PIEMONTE
FONDI STRUTTURALI E DI INVESTIMENTO EUROPEI 2014/2020
fondo sociale europeo
ITS TECNOLOGIE DELL'INFORMAZIONE E DELLA COMUNICAZIONE

FONDAMENTI DI PROGRAMMAZIONE

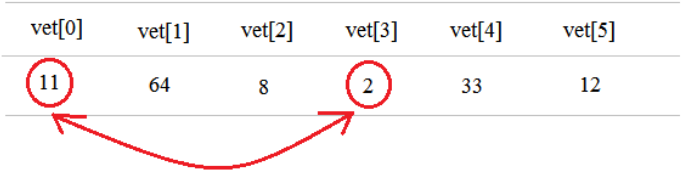
C / C++

Docente: Armando Valentino

in collaborazione con:
ISTITUTI TECNICI SUPERIORI
Ministero dell'Istruzione
UNIONE EUROPEA
REGIONE PIEMONTE
per una crescita intelligente, sostenibile ed inclusiva
www.regione.piemonte.it/europa2020
INIZIATIVA CO-FINANZIATA CON FSE

LINGUAGGIO C/C++

Ricerca e Ordinamento



Prof. Armando Valentino

POR Piemonte
FSE 2014-2020

Ordinamento Insertion Sort

Insertion sort => ordinamento per Inserzione

È il più semplice in assoluto. Si confronta l'elemento nella prima posizione con tutti gli elementi nelle posizioni successive effettuando uno scambio se, fra i due elementi confrontati, il primo è maggiore del secondo.
Alla fine del ciclo dei confronti l'elemento nella prima posizione risulta sicuramente il minore di tutti gli elementi nelle posizioni successive.
Successivamente si ripete il procedimento per l'elemento nella seconda posizione, nella terza e così via fino all'ultimo elemento.

n° di scambi:
(caso migliore) **0**
(caso peggiore) **N*(N – 1)/2**

55	-2	34	10	0	2	-5	12

Video => https://youtu.be/ROalU379l3U

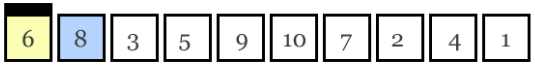
Ordinamento Insertion Sort

```
void ord_insertion(int arr[],int dim){
        for (int i = 0; i<dim-1; i++) {
            for (int j=i+1; j<dim ; j++) {
                if (arr[i]>arr[j]) {
                    int temp=arr[i];
                    arr[i]=arr[j];
                    arr[j]=temp;
                }
            }
        }
}
```

Ordinamento Selection sort

Selection Sort => ordinamento per Selezione

è utile quando **l'insieme da ordinare è composto da pochi elementi** e quindi un algoritmo non molto efficiente.
L'idea di fondo su cui si basa l'algoritmo è quella di ripetere per N volte una procedura in grado di **selezionare alla i-esima iterazione l'elemento più piccolo nell'insieme e di scambiarlo con l'elemento** che in quel momento occupa la **posizione i che fa da perno**.



Yellow is smallest number found
Blue is current item
Green is sorted list

In altre parole:

1. alla prima iterazione verrà selezionato l'elemento più piccolo dell'intero insieme {v1,v2,v3,...,vN} e sarà scambiato con quello che occupa la prima posizione;
2. alla seconda iterazione è selezionato il secondo elemento più piccolo dell'insieme, ossia l'elemento più piccolo dell'insieme "ridotto" costituito dagli elementi {v2,v3,...,vN} ed è scambiato con l'elemento che occupa la seconda posizione;
3. si ripete fino ad aver collocato nella posizione corretta tutti gli N elementi

n° di scambi:
(caso migliore) **0** (caso peggiore) **N – 1**

Ordinamento Selection sort

```
void SelectionSort (int  V[ ],  int N) {
    int temp;
    for (int i=0; i<N-1;  i++){          // i = indice elemento perno
                                         // cerca il più piccolo elemento  dalla posizione  i  alla posizione  N-1
        int jmin = i ;                   // jmin  indice dell'elemento minore
        for (int j=i+1;  j<N;  j++) {
            if (V[j] < V[jmin] ) {       // se  l'elemento minore è in posizione j,  assegna  j a jmin
                jmin = j ;
            }
        }
        if (i != jmin){                  //  se  (i != jmin) è stato trovato un elemento minore in posizione jmin
            temp=V[jmin];                // SCAMBIO elemento in posizione i con elemento in posizione jmin
            V[jmin]=V[i];
            V[i]=temp;
        }
    }
}
```

Ordinamento Bubble sort

Bubble Sort => ordinamento a Bolle

Si basa sull'idea di **far emergere pian piano gli elementi più piccoli verso l'inizio** dell'insieme da ordinare facendo **sprofondare gli elementi maggiori verso il fondo**: un po' **come le bollicine** in un bicchiere di acqua gassata da qui il nome di ordinamento a bolle.

