



UNIVERSITÀ  
DI PARMA

**algoritmi su array**  
**ricerca e ordinamento**  
*Alberto Ferrari*

- algoritmi *sequenziali*: eseguono *un* solo *passo* alla volta
- algoritmi *paralleli*: possono eseguire *più passi* per volta
- algoritmi *deterministici*: ad ogni punto di scelta, intraprendono *una sola via* determinata dalla valutazione di un'espressione
- algoritmi *probabilistici*: ad ogni punto di scelta, intraprendono *una* sola *via* determinata *a caso*
- algoritmi *non deterministici*: ad ogni punto di scelta, esplorano *tutte le vie* contemporaneamente

- dato *un problema*, possono esistere *più algoritmi* che sono *corretti* rispetto ad esso
- ... e un numero illimitato di algoritmi *errati* :(
- gli algoritmi corretti possono essere confrontati rispetto alla loro *complessità* o efficienza *computazionale*

- l'algoritmo viene *valutato* in base alle risorse utilizzate durante la sua esecuzione:
  - *tempo* di calcolo
  - *spazio* di memoria (*risorsa riusabile*)
  - *banda* trasmissiva (*risorsa riusabile*)

- *esiste sempre* un algoritmo risolutivo per un problema?



- problema *decidibile*
  - se esiste un algoritmo che produce la soluzione in tempo finito per ogni istanza dei dati di ingresso del problema
- problema *indecidibile*
  - se non esiste nessun algoritmo che produce la soluzione in tempo finito per ogni istanza dei dati di ingresso del problema

- **confronto** fra algoritmi che risolvono lo **stesso problema**
- si valuta il **tempo di esecuzione** (in numero di passi) in modo indipendente dalla tecnologia dell'esecutore
- in molti casi la complessità è legata al tipo o al **numero** dei **dati** di **input**
  - ad esempio la ricerca di un valore in un vettore ordinato dipende dalla dimensione del vettore
- il **tempo** è espresso in **funzione** della dimensione dei **dati** in ingresso  **$T(n)$**
- per confrontare le funzioni tempo ottenute per i vari algoritmi si considerano le **funzioni asintotiche**

- data la funzione **polinomiale**  $f(n)$  che rappresenta il tempo di esecuzione dell'algoritmo al variare della dimensione  $n$  dei dati di input
- la funzione asintotica **ignora** le costanti moltiplicative e i termini non dominanti al crescere di  $n$ 
  - es.
  - $f(x) = 3x^4 + 6x^2 + 10$
  - funzione asintotica =  $x^4$
  - l'**approssimazione** di una funzione con una funzione asintotica è molto utile per **semplificare i calcoli**
  - la notazione asintotica di una funzione descrive il comportamento in modo semplificato, ignorando dettagli della formula

- il tempo di esecuzione può essere calcolato in caso
  - ***pessimo***
    - dati d'ingresso che massimizzano il tempo di esecuzione
  - ***ottimo***
    - dati d'ingresso che minimizzano il tempo di esecuzione
  - ***medio***
    - somma dei tempi pesata in base alla loro probabilità

- $O(1)$  Complessità *costante*
- $O(\log n)$  Complessità *logaritmica*
- $O(n)$  Complessità *lineare*
- $O(n \cdot \log n)$  Complessità *pseudolineare*
- $O(n^2)$  Complessità *quadratica*
- $O(n^k)$  Complessità *polinomiale*
- $O(a^n)$  Complessità *esponenziale*

