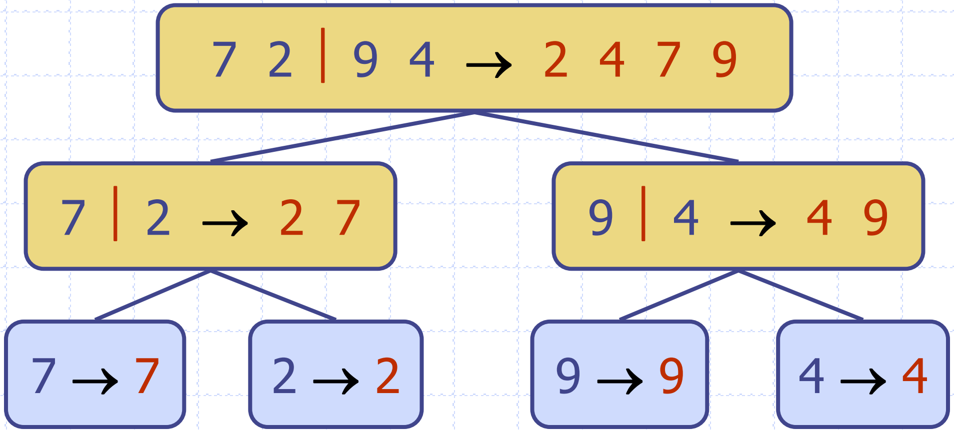
Merge Sort



Divide-et-Impera (§ 11.1.1)

- Divide et Impera è un paradigma generale di progetto degli algoritmi:
  - Divide: divide l'insieme di input S in due sottoinsiemi disgiunti S_1 e S_2
  - Ricorri: risolve i sottoproblemi associati con S_1 e S_2
  - Impera: combina le soluzioni per S_1 e S_2 in una soluzione per S
- Il caso base della ricorsione sono problemi di taglia 0 o 1.
- Merge-sort è un algoritmo di ordinamento basato sul paradigma divide et impera
- Come heap-sort
  - usa un comparator
  - ha tempo di esecuzione $O(n \log n)$
- Diversamente da heap-sort
  - Non usa una coda di priorità ausiliaria
  - Accede i dati in ordine sequenziale (adatto all'ordinamento di dati su disco)

Merge-Sort (§ 11.1)

- Merge-sort su una sequenza di input S con n elementi consiste di 3 passi:
  - Divide: partiziona S in due sequenze S_1 and S_2 di circa $n/2$ elementi ciascuna
  - Ricorri: ricorsivamente ordina S_1 e S_2
  - Impera: merge S_1 e S_2 in un'unica sequenza ordinata

Algorithm *mergeSort*(S, C)
    Input sequence S with n elements, comparator C
    Output sequence S sorted according to C
    if $S.size() > 1$
        $(S_1, S_2) \leftarrow partition(S, n/2)$
        $mergeSort(S_1, C)$
        $mergeSort(S_2, C)$
        $S \leftarrow merge(S_1, S_2)$

Fusione di due Sequenze Ordinate

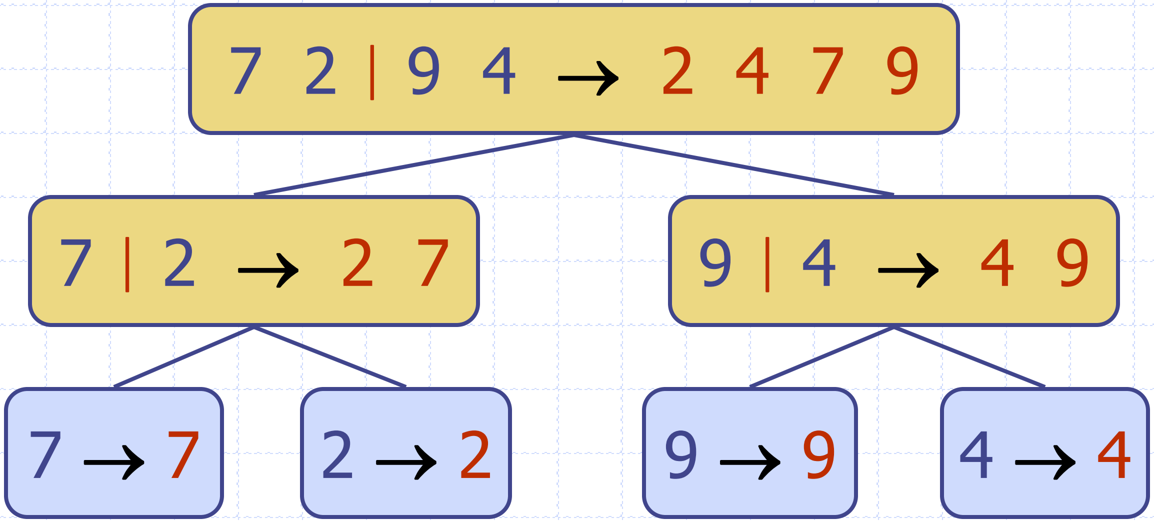
- Il passo di impera del merge-sort consiste nella fusione di due sequenze ordinate A e B in una unica sequenza ordinata S contenente l'unione degli elementi di A e B
- La fusione di due sequenze ordinate, ognuna con $n/2$ elementi, ed implementata attraverso una lista doppiamente collegata, richiede tempo $O(n)$

Algorithm *merge*(A, B)
  Input sequences A and B with
    $n/2$ elements each
  Output sorted sequence of $A \cup B$

  $S \leftarrow$ empty sequence
  while $\neg A.isEmpty() \wedge \neg B.isEmpty()$
    if $A.first().element() < B.first().element()$
      $S.insertLast(A.remove(A.first()))$
    else
      $S.insertLast(B.remove(B.first()))$
  while $\neg A.isEmpty()$
    $S.insertLast(A.remove(A.first()))$
  while $\neg B.isEmpty()$
    $S.insertLast(B.remove(B.first()))$
  return S

