ORDINAMENTO
MERGESORT
LIMITI INFERIORI ALL'ORDINAMENTO CON COMPARAZIONE

INFORMATICA

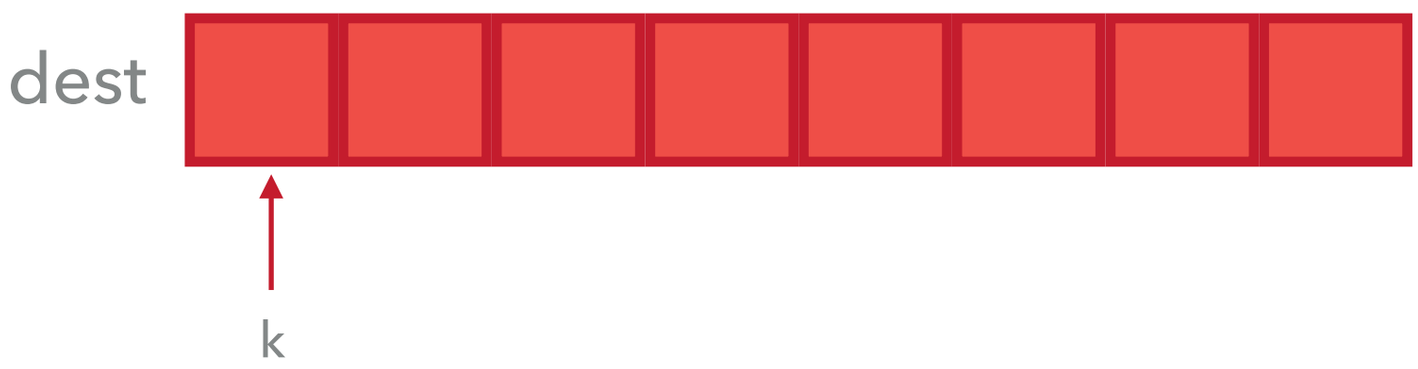
MERGESORT

- Nella prima lezione abbiamo visto un modo efficiente di ordinare un mazzo di carte
- Ora siamo in grado di formalizzare l'algoritmo e di studiarne la complessità
- Anche in questo caso l'algoritmo è definito in maniera ricorsiva.

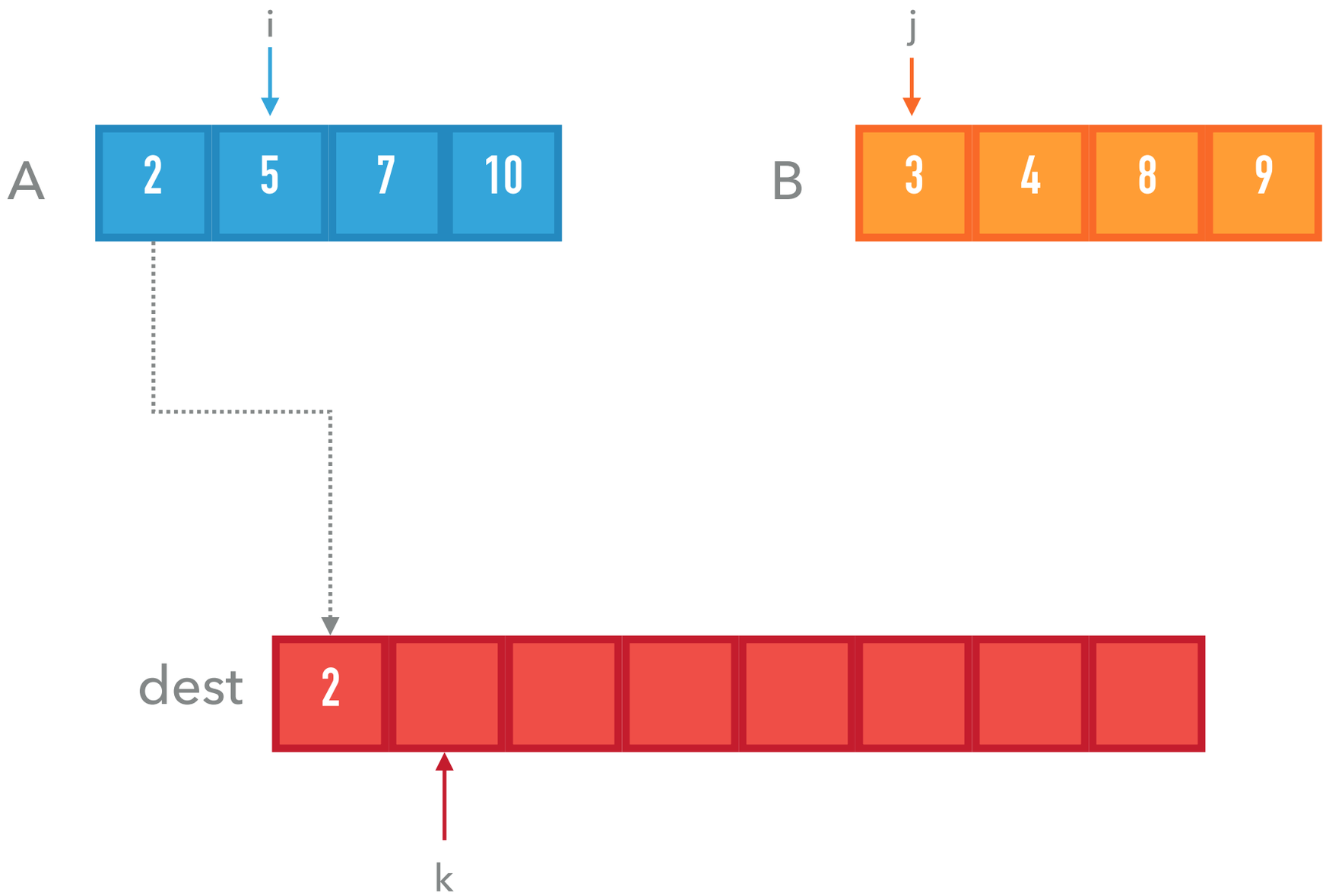
PROCEDURE DA DEFINIRE

- La prima (e unica) procedura da definire è la procedura di **merge**
- Ricordate, richiede di creare un array ordinato a partire da due array ordinati
- Dopo la procedura di merge possiamo direttamente definire il **mergesort**

