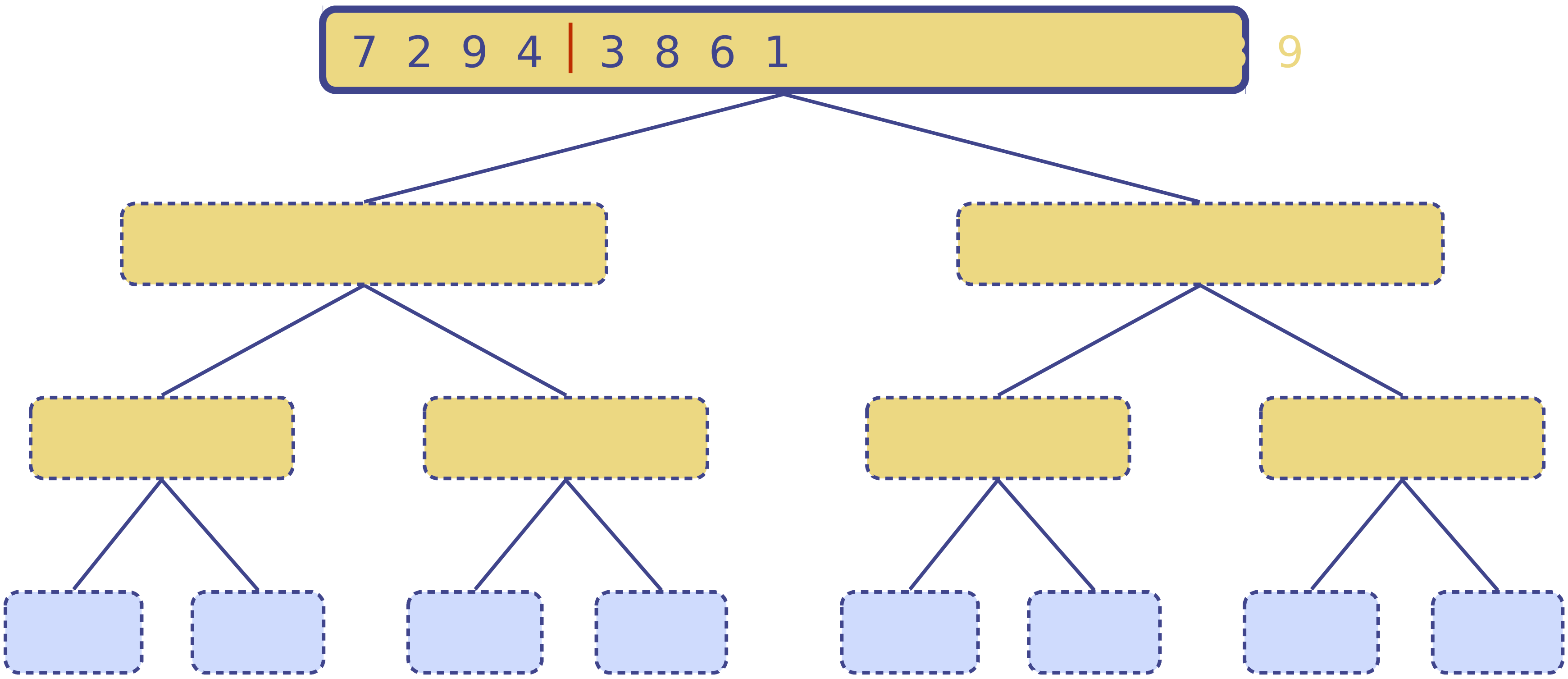
# Albero di ricorsione (merge sort tree)

- **Esecuzione descritta da un albero binario**
- **Ogni nodo rappresenta una chiamata ricorsiva del merge sort**
  - La radice rappresenta l'invocazione iniziale
  - Foglie → istanze di dimensione 1 (o 0)