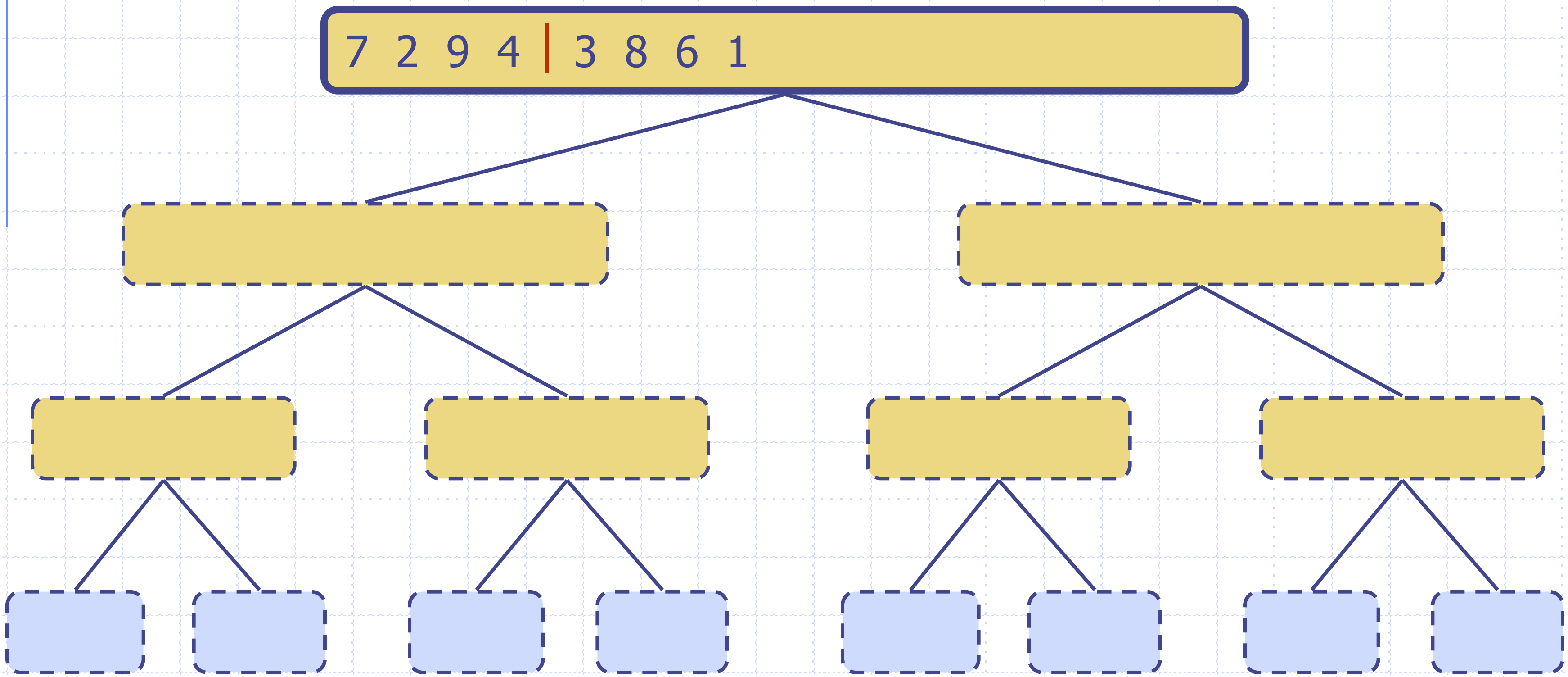
Albero di Merge-Sort

- Un esecuzione del merge-sort è illustrata da un albero binario
  - Ogni nodo rappresenta una chiamata ricorsiva del merge-sort e memorizza
    - sequenza non ordinata prima dell’esecuzione e della sua partizione
    - sequenza ordinata alla fine dell’esecuzione
  - La radice è la chiamata iniziale
  - Le foglie sono le chiamate su sottosequenze di dimensione 0 o 1



