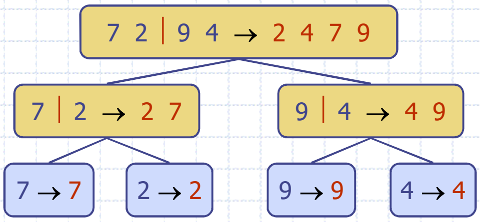
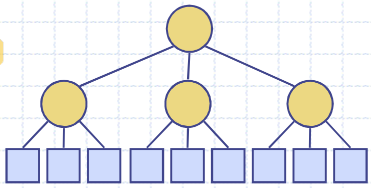
Merge Sort



1

Divide-and-Conquer

- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data S in two or more disjoint subsets S_1, S_2, …
  - Conquer: solve the subproblems recursively
  - Combine: combine the solutions for S_1, S_2, …, into a solution for S
- The base case for the recursion are subproblems of constant size (typically 0 or 1)



2

Merge-Sort

- Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm
- Merge-sort on an input sequence S with n elements consists of three steps:
  - Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
  - Recur: recursively sort S_1 and S_2
  - Conquer: merge S_1 and S_2 into a unique sorted sequence

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

grosso della complessità

3

Merging Two Sorted Sequences

- The conquer step of merge-sort consists of merging two sorted sequences A and B into a sorted sequence S containing the union of the elements of A and B
- Merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes $O(n)$ time

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 $A.first().element() < B.first().element()$
      $S.addLast(A.remove(A.first()))$
    else
      $S.addLast(B.remove(B.first()))$
  while $\neg A.isEmpty()$
    $S.addLast(A.remove(A.first()))$
  while $\neg B.isEmpty()$
    $S.addLast(B.remove(B.first()))$
  return S

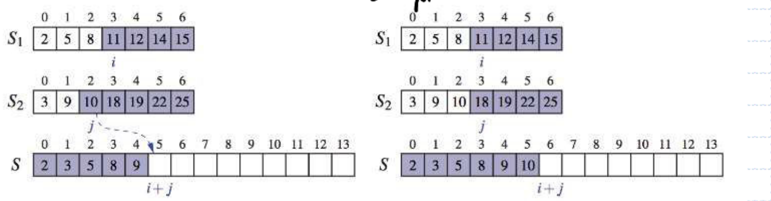
4

Operaz. fatte sono dim S_1 + dim S_2.
Se S_1 e S_2 sono $\frac{n}{2}$, la complex. è $O(n)$

Java Merge Implementation

```
/** 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
  }
}
```

→ S_1 non è finito e $S1[i]<S2[j]$
→ Se S_2 è finito.



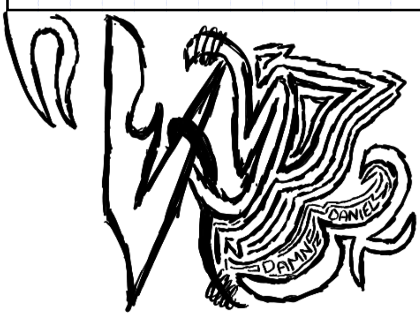
Java Merge-Sort Implementation

```
/** 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
}
```

√ CASO BASE
→ Spezzo S

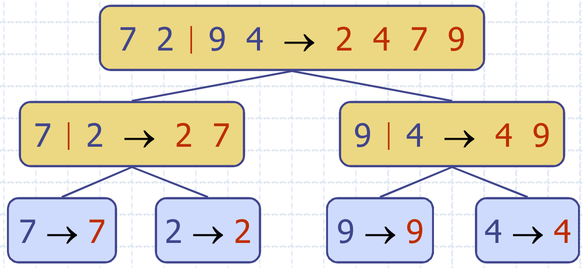
Da un prob. di dimensione n → 2 prob. di dimensione $\frac{n}{2}$

RICORSIONE BINARIA



Merge-Sort Tree

- An execution of merge-sort is depicted by a binary tree
  - each node represents a recursive call of merge-sort and stores
    - unsorted sequence before the execution and its partition
    - sorted sequence at the end of the execution
  - the root is the initial call
  - the leaves are calls on subsequences of size 0 or 1



