

Ricerca  $(3, v)$



UNIVERSITÀ  
degli STUDI  
di CATANIA

# Ricerca, ordinamento e ricorsione

Corso di programmazione I AA 2019/20

Corso di Laurea Triennale in Informatica

---

Prof. Giovanni Maria Farinella

Web: <http://www.dmi.unict.it/farinella>

Email: [gfarinella@dmf.unict.it](mailto:gfarinella@dmf.unict.it)

Dipartimento di Matematica e Informatica

## Ricerca sequenziale di un elemento in un array.

Data una struttura dati lineare (esempio: array), ricercare un elemento X tra le componenti dello array.

Nella sua forma più semplice: (ricerca tutte le occorrenze del numero nella struttura lineare).

```
1  int A[DIM];  
2  //... bool trovato = false;  
3  while (i++ < DIM && !Trovato)  
4  {  
5      if (A[i] == numero) {  
6          cout << "Found at index " << i << endl;  
7      }  
      trovato = true;
```

## Esempi Svolti

32\_01.cpp – Ricerca lineare di **tutti** gli **elementi**

32\_02.cpp – Ricerca lineare, si ferma alla **prima** **occorrenza**

32\_03.cpp – Ricerca lineare con **sentinella**

32\_04.cpp – Ricerca lineare in **array** **ordinato**

32\_05.cpp – Ricerca lineare del **massimo** **valore**

$a$ 

1	4	7	9	
---	---	---	---	--

$a[i] < a[i+1]$

7 1 9 4

## Ricerca dicotomica



**Dicotomia** == divisione in due parti.

Sia A un **array ordinato** (in modo crescente o decrescente).

Si confronta la **chiave** di ricerca con l'elemento centrale (M) dello array :

- se sono uguali, l'elemento è stato trovato;
- se la chiave di ricerca è minore di M, la ricerca prosegue iterativamente nella **prima metà di A**;
- se la chiave di ricerca è maggiore di M, la ricerca prosegue nella **seconda metà di A**;

Quindi: la ricerca dicotomica opera, ad ogni iterazione, su un array che è **la metà di quello precedente**.

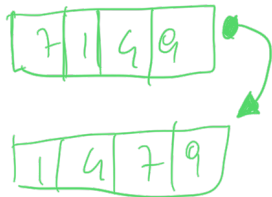
- NB: condizione necessaria è che l'array sia ordinato.

### Esempi Svolti

32\_06.cpp – Ricerca dicotomica.

Il problema dell'ordinamento consiste nel sistemare gli elementi dello array in un preciso ordine. Ad esempio:

- ordinamento crescente;
- ordinamento decrescente.




Algoritmi di ordinamento:

---

- **Bubblesort**
- **SelectionSort**
- **InsertionSort**
- QuickSort
- MergeSort
- HeapSort
- ShellSort

Si chiama così perchè gli elementi dello array raggiungono la giusta posizione nell'ordinamento finale come fossero *bolle* che salgono in superficie. **Ordinamento crescente:**

- si effettua un certo numero di **visite dell'intero array**.
- Ad ogni visita si **confrontano coppie di elementi** contigui:
  - se il primo valore è maggiore del secondo, essi vengono **scambiati**;
- Se durante una visita **non avviene alcuno scambio**, ciò significa che **l'array è ordinato**, dunque lo algoritmo può terminare.



# BubbleSort

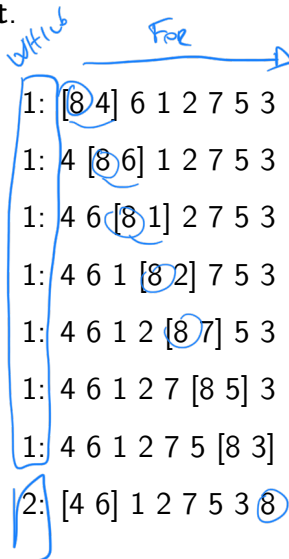
Implementazione del BubbleSort.