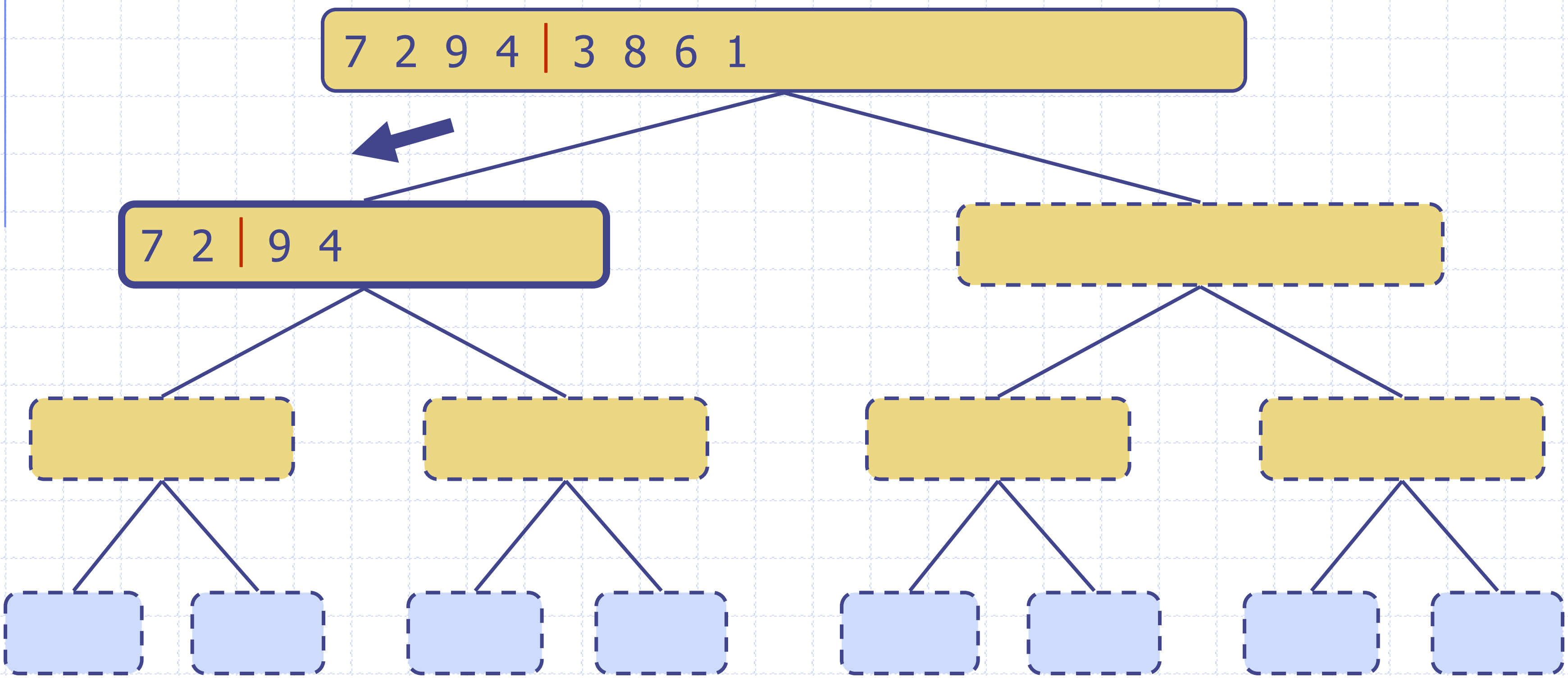
Esempio di esecuzione

- Partizione



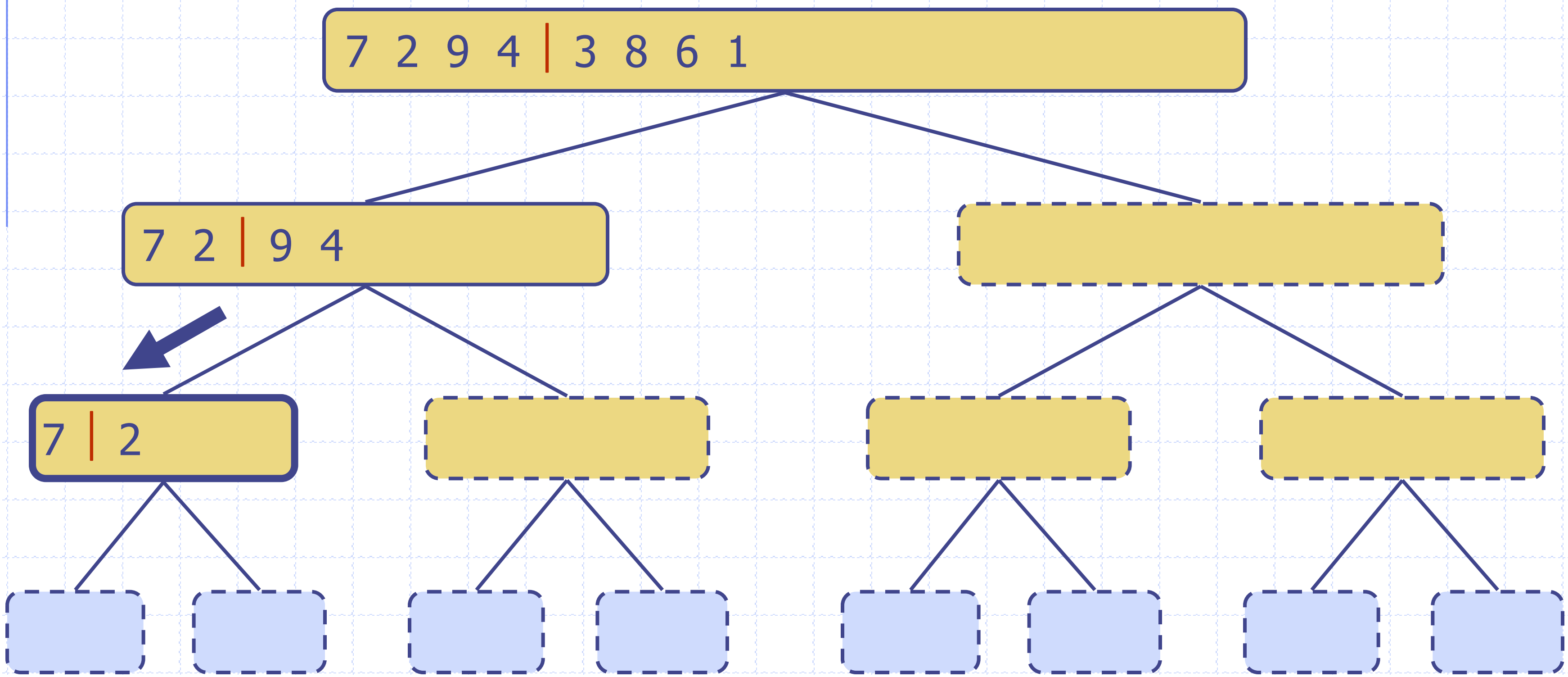
Esempio di esecuzione (cont.)

