16. Algoritmi 2 – Algortimi di ordinamento

Corso di Informatica

Corso di Laurea in Matematica (D.M. 270/04) - A.A. 2020/2021
Angelo Cardellicchio
angelo.cardellicchio@uniba.it
09/11/2020

Outline

- Il problema dell’Ordinamento
- Selection Sort
- Insertion Sort
- Paradigma divide-and-conquer
- Merge Sort
- Quick Sort

Il problema dell'Ordinamento

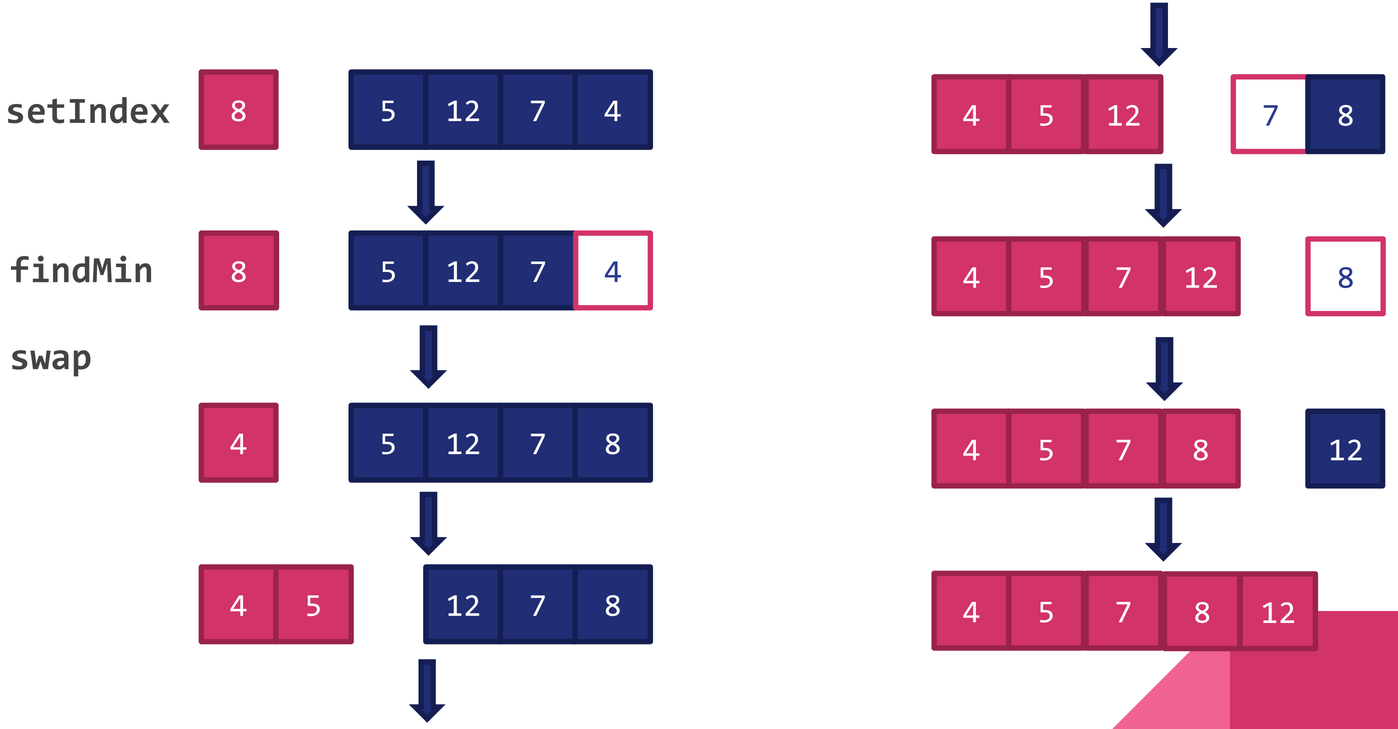
- Un algoritmo di **ordinamento** (**sorting**) aiuta ad ordinare una lista
- Normalmente, si ragiona in termini numerici, dato che le liste di questo tipo sono le più semplici da ordinare
- È però possibile estendere il concetto a qualsiasi tipo di lista: sfruttando opportunamente diverse strutture dati, infatti, è possibile ricondursi al caso numerico, ed implementare un opportuno algoritmo di ordinamento

