LOWER BOUND DELL'ORDINAMENTO PER CONFRONTI
ORDINAMENTO IN TEMPO LINEARE
COUNTING SORT
RADIX SORT

ALGORITMI E STRUTTURE DATI

LOWER BOUND ALL'ORDINAMENTO PER CONFRONTI

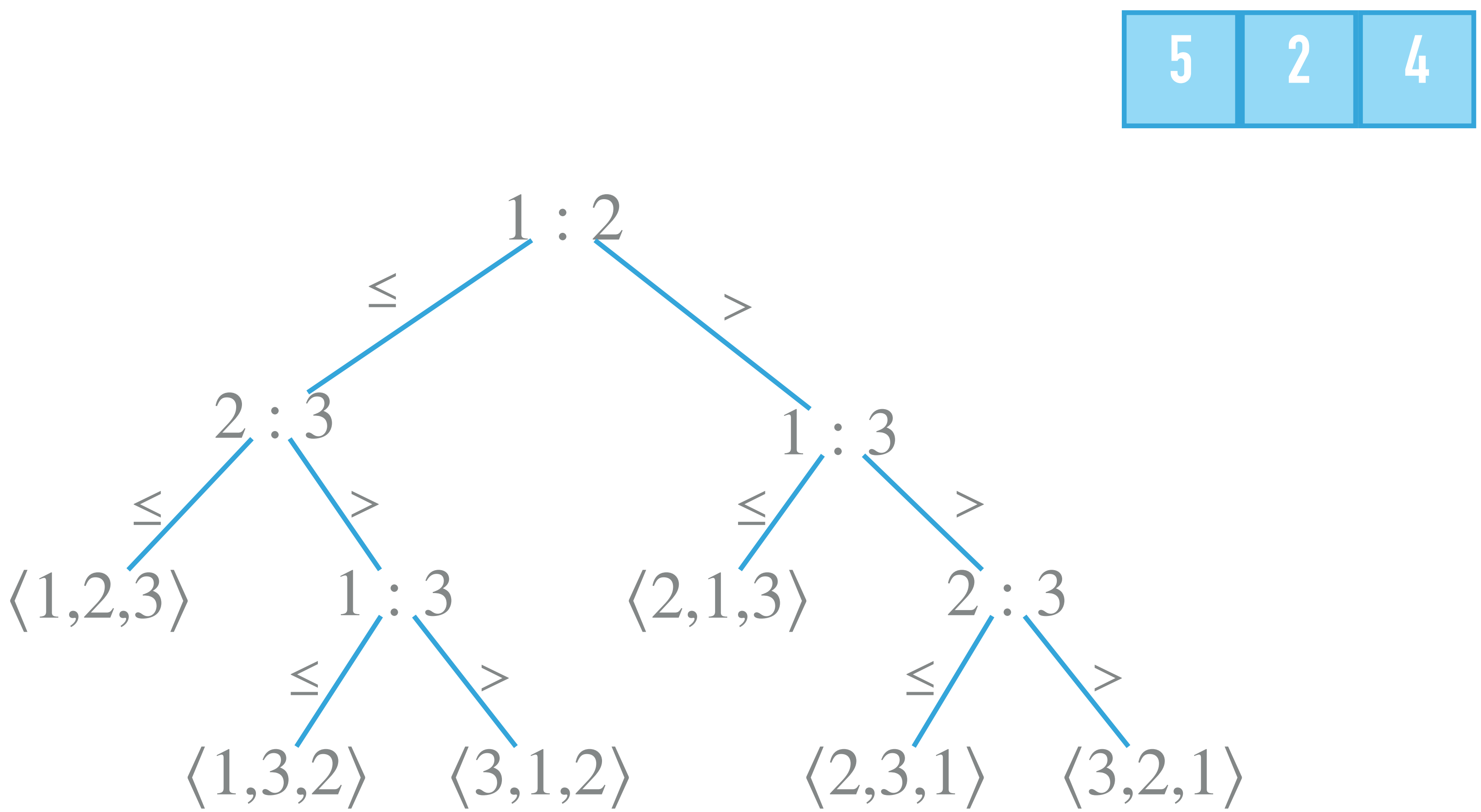
LIMITI DELL'ORDINAMENTO

- 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

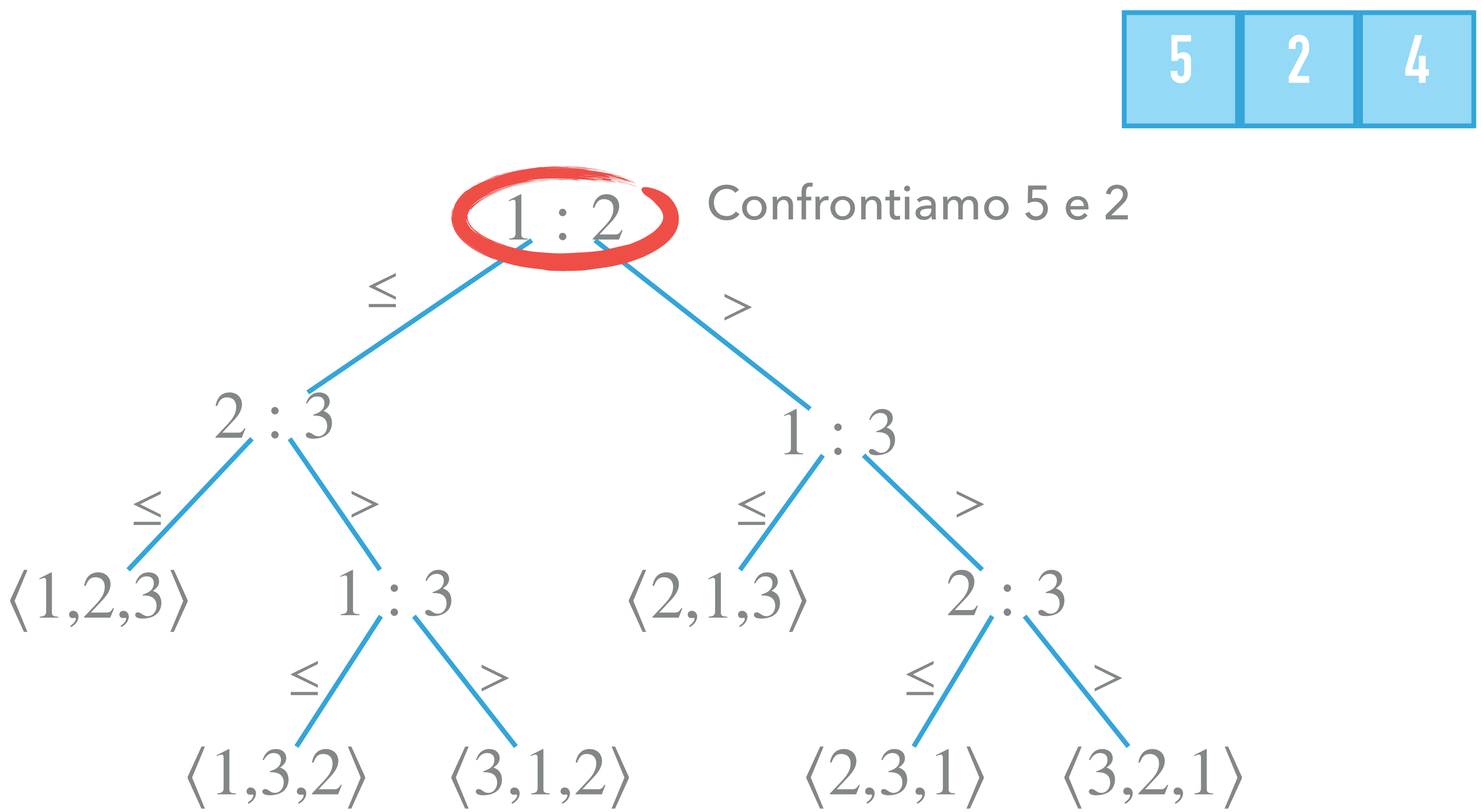
- Dato un array A di n elementi
- Consideriamo un albero binario in cui ogni nodo interno ha associata una comparazione $i : j$, che indica che compariamo l'elemento i-esimo con l'elemento j-esimo
- Ci spostiamo a sinistra se $A[i] \leq A[j]$ e a destra se $A[i] > A[j]$
- Ogni foglia è una permutazione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

