Si può fare di meglio?

- Gli algoritmi visti fino ad ora hanno costo $O(n^2)$
- È possibile fare di meglio?
  - Quanto meglio?

Algoritmi “divide et impera”

- Idea generale
  - Divide: Scomporre il problema in sottoproblemi dello stesso tipo (cioè sottoproblemi di ordinamento)
  - Risolvere ricorsivamente i sottoproblemi
  - Impera: Combinare le soluzioni parziali per ottenere la soluzione al problema di partenza
- Vedremo due algoritmi di ordinamento di tipo divide et impera
  - Quick Sort
  - Merge Sort

Quick Sort

- **Algoritmo di ordinamento**
  - **Basato su divide-et-impera**
  - **Caso medio: $O(n \log n)$, caso pessimo $O(n^2)$**
- **Caso medio vs caso pessimo**
  - **Il fattore costante di Quick Sort è migliore di Merge Sort**
  - **È possibile utilizzare tecniche "euristiche" per evitare il caso pessimo**
  - **Quindi spesso è preferito ad altri algoritmi**
- **Ulteriori dettagli**
  - **R. Sedgewick, "*Implementing Quicksort Programs*" Communications of the ACM, 21(10):847-857, 1978 http://portal.acm.org/citation.cfm?id=359631**

Quick Sort

- Inventato nel 1962 da Sir Charles Anthony Richard Hoare
  - All'epoca *exchange student* presso la Moscow State University
  - Vincitore del *Turing Award* (l'equivalente del Nobel per l'informatica) nel 1980 per il suo contributo nel campo dei linguaggi di programmazione
  - Hoare, C. A. R. *"Quicksort."* Computer Journal 5 (1): 10-15. (1962).

C. A. R. Hoare (1934—)
http://en.wikipedia.org/wiki/C._A._R._Hoare

Quick Sort

- Algoritmo ricorsivo “divide et impera”
  - Scegli un elemento x del vettore v, e partiziona il vettore in due parti considerando gli elementi ≤x e quelli >x
  - Ordina ricorsivamente le due parti
  - Restituisci il risultato concatenando le due parti ordinate
- R. Sedgewick, “*Implementing Quicksort Programs*”, Communications of the ACM, 21(10):847-857, 1978 http://portal.acm.org/citation.cfm?id=359631

Quick Sort

Input: Array *A*[*1..n*], indici *primo* e *ultimo* tali che 1 ≤ *primo* ≤ *ultimo* ≤ *n*

***Divide*: partiziona l'array *A*[*primo..ultimo*] in due sottovettori *A*[*primo..j*–1]e *A*[*j+1..ultimo*] (eventualmente vuoti) in modo che:**

$$\forall i \in [primo, j-1] \quad : \quad A[i] \leq A[j]$$
$$\forall i \in [j+1, ultimo] \quad : \quad A[i] \geq A[j]$$

A*[*j*] prende il nome di *perno

***Impera*: ordina i due sottovettori *A*[*primo..j*–1] e *A*[*j+1..ultimo*] richiamando ricorsivamente Quick Sort**

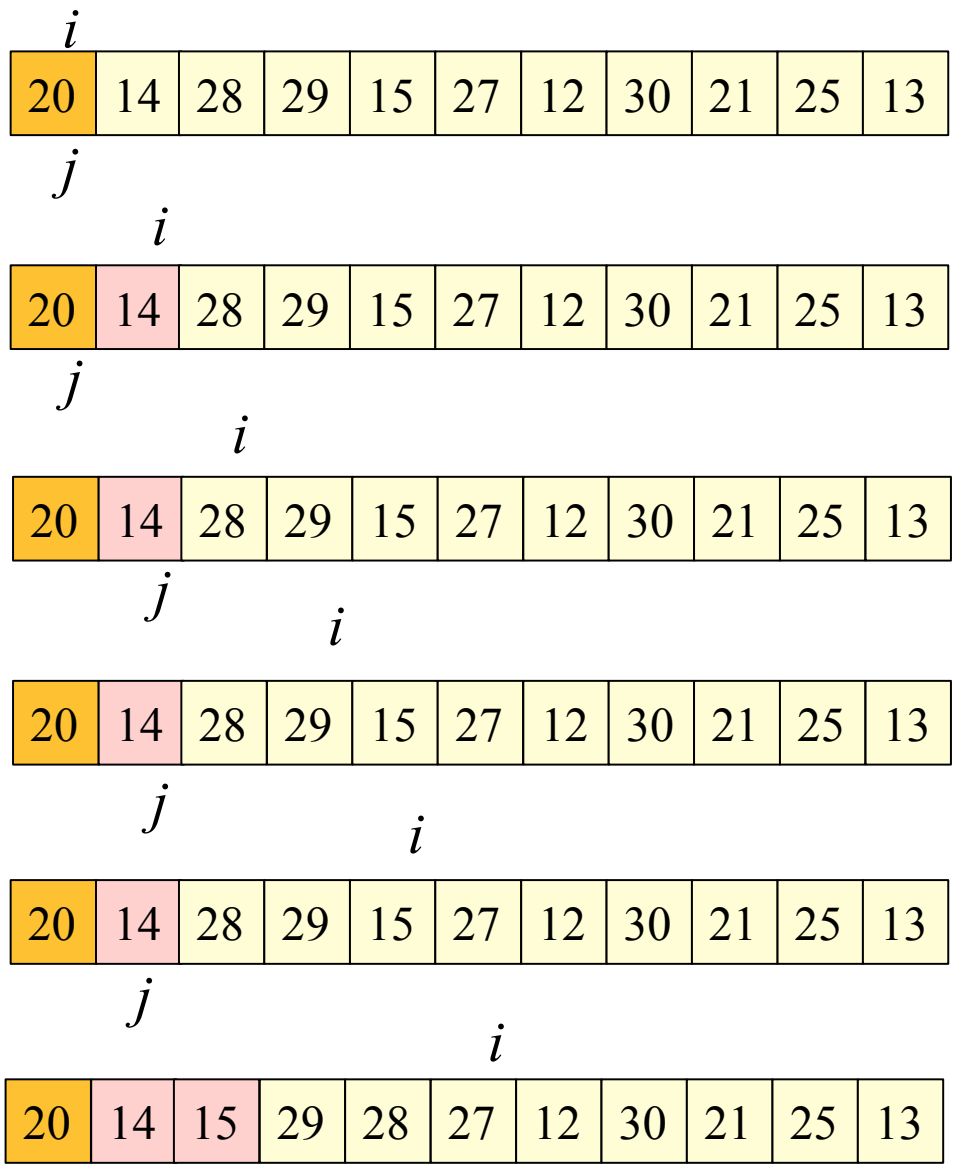
***Combina*: non fa nulla; i due sottovettori ordinati e l'elemento *A*[*j*] sono già ordinati**