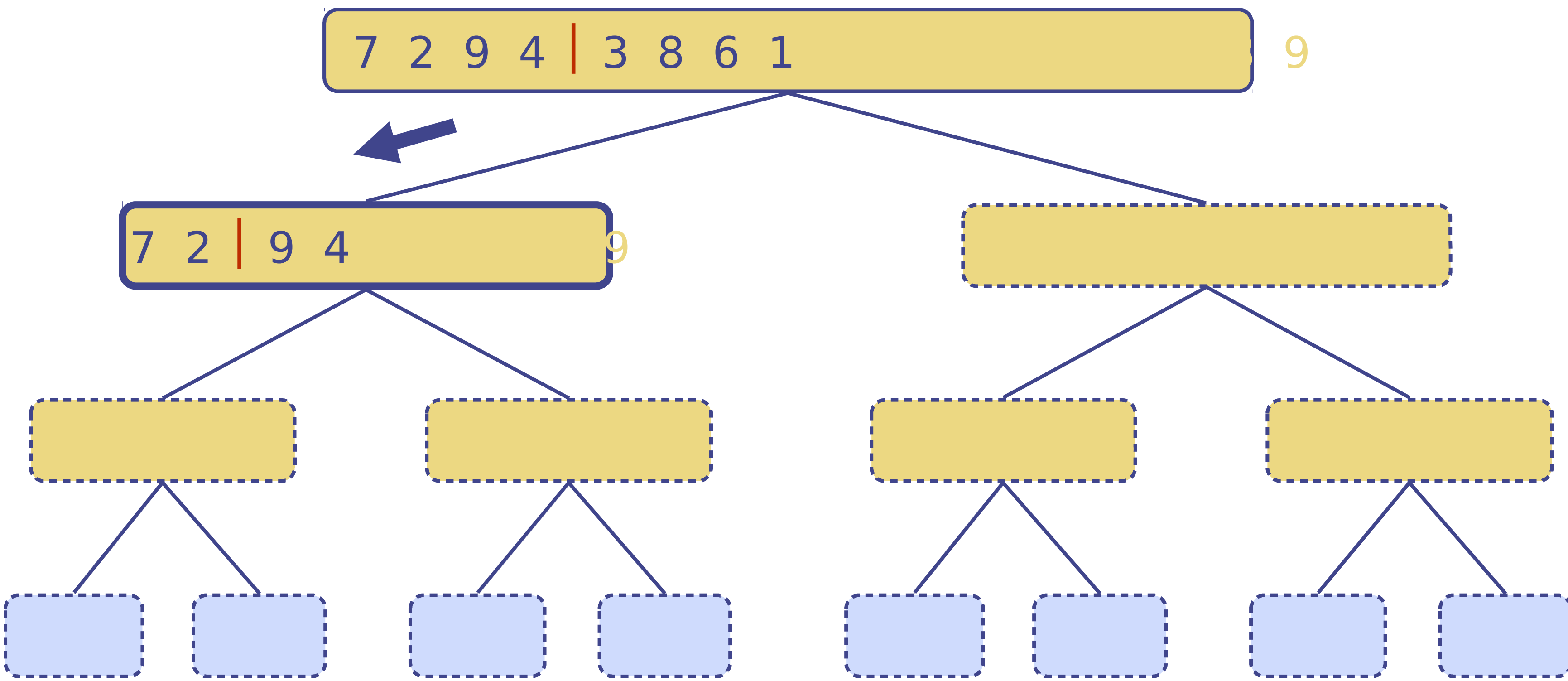
# Esempio di esecuzione

- **Partizione**



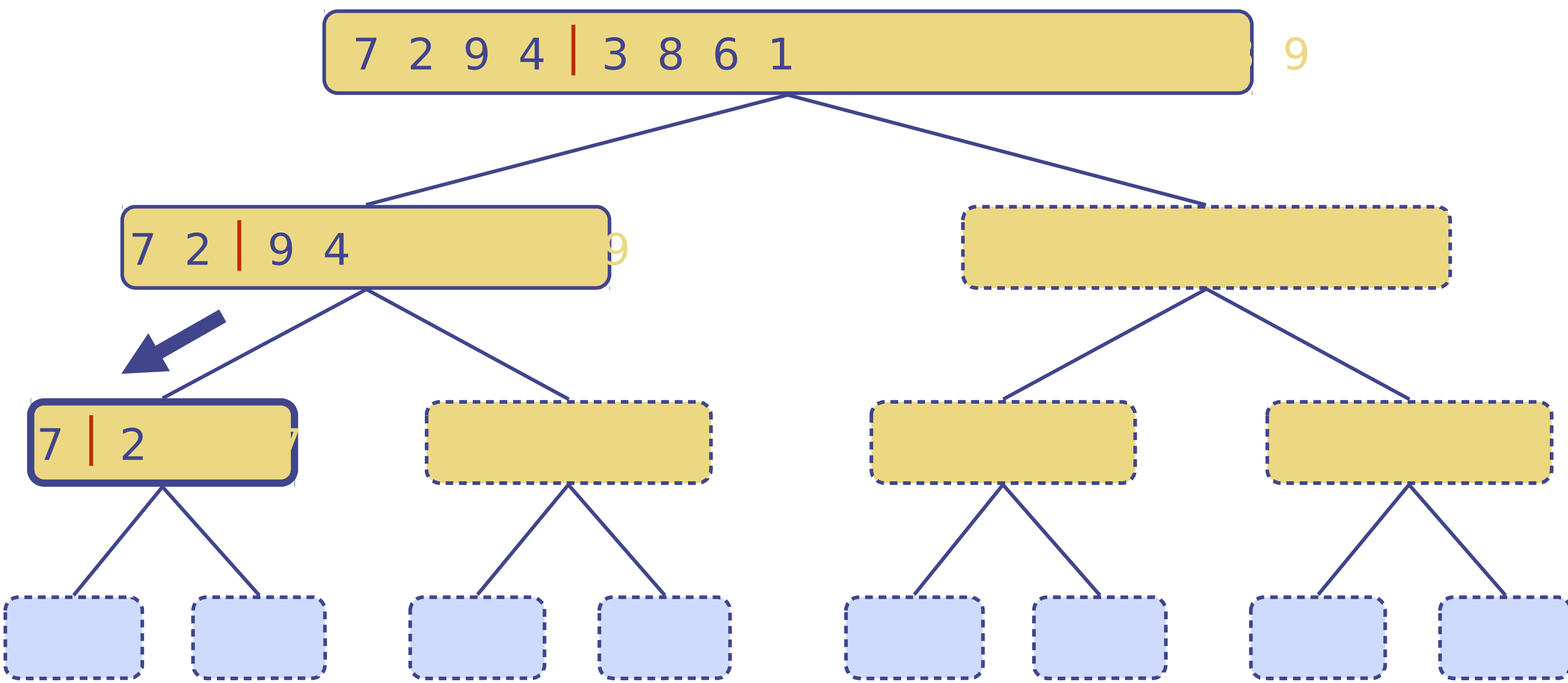
# Esempio di esecuzione

- **Chiamate ricorsiva, partizione**



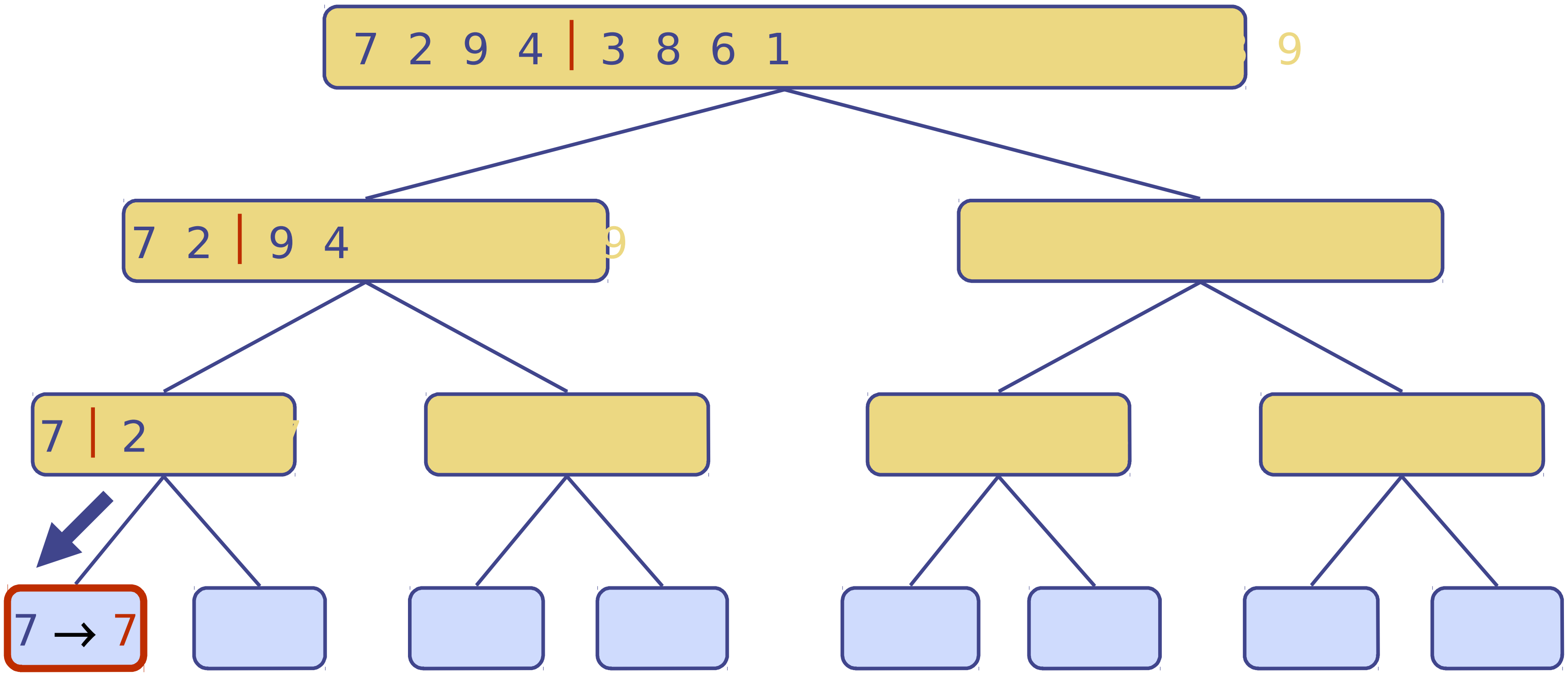
