Algoritmi ricorsivi
Tecnica Divide et Impera
Esercizi

17 di marzo 2023 d. C.

Punto della situazione

Corso di Progettazione di Algoritmi:
Analisi e progettazione di algoritmi

Abbiamo acquisito alcuni strumenti per effettuare l'analisi asintotica di algoritmi (non ricorsivi).

Cominceremo a vedere una prima tecnica di programmazione: la tecnica del Divide-et-Impera che produce algoritmi ricorsivi. Che già conoscete…

Cominciamo riprendendo gli algoritmi ricorsivi con un esempio classico: il calcolo del fattoriale

Funzioni definite per ricorsione: il fattoriale

Supponiamo di voler calcolare il fattoriale di *n.*

Partiamo dalla sua definizione:

$$n! = n \cdot \underbrace{(n-1) \cdot \;\cdots\; \cdot 3 \cdot 2 \cdot 1}_{(n-1)!}$$

... da cui otteniamo una definizione del fattoriale in termini di <u>se stesso</u>:

$$n! = n \cdot (n-1)! \qquad \text{se} \qquad n > 0$$

$$0! = 1$$

Queste equazioni definiscono anche ciascuna un **metodo** per calcolare il fattoriale

Un algoritmo iterativo per il fattoriale

Dalla definizione

$$n! = n \cdot (n-1) \cdot \cdots \cdot 3 \cdot 2 \cdot 1$$

```
Iter-Fact(n)
fact=1
For i=2 to n
     fact=fact*i
Endfor
Return fact
```

Un programma ricorsivo per il fattoriale

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot (n-1)! & \text{se } n > 0 \end{cases}$$

```
int fact (int n)
    if (n==0) return 1
        else return n * fact(n-1)
```

Chiamata ricorsiva: fact è definito **tramite se stesso**

Come funziona il programma ricorsivo per il fattoriale?

Simuliamo il calcolo di **`fact(4)`** rimpiazzando più volte fact(n) fino ad arrivare al caso base n=0

```
fact(4)

4 * fact(3)

4 * (3 * fact(2))

4 * (3 * (2 * fact(1)))

4 * (3 * (2 * (1 * fact(0))))

4 * (3 * (2 * (1 * 1)))

4 * (3 * (2 * 1))

4 * (3 * 2)

4 * 6

24
```

Algoritmi ricorsivi

Schema di un algoritmo ricorsivo (su un'istanza $\mathcal{I}$):

ALGO ($\mathcal{I}$)

If «caso base» then «esegui certe operazioni»

else «esegui delle operazioni fra le quali

ALGO($\mathcal{I}_1$), … , ALGO($\mathcal{I}_k$) »

Per assicurare che la ricorsione termini bisogna fare attenzione a che le chiamate ricorsive **si applichino a valori di "dimensione" minore del valore in ingresso**, e che le clausole di uscita **contemplino tutti i casi base**.

Algoritmi basati sulla tecnica Divide et Impera

In questo corso:

- Ricerca binaria
- Mergesort (ordinamento)
- Quicksort (ordinamento)
- Moltiplicazione di interi/ matrici

NOTA: nonostante la tecnica *Divide et impera* sembri così «semplice» ben due «top ten algorithms of the 20° century» sono basati su di essa:
Fast Fourier Transform (FFT)
Quicksort

Top ten algorithms of the 20° century

10 algorithms having "the greatest influence on the development and practice of science and engineering in the 20th century" by Francis Sullivan and Jack Dongarra, Computing in Science & Engineering, January/February 2000

1. **1946: The Metropolis Algorithm for Monte Carlo.** Through the use of random processes, this algorithm offers an efficient way to stumble toward answers to problems that are too complicated to solve exactly.
2. **1947: Simplex Method for Linear Programming.** An elegant solution to a common problem in planning and decision-making.
3. **1950: Krylov Subspace Iteration Method.** A technique for rapidly solving the linear equations that abound in scientific computation.
4. **1951: The Decompositional Approach to Matrix Computations.** A suite of techniques for numerical linear algebra.
5. **1957: The Fortran Optimizing Compiler.** Turns high-level code into efficient computer-readable code.
6. **1959: QR Algorithm for Computing Eigenvalues.** Another crucial matrix operation made swift and practical.
7. **1962: Quicksort Algorithms for Sorting.** For the efficient handling of large databases. (Divide et impera)
8. **1965: Fast Fourier Transform.** Perhaps the most ubiquitous algorithm in use today, it breaks down waveforms (like sound) into periodic components. (Divide et impera)
9. **1977: Integer Relation Detection.** A fast method for spotting simple equations satisfied by collections of seemingly unrelated numbers.
10. **1987: Fast Multipole Method.** A breakthrough in dealing with the complexity of n-body calculations, applied in problems ranging from celestial mechanics to protein folding.

Divide et Impera

- **Dividi** il problema in sottoproblemi
- Risolvi ogni sottoproblema **ricorsivamente**
- **Combina** le soluzioni ai sottoproblemi per ottenere la soluzione al problema

La ricerca binaria

Problema della ricerca binaria:

INPUT: un array A[1..n] ORDINATO e un elemento key

OUTPUT: un indice i tale che A[i]=key oppure messaggio «non c'è»

Ricerca binaria: algoritmo iterativo (già visto)

Esempio: Ricerca Binaria. Data una lista ordinata $A = a_1, \ldots a_n$ ed un valore $\texttt{key}$, determina l'indice i per cui $a_i = \texttt{key}$, se esso esiste.

$\texttt{first} \leftarrow 1, \texttt{last} \leftarrow n$

while $(\texttt{first} \leq \texttt{last})$

  $\texttt{mid} \leftarrow (\texttt{first} + \texttt{last})/2$; (calcola punto mediano)

  if $(\texttt{key} > a_{\texttt{mid}})$

   $\texttt{first} = \texttt{mid} + 1$; (ripete la ricerca nella metà di destra)

    else if $(\texttt{key} < a_{\texttt{mid}})$

     $\texttt{last} = \texttt{mid} - 1$; (ripete la ricerca nella metà di sinistra)

  else

    return(mid)

return($\texttt{non c'è}$)

Divide-et-impera per la ricerca binaria

Divide-et-Impera

- Dividi il problema in sottoproblemi
- Risolvi ogni sottoproblema ricorsivamente
- Combina le soluzioni ai sottoproblemi per ottenere la soluzione al problema

- Dividi l'array a metà (determinando l'elemento centrale)
- Risolvi **ricorsivamente** sulla metà di destra, o di sinistra, o su nessuna (a secondo del confronto con l'elemento centrale)
- Niente

Per sottoproblema si intende lo stesso problema su un'istanza di taglia inferiore

Esprime più elegantemente/naturalmente l'idea di quello iterativo

Ricerca binaria: algoritmo ricorsivo

Ricerca di **key** in **A[1..n]** dall'indice **first** all'indice **last**.

```
ricerca_binaria_ricorsiva (A[1..n], first, last, key) {
if (first > last) return «non c'è»;
        else {
                mid = (first + last) / 2;
                if (key = A[mid]) return mid;
                else
                if (key > A[mid])
                return  ricerca_binaria_ricorsiva (A, mid+1, last, key);
                else
                return ricerca_binaria_ricorsiva (A, first, mid-1, key);
                }
}
```

Analisi

Per ogni algoritmo che studieremo ci interesserà:

- Dimostrare la correttezza (fornisce l'output desiderato per ogni input)
- Analizzare il tempo di esecuzione (ed eventualmente lo spazio di memoria ausiliaria necessario)

Ricerca binaria: correttezza

```
ricerca_binaria_ricorsiva (A[1..N], first, last, key) {
if (first > last) return «non c'è»;
        else {
        mid = (first + last) / 2;
        if (key = A[mid]) return mid;
        else
        if (key > A[mid])
            return ricerca_binaria_ricorsiva (A, mid+1, last, key);
        else
            return ricerca_binaria_ricorsiva (A, first, mid-1, key);
                          }
           }
```

Taglia input
n= last - first + 1

Per induzione sulla struttura
Caso base: se n=0 key non c'è
Supponiamo che le chiamate sulle due metà correttamente restituiscano l'indice di key se c'è.
Allora la chiamata principale funziona correttamente nei 3 distinti casi:
key = A[mid], key > A[mid], key < A[mid].
Infatti, dato che l'array è ordinato, …………..

Analisi algoritmi ricorsivi

Si usano le stesse regole per l'analisi di algoritmi non ricorsivi, tranne che il tempo per le chiamate ricorsive, non conoscendolo esplicitamente, lo lasceremo indicato T(…).

Otterremo così una relazione di ricorrenza per T(n), da risolvere in risolvere in seguito, con i metodi che studieremo (o avete studiato).

Ricerca binaria: algoritmo ricorsivo

Ricerca di **key** in **A[1..N]** dall'indice **first** all'indice **last**.
Taglia n = last-first+1

```
ricerca_binaria_ricorsiva (A[1..N], first, last, key) {
if (first > last) return «non c'è»;
        else {
                mid = (first + last) / 2;
                if (key = A[mid]) return mid;
                else
                if (key > A[mid])
                return  ricerca_binaria_ricorsiva (A, mid+1, last, key);
                else
                return ricerca_binaria_ricorsiva (A, first,
                }
}
```

Caso base: $\Theta(1)$

Ulteriore tempo di calcolo: $\Theta(1)$

Non $O(\lfloor n/2 \rfloor)$!

Eventuale chiamata ricorsiva su $\lfloor n/2 \rfloor$ elementi: $T(\lfloor n/2 \rfloor)$

A Recurrence Relation for Binary Search

Def. T(n) = number of comparisons to run Binary Search on an input of size n.

Binary Search recurrence.

$$T(n) \le \begin{cases} \Theta(1) & \text{if } n = 1 \\ \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve left or right half}} + \underbrace{\Theta(1)}_{\text{comparison}} & \text{otherwise} \end{cases}$$

Solution. $T(n) = O(\log_2 n)$ (vedremo nelle prossime lezioni)

(T(n) constant in the best case).

Ordinamento

INPUT: un insieme di n oggetti $a_1, a_2, \ldots, a_n$
presi da un dominio totalmente ordinato secondo $\leq$

OUTPUT: una permutazione degli oggetti $a'_1, a'_2, \ldots, a'_n$ tale che
$$a'_1 \leq a'_2 \leq \ldots \leq a'_n$$

Applicazioni:

- Ordinare alfabeticamente lista di nomi, o insieme di numeri, o insieme di compiti d'esame in base a cognome studente
- Velocizzare altre operazioni (per es. è possibile effettuare ricerche in array ordinati in tempo O(log n))
- Subroutine di molti algoritmi (per es. *greedy*)
- ….

Sorting

Sorting: Given n elements, rearrange in ascending order.

Obvious sorting applications.