- Chiamata ricorsiva, partizione



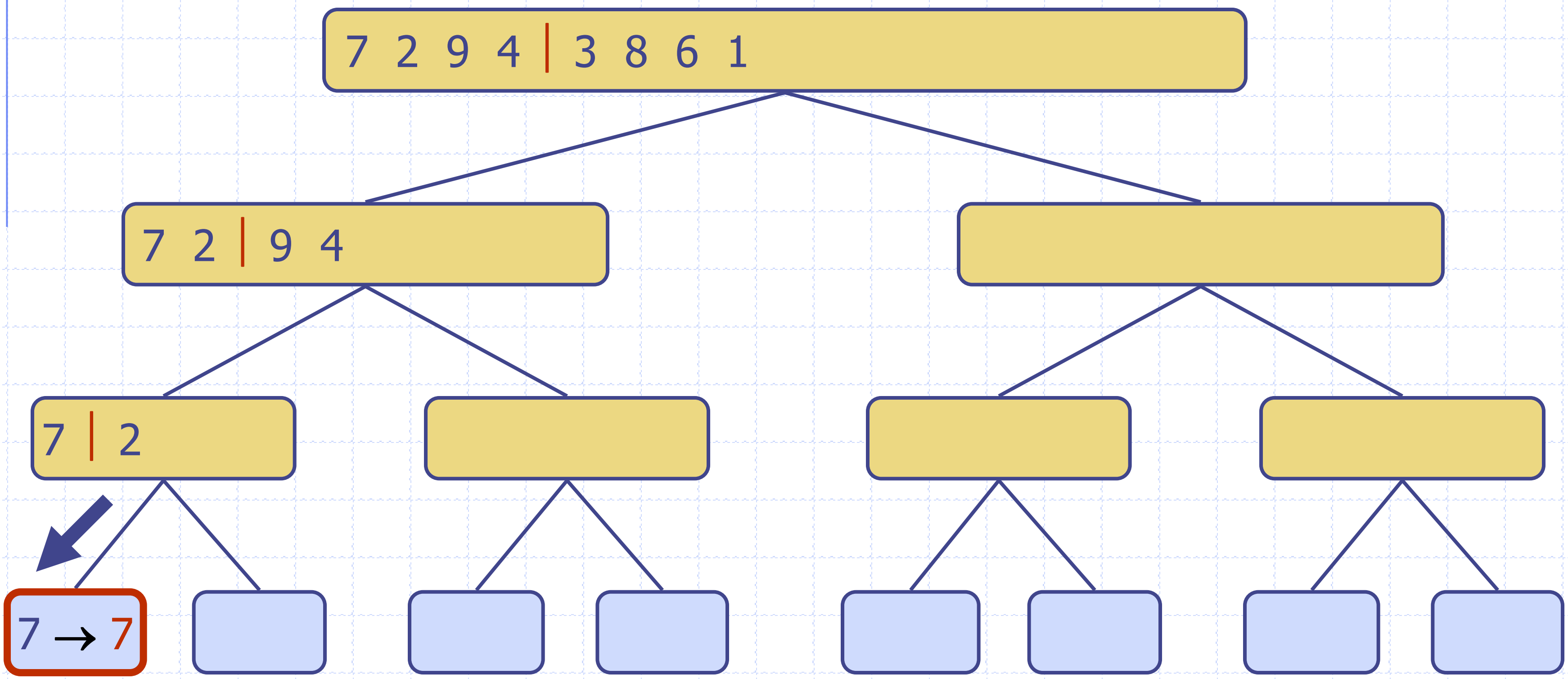
Esempio di esecuzione(cont.)

- Chiamata ricorsiva, partizione



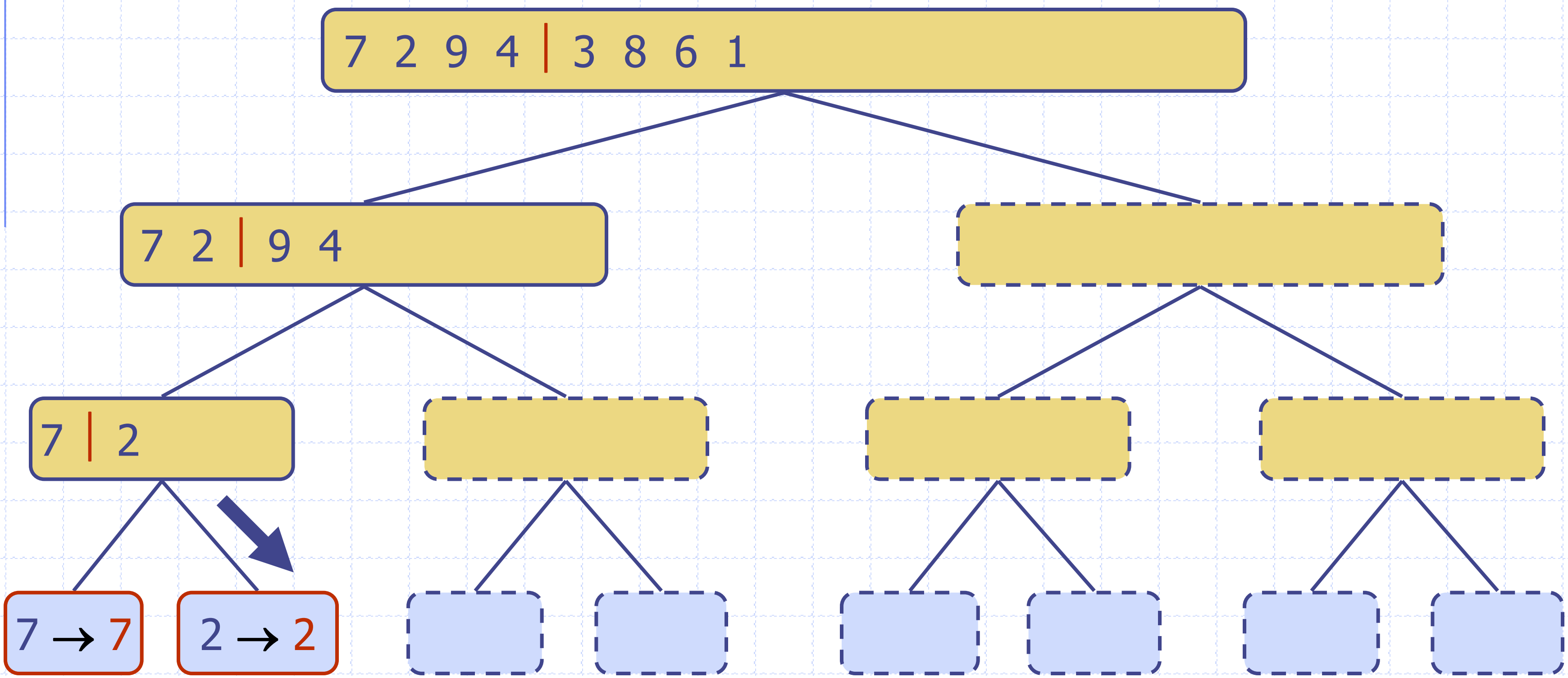
Esempio di esecuzione (cont.)