- **calcolo** della complessità di algoritmi *non ricorsivi*
  - vengono in pratica “**contate**” le operazioni eseguite
  - il tempo di esecuzione di un’istruzione di assegnamento che non contenga chiamate a funzioni è 1
  - il tempo di esecuzione di una chiamata ad una funzione è  $1 +$  il tempo di esecuzione della funzione
  - il tempo di esecuzione di un’istruzione di selezione è il tempo di valutazione dell’espressione + il tempo massimo fra il tempo di esecuzione del ramo then e del ramo else
  - il tempo di esecuzione di un’istruzione di ciclo è dato dal tempo di valutazione della condizione + il tempo di esecuzione del corpo del ciclo moltiplicato per il numero di volte in cui questo viene eseguito

```
int fattoriale(int n) {  
    int fatt, i;  
    fatt = 1;  
    for (i = 2; i <= n; i++)  
        fatt = fatt * i;  
    return(fatt);  
}
```

$$T(n) = 1 + (n-1)(1+1+1)+1 = 3n - 1 = O(n)$$

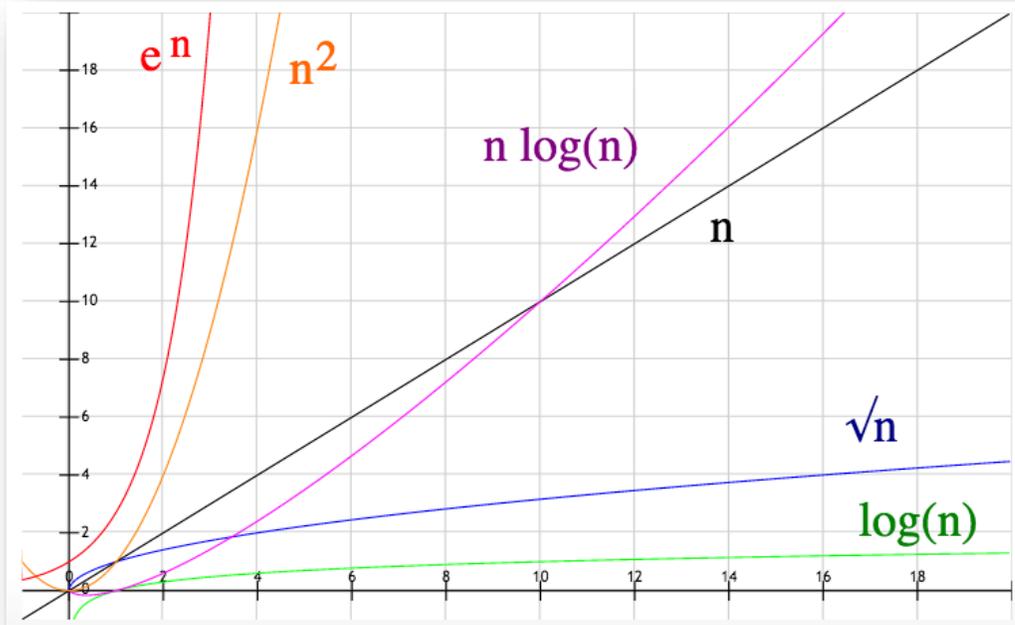
- *confrontare algoritmi corretti* che risolvono lo *stesso problema*, allo scopo di scegliere quello *migliore* in relazione a uno o più parametri di valutazione