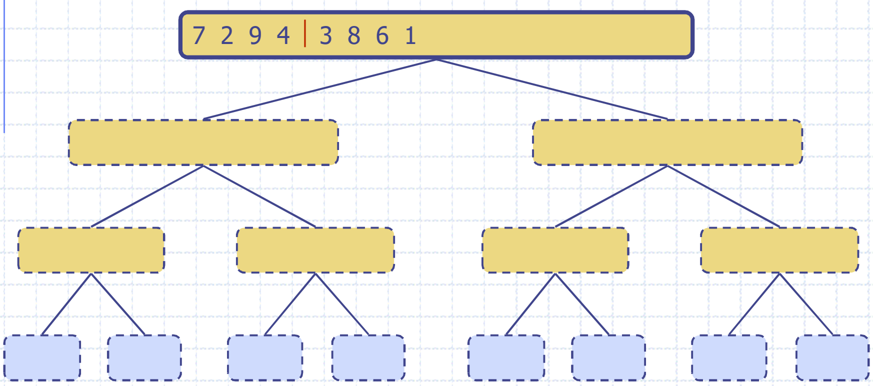
mergeSort(S_1) → return

Size<2

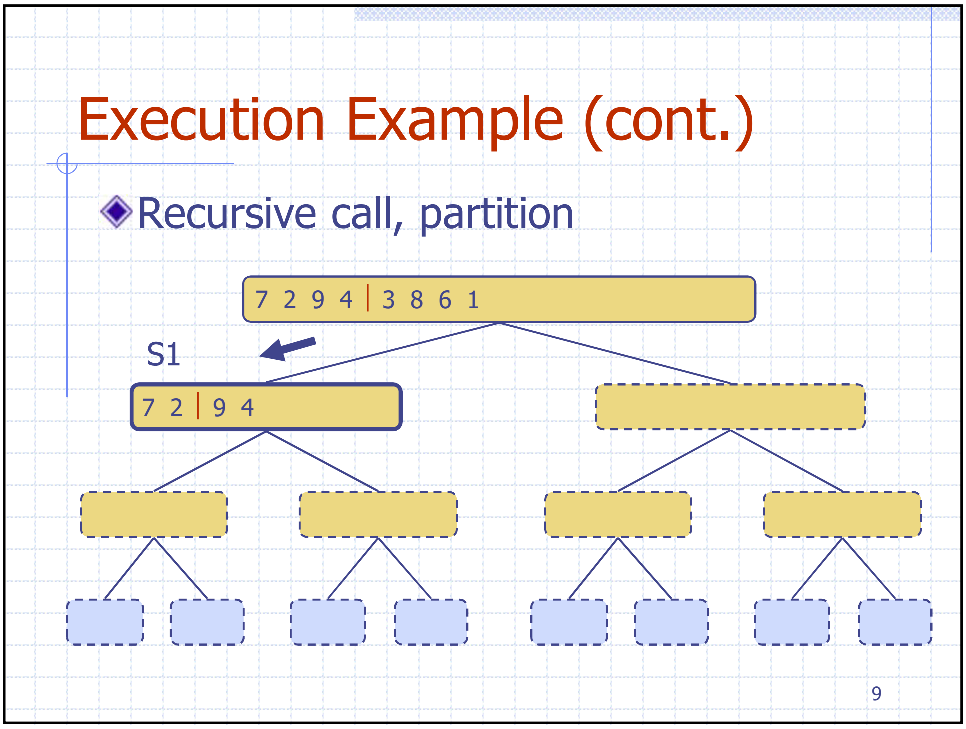
7

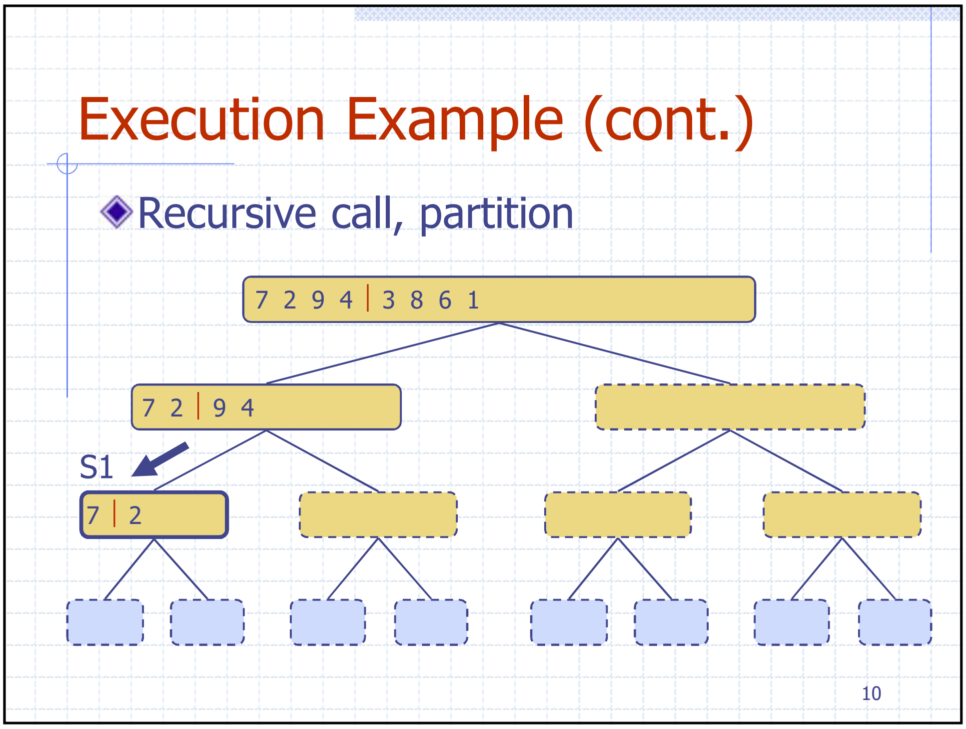
Execution Example

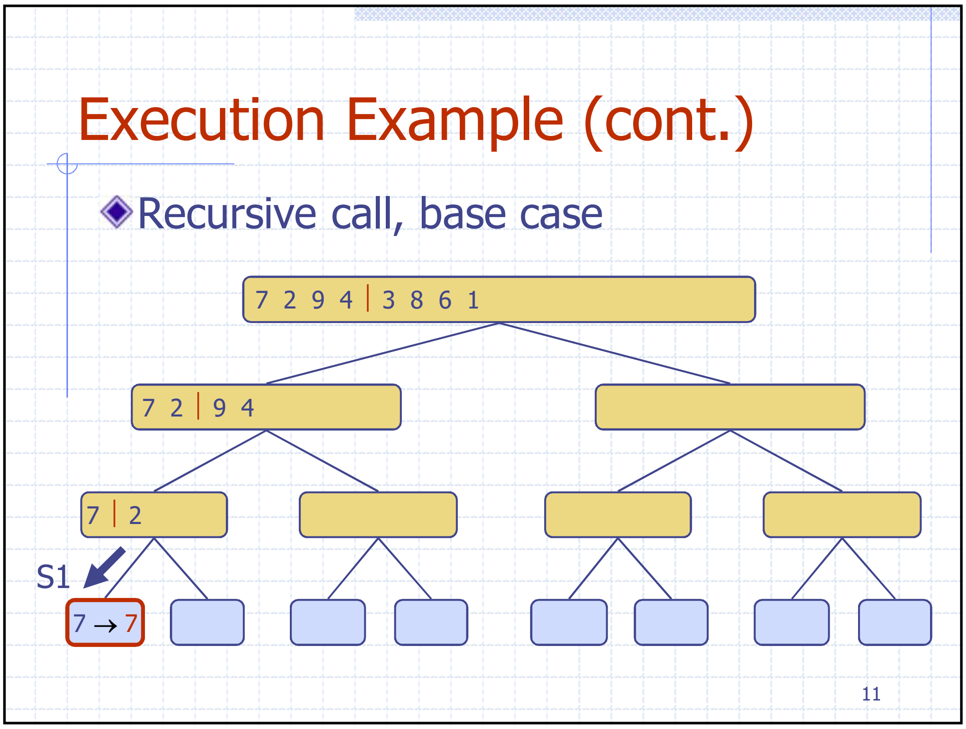
- Partition