- Chiamata ricorsiva, caso base



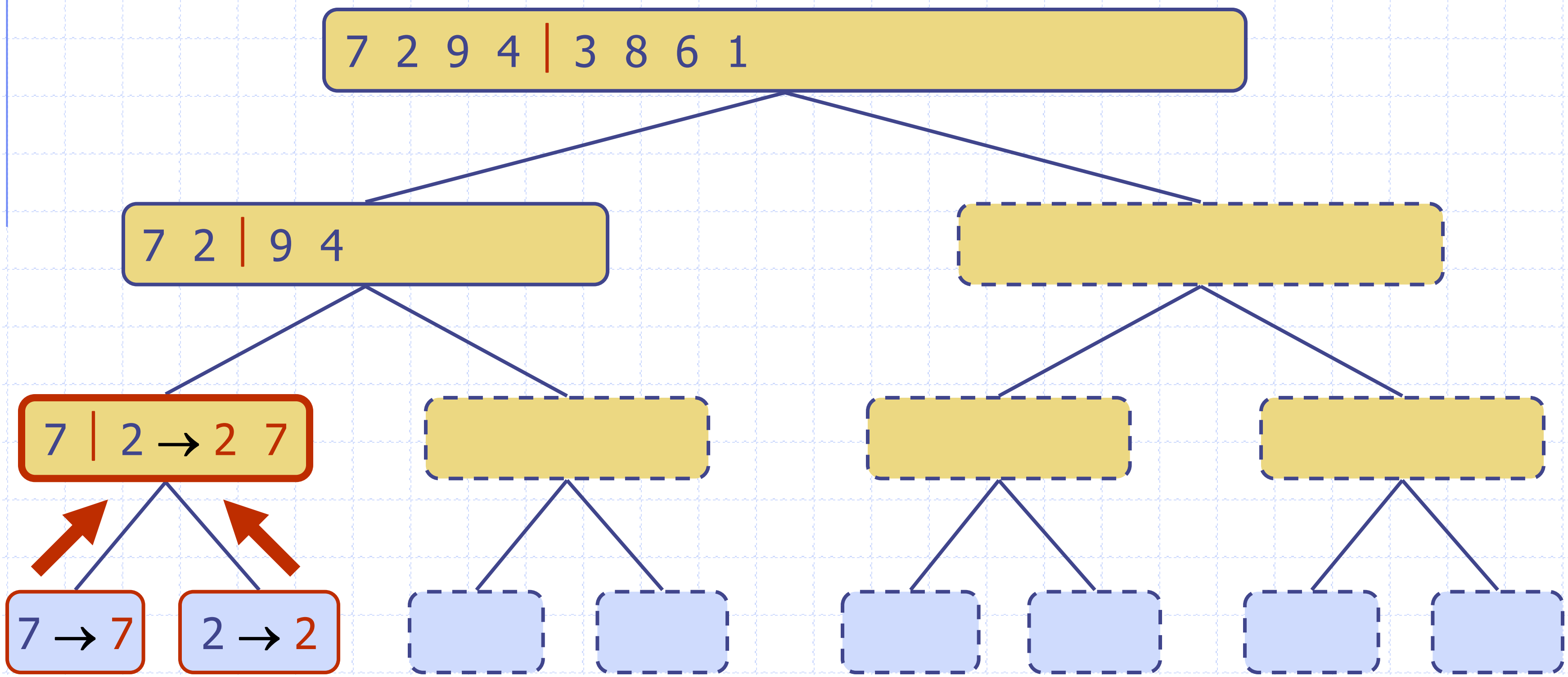
Esempio di esecuzione (cont.)

- Chiamata ricorsiva, caso base



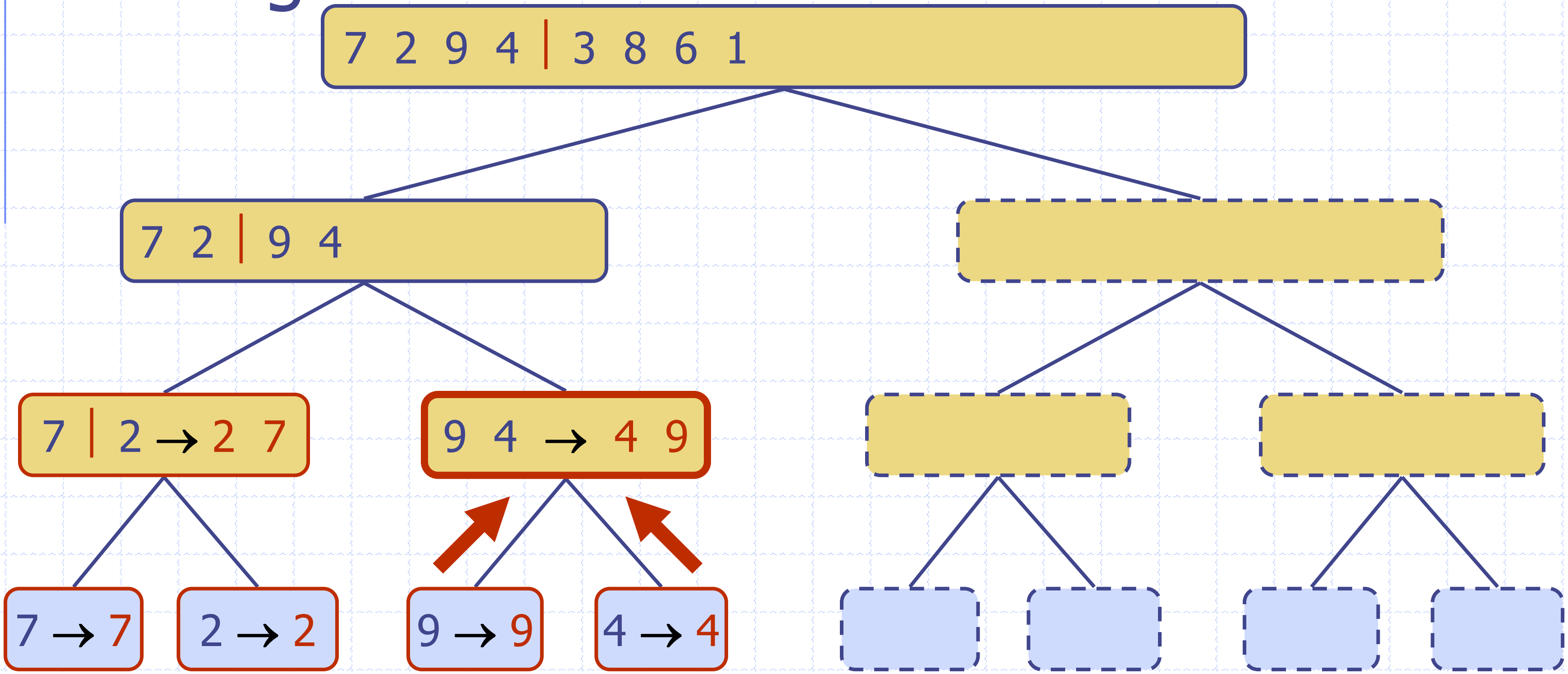
Esempio di esecuzione (cont.)