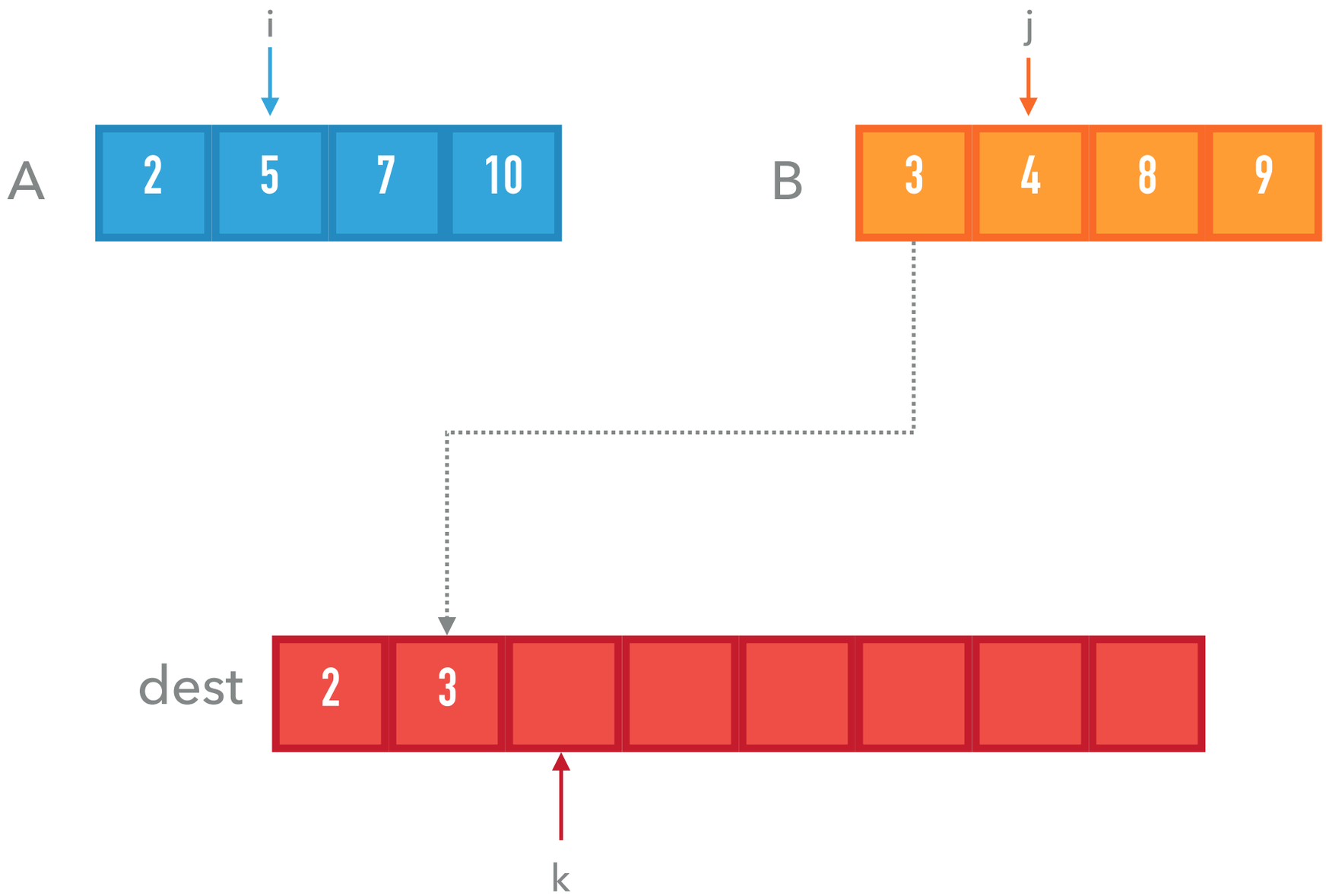
PROCEDURA DI MERGE



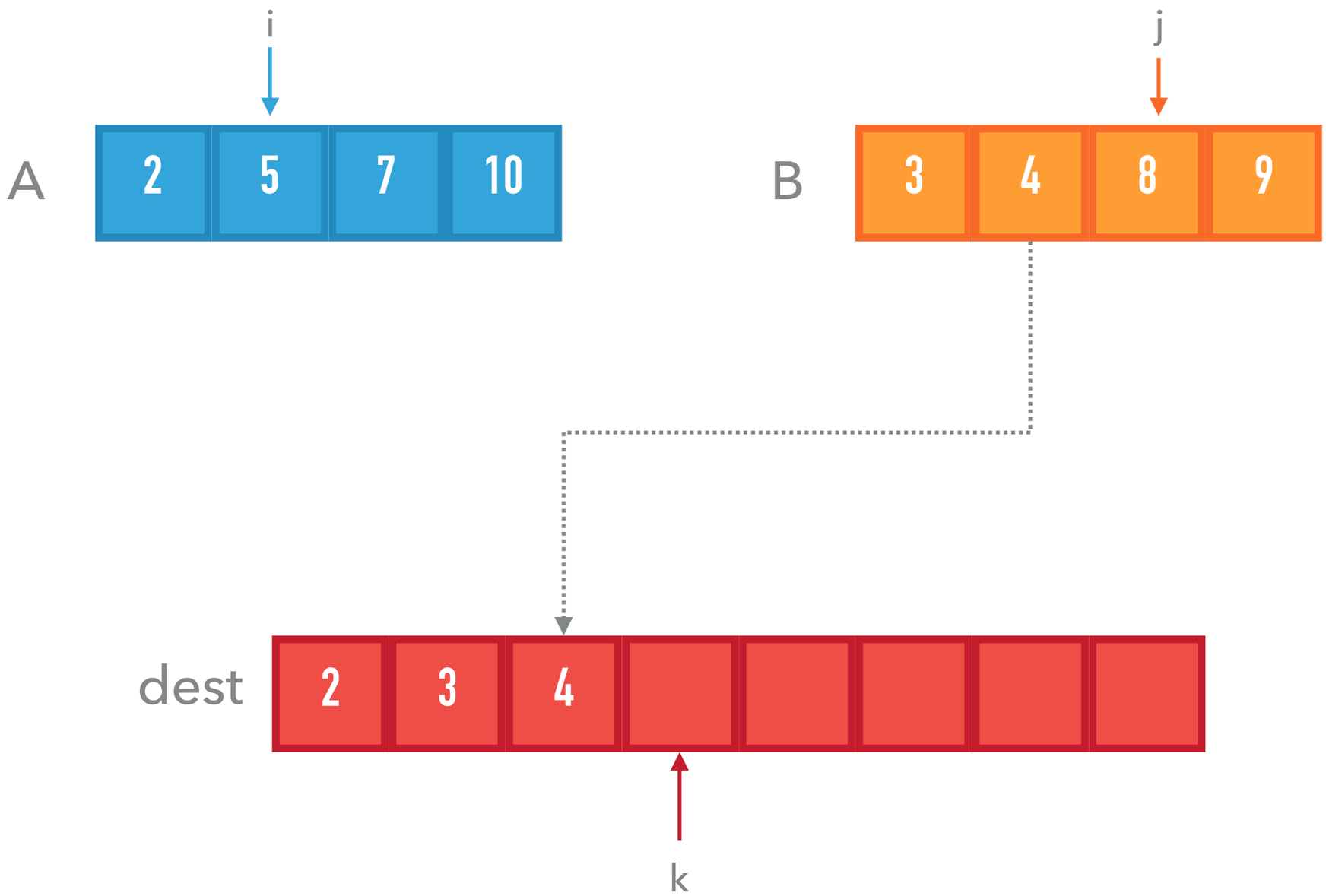
PROCEDURA DI MERGE



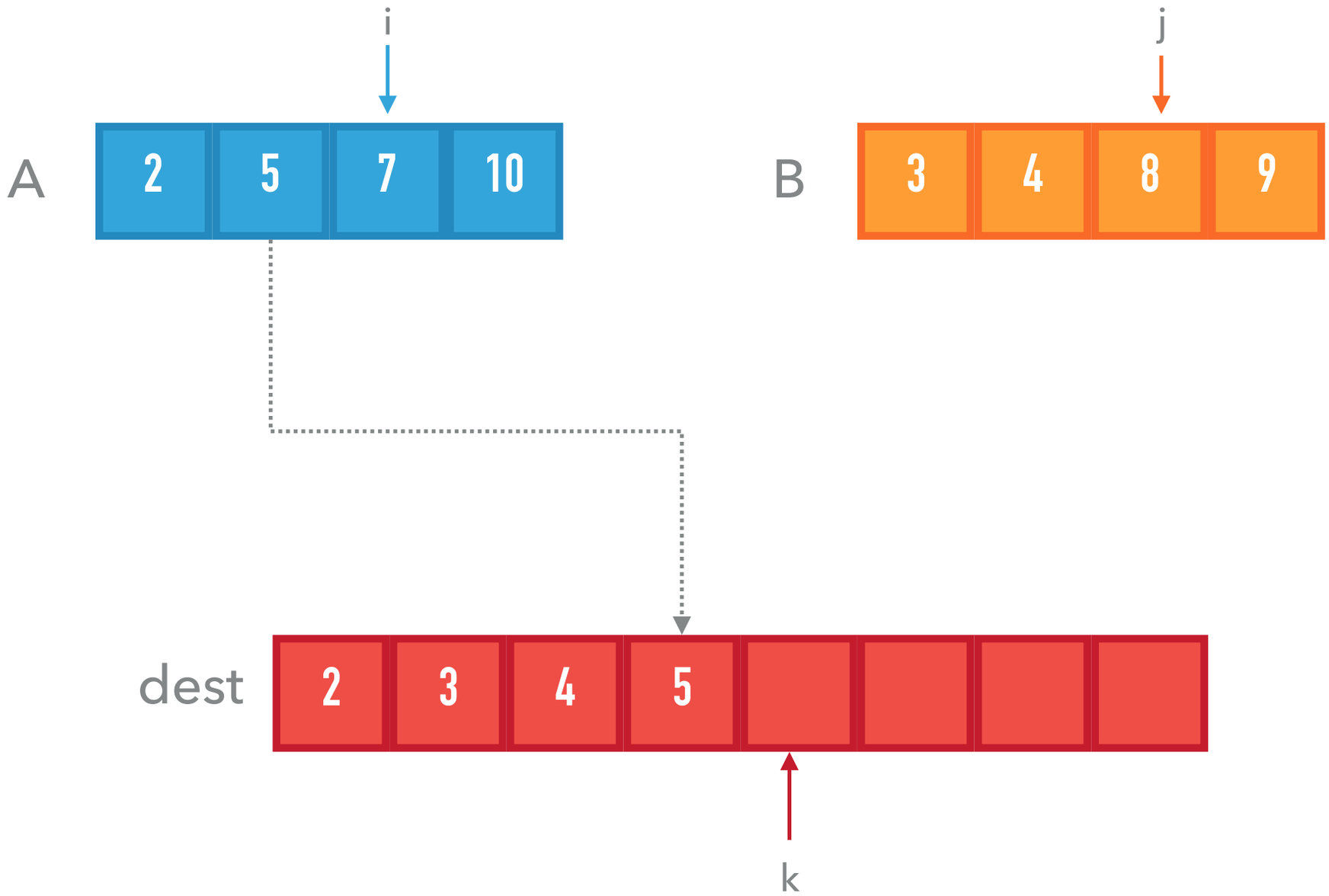
PROCEDURA DI MERGE



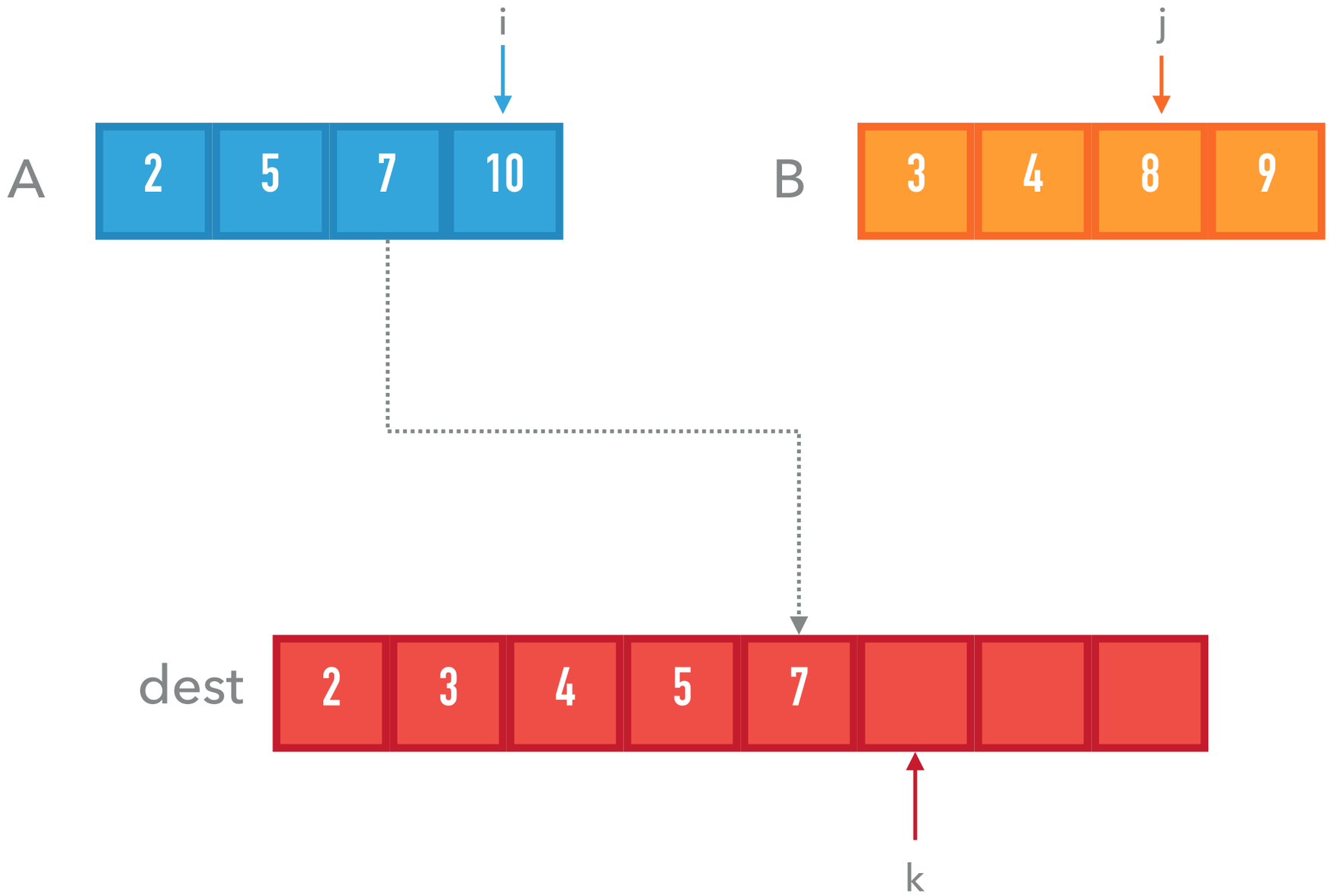
PROCEDURA DI MERGE



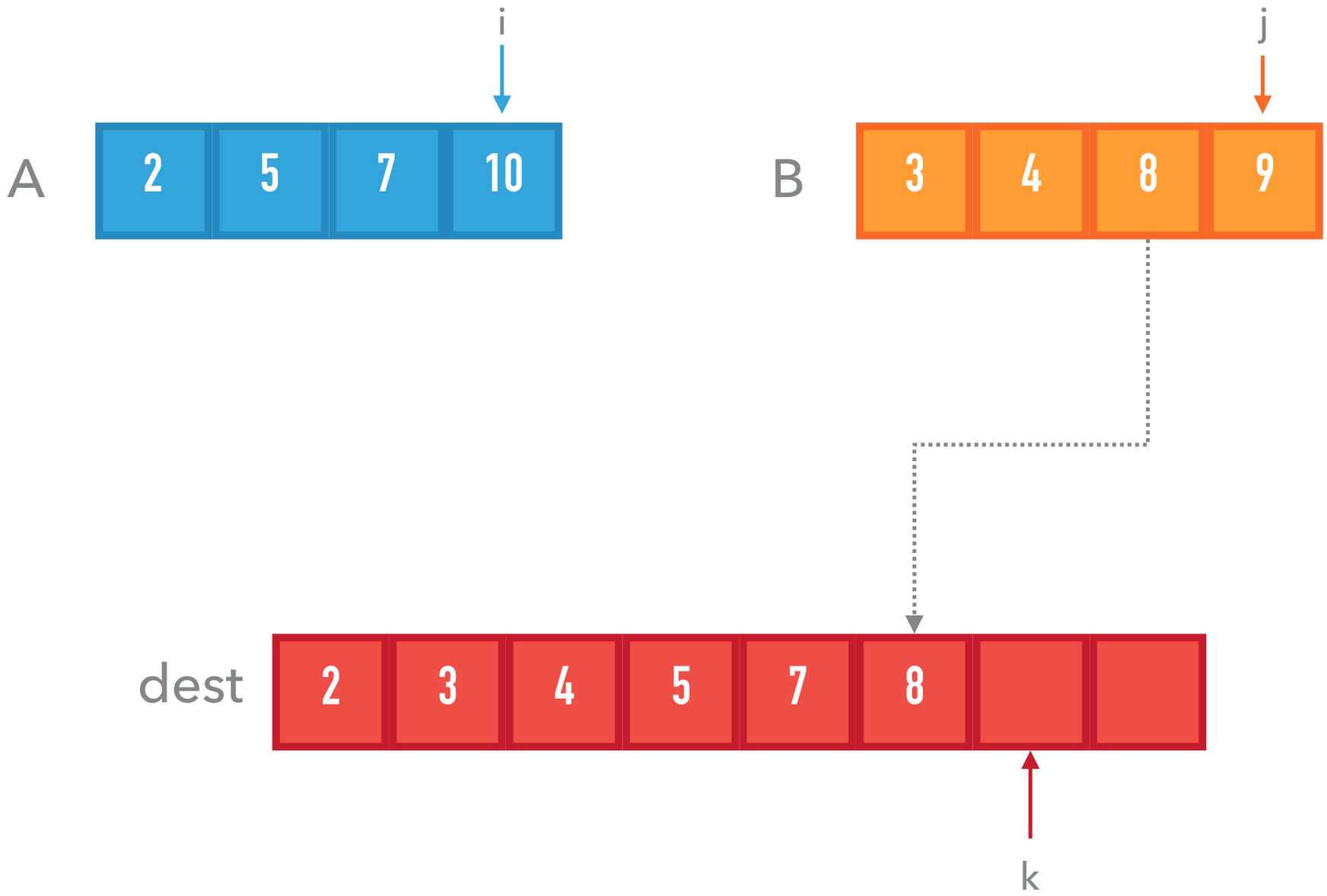
PROCEDURA DI MERGE



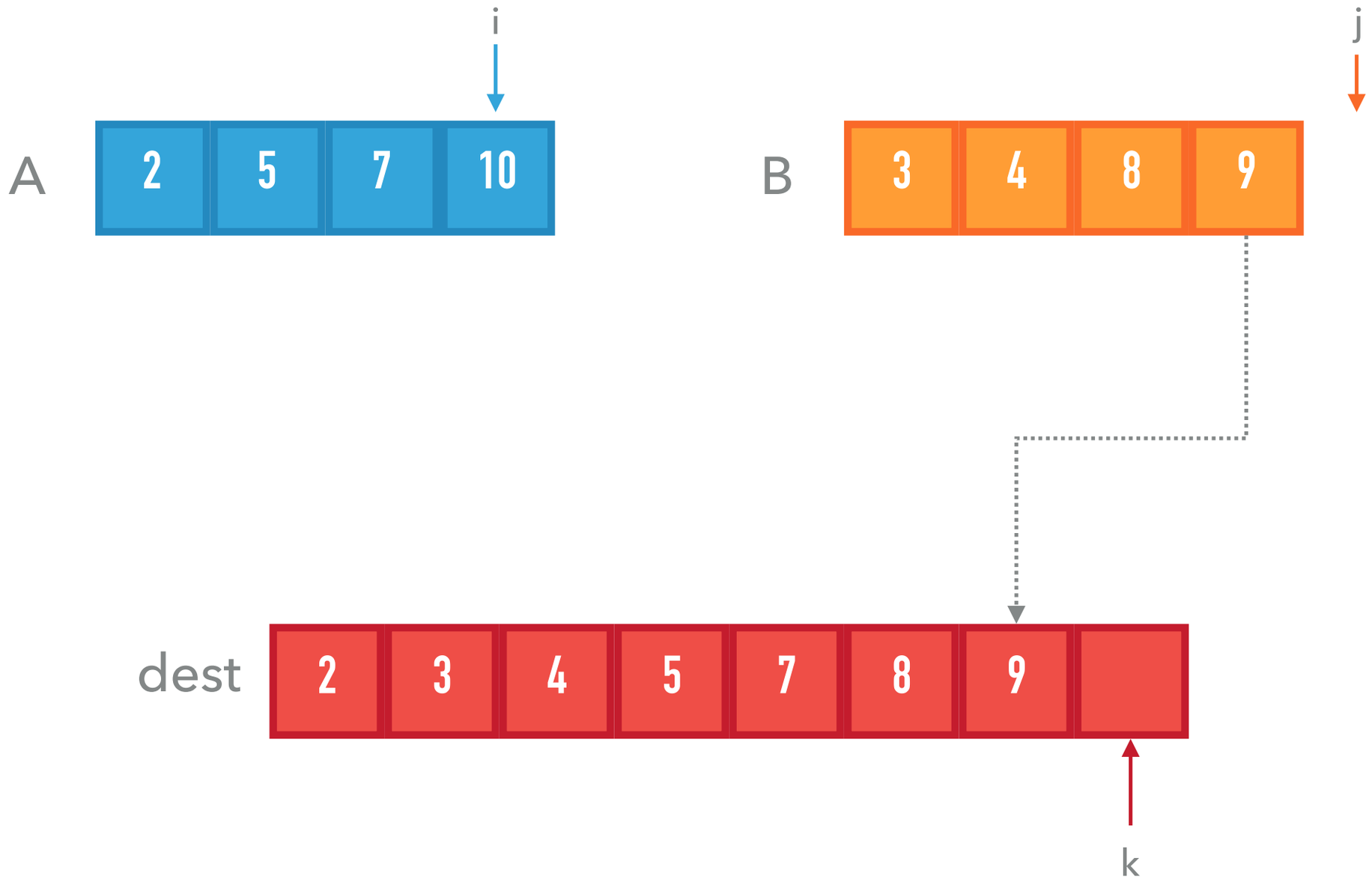
PROCEDURA DI MERGE



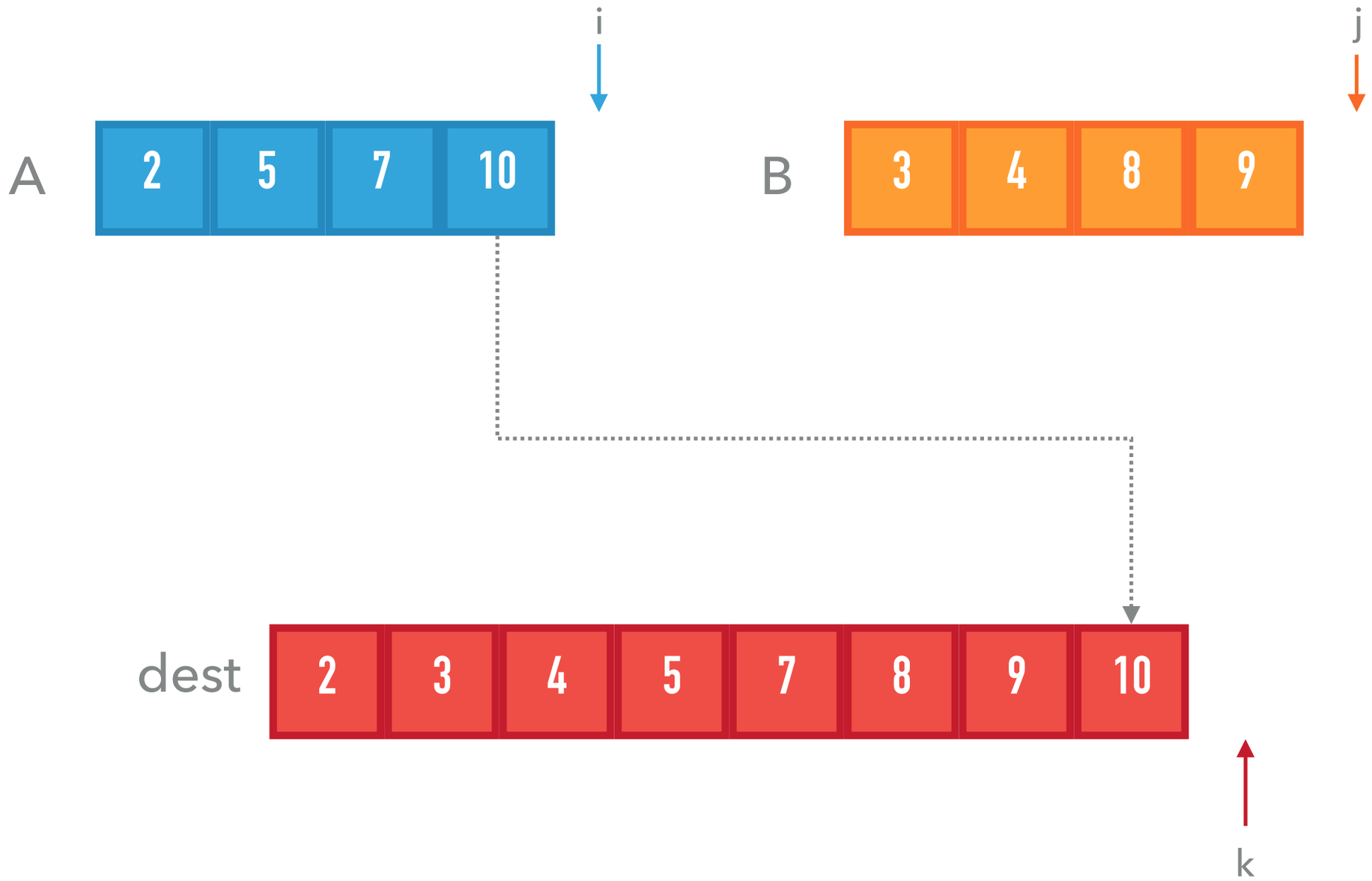
