

RECUPERO DEI DATI  
ALGORITMI DI ORDINAMENTO

---

# ALGORITMI E STRUTTURE DATI

# Indicizzazione

# Recupero dei Dati

$A = \langle a_1, \dots, a_n \rangle$  contiene alcuni dati, es. dati medici di pazienti

Ciascun elemento è associato a un **identificatore**  $A[i].id$ , es. codice fiscale

Come trovare i dati associati all'identificatore  $id_1$ ?

## La Soluzione Naïve

Scandiamo tutta la base di dati cercando un  $i$  per cui  $A[i].id = id_1$

Qual'è la complessità asintotica?



## La Soluzione Naïve

Scandiamo tutta la base di dati cercando un  $i$  per cui  $A[i].id = id_1$

Qual'è la complessità asintotica?  $O(n)$

Possiamo fare meglio?

*Suggerimento:* Come cerchiamo una pagina in un libro? E una parola nel dizionario? Perché?

# Una Tecnica più Efficiente: La Ricerca Binaria

Se  $A = \langle a_1, \dots, a_n \rangle$  è **ordinato** rispetto agli id. . .

(cioè,  $i < j$  implica  $A[i].id \leq A[j].id$ )

# Una Tecnica più Efficiente: La Ricerca Binaria

Se  $A = \langle a_1, \dots, a_n \rangle$  è **ordinato** rispetto agli id. . .

(cioè,  $i < j$  implica  $A[i].id \leq A[j].id$ )

Consideriamo il valore della mediana (cioè  $A[n/2]$ )

# Una Tecnica più Efficiente: La Ricerca Binaria

Se  $A = \langle a_1, \dots, a_n \rangle$  è **ordinato** rispetto agli id...

(cioè,  $i < j$  implica  $A[i].id \leq A[j].id$ )

Consideriamo il valore della mediana (cioè  $A[n/2]$ )

se  $A[n/2].id = id_1$

Trovato!

se  $A[n/2].id > id_1$

Concentriamoci sulla I metà di  $A$ , i.e.  $\langle a_1, \dots, a_{n/2-1} \rangle$

if  $A[n/2].id < id_1$

Concentriamoci sulla II metà di  $A$ , i.e.  $\langle a_{n/2+1}, \dots, a_n \rangle$

Ripetiamo finchè  $A$  non è vuoto

$$\begin{array}{c} n \\ \frac{n}{2^i} = 1 \\ \uparrow \\ \downarrow \\ 2^i = n \\ \uparrow \\ i = \log_2 n \end{array}$$

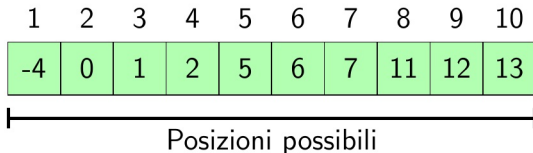
## Ricerca Dicotomica: Un Esempio

Cerchiamo 2 in  $\langle -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 \rangle$ .

1	2	3	4	5	6	7	8	9	10
-4	0	1	2	5	6	7	11	12	13

## Ricerca Dicotomica: Un Esempio

Cerchiamo 2 in  $\langle -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 \rangle$ .



## Ricerca Dicotomica: Un Esempio

Cerchiamo 2 in  $\langle -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 \rangle$ .



## Ricerca Dicotomica: Un Esempio

Cerchiamo 2 in  $\langle -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 \rangle$ .

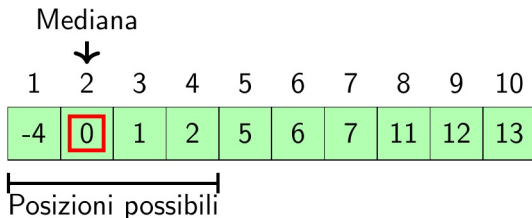
1	2	3	4	5	6	7	8	9	10
-4	0	1	2	5	6	7	11	12	13

┌───────────┐  
Posizioni possibili



## Ricerca Dicotomica: Un Esempio

Cerchiamo 2 in  $\langle -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 \rangle$ .

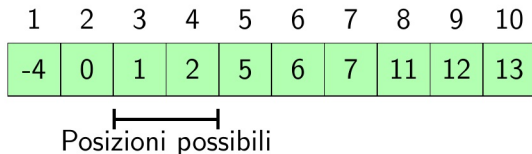


## Ricerca Dicotomica: Un Esempio

Cerchiamo 2 in  $\langle -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 \rangle$ .

