

**Università degli Studi di Padova**  
**Corso di Laurea in Matematica**  
**A.A. 2008-2009**

**INTRODUZIONE ALLA**  
**PROGRAMMAZIONE**  
**Barbara Di Camillo**

Si ringraziano il Dott. Enrico Grisan e il Dott. Fabio Aiolli per il materiale didattico fornito

# Operatori

## Operatori

- Aritmetici: +, -, \*, /, %
- Incremento e decremento: ++, --
- Relazionali: >, >=, <, <=, ==, !=
- Logici: &&, ||, !
- bitwise: &, |, ^, <<, >>, ~
- Assegnazione: =, +=, -=, \*=, ecc.
- Condizionale ternario: expr1 ? expr2 : expr

# Operatori aritmetici

- E' necessario fare attenzione al significato di un'espressione come  $a + b * c$  dove potremmo volere sia l'effetto di  $(a + b) * c$  sia quello di  $a + (b * c)$
- Tutti gli operatori hanno una propria priorita', e gli operatori ad alta priorita' sono valutati prima di quelli a bassa priorita'. Gli operatori con la stessa priorita' sono valutati da sinistra a destra; Così  $a - b - c$  e' valutato  $(a - b) - c$  come ci si puo' aspettare.
- L'operatore "%" (parte intera della divisione) puo' essere utilizzato solamente con le variabili di tipo integer
- La divisione "/" e' utilizzata sia per gli integer che per i float.

# Operatori aritmetici

- $z=3/2$
- di regola, se entrambi gli argomenti della divisione sono integer, allora verra' effettuata una divisione integer. Per avere 1.5 come risultato, sara' necessario scrivere:  $z=3.0/2$  oppure  $z=3/2.0$  o, ancora meglio,  $z=3.0/2.0$
- Esiste una forma contratta per espressioni del tipo  $expr1 = expr1 \text{ op } expr2$  che diventano:  $expr1 \text{ op } = expr2$
- $i=i+2$  puo' essere scritta come  $i+=2$
- $x=x*(y+3)$  puo' essere scritta come  $x*=y+3$
- Nota: l'espressione  $x*=y+3$  corrisponde a  $x=x*(y+3)$  e non a  $x=x*y+3$

# Operatori incremento e decremento

- Gli operatori incremento ++ e decremento --, possono essere preposti o posposti all'argomento.
- Un'istruzione del tipo **x++** e' piu' veloce di **x=x+1**
- Se sono preposti il valore e' calcolato prima che l'espressione sia valutata, mentre se sono posposti il valore viene calcolato dopo la valutazione della espressione.
- Ad esempio:

```
int x, z=2;
```

```
x=(++z)-1; //A questo punto x=2 e z=3
```

```
int x, z=2
```

```
x=(z++)-1; //A questo punto x=1 e z=3
```

# Operatori Relazionali

- Per testare l'uguaglianza si usa "==" mentre per la disuguaglianza "!=".
- NB.

**if (i==j)** ... esegue il contenuto dell'if se i e' uguale a j

**if (i=j)** ... e' ancora sintatticamente esatto ma effettua l'assegnazione del valore di j ad i e procede se j e' diverso da zero. Di fatto e' come scrivere if i ... con i diverso da zero).

# Operatori Logici

| Expr1 | Expr2 | Expr1 && Expr2 | Expr1    Expr2 | ! Expr1 |
|-------|-------|----------------|----------------|---------|
| 0     | 0     | 0              | 0              | 1       |
| 0     | ≠0    | 0              | 1              | 1       |
| ≠0    | 0     | 0              | 1              | 0       |
| ≠0    | ≠0    | 1              | 1              | 0       |

# Valutazione short-circuit

La valutazione degli operatori logici `&&` e `||` avviene in maniera short-circuit, cioè da sinistra verso destra e si interrompe non appena il risultato diventa noto.

`(a != 0) && (++i < 10)`

`(a!=0) || (++i < 10)`

# Operatori bitwise

Gli operatori di bitwise (che operano sui singoli bit) sono:

- "&" AND
- "|" OR
- "^" XOR
- "~" Complemento a 1 (0=>1, 1=>0)
- "<<" shift a sinistra
- ">>" shift a destra

da non confondere con && (logical and), || (logical or), ecc.

• "~" e' un operatore unario, cioe' opera su un solo argomento indicato a destra dell'operatore.

# Operatori bitwise

- Gli operatori di shift eseguono un appropriato shift dall'operatore indicato a destra a quello indicato a sinistra. L'operatore destro deve essere positivo. I bits liberati vengono riempiti con zero
- $z \ll 2$  shifta i bit in  $z$  di due posti verso sinistra  
se  $z = 00000010$  (2 decimale)  
 $z \gg 2 \Rightarrow z = 00000000$  (0 decimale)  
 $z \ll 2 \Rightarrow z = 00001000$  (8 decimale)
- Uno shift a sinistra e' equivalente ad una moltiplicazione per 2; uno shift a destra equivale ad una divisione per 2.
- L'operazione di shift e' molto piu' veloce della moltiplicazione (\*) o divisione (/); cosi', se occorrono veloci moltiplicazioni o divisioni per 2 si puo' utilizzare lo shift.

