COMPLESSITA’ DEGLI ALGORITMI

- **Tra i problemi che ammettono soluzione ne esistono di più *“facili”* e di più *“difficili”*.**
- ***Teoria della complessità (anni ’70):***
  - **complessità di un *problema***
  - **complessità di un *programma***
  - **valutazione dell’efficienza di un *algoritmo***
- Un programma richiede ***spazio di memoria*** e ***tempo di calcolo***.

COMPLESSITA’ DI UN ALGORITMO

- **Come valutare la complessità *di uno specifico algoritmo*?**
  - ***Contando il numero di operazioni* aritmetiche, logiche, di accesso ai file, etc.**
- *Ipotesi semplificative:*
  - ogni operazione ha costo unitario
  - il tempo globalmente impiegato è proporzionale al numero di operazioni eseguite.
- *Non ci si riferisce ad una specifica macchina.*

MOTIVAZIONI

- **Perché valutare la complessità di un algoritmo?**
  - per scegliere l’algoritmo *più efficiente*
- ***Da cosa dipende* la complessità di un algoritmo?**
  - dall’algoritmo stesso (ovvio...)
  - *dalla **“dimensione” dei dati** a cui l’algoritmo si applica.*

La ***complessità*** dell’algoritmo viene dunque espressa in funzione della ***dimensione dei dati***.

MOTIVAZIONI

- Si consideri un algoritmo che risolve il generico problema P.
- Avere

$$\mathbf{time_{Alg(P)}(N) = 2^N}$$

è molto diverso da avere

$$\mathbf{time_{Alg(P)}(N) = 4*N^3}$$

perché *cambia l’ordine di grandezza* del problema.

ORDINI DI GRANDEZZA

- Tanto per quantificare:

N	$N*\log_2 N$	N^2	N^3	2^N
2	2	4	8	4
10	33	100	10^3	$> 10^3$
100	664	10.000	10^6	$>> 10^{25}$
1000	9.966	1.000.000	10^9	$>> 10^{250}$
10000	13.288	100.000.000	10^{12}	$>> 10^{2500}$

- Se un elaboratore esegue 1000 operazioni/sec, un algoritmo il cui tempo sia dell'ordine di 2^N richiede:

N	*tempo*
10	1 sec
20	1000 sec (17 min)
30	10^6 sec (>10giorni)
40	(>>10 anni)

COMPORTAMENTO ASINTOTICO

<u>Problema</u>:

- **individuare con esattezza l'espressione di $time_A(N)$ è spesso *molto difficile***
- **D'altronde, *interessa capire cosa succede quando i dati sono di grandi dimensioni***
  - con N piccolo, in pratica, qualunque algoritmo è OK
  - ***è con N grande che la situazione può diventare critica* (in particolare: per $N \to \infty$)**
- **Per questo ci interessa il *comportamento asintotico* della funzione $time_A(N)$.**

COMPORTAMENTO ASINTOTICO

- **Anche individuare il comportamento asintotico di $time_A(N)$ non è sempre semplice**
- **CI *interessa* non tanto l'espressione esatta, quanto *l'ordine di grandezza***
  - costante al variare di N
  - lineare, quadratico... (polinomiale) al variare di N
  - logaritmico al variare di N
  - esponenziale al variare di N
- **Si usano notazioni che "danno un'idea" del *comportamento asintotico* della funzione.**

CLASSI DI COMPLESSITA'

- **Le notazioni O e Ω consentono di *dividere gli algoritmi in classi,* in base all'ordine di grandezza della loro complessità.**
  - costante — 1, ... k, ...
  - sotto-lineare — log N oppure N^k con k<1
  - lineare — N
  - sovra-lineare — N*log N, e N^k con k>1
  - esponenziale — c^N oppure N^N
- **<u>Obiettivo</u>: dati due algoritmi, *capire se sono della stessa complessità o se uno è "migliore" (più efficiente, meno complesso) dell'altro.***

COMPLESSITA' DI UN PROBLEMA

- **Interessa anche capire se *il problema in quanto tale* abbia una sua complessità, cioè se sia "intrinsecamente facile" o "intrinsecamente difficile" *indipendentemente dall'algoritmo* che possiamo inventare per risolverlo.**
- **<u>Problema intrattabile</u>:** un problema per cui non esistono algoritmi risolventi di complessità polinomiale (esempio: commesso

DIPENDENZA DAI DATI DI INGRESSO

- **Spesso accade che il costo di un algoritmo dipenda non solo dalla *dimensione* dei dati di ingresso, ma anche dai *particolari valori dei dati di ingresso***
  - ad esempio, un algoritmo che ordina un array può avere un costo diverso secondo se l'array è "molto disordinato" o invece "quasi del tutto ordinato"
  - analogamente, un algoritmo che ricerca un elemento in un array può costare poco, se l'elemento viene trovato subito, o molto di più, se l'elemento si trova "in fondo" o è magari del tutto assente.

DIPENDENZA DAI DATI DI INGRESSO

- **In queste situazioni occorre distinguere diversi casi:**
  - **caso *migliore***
  - **caso *peggiore***
- **Solitamente la complessità si valuta sul *caso peggiore***

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.***

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

<u>*Valutazione di complessità*</u>

- Il numero di *confronti* necessari vale <u>*sempre*</u>:

  (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.**

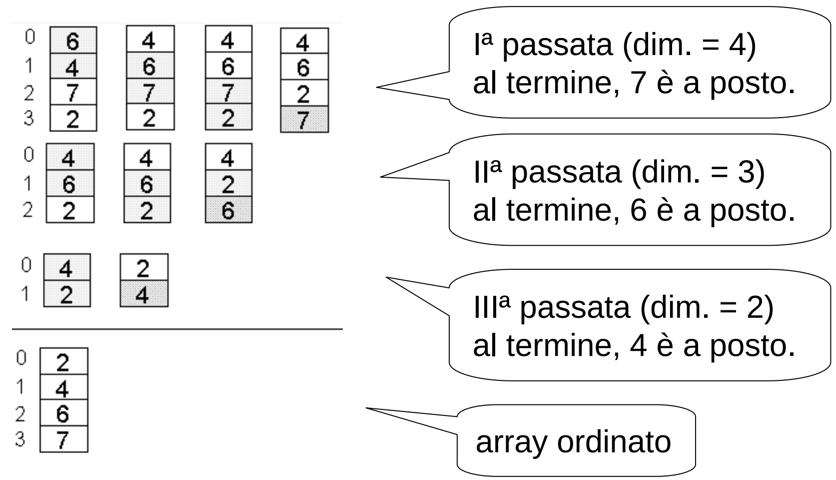
BUBBLE SORT

<u>*Codifica*</u>

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

BUBBLE SORT

<u>*Esempio*</u>



BUBBLE SORT

<u>*Valutazione di complessità*</u>

- 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;
 boolean ordinato = false;
 while (n>1 && !ordinato){
  ordinato = true;
  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 = false; }
  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

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

Protezione dalle inclusioni multiple

`element` usa `boolean`

IL TIPO `element`

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"
#ifndef element_h
#define element_h
typedef persona element;
#endif
/* il resto non si tocca ! */
...
```

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;
}
...
```

Potrei anche stabilire di considerare uguali due persone se hanno anche solo il cognome uguale, o magari cognome e nome *ma non l'anno, etc.*
Il criterio di "uguaglianza" lo stabiliamo noi.

ESEMPIO `element=persona`

- **element.c**
  - *l'ordinamento fra persone* (<u>ipotesi</u>: 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 (cognomeMinore<0);
 return strcmp(e1.nome, e2.nome)<0;
}
```