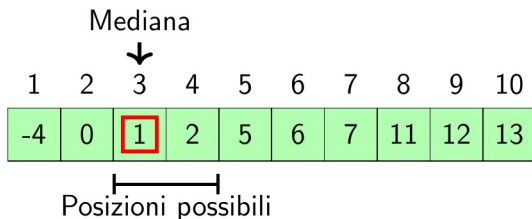
1	2	3	4	5	6	7	8	9	10
-4	0	1	2	5	6	7	11	12	13

Posizioni possibili

A diagram illustrating a binary search step. It shows a sorted array of 10 elements. The indices are labeled 1 through 10 above the array. The values are -4, 0, 1, 2, 5, 6, 7, 11, 12, 13. A horizontal bracket is drawn below the array, spanning from index 3 to index 4. Below the bracket, the text "Posizioni possibili" is written, indicating that the search range is now narrowed to these two positions.

## Ricerca Dicotomica: Un Esempio

Cerchiamo 2 in  $\langle -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 \rangle$ .



## Ricerca Dicotomica: Un Esempio

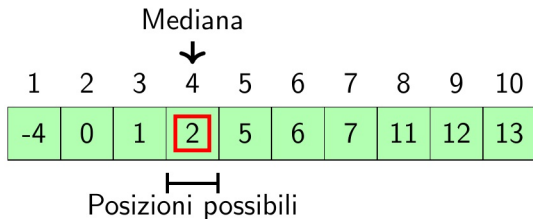
Cerchiamo 2 in  $\langle -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 \rangle$ .

1	2	3	4	5	6	7	8	9	10
-4	0	1	2	5	6	7	11	12	13

┌───┐  
Posizioni possibili

## Ricerca Dicotomica: Un Esempio

Cerchiamo 2 in  $\langle -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 \rangle$ .



**Trovato:**  $A[4] = 2$

# Ricerca Dicotomica: Pseudo-Codice e Complessità

```
def di_find(A, a):  
    (l, r) ← (1, |A|)  
    while r ≥ l:  
        m ← ⌊(l+r)/2⌋  
        if A[m] = a:  
            return m  
        elif A[m] > a:  
            r ← m-1  
        else:  
            l ← m+1  
        endif  
    endwhile  
    return 0  
endef
```

$O(1)$

$$\frac{O(\log_2 n)}{n = |A|}$$

$$O(1)O(\log_2 n) = O(\log_2 n)$$

$l - r$  si dimezza a ogni iterazione

Se  $|A| = 2^m$  di\_find termina in al più  $m$  iterazioni

Il blocco del while costa  $\Theta(1)$

La complessità di di\_find è

$$O(\log_2 n)$$

$$m = \log_2 n$$

A,  $|A| = m$

m ricerche/query

1) RICERCA LINEARE  $O(m \cdot n)$

2) RICERCA BINARIA + ORDINARE  $O(\underbrace{m \cdot \log n}_{\text{ordinare}} + m \cdot \log n)$

## Ricerca Dicotomica vs Ricerca Lineare: un Test

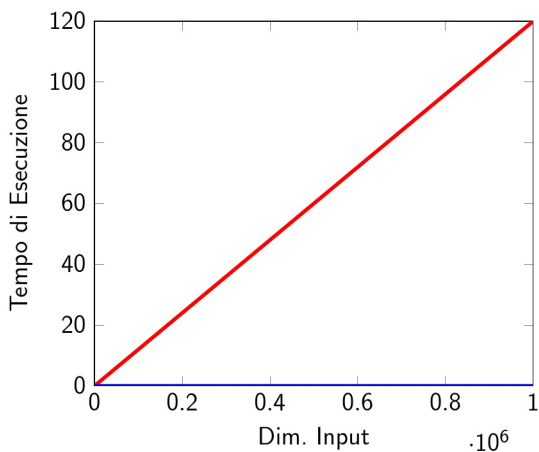
Tempo di esecuzione per  $1 \times 10^5$  ricerche casuali.