Flag *swapped* utile a capire se vanno fatti altri cicli.

```
1    bool swapped = true;
2    while(swapped){
3        swapped = false;
4    for(int i = 0; i < length - 1; i++ )
5        if ( array[i] > array[i+1] ){
6            swap( array, i, i+1 );
7            swapped = true;
8        }
9    }
```

# BubbleSort

Esempio di output.



# BubbleSort

In pratica, nel BubbleSort, **dopo la k-esima visita dello array**, il k-esimo elemento più grande trova il giusto posto nell'ordinamento finale.

1: [8 4] 6 1 2 7 5 3

1: 4 [8 6] 1 2 7 5 3

1: 4 6 [8 1] 2 7 5 3

...

2: [4 6] 1 2 7 5 3 8

**Worst case:**  $(n - 1)$  ·  $(n - 1)$  iterazioni.

Esempi svolti

32\_07.cpp – Bubblesort.

Implementare una versione ottimizzata del BubbleSort sulla base della precedente osservazione:

*dopo la  $k$ -esima visita dello array, il  $k$ -esimo elemento più grande trova il giusto posto nello ordinamento finale.*

- ricerca il minimo dello intero array ( $A[0 \dots n-1]$ )
- scambia il minimo con l'elemento di posto zero;
- ricerca il minimo nel sottoarray  $A[1 \dots n-1]$ ;
- scambia il minimo con l'elemento di posto 1;
- ...

## SelectionSort



```
1  for (int index = 0; index < length - 1; index++)
2  {
3      //selezione o ricerca del minimo
4      min = index;
5      for (int i = index + 1; i < length; i++)
6          if (array[i] < array[min])
7              min = i;
8
9      //scambia minimo ed elemento di indice index
10     swap(array, min, index);
11 }
```

Esempio di output.

[8] 4 6 [1] 2 7 5 3

1 [4] 6 8 [2] 7 5 3

1 2 [6] 8 4 7 5 [3]

1 2 3 [8] [4] 7 5 6

1 2 3 4 [8] 7 [5] 6

1 2 3 4 5 [7] 8 [6]

1 2 3 4 5 6 [8] [7]



## Esempi Svolti

32\_09.cpp – SelectionSort

Si basa sull'inserimento di ogni elemento in un **sottoarray ordinato**. Descrizione (informale) dello algoritmo:

- si consideri il **primo elemento** dello array. Esso rappresenta un sottoarray di lunghezza 1;
- si **inserisca** il **secondo elemento** al posto giusto nel sottoarray ordinato di lunghezza 1:
  - se questo è minore del primo e unico elemento, quest'ultimo si sposterà a destra;
- si **inserisca** al posto giusto il **terzo elemento**, spostando li elementi del sottoarray, se necessario, per mantenere l'ordinamento;
- ...

# InsertionSort

```
1  for (int index = 1; index < length; index++){
2      int key = array[index];
3      int position = index;
4
5      // shift valori piu' grandi di key a destra
6      while (position > 0 && array[position - 1] > key){
7          //shift a destra
8          array[position] = array[position - 1];
9          position --;
10     }
11     array[position] = key; //inserimento
12 }
```



11(64)

Esempio di output.

[8] 4 6 1 2 7 5 3

[4 8] 6 1 2 7 5 3

[4 6 8] 1 2 7 5 3

[1 4 6 8] 2 7 5 3

[1 2 4 6 8] 7 5 3

[1 2 4 6 7 8] 5 3

[1 2 4 5 6 7 8] 3

Esempi Svolti

32\_10.cpp – InsertionSort

**L'efficienza di un algoritmo di sorting** può essere stabilita **analiticamente** contando il numero di confronti, **in funzione della dimensione  $n$  dello input**.

Gli algoritmi di ordinamento {Bubble, Selection, Insertion} Sort operano in modo simile:

- ciclo esterno con numero di iterazioni approssimativamente uguale alla lunghezza dello array;
- ciclo interno che scandisce approssimativamente tutti gli elementi dello array;
- di conseguenza, **circa  $n^2$  confronti**

Una funzione si dice ricorsiva quando è definita in termini di se stessa, ovvero quando nel corpo della funzione sono presenti **chiamate alla funzione stessa** (funzioni ricorsive):

- uno o più **casi base** (condizioni di terminazione)
- un passo ricorsivo, che si basa su una o più **chiamate alla funzione stessa** ma su un **input “ridotto”**

Esempio di funzione matematica definita mediante **induzione** anche detta **relazione di ricorrenza**.

