

ORDINAMENTO IN TEMPO LINEARE
COUNTING SORT
RADIX SORT

INFORMATICA

ORDINARE SENZA COMPARARE

- ▶ Abbiamo visto che ogni algoritmo di ordinamento basato 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

- ▶ Se sappiamo quali sono i valori possibili da ordinare, possiamo contare per ogni valore 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



Supponiamo tutti i valori siano in $[0,4]$



Allochiamo un array di k elementi

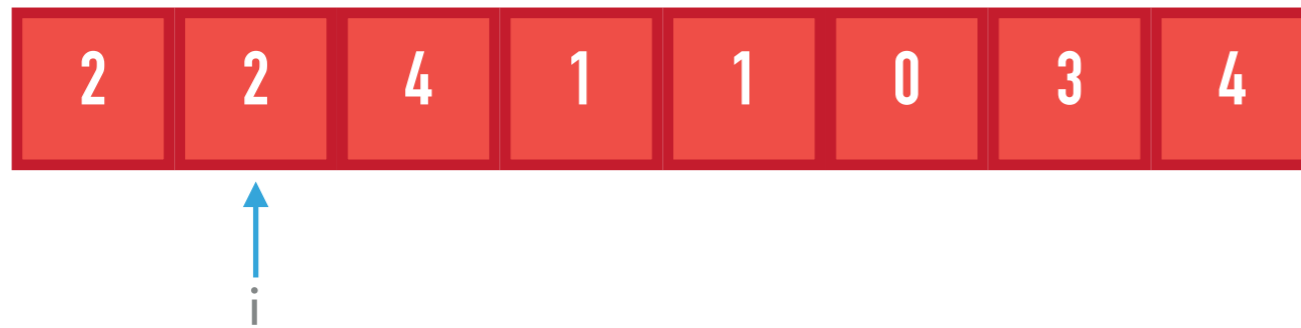
CONTEGGIO DEI VALORI



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 $[0,4]$

↑
i



CONTEGGIO DEI VALORI



Supponiamo tutti i valori siano in $[0,4]$

↑
i



CONTEGGIO DEI VALORI



Supponiamo tutti i valori siano in $[0,4]$

i



CONTEGGIO DEI VALORI



Supponiamo tutti i valori siano in $[0,4]$

↑
i



CONTEGGIO DEI VALORI



Supponiamo tutti i valori siano in $[0,4]$



CONTEGGIO DEI VALORI



Supponiamo tutti i valori siano in $[0,4]$



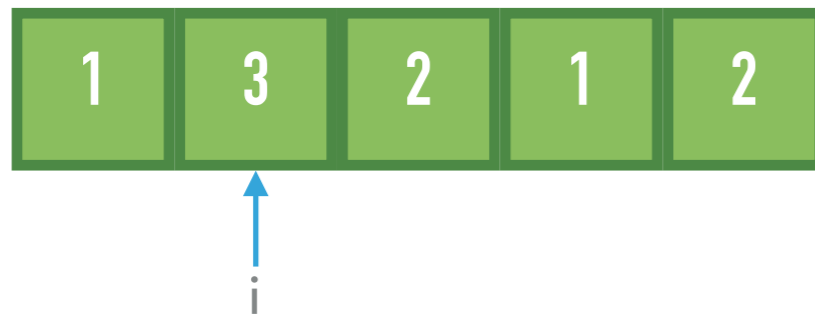
Ora abbiamo a disposizione per ogni valore in $[0,k]$ il numero di occorrenze

