ALGORITMI DI ORDINAMENTO

Scopo: ***ordinare una sequenza di elementi*** **in base a una certa** ***relazione d'ordine***

- **lo scopo finale è ben definito**
  → ***algoritmi equivalenti***
- **diversi algoritmi possono avere** ***efficienza assai diversa***

- **Ipotesi:**
  gli elementi siano memorizzati in un array

ALGORITMI DI ORDINAMENTO

Principali algoritmi di ordinamento:

- ***naïve sort*** (semplice, intuitivo, poco efficiente)
- ***bubble sort*** (semplice, un po' più efficiente)
- ***insert sort*** (intuitivo, abbastanza efficiente)
- ***quick sort*** (non intuitivo, alquanto efficiente)
- ***merge sort*** (non intuitivo, molto efficiente)

Per "misurare le prestazioni" di un algoritmo, conteremo quante volte viene svolto il ***confronto fra elementi dell'array***

==> Studio della Complessità: non lo tratteremo formalmente

NAÏVE SORT

Molto intuitivo e semplice, è il primo che viene in mente

Specifica (sia **n** la dimensione dell'array **v**)

while (*<array non vuoto>***) {**

<trova la posizione ***p*** *del massimo>*

if (p<n-1) *<scambia* ***v[n-1]*** *e* ***v[p]*** *>*

/* invariante: v[n-1] contiene il massimo */

<restringi l'attenzione alle prime ***n-1*** *caselle dell'array, ponendo* ***n' = n-1*** *>*

}

NAÏVE SORT

Codifica

```
void naiveSort(int v[], int n){
 int p;
 while (n>1) {
   p = trovaPosMax(v,n);
   if (p<n-1) scambia(&v[p],&v[n-1]);
   n--;
 }
}
```

La dimensione dell'array cala di 1 a ogni iterazione

NAÏVE SORT

Codifica

```
int trovaPosMax(int v[], int n){
  int i, posMax=0;
  for (i=1; i<n; i++)
    if (v[posMax]<v[i]) posMax=i;
  return posMax;
}
```

All'inizio si assume v[0] come max di tentativo

Si scandisce l'array e, se si trova un elemento maggiore del max attuale, lo si assume come nuovo max, memorizzandone la posizione

NAÏVE SORT

Valutazione di complessità

- Il numero di *confronti* necessari vale *sempre*:

  (N-1) + (N-2) + (N-3) + … + 2 + 1 =
  = N*(N-1)/2 = proporzionale a $N^2/2$

- *Nel caso peggiore*, questo è anche il numero di scambi necessari (in generale saranno meno)
- ***Importante*: la complessità non dipende dai particolari dati di ingresso**
  - **l'algoritmo fa gli stessi confronti sia per un array disordinato, sia per un array *già ordinato***

BUBBLE SORT (ordinamento a bolle)

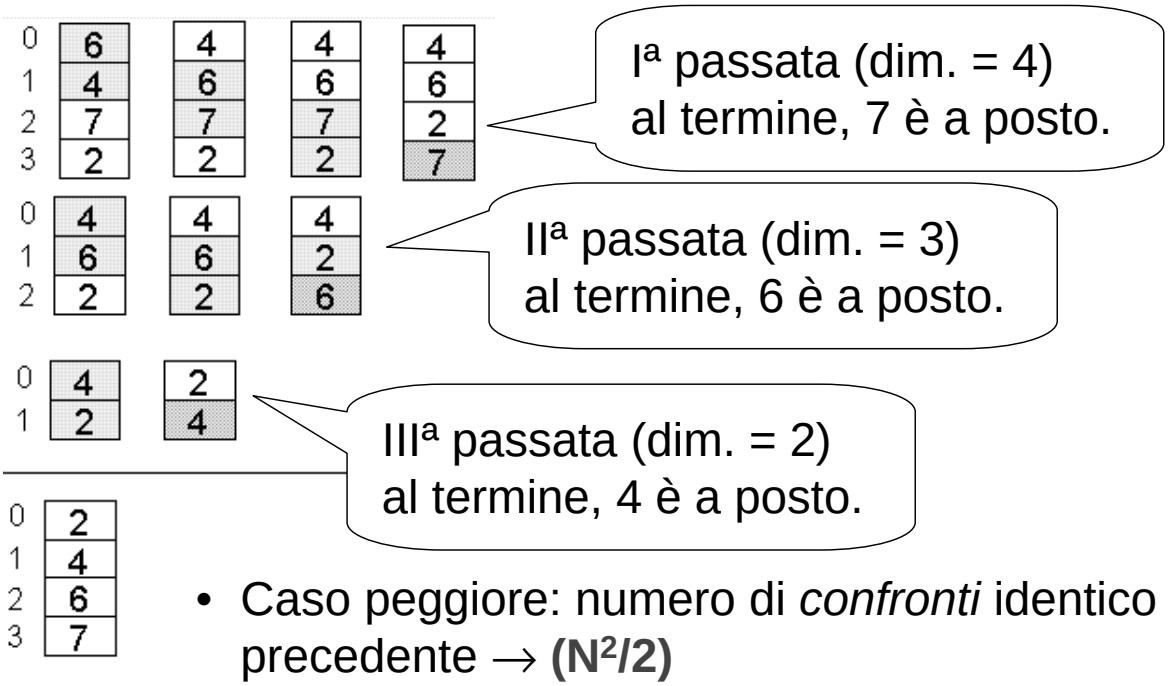
- **Corregge il difetto principale del naïve sort: quello di *non accorgersi se l'array, a un certo punto, è già ordinato***
- **Opera per *"passate successive"* sull'array:**
  - **a ogni iterazione, considera una ad una *tutte le possibili coppie di elementi adiacenti*, scambiandoli se risultano nell'ordine errato**
  - **così, dopo ogni iterazione, l'elemento massimo è in fondo alla parte di array considerata**
- **Quando non si verificano scambi, l'array è ordinato, e l'algoritmo termina**

