ALGORITMI E STRUTTURE DATI

Prof. Manuela Montangero

A.A. 2022/23

ALGORITMI DI ORDINAMENTO: MergeSort

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

MergeSort

USIAMO la TECNICA DIVIDE&IMPERA

- **DIVIDE**: dividi l'array in due sequenze di $n/2$ elementi ciascuna
- **IMPERA**: ordina ciascuna sequenza ricorsivamente
- **COMBINA**: "fondi" le due metà ordinate in un'unica sequenza ordinata (MERGE)

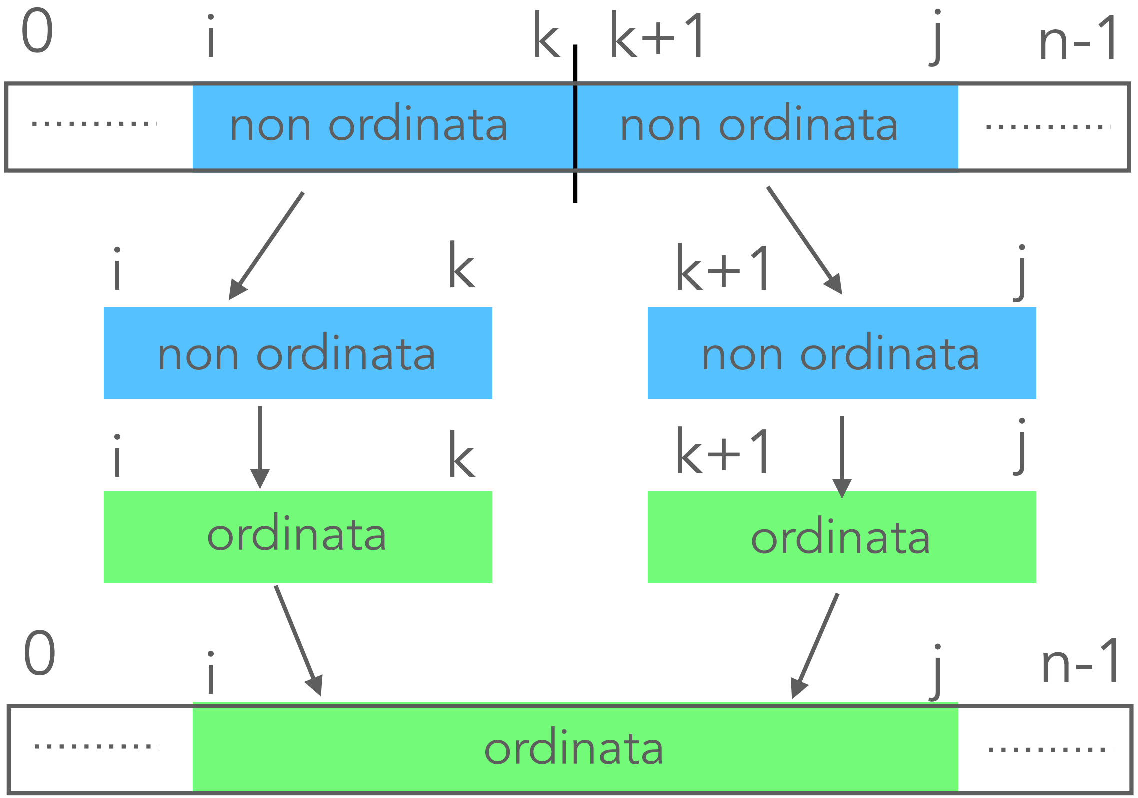
CASO BASE: in quali condizioni il problema è facile da risolvere?

Risposta: quando la sequenza ha un solo elemento (e' già ordinata)

MergeSort

Scriviamo una procedura ricorsiva che ordini una porzione di un array `A`, compresa tra gli indici `i` (a sinistra) e `j` (a destra), estremi inclusi

```
MERGESORT(A,i,j)
  if i < j
   then
    k := ⌊(i+j)/2⌋
    MERGESORT(A,i,k)
    MERGESORT(A,k+1,j)
    MERGE(A,i,k,j)
```