ALBERO DI DECISIONE

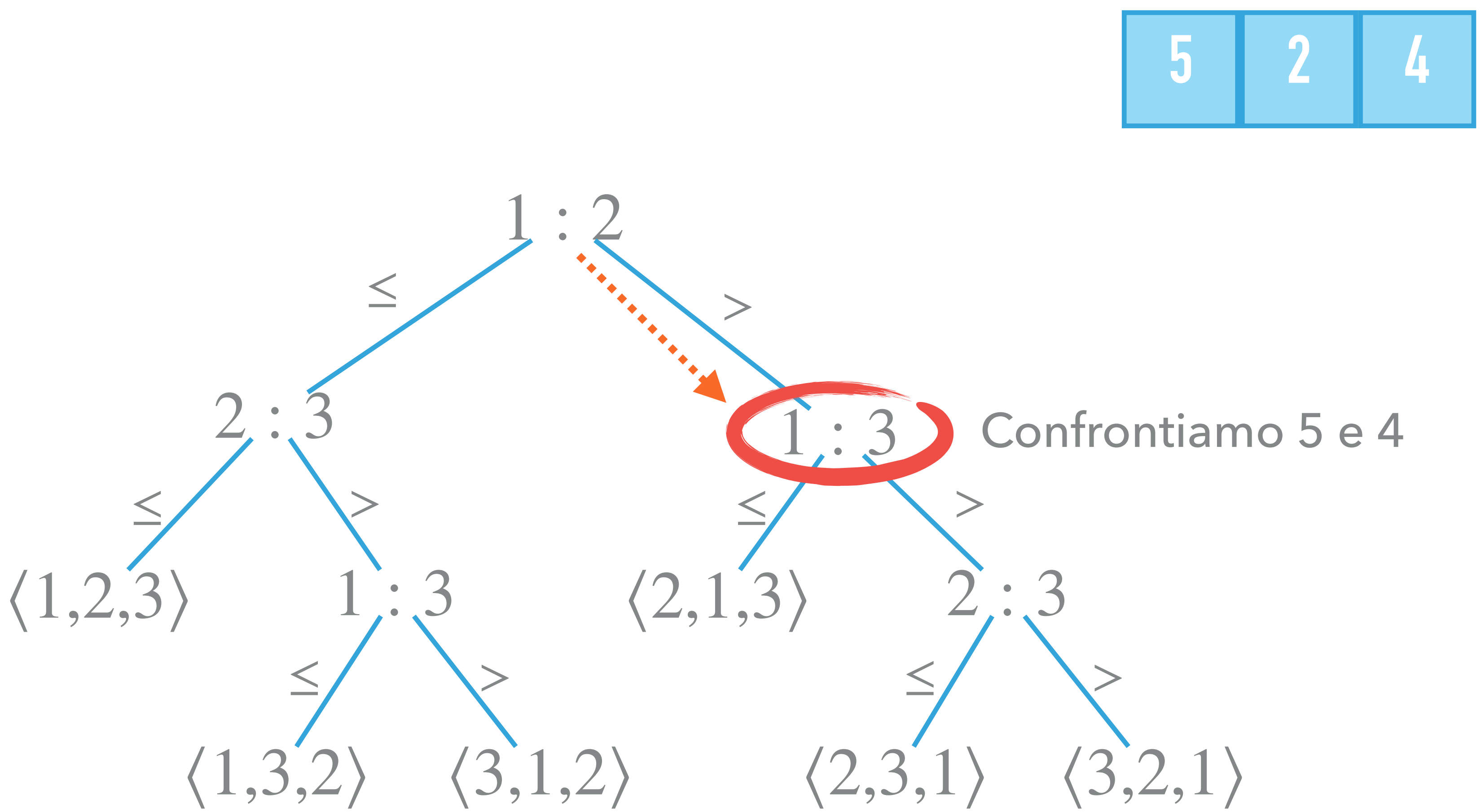


In questo esempio gli indici dell’array partono da 1, come sul libro di testo

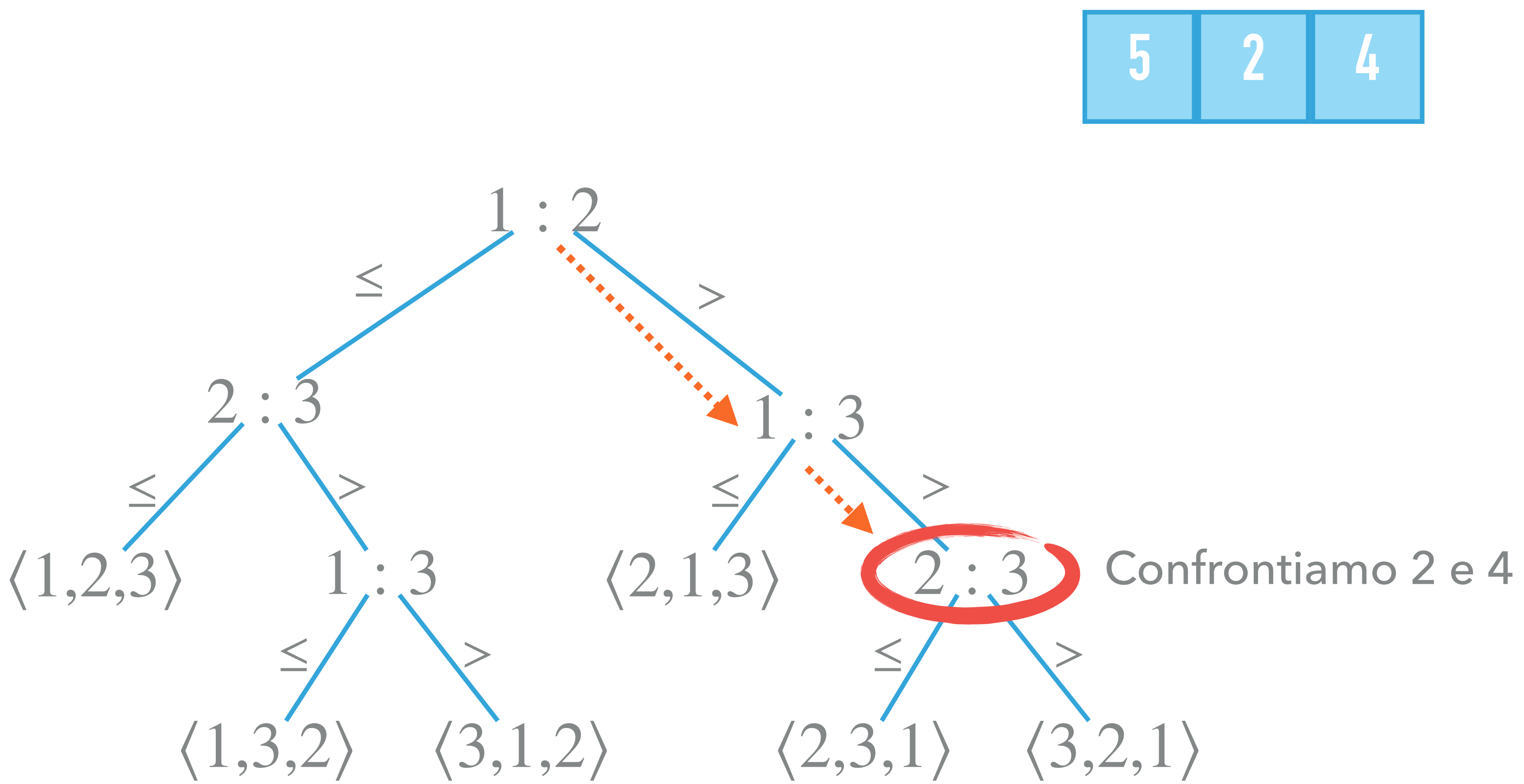
ALBERO DI DECISIONE



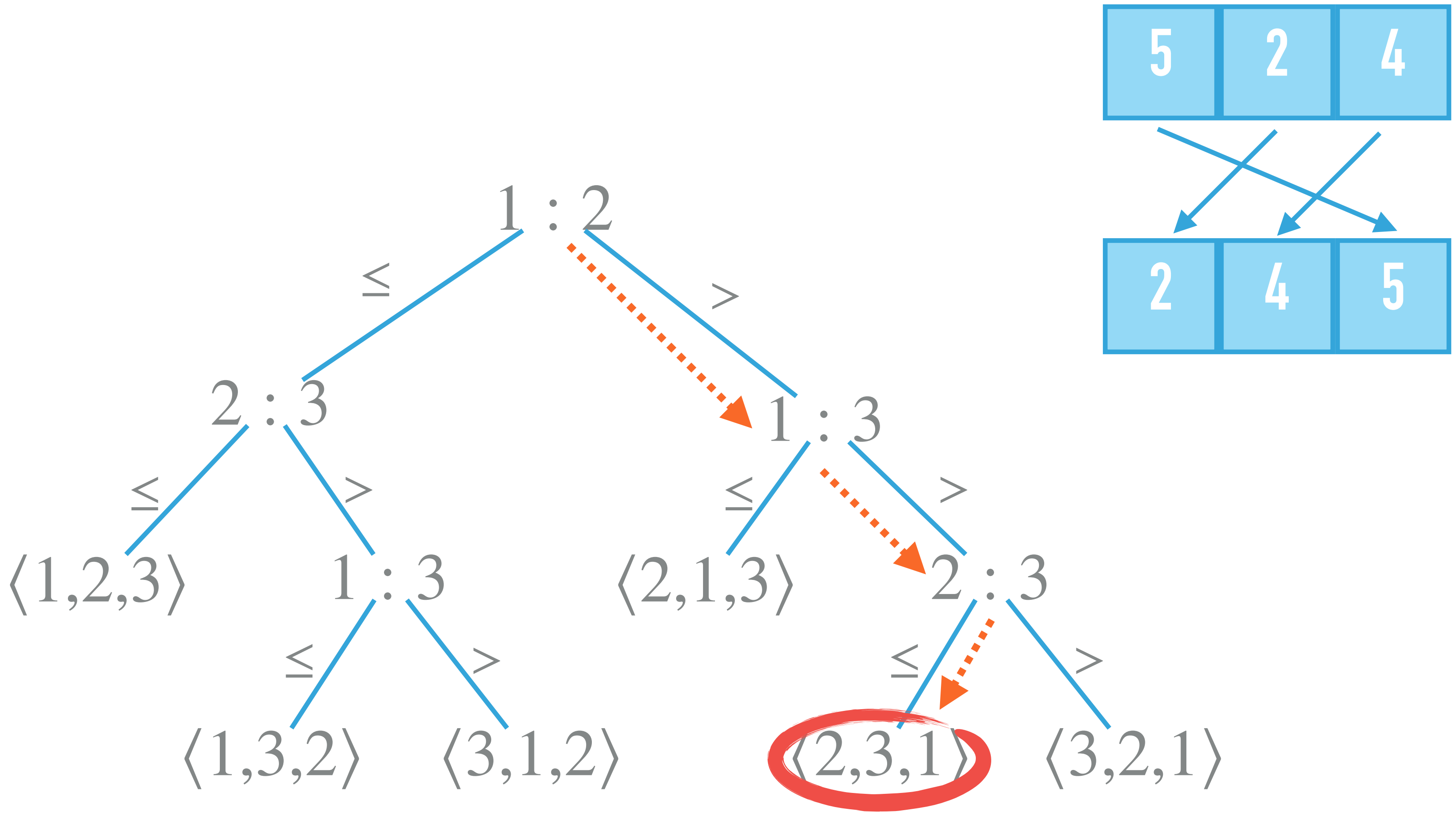
ALBERO DI DECISIONE



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

ORDINAMENTO LINEARE

ORDINARE SENZA COMPARARE

- Abbiamo visto che ogni algoritmo di ordinamento bastato sulla comparazione richiede tempo $\Omega(n \log n)$
- Il punto chiave è "basato sulla comparazione"
- Possiamo scrivere algoritmo di ordinamento che non comparano gli elementi tra di loro
- Questi algoritmi non sono totalmente generici, ma richiedono delle assunzioni aggiuntive

ORDINARE SENZA COMPARARE

- Che tipo di assunzioni sono?
  - Sappiamo esattamente il range degli numeri da ordinare, $[0,k-1]$ e $k = O(n)$. **Counting sort**
  - Sappiamo il numero di cifre necessarie a rappresentare i numeri da ordinare. **Radix sort**
  - I numeri da ordinare sono estratti da una distribuzione uniforme. **Bucket sort**

COUNTING SORT

COUNTING SORT

- Se sappiamo quali sono i valori possibili da ordinare, possiamo contare per ogni valori quanti sono quelli più piccoli e quindi stabilire la posizione finale
- Se, per esempio abbiamo i numeri da 0 a 10 e sappiamo che ci sono 5 numeri più piccoli di 7, sappiamo che se incontriamo 7 dovremo metterlo in sesta posizione
- Questo non richiede di confrontare direttamente i numeri

CONTEGGIO DEI VALORI

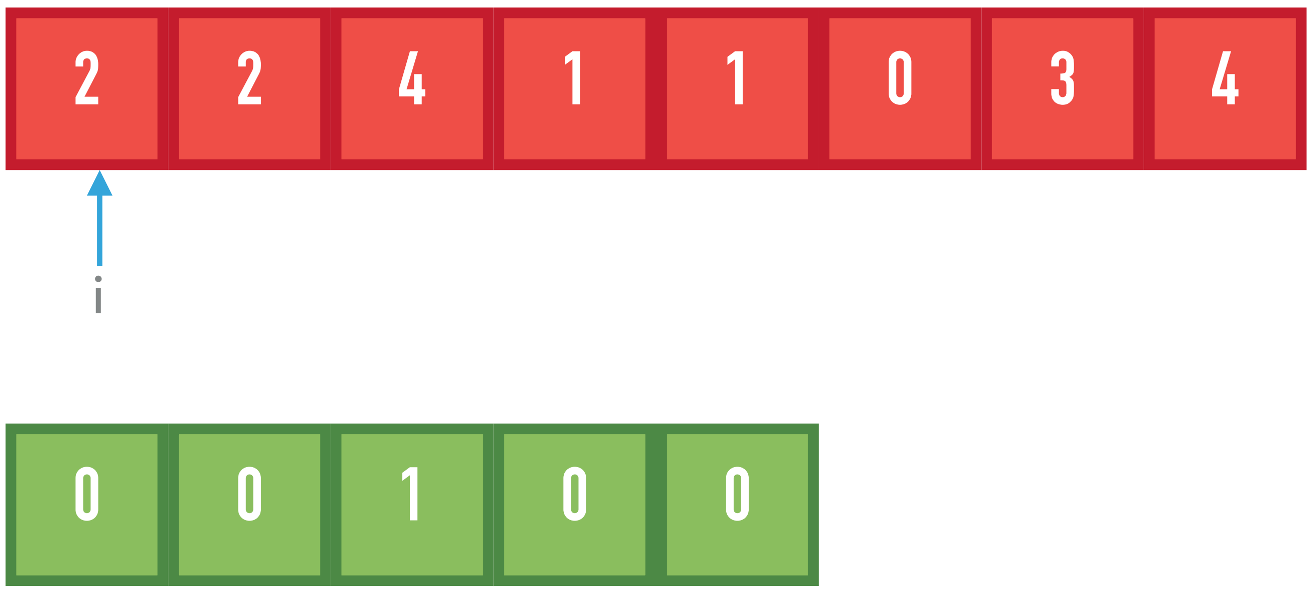
2	2	4	1	1	0	3	4

Supponiamo tutti i valori siano in [0,4]

0	0	0	0	0

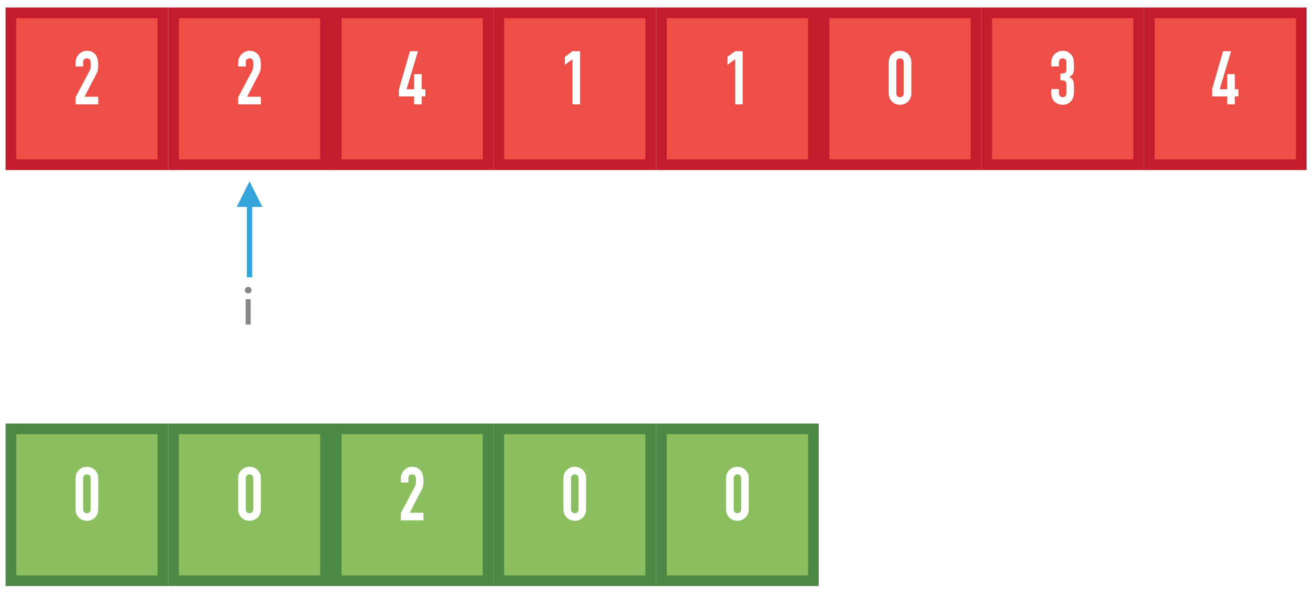
Allochiamo un array di k=5 elementi

CONTEGGIO DEI VALORI



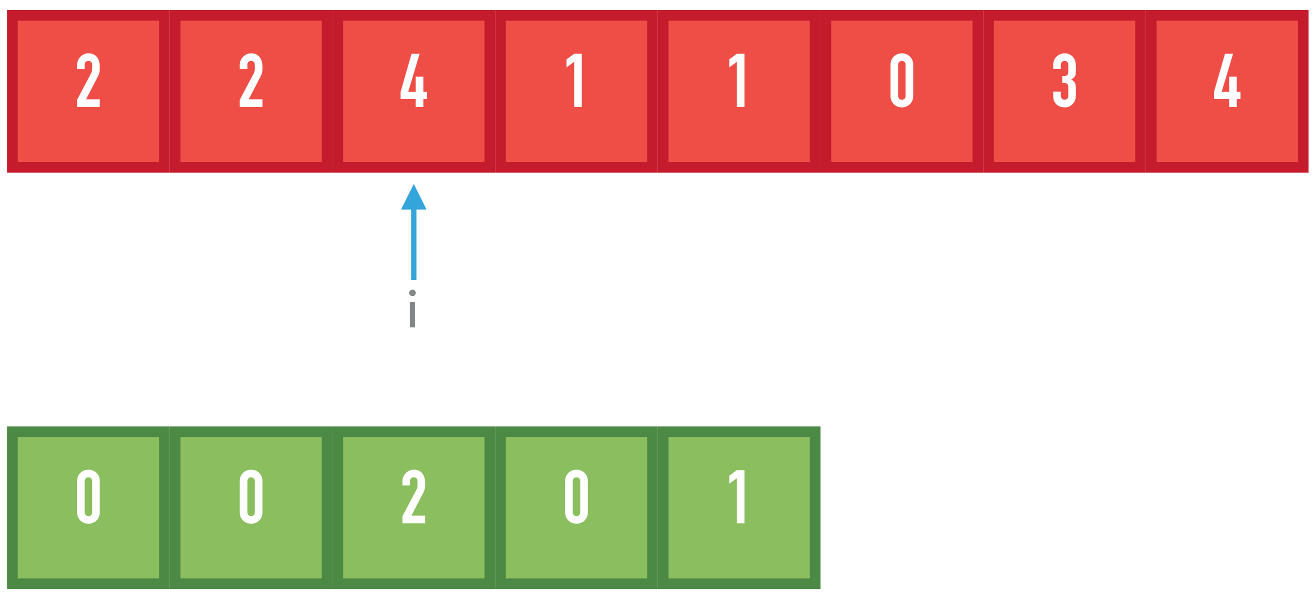
Supponiamo tutti i valori siano in [0,4]

CONTEGGIO DEI VALORI



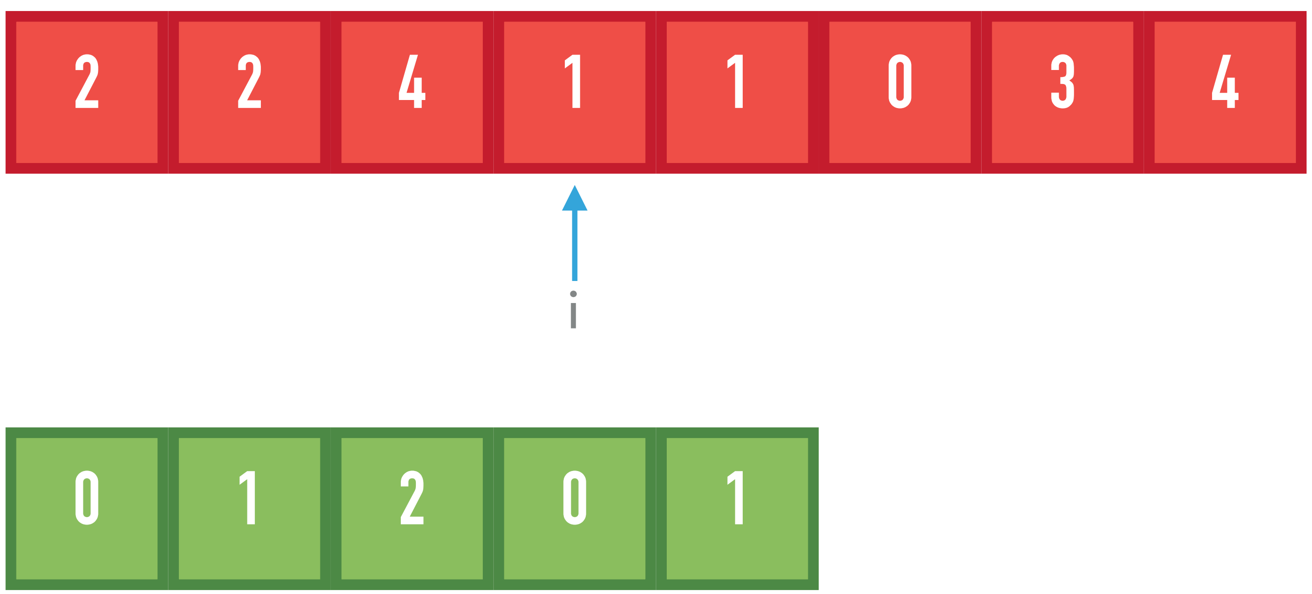
Supponiamo tutti i valori siano in [0,4]