# Esempio di esecuzione

- **Chiamate ricorsiva, partizione**



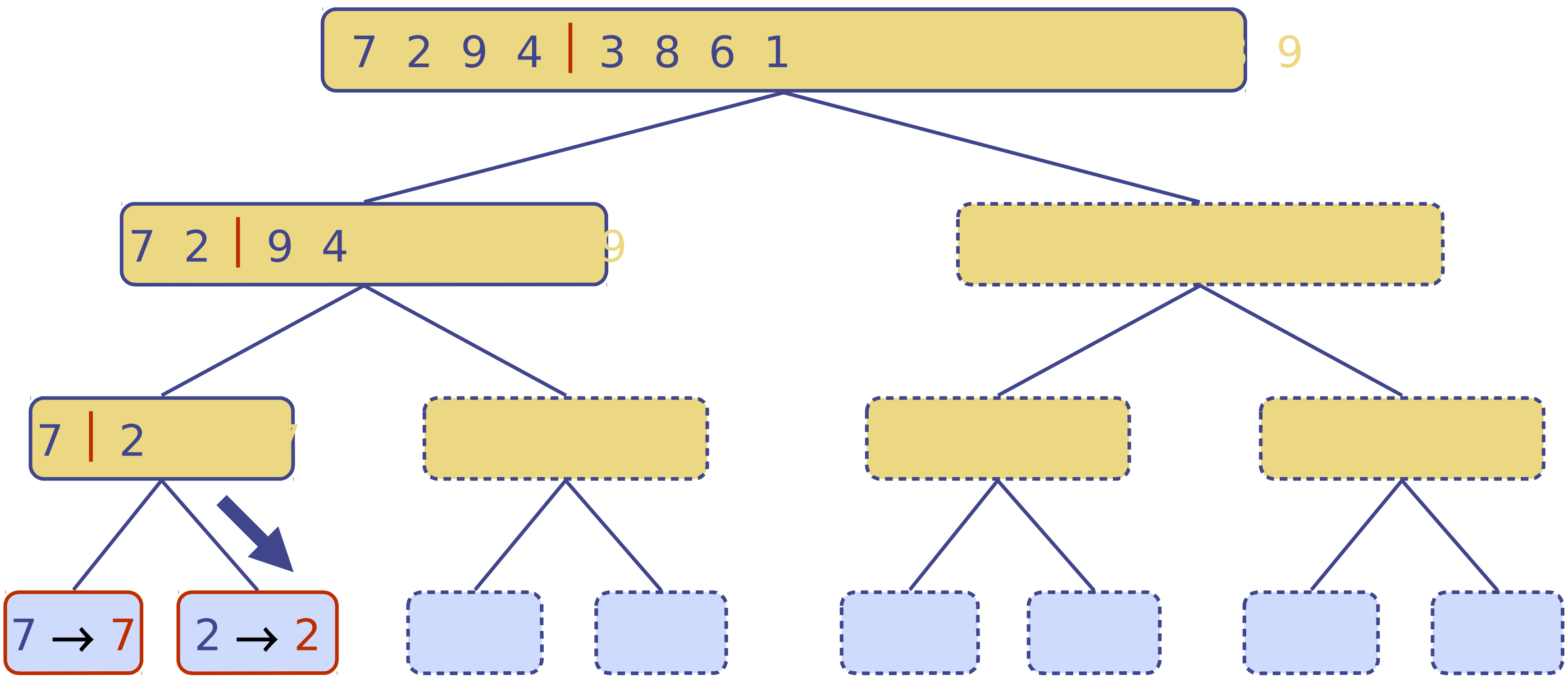
# Esempio di esecuzione

- **Chiamate ricorsiva, *caso base***



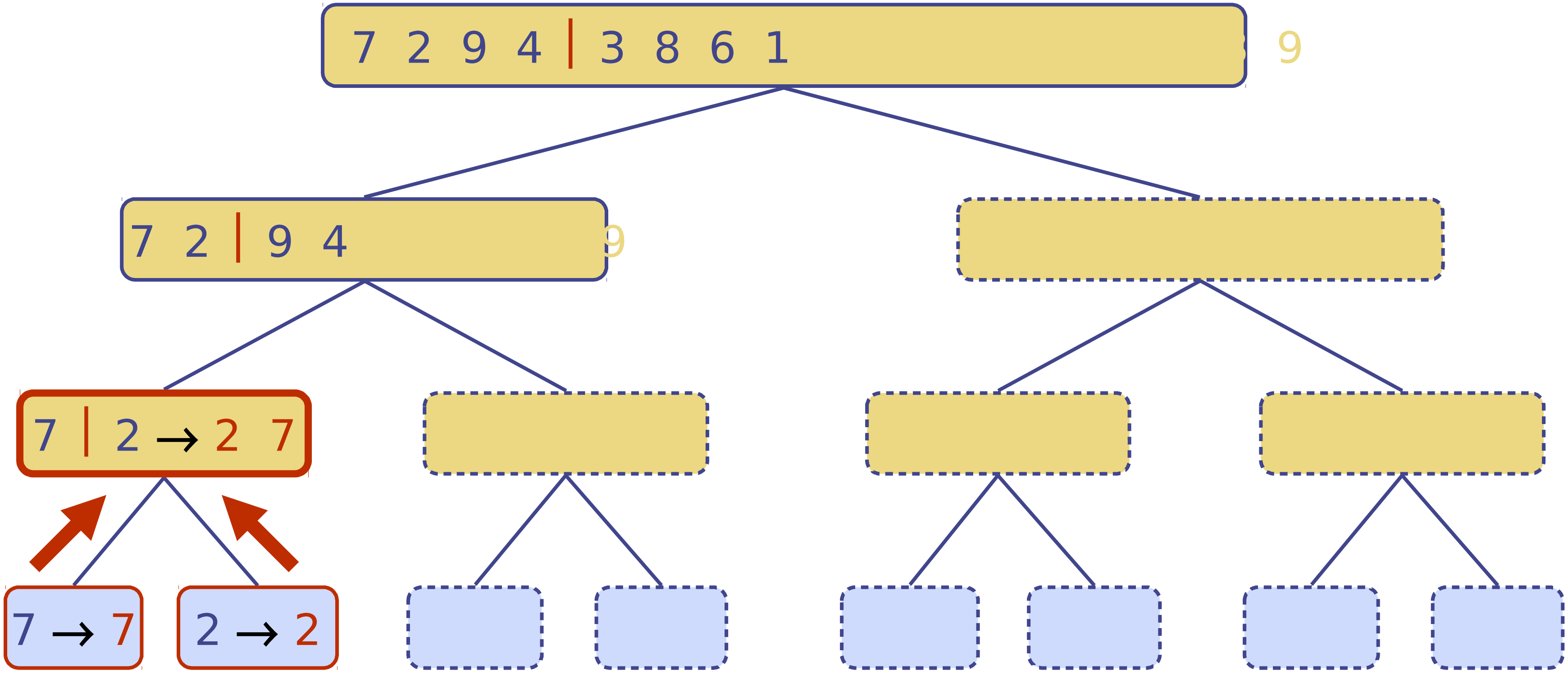
# Esempio di esecuzione

- **Chiamate ricorsiva, *caso base***