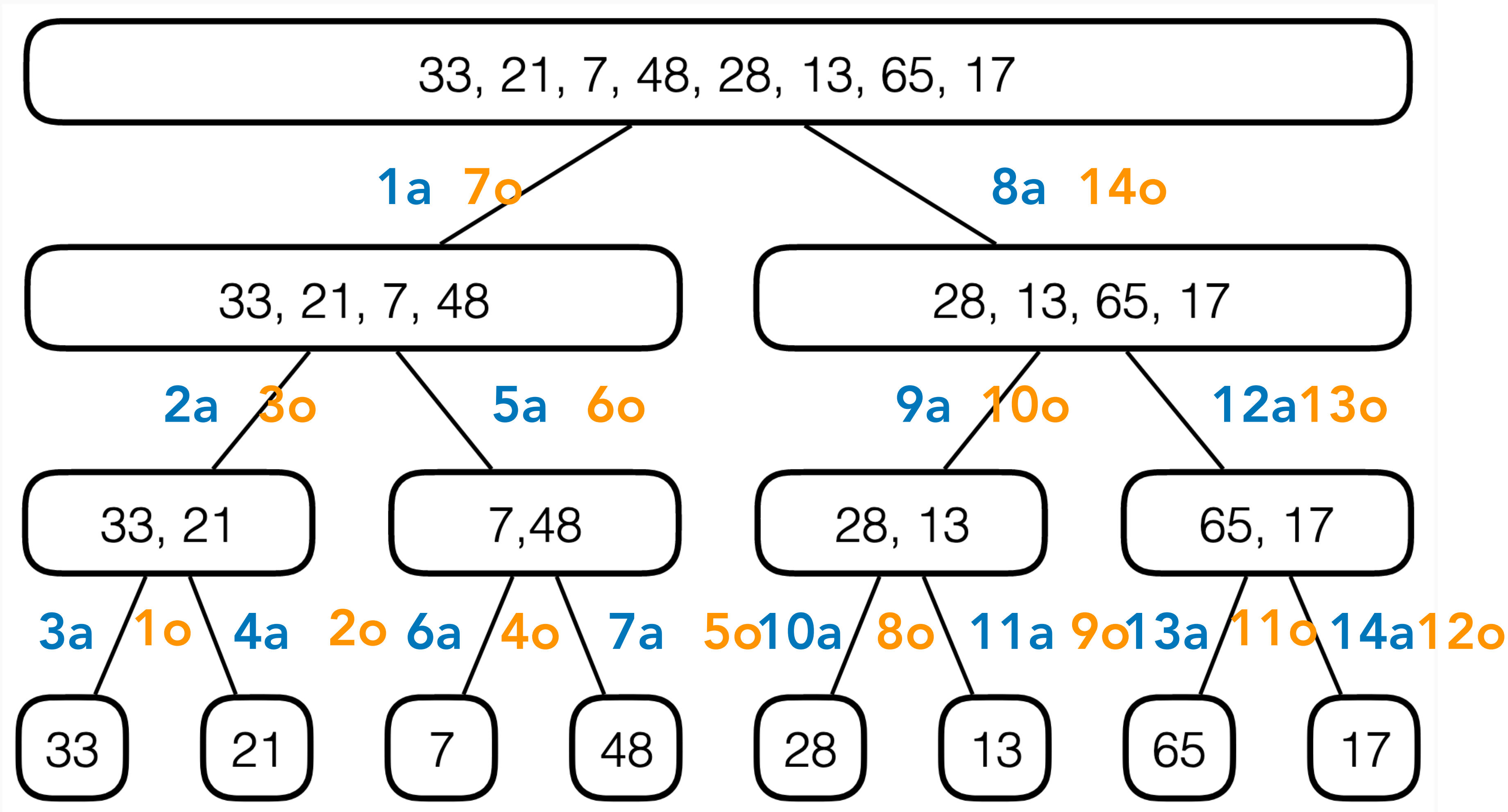


Chiamata principale

`MERGESORT(A,0,n-1)`

MergeSort

ESEMPIO: chiamate ricorsive `Mergesort`



numero chiamata **numero return**

Immagina originale di A. Montresor

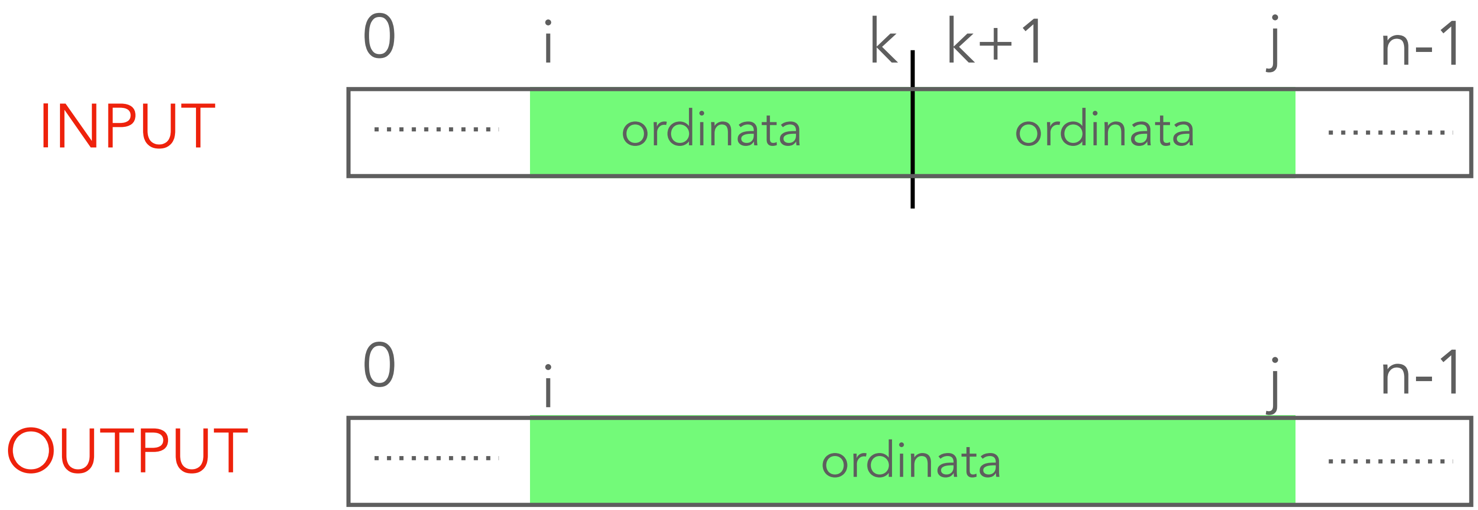
UNIMORE

MergeSort

Scriviamo una procedura (`Merge`) che risolve il seguente problema:

INPUT: due porzioni ordinate dell'array `A`. La prima dall'indice `i`, all'indice `k`, inclusi gli estremi. La seconda dall'indice `k+1` all'indice `j`, estremi inclusi.

OUTPUT: la porzione di `A` che va dall'indice i all'indice j ordinata.