- List files in a directory.
- Organize an MP3 library.
- List names in a phone book.
- Display Google PageRank results.

Problems become easier once sorted.

- Find the median.
- Find the closest pair.
- **Binary search in a database.**
- Identify statistical outliers.
- Find duplicates in a mailing. list

Non-obvious sorting applications.

- Data compression.
- Computer graphics.
- **Interval scheduling**.
- Computational biology.
- **Minimum spanning tree**.
- Supply chain management.
- Simulate a system of particles.
- Book recommendations on Amazon.
- Load balancing on a parallel computer.
- . . .

Algoritmi per l'ordinamento

Data l'importanza, esistono svariati algoritmi di ordinamento, basati su tecniche diverse:

Insertionsort
Selectionsort
Heapsort
Mergesort
Quicksort
Bubblesort
Countingsort
…..

Ognuno con i suoi aspetti positivi e negativi.

Il Mergesort e il Quicksort sono entrambi basati sulla tecnica Divide et Impera, ma risultano avere differenti prestazioni

Mergesort

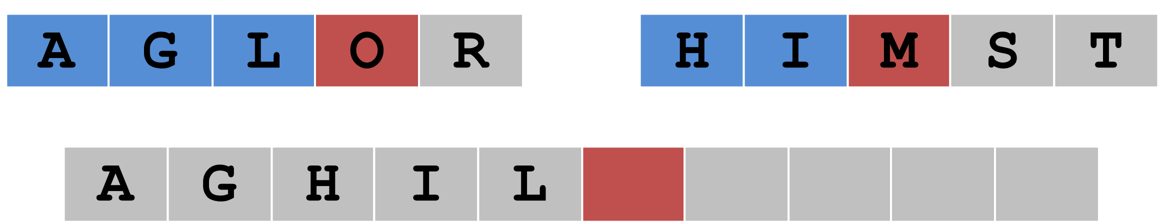
Mergesort su una sequenza ***S*** con ***n*** elementi consiste di tre passi:

1. Divide: separa ***S*** in due sequenze ***S***1 e ***S***2, ognuna di circa ***n***/2 elementi;
2. Ricorsione: ricorsivamente ordina ***S***1 e ***S***2
3. Conquer (impera): fondi ***S***1 e ***S***2 in un'unica sequenza *ordinata*

Conquer: Merging

- Merging. Combine two pre-sorted lists into a sorted whole.

- How to merge efficiently?
  - Linear number of comparisons. $T_{MERGE}(n) = \Theta(n)$
  - Use temporary array.



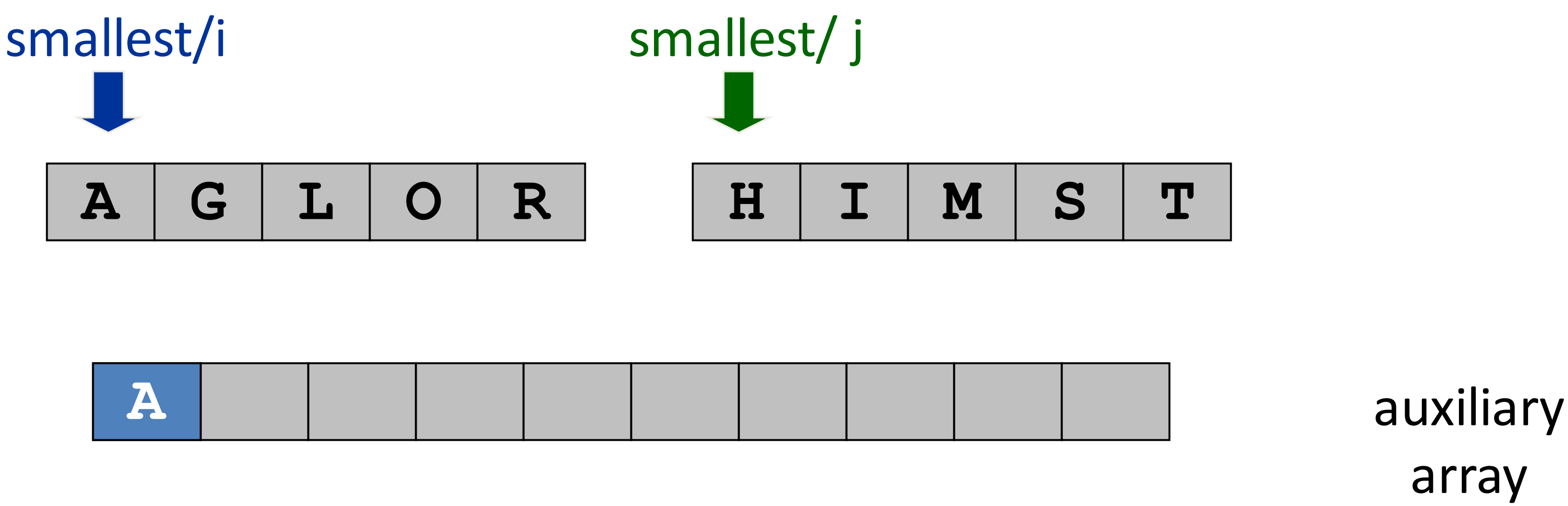
Merging

Merge.

Keep track of smallest element in each sorted half.

Insert smallest of two elements into auxiliary array.

Repeat until done.



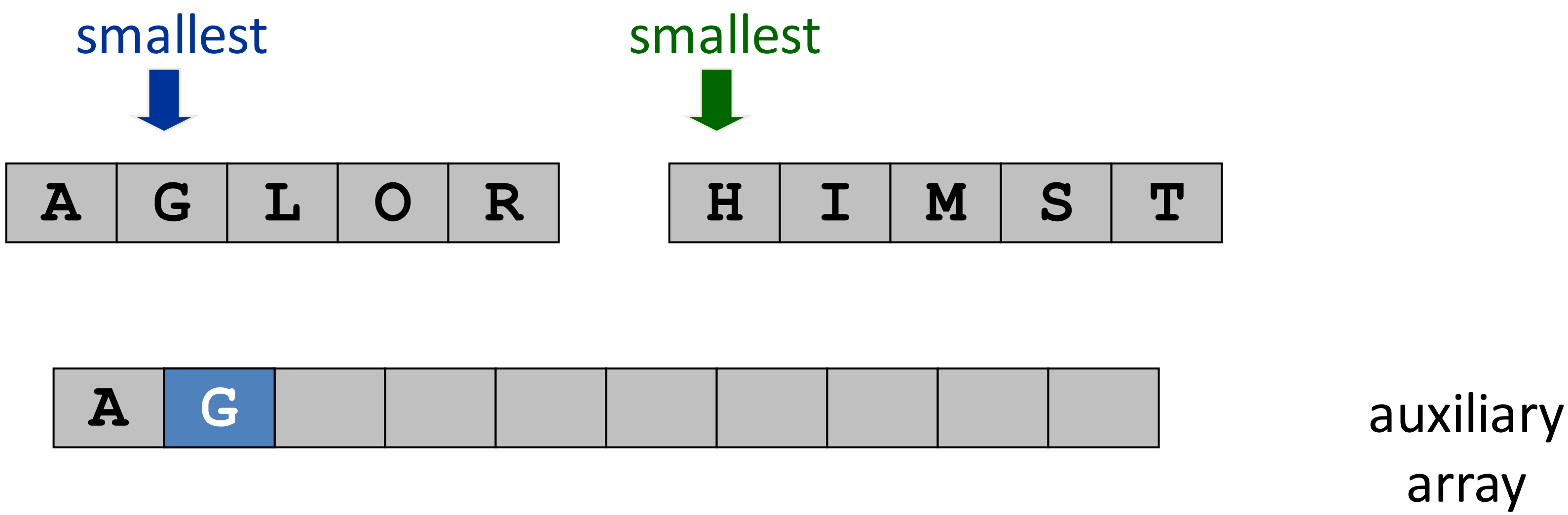
Merging

Merge.

Keep track of smallest element in each sorted half.

Insert smallest of two elements into auxiliary array.

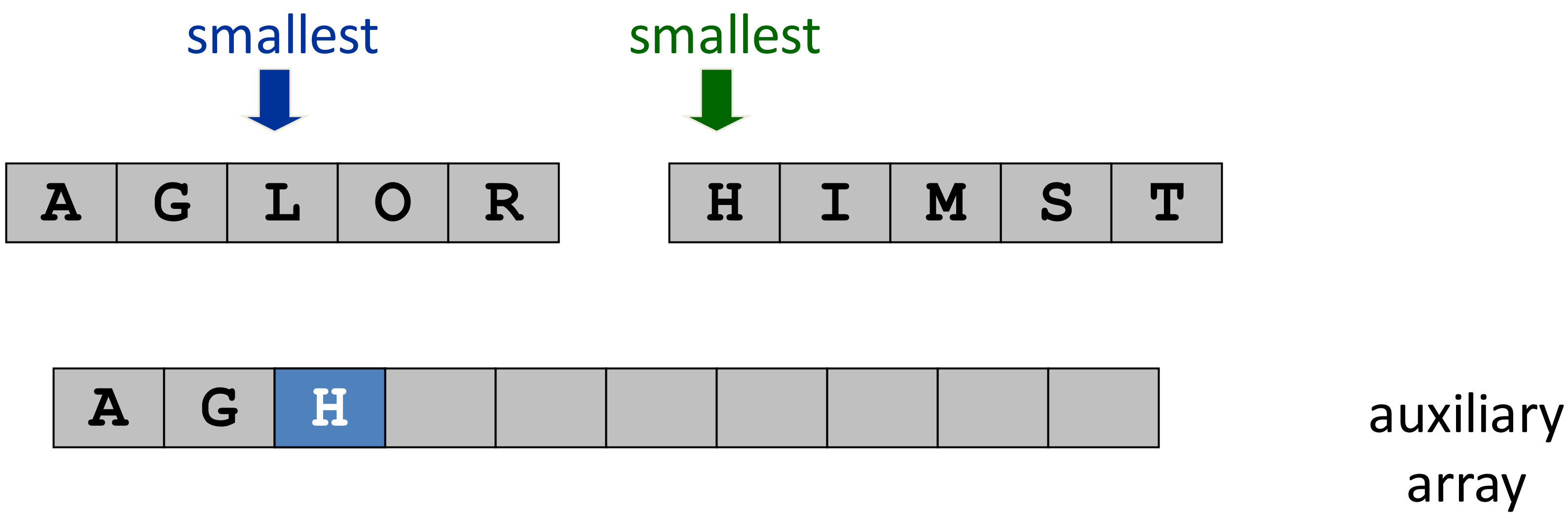
Repeat until done.



Merging

Merge.

- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



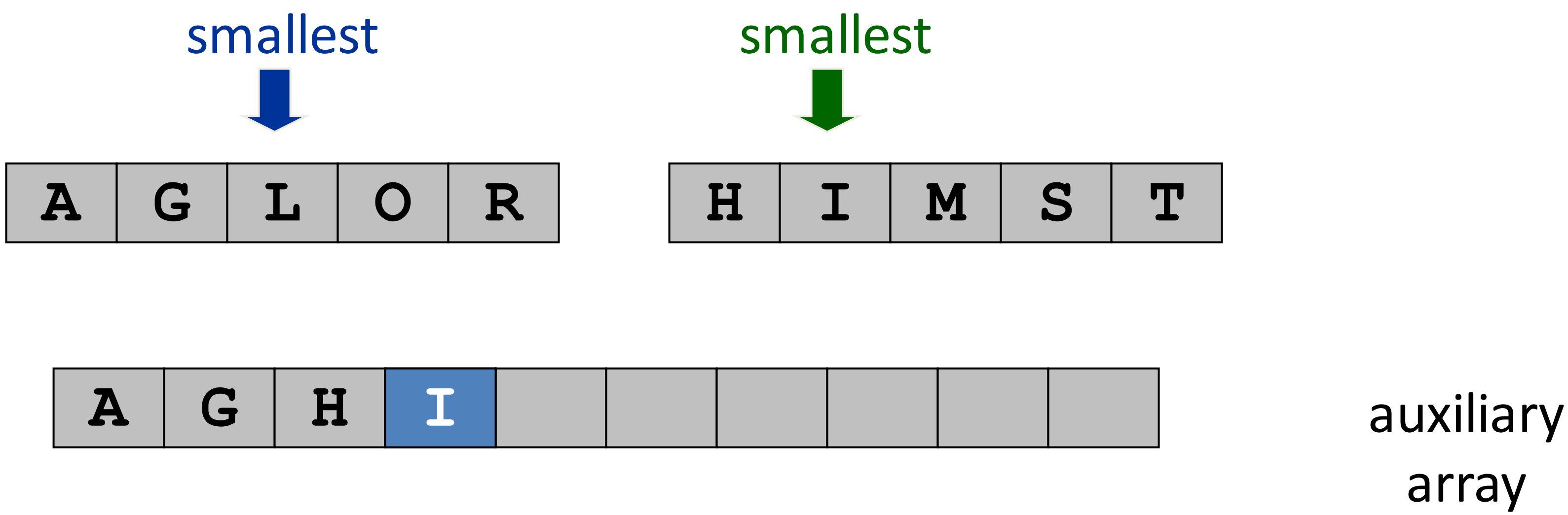
Merging

Merge.

Keep track of smallest element in each sorted half.

Insert smallest of two elements into auxiliary array.

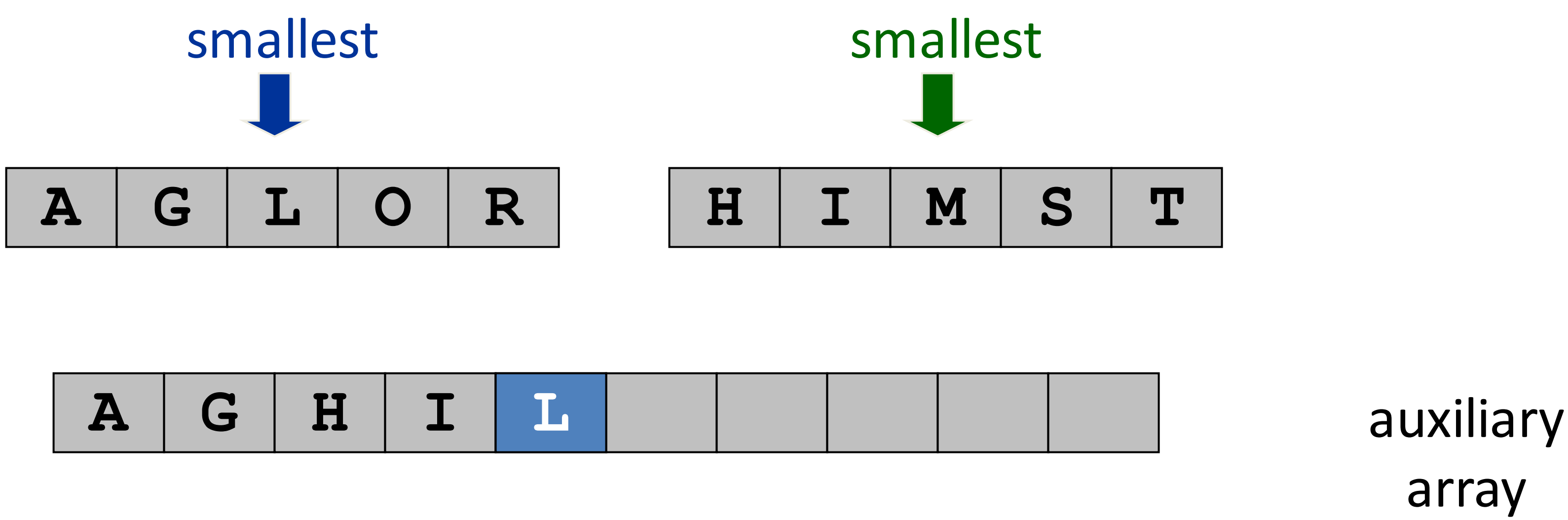
Repeat until done.



Merging

Merge.

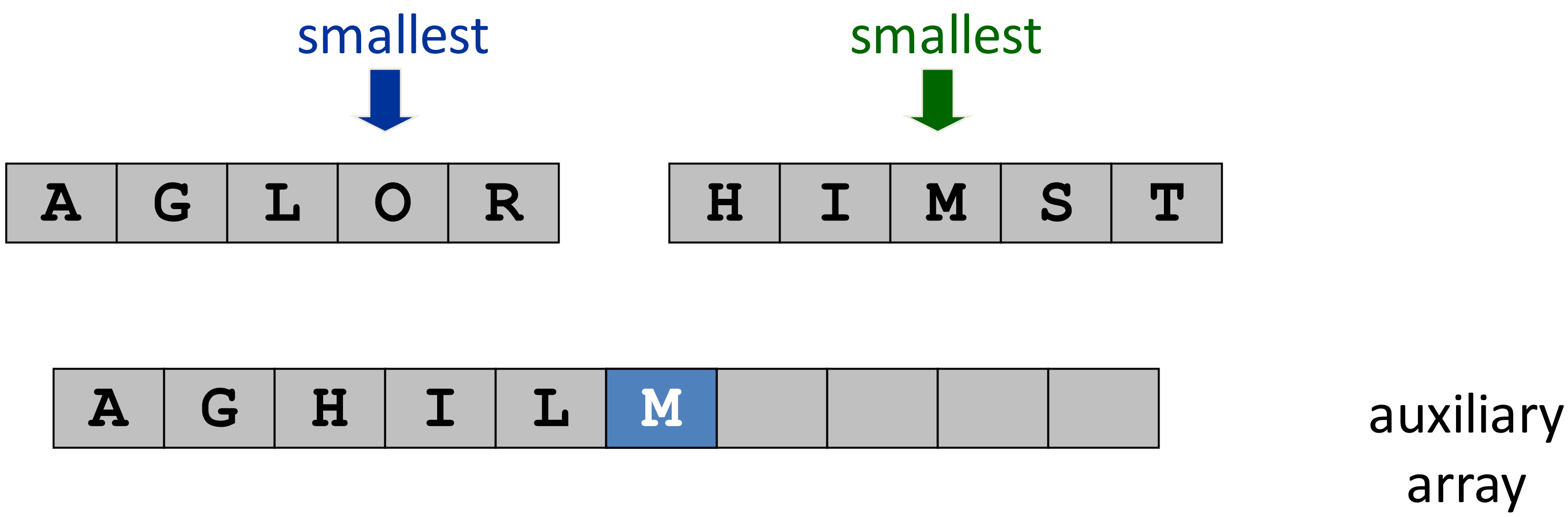
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Merge.

- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



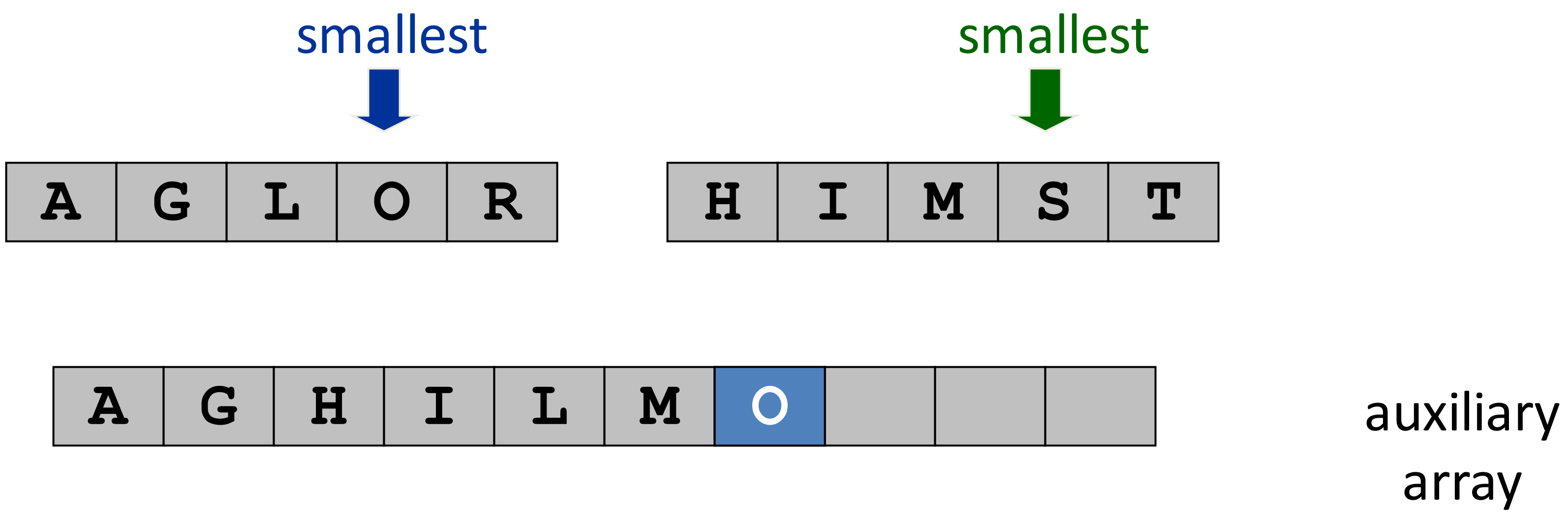
Merging

Merge.

Keep track of smallest element in each sorted half.

Insert smallest of two elements into auxiliary array.

Repeat until done.