Selection Sort – Descrizione dell'algoritmo

- Il **selection sort** è un algoritmo **iterativo**
- La lista iniziale è suddivisa in due sottoliste
  - *Quella a sinistra rappresenta gli elementi già ordinati, quella a destra quelli non ancora ordinati*
- Ogni iterazione ha tre passi fondamentali
  - *Nel primo (che chiamiamo* **`setIndex`***) è aggiornato l'indice dell'elemento attualmente analizzato*
  - *Nel secondo (che chiamiamo* **`findMin`***) è individuato il valore minimo tra quelli non ordinati*
  - *Nel terzo (che chiamiamo* **`swap`***) sostituiamo l'elemento attualmente analizzato con quello trovato nel primo step*

8	5	12	7	4

Selection sort – Descrizione dell’algoritmo

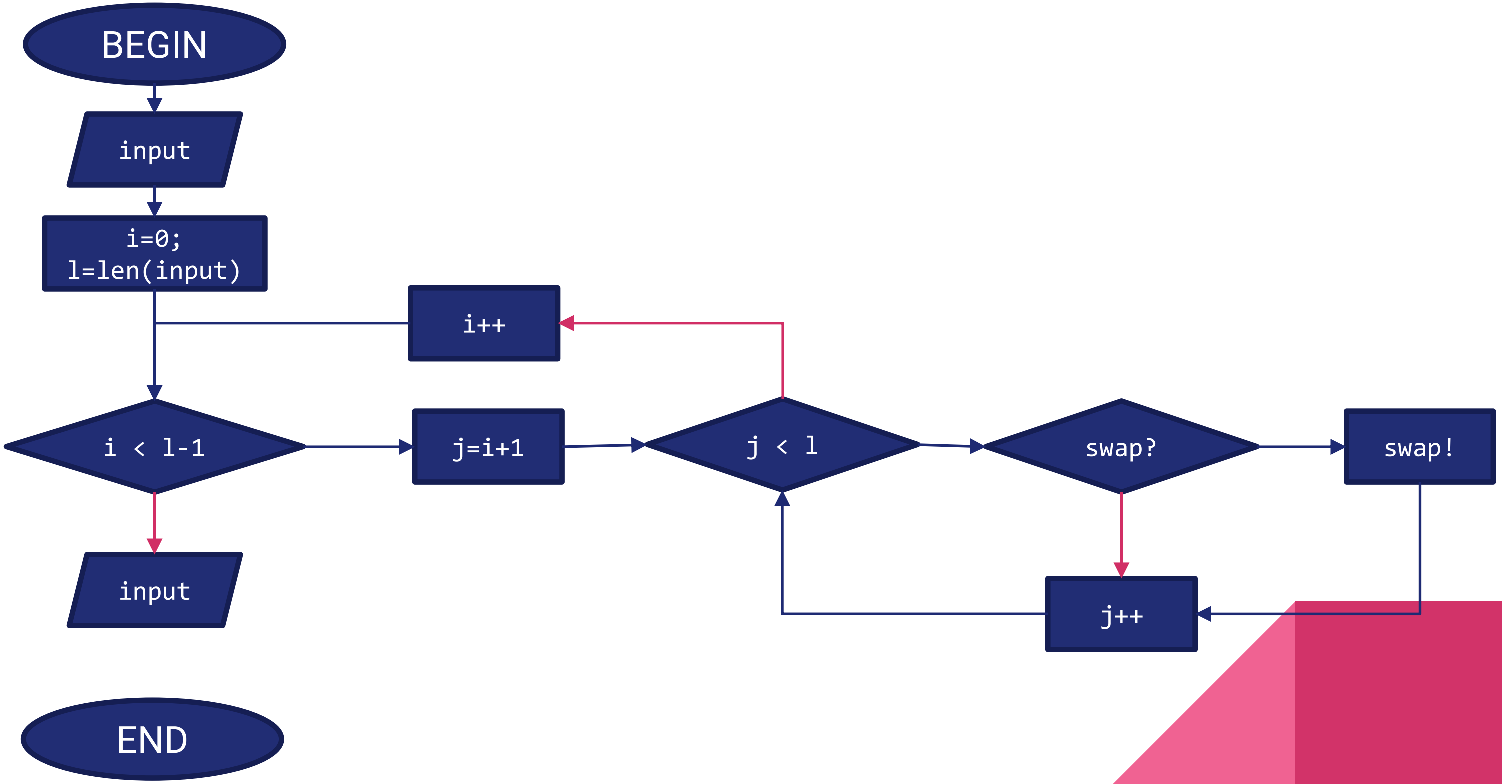


Selection sort – Pseudocodice

```
STEP 1 -> split(LIST);
STEP 2 -> i = 0;
STEP 3 -> setIndex(i);
STEP 4 -> MIN = LEFT_LIST(i);
STEP 5 -> findMin(MIN, RIGHT_LIST);
STEP 6 -> swap();
STEP 7 -> if length(RIGHT_LIST) IS 0