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



Supponiamo tutti i valori siano in $[0,4]$

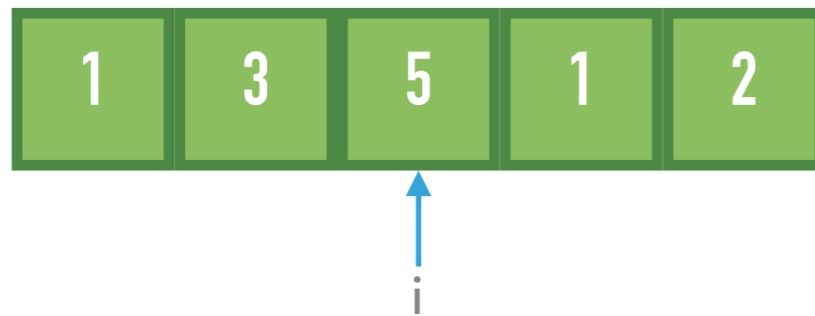


$$B[i] = B[i] + B[i-1]$$

CONTEGGIO DEI VALORI



Supponiamo tutti i valori siano in $[0,4]$



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

CONTEGGIO DEI VALORI



Supponiamo tutti i valori siano in $[0,4]$



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

CONTEGGIO DEI VALORI



Supponiamo tutti i valori siano in $[0,4]$



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

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

ORDINAMENTO



Supponiamo tutti i valori siano in $[0,4]$

Scorriamo dal fondo



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



**QUESTA OPERAZIONE
RICHIEDE UNA SPIEGAZIONE**

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

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



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



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



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

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



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



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

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



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



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

ORDINAMENTO

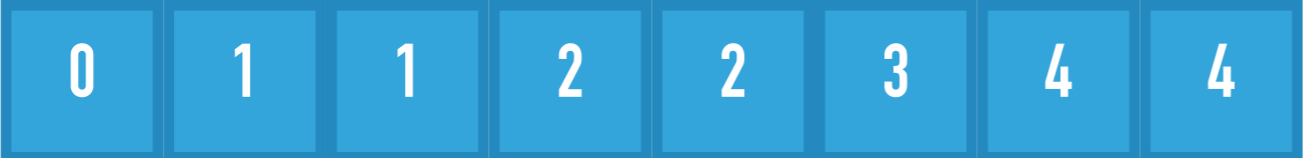


Supponiamo tutti i valori siano in $[0,4]$

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  $\leq i$ 
```

```
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

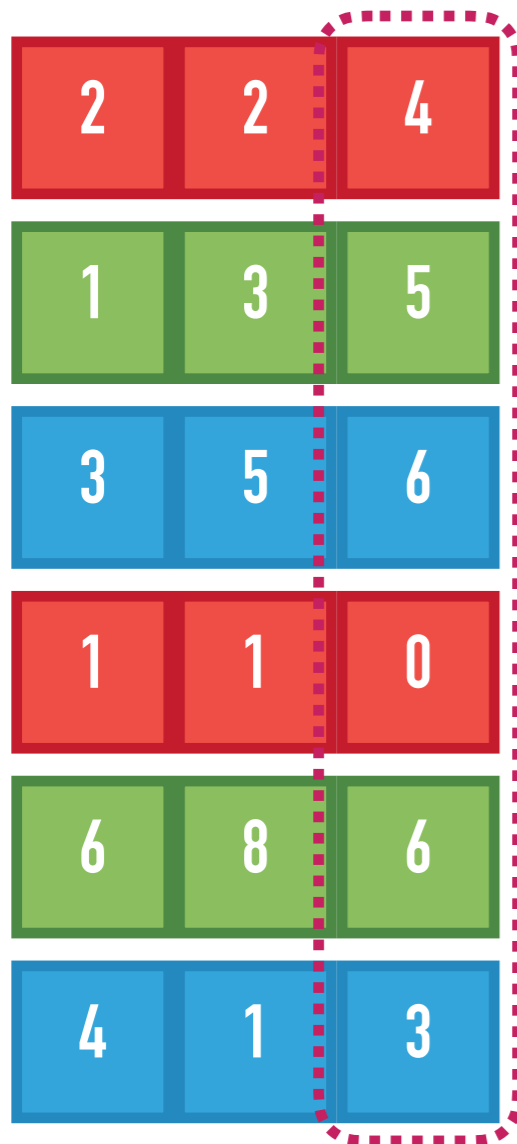
- ▶ In diversi casi la stabilità dipende da dettagli implementativi, in particolare da come sono trattati i valori identici
- ▶ In particolare a noi servirà avere il counting sort stabile
- ▶ La nostra implementazione lo è...
- ▶ ...perché inseriamo i valori a partire dal fondo
- ▶ Se avessi cambiato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

RADIX SORT

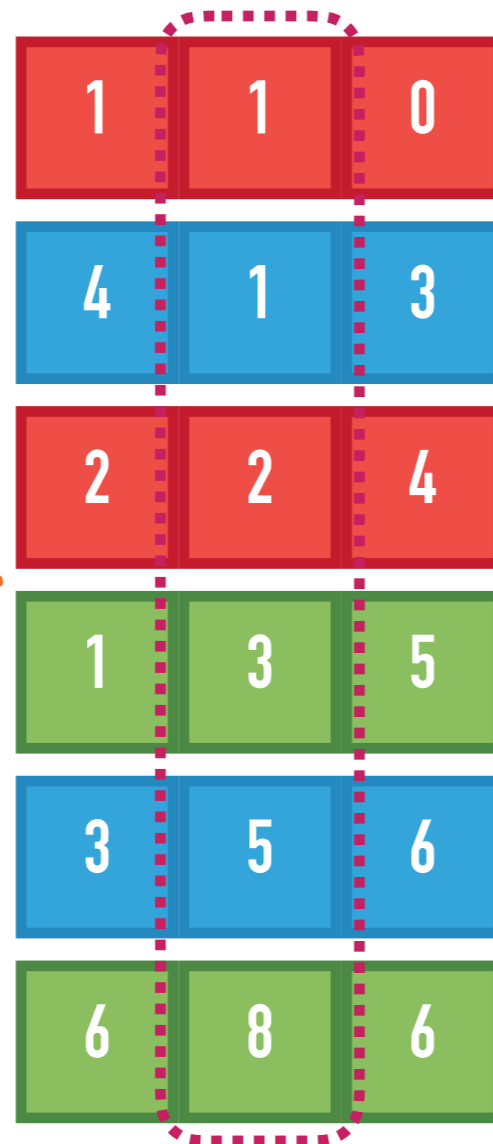
- ▶ Il radix sort richiede un ordinamento stabile
- ▶ È nato per ordinare le schede perforate, quindi le prime implementazioni di radix sort non erano con il codice, ma con strumenti meccanici!
- ▶ Idea di base: se abbiamo numeri di d cifre possiamo ordinarli una cifra alla volta usando un algoritmo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