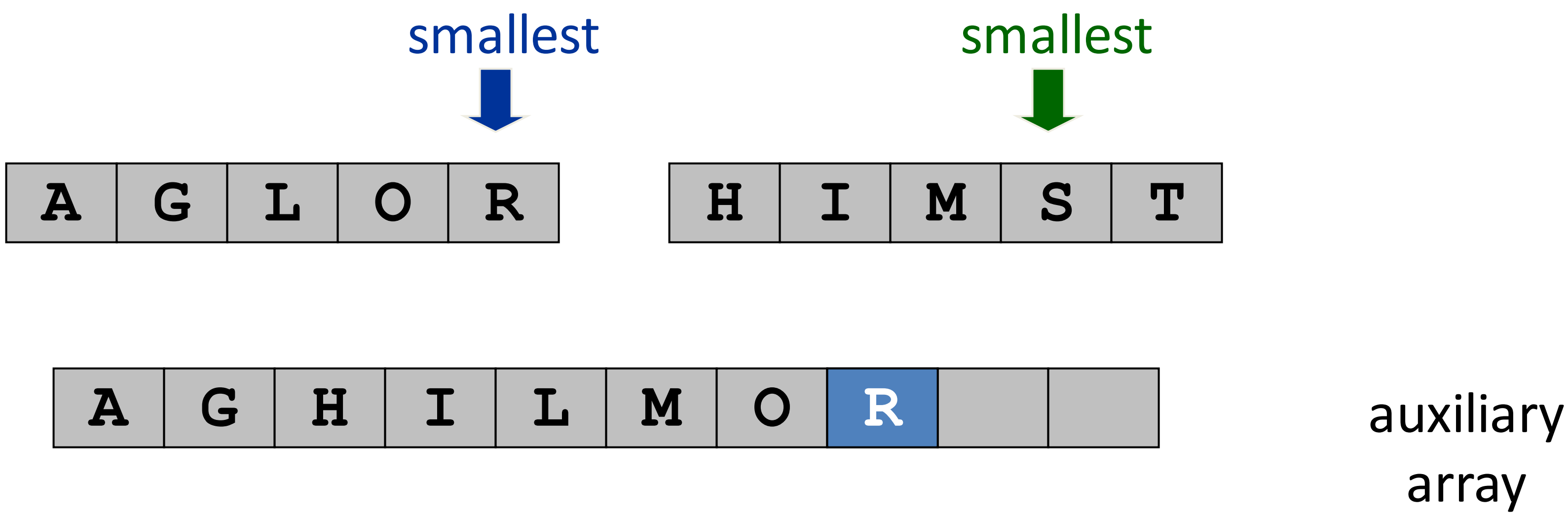
Merging

Merge.

Keep track of smallest element in each sorted half.

Insert smallest of two elements into auxiliary array.

Repeat until done.



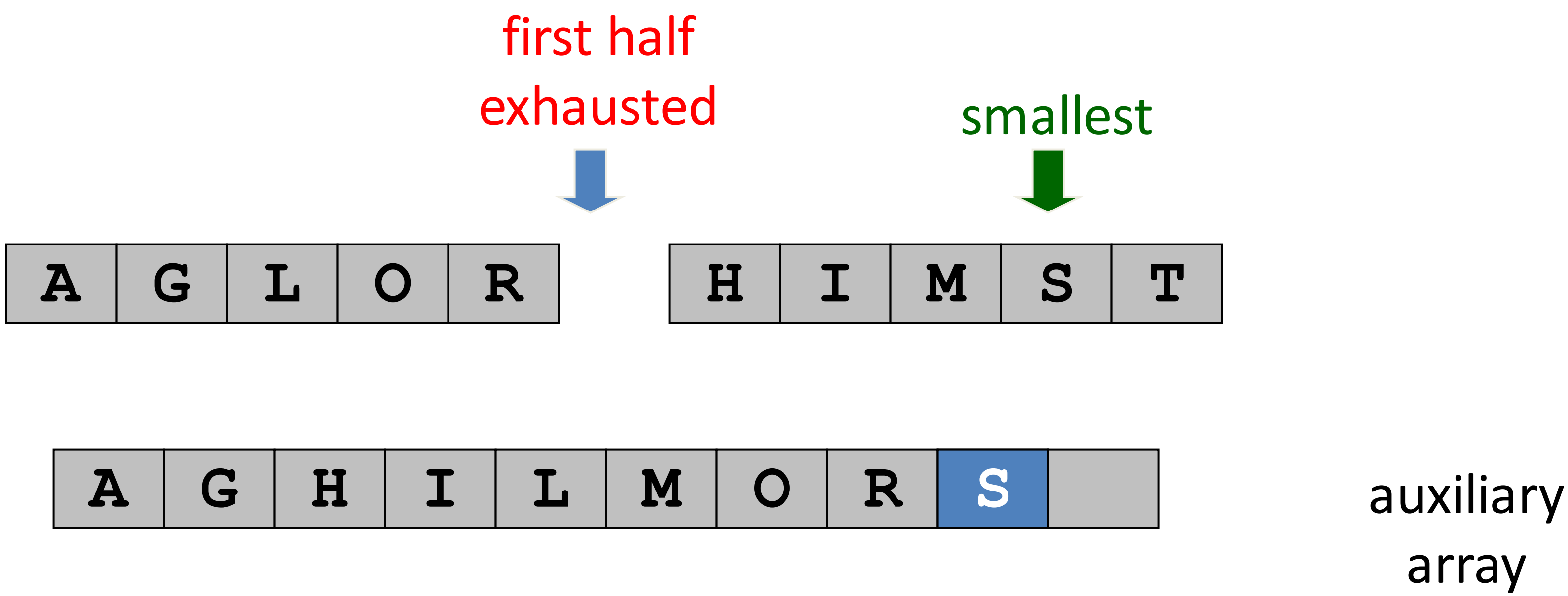
Merging

Merge.

Keep track of smallest element in each sorted half.

Insert smallest of two elements into auxiliary array.

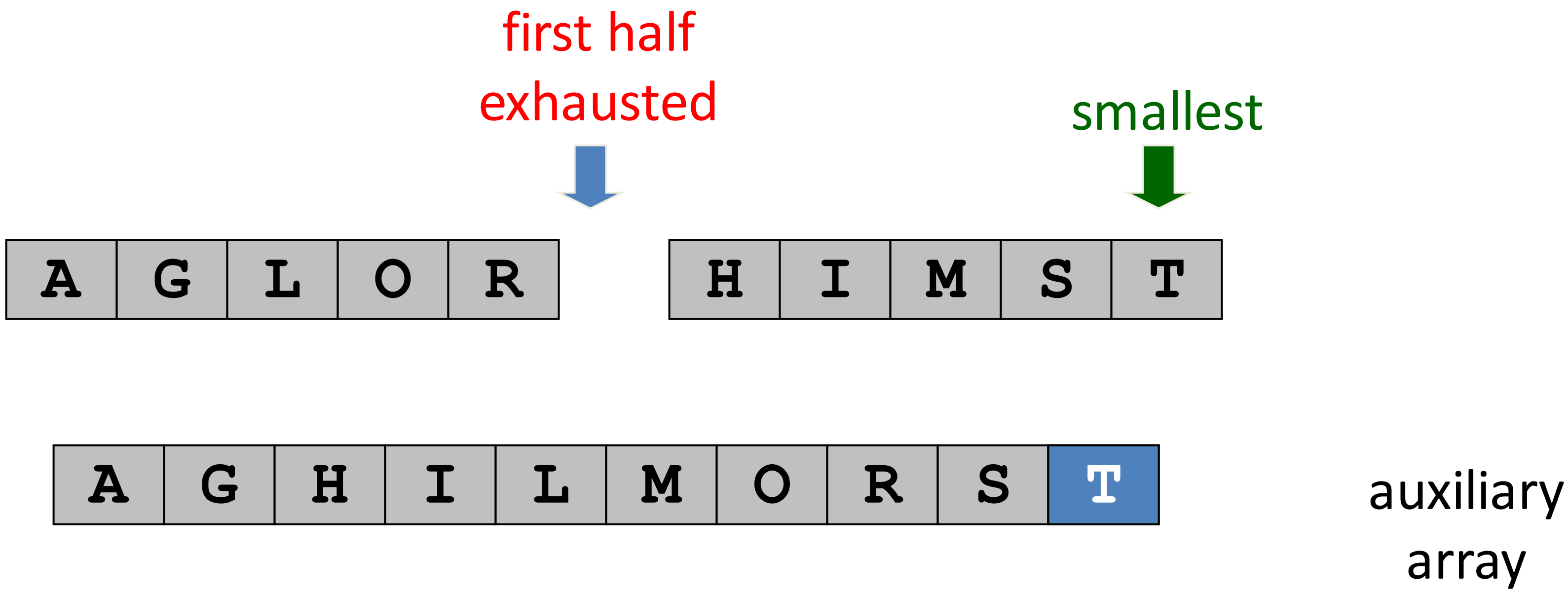
Repeat until done.



Merging

Merge.

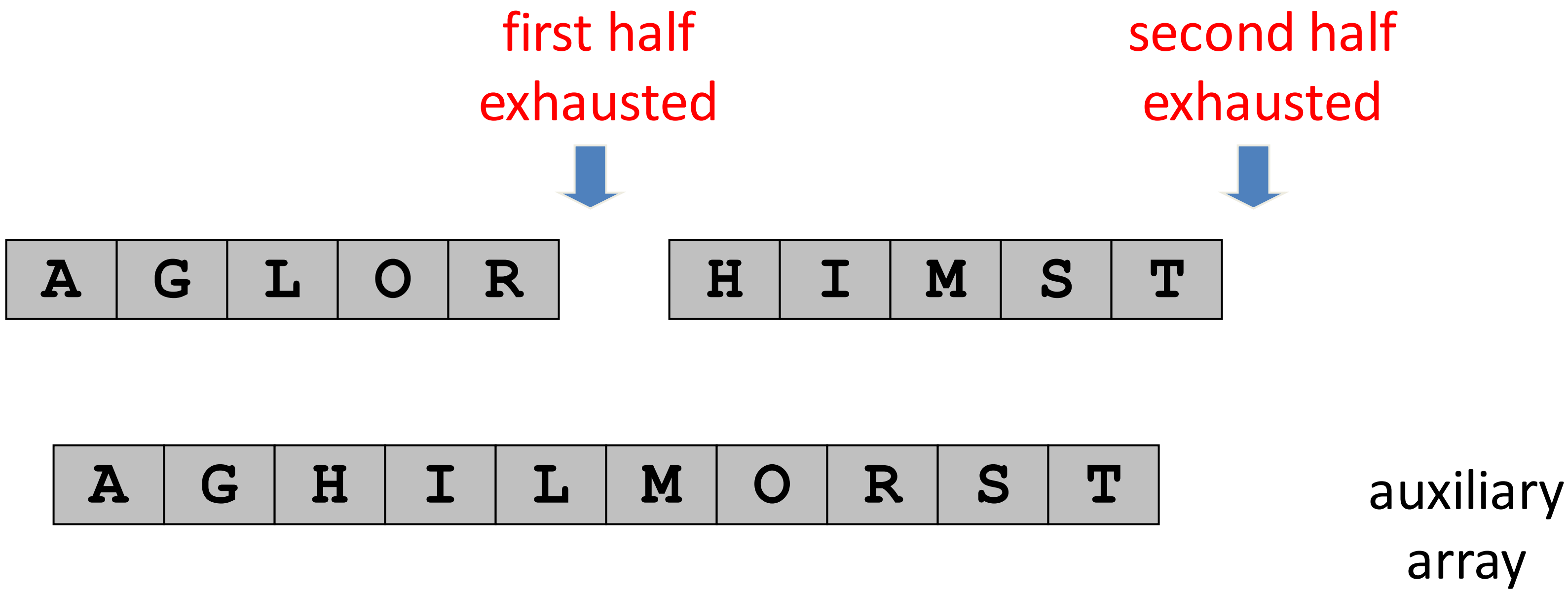
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