Quick Sort: pseudocodice

```
QuickSort(ITEM[] A, integer primo, integer ultimo)
    if primo < ultimo then
        integer j ← perno(A, primo, ultimo)
        QuickSort(A, primo, j − 1)
        QuickSort(A, j + 1, ultimo)
```

```
integer perno(ITEM[] A, integer primo, integer ultimo)
    ITEM x ← A[primo]
    integer j ← primo
    for integer i ← primo to ultimo do
        if A[i] < x then
            j ← j + 1
            A[i] ↔ A[j]
    A[primo] ← A[j]
    A[j] ← x
    return j
```

Quick Sort: Esempio di funzionamento partition



$A[i] \geq x$

$A[i] < x$: $j \leftarrow j + 1$, $A[i] \leftrightarrow A[j]$

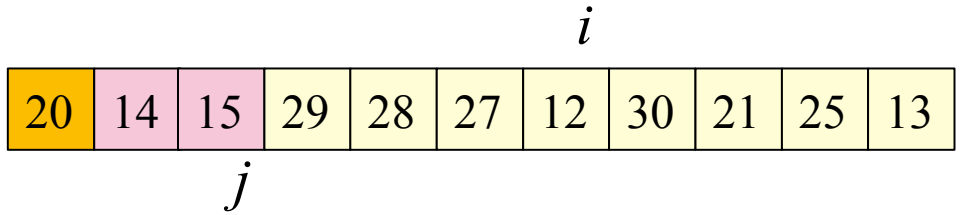
$A[i] \geq x$

$A[i] \geq x$

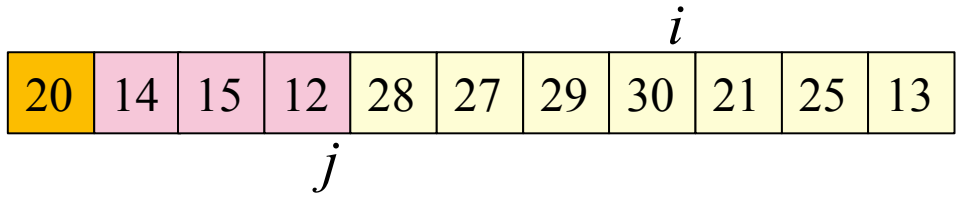
$A[i] < x$: $j \leftarrow j + 1$, $A[i] \leftrightarrow A[j]$

$A[i] \geq x$

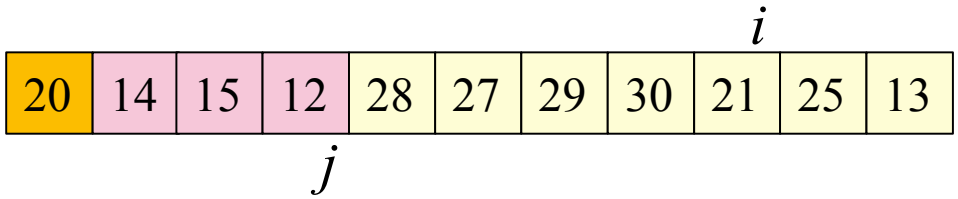
Quick Sort: Esempio di funzionamento Partition



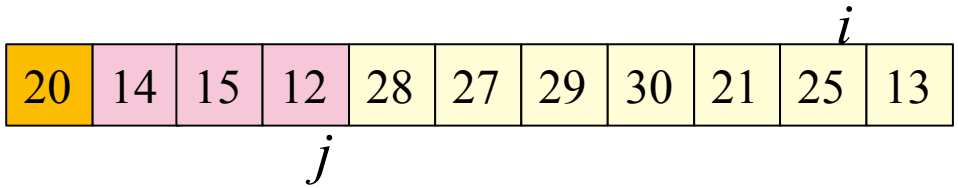
$A[i] < x$: $j \leftarrow j + 1$, $A[i] \leftrightarrow A[j]$