ESEMPIO DI RADIX SORT

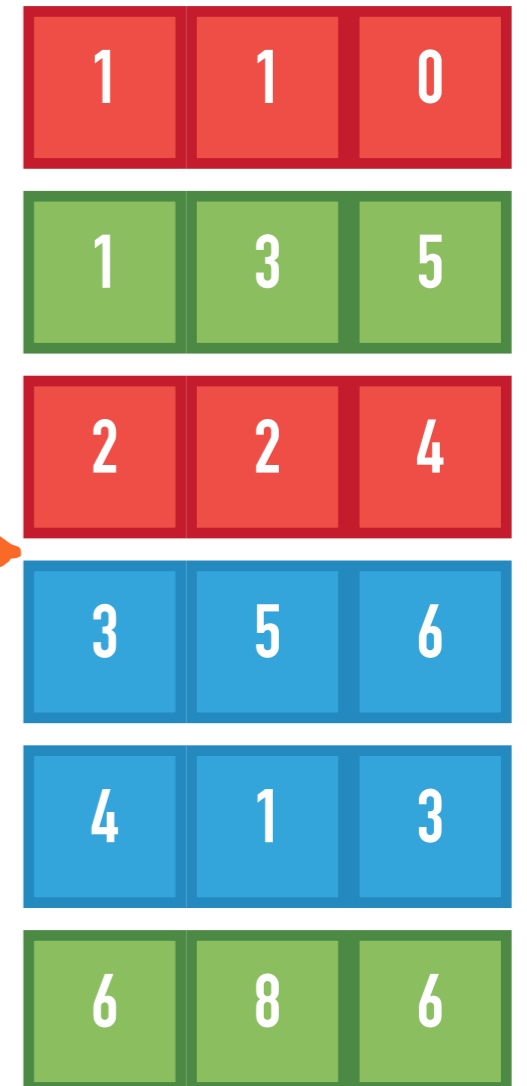
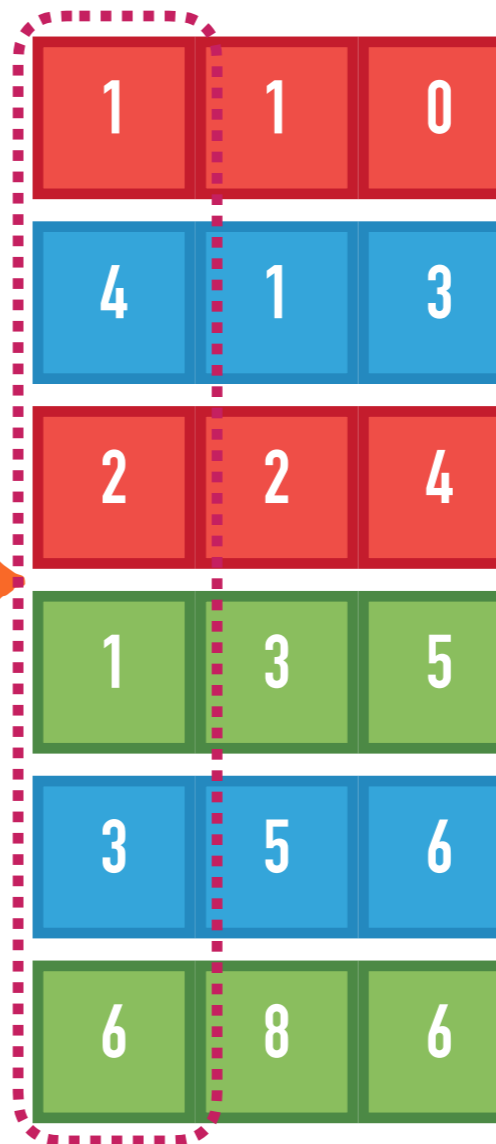
Ordiniamo rispetto all'ultima cifra



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 chela 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 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



i

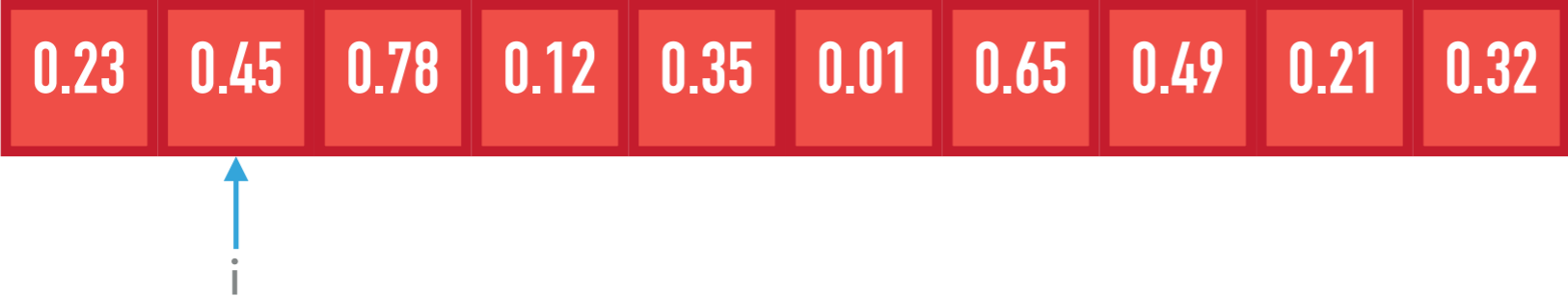
Conterrà i numeri tra 0 (incluso) e 0.1 (escluso)



Array di n liste vuote

Conterrà i numeri tra 0.7 (incluso) e 0.8 (escluso)