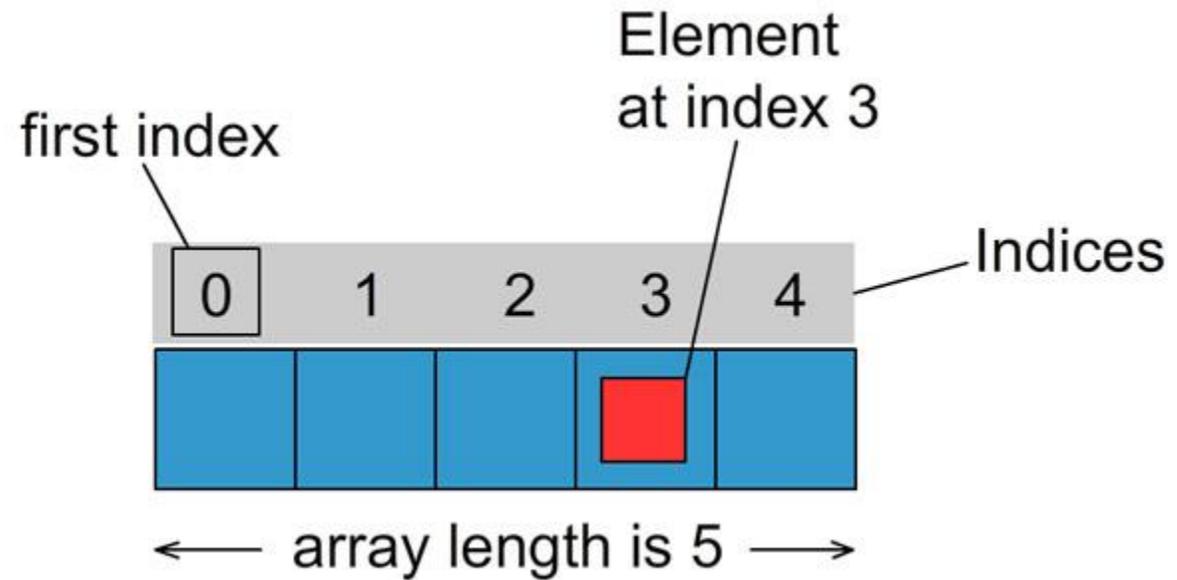
- se si ha a disposizione un solo parametro per valutare un algoritmo, per esempio il tempo d'esecuzione, è semplice la scelta: il più veloce.
- ogni altra caratteristica non viene considerata.



- nel caso di due parametri normalmente si considera
  - il *tempo*
    - numero di *passi* (istruzioni) che occorrono per produrre il risultato finale.
    - passi e non secondi o millisecondi perché il tempo varia al variare delle potenzialità del calcolatore.
  - lo *spazio*
    - occupazione di *memoria*

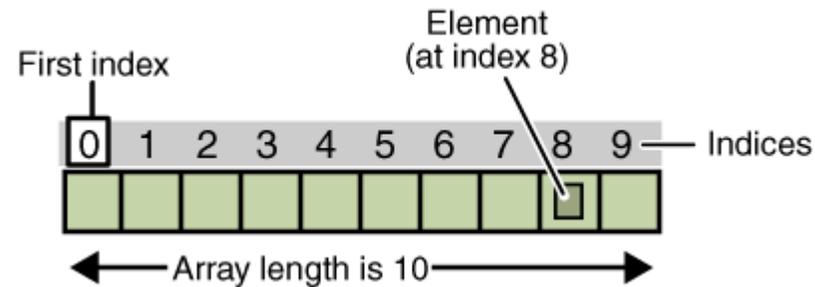


n	n/2	log(n)
10	5	3,321928
20	10	4,321928
30	15	4,906891
40	20	5,321928
50	25	5,643856
60	30	5,906891
70	35	6,129283
80	40	6,321928
90	45	6,491853
100	50	6,643856
300	150	8,228819
1000	500	9,965784
10000	5000	13,28771
100000	50000	16,60964



array

- struttura ***statica omogenea***
  - non in tutti i linguaggi ... *array dinamici*
- ***accesso diretto*** a ogni elemento attraverso l'***indice***
  - complessità dell'accesso ***O(1)***



- percorrere *una e una sola volta* tutti gli *elementi*

```
void visita_array(int a[], int n) {  
    for (int i = 0; i < n; i++)  
        elabora(a[i]);  
}
```

- se *elabora* ha complessità  $x$  passi la complessità dell'algoritmo risulta:  
 $1+n(1+x+1)+1 = (x+2)n+2 = \mathbf{O(n)}$