              THEN RETURN LEFT_LIST
          ELSE
              i++;
              GOTO STEP 3;
```

```
findMin(MIN, RIGHT_LIST)
STEP 1 -> j = 0;
STEP 2 -> k = length(RIGHT_LIST)
STEP 3 -> WHILE j < k - 1
              IF RIGHT_LIST(k) < MIN
                  MIN = RIGHT_LIST(k)
              j++
STEP 4 -> RETURN MIN
```

Selection sort – Diagramma di flusso



Selection sort – Analisi Computazionale

- Il selection sort itera su tutti gli indici di un array
- Supponiamo un array di n elementi
  - *Alla prima iterazione, avremo al più* $(n-1)$ *confronti, ed uno swap*
  - *Alla seconda iterazione, avremo al più* $(n-2)$ *confronti, ed uno swap, e così via*
- Ciò significa che avremo bisogno al più di $\mathrm{n} + (\mathrm{n}-1) + \cdots + 1$ operazioni
- Questo valore equivale ad una serie aritmetica pari a

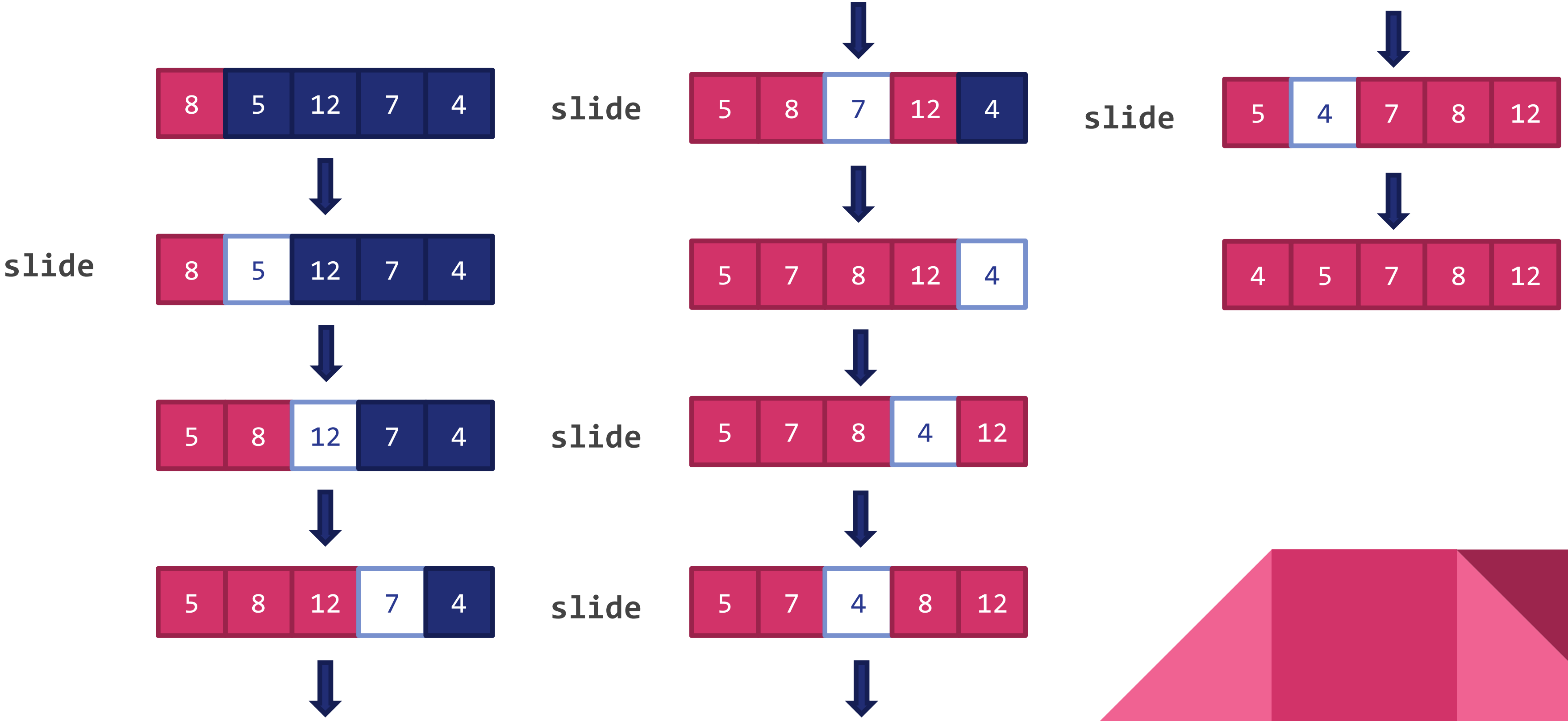
$$\frac{n^2}{2} + \frac{n}{2} \Rightarrow T \in O(n^2)$$