ORDINAMENTO



ORDINAMENTO



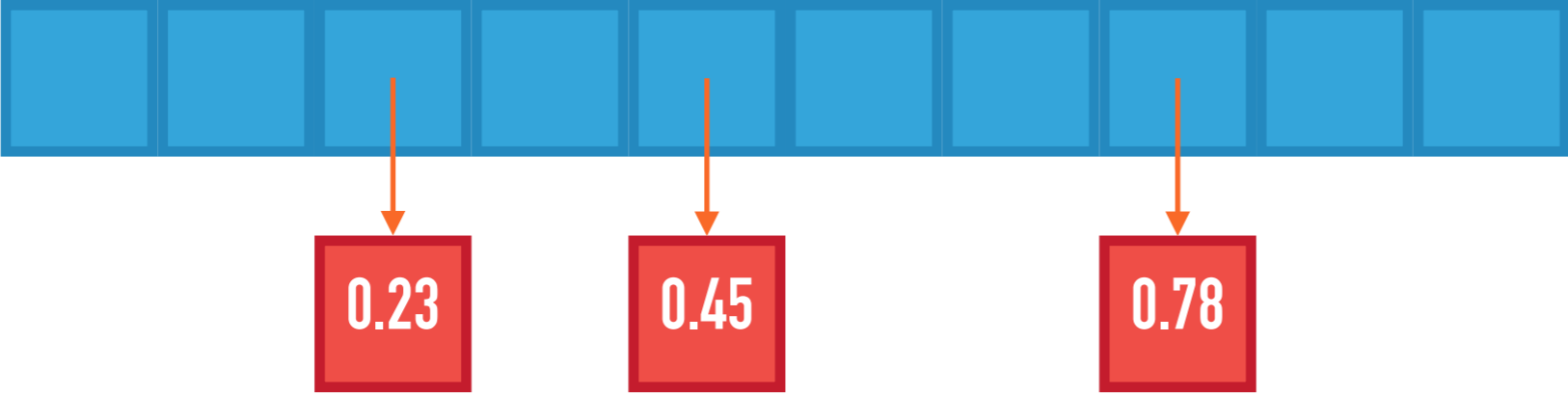
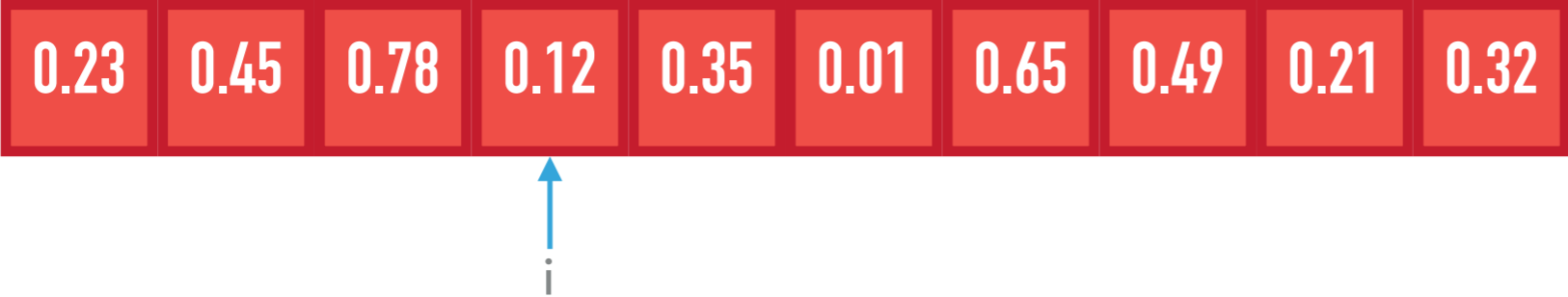
i