- stabilire se un *valore* è *presente* all'interno dell'array restituendo l'*indice* dell'elemento o *-1* se non presente
- ricerca *sequenziale* (*lineare*): algoritmo per trovare un elemento in un insieme *non ordinato*
  - si effettua la scansione dell'array *sequenzialmente*
- ricerca *binaria* (*dicotomica*): algoritmo per trovare un elemento in un insieme *ordinato*
  - si inizia la ricerca dall'*elemento centrale*, si confronta questo elemento con quello cercato:
    - se *corrisponde*, la ricerca termina con successo
    - se è *superiore*, la ricerca viene ripetuta sugli elementi *precedenti*
    - se è *inferiore*, la ricerca viene ripetuta sugli elementi *successivi*

```
int linearSearch(int array[],  
                int size, int val) {  
    for (int i=0;i<size;++i)  
        if (array[i]==val)  
            return i;  
    return -1;  
}
```

- *complessità* computazionale
  - caso *pessimo*  $O(n)$
  - caso *ottimo*  $O(1)$
  - caso *medio*  $O(n/2)$

```
int binarySearch(int array[], int size,  
                 int val) {  
    int first, last, medium;  
    first = 0;  
    last = size - 1;  
    while(first <= last) {  
        medium = (first + last) / 2;  
        if(array[medium] == val)  
            return medium; // value found  
        if(array[medium] < val)  
            first = medium + 1;  
        else  
            last = medium - 1;  
    }  
    return -1; // not found  
}
```

- *complessità* computazionale
  - caso *pessimo*  $O(\log_2 n)$
  - caso *ottimo*  $O(1)$
  - caso *medio*  $O(\log_2 n)$

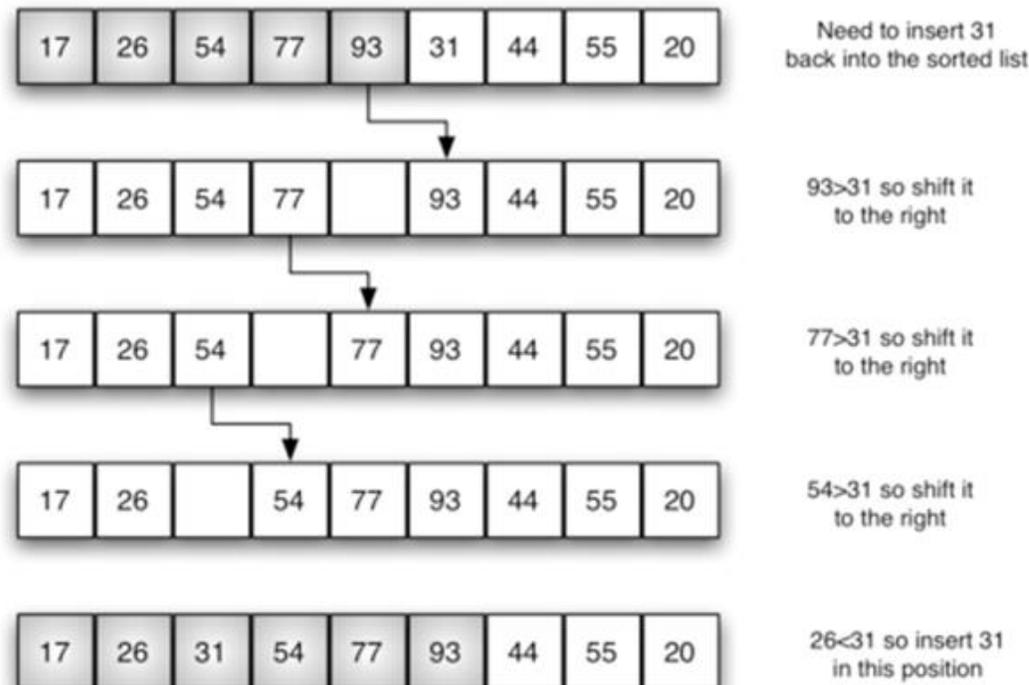
*array – algoritmi di*

# ordinamento

- l'**ordinamento** degli elementi di un array avviene considerando il valore della **chiave primaria**
- negli esempi le chiavi sono interi e la **relazione d'ordine totale** è  $\leq$
- caratteristiche degli algoritmi di ordinamento:
  - **efficienza** (complessità computazionale)
  - **stabilità**: l'algoritmo è stabile se **non altera l'ordine** relativo di elementi dell'array aventi la stessa chiave primaria
  - **sul posto**: l'algoritmo opera sul posto se la **dimensione** delle **strutture ausiliarie** di cui necessita è **indipendente dal numero di elementi** dell'array da ordinare