Insertion sort – Descrizione dell’algoritmo

- Algoritmo basato sull’**inserimento** di un valore in un subarray ordinato
- Parte considerando il primo elemento dell’array come un subarray ordinato, ed usa un elemento (chiamato **key**, **chiave**) per la comparazione
- La chiave viene comparata con l’elemento alla sua sinistra e, se inferiore, viene effettuata l’operazione di **slide**

8	5	12	7	4

Insertion sort – Descrizione dell’algoritmo



Insertion sort – Pseudocodice

```
STEP 1 -> SORTED = ARRAY[0];
STEP 2 -> KEY = ARRAY[1];
STEP 3 -> IF LENGTH(SORTED) = LENGTH(ARRAY)
              RETURN SORTED
          ELSE
              FOR I = 0; I ++; I < LENGTH(SORTED)
                  IF KEY < SORTED[I]
                      slide(KEY, SORTED[I])
```

Insertion sort – Analisi Computazionale

- Caso peggiore: array completamente ordinato
  - *Alla prima iterazione, dovremo effettuare al più uno slide*
  - *Alla* n*-ma, al più* $(n-1)$ *slide*
- Il numero massimo di operazioni è:

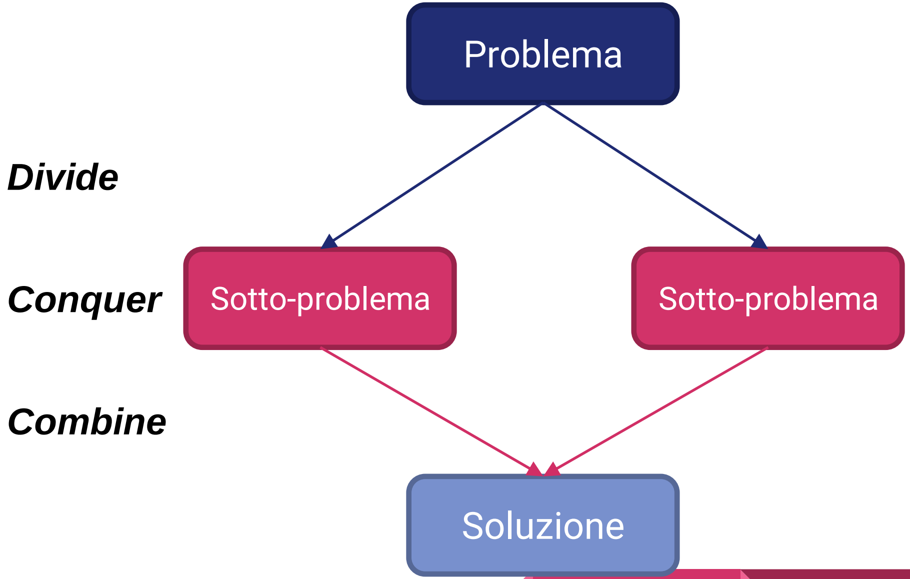
$$(n-1)+(n-2)+\cdots+2+1$$

- È praticamente la stessa del selection sort!
- Cosa succede se l'array è già ordinato?
  - *In questo caso, ci sono soltanto* $n-1$ *operazioni, per cui* $T \in O(n)$
- Dobbiamo però sempre considerare quella di caso peggiore, quindi

$$T \in O(n^2)$$

Paradigma divide-and-conquer

- Paradigma di tipo ricorsivo
- Schematizzabile in tre parti
  - ***Divide***: *suddivide il problema in sotto-problemi*
  - ***Conquer***: *risolve i sotto-problemi, se riconducibili al caso base*
  - ***Combine***: *combina le soluzioni*



Merge Sort – Descrizione dell'algoritmo

- Il **Merge Sort** si basa sul paradigma divide-and-conquer
- Dobbiamo individuare il sotto-problema da risolvere
- **Idea**: ordinare array dalle dimensioni inferiori rispetto a quello iniziale!
  - *Utilizziamo la notazione* $array[l,r]$ *per indicare il sotto-array che va dall'indice* l *all'indice* r