Array di n liste vuote

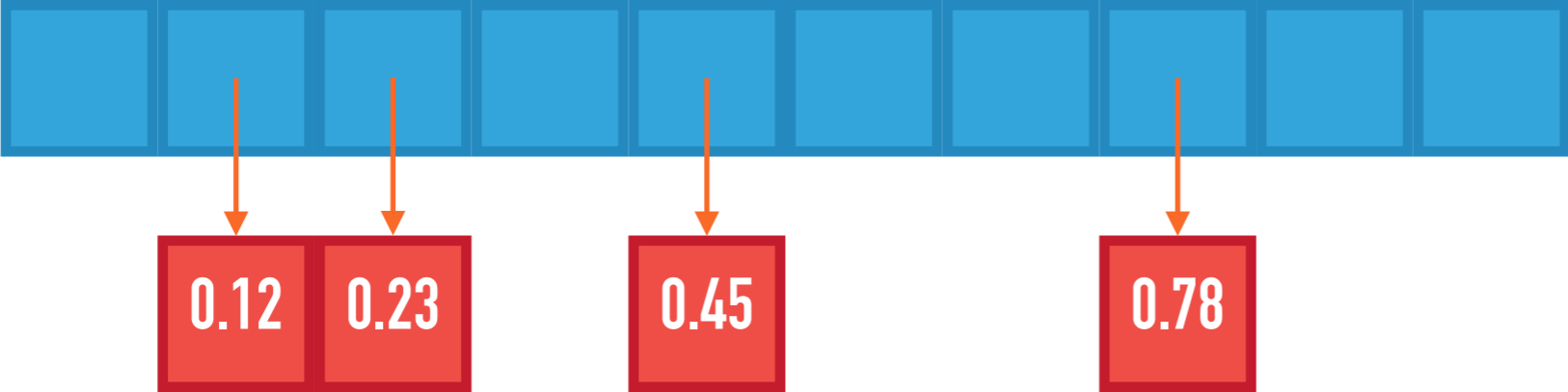


ORDINAMENTO



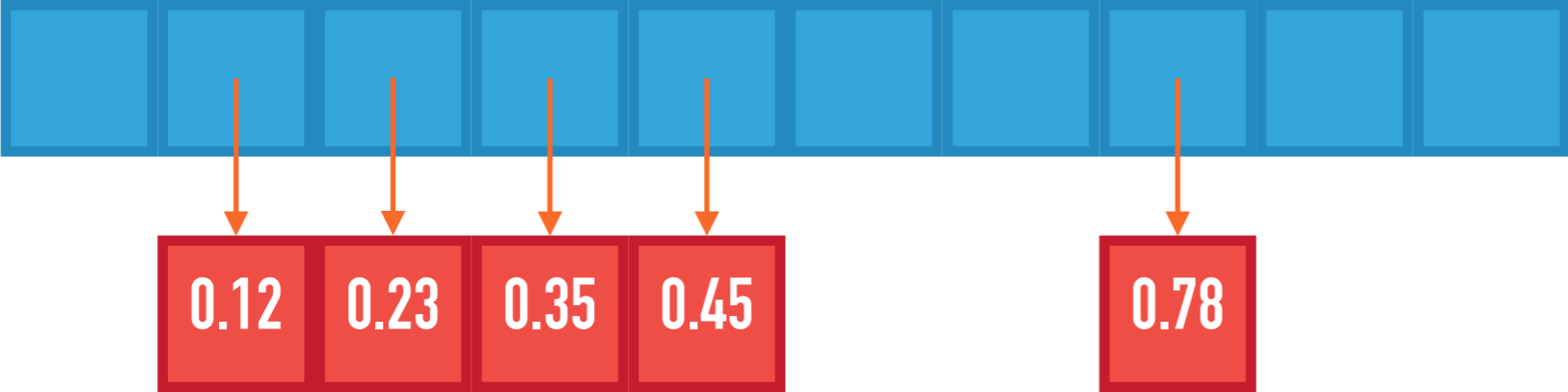
Array di n liste vuote

ORDINAMENTO

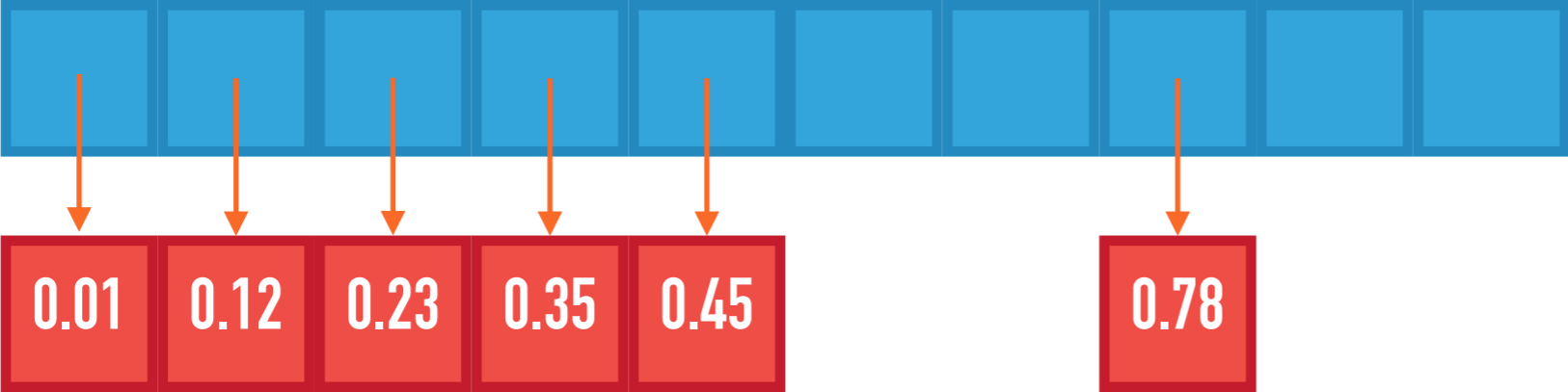
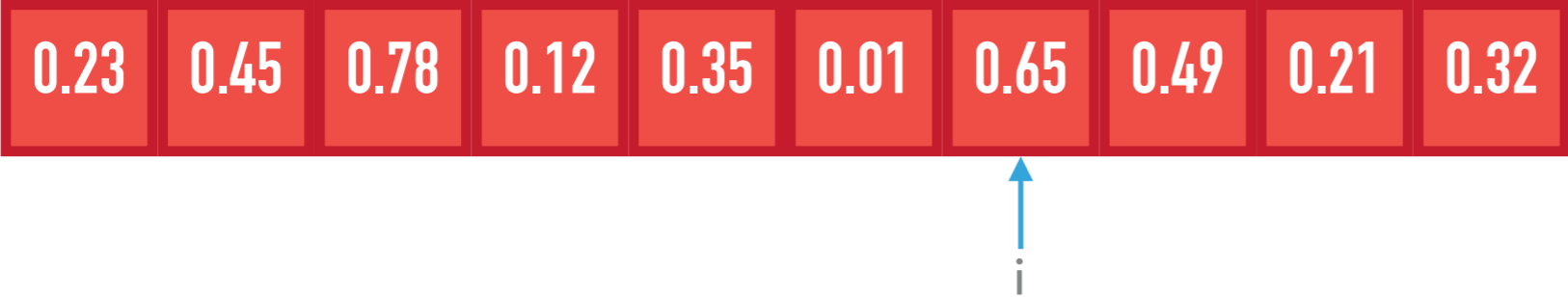