# Priorità degli operatori

- L'ordine di priorità, dalla più alta alla più bassa, degli operatori in C è:

|                                             |                             |
|---------------------------------------------|-----------------------------|
| 1. <code>)[]-&gt;</code>                    | 8. <code>^</code>           |
| 2. <code>!~-*&amp; sizeof cast ++ --</code> | 9. <code> </code>           |
| 3. <code>*/%</code>                         | 10. <code>&amp;&amp;</code> |
| 4. <code>+-</code>                          | 11. <code>  </code>         |
| 5. <code>&lt; &lt;= &gt;= &gt;</code>       | 12. <code>?:</code>         |
| 6. <code>== !=</code>                       | 13. <code>= += -=</code>    |
| 7. <code>&amp;</code>                       | 14. <code>,</code>          |

- Quindi: `"a < 10 && 2 * b < c"` è interpretato come: `"(a < 10) && ((2 * b) < c)"`.

# Strutture di controllo

- if
- Operatore "?"
- switch
- for
- while
- do while
- break e continue

# if (else)

```
if (condizione) {
```

```
    /* blocco eseguito se la  
    condizione e' vera */
```

```
} else {
```

```
    /* blocco eseguito se la  
    condizione e' falsa */
```

```
}
```

# if

L'istruzione "if" in C puo' avere tre forme:

1. `if (condizione) {espressione 1}`
2. `if (condizione) {espressione 1} else {espressione 2}`
3. `if (condizione1) {espressione 1} else if (condizione2) {espressione 2} else {espressione 3}`

# Operatore ?

- L' operatore ? (ternary condition) e' la forma piu' efficiente per esprimere semplici if statements.

- Ha la seguente forma:

condizione ? espressione 1 : espressione 2

che equivale a:

if (condizione) {espressione 1} else {  
espressione 2}

# switch case

```
switch (espressione) {  
    case espr-costante1: istruzioni1; break;  
    case espr-costante2: istruzioni2; break;  
    case espr-costante3: istruzioni3; break;  
    case espr-costante4: istruzioni4; break;  
    ...  
    default: istruzioni_default; break;  
}
```

# switch case

- Permette scelte multiple tra un insieme di items.
- Il valore degli item deve essere una costante (le variabili non sono permesse).
- Il break serve per terminare lo switch dopo l'esecuzione di una scelta, altrimenti verrebbe valutato anche il caso successivo (questo, a differenza di molti altri linguaggi).
- E' possibile anche avere un'istruzione nulla, includendo solamente un ";"
- Il caso "default" e' facoltativo e raggruppa tutti gli altri casi.

# switch case

```
switch (letter) {  
case 'A':  
case 'E':  
case 'I':  
case 'O':  
case 'U':  
numervocali++;  
break;  
case " ": numerospazi++;  
break;  
default: numerocostanti++;  
break;  
}
```

Se letter e' una vocale viene incrementato il valore della varibile numervocali, se e' uno spazio (" ") si incrementa numerospazi e altrimenti (se nessuno dei casi precedenti e' vero) viene eseguita la condizione di default: viene incrementato numerocostanti.

# for

**for** (expression1; expression2; expression3)  
{statement}

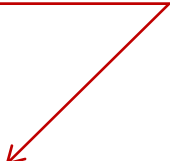
- expression1 inizializza,
- expression2 e' il test di termine
- expression3 e' il modificatore (che puo' anche essere piu' di un semplice incremento).

Nota: fondamentalmente il C tratta le istruzioni "for" come i cicli di tipo "while".

# for

```
int x;  
main() {  
for (x=0;x<3;x++) printf("x=%d\n", x);  
}
```

Stampa un  
numero intero  
decimale



- genera come output sullo schermo (purché includa le librerie): x=0 x=1 x=2

# while

**while** (expression) {statement}

```
int x=3;  
main() {  
while (x>0)  
    { printf("x=%d\n",x);  
      x--; }  
}
```

genera come output sullo schermo:  
x=3 x=2 x=1

# while

- while puo' accettare non solo condizioni ma anche espressioni, per cui risultano corrette le seguenti istruzioni:
  - `while (x--);`
  - `while (x=x+1);`
  - `while (x+=5);`
- solo quando il risultato di `x--`, `x=x+1` oppure `x+=5` ha valore 0 la condizione di while fallisce e si esce dal loop.