Dim. Input	Ricerca Lineare	Ricerca Dicotomica
$1 \times 10^1$	$3.3 \times 10^{-3}$ s	$3.2 \times 10^{-3}$ s
$1 \times 10^2$	$1.4 \times 10^{-2}$ s	$4.3 \times 10^{-3}$ s
$1 \times 10^3$	$1.2 \times 10^{-1}$ s	$5.9 \times 10^{-3}$ s
$1 \times 10^4$	1.2 s	$7.8 \times 10^{-3}$ s
$1 \times 10^5$	$1.2 \times 10^1$ s	$8.7 \times 10^{-3}$ s
$1 \times 10^6$	$1.2 \times 10^2$ s	$1.2 \times 10^{-2}$ s



# Ricerca Dicotomica vs Ricerca Lineare: un Test

Tempo di esecuzione per  $1 \times 10^5$  ricerche casuali.



# Ordinamento

## II Problema dell'Ordinamento

**Input:** Un array  $A$  di numeri

**Output:** L'array  $A$  **ordinato**, cioè, se  $i < j$  allora  $A[i] \leq A[j]$

Es.,

1	2	3	4	5	6	7	8	9	10
13	5	7	2	-4	4	1	11	6	0



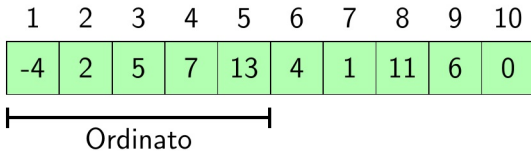
1	2	3	4	5	6	7	8	9	10
-4	0	1	2	4	5	6	7	11	13

Qualche suggerimento su un possibile algoritmo? Qual'è la complessità attesa?

# Insertion Sort

# Insertion Sort: Intuizione

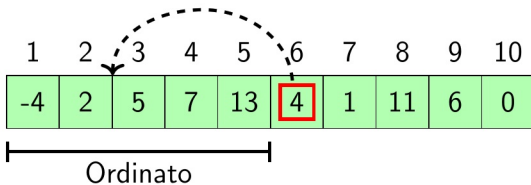
Se la prima parte dell'array è già ordinata



# Insertion Sort: Intuizione

Se la prima parte dell'array è già ordinata

possiamo "*allargarla*" inserendo il valore successivo **v** nella posizione corretta

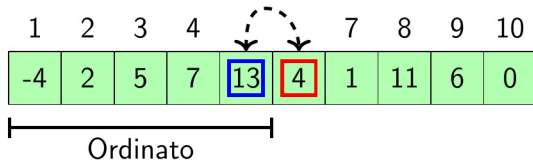


# Insertion Sort: Intuizione

Se la prima parte dell'array è già ordinata

possiamo “*allargarla*” inserendo il valore successivo **v** nella posizione corretta

scambiando **v** con il valore precedente nell'array **p**

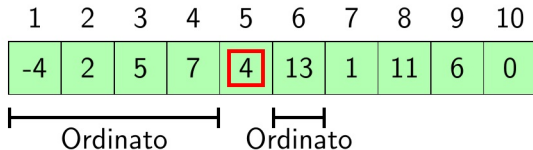


# Insertion Sort: Intuizione

Se la prima parte dell'array è già ordinata

possiamo “*allargarla*” inserendo il valore successivo **v** nella posizione corretta

scambiando **v** con il valore precedente nell'array **p**



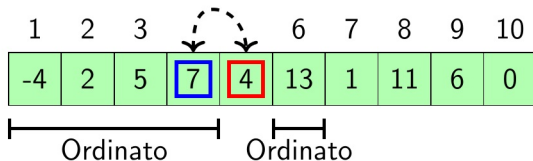


# Insertion Sort: Intuizione

Se la prima parte dell'array è già ordinata

possiamo “*allargarla*” inserendo il valore successivo **v** nella posizione corretta

scambiando **v** con il valore precedente nell'array **p**

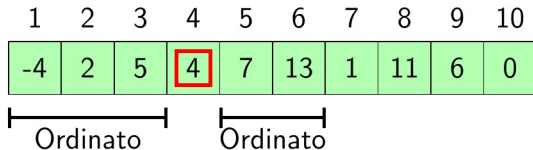


# Insertion Sort: Intuizione

Se la prima parte dell'array è già ordinata

possiamo “*allargarla*” inserendo il valore successivo **v** nella posizione corretta

scambiando **v** con il valore precedente nell'array **p**

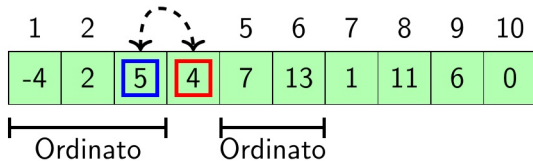


# Insertion Sort: Intuizione

Se la prima parte dell'array è già ordinata

possiamo “*allargarla*” inserendo il valore successivo **v** nella posizione corretta