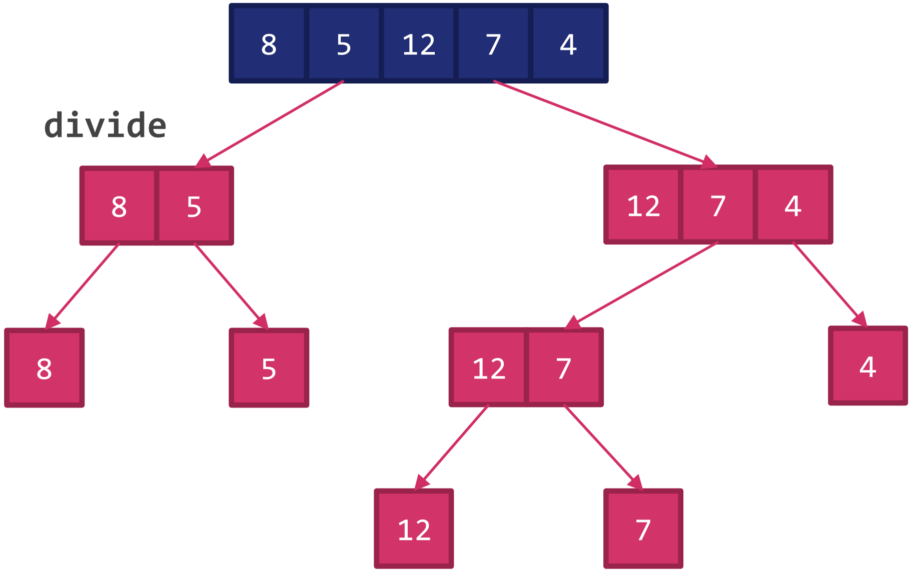
$array$	8	5	12	7	4

$array[0,2]$	8	5	12

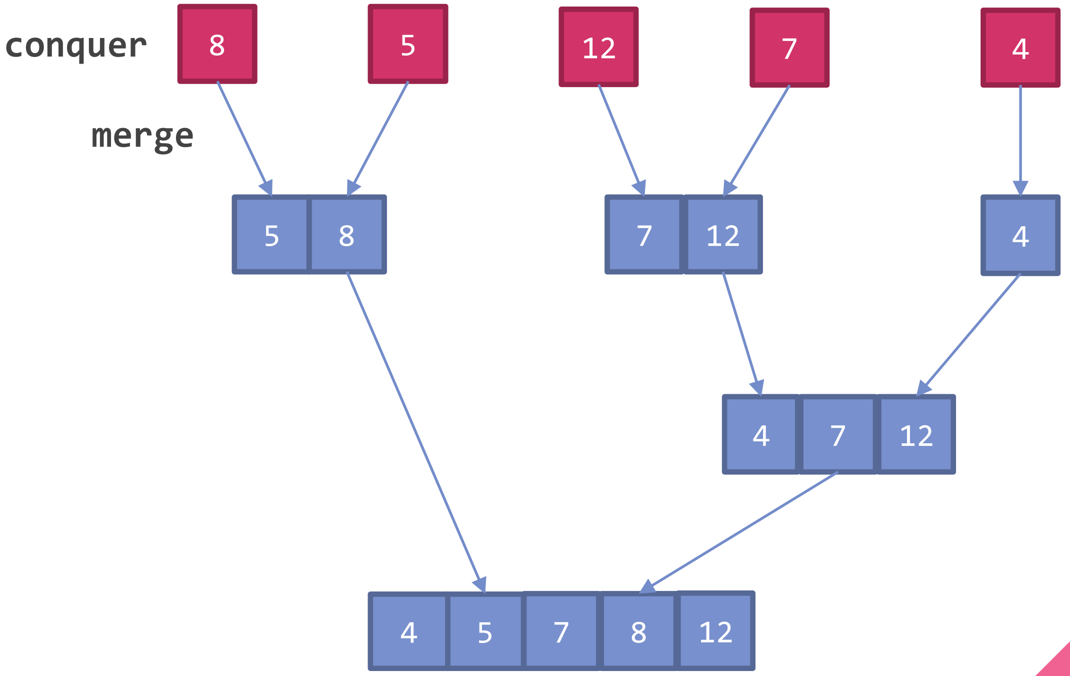
Merge Sort – Descrizione dell’algoritmo

- **Divide:** trova il valore intermedio q tra l ed r
  - *Nel nostro caso, $q = 2.5$, per cui lo approssimiamo all’intero immediatamente inferiore.*
- **Conquer**: ordina i due sotto array creati nel passo precedente
  - *Nel nostro caso, opererà sugli array $array[l, q]$ ed $array[q + 1, r]$*
- **Combine**: combina i risultati ottenuti al passo precedente.
  - *Uniremo i due sotto-array combinati in un unico $array[l, r]$*
- Il caso base si verifica quando abbiamo array di dimensione unitaria o nulla
  - *È facile verificare che questi array sono già ordinati*
  - *Nel caso base, $l \geq r$, per cui consideriamo la ricorsione fino ad $l < r$*