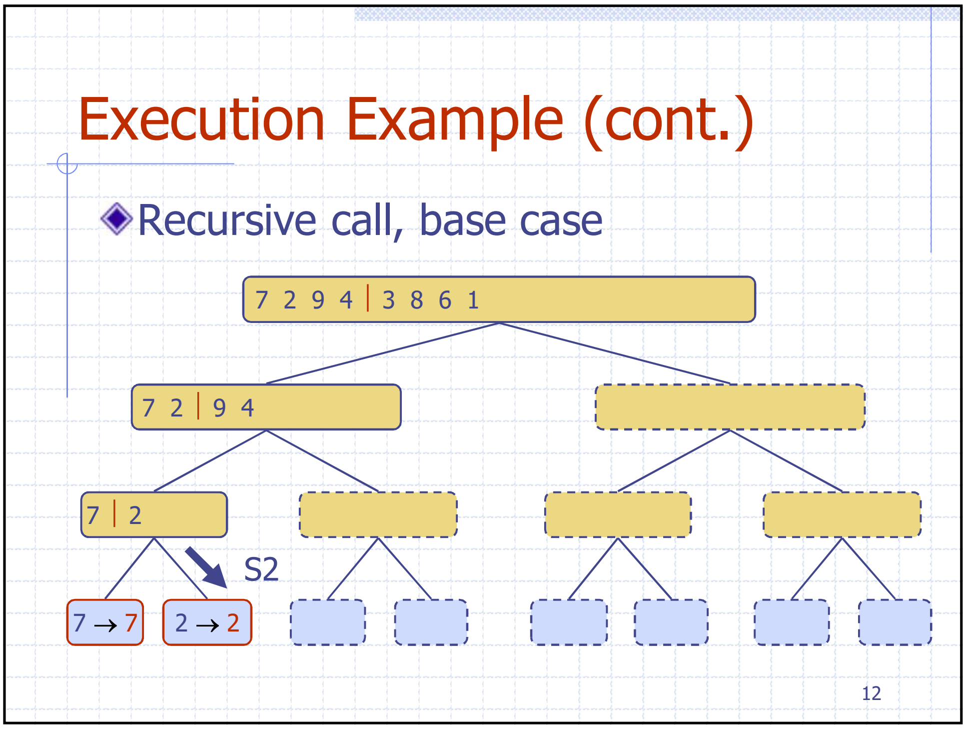


8


Execution Example (cont.)
Recursive call, partition
7 2 9 4 | 3 8 6 1
S1
7 2 | 9 4
9

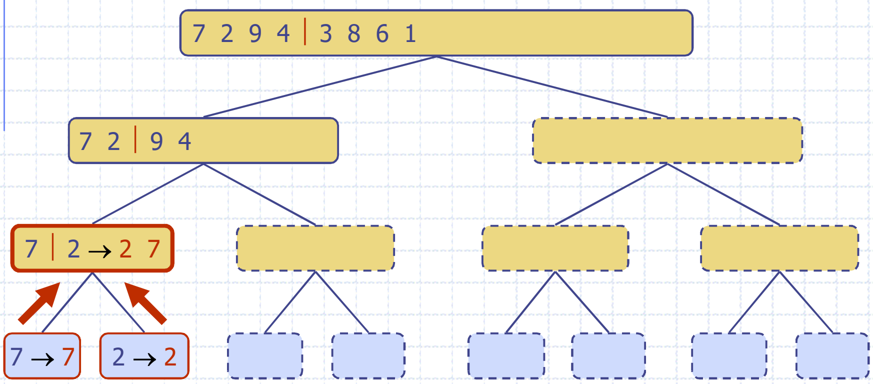

Execution Example (cont.)
Recursive call, partition
7 2 9 4 | 3 8 6 1
7 2 | 9 4
S1
7 | 2
10


Execution Example (cont.)
Recursive call, base case
7 2 9 4 | 3 8 6 1
7 2 | 9 4
7 | 2
S1
7 → 7
11


Execution Example (cont.)
Recursive call, base case
7 2 9 4 | 3 8 6 1
7 2 | 9 4
7 | 2
S2
7 → 7
2 → 2
12

Execution Example (cont.)