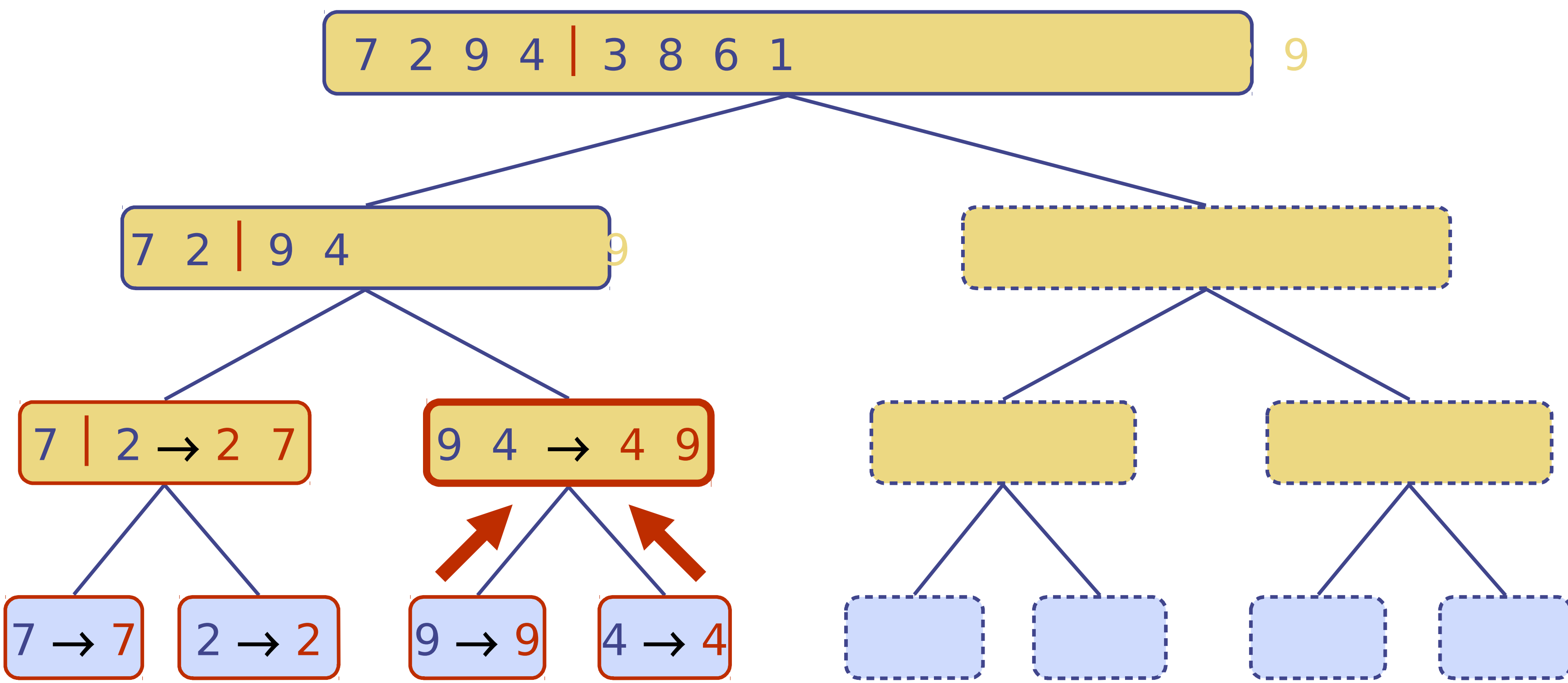
# Esempio di esecuzione

- **Merge**



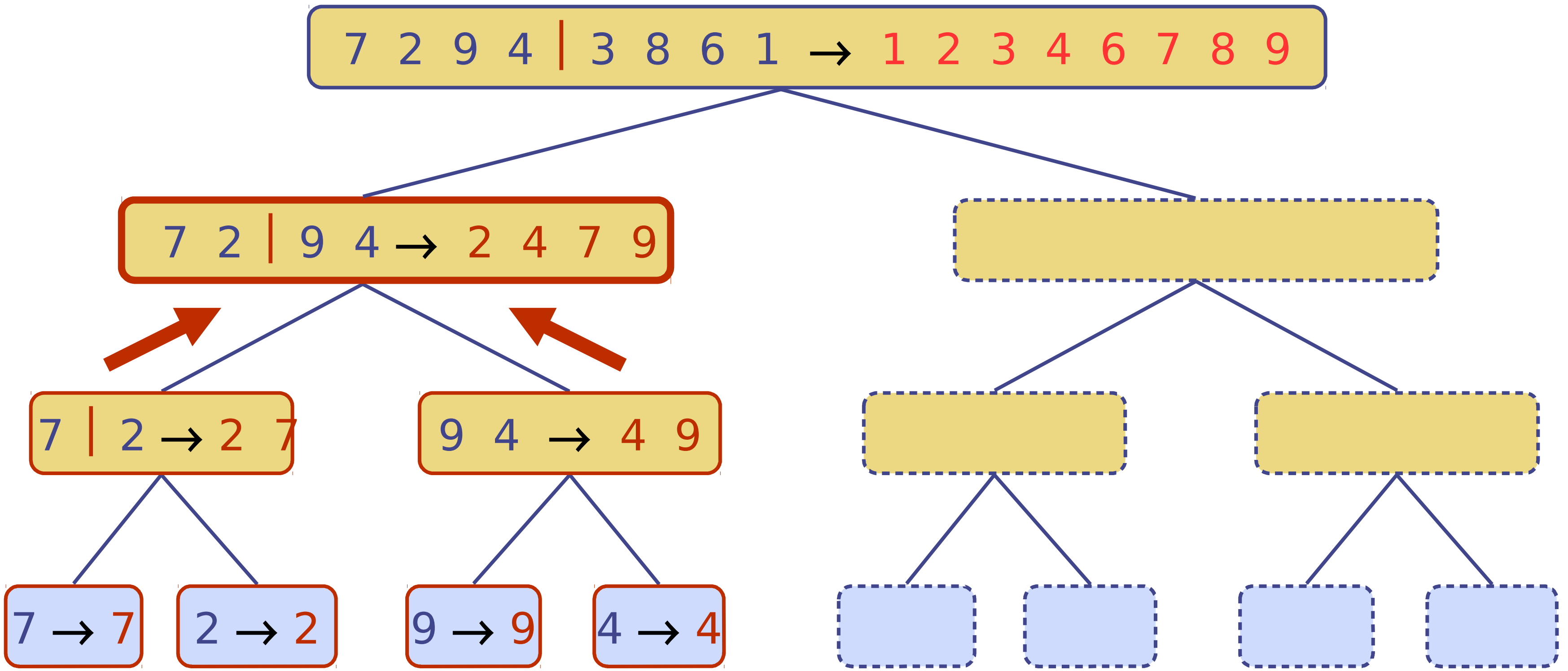
# Esempio di esecuzione

- **Chiamata ricorsiva, caso base, merge ...**



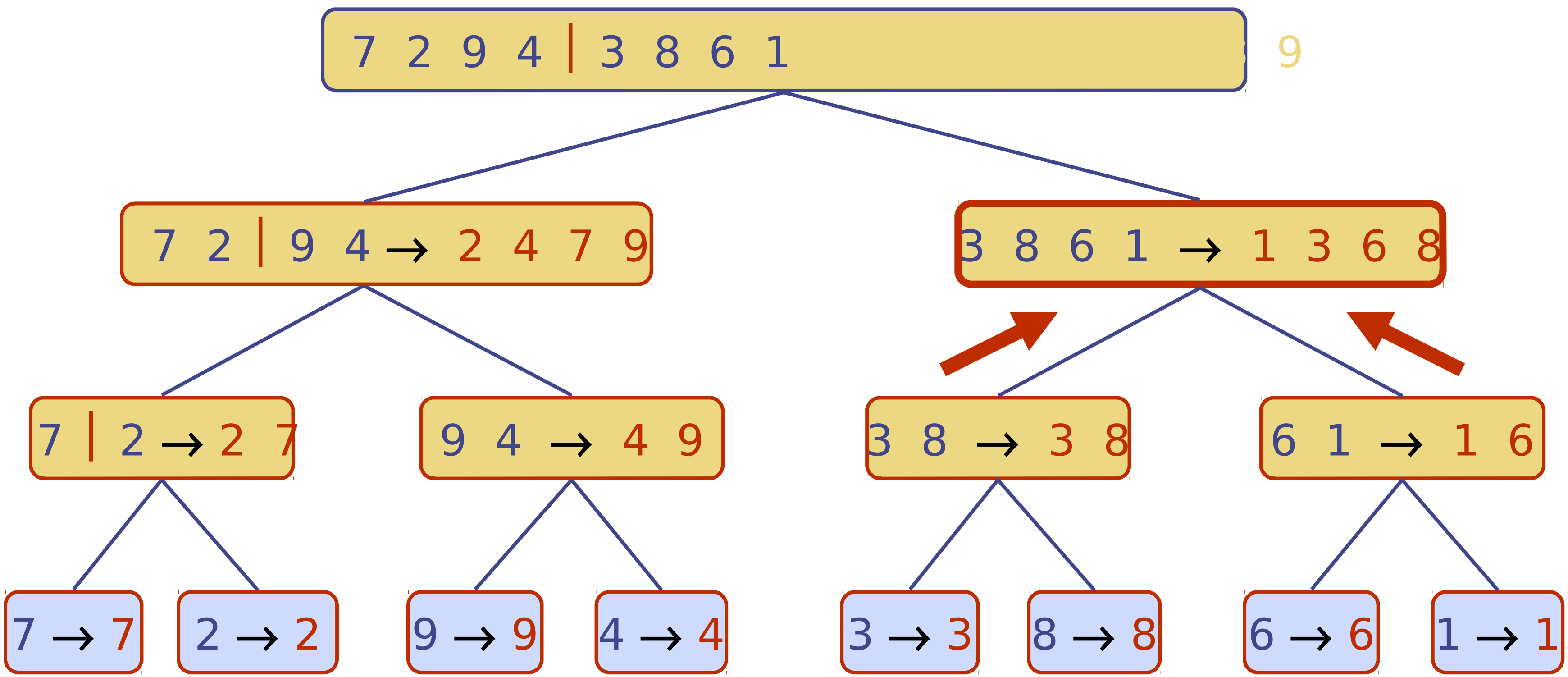
# Esempio di esecuzione

- **Merge**