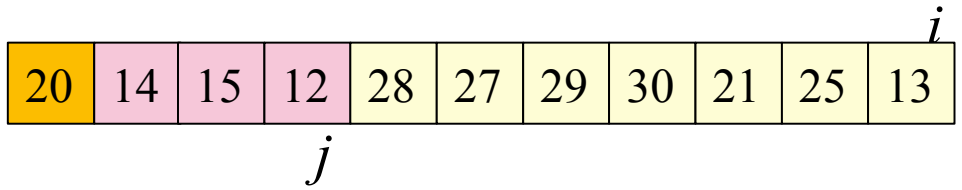
$A[i] \geq x$



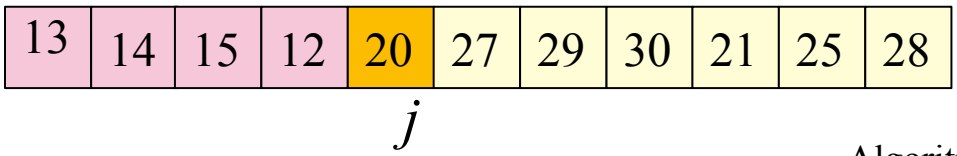
$A[i] \geq x$



$A[i] \geq x$

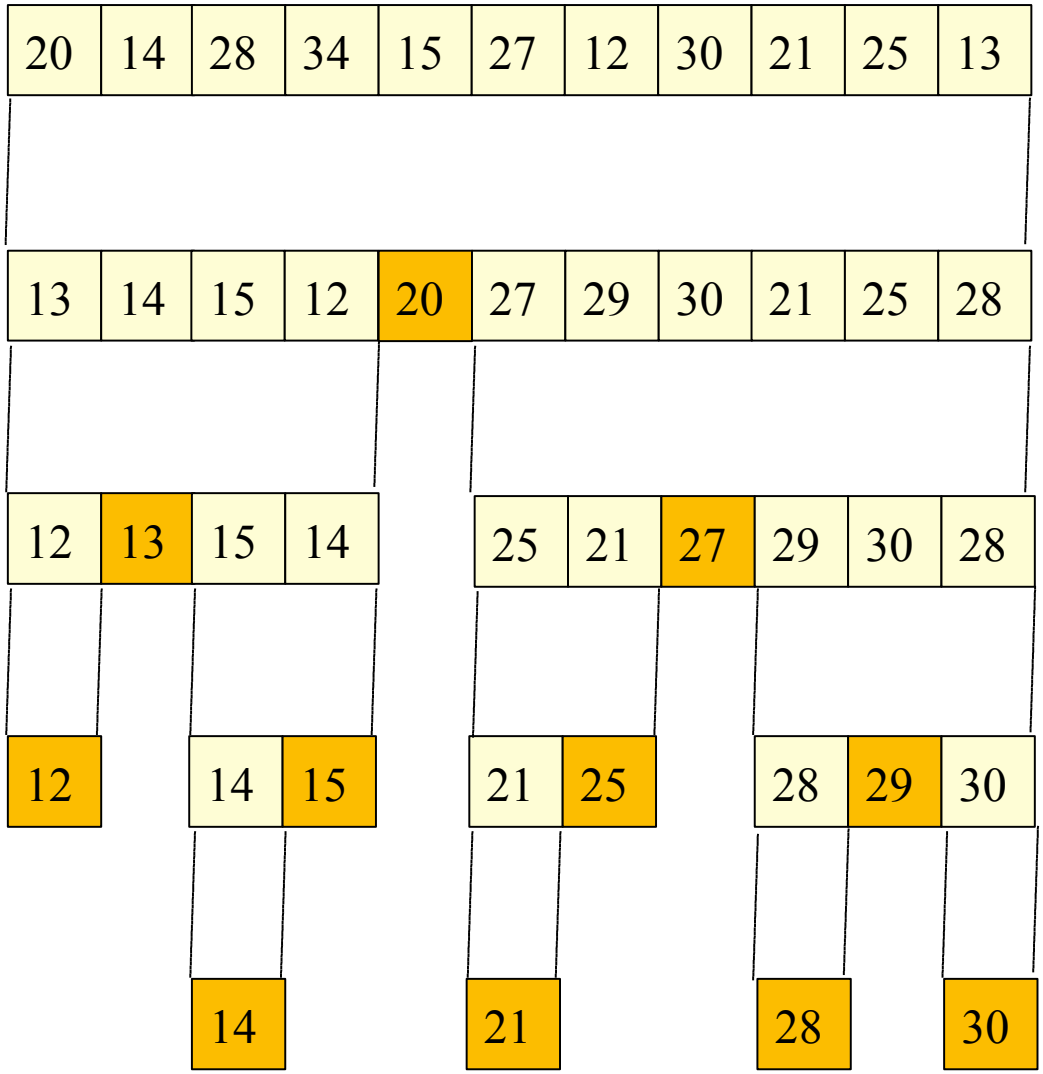


$A[i] < x$: $j \leftarrow j + 1$, $A[i] \leftrightarrow A[j]$

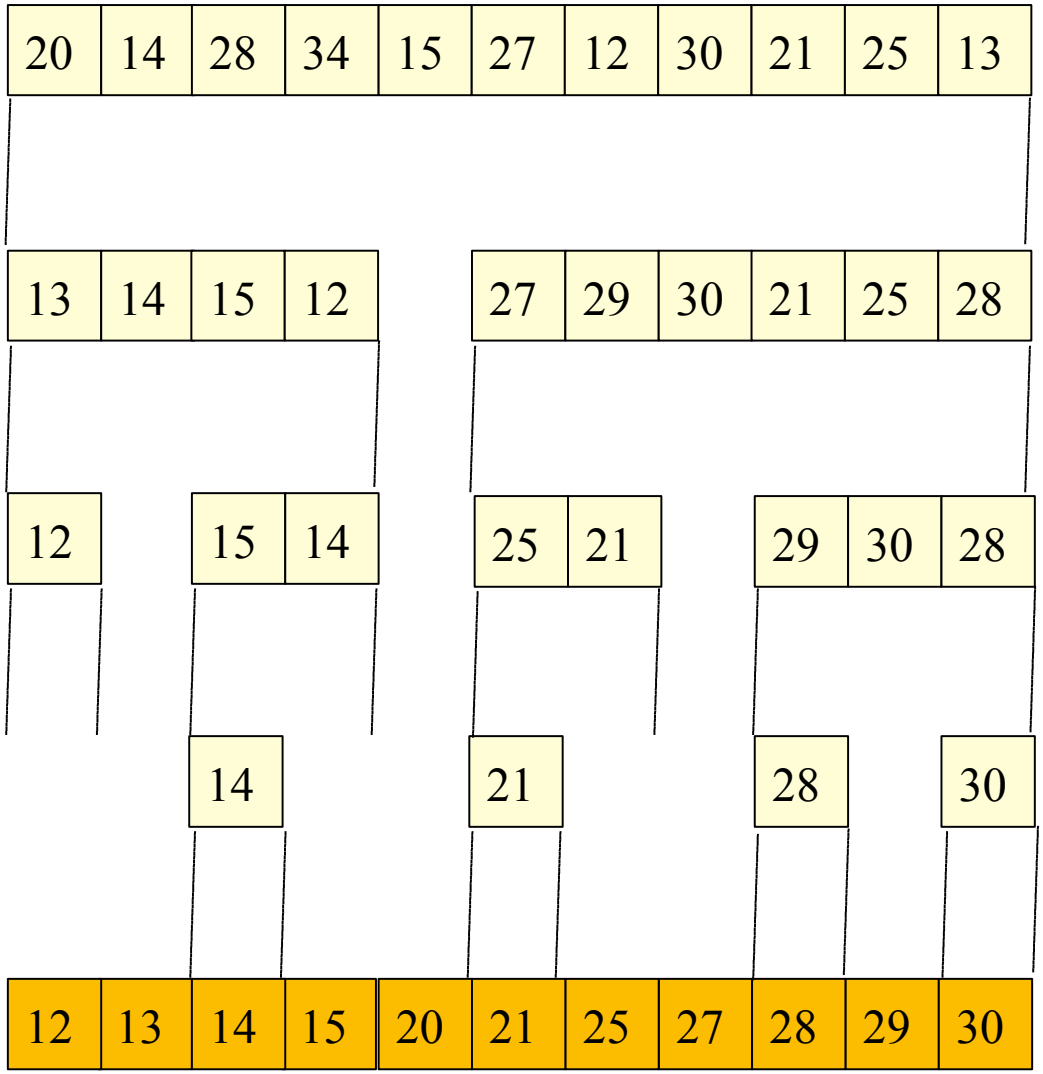


$A[primo] \leftarrow A[j]$; $A[j] \leftarrow x$

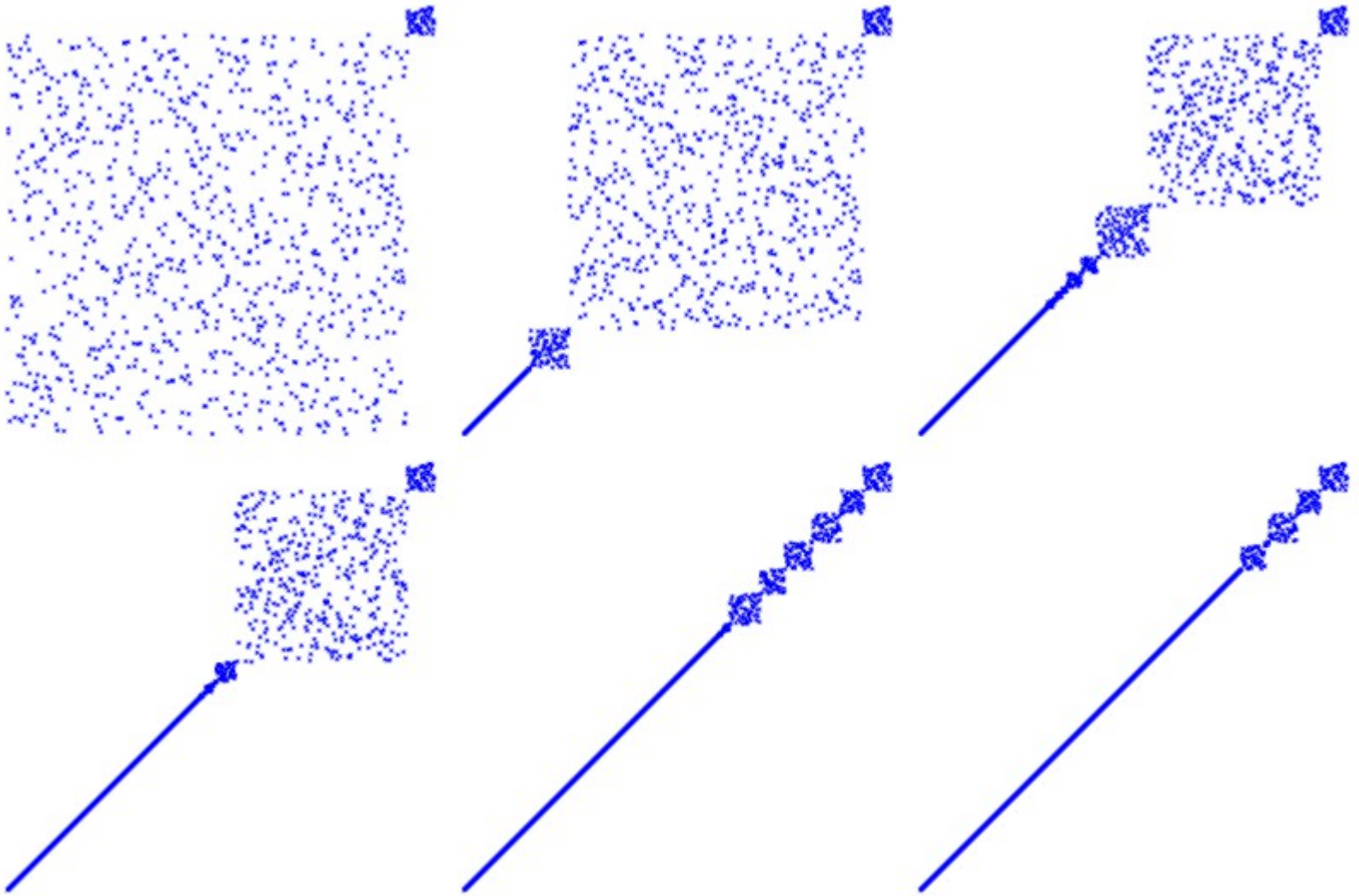
Quick Sort: esempio di ricorsione



Quick Sort: esempio di ricorsione



Quick Sort per immagini



Quick Sort: Analisi del costo

- Costo di partition(): $\Theta(f-i)$
- Costo Quick Sort: Dipende dal partizionamento
- Partizionamento peggiore
  - Dato un problema di dimensione n, viene sempre diviso in due sottoproblemi di dimensione 0 e n-1
  - $T(n) = T(n-1)+T(0)+\Theta(n) = T(n-1) + \Theta(n) = \Theta(n^2)$
- **Domanda**: Quando si verifica il caso pessimo?
- Partizionamento migliore