Array di n liste vuote

ORDINAMENTO

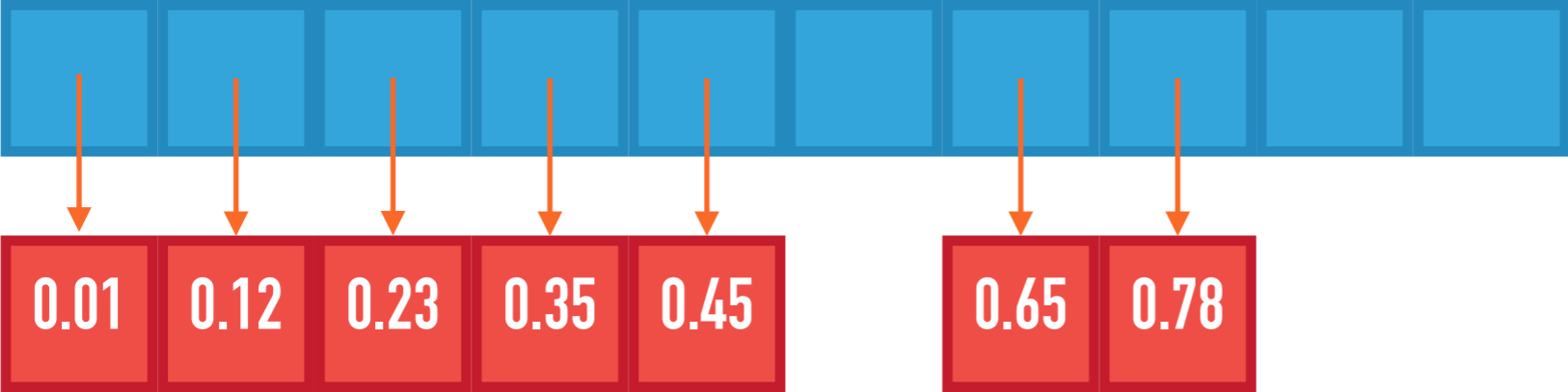
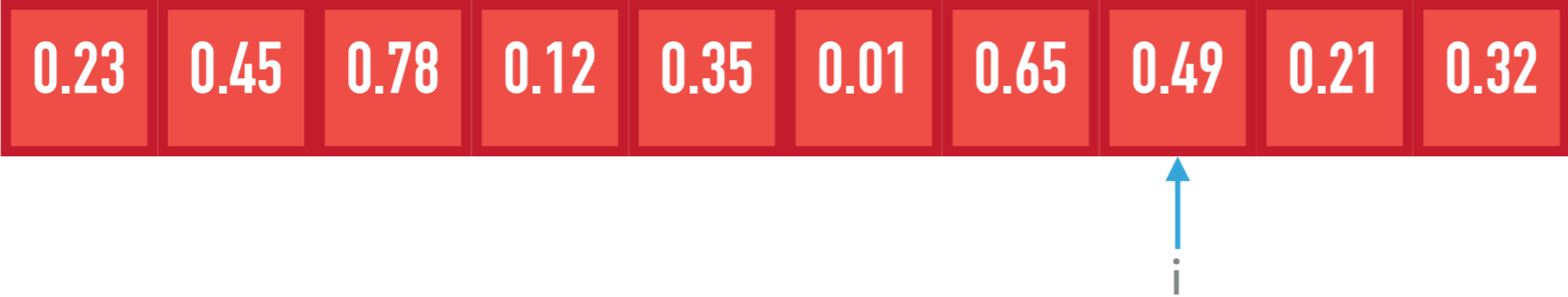