<https://www.toptal.com/developers/sorting-algorithms/>

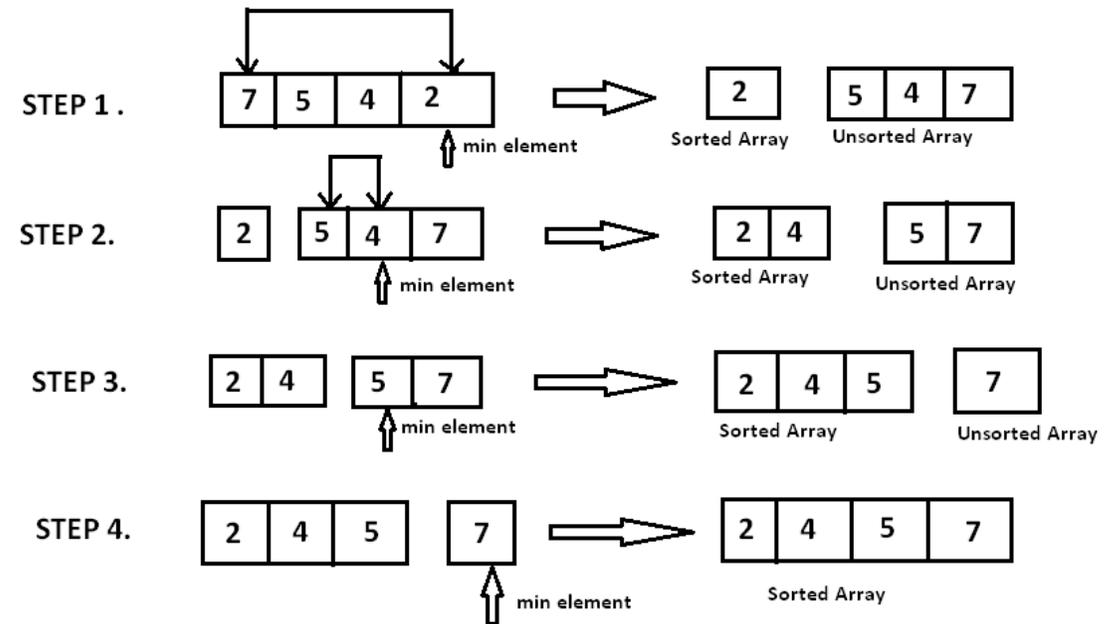
- al generico passo  $i$  l'array è considerato *diviso* in
  - una sequenza di *destinazione*  $a[0] \dots a[i - 1]$  già ordinata
  - una sequenza di *origine*  $a[i] \dots a[n - 1]$  ancora da ordinare
- l'obiettivo è di *inserire* il valore contenuto in  $a[i]$  al *posto giusto* nella sequenza di destinazione facendolo scivolare a ritroso, in modo da *ridurre* la sequenza di origine di un elemento



```
void insertsort(int array[], int size) {  
    int i, j, app;  
    for (i=1; i<size; i++) {  
        app = array[i];  
        j = i-1;  
        while (j>=0 && array[j]>app) {  
            array[j+1] = array[j];  
            j--;  
        }  
        array[j+1] = app;  
    }  
    return;  
}
```

- *sul posto*
- *stabile*
- *complessità  $O(n^2)$*
- efficiente su array già *parzialmente ordinati*