Merge Sort – Descrizione dell’algoritmo



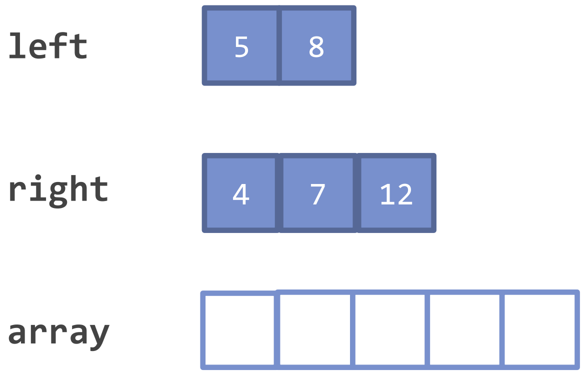
Merge Sort – Descrizione dell’algoritmo



Merge Sort – Descrizione dell'algoritmo

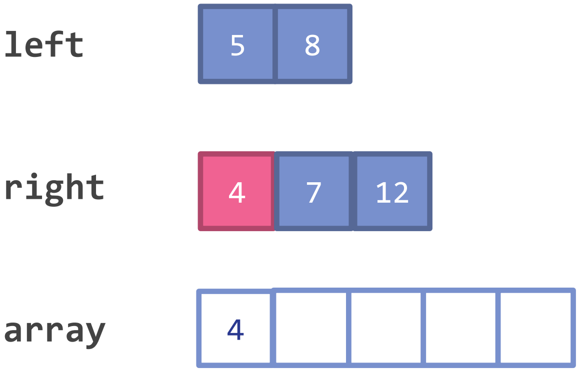
- **Combine:** è dove i valori vengono effettivamente riordinati
- Chiamiamo $left = array[l, q]$ e $right = array[q + 1, r]$
- Il nostro obiettivo è copiare in $array[l]$ il valore più piccolo tra quelli presenti in $left$ e $right$
- Avremo bisogno di tre contatori:
    - i: contatore su $left$
    - j: contatore su $right$
    - k: contatore su $array$