CONTEGGIO DEI VALORI



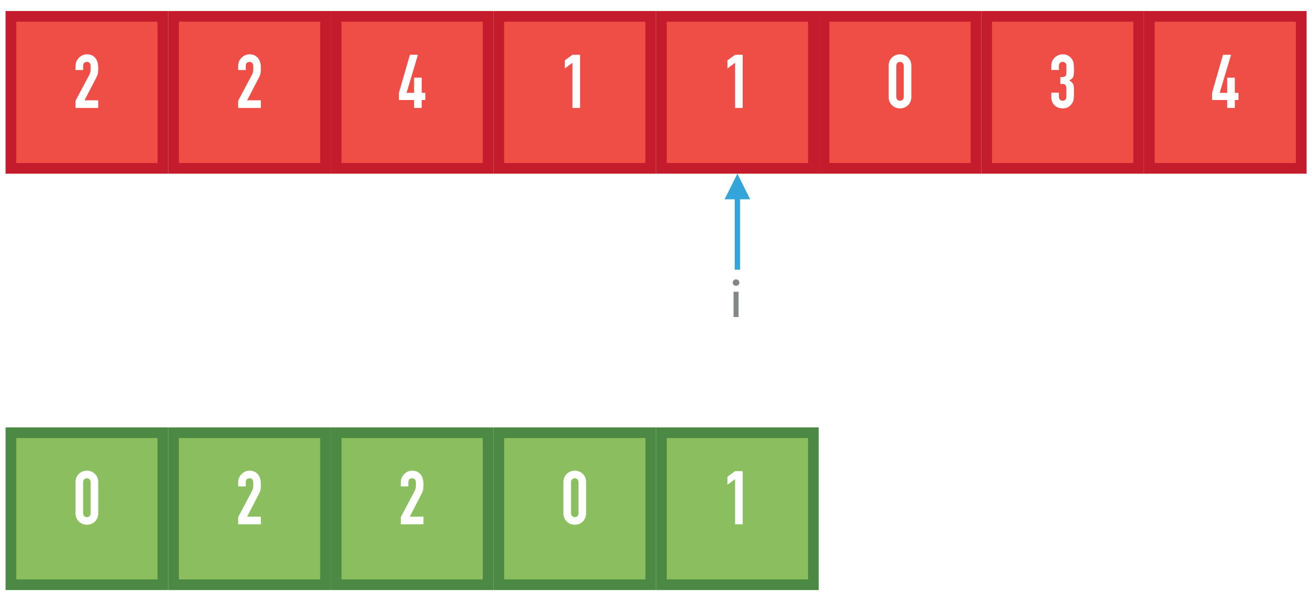
Supponiamo tutti i valori siano in [0,4]

CONTEGGIO DEI VALORI



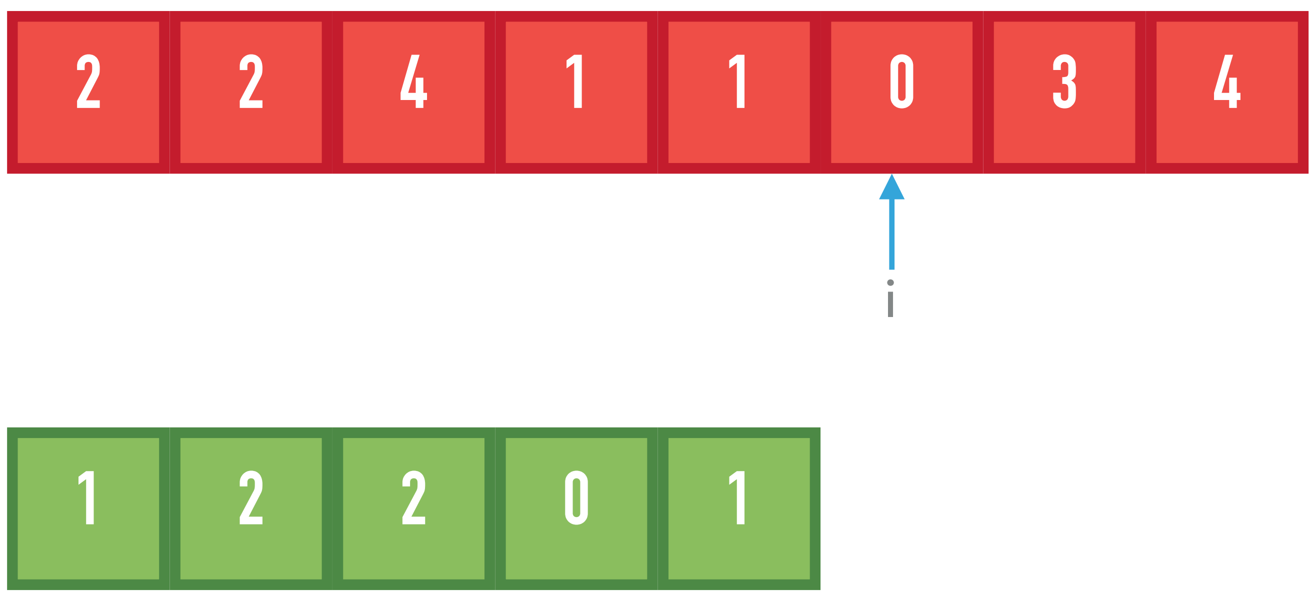
Supponiamo tutti i valori siano in [0,4]

CONTEGGIO DEI VALORI



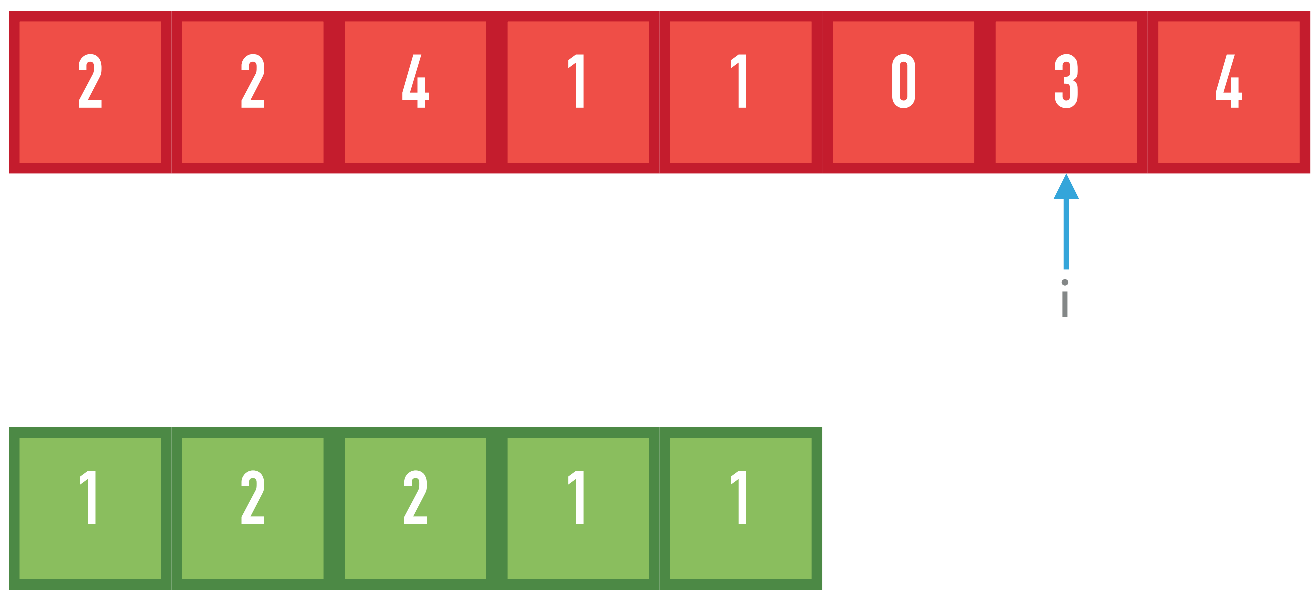
Supponiamo tutti i valori siano in [0,4]

CONTEGGIO DEI VALORI



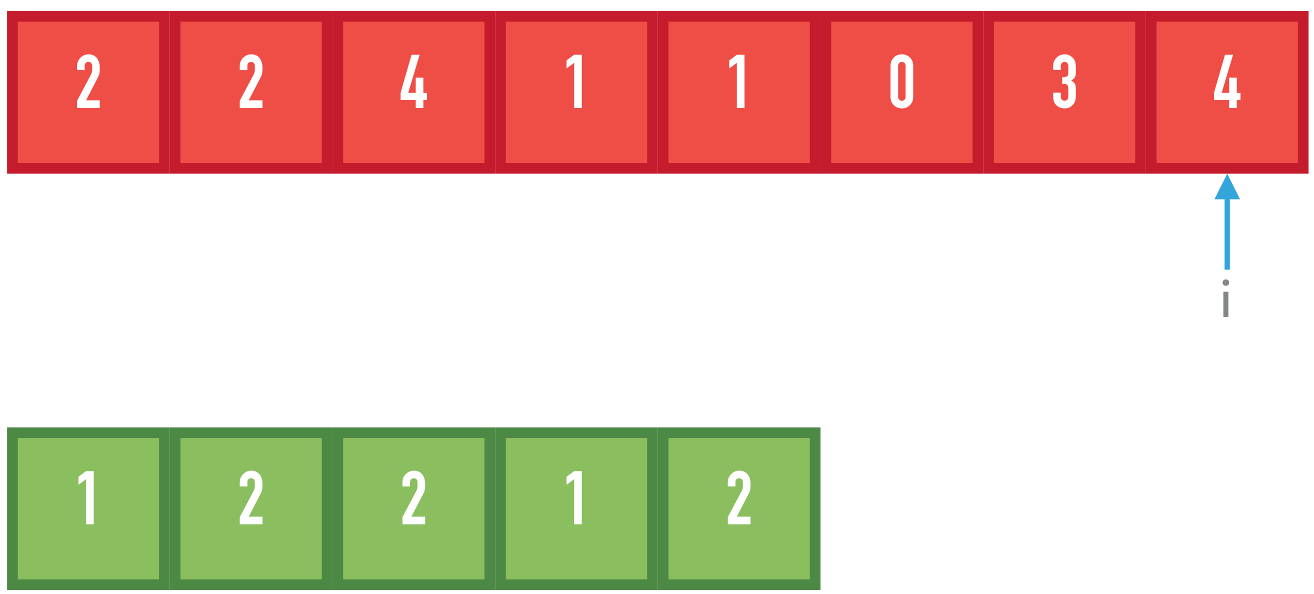
Supponiamo tutti i valori siano in [0,4]

CONTEGGIO DEI VALORI



Supponiamo tutti i valori siano in [0,4]

CONTEGGIO DEI VALORI



Supponiamo tutti i valori siano in [0,4]

Ora abbiamo a disposizione per ogni valore in [0,k] il numero di occorrenze

Vogliamo contare per ogni valore il numero di elementi **minori o uguali** a quel valore che sono presenti nell'array di partenza

CONTEGGIO DEI VALORI

2	2	4	1	1	0	3	4

Supponiamo tutti i valori siano in [0,4]

1	3	2	1	2

i

B[i] = B[i] + B[i-1]

CONTEGGIO DEI VALORI

2	2	4	1	1	0	3	4

Supponiamo tutti i valori siano in [0,4]

1	3	5	1	2

i

B[i] = B[i] + B[i-1]

CONTEGGIO DEI VALORI

2	2	4	1	1	0	3	4

Supponiamo tutti i valori siano in [0,4]

1	3	5	6	2

i

B[i] = B[i] + B[i-1]

CONTEGGIO DEI VALORI

2	2	4	1	1	0	3	4

Supponiamo tutti i valori siano in [0,4]

1	3	5	6	8

i

B[i] = B[i] + B[i-1]

Ora abbiamo in B[i] l'ultima posizione in cui il valore "i" deve apparire

ORDINAMENTO

2	2	4	1	1	0	3	4

Supponiamo tutti i valori siano in [0,4]

i

Scorriamo dal fondo

1	3	5	6	8

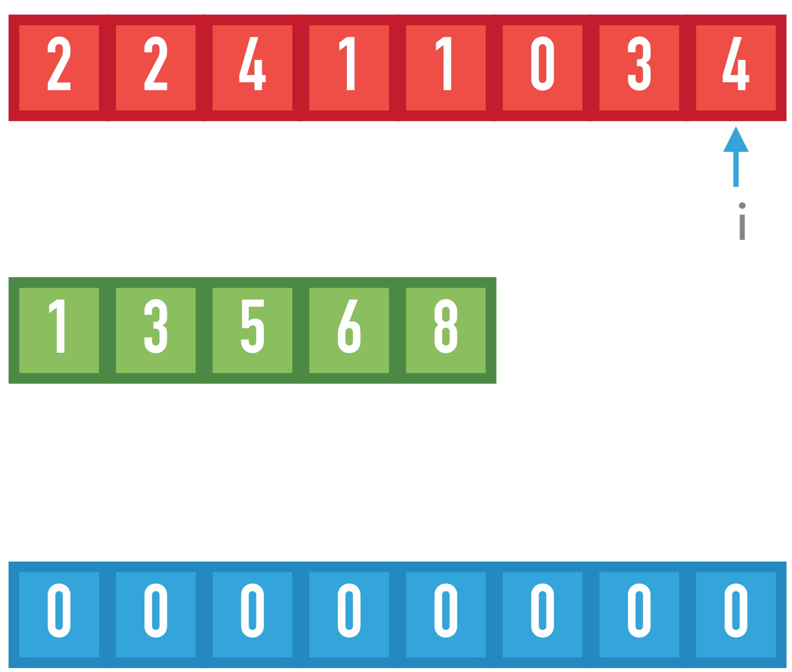
B[A[i]] = B[A[i]] - 1

QUESTA OPERAZIONE RICHIEDE UNA SPIEGAZIONE

0	0	0	0	0	0	0	0

C[B[A[i]]-1] = A[i]