- al generico passo  $i$  vede l'array diviso in:
  - una sequenza di *destinazione*  $a[0] \dots a[i - 1]$  già *ordinata*
  - una sequenza di *origine*  $a[i] \dots a[n - 1]$  da ordinare
- l'*obiettivo* è *scambiare* il valore *minimo* della seconda sequenza *con* il valore contenuto in  $a[i]$  in modo da ridurre la sequenza di origine di un elemento

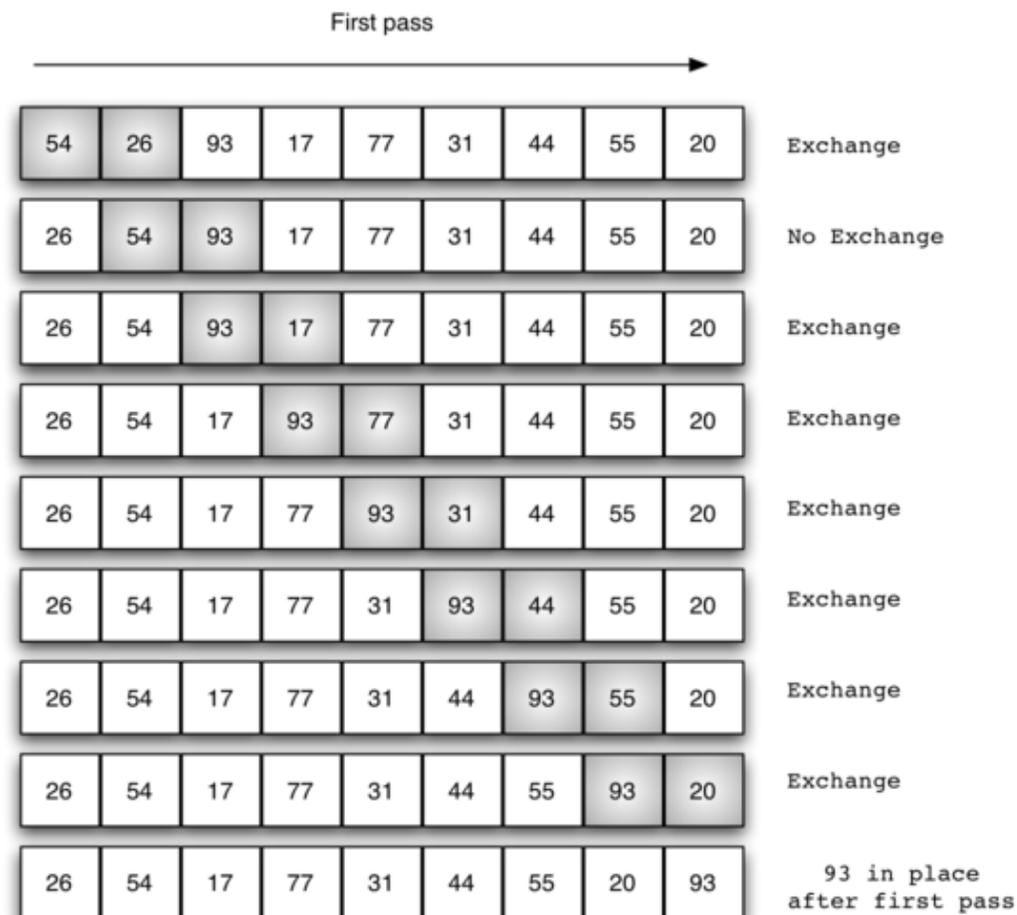


```
void selectsort(int array[],int size){  
    int i,j,min;  
    for (i=0; i<size; i++) {  
        min = i;  
        for (j=i+1; j<size; j++)  
            if (array[min]> array[j])  
                min = j;  
        if (min != i)  
            swap(array[i],array[min]);  
    }  
}
```

- *sul posto*
- *non stabile*
- *complessità  $O(n^2)$*

```
void swap(int &x, int &y) {  
    int temp = x;  
    x = y;  
    y = temp;  
}
```