scambiando **v** con il valore precedente nell'array **p**



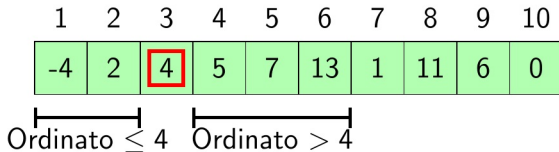
# Insertion Sort: Intuizione

Se la prima parte dell'array è già ordinata

possiamo “*allargarla*” inserendo il valore successivo **v** nella posizione corretta

scambiando **v** con il valore precedente nell'array **p**

finchè **p** (se esiste) è più grande di **v**



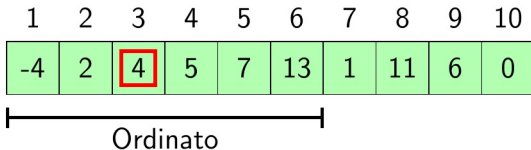
# Insertion Sort: Intuizione

Se la prima parte dell'array è già ordinata

possiamo “*allargarla*” inserendo il valore successivo **v** nella posizione corretta

scambiando **v** con il valore precedente nell'array **p**

finchè **p** (se esiste) è più grande di **v**



## Insertion Sort: Codice e Complessità

```
def insertion_sort(A):  
    for i in 2..|A|:  
        j ← i  
        while (j > 1 and  
              A[j] < A[j - 1]):  
            swap(A, j - 1, j)  
            j ← j - 1  
        endwhile  
    endfor  
enddef
```

#  $O(i)$

$\Omega(1)$

$\left\{ \begin{array}{l} O(i) \\ \Omega(1) \end{array} \right\}$

Il blocco del while costa  $\Theta(1)$

Viene iterato  $O(i)$  ( $\Omega(1)$ ) volte  
per ogni  $i \in [2, n]$

$$\sum_{i=2}^n O(i) * O(1) = O\left(\sum_{i=2}^n i\right) = O(n^2)$$

$$\sum_{i=2}^n \Omega(1) * \Omega(1) = \Omega\left(\sum_{i=2}^n 1\right) = \Omega(n)$$

## **Bolle, Selezioni e Varie**

# Ordinare Scegliendo il Massimo

Trova il massimo

1	2	3	4	5	6	7	8	9	10
13	5	7	2	-4	4	1	11	6	0



# Ordinare Scegliendo il Massimo

Trova il massimo

**Sposta il massimo alla fine dell'array**

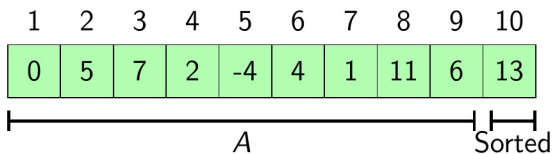
1	2	3	4	5	6	7	8	9	10
0	5	7	2	-4	4	1	11	6	13

# Ordinare Scegliendo il Massimo

Trova il massimo

Sposta il massimo alla fine dell'array

Se  $|A| > 1$ , **ripeti sul frammento iniziale di  $A$**

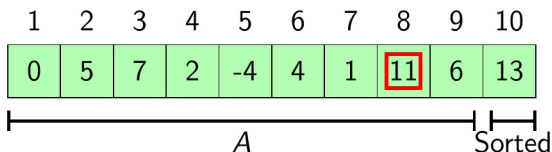


# Ordinare Scegliendo il Massimo

**Trova il massimo**

Sposta il massimo alla fine dell'array

Se  $|A| > 1$ , ripeti sul frammento iniziale di  $A$

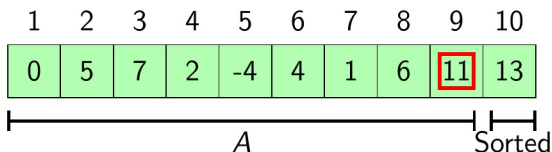


# Ordinare Scegliendo il Massimo

Trova il massimo

**Sposta il massimo alla fine dell'array**

Se  $|A| > 1$ , ripeti sul frammento iniziale di  $A$



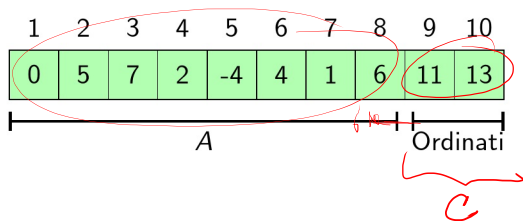
# Ordinare Scegliendo il Massimo

Trova il massimo

Sposta il massimo alla fine dell'array

Se  $|A| > 1$ , ripeti sul frammento iniziale di  $A$

INVARIANTE  
→  $C$  è ordinato  
e  $\forall x \in C, \forall y \in A$   
 $x \geq y$