La strategia adottata è quella di scorrere più volte la sequenza da ordinare, **verificando ad ogni passo l'ordinamento reciproco degli elementi contigui, v_i e v_{i+1}** ed eventualmente **scambiando di posizione quelle coppie di elementi non ordinati**

Si devono fare più cicli di scansione degli elementi scambiando gli elementi a due a due e questo deve continuare fino a quando tutti gli elementi sono ordinati. Controllando gli scambi, si avrà l'ordinamento completato quando non ci sono più scambi.

n° di scambi:	
(caso migliore)	**0**
(caso peggiore)	**N*(N – 1)/2**

Video: http://www.youtube.com/watch?v=gWkvvsJHbwY&feature=player_detailpage

Ordinamento Bubble sort

```
void BubbleSort (int V[], int N)  {                    // ordinamento Crescente
        // la variabile Scambio indica se durante una scansione si è verificato almeno uno scambio
        bool Scambio=true;
        int Temp;
        int Ultimo;
        Ultimo=N-1;
        // il while ripete  fintantoché c'è stato almeno uno scambio durante l'ultima scansione
        while (Scambio==true)  {
            // inizio nuova scansione
            Scambio=false;
            for (int i=0; i<Ultimo; i++)  {
                if (V[i]>V[i+1]) {          // scambia e imposta la variabile booleana a true
                    Temp=V[i];
                    V[i]=V[i+1];
                    V[i+1]=Temp;
                    Scambio=true;
                }
            }
        }
}
```

Ricerca in un Array

Problema:

Dati:

1. Una sequenza A di elementi,
2. un elemento (chiave) x

Verificare se x fa parte della sequenza oppure x non fa parte della sequenza stessa.

Esempio

- A = | 7 | 2 | 4 | 5 | 3 | 1 | 5 | 6 |
- x = 5 → x∈A ?? SI
- x = 9 → x∈A ?? NO

Ricerca SEQUENZIALE (o lineare)

L'algoritmo di ricerca SEQUENZIALE (lineare) **è basato sulla seguente strategia:**

– Gli elementi dell'array vengono analizzati in sequenza, confrontandoli con la chiave per determinare se almeno uno degli elementi è uguale alla chiave.

– Quando si trova un elemento uguale alla chiave la ricerca termina.

Ricerca SEQUENZIALE

```
#include <iostream>
using namespace std;
int main()
{
      int dim=8, k;
      int arr[dim]={2,5,3,8,4,1,6,7};
      cout << "che numero vuoi cercare? ";
      cin >> k;
      for (int i=0; i < dim; i++ ){
         if(arr[i]==k){
            cout << k << "è nella posizione: "<< i+1;
            break;
         }
      }
}
return 0;
}
```

Considerazioni relative al costo

- L'algoritmo di ricerca lineare richiede che al più tutti gli elementi dell'array vengano confrontati con la chiave. Questo è necessario perché la sequenza non è ordinata.

- Considerando il confronto(A[i]==x) come operazione predominante il costo della ricerca è pari a

```
1 se x==A[0]  se si trova in prima posizione
N =A.length  se si trova in ultima posizione
N=A.length se non viene trovato
```

In generale 1<=costo<=numero elementi array

Ricerca BINARIA (o DICOTOMICA)

Si applica ad un array già ordinato.
Sfrutta ordinamento per ridurre il numero di confronti nella ricerca; è un algoritmo molto efficiente per la ricerca di un elemento in un insieme ordinato

Si confronta la chiave *k* con l'elemento in posizione centrale dell'array e si continua a cercare nel semi-array inferiore (dalla posizione 0 alla posizione dell'elemento di mezzo-1) o superiore (dalla posizione dell'elemento di mezzo + 1 alla posizione n-1) a seconda che il valore *k* sia più piccolo o più grande dell'elemento di mezzo. Il procedimento continua iterativamente confrontando la chiave con l'elemento di mezzo del semi-array inferiore o superiore via via individuato.

La ricerca termina per *elemento trovato (ricerca con successo)* o per *sotto-array vuoto, cioè inizio>fine (ricerca senza successo).*

Ricerca BINARIA

```
int ricbin(int vet[],int dim,int k){
        int inizio, fine, medio,pos;
        bool trovato;
        trovato=false;
        inizio=0;
        fine=dim-1;
        while (inizio<=fine && trovato==false) {
                medio=(inizio+fine)/2;
                cout << medio<<endl;
                if (vet[medio] > k) {
                        fine=medio-1;
                }
                if (vet[medio] < k) {
                        inizio=medio+1;
                }
                if (vet[medio]==k){
                        trovato =true;
                        pos=medio;
                }
        }
        if(trovato==true){
                        return pos+1;
        }else{
                        return -1;
        }
}
```

Considerazioni relative al costo

•**Caso migliore:** l'elemento ricercato si trova nella posizione centrale dell'array
Costo = 1

•**Caso medio:** verranno effettuati circa la metà dei confronti del caso peggiore.
Costo = $(log_2 N)/2$;

• **Caso peggiore:** l'elemento ricercato non è presente nell'array.
Poiché ad ogni iterazione la dimensione degli elementi analizzati viene dimezzata, verrano effettuati $log_2 N$ confronti;
Costo = $log_2 N$

Licenza

Quest'opera è distribuita con Licenza Creative Commons Attribuzione - Non commerciale - Non opere derivate 4.0 Internazionale.