BUBBLE SORT

Codifica

```
void bubbleSort(int v[], int n){
  int i; ordinato = 0;
  while (n>1 && !ordinato){
    ordinato = 1;
    for (i=0; i<n-1; i++)
      if (v[i]>v[i+1]) {
        scambia(&v[i],&v[i+1]);
        ordinato = 0; }
    n--;
  }
}
```

BUBBLE SORT



- Caso peggiore: numero di *confronti* identico al precedente → **($N^2/2$)**
- ***Nel caso migliore, però, basta una sola passata***, con N-1 confronti → **(N)**

ORDINARE ARRAY DI TIPI COMPLESSI

Finora abbiamo considerato sempre array *di interi*, ai quali sono applicabili gli operatori

- uguaglianza ==
- disuguaglianza !=
- minore <
- maggiore >
- minore o uguale <=
- maggiore o uguale >=

- **Nella realtà però accade spesso di trattare strutture dati di tipi *non primitivi***

ORDINARE ARRAY DI TIPI COMPLESSI

Ad esempio, un *array di `persona`*:

```
typedef struct {
  char nome[20], cognome[20];
  int  annoDiNascita;
} persona;
```

A una `persona` non si possono applicare gli operatori predefiniti. Come generalizzare gli algoritmi di ordinamento a casi come questo?

ORDINARE ARRAY DI TIPI COMPLESSI

Per generalizzare gli algoritmi di ordinamento a casi come questo, occorre:

- **eliminare dagli algoritmi *ogni occorrenza degli operatori relazionali predefiniti* (==, >, etc.)**
- **sostituirli con *chiamate a funzioni da noi definite* che svolgano il confronto *nel modo specifico del tipo da trattare***

Così, ad esempio, potremmo avere:

- uguaglianza `boolean isEqual(...)`
- minore `boolean isLess(...)`
- `...`

GENERALIZZARE GLI ALGORITMI

Una soluzione semplice e pratica consiste nell' **impostare tutti gli algoritmi in modo che operino su *array di* `element`**

Il tipo `element` sarà poi da noi definito caso per caso:

- nel caso banale,

```
element = int
element = float
…
```

- in generale,

```
element = persona
element = ...
```

GENERALIZZARE GLI ALGORITMI

Ad esempio, il bubble sort diventa:

```
void bubbleSort(element v[], int n){
 int i; element t;
 int ordinato = 0;
 while (n>1 && !ordinato){
  ordinato = 1;
  for (i=0; i<n-1; i++)
  if (isLess(v[i+1],v[i])) {
   t=v[i]; v[i]=v[i+1]; v[i+1]=t;
   ordinato = 0; }
  n--;
 }
}
```

La procedura scambia(...) è stata eliminata perché *non è generica*

IL TIPO `element`

Per definire `element` si usa la struttura:

- **element.h**
  - **fornirà la *definizione del tipo* `element`** (adeguatamente protetta dalle inclusioni multiple)
  - **conterrà le *dichiarazioni* degli operatori**
- **element.c**
  - **includerà element.h**
  - **conterrà le *definizioni* degli operatori**
- ***il cliente (algoritmo di ordinamento)***
  - **includerà element.h**
  - **userà il tipo element per operare**

IL TIPO `element`

element.h

```
typedef .... element;
#include "boolean.h"
boolean isEqual(element, element);
boolean isLess(element, element);
void printElement(element);
void readElement(element*);
```

`element` usa `boolean`

element.c

```
#include "element.h"
boolean isEqual(element e1, element e2){ ...
}
boolean isLess(element e1, element e2){ ...
}
...
```

ESEMPIO element=persona

Ipotesi: persona è definita in persona.h

element.h

```
#include "persona.h"
typedef persona element;
/* il resto rimane invariato */
...
```

Include la definizione del tipo persona

Stabilisce che in questo caso element equivale a persona

ESEMPIO element=persona

element.c

– *l'uguaglianza fra persone:*

```
#include "element.h"
boolean isEqual(element e1, element e2){
 return
  strcmp(e1.nome,e2.nome)==0 &&
  strcmp(e1.cognome, e2.cognome)==0 &&
  e1.annoDiNascita==e2.annoDiNascita;
}
...
```

Se utile, si può anche stabilire di considerare uguali due persone se hanno anche solo il cognome uguale, o magari cognome e nome *ma non l'anno, etc.*

ESEMPIO element=persona

element.c

– *l'ordinamento fra persone* (ipotesi: vogliamo ordinare le persone per cognome *e in subordine per nome*)

```
...
boolean isLess(element e1, element e2){
 int cognomeMinore =
     strcmp(e1.cognome, e2.cognome);
 if (cognomeMinore<0) return 1;
 if (cognomeMinore>0) return 0;
 return strcmp(e1.nome, e2.nome)<0;
}
```

ESEMPIO element=persona

element.c

– *l'ordinamento fra persone* (ipotesi: vogliamo ordinare le persone per cognome *e in subordine per nome*)

```
...
boolean isLess(element e1, element e2){
 int cognomeMinore =
     strcmp(e1.cognome, e2.cognome);
 if (cognomeMinore<0) return 1;
 if (cognomeMinore!=0) return (cognomeMinore<0);
}
```