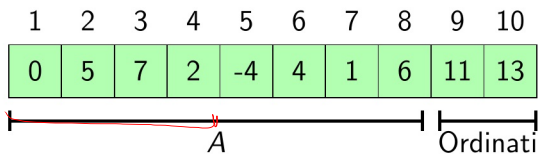


# Ordinare Scegliendo il Massimo

Trova il massimo

Sposta il massimo alla fine dell'array

Se  $|A| > 1$ , ripeti sul frammento iniziale di  $A$



La complessità è  $\sum_{i=1}^{|A|} (T_{\max}(i) + \Theta(1))$

*TROVO MAX SU A di i ELEMENTI*

# Come Trovare il Massimo?

Usando ...

- ▶ spostandolo verso destra con scambi  $\Rightarrow$  Bubble Sort

$$T(|A|) \in \sum_{i=1}^{|A|} (\Theta(i) + O(i)) = \Theta(|A|^2)$$

- ▶ ricerca lineare sulla porzione non ordinata  $\Rightarrow$  Selection Sort

$$T(|A|) \in \sum_{i=1}^{|A|} (\Theta(i) + \Theta(1)) = \Theta(|A|^2)$$

# Bubble Sort: Pseudo-Codice e Complessità

```
def bubble_sort(A):  
    for i in |A|..2:  
        for j in 1..i-1:  
            if (1) if A[j] > A[j+1]:  
                swap(A, j, j+1)  
            endif  
        endfor  
    endfor  
enddef
```

il costrutto if costa  $\Theta(1)$

Viene iterato  $i - 1$  volte per ogni  $i \in [2, n]$

$$\sum_{i=2}^n \sum_{j=1}^{i-1} \Theta(1) = \Theta(n^2)$$
$$= \sum_{i=2}^n \Theta(i) = \Theta(n^2)$$



# Selection Sort: Pseudo-Codice e Complessità

```
def selection_sort(A):  
    for i in |A|..2:  
        max_i ← i  
        for j in 1..i-1:  
            if A[j] > A[max_i]:  
                max_i ← j  
            endif  
        endfor  
        swap(A, max_i, i)  
    endfor  
enddef
```

il costrutto **if**, l'assegnamento e lo scambio costano  $\Theta(1)$

il costrutto **if** viene iterato  $i - 1$  volte per ogni  $i \in [2, n]$

$$\sum_{i=2}^n \left( \Theta(1) + \sum_{j=1}^{i-1} \Theta(1) \right) = \Theta(n^2)$$

# Merge Sort

## Divide-et-Impera

Il paradigma divide-et-impera è uno strumento tipico di progettazione degli algoritmi.

L'idea di fondo è di risolvere ricorsivamente istanze più semplici del problema e combinare queste soluzioni parziali in una soluzione del problema originale.

Se le istanze sono molto semplici, la soluzione è di solito banale.

1	2	3	4	5	6	7	8	9	10
13	5	7	2	-4	4	1	11	6	0

## Divide-et-Impera

Il paradigma divide-et-impera è uno strumento tipico di progettazione degli algoritmi.


L'idea di fondo è di risolvere ricorsivamente istanze più semplici del problema e combinare queste soluzioni parziali in una soluzione del problema originale.

Se le istanze sono molto semplici, la soluzione è di solito banale.

Un'idea per ordinare un array:

1. ordino la prima metà

1	2	3	4	5	6	7	8	9	10
13	5	7	2	-4	4	1	11	6	0



## Divide-et-Impera

Il paradigma divide-et-impera è uno strumento tipico di progettazione degli algoritmi.


L'idea di fondo è di risolvere ricorsivamente istanze più semplici del problema e combinare queste soluzioni parziali in una soluzione del problema originale.

Se le istanze sono molto semplici, la soluzione è di solito banale.