ORDINAMENTO



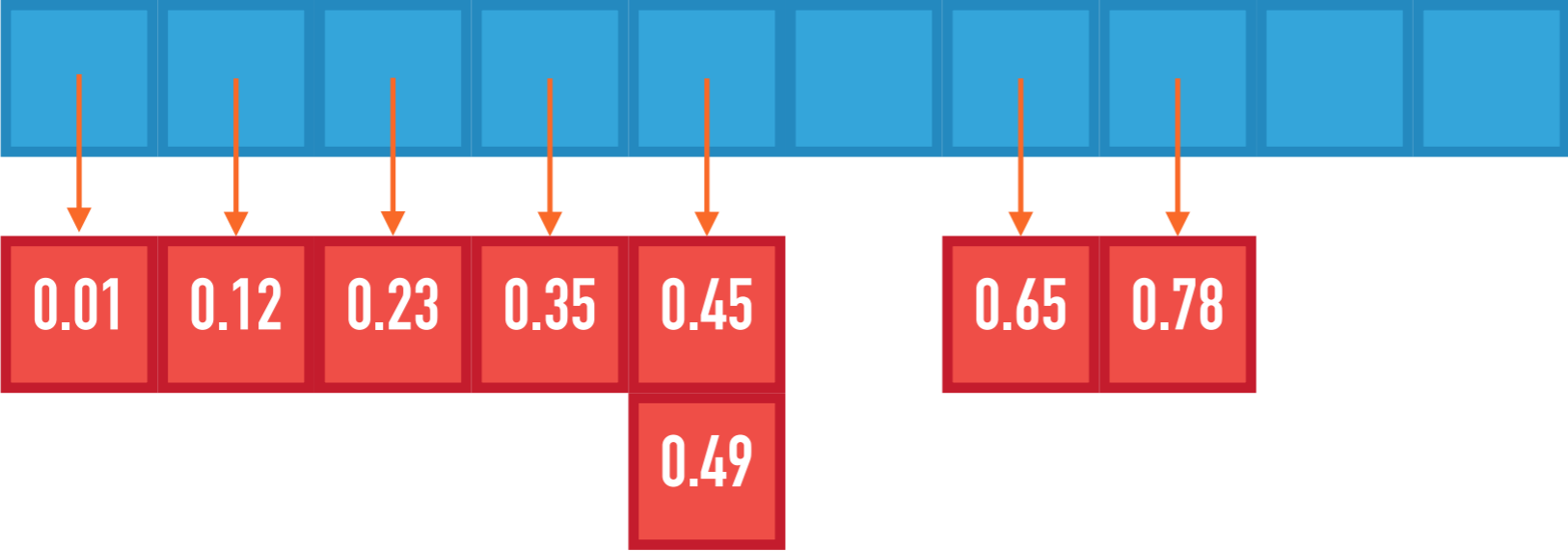
Array di n liste vuote

ORDINAMENTO

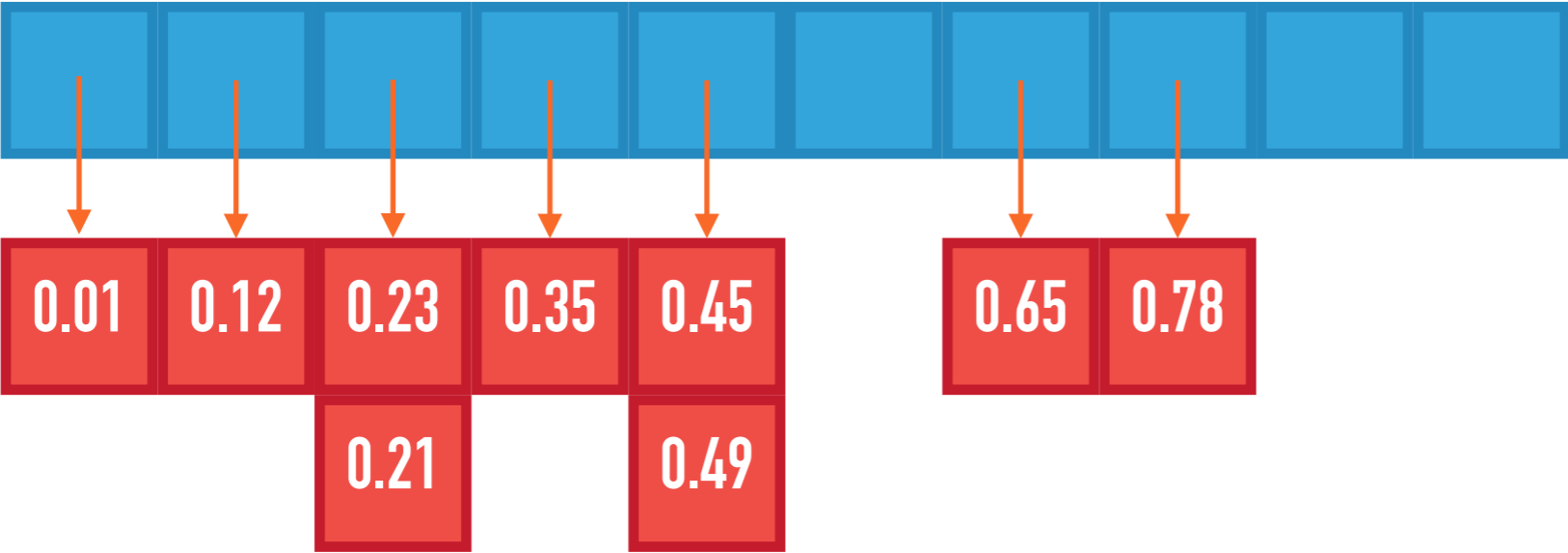
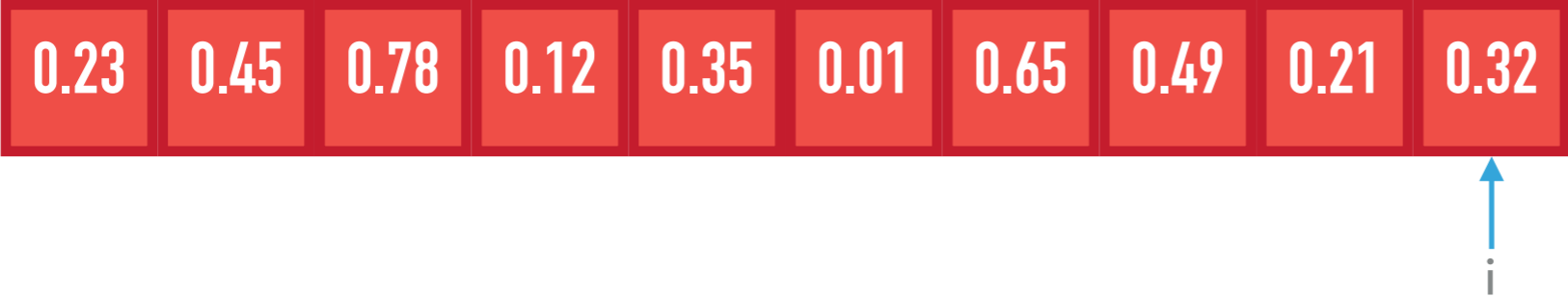