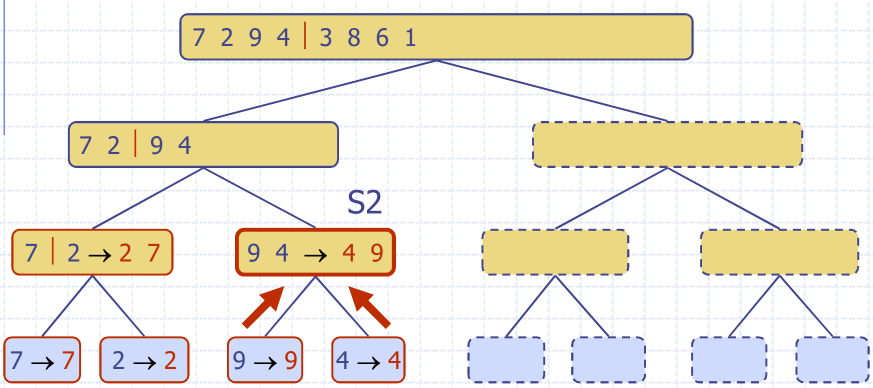
- Merge



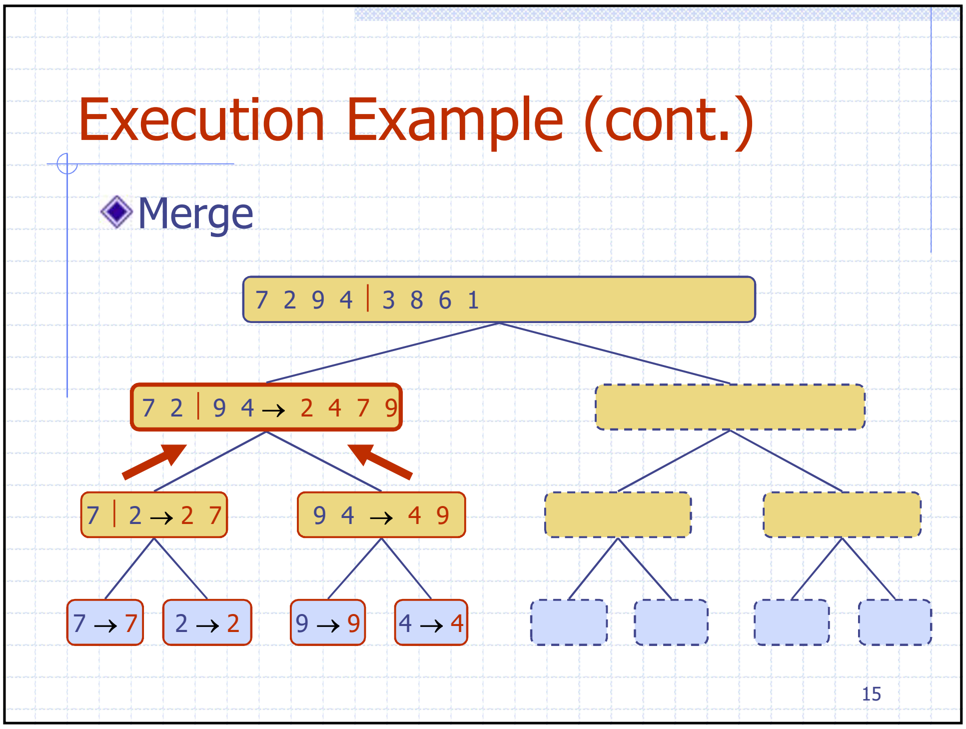
13

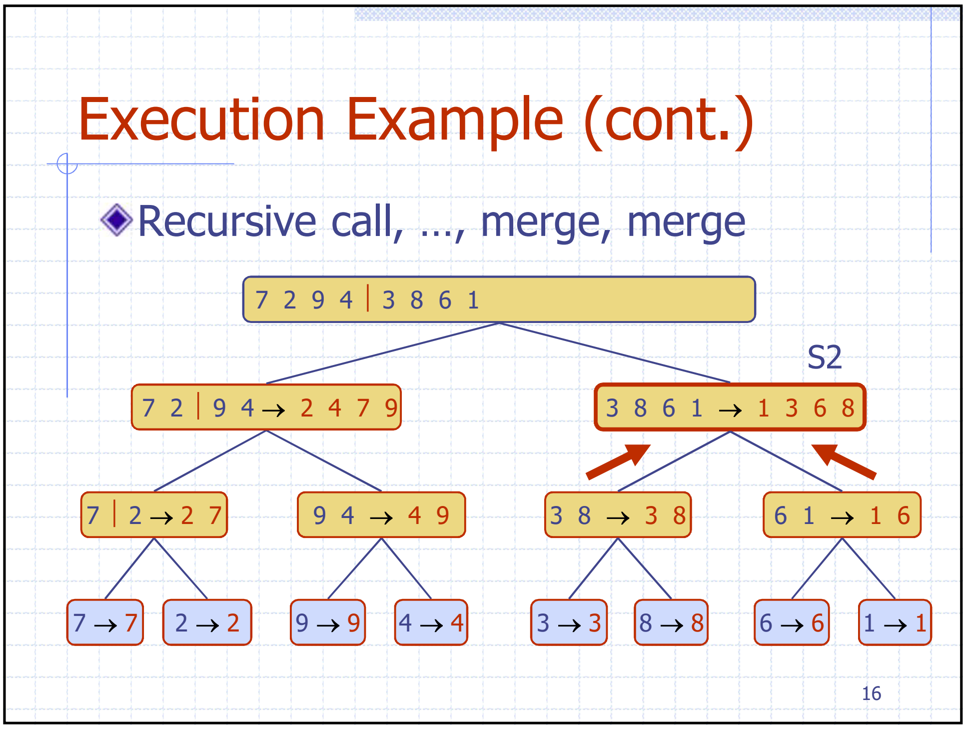
Execution Example (cont.)