# while

- È possibile avere anche complete operazioni di esecuzione nelle espressioni "while":

- `while (i++<10)`

incrementa i fino a raggiungere il valore 10;

- `while ((ch=getchar()) != 'q') putchar(ch);`

usa le funzioni `getchar()` e `putchar()` delle librerie standard, che rispettivamente leggono un carattere dalla tastiera e scrivono un determinato carattere sullo schermo. Il loop `while` continua a leggere dalla tastiera e a visualizzare sullo schermo il carattere digitato, fino a quando non venga battuto il carattere "q".

# do while

- `do {statement} while (expression);`

```
int x=3;
```

```
main() {
```

```
do {printf("x=%d\n",x--);} 
```

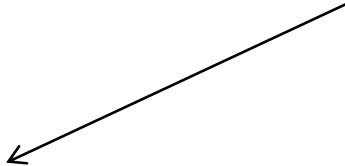
```
/* le graffe sono superflue, visto che  
   racchiudono solamente una istruzione */
```

```
while (x>0);
```

```
}
```

L'output e': x=3 x=2 x=1

viene usato il valore  
corrente di x in  
stampa, e poi viene  
decrementato x.



# break e continue

- Il C fornisce due comandi per controllare i loop:
  - break esce da un loop o da uno switch
  - continue salta una iterazione del loop
- Consideriamo il seguente esempio. Consideriamo una variabile di nome `value`. Se `value` ha valore negativo, dovremo stampare un messaggio di errore ed abbandonare il loop. Se `value` ha valore maggiore di 100, dovremo continuare con il successivo valore in input. Se il valore è 0, dovremo terminare il loop.

# break e continue

```
/* Viene letto un valore intero ed elaborato purché sia  
   maggiore di 0 e minore di 100 */
```

```
while (value !=0) {  
    if (value<0) { printf("Valore non  
        ammesso\n"); break; } /* Abbandona il loop */  
    if (value>100) { printf("Valore non  
        ammesso\n"); continue;} /*Torna nuovamente  
        all'inizio del loop */  
    . . . /* qui ci saranno delle istruzioni utilizzano  
        value, che è sicuramente > 0 e <=100 */  
    . . . /*qui ci saranno delle istruzioni per aggiornare  
        il valore value, ad esempio leggendolo da tastiera*/  
}
```

# Le funzioni

- I programmi C sono costituiti da definizioni di variabili e funzioni.
- Una definizione di funzione ha il seguente formato:

```
tipo-ritornato nome-funzione(lista-parametri)
{
dichiarazioni
istruzioni
}
```

- Le definizioni di funzioni possono comparire in qualsiasi ordine all'interno di uno o più file sorgente

# Le funzioni

- La seguente funzione eleva al quadrato il numero reale  $n$  dato in ingresso e restituisce il numero  $n^2$

```
float quad(float n) {  
    float res;  
    res=pow(n, 2) ;  
    return(res) ; }
```

# Le funzioni

- Tuttavia, per poterla utilizzare, una funzione deve essere dichiarata all'interno di un programma.
- Il programma qui di fianco richiama dal main la funzione quad

```
#include<stdio.h>
float quad(float n);

main(){float x;
      float k=5;
      x=quad(k);

      printf("k=\t%f",k);

      printf("\nk^2=\t%f",
x);
      }

float quad(float n){
      float res;
```

# Le funzioni

- La dichiarazione `float quad(float n);` all'inizio del programma è detta prototipo della funzione; indica che quad si aspetta un argomento float e restituisce un float.
- Il prototipo deve essere in accordo con la definizione della funzione stessa (a parte per i nomi dei parametri).
- L'istruzione `return` ritorna il controllo alla funzione chiamante (main), restituendo il valore specificato.
- Anche la funzione speciale main può avere un'istruzione `return`, che ritorna il controllo al chiamante, cioè all'ambiente in cui il programma viene eseguito.

- Abbandoniamo ora il linguaggio C.
- Quello che abbiamo visto è più che sufficiente per continuare a lavorare in laboratorio e per considerare alcune caratteristiche generali degli algoritmi

# Il calcolo del fattoriale

- La funzione fattoriale, molto usata nel calcolo combinatorio, è così definita

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

dove  $n$  è un numero intero non negativo

# Il calcolo del fattoriale

$$5! = 5(5-1)! = 54 = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$$

$$4! = 4(4-1)! = 43 = 4 \cdot 3 \cdot 2 \cdot 1 = 24$$

$$3! = 3(3-1)! = 32! = 3 \cdot 2 \cdot 1 = 6$$

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

$$1! = 1(1-1)! = 1 \cdot 0! = 1 \cdot 1 = 1$$

$$0! = 1$$

- Quindi, per ogni  $n$  intero positivo, il fattoriale di  $n$  è il prodotto dei primi  $n$  numeri interi positivi