ORDINAMENTO

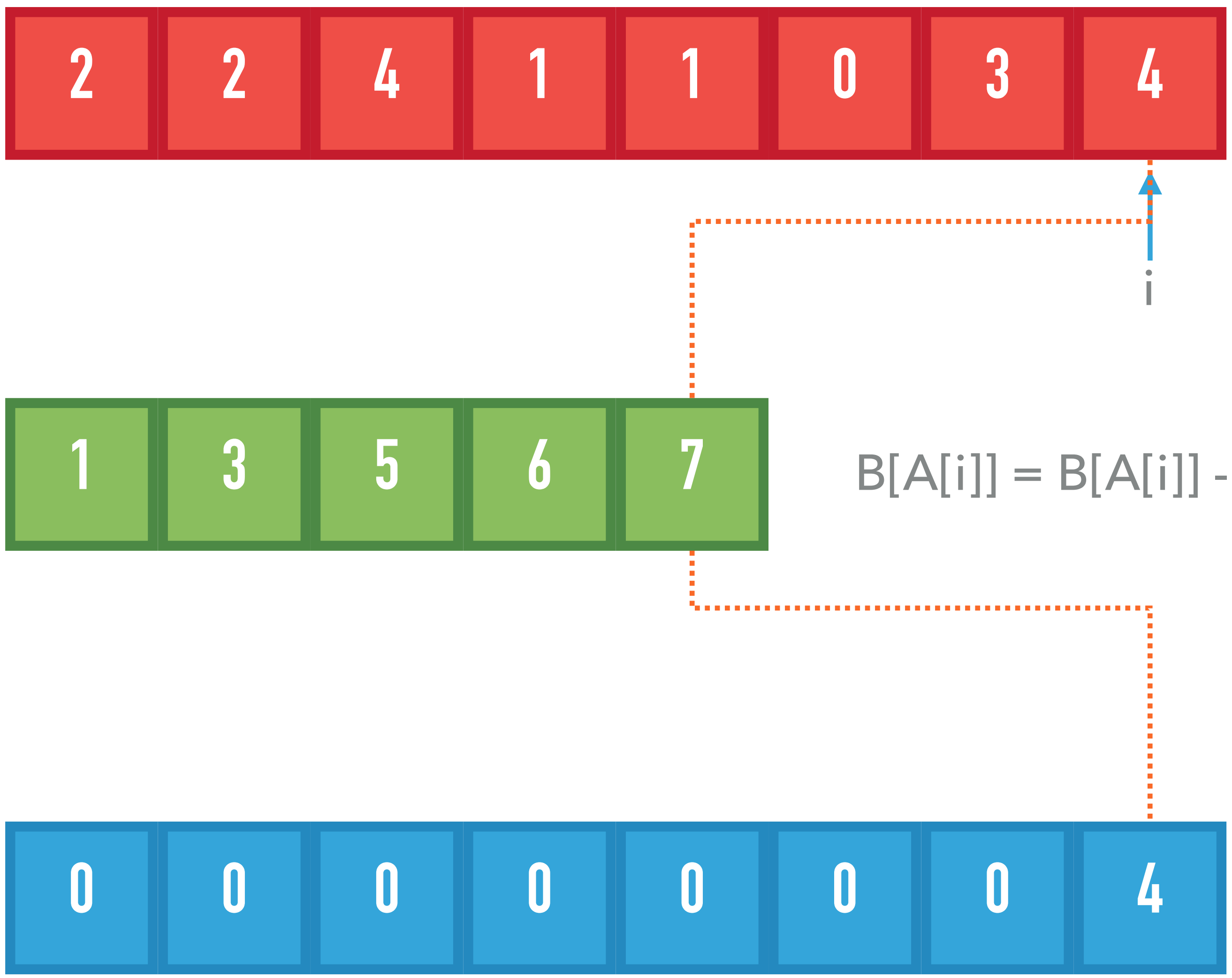


Elemento da ordinare

$$C[B[A[i]]-1] = A[i]$$

Posizione in cui mettere l'elemento
(indicizzata dall'elemento stesso)

ORDINAMENTO



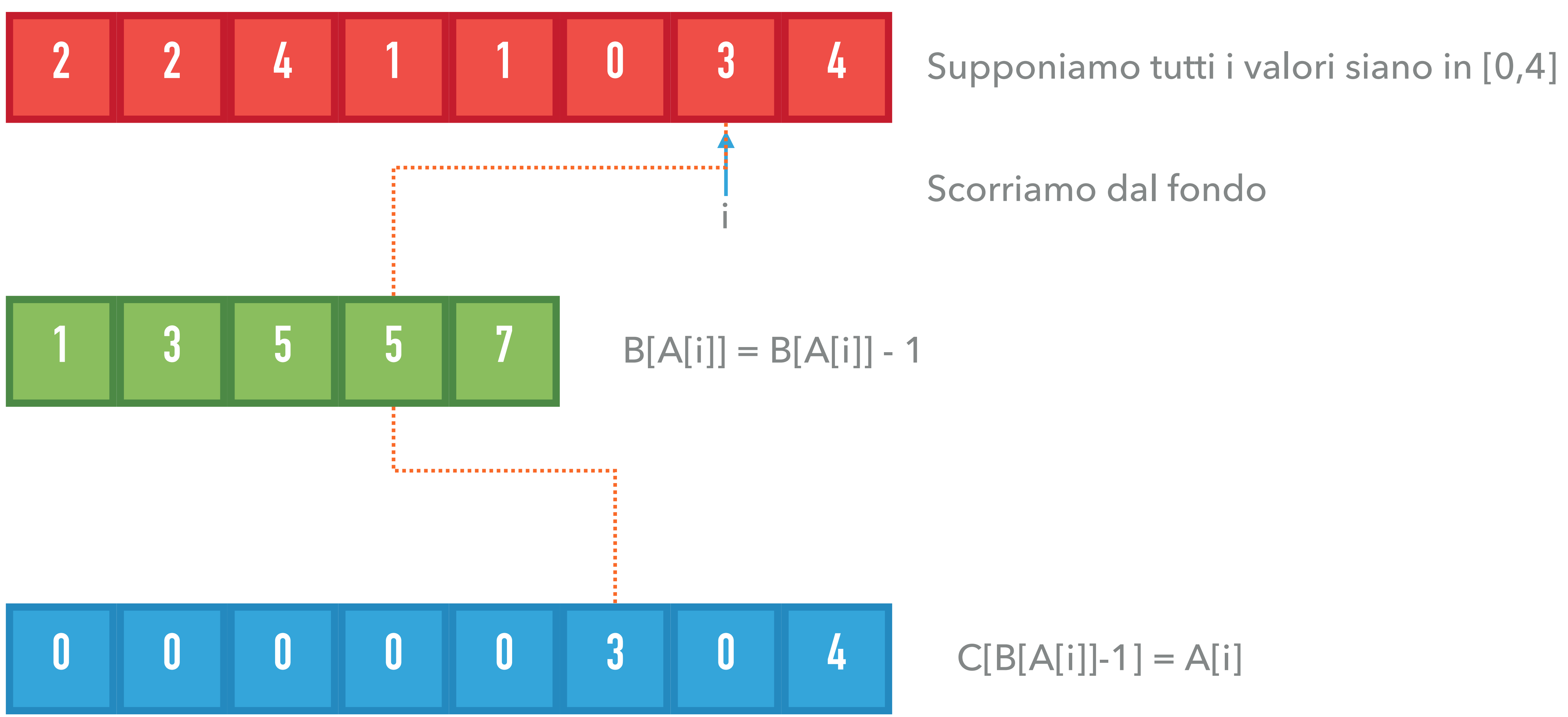
Supponiamo tutti i valori siano in [0,4]

Scorriamo dal fondo

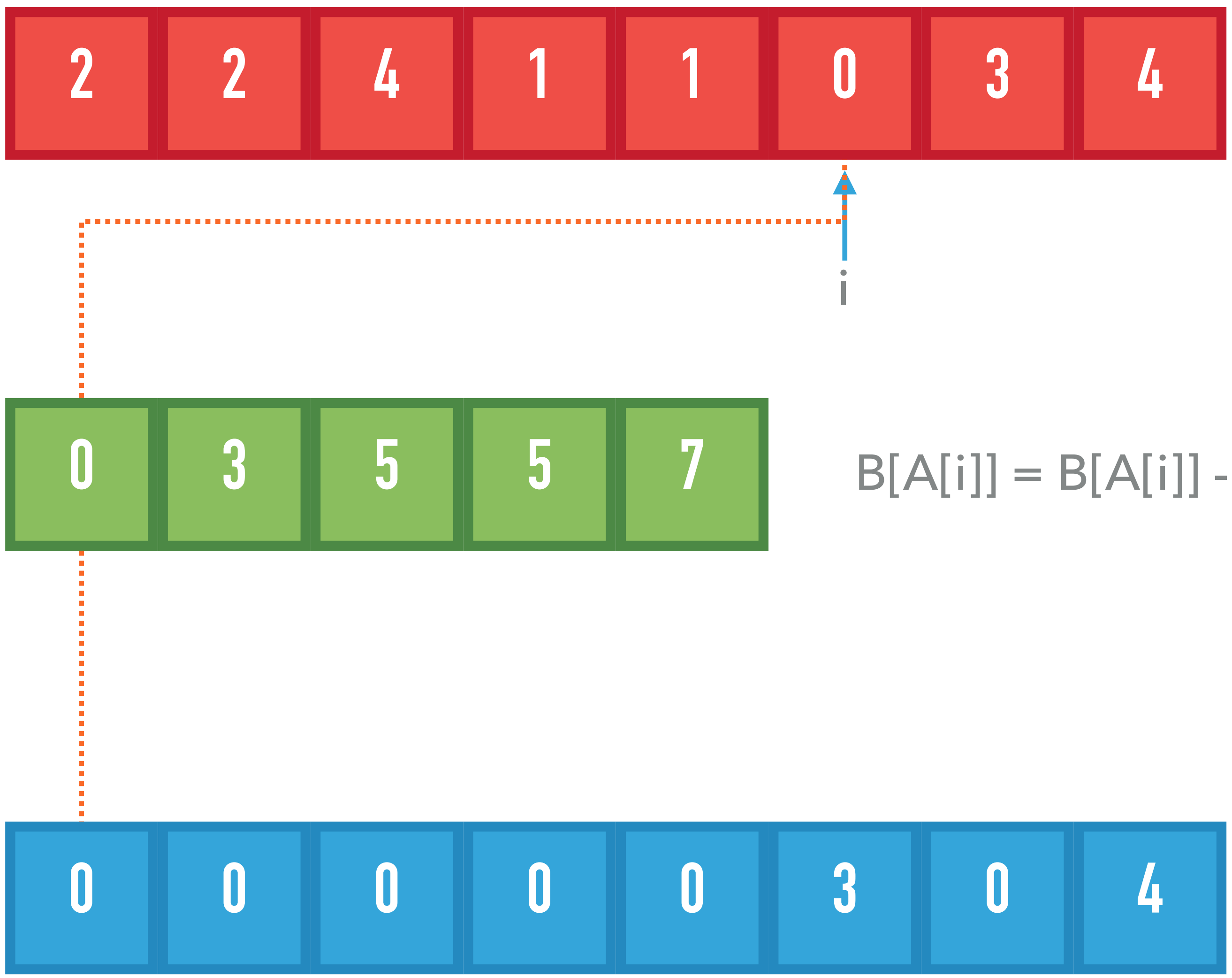
B[A[i]] = B[A[i]] - 1

C[B[A[i]]-1] = A[i]

ORDINAMENTO



ORDINAMENTO



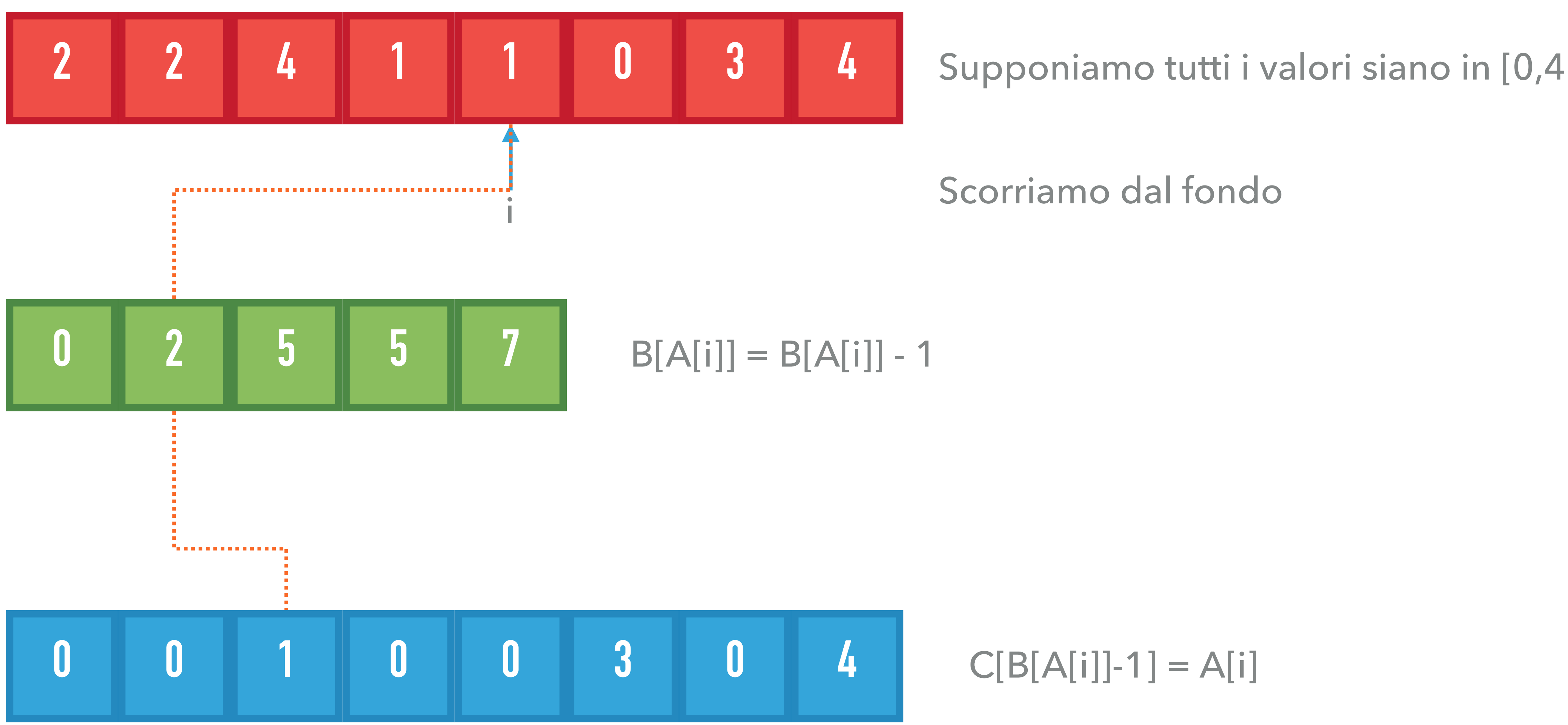
Supponiamo tutti i valori siano in [0,4]