- Recursive call, …, base case, merge



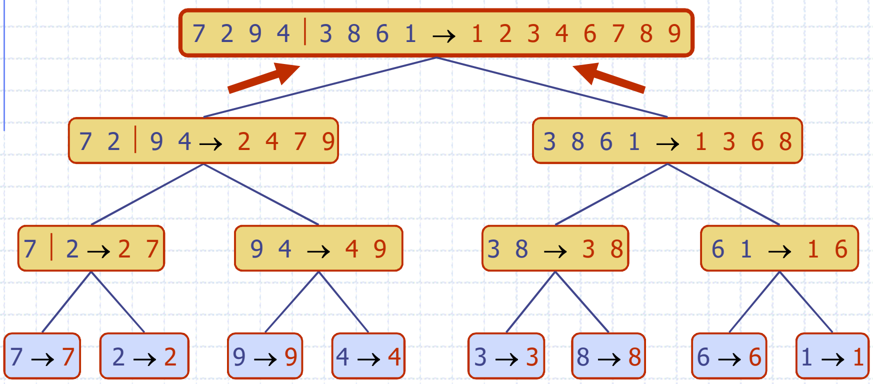
14


Execution Example (cont.)
Merge
7 2 9 4 | 3 8 6 1
7 2 | 9 4 → 2 4 7 9
7 | 2 → 2 7
9 4 → 4 9
7 → 7
2 → 2
9 → 9
4 → 4
15


Execution Example (cont.)
Recursive call, …, merge, merge
7 2 9 4 | 3 8 6 1
S2
7 2 | 9 4 → 2 4 7 9
3 8 6 1 → 1 3 6 8
7 | 2 → 2 7
9 4 → 4 9
3 8 → 3 8
6 1 → 1 6
7 → 7
2 → 2
9 → 9
4 → 4
3 → 3
8 → 8
6 → 6
1 → 1
16

Execution Example (cont.)

- Merge



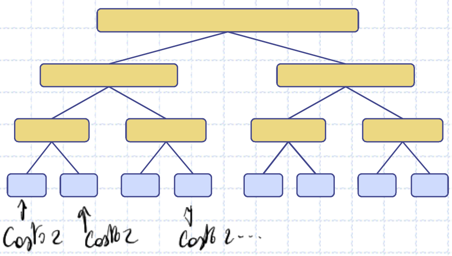
17

Fase di discesa: Non costa niente dividere in due

Analysis of Merge-Sort

- The height h of the merge-sort tree is $O(\log n)$
  - at each recursive call we divide in half the sequence,
- The overall amount or work done at the nodes of depth i is $O(n)$
  - we partition and merge 2^i sequences of size $n/2^i$
  - we make 2^{i+1} recursive calls
- Thus, the total running time of merge-sort is $O(n \log n)$

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



18

Merge: Costo = Somma delle dim. delle 2 sottoliste
Per passare da ultimo livello a penultimo: n.
Adesso ho 8 liste da 1 → 4 liste da 2.
Costo 4 per due merging. ⇒ 8.

Ogni livello di Merging ha costo di $O(n)$.

Quanti sono livelli? Da n per dimezz. successivi,

per arrivare a 1 ho $\log n$ suddivisione.

$\log n$ livelli, ogni livello ha numero di nodi pari a 2^i, $\frac{n}{2^i}$ elementi

per nodo. Prodotto: sempre n. Numero nodi $\times$ numero elementi ogni nodo.

Complessità per ogni livello n.

$\Rightarrow$ Big O: $O(n \log n)$

Merge Sort ha sempre $O(n \log n)$ perché

non dipende dai dati. Le operaz. fatte sono

sempre le stesse.