  - Data un problema di dimensione n, viene sempre diviso in due sottoproblemi di dimensione n/2
  - $T(n) = 2T(n/2)+\Theta(n) = \Theta(n \log n)$ (caso 2 Master Theorem)

Merge Sort

- Inventato da John von Neumann nel 1945
- Algoritmo *divide et impera*
- Idea:
  - Dividere A[] in due meta' A1[] e A2[] (senza permutare) di dimensioni uguali;
  - Applicare ricorsivamente Merge Sort a A1[] e A2[]
  - Fondere (*merge*) gli array ordinati A1[] e A2[] per ottenere l'array A[] ordinato

John von Neumann (1903—1957)
http://en.wikipedia.org/wiki/John_von_Neumann

- **Merge Sort**
  - **E' basato sulla tecnica *divide-et-impera* vista in precedenza**
  - **Divide:**
    - **Dividi l'array di *n* elementi in due sottovettori di *n/2* elementi**
  - **Impera:**
    - **Chiama MergeSort ricorsivamente su i due sottovettori**
  - **Combina:**
    - **Unisci (*merge*) le due sequenze ordinate**

Merge Sort vs Quick Sort

- Quick Sort:
  - partizionamento complesso, merge banale (di fatto nessuna operazione di merge è richiesta)
- Merge Sort:
  - partizionamento banale, operazione merge complessa

Merge Sort

```
Merge(ITEM A[ ], integer primo, integer ultimo, integer mezzo)
integer i, j, k, h
i ← primo; j ← mezzo + 1; k ← primo
while i ≤ mezzo and j ≤ ultimo do
    if A[i] ≤ A[j] then
        B[k] ← A[i]
        i ← i + 1
    else
        B[k] ← A[j]
        j ← j + 1
    k ← k + 1
j ← ultimo
for h ← mezzo downto i do
    A[j] ← A[h]
    j ← j − 1
for j ← primo to k − 1 do  A[j] ← B[j]
```

Merge Sort

- **Come funziona** merge():

A							*B*				
1	5	7	+	2	4	6					
	5	7	+	2	4	6	1				
	5	7	+		4	6	1	2			
	5	7	+			6	1	2	4		
		7	+			6	1	2	4	5	
		7	+				1	2	4	5	6
			+			7	1	2	4	5	6
1	2	4	+	5	6	7					