Merge Sort – Descrizione dell’algoritmo




```
i = 0;
j = 0;
k = 1;
```

Merge Sort – Descrizione dell’algoritmo



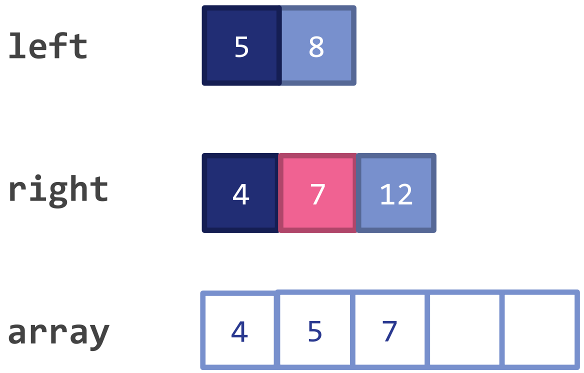

```
i = 0;
j = 1;
k = 1;
```

Merge Sort – Descrizione dell’algoritmo

left	5	8			
right	4	7	12		
array	4	5			

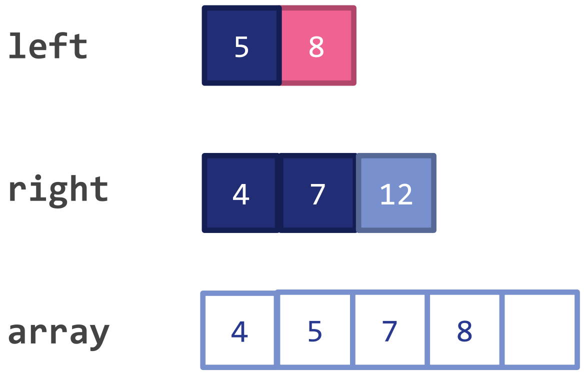
```
i = 1;
j = 1;
k = 2;
```

Merge Sort – Descrizione dell'algoritmo




```
i = 1;
j = 2;
k = 3;
```

Merge Sort – Descrizione dell’algoritmo




```
i = 2;
j = 2;
k = 4;
```

Merge Sort – Descrizione dell’algoritmo

`left`	5	8			
`right`	4	7	12		
`array`	4	5	7	8	12

```
i = 2;
j = 3;
k = 5;
```

Merge Sort – Analisi Computazionale

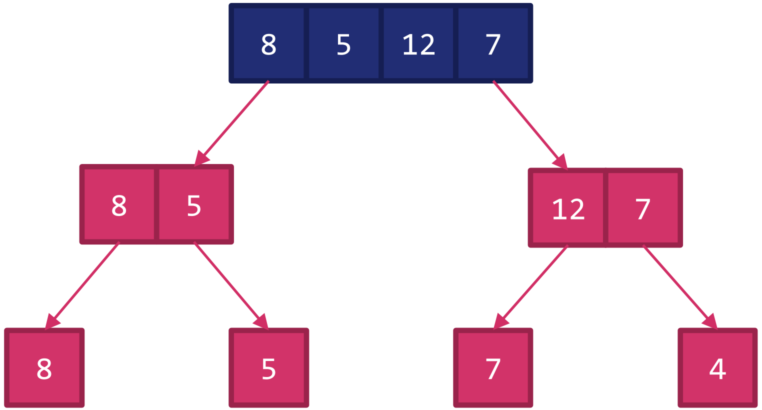
- **Combine**: ha complessità di $O(n)$
  - *Questo perché nell'array ci sono al più n elementi da confrontare*
- **Divide**: complessità trascurabile ($O(1)$)
- **Conquer**: chiamiamo ricorsivamente la funzione su un numero sempre dimezzato di elementi
  - *È qui che individuiamo il carico computazionale vero e proprio*

Merge Sort – Analisi Computazionale

- **Conquer**
  - *Notiamo che eseguire l'algoritmo su un array di n elementi equivale (circa) ad eseguirlo due volte su un array di $n/2$ elementi*
- Il numero di operazioni si **dimezza** ad ogni livello di ricorsione, ma **raddoppiano** gli array su cui effettuarle
  - *Questi effetti si annullano tra loro, per cui il costo ad ogni livello è costante e pari ad n*
- Di conseguenza, per analizzare il costo complessivo è necessario contare il numero di livelli su cui viene distribuita la ricorsione
  - *Per semplicità, supponiamo n pari e potenza di due*
  - *In questo caso, il numero di livelli $c = \log_2 n$*

Merge Sort – Analisi Computazionale

c = 3



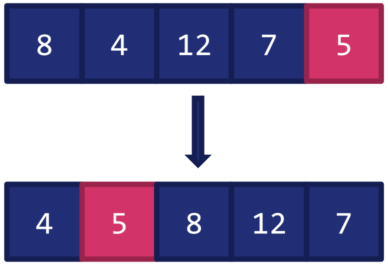
- Ne consegue che la complessità è pari ad n (numero di operazioni per ciascun livello) per $c = \log_2 n$ (numero di livelli)
  - *Ergo, siamo in* $O(n \cdot \log_2 n)$

Quick Sort – Descrizione dell’algoritmo

- Utilizza anch’esso il paradigma divide-and-conquer
- Approccio differente rispetto al merge sort
    - *Nel quick sort, è lo step* ***divide*** *a svolgere la maggior parte del lavoro*
- **Divide**: effettua il *partitioning* dell’array
- **Conquer**: ordina i due sotto array creati nel passo precedente
- **Combine**: combina i risultati ottenuti al passo precedente

Quick Sort – Descrizione dell’algoritmo

- **Partitioning**
  - *Parte da un elemento **pivot***
- Suddivide l’array in modo che:
  - *alla sinistra del pivot ci siano solo gli elementi minori o uguali del pivot;*
  - *alla destra del pivot ci siano solo gli elementi maggiori del pivot.*



Quick Sort – Descrizione dell'algoritmo

- Consideriamo tre diversi gruppi
  - *Il gruppo L racchiude tutti gli elementi minori o uguali al pivot*
  - *Il gruppo G racchiude tutti gli elementi maggiori del pivot*
  - *Il gruppo U racchiude gli elementi che non sono stati ancora confrontati con il pivot*
- Scegliamo come pivot l'elemento più a destra nell'array ($array[r]$)
- Usiamo due variabili di supporto, q e j
  - q rappresenta il nuovo indice del pivot
  - j rappresenta un contatore
- Ad ogni iterazione, compariamo $array[j]$ con il pivot