PROCEDURA DI MERGE



PROCEDURA DI MERGE



PROCEDURA DI MERGE



MERGE: PSEUDOCODICE (PARTE 1)

```
Parametri: A (primo array), B (secondo array)
i = 0 # indice nell'array A
j = 0 # indice nell'array B
dest = [array di len(A) + len(B) elementi]
k = 0 # indice nell'array di destinazione
```

Definiamo i due indici con cui scorrere gli array A e B e prepariamo un array di destinazione "dest"

```
while i < len(A) and j < len(B)
    if A[i] < B[j]
        dest[k] = A[i]
        i = i + 1
    else
        dest[k] = B[j]
        j = j + 1
    k = k + 1
```

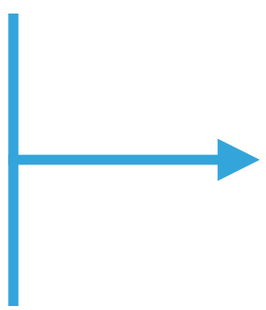
Minimo in A

Minimo in B

Finché non siamo arrivati alla fine di uno dei due array copiamo di volta in volta l'elemento minore ed incrementiamo l'indice corrispondente

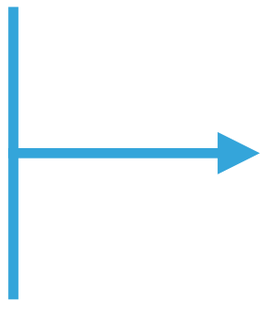
MERGE: PSEUDOCODICE (PARTE 2)

- `while i < len(A)`
  - `dest[k] = A[i]`
  - `i = i + 1`
  - `k = k + 1`



Finiamo di copiare gli elementi dell'array A nella destinazione nel caso ne fossero rimasti

- `while j < len(B)`
  - `dest[k] = B[j]`
  - `j = j + 1`
  - `k = k + 1`



Finiamo di copiare gli elementi dell'array B nella destinazione nel caso ne fossero rimasti

Nota: la procedura di merge funziona sotto l'assunzione che i due array forniti in input siano ordinati!