Un'idea per ordinare un array:

1. ordino la prima metà

1	2	3	4	5	6	7	8	9	10
-4	2	5	7	13	4	1	11	6	0



## Divide-et-Impera

Il paradigma divide-et-impera è uno strumento tipico di progettazione degli algoritmi.


L'idea di fondo è di risolvere ricorsivamente istanze più semplici del problema e combinare queste soluzioni parziali in una soluzione del problema originale.

Se le istanze sono molto semplici, la soluzione è di solito banale.

Un'idea per ordinare un array:

1. ordino la prima metà
2. ordino la seconda metà

1	2	3	4	5	6	7	8	9	10
-4	2	5	7	13	4	1	11	6	0



## Divide-et-Impera

Il paradigma divide-et-impera è uno strumento tipico di progettazione degli algoritmi.


L'idea di fondo è di risolvere ricorsivamente istanze più semplici del problema e combinare queste soluzioni parziali in una soluzione del problema originale.

Se le istanze sono molto semplici, la soluzione è di solito banale.

Un'idea per ordinare un array:

1. ordino la prima metà
2. ordino la seconda metà

1	2	3	4	5	6	7	8	9	10
-4	2	5	7	13	0	1	4	6	11



## Divide-et-Impera

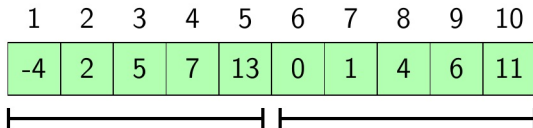
Il paradigma divide-et-impera è uno strumento tipico di progettazione degli algoritmi.

L'idea di fondo è di risolvere ricorsivamente istanze più semplici del problema e combinare queste soluzioni parziali in una soluzione del problema originale.

Se le istanze sono molto semplici, la soluzione è di solito banale.

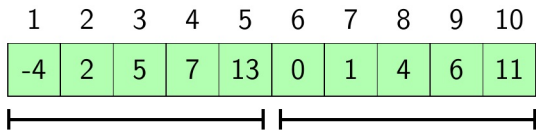
Un'idea per ordinare un array:

1. ordino la prima metà
2. ordino la seconda metà
3. unisco in qualche modo le due metà



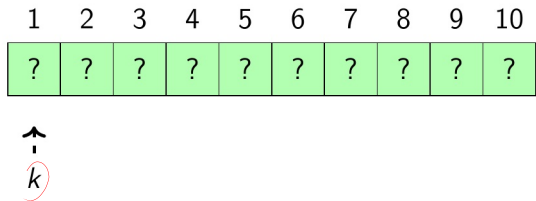
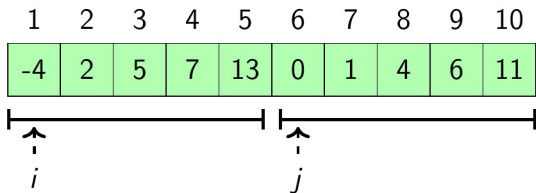


Come unisco (**merge**) due array ordinati?

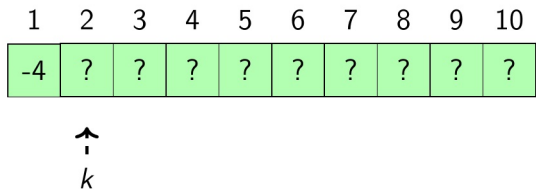
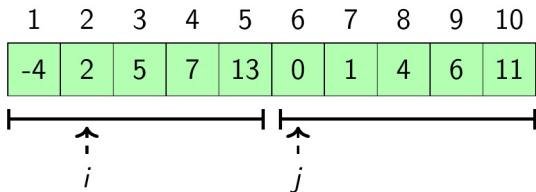




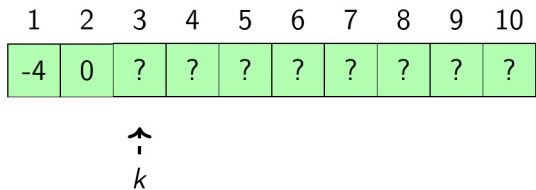
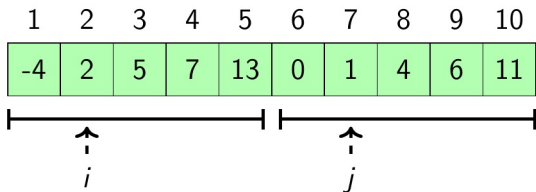
Come unisco (**merge**) due array ordinati?