 UNIMORE

MergeSort

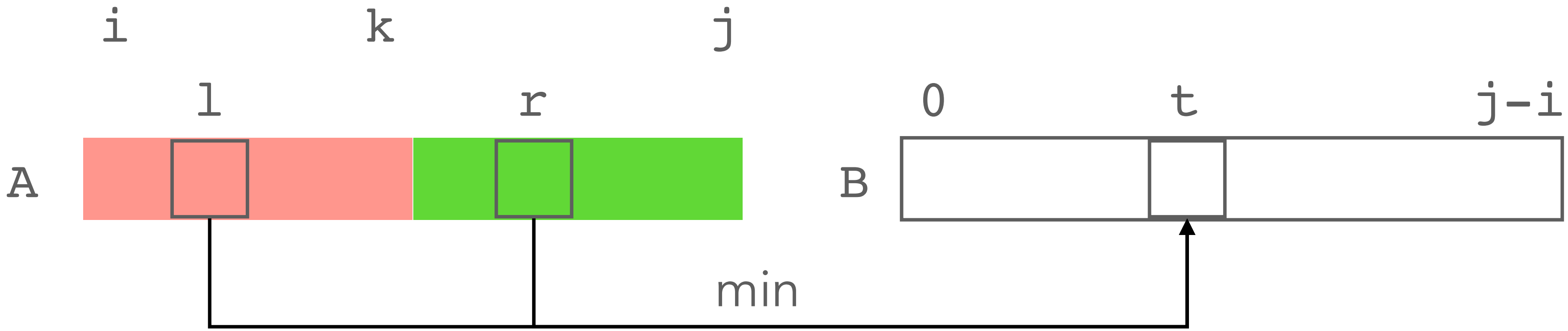
IDEA per procedura Merge:

- Usiamo un array di appoggio **`B[0..j-i]`** con `j-i+1` elementi
- Usiamo tre indici **`l`** (left), **`r`** (right) e **`t`** per scorrere, rispettivamente, la porzione di **`A`** di sinistra **`A[i..k]`**, la porzione di **`A`** di destra **`A[k+1..j]`**, e **`B`**

MergeSort

IDEA per procedura Merge:

- Usiamo un array di appoggio **`B[0..j-i]`** con `j-i+1` elementi
- Usiamo tre indici **`l`** (left), **`r`** (right) e **`t`** per scorrere, rispettivamente, la porzione di **`A`** di sinistra **`A[i..k]`**, la porzione di **`A`** di destra **`A[k+1..j]`**, e **`B`**
- Ad ogni passo, confrontiamo **`A[l]`** e **`A[r]`** e copiamo il più piccolo in **`B[t]`**, poi incrementiamo **`t`** e **`l`** o **`r`**, opportunamente

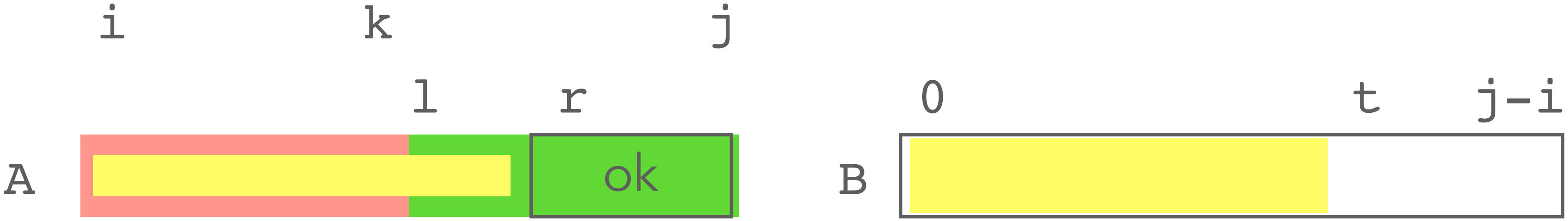


MergeSort

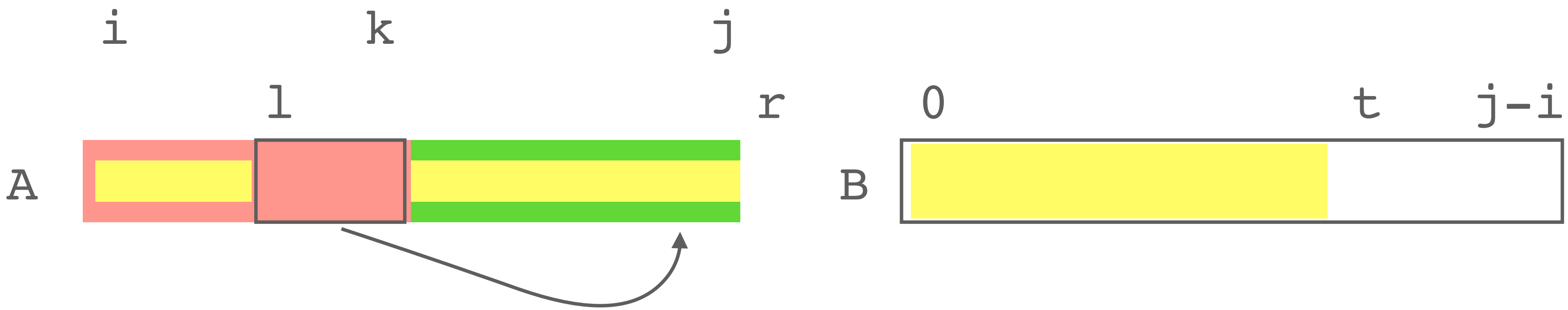
IDEA per procedura Merge:

- Usiamo un array di appoggio **`B[0..j-i]`** con `j-i+1` elementi
- Usiamo tre indici **`l`** (left), **`r`** (right) e **`t`** per scorrere, rispettivamente, la porzione di **`A`** di sinistra **`A[i..k]`**, la porzione di **`A`** di destra **`A[k+1..j]`**, e **`B`**
- Ad ogni passo, confrontiamo **`A[l]`** e **`A[r]`** e copiamo il più piccolo in **`B[t]`**, poi incrementiamo **`t`** e **`l`** o **`r`**, opportunamente
- Continuiamo fino a quando una delle due porzioni è stata completamente copiata in **`B`**

- Continuiamo fino a quando una delle due porzioni è stata completamente copiata in **`B`**
- Se è "esaurita" la porzione di sinistra, allora gli ultimi elementi di quella di destra (in **`A[r..j]`**) sono già al posto giusto



- Se è "esaurita" la porzione di destra, allora gli ultimi elementi di quella di sinistra (in **`A[l..k]`**) sono i più grandi e vengono copiati alla fine di **`A`** (in **`A[k-l+1..n-1]`**)
- Copiamo gli elementi in **`B`** in **`A[i..i+t-1]`**



MergeSort

Quanti confronti?