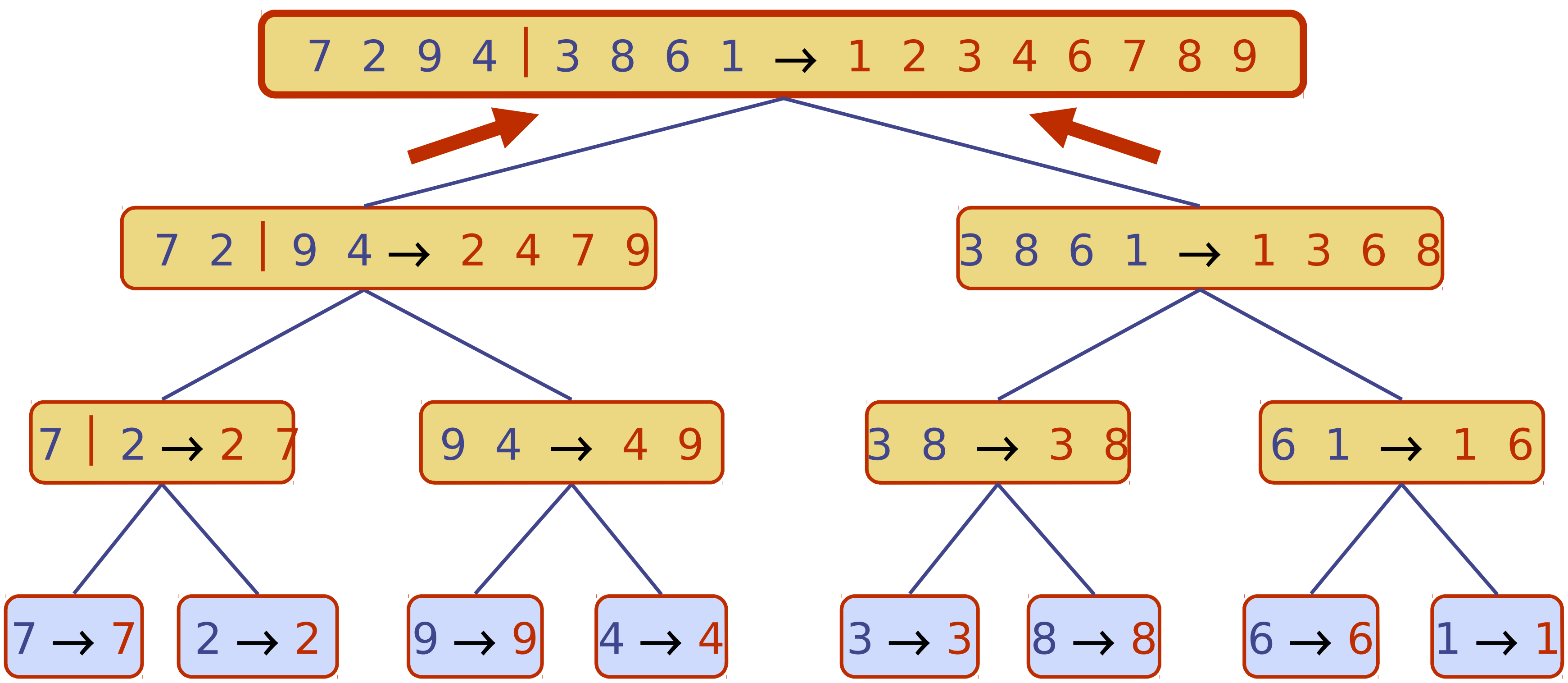
# Esempio di esecuzione

- **Chiamata ricorsiva, merge, merge …**



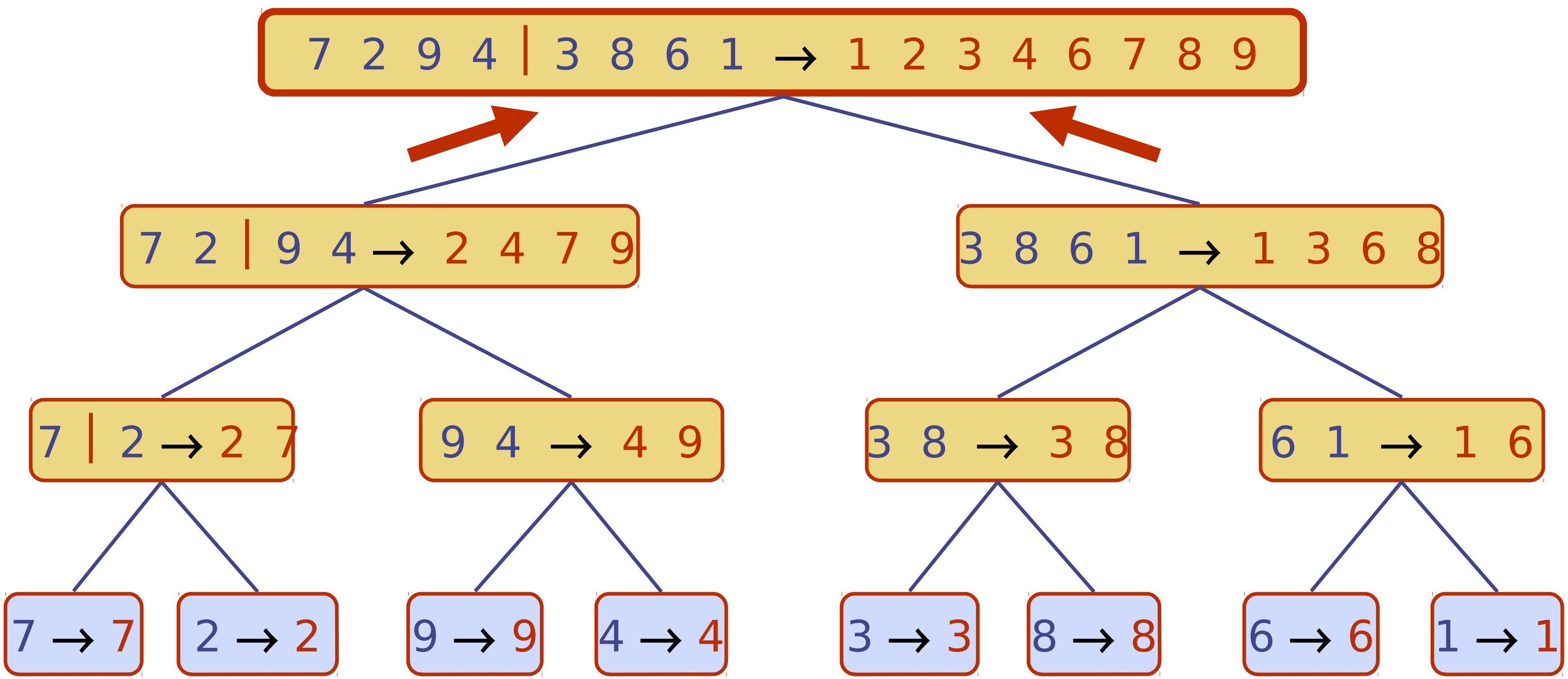
# Esempio di esecuzione

- **Merge**



# Analisi

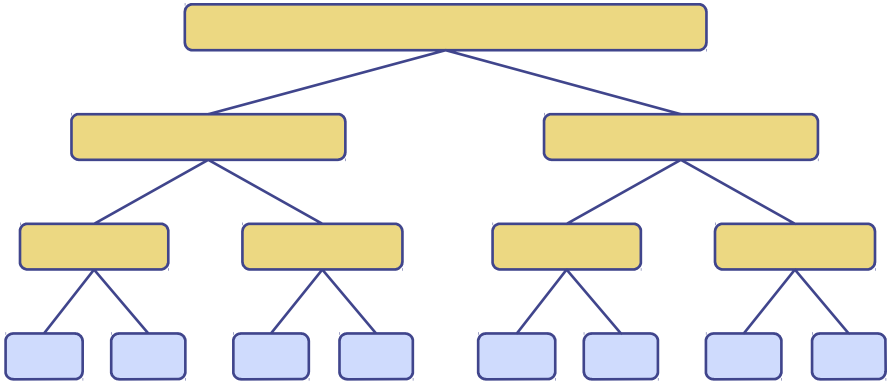
- **Altezza dell'albero di ricorsione → O(log n)**
- **Lavoro totale svolto nei nodi a profondità i**



# Analisi (cont.)

- **Lavoro complessivo a profondità i → O(n)**
- **O(log n) livelli → complessità O(n log n)**

| prof. | #seqs | dim. |
|---|---|---|
| $0$ | $1$ | $n$ |
| $1$ | $2$ | $n/2$ |
| $i$ | $2^i$ | $n/2^i$ |
| ... | ... | ... |



# Altre considerazioni

- **Memoria aggiuntiva necessaria per effettuare il merge**
- **Analisi del costo usando la ricorrenza**
  - Sia T(x) il costo nel caso peggiore del Merge Sort per ordinare un array di x elementi
  - $T(n) \leq \underbrace{2\,T(n/2)}_{\text{Ricorsione}} + \underbrace{cn}_{\text{Merge}}$, dove c è una costante
  - La disugualianza sopra può essere risolta
  - Il risultato in forma chiusa è O(n logn) come già sappiamo