Quick Sort – Descrizione dell’algoritmo


```
L = {};
G = {};
U = {8, 4, 12, 7};
q = 0;
j = 0;

array[j] >= pivot
j++;
```

Quick Sort – Descrizione dell’algoritmo


```
L = {};
G = {8};
U = {4, 12, 7};
q = 0;
j = 1;

array[j] < pivot
swap;
j++;
q++;
```

Quick Sort – Descrizione dell’algoritmo

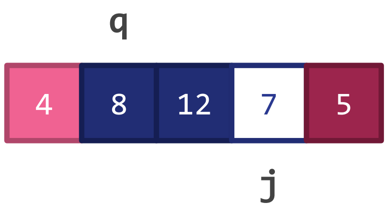
q

4	8	12	7	5

j

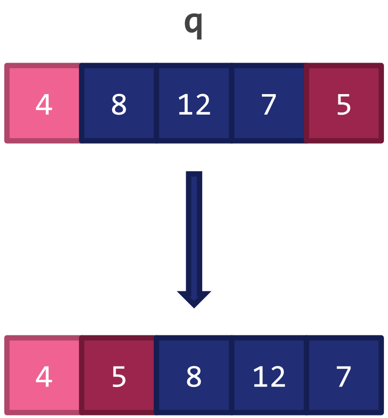
```
L = {4};
G = {8};
U = {12, 7};
q = 1;
j = 2;

array[j] >= pivot
j++;
```

Quick Sort – Descrizione dell’algoritmo




```
L = {4};
G = {8, 12};
U = {7};
q = 1;
j = 3;

array[j] < pivot
swap;
j++;
q++;
```

Quick Sort – Descrizione dell’algoritmo



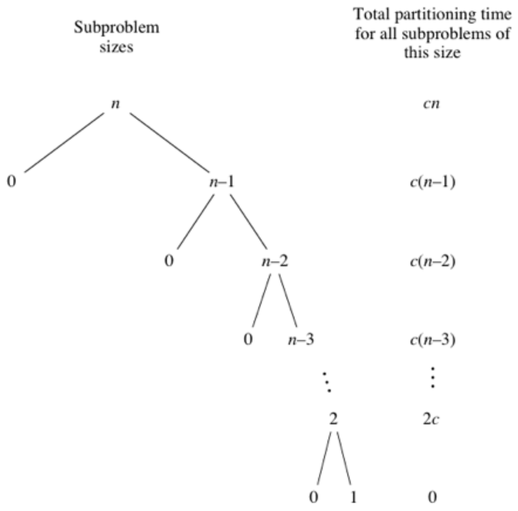

```
L = {4, 7};
G = {8, 12};
U = {};
q = 1;


swap(pivot, q);
```

Quick Sort – Analisi Computazionale

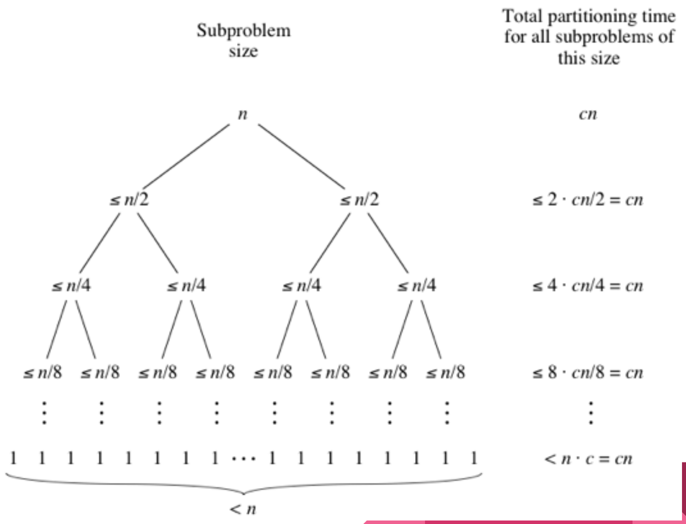
- **Divide**
  - *L'algoritmo di partizionamento è un* $O(n)$
- **Combine** è trascurabile; per **conquer** occorre fare un'analisi simile a quella del merge sort
- Vediamo cosa accade nel **caso peggiore** ed in quello **migliore**

Quick Sort – Analisi Computazionale

Caso peggiore



Caso migliore



Domande?