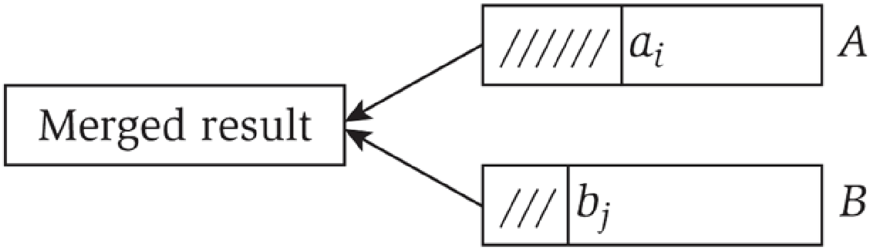
Merge.

- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merge

Merge. Combine two sorted lists $A = a_1, a_2, \ldots, a_n$ with $B = b_1, b_2, \ldots, b_n$ into sorted whole.




```
i = 1, j = 1
while (both lists are nonempty) {
   if (aᵢ ≤ bⱼ) append aᵢ to output list and increment i
   else(aᵢ > bⱼ)append bⱼ to output list and increment j
}
append remainder of nonempty list to output list
```

Correttezza di Merge

Fonde due array ordinati A e B in un unico array ordinato C.

A =

A	G	L	O	R

B =

H	I	M	S	T

C =

A	G	H	I	L					

Perché funziona?

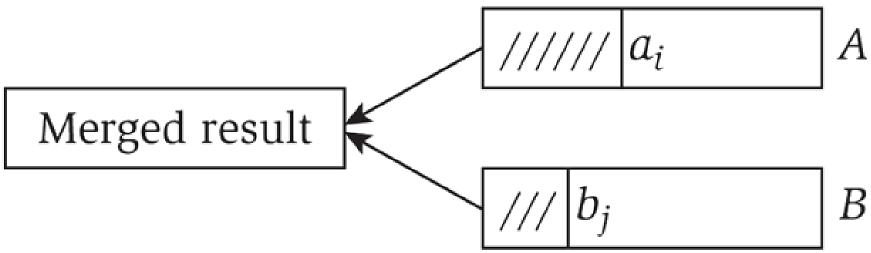
Ad ogni iterazione:

la parte verde del vettore risultante C è ordinata e contiene tutti gli elementi già considerati nella parte blu di A e B, mentre i prossimi elementi di A e B (in arancione) sono entrambi maggiori di quelli nella parte verde e sappiamo sono i minimi fra gli elementi rimanenti grigi (perché A e B sono ordinati).

Tale affermazione è vera all'inizio e si mantiene vera ad ogni iterazione (per induzione); alla fine implica la correttezza di Merge.

Running time for Merge (senza Sort!)

Merge. Combine two sorted lists $A = a_1, a_2, \ldots, a_n$ with $B = b_1, b_2, \ldots, b_n$ into sorted whole.




```
i = 1, j = 1
while (both lists are nonempty) {
   if (a_i ≤ b_j) append a_i to output list and increment i
   else(a_i > b_j)append b_j to output list and increment j
}
append remainder of nonempty list to output list
```

Claim. Merging two lists of size n takes $\Theta(n)$ time.

Pf. After each comparison, the length of output list increases by 1.

MERGE(A, p, q, r) in loco

Nel seguito useremo una procedura MERGE(A, p, q, r) che «fonde» correttamente le sequenze A[p,…, q] e A[q+1, …, r] in tempo lineare.

Potremo considerare che tale procedura prima copi la parte di array da p a q e quella di q+1 ad r, in due nuovi array ausiliari distinti e poi operi come visto prima.

Oppure, potremo considerare che MERGE(A, p, q, r) sia l'algoritmo di Kronrud del 1969, che esegue la «fusione» in loco, cioè senza utilizzo di array ausiliari, sempre in tempo lineare. Più complicato non lo vedremo.

Mergesort

```
MERGE-SORT(A, p, r)
1 if p < r
2    then q ← ⌊(p + r)/2⌋
3         MERGE-SORT(A, p, q)
4         MERGE-SORT(A, q + 1, r)
5         MERGE(A, p, q, r)
```

Qui supponiamo che MERGE(A, p, q, r) fonda le sequenze A[p,…, q] e A[q+1, …, r] in loco. Quindi la divisione dell'array in due metà si fa semplicemente calcolando q nella linea 2 (con costo costante). Altrimenti, la divisione avrebbe comportato l'allocazione di due sotto-array e la copia degli elementi (con costo lineare).
Anche MERGE-SORT sarà in loco, non avrà quindi bisogno di memoria ausiliaria.

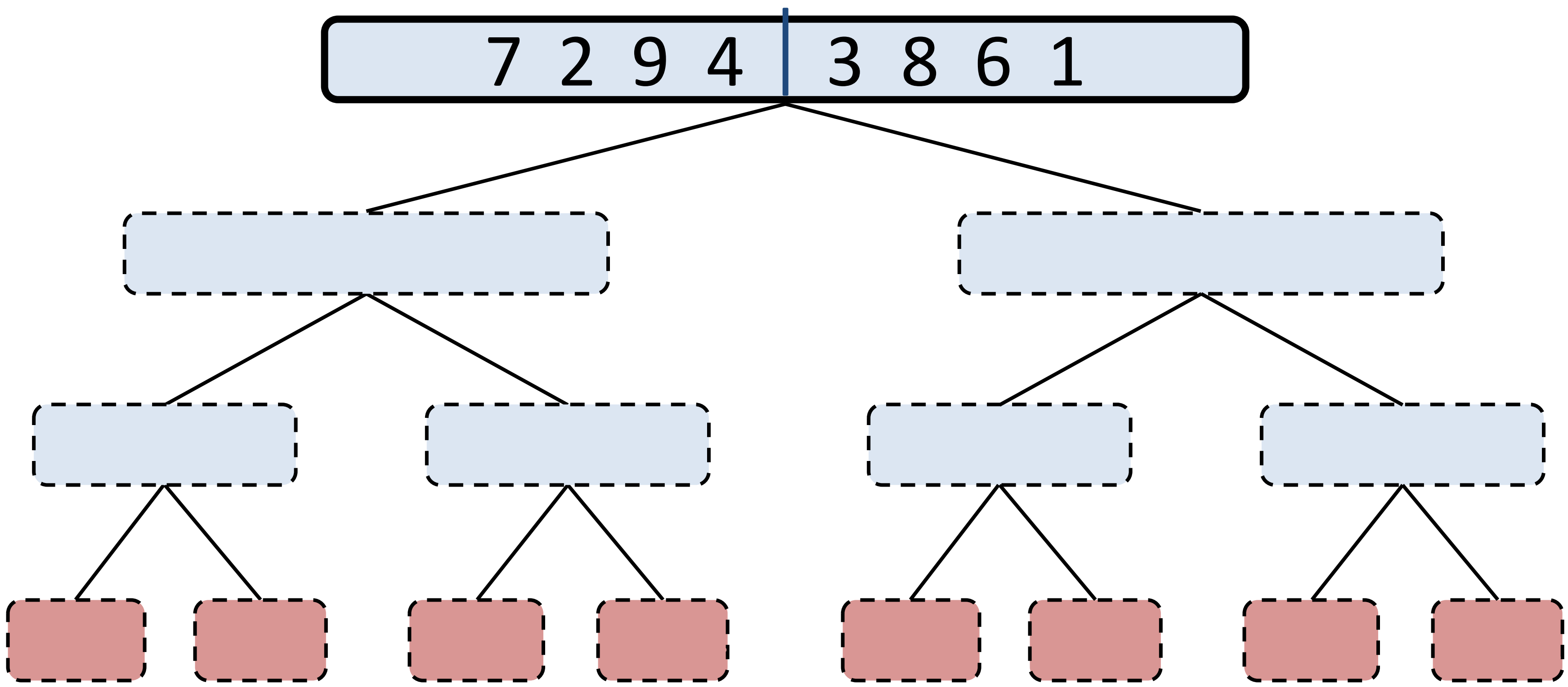
Memento

- Ricordare che in un algoritmo ricorsivo non si può procedere oltre una chiamata ricorsiva se questa non è stata completata del tutto.

- Quindi nel `MERGE-SORT`, l'esecuzione della linea 4 comincia una volta finita l'esecuzione della linea 3, cioè completata la chiamata ricorsiva `MERGE-SORT(A,p,q)` con tutte le sue sotto-chiamate.