- Merge



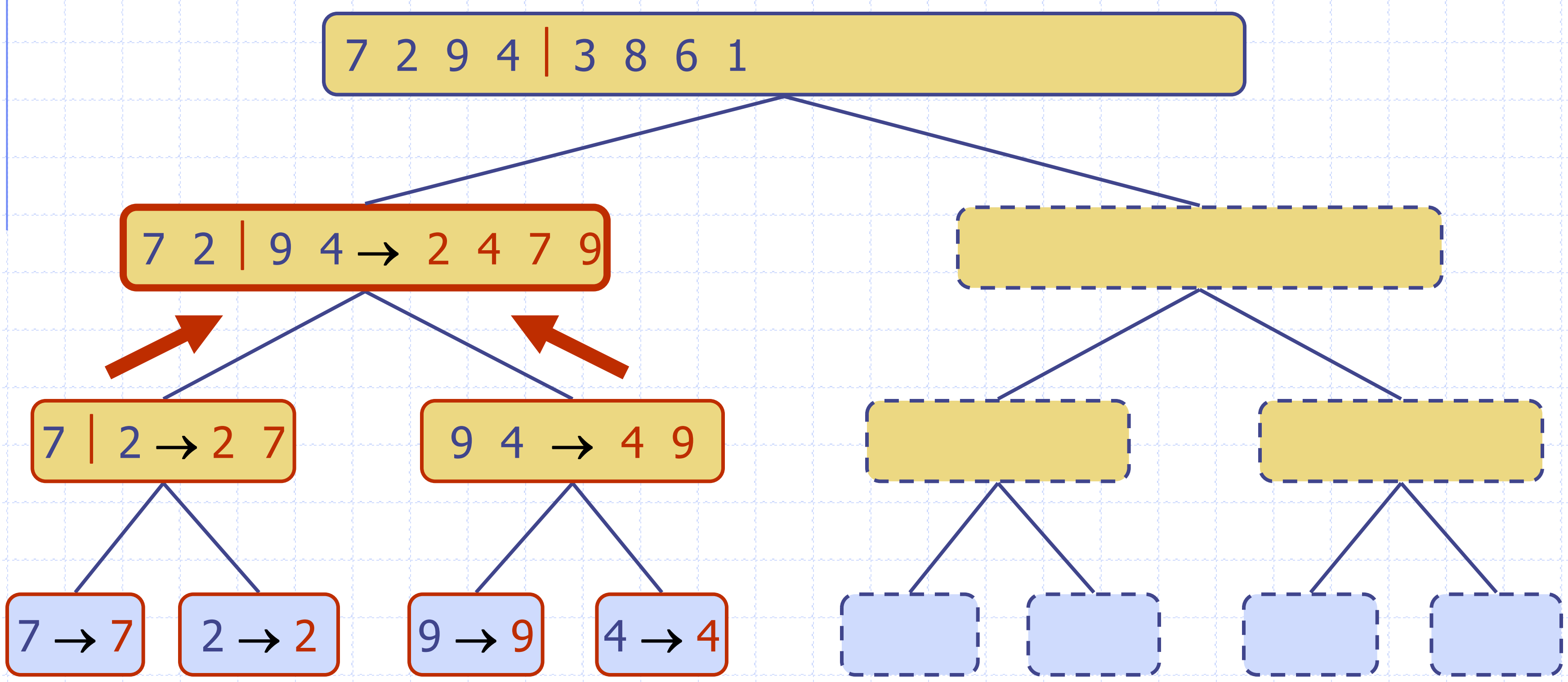
Esempio di esecuzione (cont.)