- **Domanda**
  - **Costo computazionale di merge()**

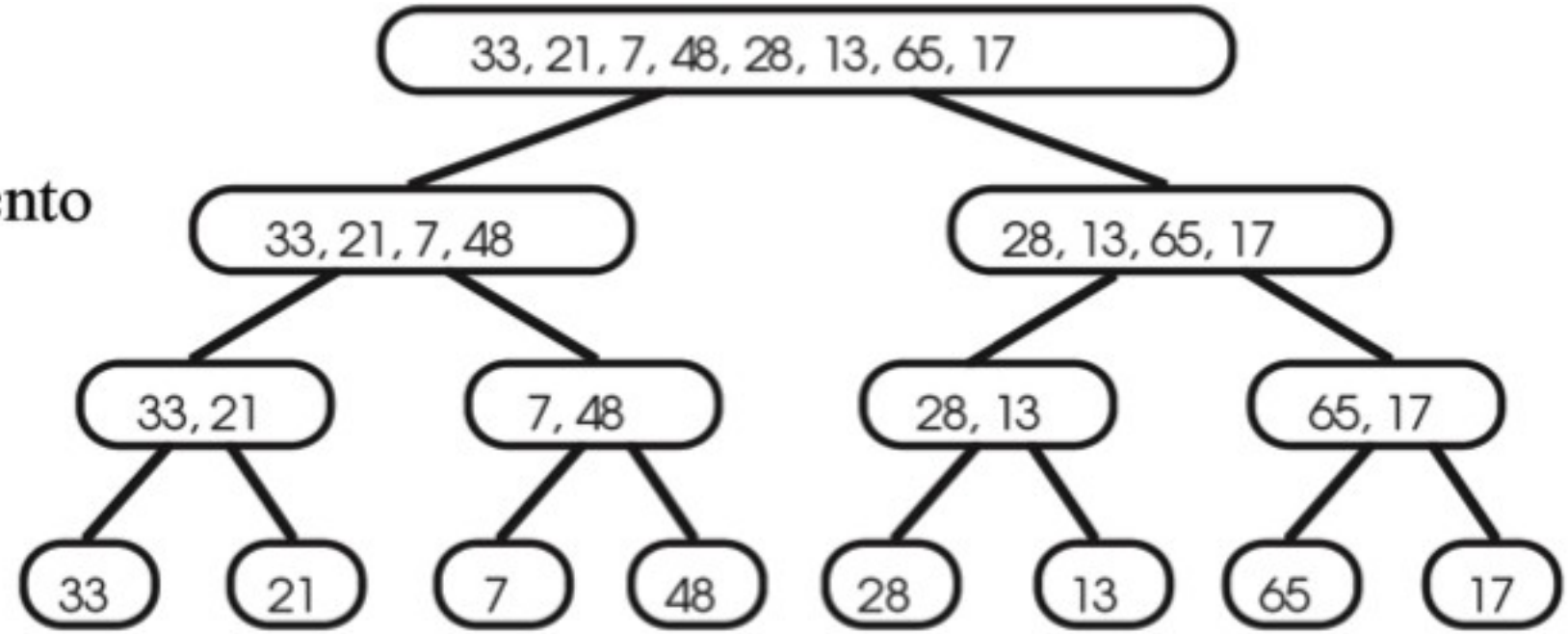
Programma completo

- Chiama ricorsivamente se stesso e usa **merge()** per unire i risultati
- Caso base: sequenze di lunghezza ≤ 1 sono già ordinate

```
MergeSort(ITEM A[ ], integer primo, integer ultimo)
if primo < ultimo then
    integer mezzo ← ⌊(primo + ultimo)/2⌋
    MergeSort(A, primo, mezzo)
    MergeSort(A, mezzo + 1, ultimo)
    Merge(A, primo, ultimo, mezzo)
```