MERGE: COMPLESSITÀ

- La complessità è lineare rispetto al numero di elementi del vettore:
  - Primo ciclo for richiede $O(n)$ passi
  - Gli altri cicli for sono comunque limitati da $O(n)$ passi
  - La procedura di merge richiede quindi tempo $O(n)$
- Possiamo notare che, dato che comunque dobbiamo riempire l'array di destinazione, il tempo è $\Theta(n)$

MERGESORT: PSEUDOCODICE

- Parametri: A (array)
- if len(A) > 1
  - center = $\lfloor$len(A)/2$\rfloor$
  - left = mergesort(A[0…center])
  - right = mergesort(A[center…len(A)])
  - return merge(left, right)
- else
  - return A

Due chiamate ricorsive su array di dimensione $n/2$

MERGESORT: COMPLESSITÀ

- Ad ogni passo eseguiamo la procedura di merge, che richiede tempo $\Theta(n)$
- Effettuiamo due chiamate ricorsive ognuna su array di dimensione $n/2$
- L'equazione di ricorrenza è quindi

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

MERGESORT: COMPLESSITÀ

- Applichiamo il secondo caso del teorema principale, ricordando che $\Theta(n)$ è esprimibile come $\Theta(n^{\log_2(2)})$:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n^{\log_2(2)})$$

- Quindi il mergesort ha complessità temporale $\Theta(n^{\log_2(2)} \log n) = \Theta(n \log n)$

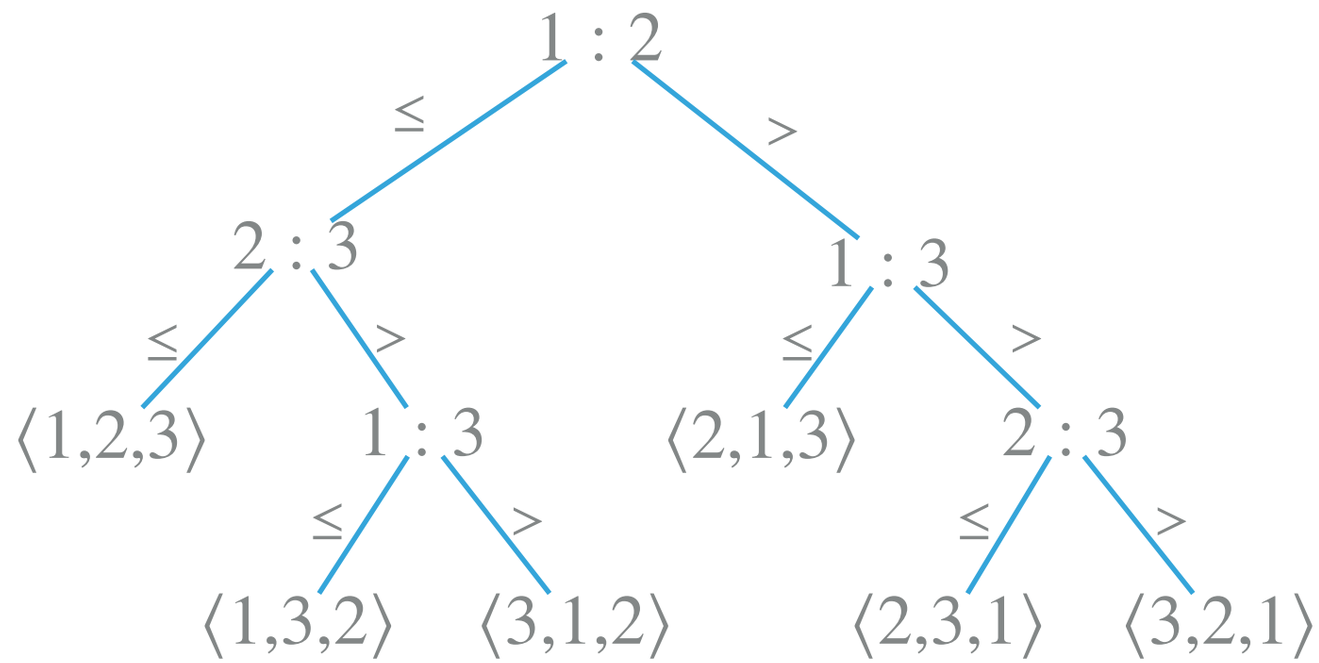
LIMITI DELL'ORDINAMENTO

- Sia mergesort che heapsort richiedono tempo $O(n \log n)$
- Sappiamo di non poter andare al di sotto del tempo lineare: dobbiamo *almeno* leggere l'input
- Possiamo migliorare e avere, per esempio, un algoritmo di ordinamento che richiede tempo lineare?
- Mostriamo che gli algoritmi di ordinamento che usano la comparazione sono in $\Omega(n \log n)$

ALBERO DI DECISIONE

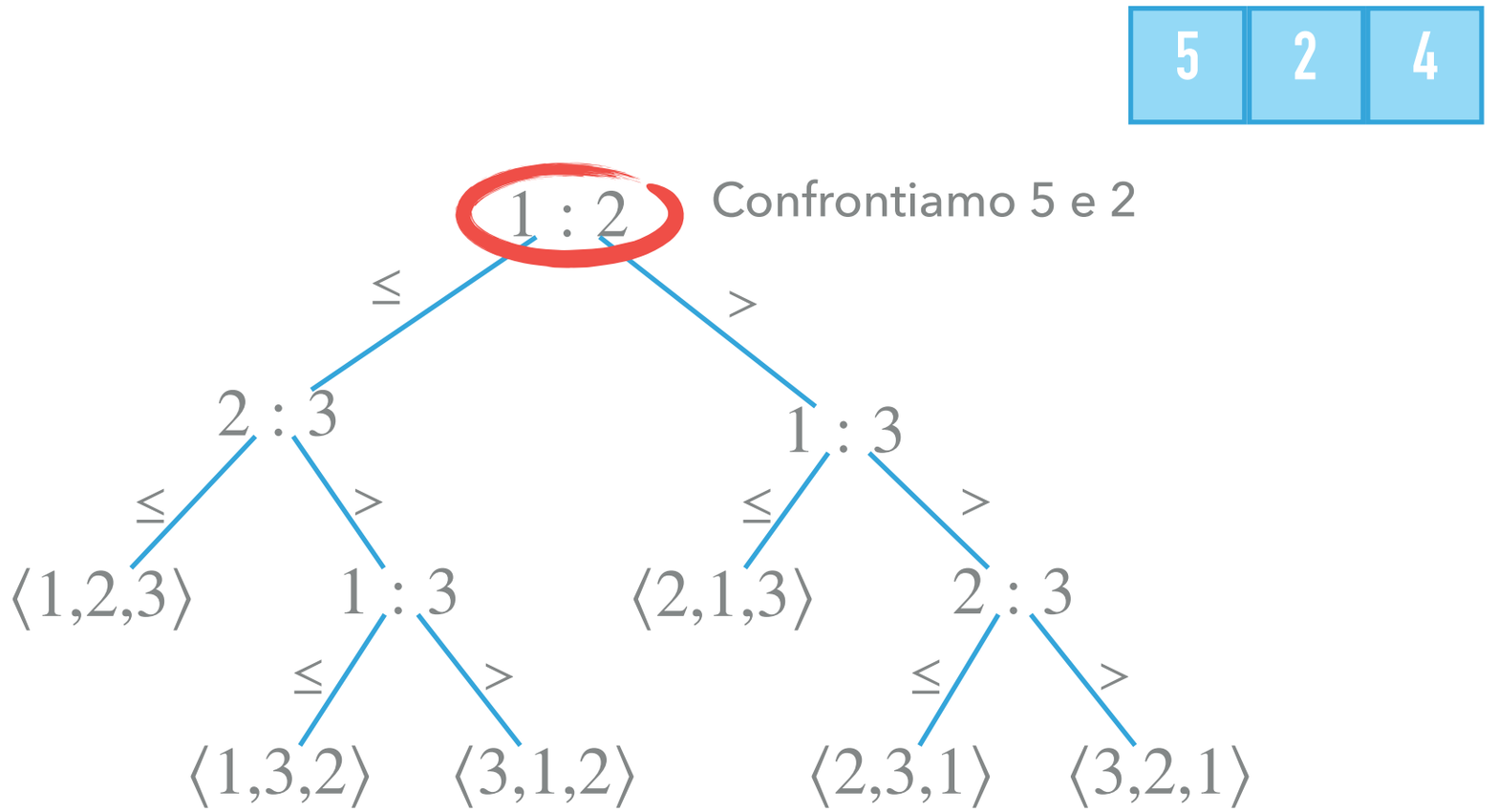
- Dato un array A di n elementi
- Consideriamo un albero binario in cui ogni nodo interno ha associata una comparazione $i : j$, che indica che compariamo l'elemento i-esimo con l'elemento j-esimo
- Ci spostiamo a sinistra se $A[i] \leq A[j]$ e a destra se $A[i] > A[j]$
- Ogni foglia è una permutazione π di $0 \ldots n - 1$, gli indici di A