Inizializzazione

```
Merge(A,i,k,j)
l := i
r := k+1
t := 0
B[0..j-i] nuovo array
while (l ≤ k AND r ≤ j) do
 if A[l] ≤ A[r]
  then B[t] := A[l]
       l := l+1
  else B[t] := A[r]
       r := r+1
 t := t+1
for h=k downto l do
 A[j] := A[h]
 j := j-1
for h = 0 to t-1 do
 A[i+h] := B[h]
```

MergeSort

```
Merge(A,i,k,j)
l := i
r := k+1
t := 0
B[0..j-i] nuovo array
while (l ≤ k AND r ≤ j) do
 if A[l] ≤ A[r]
  then B[t] := A[l]
       l := l+1
  else B[t] := A[r]
       r := r+1
 t := t+1
for h=k downto l do
 A[j] := A[h]
 j := j-1
for h = 0 to t-1 do
 A[i+h] := B[h]
```

Quanti confronti?

Inizializzazione

Riempimento di B

$(j-i)$

nel caso peggiore

MergeSort

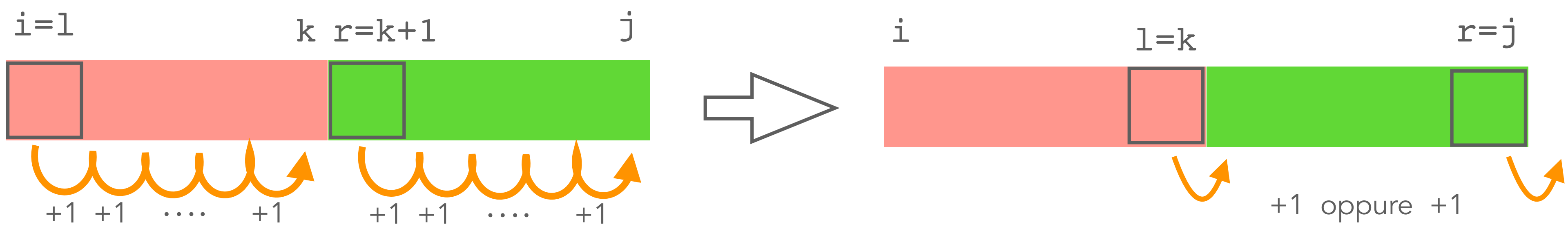
```
while (l ≤ k AND r ≤ j) do
  if A[l] ≤ A[r]
    then B[t] := A[l]
         l := l+1
    else B[t] := A[r]
         r := r+1
  t := t+1
```

Riempimento di B

Quanti confronti?

al più $(j-i)$ confronti

- Abbiamo un confronto per ogni iterazione del ciclo while
- Ogni iterazione può essere associata ad un avanzamento di uno (e solo uno) dei due indici `l` e `r`
- Il massimo numero di iterazioni del ciclo while si ha quando entrambi gli indici `l` e `r` arrivano fino alle fine della rispettiva partizione e, nell'iterazione successiva, uno dei due "esce" dalla partizione
- L'indice `l` può avanzare dalla posizione i alla posizione k, per un totale di k-i avanzamenti; l'indice `r` può avanzare dalla posizione k+1 alla posizione j, per un totale di j-(k+1) avanzamenti; uno dei due indici avanza ancora di una posizione, per un totale di un avanzamento extra. In totale: (k-i)+j-(k+1)+1= j-i avanzamenti.



UNIMORE

MergeSort

```
while (l ≤ k AND r ≤ j) do
 if A[l] ≤ A[r]
  then B[t] := A[l]
       l  := l+1
  else B[t] := A[r]
       r  := r+1
 t  := t+1
```

Riempimento di `B`

Quanti confronti?

almeno $(j - i)$ confronti

nel caso peggiore

Esiste un'istanza che porta sia l'indice `l` a `k` che l'indice `r` a `j` prima di eseguire l'ultima iterazione del ciclo while?

SI: ce ne sono tante! per esempio quelle per cui

$$A[k-1] < A[k+1] \text{ e } A[k] > A[j]$$

ESEMPIO: $A[i\,..\,k] = < 2{,}3{,}4{,}5{,}6{,}14 >$ e $A[k+1..j] = < 7{,}8{,}9{,}10{,}11{,}12 >$

non sono le uniche, per esercizio trovarne altre

MergeSort

```
Merge(A,i,k,j)
l := i
r := k+1
t := 0
B[0..j-i] nuovo array
while (l ≤ k AND r ≤ j) do
 if A[l] ≤ A[r]
  then B[t] := A[l]
       l := l+1
  else B[t] := A[r]
       r := r+1
 t := t+1
for h=k downto l do
 A[j] := A[h]
 j := j-1
for h = 0 to t-1 do
 A[i+h] := B[h]
```

Quanti confronti?

Inizializzazione

Riempimento di B — $(j-i)$ nel caso peggiore

Copia in A

MergeSort

```
Merge(A,i,k,j)
l := i
r := k+1
t := 0
B[0..j-i] nuovo array
while (l ≤ k AND r ≤ j) do
 if A[l] ≤ A[r]
  then B[t] := A[l]
       l := l+1
  else B[t] := A[r]
       r := r+1
 t := t+1
for h=k downto l do
 A[j] := A[h]
 j := j-1
for h = 0 to t-1 do
 A[i+h] := B[h]
```

Quanti confronti?