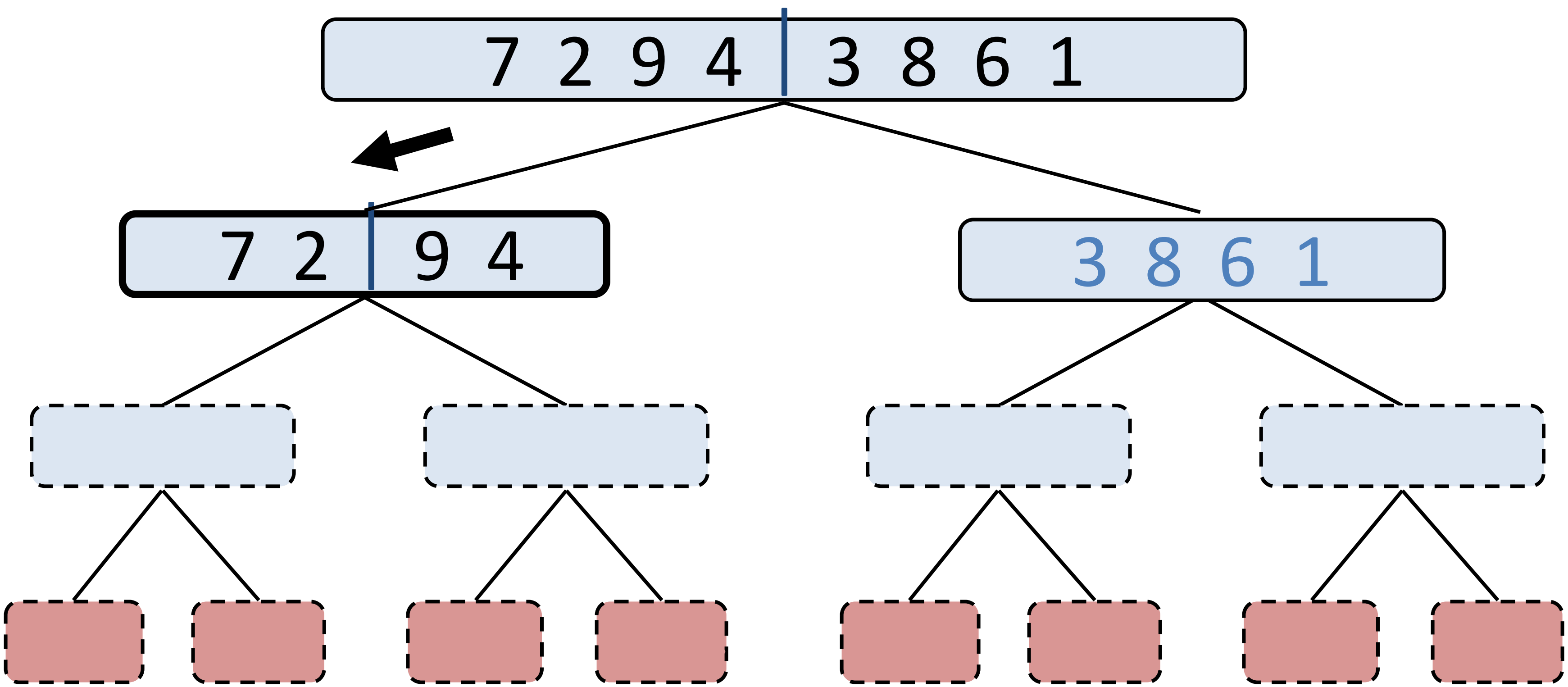
Esempio di esecuzione di MergeSort

- Divide



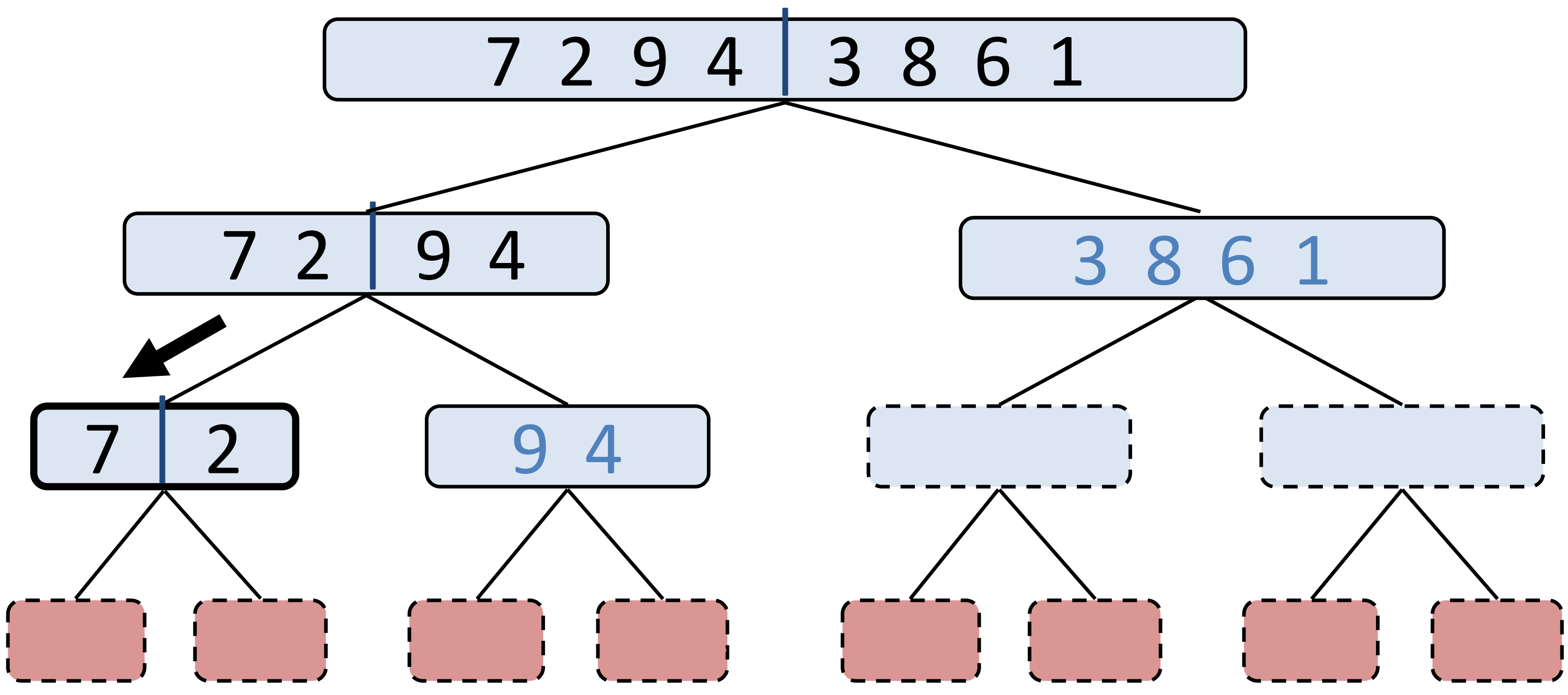
Esempio di esecuzione (cont.)

- Chiamata ricorsiva, divide



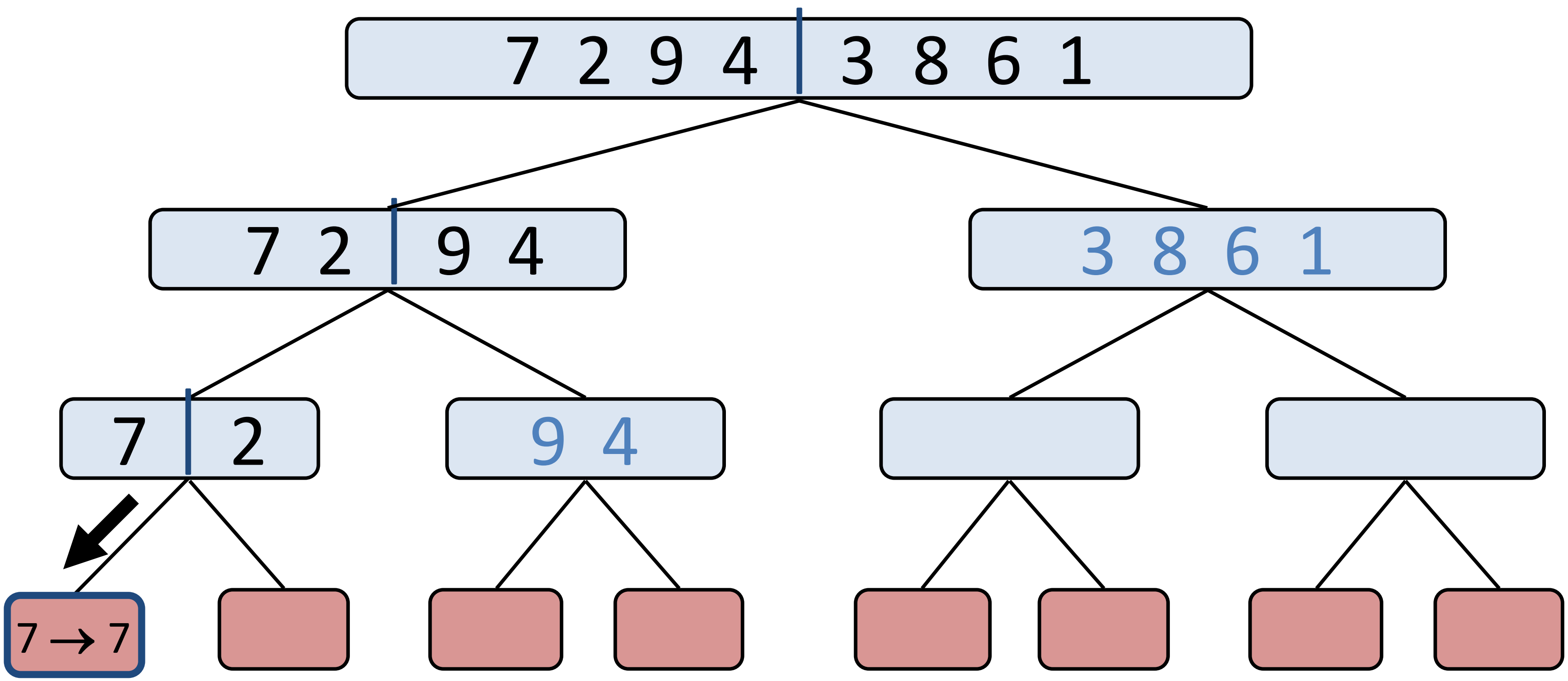
Esempio di esecuzione (cont.)