ALBERO DI DECISIONE

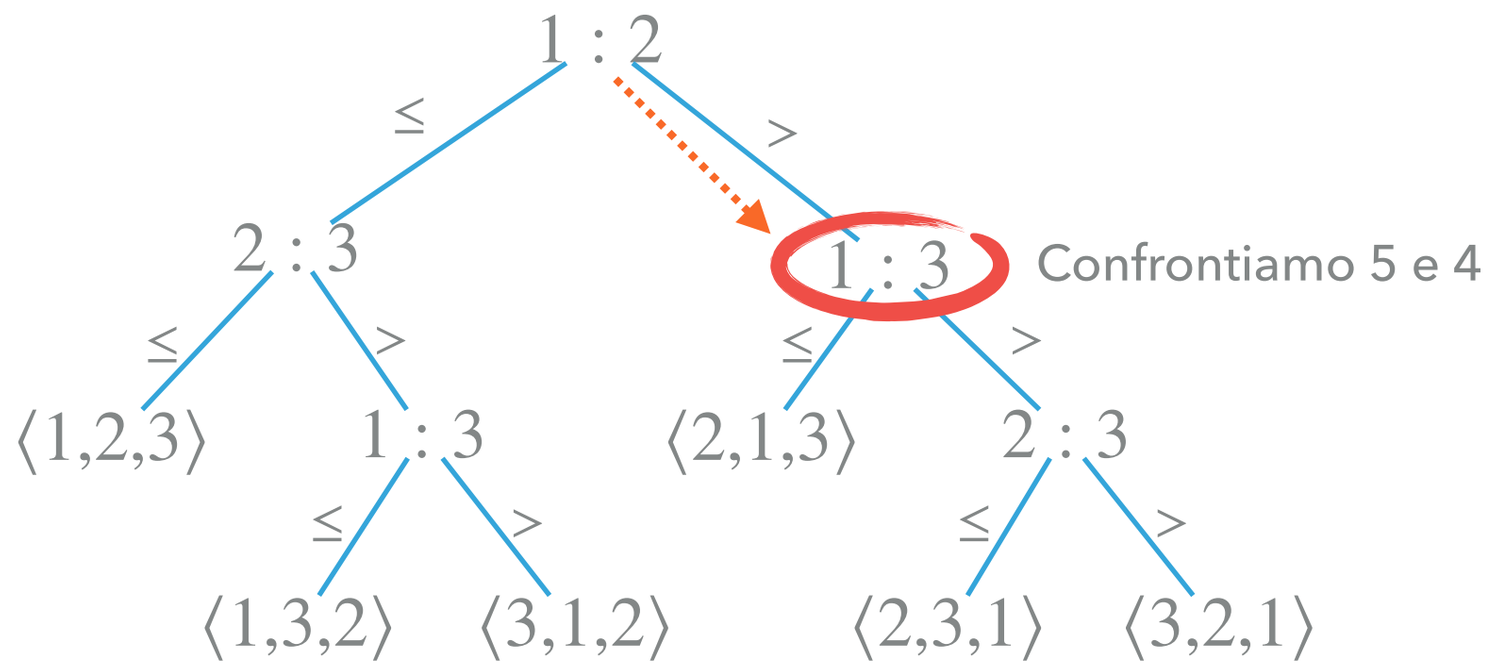


In questo esempio gli indici dell'array partono da 1, come sul libro di testo

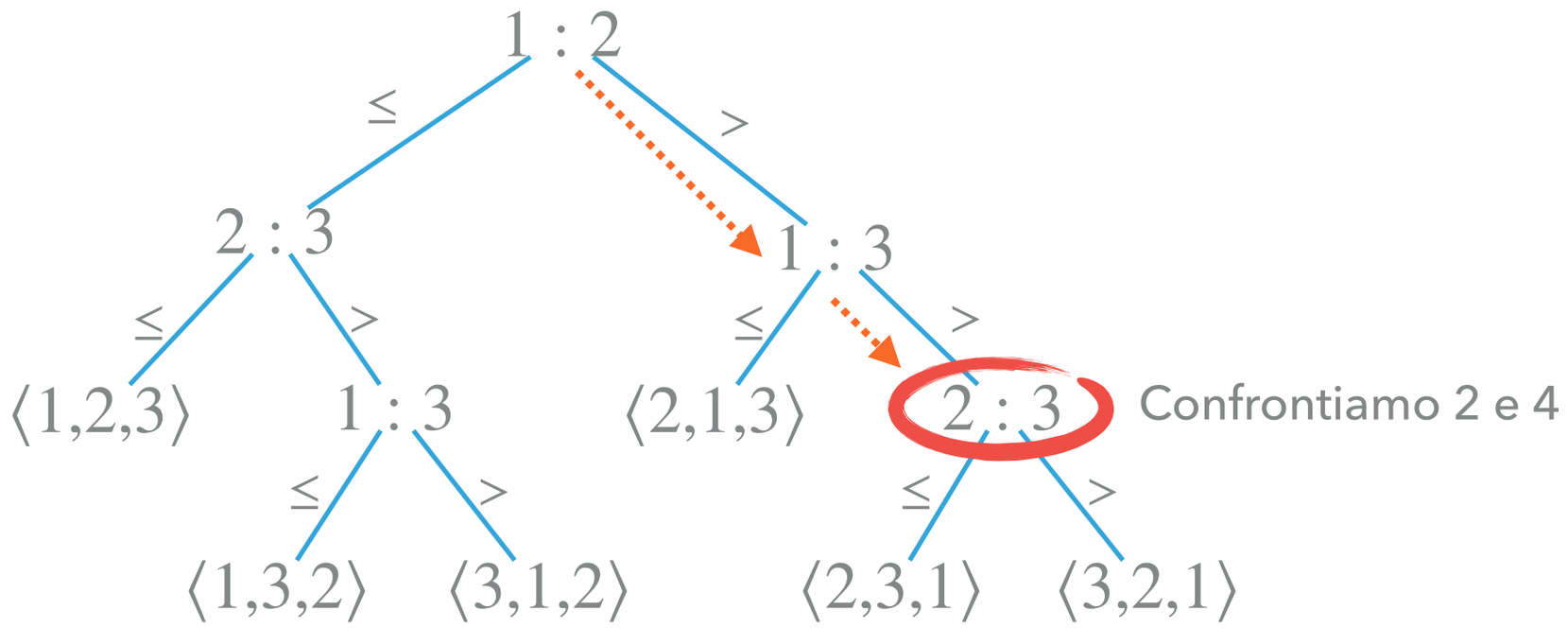
ALBERO DI DECISIONE



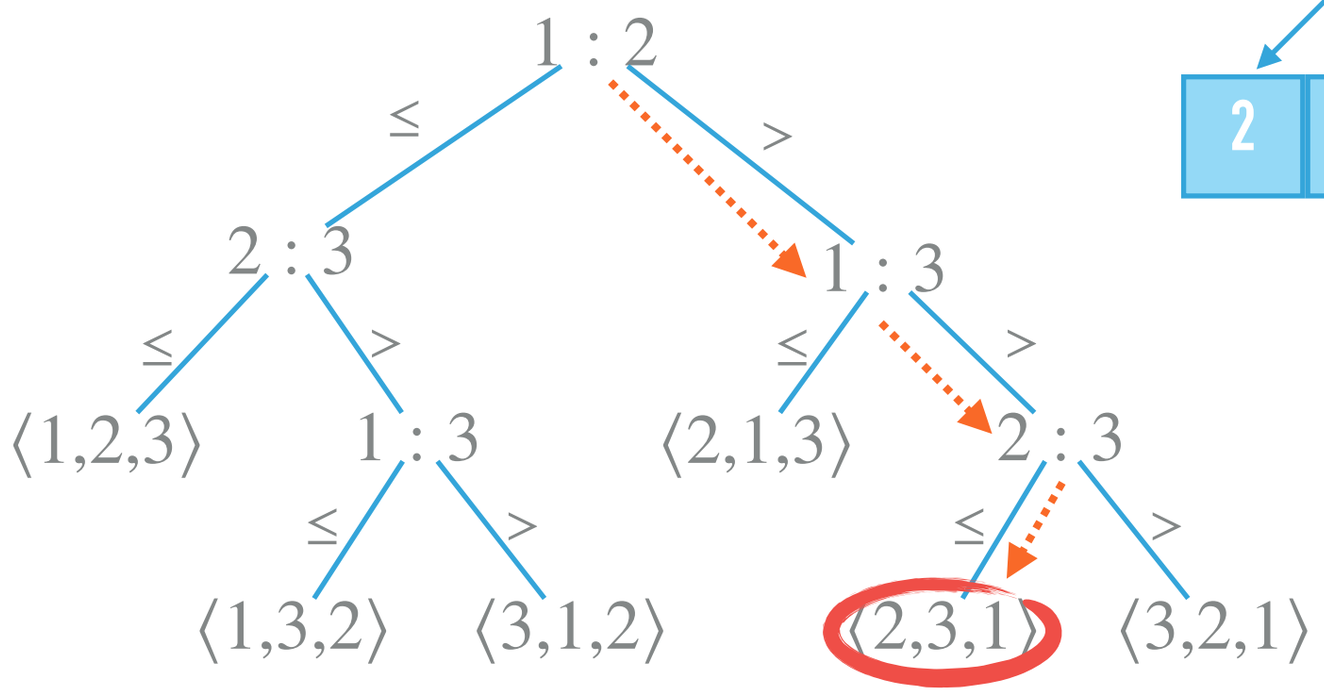
ALBERO DI DECISIONE

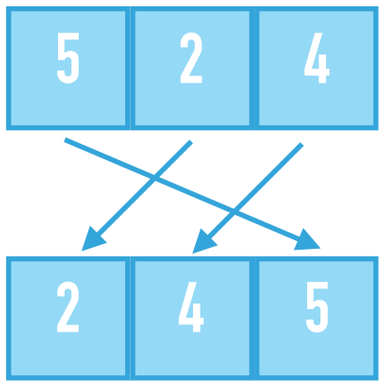


ALBERO DI DECISIONE



ALBERO DI DECISIONE





Permutazione da applicare all'array

ALBERO DI DECISIONE

- Dato un qualsiasi algoritmo di ordinamento che usa la comparazione di elementi dell'array per compiere le scelte, possiamo rappresentarlo come un albero di decisione
- Algoritmi diversi corrispondono ad alberi diversi
- Dato un array in input la sua esecuzione individuerà un percorso dalla radice ad una delle permutazioni nelle foglie

ALBERO DI DECISIONE

- Come conseguenza **ogni** permutazione deve essere presente nelle foglie!
- Le permutazioni di n elementi sono $n!$
- Quale è l'altezza minima h di un albero con $\ell \geq n!$ foglie?
- Questa altezza minima ci indica il numero di comparazioni minimo nel caso peggiore che siamo costretti a fare in un algoritmo di ordinamento basato sulla comparazione

ALBERO DI DECISIONE

- Ricordiamo che un albero di profondità h ha al più 2^h foglie