Inizializzazione

Riempimento di B — $(j-i)$ nel caso peggiore

Copia in A

Totale — $(j-i)$ nel caso peggiore

MergeSort

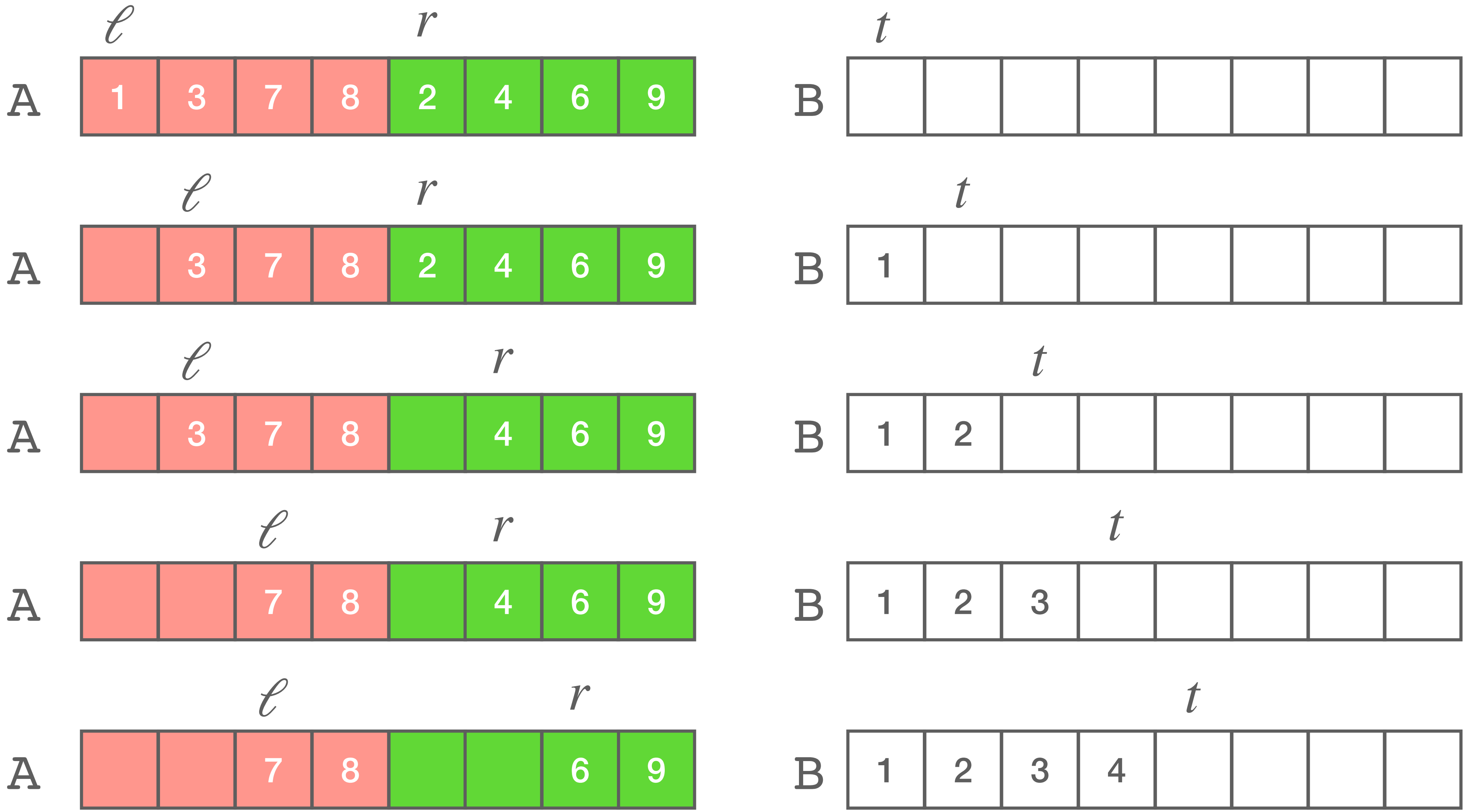
ESEMPIO 1: esecuzione `Merge`

	ℓ				r			
A	1	3	7	8	2	4	6	9

	t							
B								

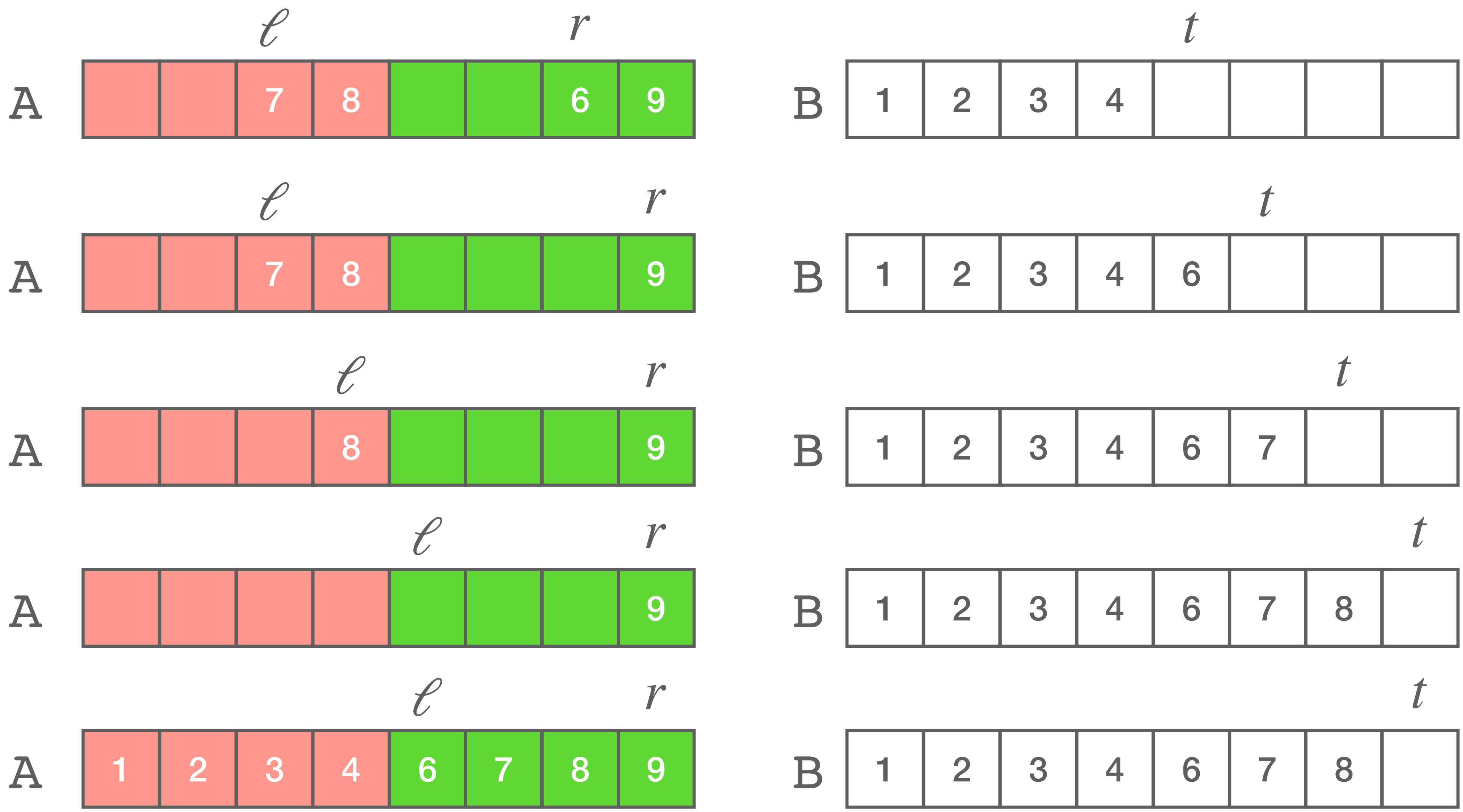
MergeSort

ESEMPIO 1: esecuzione `Merge`



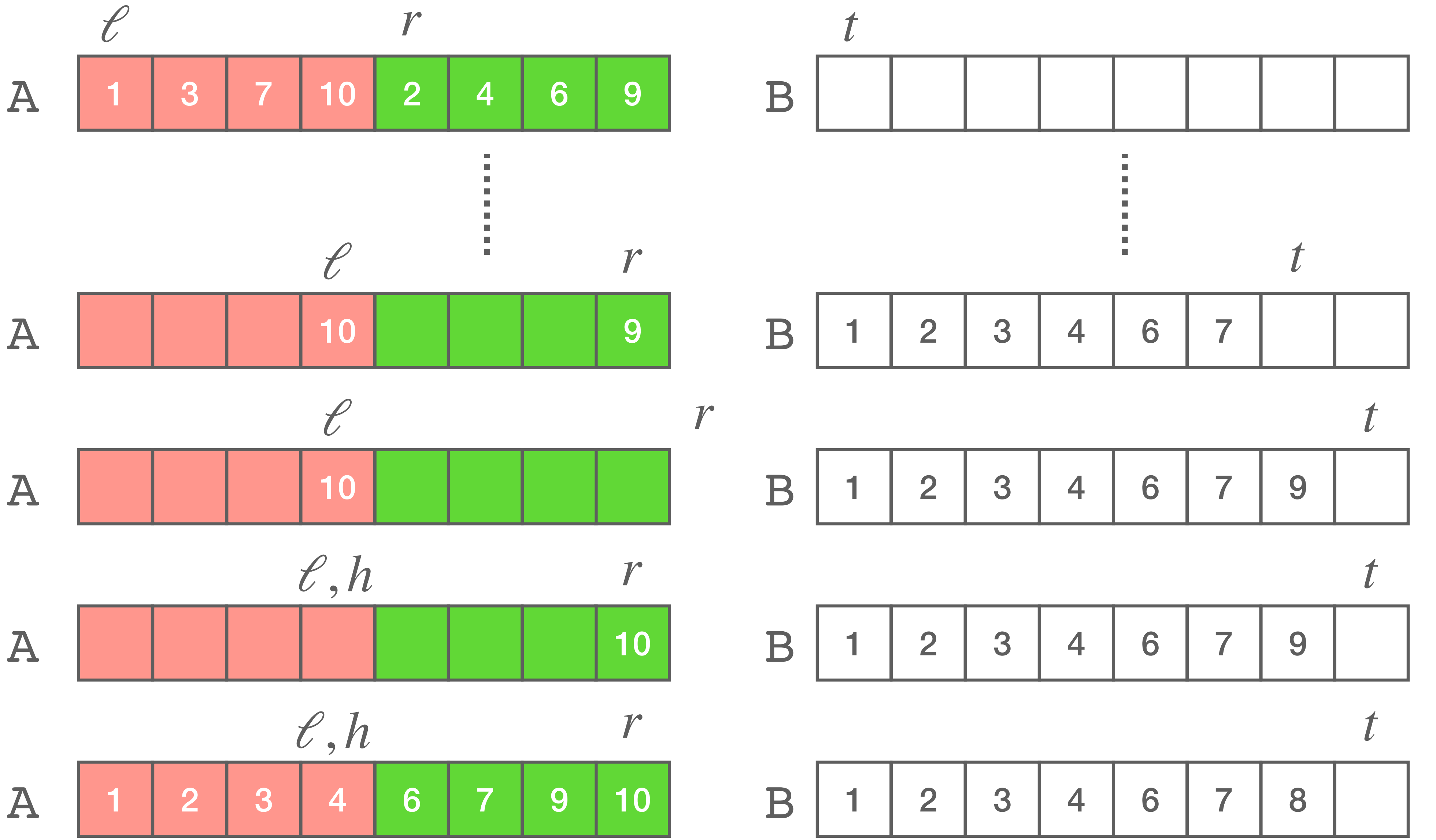
MergeSort

ESEMPIO 1: esecuzione `Merge`



MergeSort

ESEMPIO 2: esecuzione `Merge`

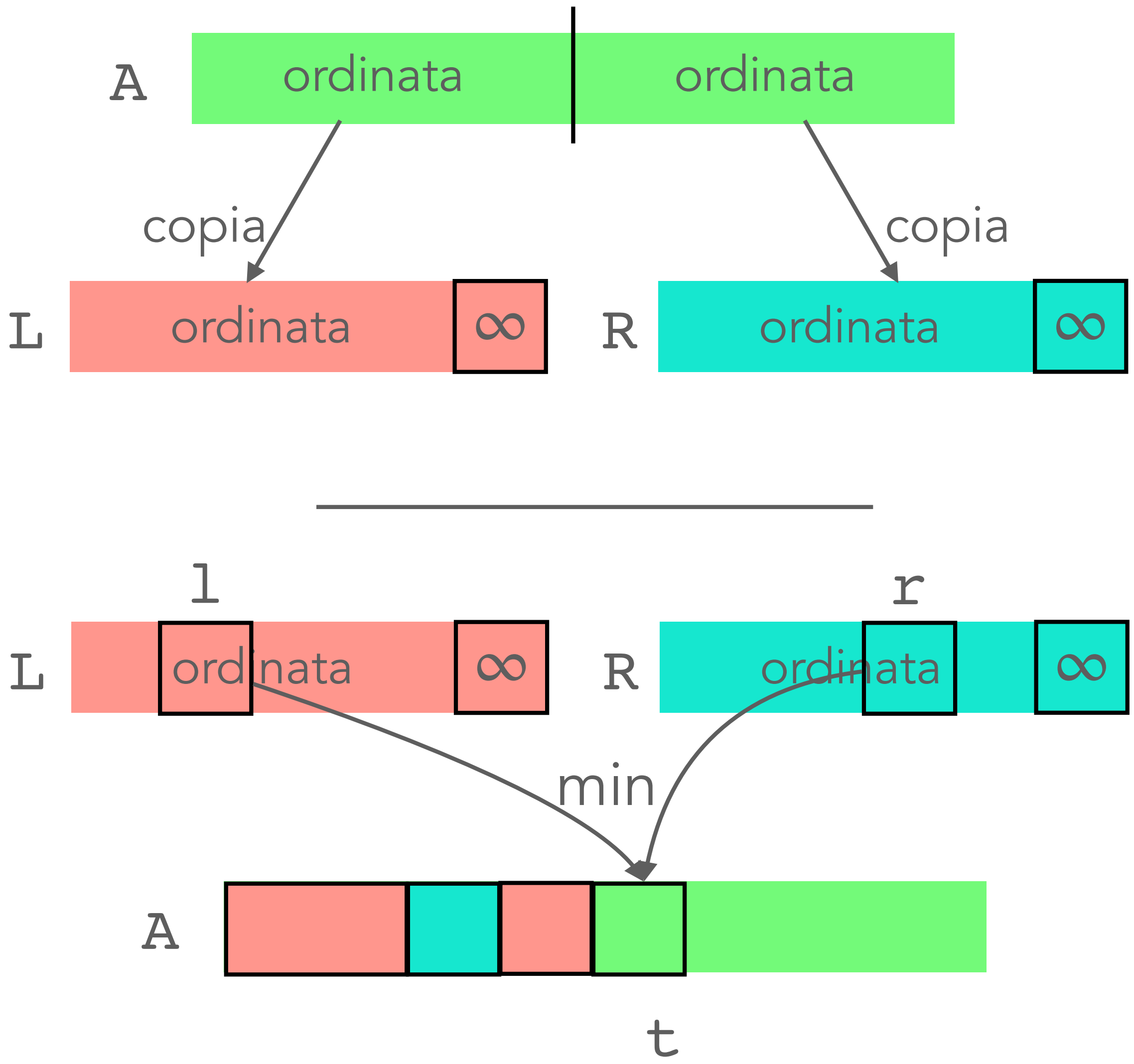


MergeSort

`Merge` alternativo