- Chiamata ricorsiva, divide



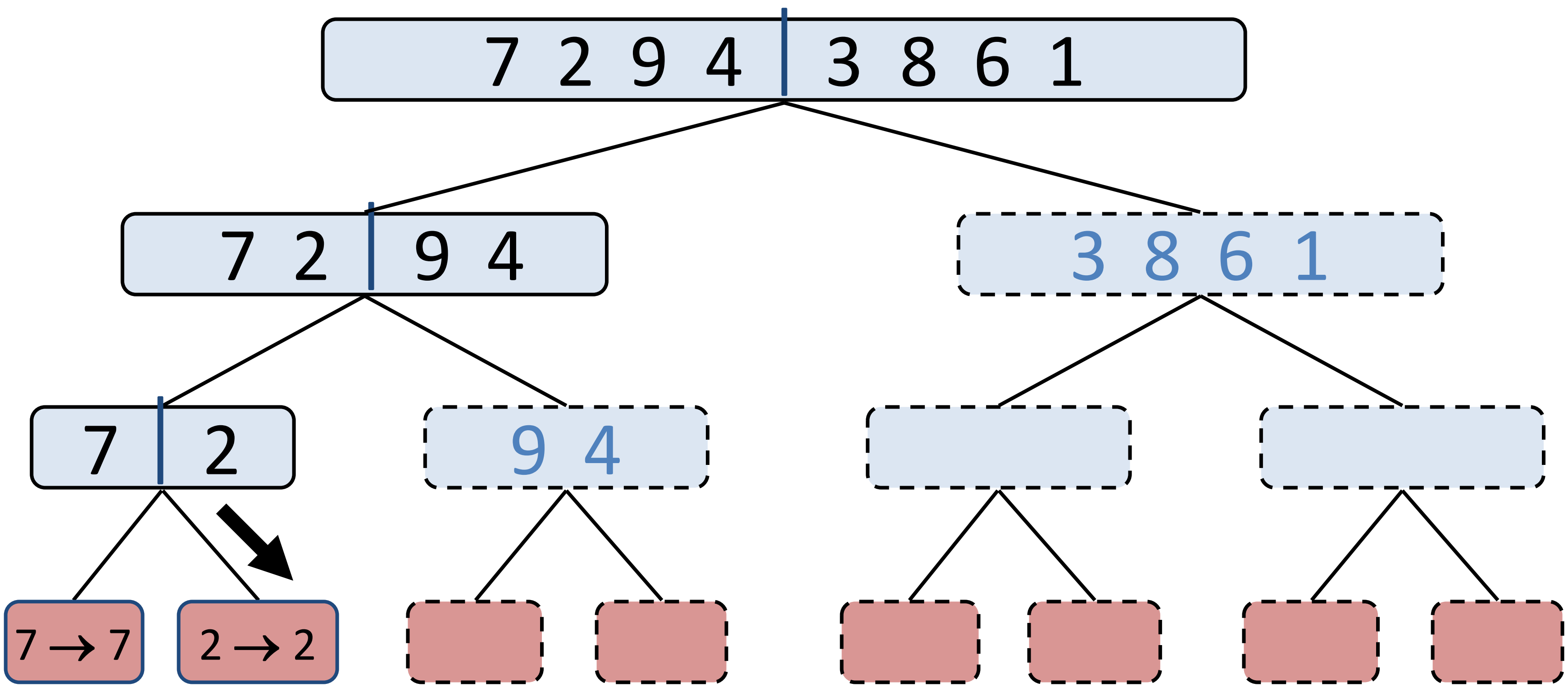
Esempio di esecuzione (cont.)

- Chiamata ricorsiva: caso base



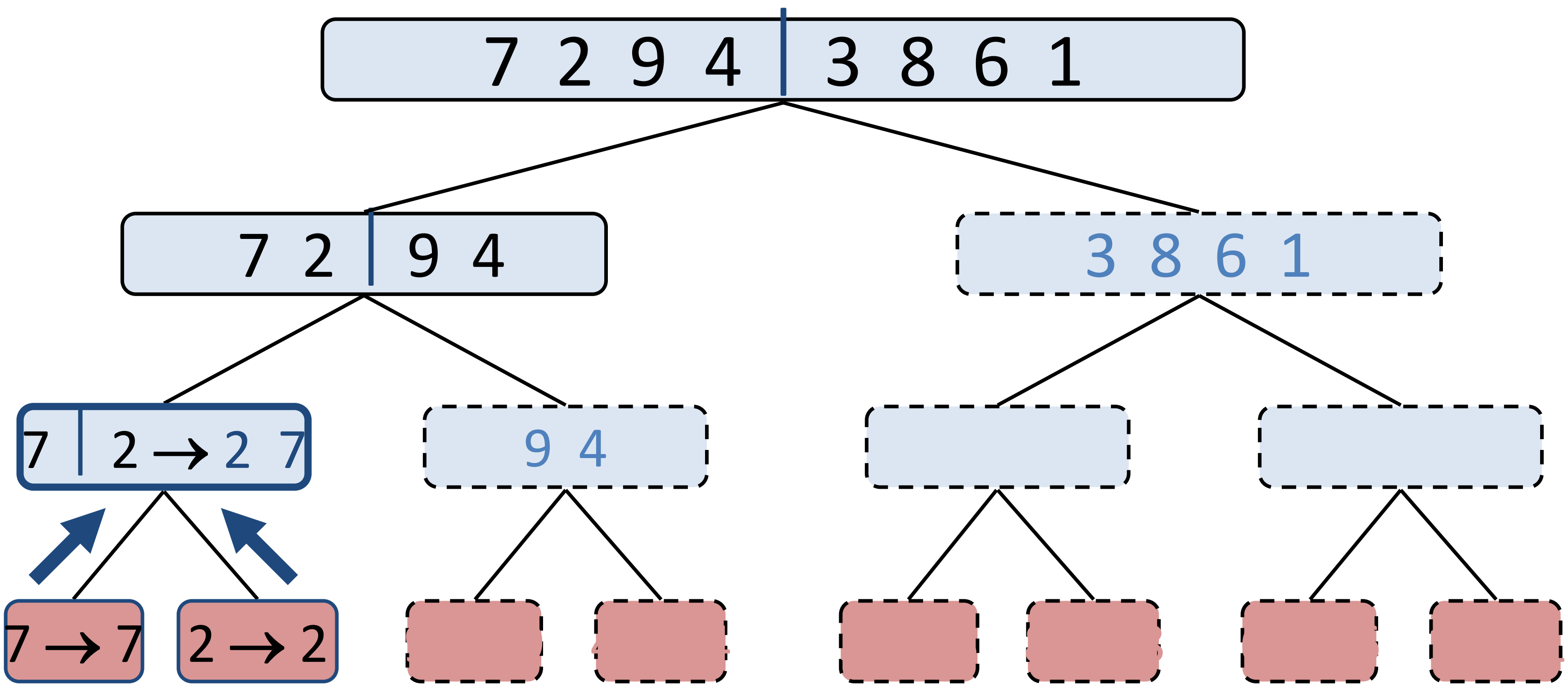
Esempio di esecuzione (cont.)

- Chiamata ricorsiva: caso base



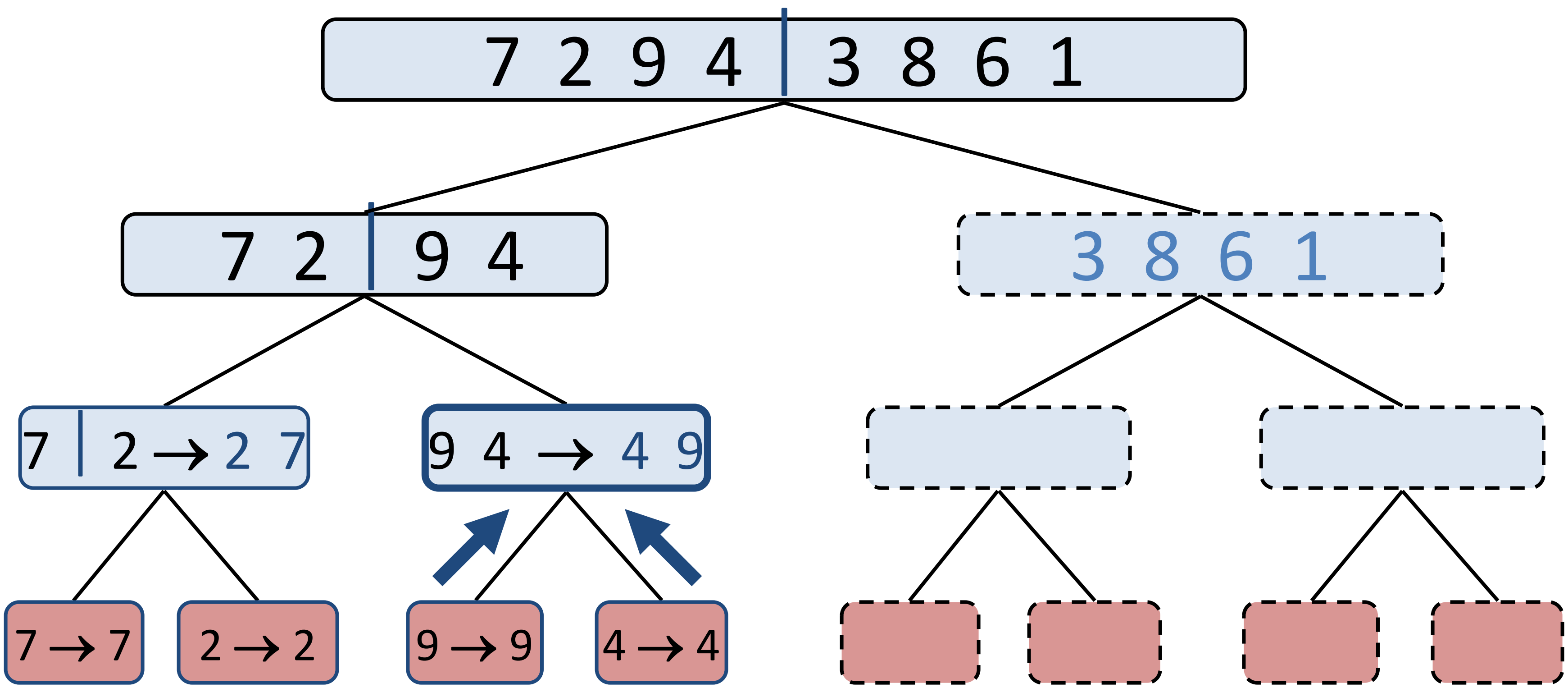
Esempio di esecuzione (cont.)

- Merge



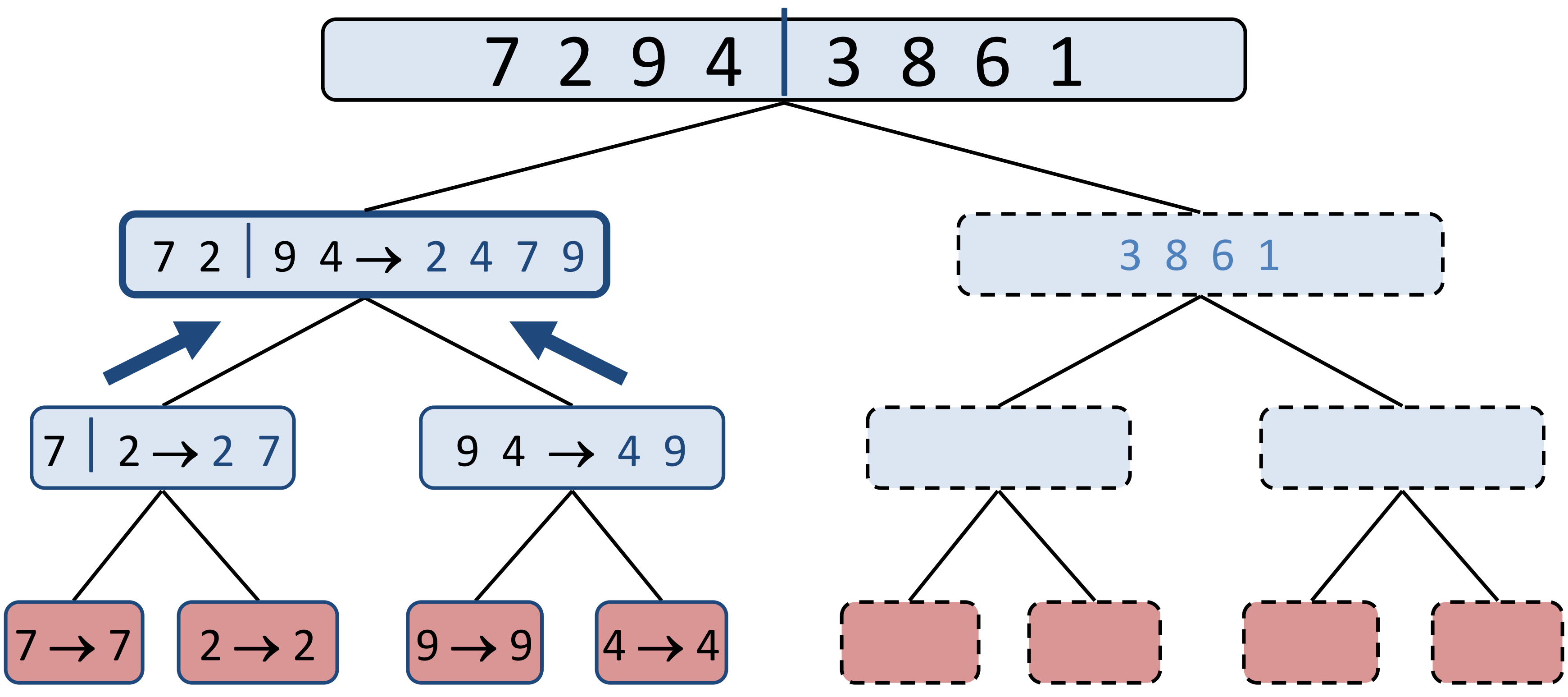
Esempio di esecuzione (cont.)