- al generico passo  $i$  vede l'array diviso in:
  - una sequenza di *destinazione*  $a[0] \dots a[i - 1]$  già *ordinata*
  - una sequenza di *origine*  $a[i] \dots a[n - 1]$  da *ordinare*
- l'obiettivo è di far emergere il valore *minimo* della sequenza di origine confrontando e *scambiando sistematicamente* i valori di elementi adiacenti a partire dalla fine dell'array, in modo da ridurre la sequenza di origine di un elemento
  - sul posto
  - stabile
  - complessità  $O(n^2)$
  - efficiente per array parzialmente ordinati (*versione ottimizzata*)



- versione semplificata

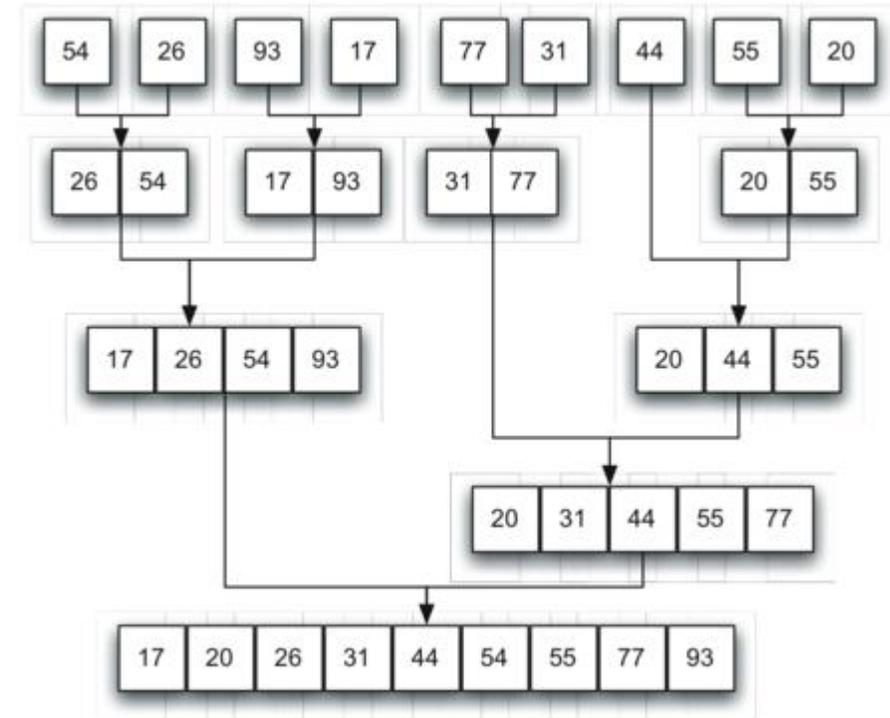
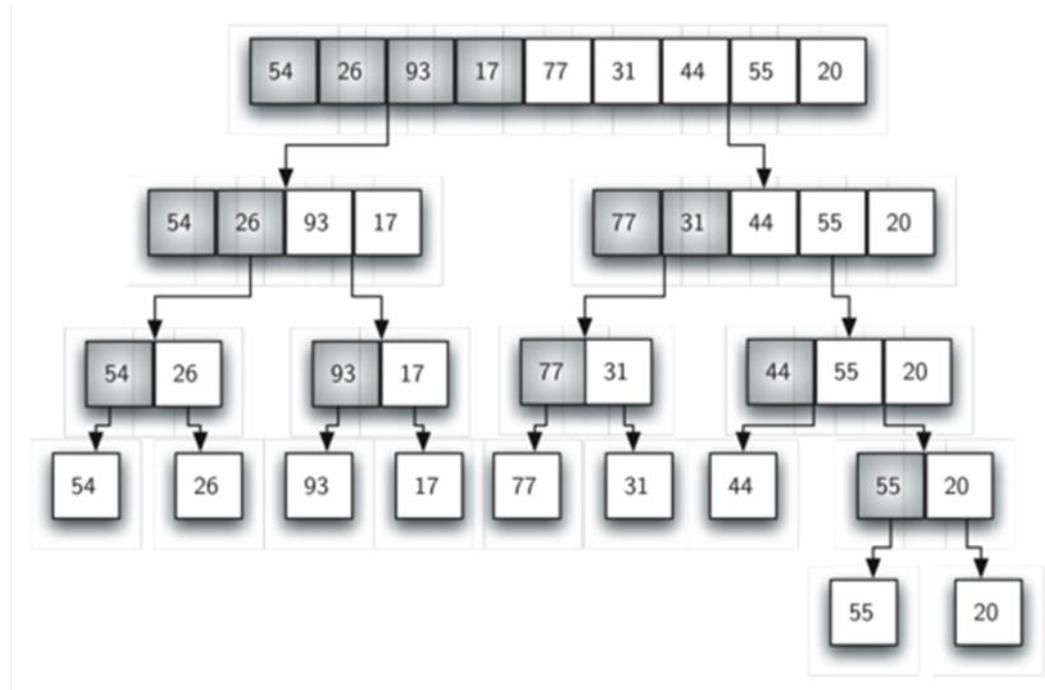
```
void bubble_sort(int array[], int size) {
    int i, last;
    for (last = size - 1; last > 0; last--) {
        for (i=0; i<last; i++){
            if (array[i]>array[i+1])
                swap(array[i], array[i+1]);
        }
    }
}
```