In questa versione il numero di confronti è esattamente $(j - i)$ per ogni istanza

```
MERGE(A,i,k,j)
 n1 := k-i+1
 n2 := j-k
 crea L[0..n1] e R[0..n2]
 for t = 0 to n1-1
  L[t] := A[i+t]
 for t = 0 to n2-1
  R[t] := A[k+1+t]
 L[n1] := ∞
 R[n2] := ∞
 l := 0
 r := 0
 for t = i to j
  if L[l] ≤ R[r]
   then
    A[t] := L[l]
    l:= l + 1
   else
    A[t] := R[r]
    r := r + 1
```



MergeSort

COSTO COMPUTAZIONALE

```
MERGESORT(A,i,j)
 if i < j
  then
   k := ⌊(i+j)/2⌋
   MERGESORT(A,i,k)
   MERGESORT(A,k+1,j)
   MERGE(A,i,k,j)
```

$$T(n) = \begin{cases} 0 & \text{se } n = 1 \\ 2 \cdot T(n/2) + O(n) & \text{altrimenti} \end{cases}$$

Costo di **Merge**

Usando il Master Theorem

$$a = 2, b = 2, d = 1 \quad \Rightarrow \log_b a = \log_2 2 = 1 = d \quad \Rightarrow T(n) \in O(n \log n)$$

MergeSort

COSTO COMPUTAZIONALE

```
MERGESORT(A,i,j)
 if i < j
  then
   k := ⌊(i+j)/2⌋
   MERGESORT(A,i,k)
   MERGESORT(A,k+1,j)
   MERGE(A,i,k,j)
```

$$T(n) = \begin{cases} 0 & \text{se } n = 1 \\ 2 \cdot T(n/2) + O(n) & \text{altrimenti} \end{cases}$$

$$T(n) \in O(n \log n)$$

Siamo veramente contenti?