- Chiamata ricorsiva, …, caso base, merge



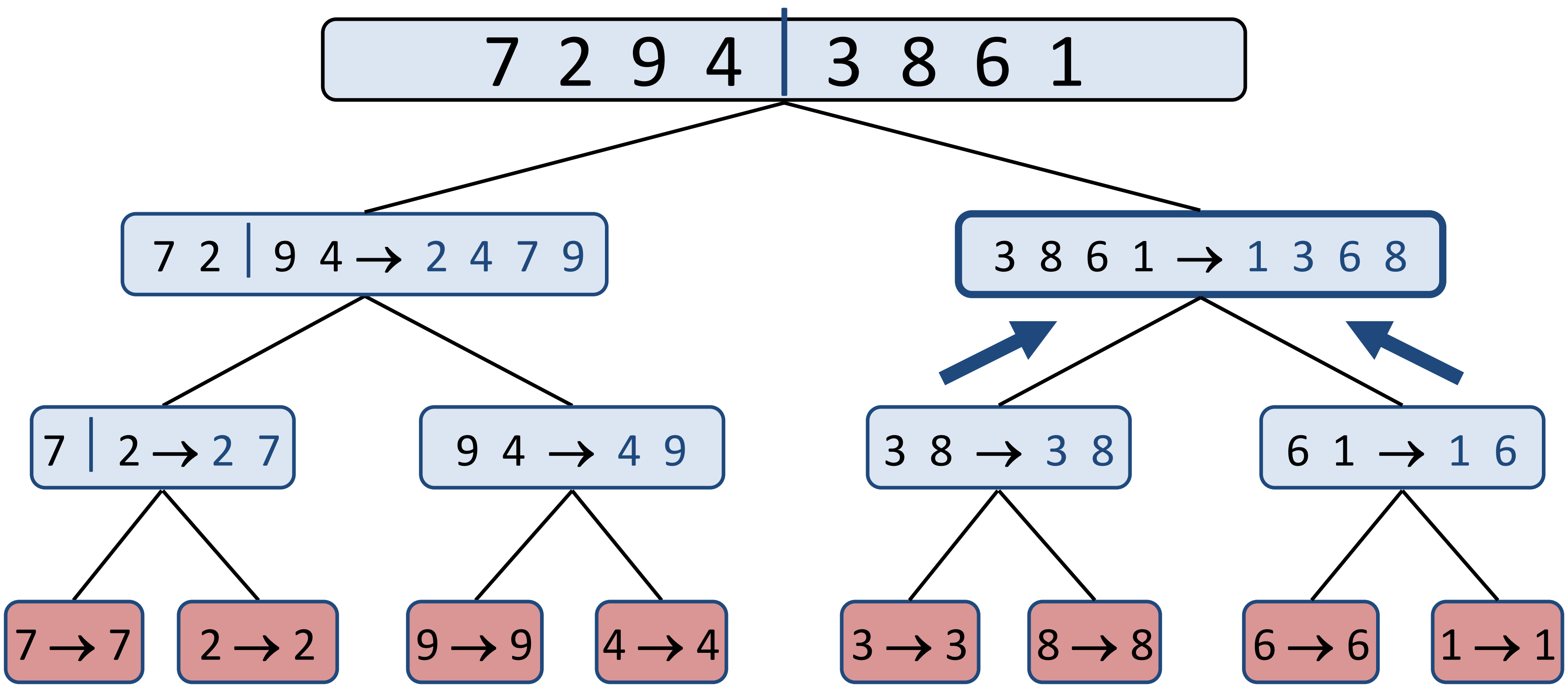
Esempio di esecuzione (cont.)

- Merge



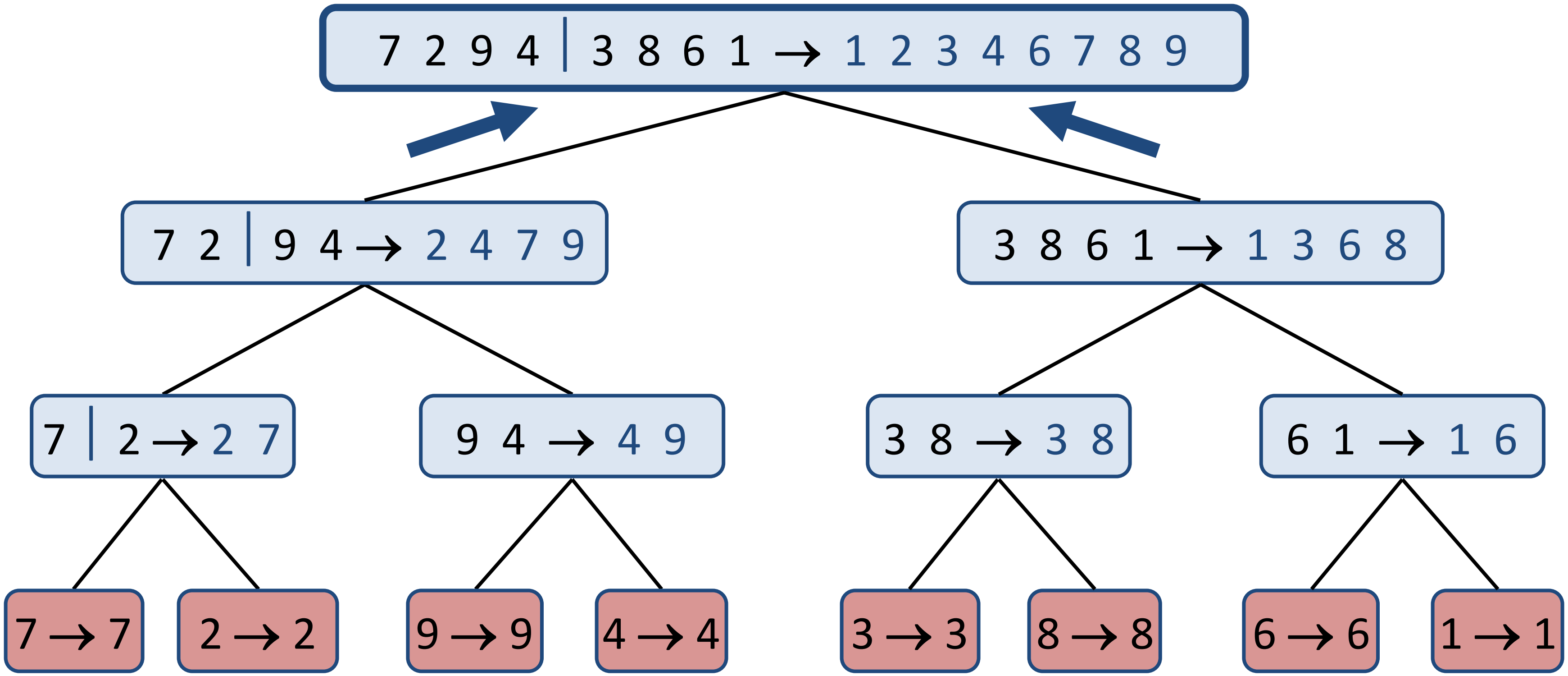
Esempio di esecuzione (cont.)

- Chiamata ricorsiva, …, merge, merge



Esempio di esecuzione (fine)

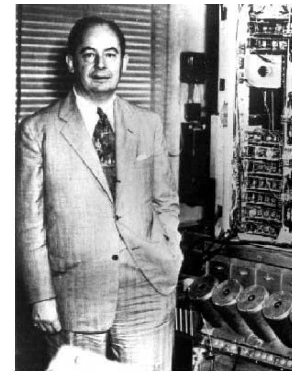
- Ultimo Merge



Mergesort

Mergesort.

- Divide array into two halves.
- Recursively sort each half.
- Merge two halves to make sorted whole.



Jon von Neumann (1945)

A	L	G	O	R	I	T	H	M	S

A	L	G	O	R

I	T	H	M	S

A	G	L	O	R

H	I	M	S	T

Recursively sort: as a black box

A	G	H	I	L	M	O	R	S	T

Correttezza di Mergesort

```
MERGE-SORT(A, p, r)
if p < r
   then q ← ⌊(p + r)/2⌋
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q + 1, r)
        MERGE(A, p, q, r)
```

Perché funziona?

Nel caso base (n=1), un array di 1 elemento è già ordinato e l'algoritmo correttamente non esegue nessuna operazione su di esso.

Nel caso generale di una chiamata principale su n elementi, l'algoritmo ordina correttamente la porzione di array perché:

- Possiamo supporre per induzione che le due chiamate di MERGE-SORT su n/2 elementi restituiscano gli array ordinati
- MERGE chiamato su due array ordinati, fonde correttamente i due array ordinati in un solo array ordinato

Relazione di ricorrenza per `ric-fact`

Determinare la relazione di ricorrenza per il tempo di esecuzione dell'algoritmo ricorsivo per il fattoriale

```
int ric-fact (int n)
     if (n==0) return 1
           else return n * ric-fact(n-1)
```

Visualizzazione

Potete visualizzare l'esecuzione del MergeSort (in loco), e non solo, a questi link

- https://algorithm-visualizer.org/divide-and-conquer/merge-sort
- https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html