- Ne segue $n! \leq \ell \leq 2^h$
- Prendiamo il logaritmo: $h \geq \log_2(n!)$
- Mostriamo ora che $\log_2(n!)$ è $\Theta(n \log n)$

ALBERO DI DECISIONE

- Per approssimazione di Stirling:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\Theta\left(\frac{1}{n}\right)\right)$$

- Prendendo il logaritmo dell'approssimazione passiamo da prodotti a somme:

$$\log_2\sqrt{2\pi n} + \log_2 n^n + \log_2 e^{-n} + \log_2(1+\Theta(n^{-1}))$$

- Il termine che domina asintoticamente è $\log_2 n^n = n\log_2 n$, quindi $\log n! = \Theta(n\log n)$

ALBERO DI DECISIONE

- Abbiamo quindi che $h = \Omega(n \log n)$
  (non usiamo $\Theta(n \log n)$ perché h è $\geq$ di $\log_2 n!$)
- **Teorema**. Ogni algoritmo di ordinamento basato sulla comparazione richiede almeno $\Omega(n \log n)$ comparazioni nel caso peggiore
- Come conseguenza sia heapsort che mergesort sono ottimali tra gli algoritmi di ordinamento basati sulla comparazione

DISCUSSIONE

- Abbiamo visto due diversi algoritmi che richiedono lo tempo, $\Theta(n \log n)$
- Abbiamo visto tre diversi algoritmi che richiedono tempo quadratico $O(n^2)$
- Perché è sensato avere più algoritmi con la stessa complessità asintotica? Quando uno può essere meglio dell'altro?