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 1$$

dunque:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot (n - 1)! & \text{se } n > 0 \end{cases}$$



ESEMPIO: Funzione ricorsiva per il calcolo del fattoriale.

```
1  long fattoriale(int n){  
2      if(n == 0)  
3          return 1;  
4      return  
5          n * fattoriale(n-1); //chiamata ricorsiva  
6  }
```

Caso base o condizione di **terminazione**:  $n=0$ ;

Chiamata ricorsiva con input ridotto ( $n-1$ );

Dato un certo problema, una soluzione che faccia uso di una funzione **ricorsiva**:

(+) è intuitivamente **più semplice** da concepire;

(+) **minor numero di linee di codice**;

(-) consuma **molta memoria** rispetto ad una soluzione iterativa;

(-) consuma **molto tempo** rispetto ad una soluzione iterativa;

Il **consumo di memoria** è dovuto alla allocazione dei record di attivazione dello stack dovuti alla **sequenza di chiamate ricorsive**, una dopo l'altra.

Il **consumo di tempo** è dovuto all'allocazione dei record sullo stack, la copia dell'indirizzo di ritorno e delle variabili locali.

Qualsiasi funzione ricorsiva si può sempre esprimere in forma non ricorsiva.

- Ricorsione **di coda**: la chiamata ricorsiva è l'ultima azione della funzione ricorsiva.
- Ricorsione **non di coda**: la ricorsione può essere eliminata con l'ausilio di uno **stack esterno**.

Anche le funzioni con una doppia chiamata ricorsiva **non si possono riscrivere** con una banale iterazione.

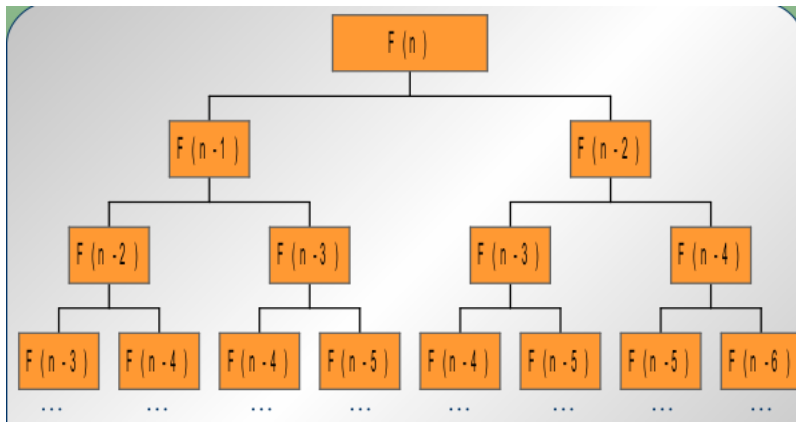
## Numeri di Fibonacci

$$F(0) = F(1) = 1;$$

$$F(n) = F(n-1) + F(n-2);$$

```
1  int fibo(int n){  
2      if(n<=1)  
3          return 1;  
4      else  
5          return fibo(n-1) + fibo(n-2);  
6  }
```

**Inefficienza.** (chiamate duplicate!)



## Esempi Svolti

32\_11.cpp – Fattoriale.

32\_12.cpp – SelectionSort ricorsivo.

32\_13.cpp – Ricerca dicotomica ricorsiva.

32\_14.cpp – Numeri di Fibonacci.

Codificare in C++ la seguente funzione definita in modo ricorsivo.

$$A(n) = \begin{cases} 2 \cdot A(n-1) \cdot A(n-2), & \text{se } n \text{ è pari} \\ 3 \cdot A(n-2) + A(n-1) & \text{altrimenti} \end{cases}$$



Codificare una funzione non ricorsiva per il calcolo del fattoriale di un numero intero.

FINE