- Chiamata ricorsiva, …, caso base, merge



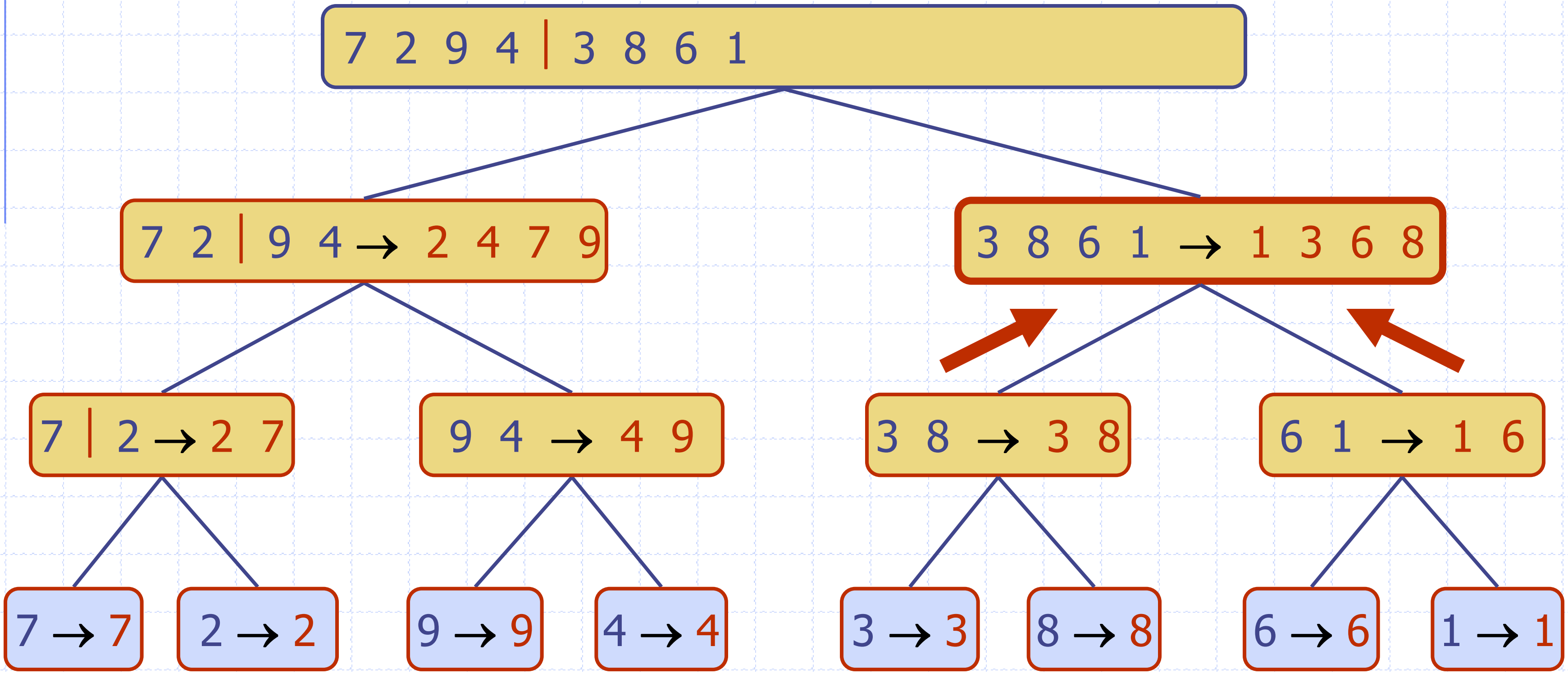
Esempio di esecuzione (cont.)

- Merge



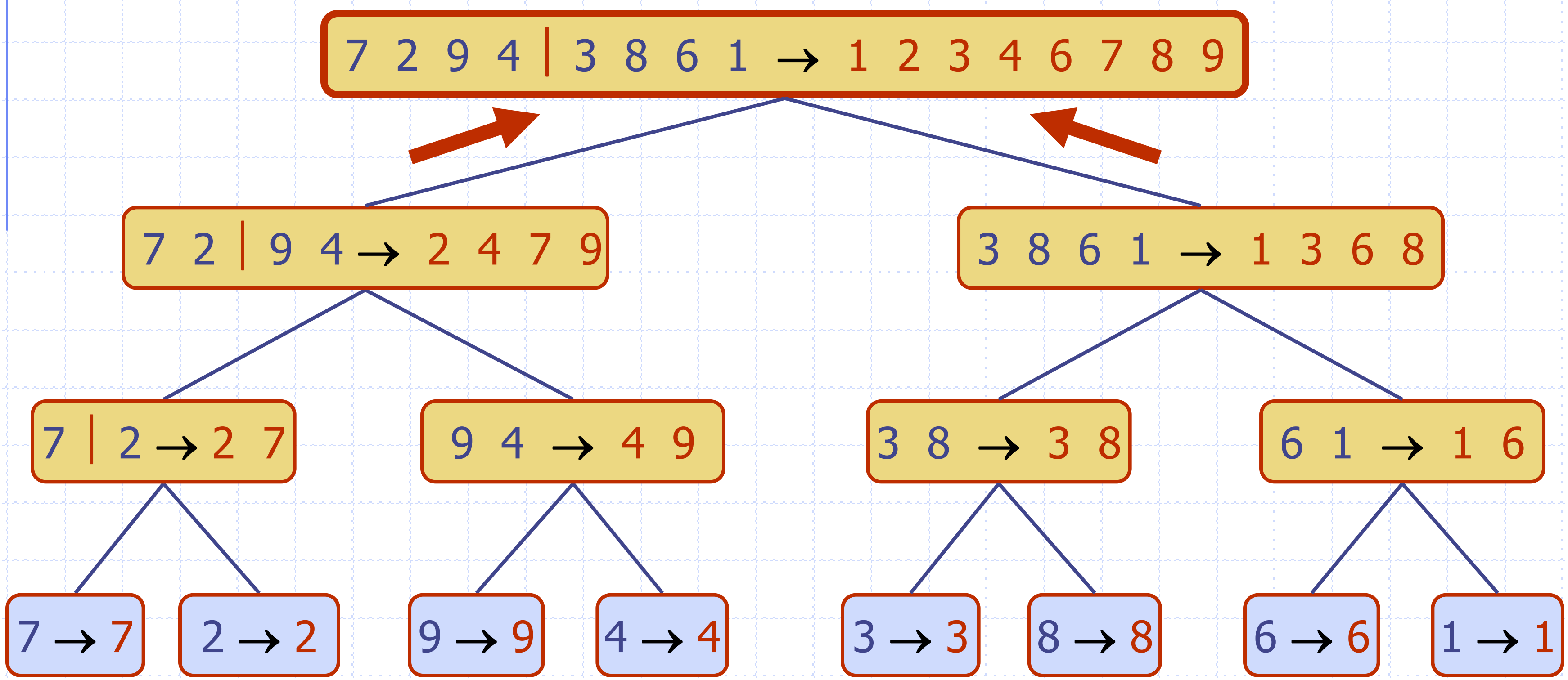
Esempio di esecuzione (cont.)