- se *non* avvengono *scambi*  
l'array risulta *ordinato*

```
void bubble_sort(int array[], int size) {
    int i, last;
    bool swapped;
    for (last = size - 1; last > 0; last--) {
        swapped = false;
        for (i=0; i<last; i++){
            if (array[i]>array[i+1])
                swap = true;
                swap(array[i], array[i+1]);
        }
        if (!swapped)
            return;
    }
}
```

- algoritmo *ricorsivo*
- sfrutta la tecnica del *divide et impera*
  - suddivisione del problema in *sottoproblemi* della stessa natura di *dimensione* via via *più piccola*
- *non* opera *sul posto*: nella fusione usa un array di appoggio il cui numero di elementi è proporzionale al numero di elementi dell'array da ordinare
- *stabile*
- complessità  $O(n \log(n))$

- se la sequenza da ordinare ha *lunghezza 0* o *1*, è *ordinata*
- altrimenti la sequenza viene divisa (*divide*) in due *metà* (*se numero dispari di elementi la prima ha un elemento in più della seconda*)
- ogni *sottosequenza* viene *ordinata*, applicando *ricorsivamente* l'algoritmo (*impera*)
- le due *sottosequenze ordinate* vengono *fuse* (*combina*)
- si estrae ripetutamente il *minimo* delle due sottosequenze e lo si pone nella *sequenza in uscita*, che risulterà *ordinata*



```
void merge_sort(int array[],
                int left, int right) {
    int center; // middle index
    if(left<right) {
        center = (left+right)/2;
        //sort first half
        merge_sort(array, left, center);
        //sort second half
        merge_sort(array, center+1, right);
        //merge sorted arrays
        merge(array, left, center, right); }
}
```

```
void merge(int array[], int left,
           int center, int right){
    int i = left;      //index first array
    int j = center+1; //index second array
    int k = 0;        //index new temporary array
    int temp[DIM_ARRAY]; //temporary array
    while ((i<=center) && (j<=right)) {
        if (array[i] <= array[j]) {
            temp[k] = array[i]; i++; }
        else { temp[k] = array[j]; j++; }
        k++;
    }
    while (i<=center) {
        temp[k] = array[i]; i++; k++; }
    while (j<=right) {
        temp[k] = array[j]; j++; k++; }
    for (k=left; k<=right; k++){
        array[k] = temp[k-left]; }
}
```