Scorriamo dal fondo

B[A[i]] = B[A[i]] - 1

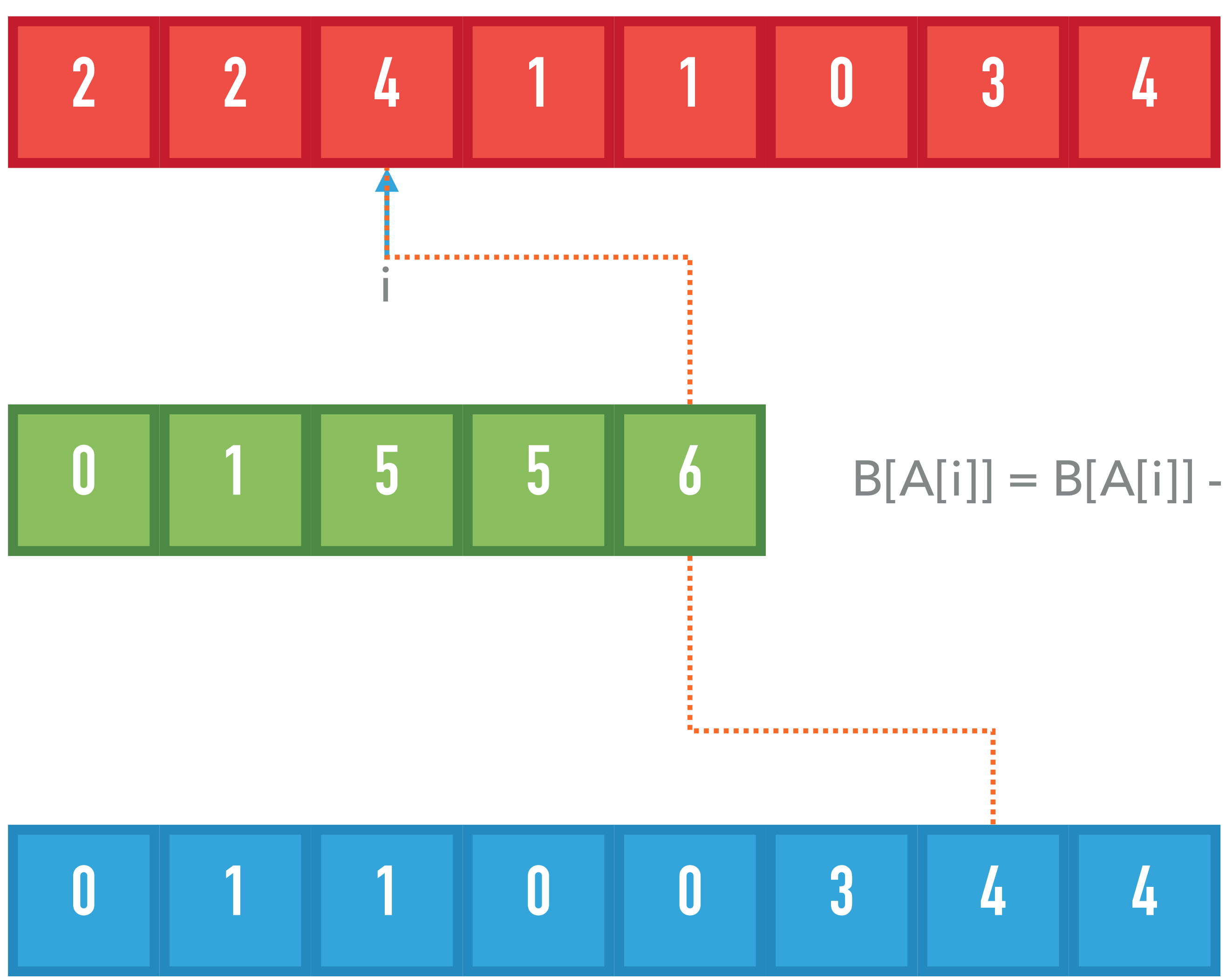
C[B[A[i]]-1] = A[i]

ORDINAMENTO



ORDINAMENTO

ORDINAMENTO

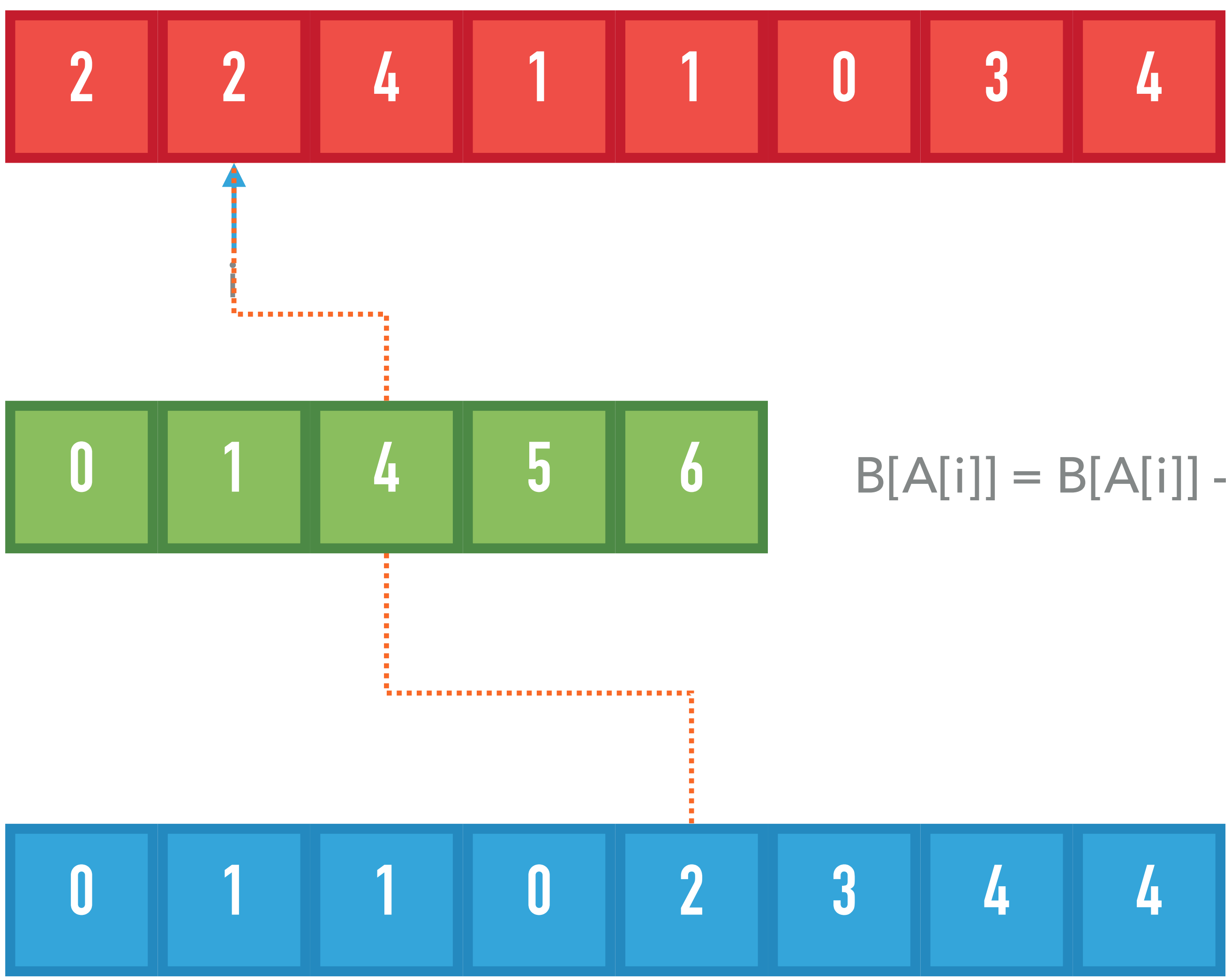


Supponiamo tutti i valori siano in [0,4]

Scorriamo dal fondo

C[B[A[i]]-1] = A[i]

ORDINAMENTO



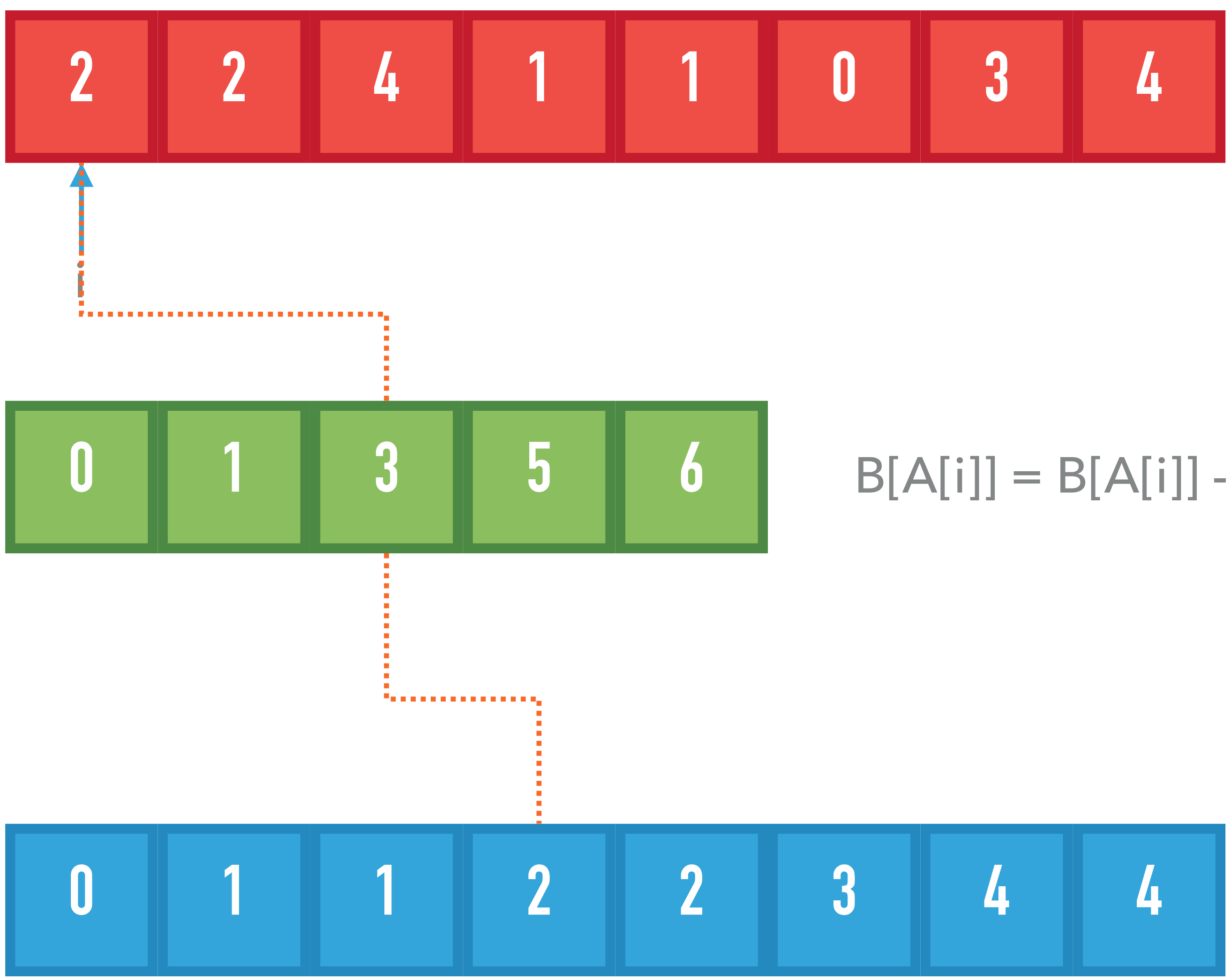
Supponiamo tutti i valori siano in [0,4]

Scorriamo dal fondo

B[A[i]] = B[A[i]] - 1

C[B[A[i]]-1] = A[i]

ORDINAMENTO



Supponiamo tutti i valori siano in [0,4]

Scorriamo dal fondo

B[A[i]] = B[A[i]] - 1

C[B[A[i]]-1] = A[i]

COUNTING SORT: PSEUDOCODICE

```
Argomenti: A (array), k (valore massimo)
B = array di k elementi inizialmente zero
C = array di len(A) elementi
for i in range(0, len(A))
   B[A[i]] = B[A[i]] + 1 # incrementa il numero di valori A[i] trovati
for i in range(1,k)
   B[i] = B[i] + B[i-1] # in modo da avere in B[i] il numero di elementi ≤ i
for i in range(len(A), -1, -1) # contiamo dalla fine all'inizio
   C[B[A[i]]-1] = A[i] # trasferiamo A[i] nella sua posizione in C
   B[A[i]] = B[A[i]] - 1
return C
```

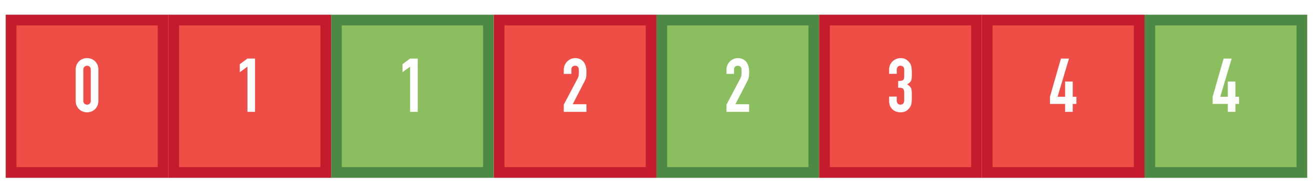
COUNTING SORT: COMPLESSITÀ

- Questa volta la complessità è funzione di **due** parametri: n (la dimensione dell'array) e k (il numero di valori distinti)
- Due cicli for che sono lunghi n cicli
- Un ciclo for che è lungo $k - 1$ cicli
- Risultato: $\Theta(n + k)$
- Se abbiamo che $k = O(n)$ otteniamo che l'algoritmo richiede tempo $\Theta(n)$

ORDINAMENTO STABILE

- Possiamo fare una distinzione aggiuntiva tra ordinamenti stabili e non stabili
- Un ordinamento stabile preserva l’ordine relativo di elementi con lo stesso valore

Ordiniamo solo guardando i numeri ma le celle verdi vengono sempre dopo le celle rosse con lo stesso valore



Un ordinamento stabile preserva questa proprietà

Un ordinamento non stabile potrebbe non preservarla

ORDINAMENTO STABILE

- In diversi casi la stabilità dipende da dettagli implementativi, in particolare da come sono trattati i valori identici
- In particolare a noi servirà avere il counting sort stabile
- La nostra implementazione lo è…
- …perché inseriamo i valori a partire dal fondo
- Se avessi cambiato l’ordine di inserimento non sarebbe stato stabile