Array di n liste vuote

ORDINAMENTO



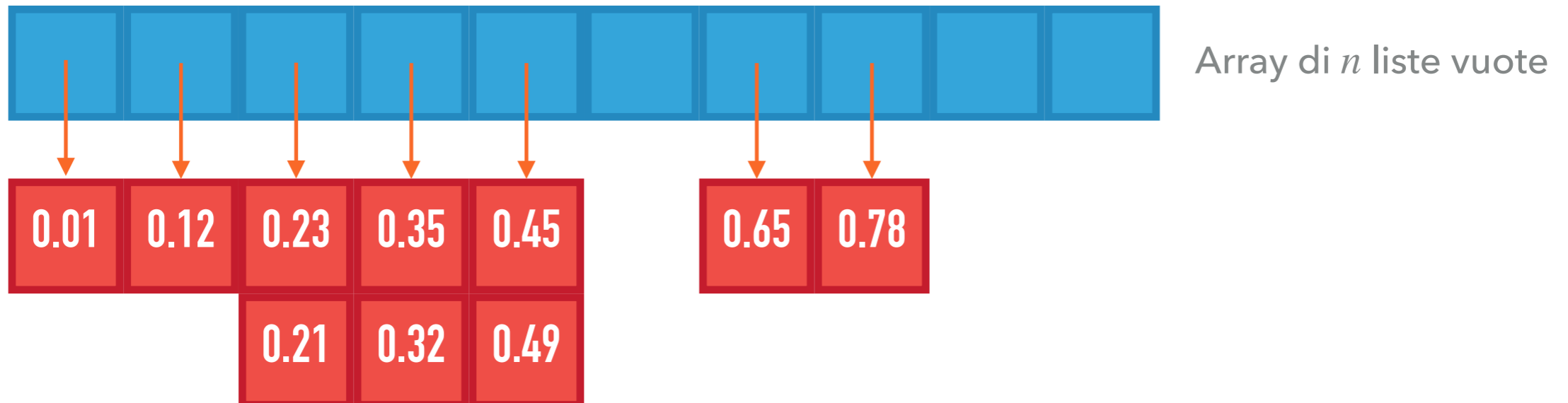
ORDINAMENTO



ORDINAMENTO



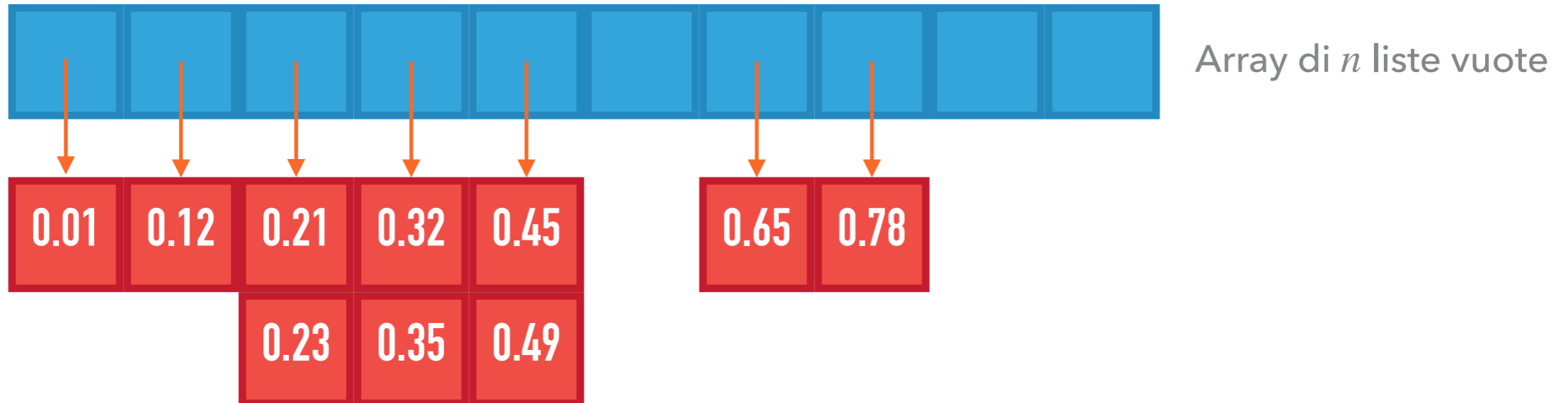
Ora dobbiamo ordinare i singoli bucket usando, per esempio, insertion sort



ORDINAMENTO



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



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 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 $\Theta(n)$