- Chiamata ricorsiva, …, merge, merge



Esempio di esecuzione (cont.)

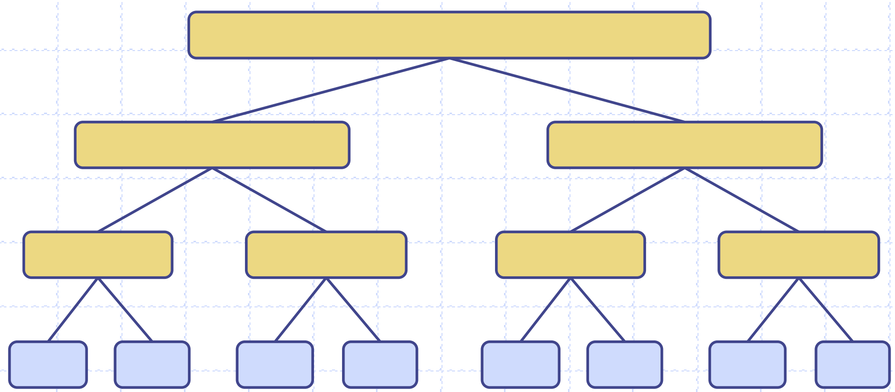
- Merge



Analisi di Merge-Sort

- L'altezza h dell'albero di merge-sort è $O(\log n)$
  - Ad ogni chiamata ricorsiva dividiamo a metà la sequenza,
- La quantità globale di lavoro svolto ai nodi di profondità i è $O(n)$
  - partizioniamo e uniamo 2^i sequenze di dimensione $n/2^i$
  - realizziamo 2^{i+1} chiamate ricorsive
- Quindi, il tempo totale di esecuzione è $O(n \log n)$

depth	#seqs	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
…	…	…



Equazioni di ricorrenza

- Tempo di esecuzione di algoritmi ricorsivi descritti con equazioni di ricorrenza. Ex: MergeSort:

$$f(n) = \begin{cases} \theta(1), & n = 2 \\ f(\lceil n/2 \rceil) + f(\lfloor n/2 \rfloor) + \theta(n), & n > 2 \end{cases}$$

- Semplificazioni:
  - Argomenti non interi.

$$f(n) = \begin{cases} c_1, n = 1 \\ 2f(n/2) + c_2 n, n \geq 2 \end{cases}$$

  - Condizioni al contorno: $\theta(1)$ per n piccolo

Soluzione di equazioni di ricorrenza

- Metodo per sostituzione. Si tenta una soluzione *g(n)* e si verifica se soddisfa l'equazione di ricorsione.
- Dimostrazione per induzione.
- Base dell'induzione: $f(1) \le g(1)$
- Ipotesi induttiva: $f(n/2) \le g(n/2)$
- Passo induttivo: $f(n) \le g(n)$
- Per Merge Sort:

$$g(n) = cn \log n + a$$

Soluzione equazioni di ricorrenza per Merge Sort

- Base: $f(1) = c_1 \leq g(1) = a$
- Ipotesi induttiva: $f(n/2) \leq g(n/2)$
- Passo induttivo:

$$
\begin{aligned}
f(n) &\leq 2g(n/2) + c_2 n \\
&\leq 2c(n/2)\log(n/2) + c_2 n + 2a \\
&\leq cn\log(n/2) + c_2 n + 2a \\
&\leq cn\log n - cn + c_2 n + 2a \\
&\leq cn\log n + a \qquad c_1 + c_2 \leq c
\end{aligned}
$$

Riepilogo degli algoritmi di ordinamento

Algoritmo	Tempo	Proprietà
selection-sort	$O(n^2)$	◈ lento ◈ sul posto ◈ per insiemi piccoli di dati (< 1K)
insertion-sort	$O(n^2)$	◈ lento ◈ sul posto ◈ per insiemi piccoli di dati (< 1K)
heap-sort	$O(n \log n)$	◈ veloce ◈ sul posto ◈ per insiemi grandi di dati (1K — 1M)
merge-sort	$O(n \log n)$	◈ veloce ◈ accesso sequenziale ai dati

Merge-Sort Non Ricorsivo

Unisce esecuzioni di lunghezza 2, quindi 4, quindi 8, e cosi’ via

```
public static void mergeSort(Object[] orig, Comparator c) { // nonrecursive
  Object[] in = new Object[orig.length]; // make a new temporary array
  System.arraycopy(orig,0,in,0,in.length); // copy the input
  Object[] out = new Object[in.length]; // output array
  Object[] temp; // temp array reference used for swapping
  int n = in.length;
  for (int i=1; i < n; i*=2) { // each iteration sorts all length-2*i runs
    for (int j=0; j < n; j+=2*i)  // each iteration merges two length-i pairs
      merge(in,out,c,j,i); // merge from in to out two length-i runs at j
    temp = in; in = out; out = temp; // swap arrays for next iteration
  }
  // the "in" array contains the sorted array, so re-copy it
  System.arraycopy(in,0,orig,0,in.length);
}
```

unisce due esecuzioni nell’out array

```
protected static void merge(Object[] in, Object[] out, Comparator c, int start,
    int inc) { // merge in[start..start+inc-1] and in[start+inc..start+2*inc-1]
  int x = start; // index into run #1
  int end1 = Math.min(start+inc, in.length); // boundary for run #1
  int end2 = Math.min(start+2*inc, in.length); // boundary for run #2
  int y = start+inc; // index into run #2 (could be beyond array boundary)
  int z = start; // index into the out array
  while ((x < end1) && (y < end2))
    if (c.compare(in[x],in[y]) <= 0) out[z++] = in[x++];
    else out[z++] = in[y++];
  if (x < end1) // first run didn't finish
    System.arraycopy(in, x, out, z, end1 - x);
  else if (y < end2) // second run didn't finish
    System.arraycopy(in, y, out, z, end2 - y);
}
```