RADIX SORT

RADIX SORT

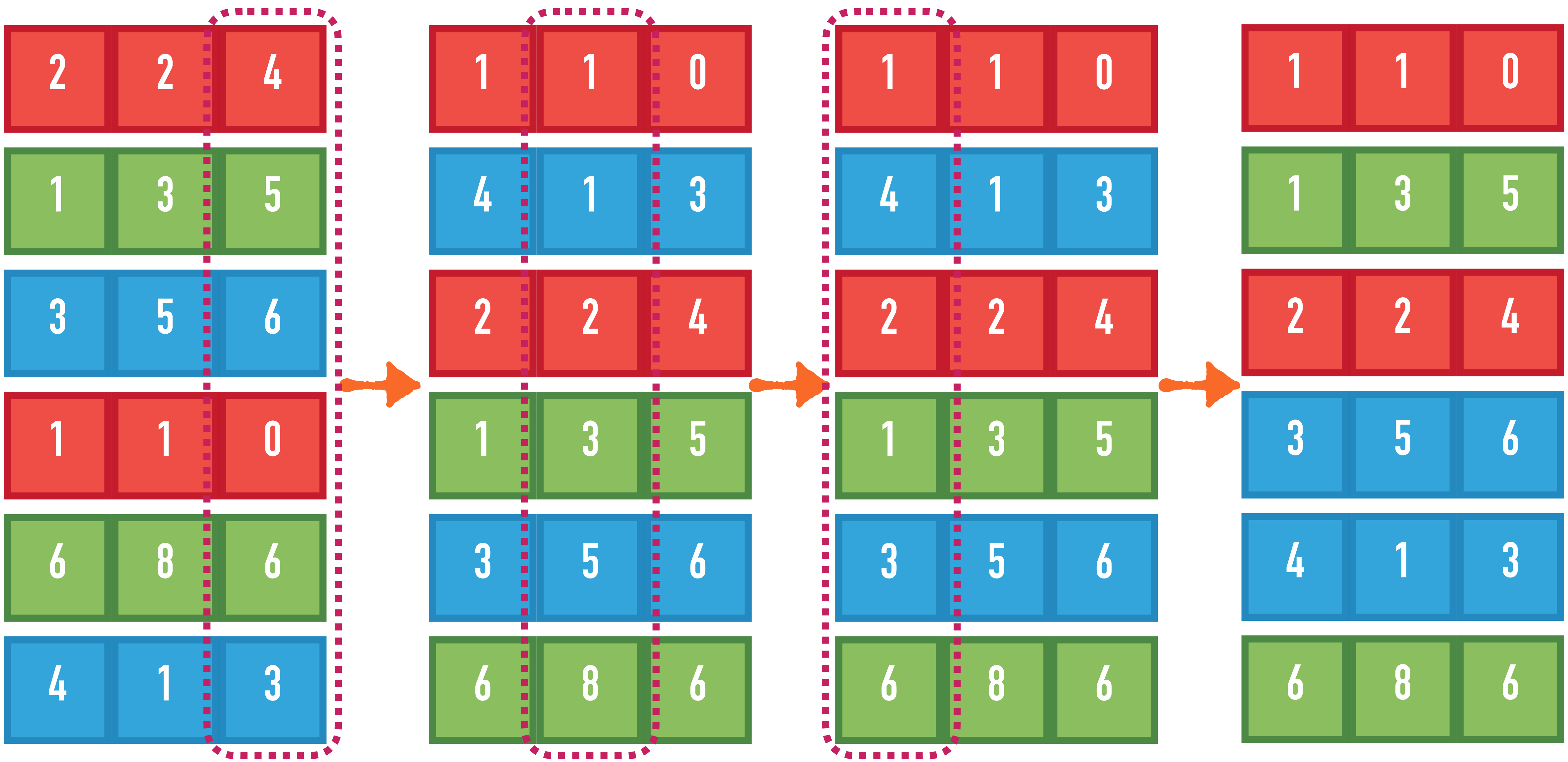
- Il radix sort richiede un ordinamento stabile
- È nato per ordinare le schede perforate, quindi le prime implementazioni di radix sort non erano con il codice, ma con strumenti meccanici!
- Idea di base: se abbiamo numeri di d cifre possiamo ordinarli una cifra alla volta usando un algoritmo di ordinamento stabile

ESEMPIO DI RADIX SORT

Ordiniamo rispetto all'ultima cifra

Ordiniamo rispetto alla penultima cifra

Ordiniamo rispetto alla prima cifra



RADIX SORT: PSEUDOCODICE

```
Argomenti: A (array), k (cifre), d (numero di cifre)
for i in range(d-1,-1,-1) # dall'ultima alla prima cifra
     ordina stabilmente rispetto alla cifra i-esima
```

- Dato che le possibili cifre sono limitate, la scelta dell'ordinamento stabile è solitamente il **counting sort**

RADIX SORT: PERCHÉ FUNZIONA?

- Dopo aver ordinato per la cifra meno significativa abbiamo tutti i numeri che terminano con 0 che precedono quelli che terminano con 1, etc.
- Dopo aver ordinato rispetto alla penultima cifra abbiamo tutti i numeri che terminano con 00 che precedono quelli che che terminano con 01, …, 10, 11, …, 99
- In questo punto è importante che l'ordinamento sia stabile, altrimenti non è detto che sia preservato l'ordine relativo sull'ultima cifra!
- Finito di ordinare su tutte le d cifre abbiamo che gli elementi risultano ordinati

RADIX SORT: COMPLESSITÀ

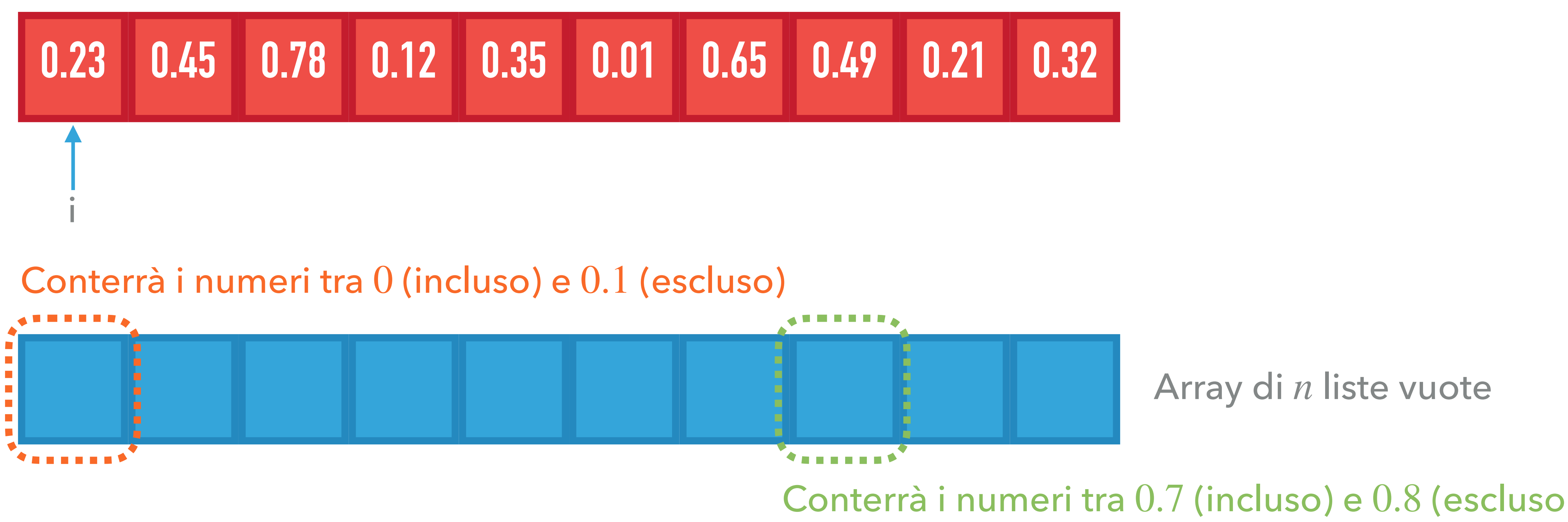
- Questa volta abbiamo tre parametri: n, k e d
- Se usiamo counting sort all'interno del ciclo for, ogni ordinamento stabile rispetto ad una cifra ha costo $\Theta(n + k)$
- Dato che dobbiamo ripetere questa procedura per d cifre, la complessità temporale risultante è $\Theta(d(n + k))$
- Quando k è $O(n)$ – o addirittura costante, abbiamo $\Theta(dn)$
- Quindi dipende tutto da quante cifre abbiamo, per esempio se d è costante, radix sort richiede tempo $\Theta(n)$

BUCKET SORT

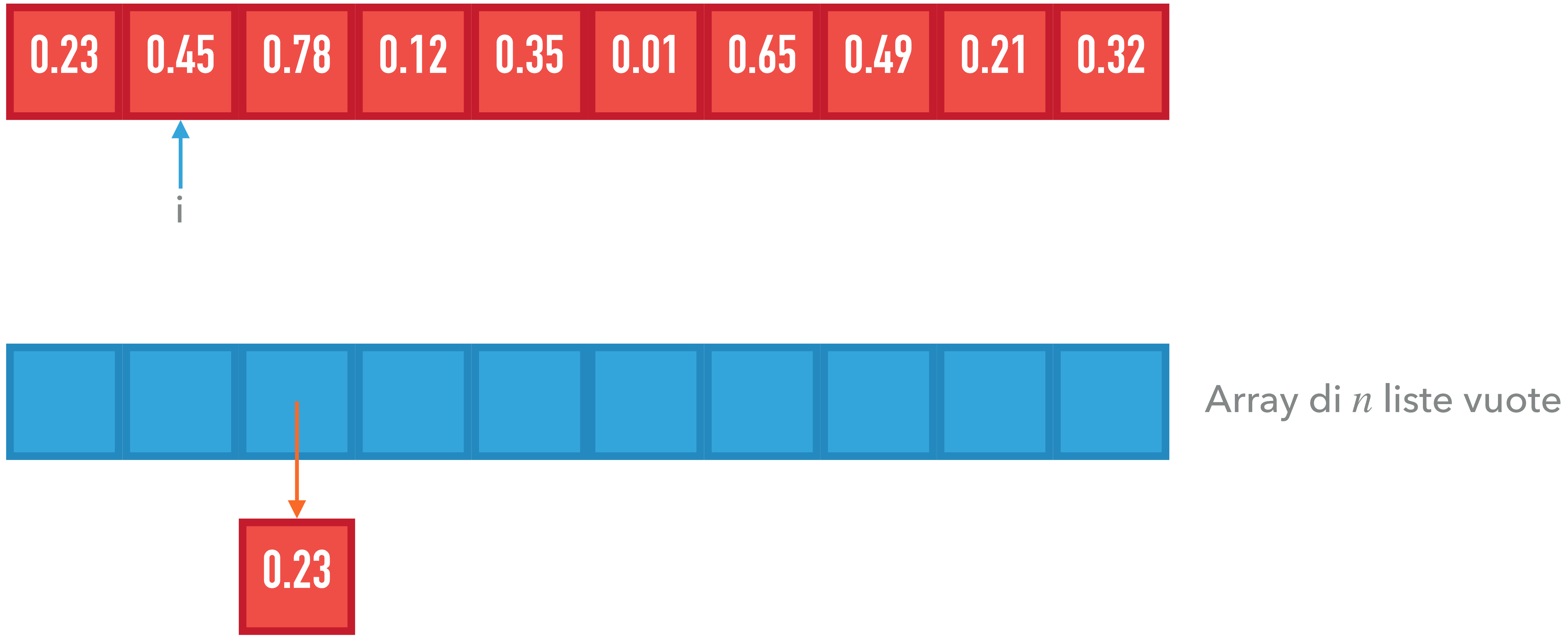
BUCKET SORT

- Bucket sort richiede che l'input sia distribuito uniformemente in un intervallo. Noi useremo [0,1)
- Sotto questa ipotesi il tempo medio richiesto da bucket sort è $\Theta(n)$
- L'idea è di usare, per un array di n elementi, n "secchi" distinti in cui inserire i valori
- Ordiniamo ciascuno dei "secchi" con insertion sort
- Concateniamo ciascuno dei "secchi" in ordine