SI: nessun algoritmo che risolve il problema utilizzando confronti tra elementi dell'array può fare di meglio (nel caso peggiore)

Lower Bound Ordinamento

TEOREMA

Un algoritmo di ordinamento per CONFRONTI deve effettuare almeno $\Omega(n \log n)$ confronti tra elementi della sequenza da ordinare per risolvere il problema

DIMOSTRAZIONE

OGNI algoritmo per l'ordinamento che procedere per confronti deve effettuare un certo numero di confronti che gli permetta di decidere quale è quella, tra le $n!$ permutazioni degli elementi della sequenza, che permette di ordinarli

Lower Bound Ordinamento

DIMOSTRAZIONE

p_i = numero di permutazioni tra cui scegliere dopo il confronto i-esimo

Con un confronto tra A[i] e A[j]:

- si possono scartare tutte le permutazioni in cui A[i] e A[j] sono ordinati in ordine inverso rispetto al risultato del confronto (sia p_i_KO il loro numero)
- si devono continuare a tenere in considerazione tutte le altre (sia p_i_OK il loro numero)

ESEMPIO: se A[i] < A[j], allora possiamo scartare tutte le permutazioni che mettono il valore in posizione i DOPO (a destra) del valore in posizione j. Per tutte le altre non possiamo ancora dire niente.

Lower Bound Ordinamento

DIMOSTRAZIONE

p_i = numero di permutazioni tra cui scegliere dopo il confronto i-esimo

Con il confronto $i + 1$ tra gli elementi A[i] e A[j]:

- si possono scartare tutte le permutazioni in cui A[i] e A[j] sono ordinati in ordine inverso rispetto al risultato del confronto (sia p_i_KO il loro numero)
- si devono continuare a tenere in considerazione tutte le altre (sia p_i_OK il loro numero)

Abbiamo che $p_i\_KO + p_i\_OK = p_i$

e che $p_{i+1} = p_i\_OK$

Lower Bound Ordinamento

DIMOSTRAZIONE

Abbiamo che

- $p_0 = n!$ perché non è ancora stato fatto nessun confronto
- $p_0 = p_{0\_}KO + p_{0\_}OK = n!$
- $p_1 = p_{0\_}OK = \max\{p_{0\_}KO, p_{0\_}OK\}$ nel caso peggiore (che rimanga il sottoinsieme più grande)
- Nel caso peggiore $p_1 = \max\{p_{0\_}KO, p_{0\_}OK\} \geq p_0/2 = n!/2$

Lower Bound Ordinamento

DIMOSTRAZIONE

Analogamente

- $p_0 = n!$
- $p_1 \geq n!/2$
- $p_2 \geq n!/4$
- $p_3 \geq n!/8$

....

- dopo il confronto i-esimo: $p_i \geq n!/2^i$

Per poter scegliere la permutazione è necessario che il numero di permutazioni tra cui scegliere sia UNA sola.

Dopo quanti confronti (quale i) abbiamo $1 = p_i \geq n!/2^i$?

per $i \geq \log n!$

Lower Bound Ordinamento

DIMOSTRAZIONE

Studiamo $\log n!$

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

$$\geq n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot \left(\frac{n}{2} + 1\right)$$

$$> \underbrace{\frac{n}{2} \cdot \frac{n}{2} \cdot \ldots \cdot \frac{n}{2}}_{n/2 \text{ volte}} = \left(\frac{n}{2}\right)^{\frac{n}{2}}$$

assumiamo n pari

limitiamo il prodotto ai primi n/2 fattori

ogni fattore è maggiore di n/2

quindi $\quad i \geq \log n! \geq \log \left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2} \log \left(\frac{n}{2}\right) \in \Omega(n \log n)$

UNIMORE

Lower Bound Ordinamento

TEOREMA

Un algoritmo di ordinamento per CONFRONTI deve effettuare almeno $\Omega(n \log n)$ confronti tra elementi della sequenza da ordinare per risolvere il problema

DIMOSTRAZIONE

vedremo una dimostrazione alternativa quando parleremo di alberi

MergeSort

COSTO COMPUTAZIONALE

```
MERGESORT(A,i,j)
 if i < j
  then
   k := ⌊(i+j)/2⌋
   MERGESORT(A,i,k)
   MERGESORT(A,k+1,j)
   MERGE(A,i,k,j)
```

$$T(n) = \begin{cases} 0 & \text{se } n = 1 \\ 2 \cdot T(n/2) + O(n) & \text{altrimenti} \end{cases}$$

$$T(n) \in O(n \log n)$$

Siamo veramente contenti?

NO: - usiamo spazio aggiuntivo $\Theta(n)$ —> altri algoritmi

- in pratica le chiamate riscorsive sono costose dal punto di vista del tempo di esecuzione —> riduciamo il numero di chiamate ricorsive

MergeSort + InsertionSort

Limitiamo il numero di chiamate ricorsive

- Il caso base diventa un po' più "grande"
  —> array con un numero di elementi **costante s** > 1
- Usiamo InsertionSort per risolvere il caso base

ESERCIZIO
Scrivere lo pseudocodice
prima di guardare la slide successiva

MergeSort + InsertionSort

Limitiamo il numero di chiamate ricorsive

- Il caso base diventa un po' più "grande"
  —> array con un numero di elementi **costante** > 1
- Usiamo InsertionSort per risolvere il caso base

Versione modificata di `MERGESORT` che prende in input anche un valore soglia per decidere quando chiamare `INSRETIONSORT` invece di una chiamata ricorsiva

```
MERGESORT(A,i,j,s)
  k := ⌊(i+j)/2⌋
  if k-i+1 ≤ s
    then
      INSERTIONSORT(A,i,k)
    else
      MERGESORT(A,i,k,s)
  if j-k ≤ s
    then
      INSERTIONSORT(A,k+1,j)
    else
      MERGESORT(A,k+1,j,s)
  MERGE(A,i,k,j)
```

Versione modificata di `INSERTIONSORT` in cui in viene ordinata la sottosequenza compresa tra gli indici in input (estremi inclusi)

Chiamata principale `MERGESORT(A,0,n-1,s)`