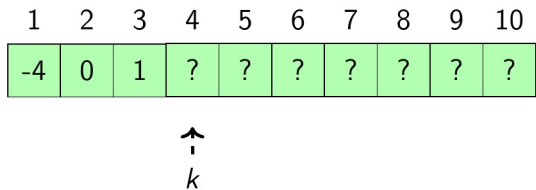
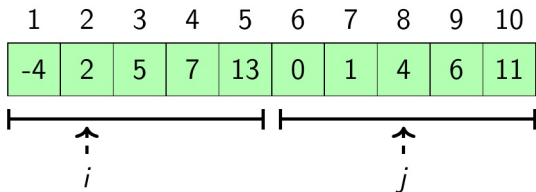
Come unisco (**merge**) due array ordinati?



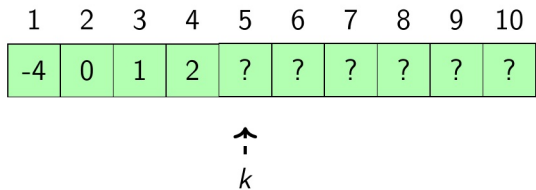
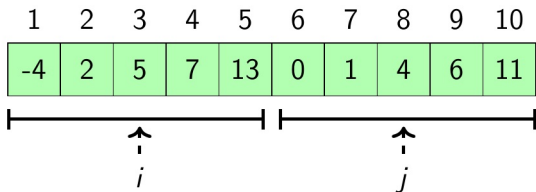
Come unisco (**merge**) due array ordinati?



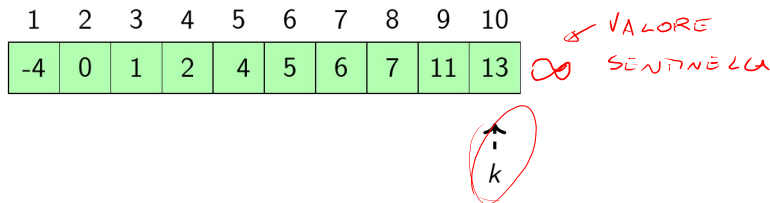
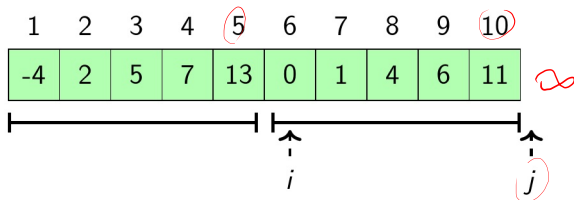
Come unisco (**merge**) due array ordinati?



Come unisco (**merge**) due array ordinati?



Come unisco (**merge**) due array ordinati?





## Merge sort: Pseudo-Code

```
def MERGESORT(A, begin=1, end=|A|):  
    if end > begin:  
        median  $\leftarrow$   $(begin + end) / 2$   
        MERGESORT(A, begin, median) * ORDINATO  
        MERGESORT(A, median+1, end) * ORDINATO  
  
        MERGE(A, begin, median, end) * ORDINATO  
    endif
```

Per INDUZIONE STRUTTURALE

BASE  $n = 1$

$n < n \Rightarrow n$

## Merge: Pseudo-Code

```
def MERGE(A, begin, median, end):
```

```
  L ← A[begin:median]
```

```
  R ← A[median+1:end]
```

```
  i, j ← 1, 1
```

```
  for k in range(begin, end):
```

```
    if (i > len(L) or
```

```
        (j ≤ len(R) and R[j] ≤ L[i])):
```

```
      A[k] ← R[j]
```

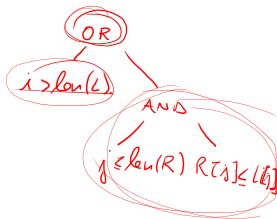
```
      j ← j+1
```

```
    else:
```

```
      A[k] ← L[i]
```

```
      i ← i+1
```

```
  endif
```



Esercizio: MERGE con  
valori sentinella.

$O(n)$

$O(1)$

# La Complessità di Merge Sort

Merge prende tempo  $\Theta(n)$

Ma mergesort contiene due chiamate ricorsive. Come facciamo a calcolare la complessità in questo caso?

COMPLESSITÀ PER ISTANZA di dimensione  $n$ .

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

Equazioni ricorsive