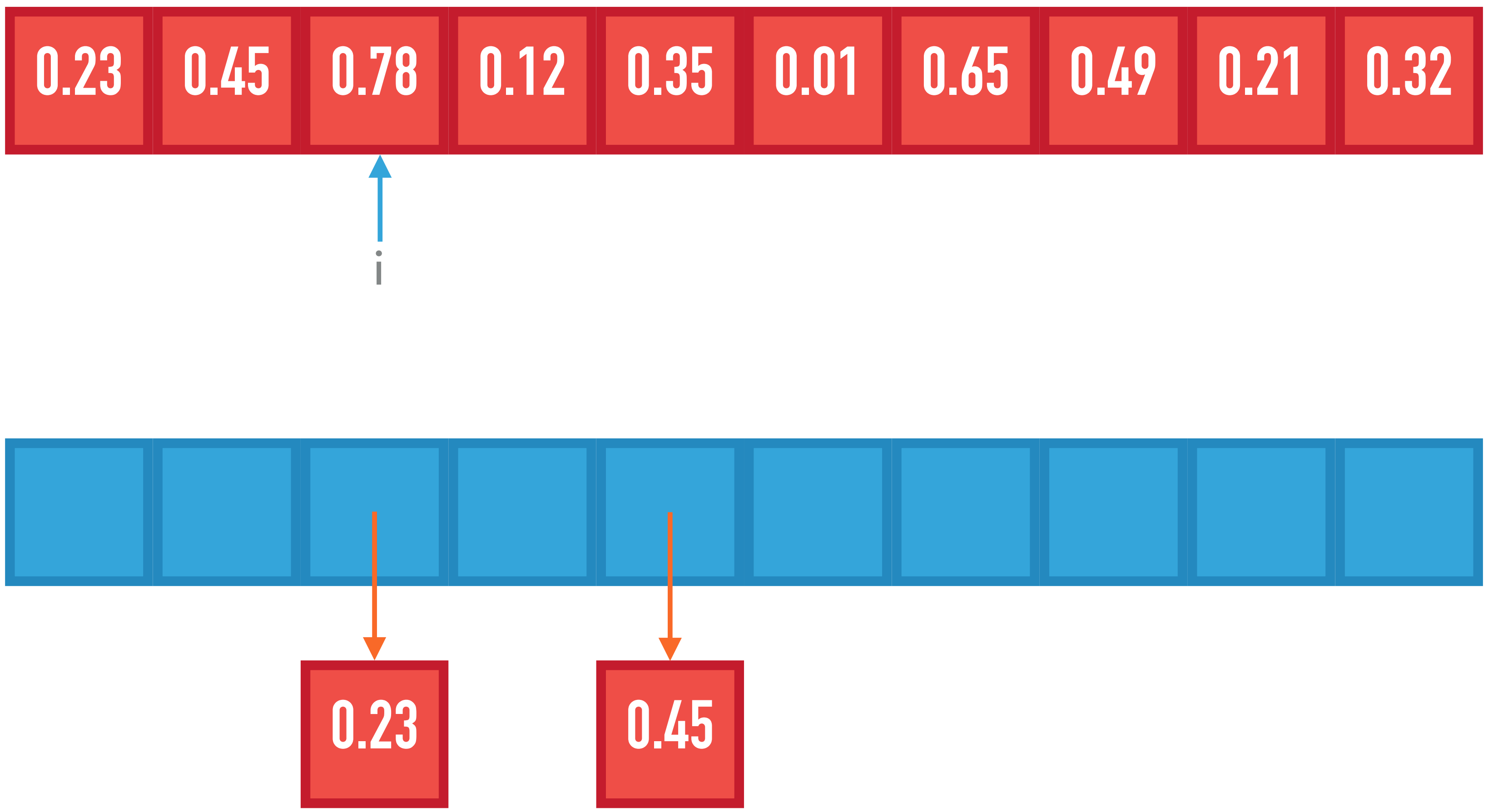
ORDINAMENTO



ORDINAMENTO

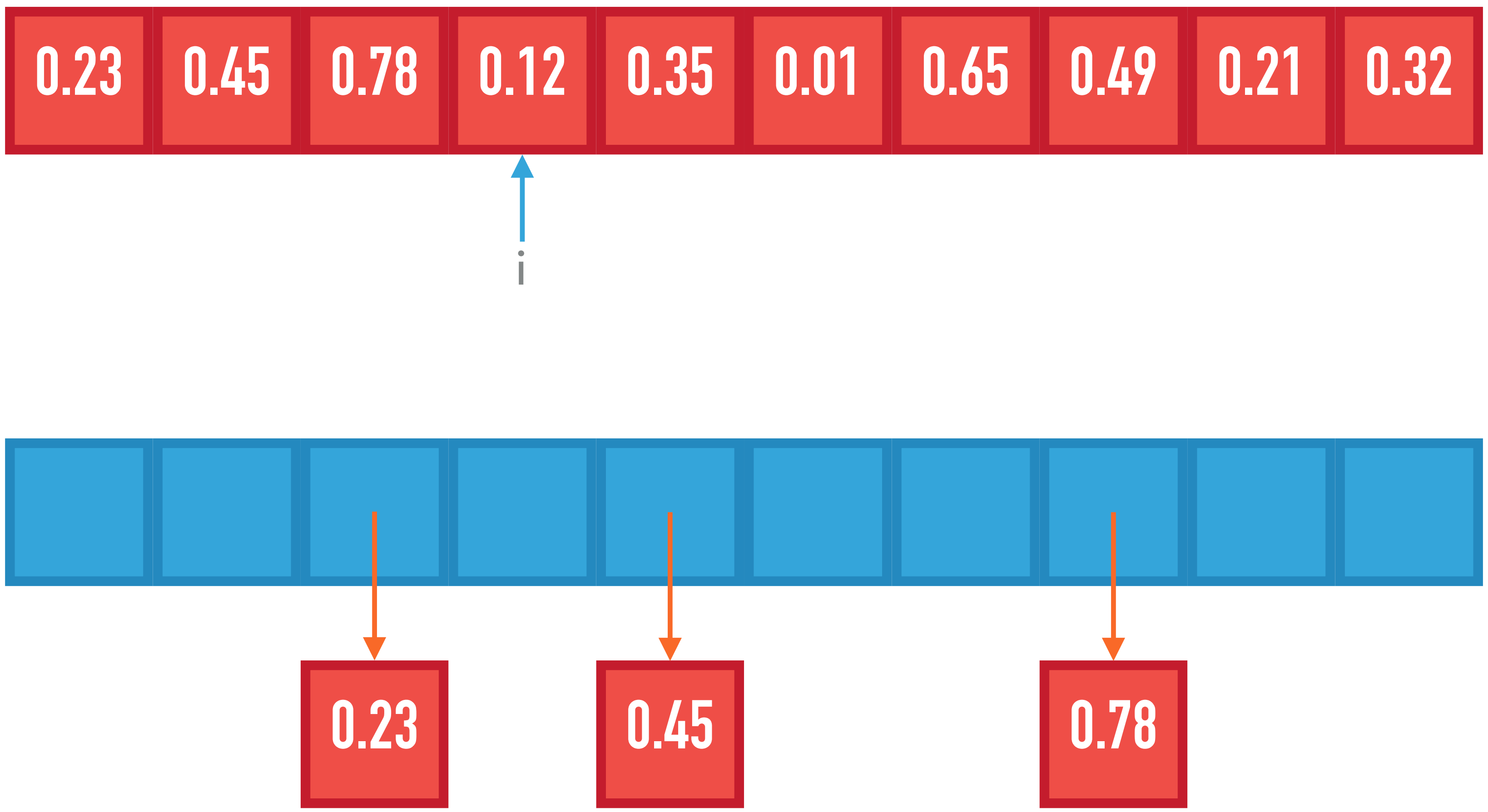


ORDINAMENTO



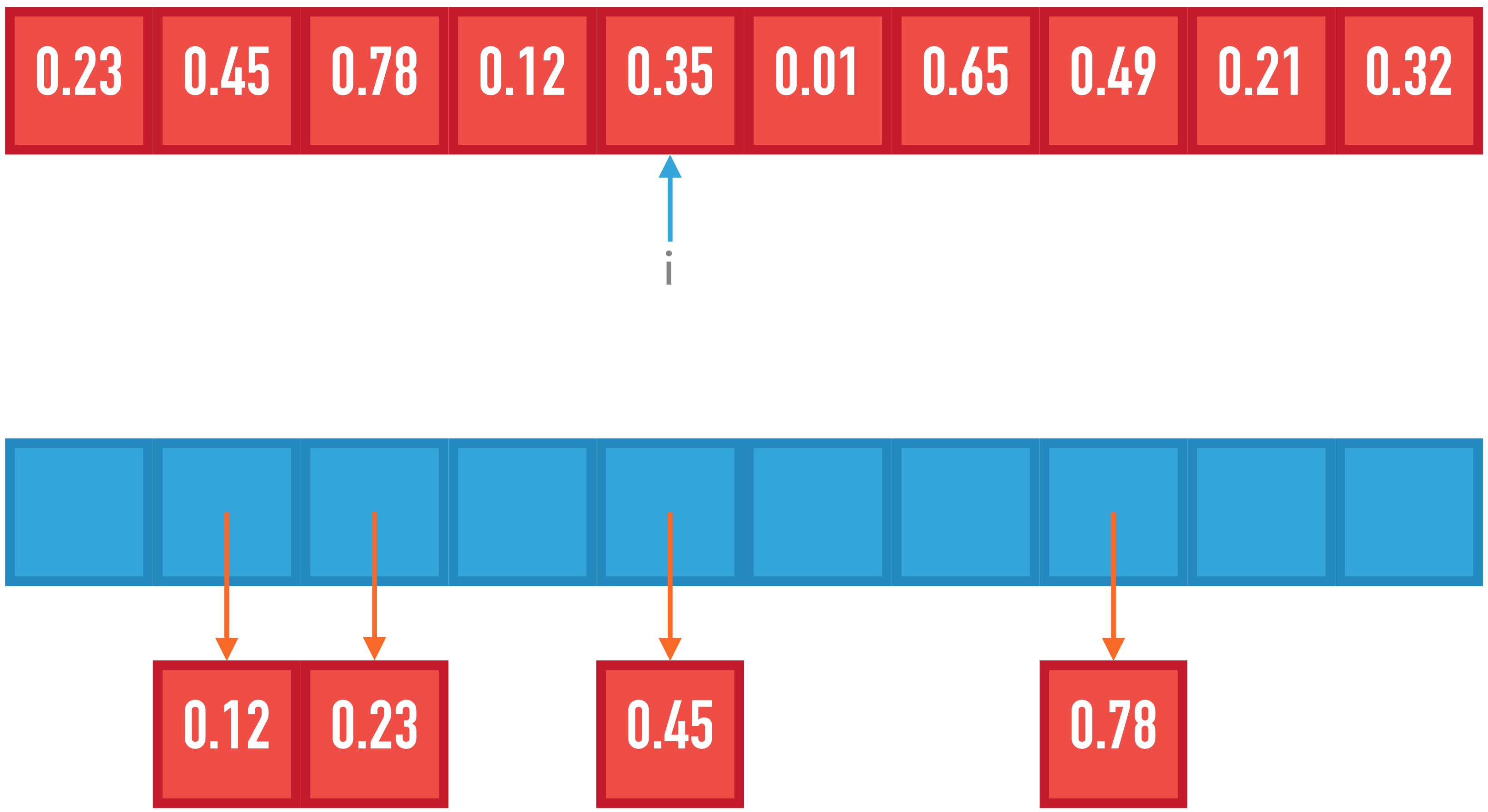
Array di n liste vuote

ORDINAMENTO



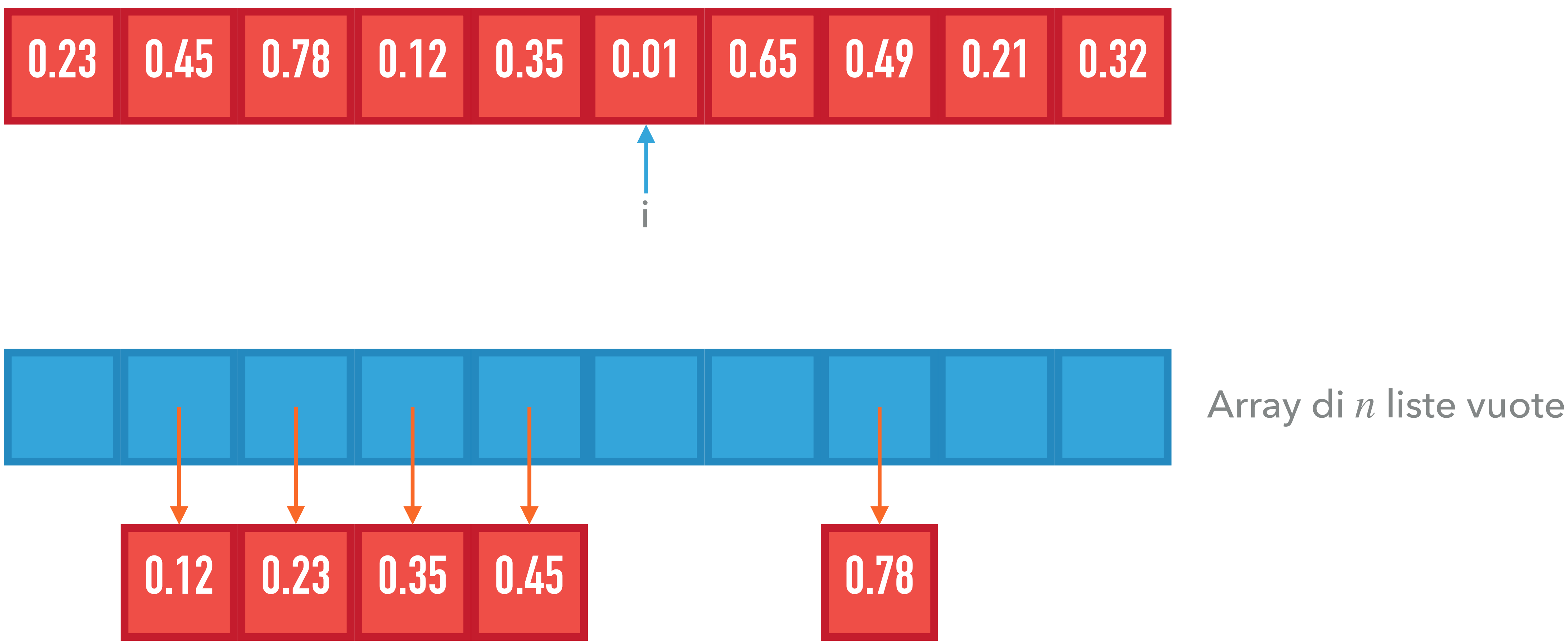
Array di n liste vuote

ORDINAMENTO

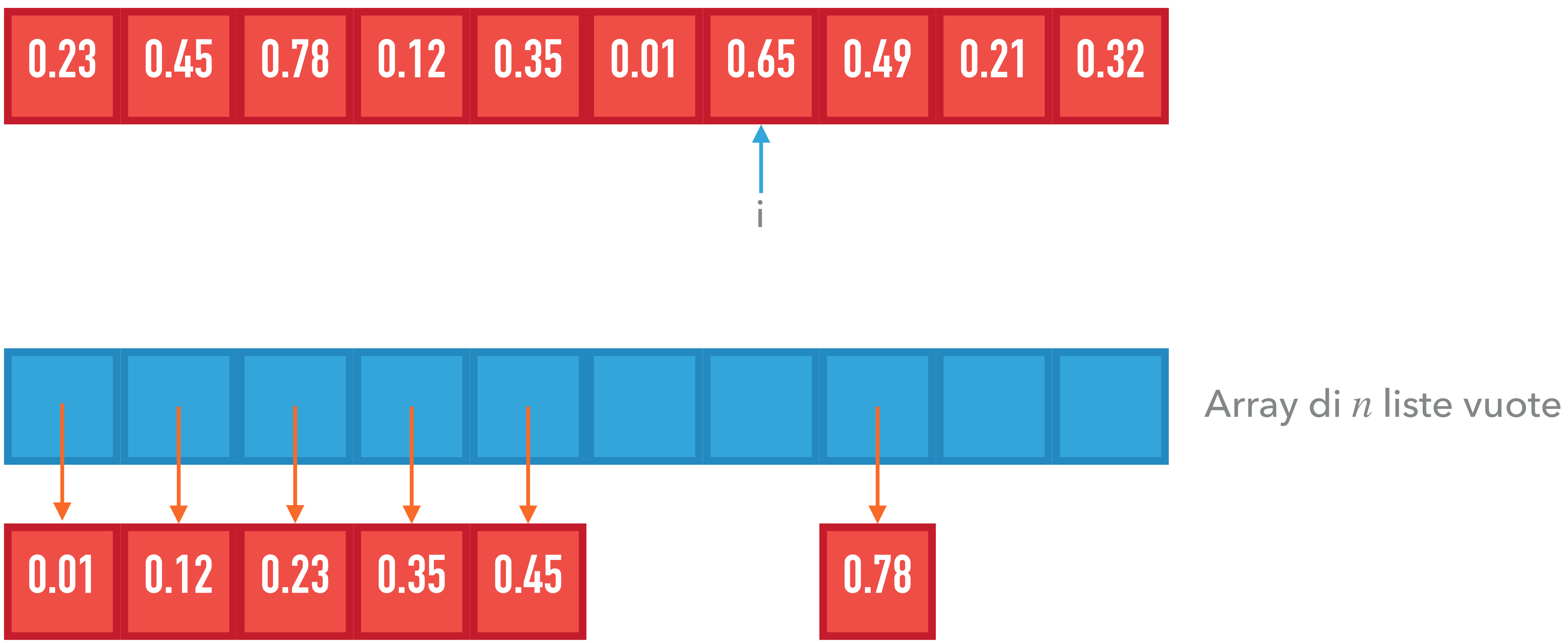


Array di n liste vuote

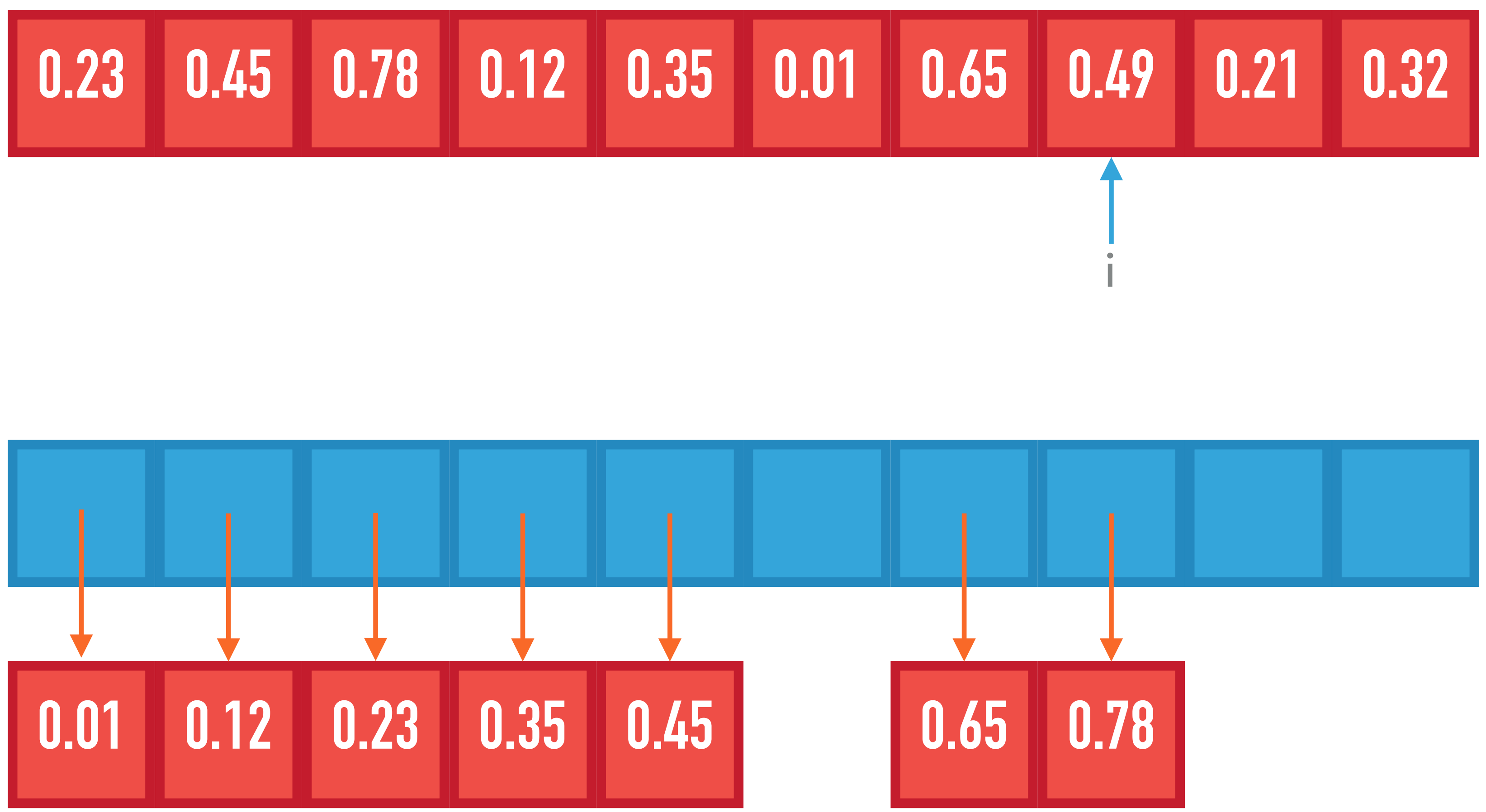
ORDINAMENTO



ORDINAMENTO

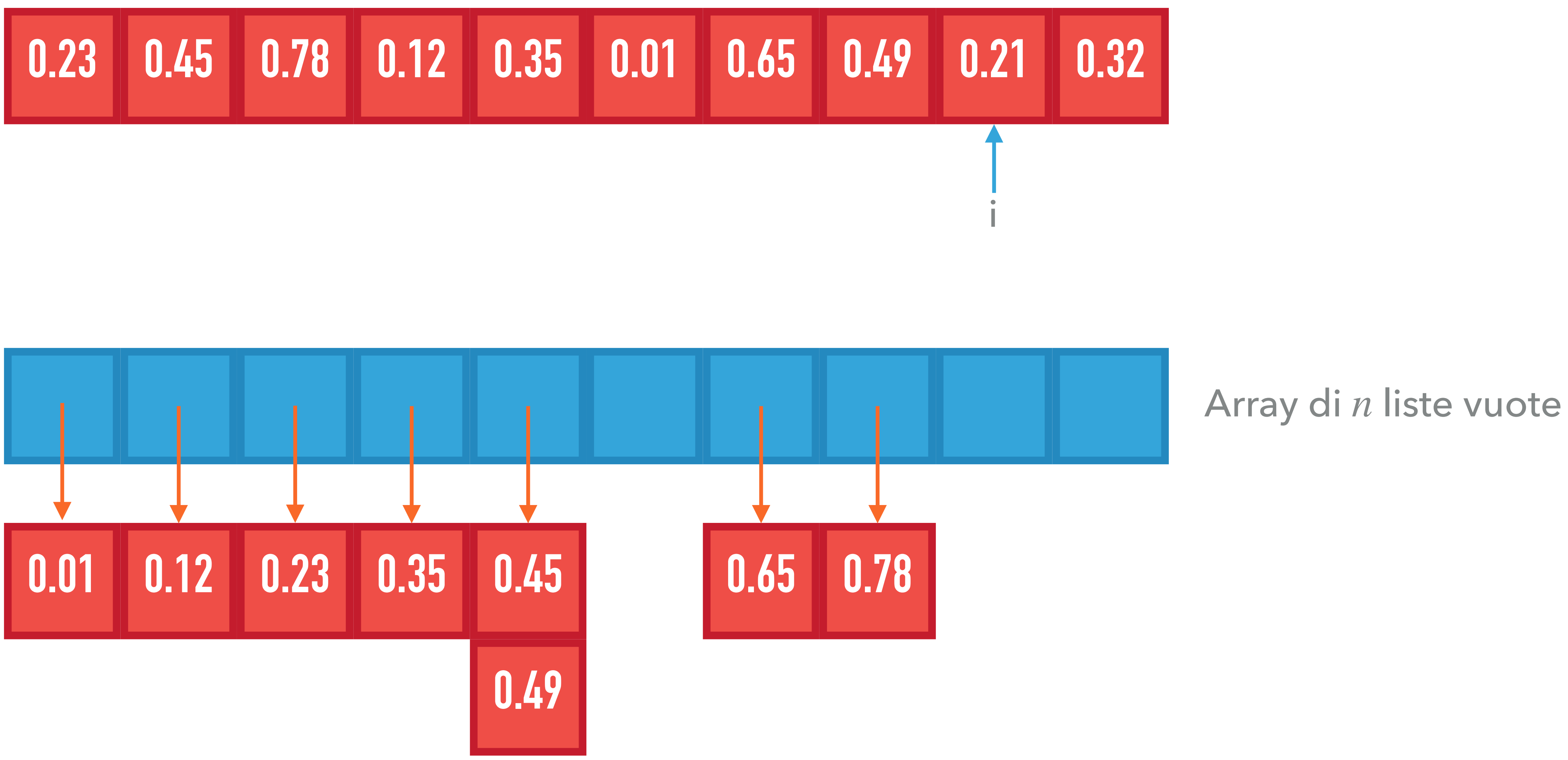


ORDINAMENTO

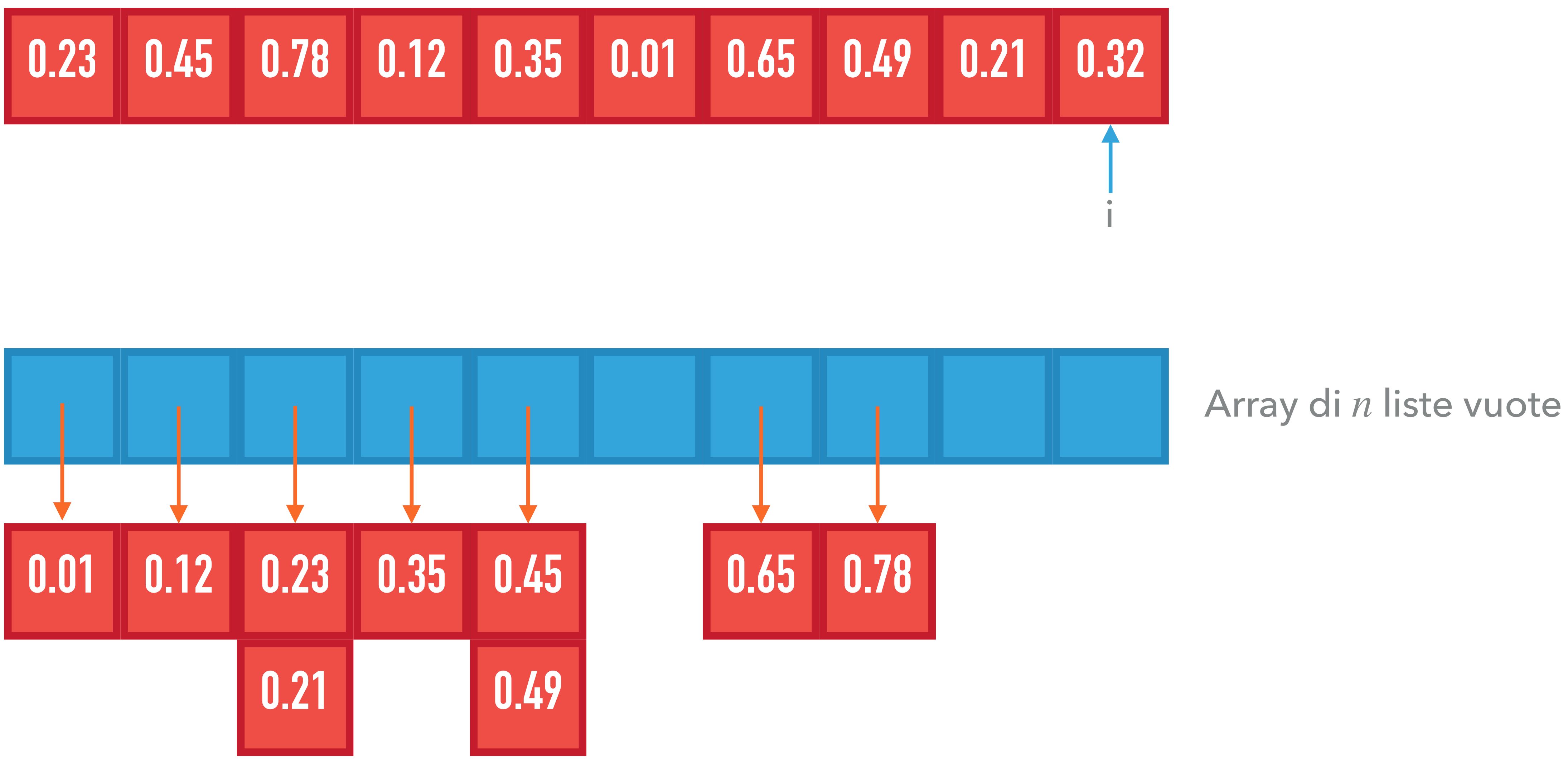


Array di n liste vuote

ORDINAMENTO

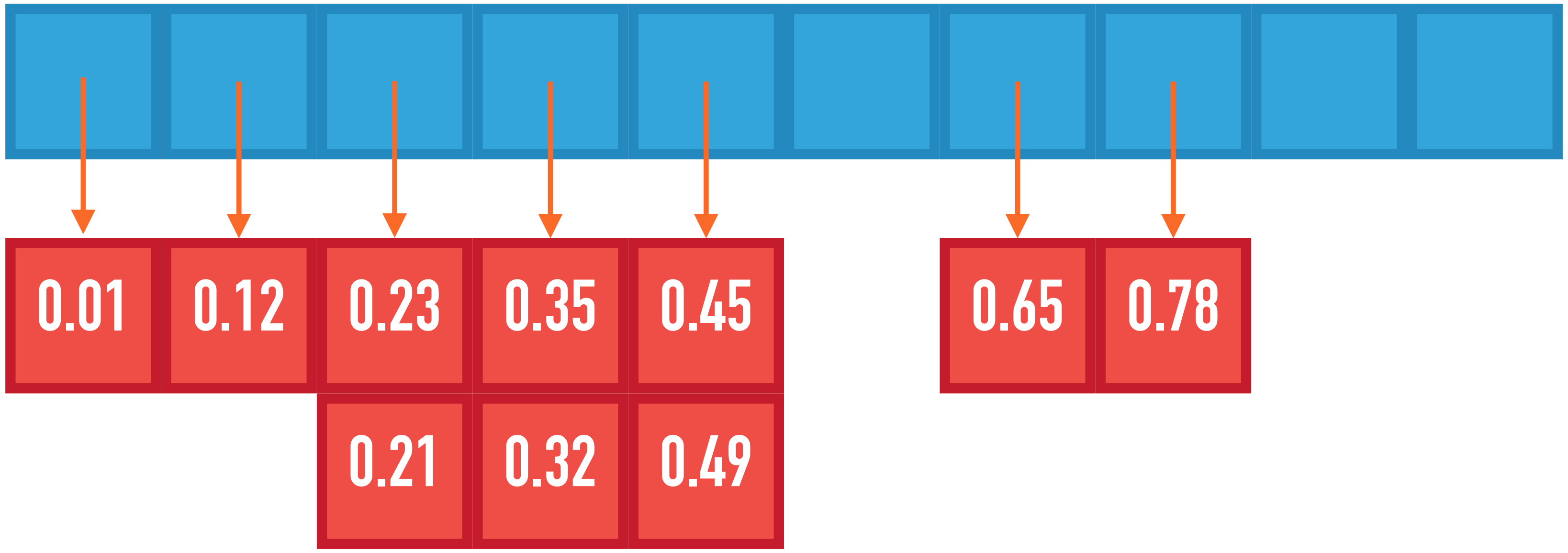


ORDINAMENTO



ORDINAMENTO

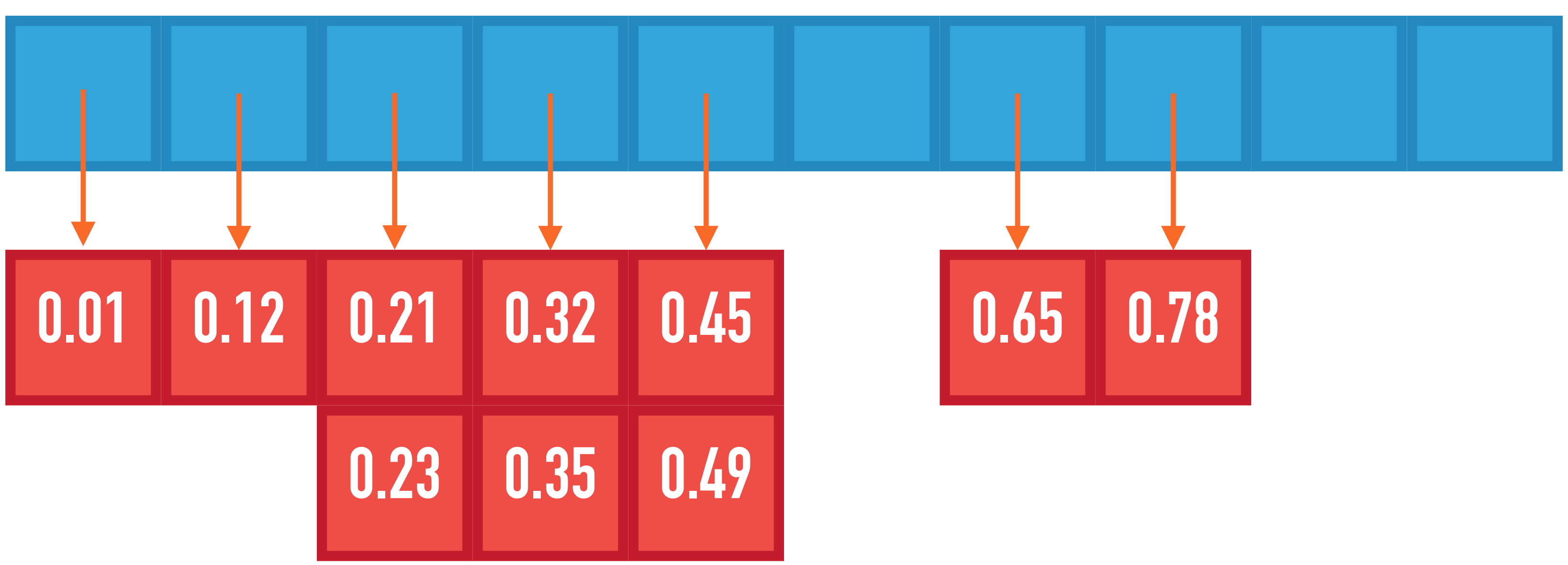
Ora dobbiamo ordinare i singoli bucket usando, per esempio, insertion sort



Array di n liste vuote

ORDINAMENTO

L'ordinamento è rapido perché nella maggior parte dei casi i bucket contengono pochi elementi



Array di n liste vuote

Ora passando i bucket da sinistra a destra e in ordine all'interno degli stessi otteniamo l'array ordinato

BUCKET SORT: PSEUDOCODICE

- `Argomenti: A (array)`
- `Alloca un array B di` n `liste vuote`
- `for i in range(0, len(A))`
  - `Aggiungi A[i] a B[`$\lfloor n \times A[i] \rfloor$`]`
- `for i in range(0, len(B))`
  - `Ordina B[i] con insertion sort`
- `Concatena B[0], B[1], … B[n-1]`

- Senza andare troppo nei dettagli del tempo di calcolo, se la distribuzione è uniforme, il contenuto della maggior parte dei bucket sarà ridotto e quindi l'ordinamento rapido.
- In particolare si trova che il tempo atteso è $\Theta(n)$