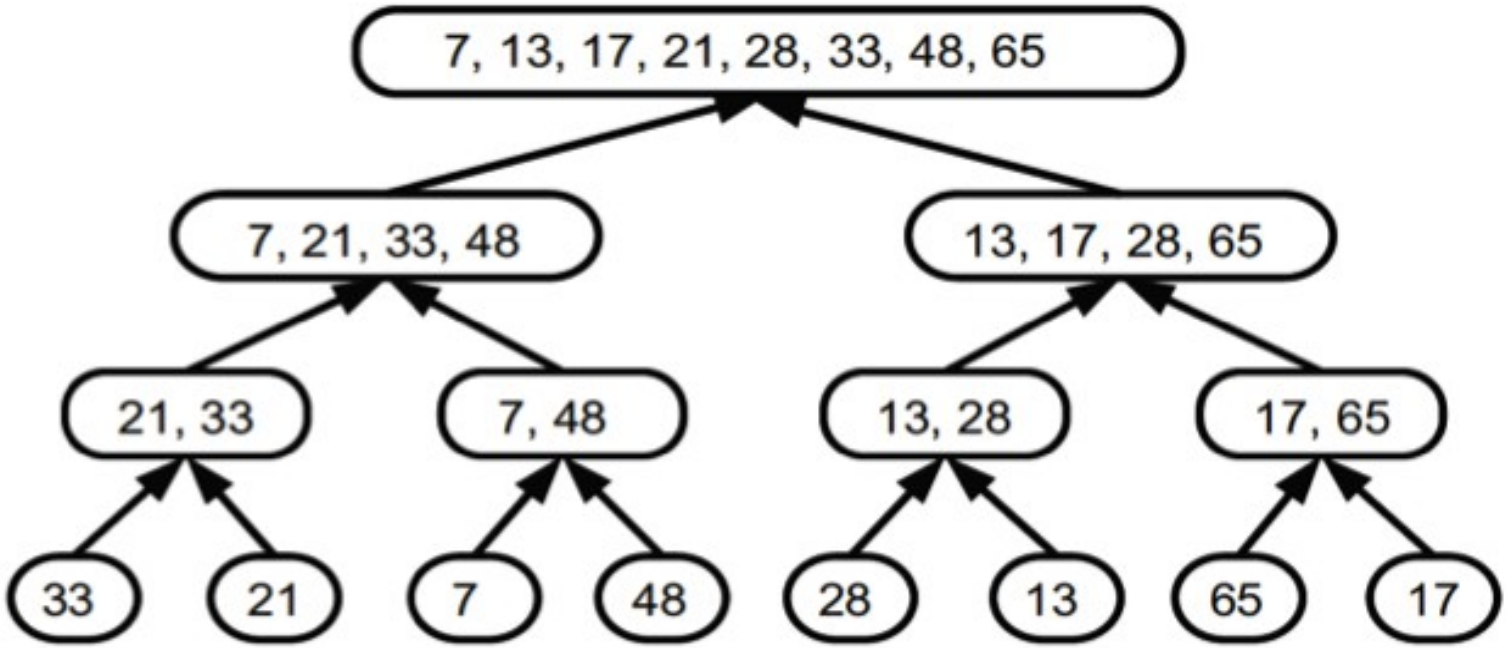
Merge Sort

Partizionamento

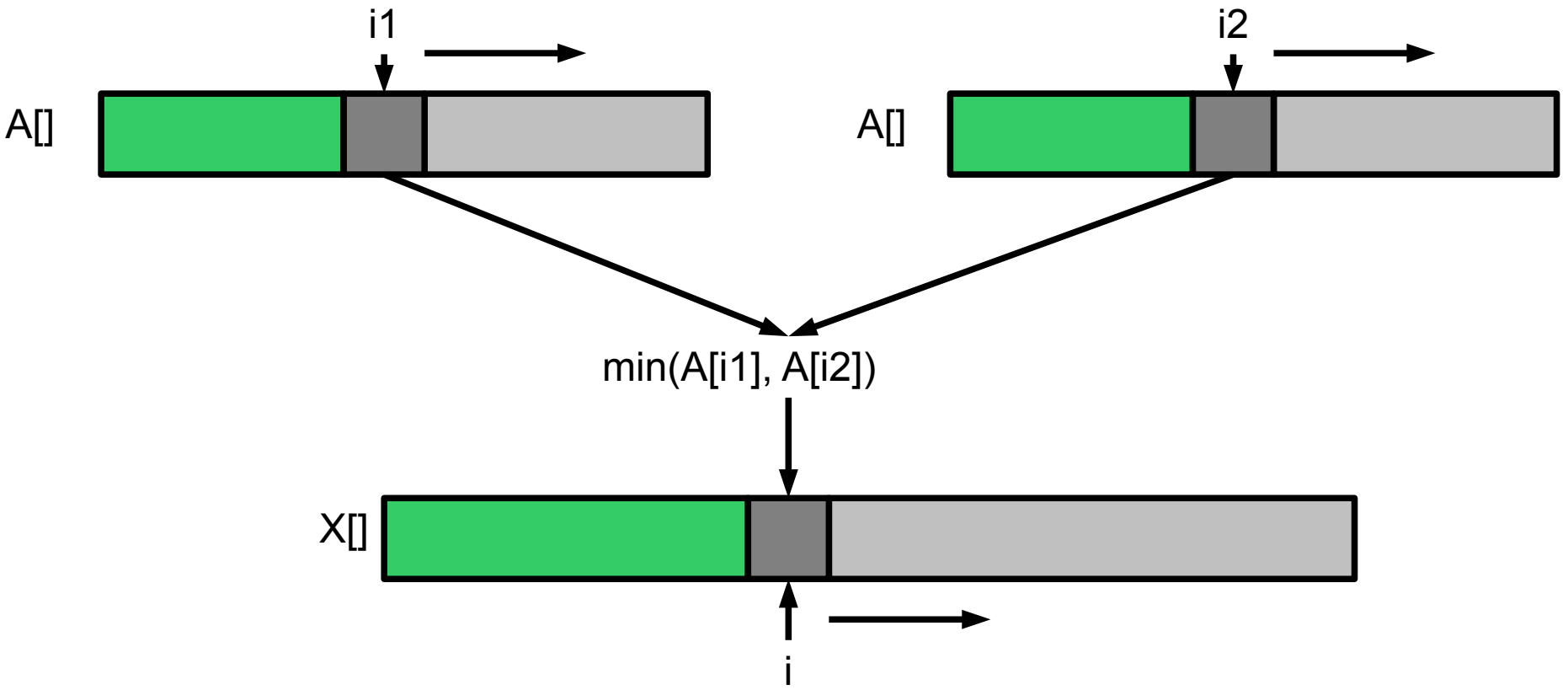


Merge Sort

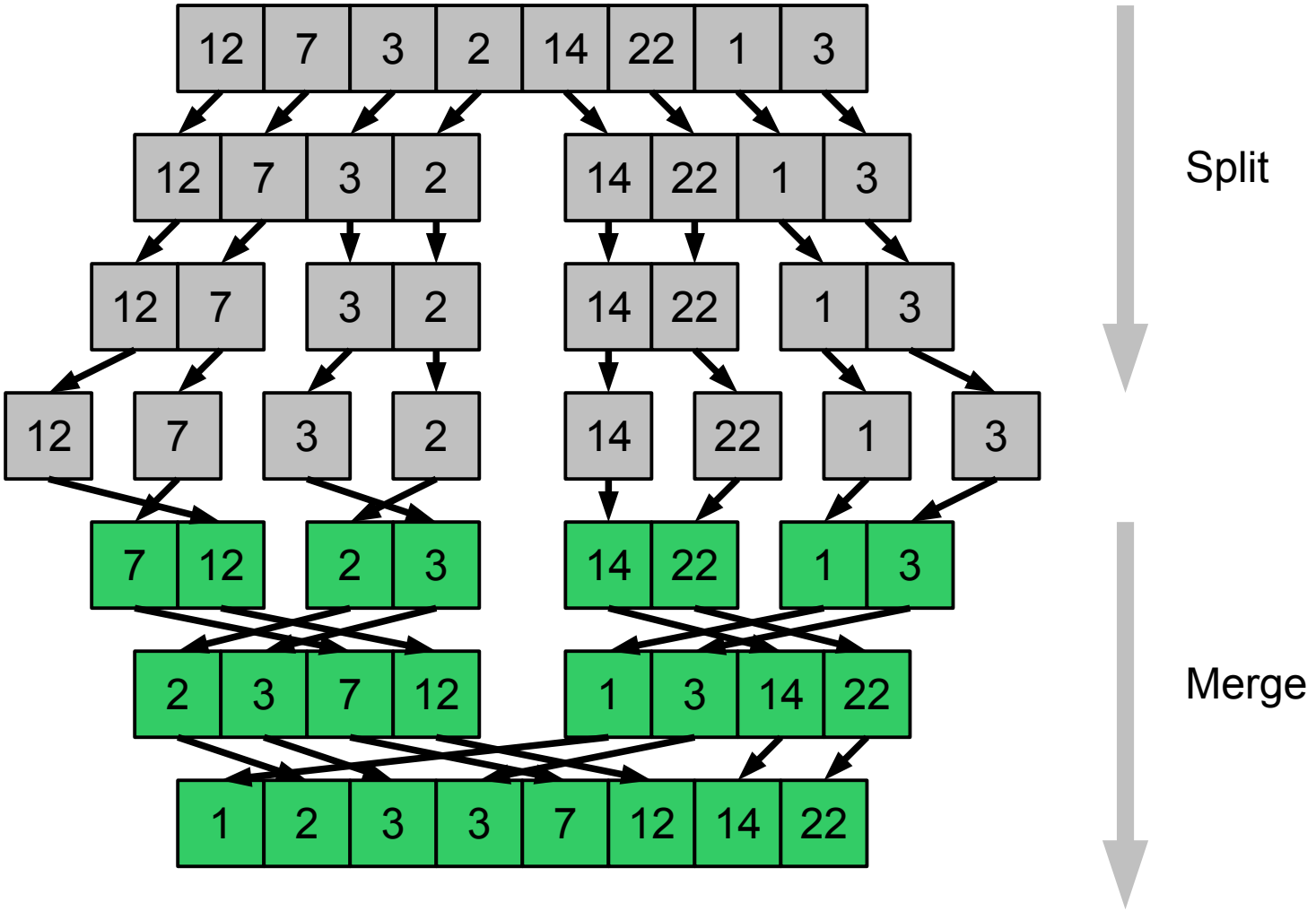
Merge



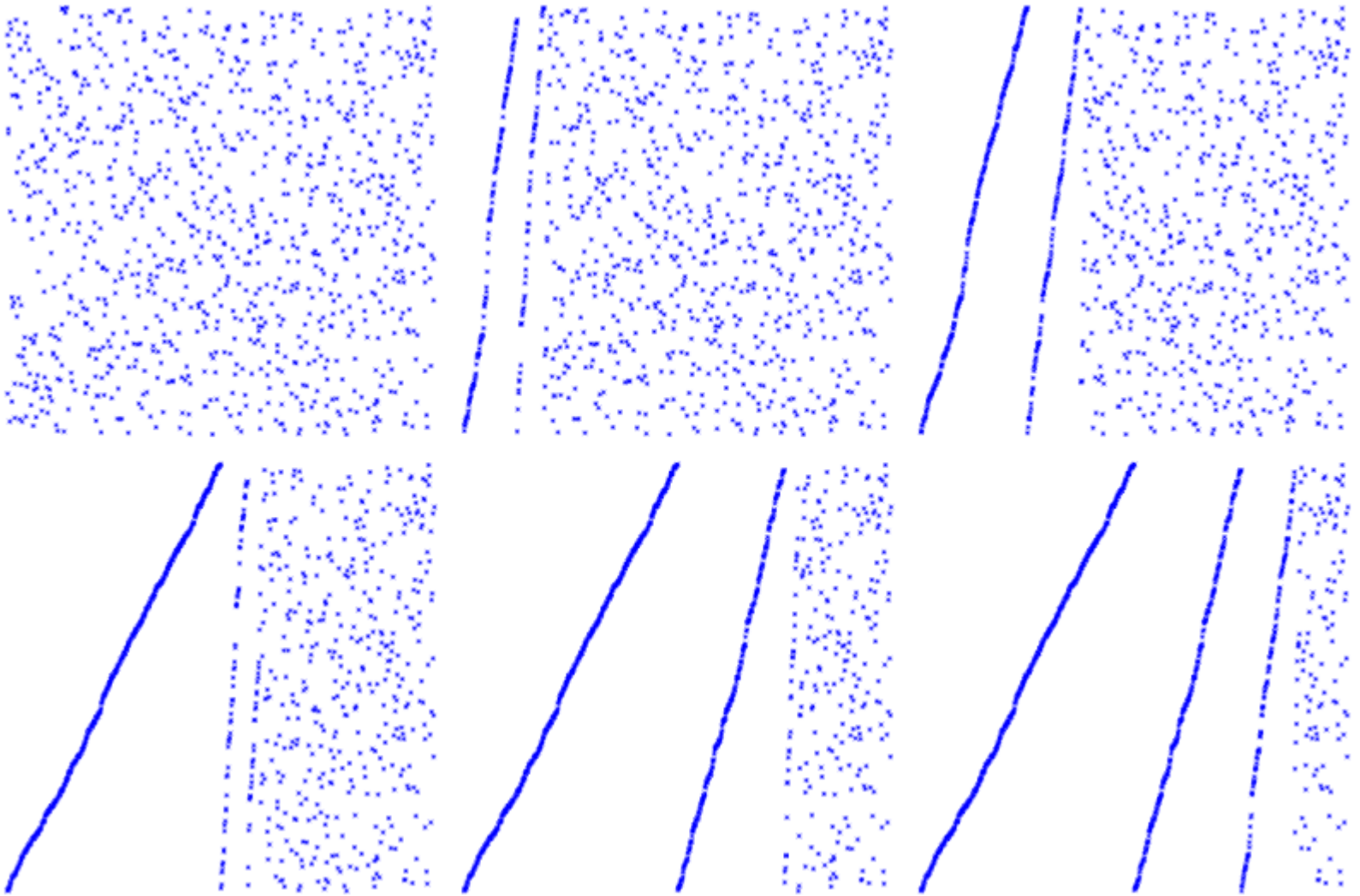
Operazione merge()



Merge Sort: esempio



Merge Sort per immagini



Merge Sort: complessità

- $T(n) = 2T(n/2) + \Theta(n)$
- In base al Master Theorem (caso 2), si ha
$$T(n) = \Theta(n \log n)$$
- La complessità di Merge Sort non dipende dalla configurazione iniziale dell'array da ordinare
  - Quindi il limite di cui sopra vale nei casi ottimo/pessimo/medio
- Svantaggi rispetto a Quick Sort: Merge Sort richiede ulteriore spazio (non ordina in-place)
  - Jyrki Katajainen, Tomi Pasanen, Jukka Teuhola, "*Practical in-place mergesort*", http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.8523