# Il calcolo del fattoriale

- Supponiamo di voler progettare un algoritmo per il calcolo del fattoriale

```
int fattor(int n) {  
    int res=1;  
    int i;  
    if (n < 0) res = -1;  
    else      { for (i=n;i>0;i-  
-) res=res*i; }  
    return(res);  
}
```

# Il calcolo del fattoriale

- Realizzando direttamente la definizione, sarebbe stato più naturale scrivere:

```
int fattor(int n) {  
    int res;  
    if (n == 0)      res  
= 1;  
    else  
        res=n*fattor(n-  
1) ;  
    return(res) ;  
}
```

# La ricorsione

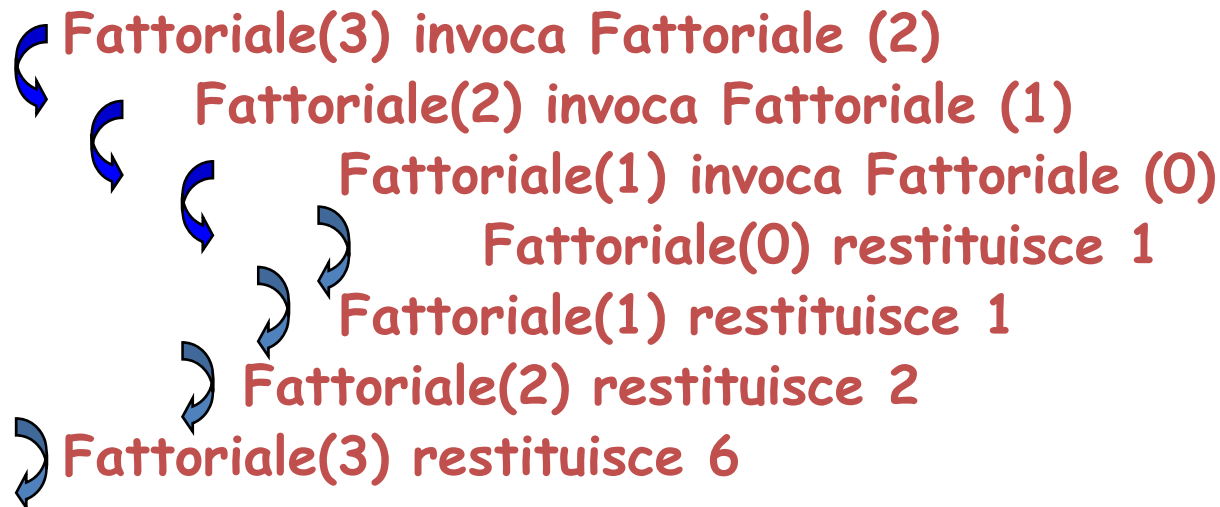
- *Invocare un metodo* mentre si esegue *lo stesso metodo* è un paradigma di programmazione che si chiama **ricorsione**  
e un metodo che ne faccia uso si chiama **metodo ricorsivo**
- La ricorsione è uno strumento molto potente per realizzare alcuni algoritmi, ma è anche fonte di molti errori di difficile diagnosi

# La ricorsione: come funziona?

- Quando una funzione viene invocata:
  - si sospende l'esecuzione del metodo invocante (le variabili locali rimangono congelate)
  - si esegue il metodo invocato fino alla sua terminazione (con nuove variabili locali)
  - si riprende l'esecuzione del metodo invocante dal punto in cui era stata sospesa (recuperando le variabili locali)
- La stessa sequenza di azioni viene compiuta quando un metodo invoca sé stesso.

# La ricorsione

- Vediamo la sequenza usata per calcolare  $3!$  si invoca `Fattoriale(3)`



- Si crea quindi una lista LIFO (last in first out) di metodi "in attesa", ciascuno con le sue variabili locali, che si allunga e che poi si accorcia fino ad estinguersi

# La ricorsione

- In ogni caso, anche se il meccanismo di funzionamento della ricorsione può sembrare complesso, la chiave per un suo utilizzo proficuo è
  - dimenticarsi come funziona la ricorsione, ma sapere come vanno scritti i metodi ricorsivi perché il tutto funzioni!
- Esistono infatti **due regole** ben definite che vanno utilizzate per scrivere metodi ricorsivi che funzionino

# Caso base

- Prima regola

- il metodo ricorsivo *deve fornire la soluzione del problema in almeno un caso particolare, senza ricorrere ad una chiamata ricorsiva*
- tale caso si chiama **caso base** della ricorsione
- Nell'esempio del fattoriale, il caso base era

```
if (n == 0) res = 1;
```

- a volte ci sono più casi base, non è necessario che il caso base sia unico

# Passo ricorsivo

- **Seconda regola**
  - il metodo ricorsivo *deve effettuare la chiamata ricorsiva dopo aver **semplificato** il problema*
  - nel nostro esempio, per il calcolo del fattoriale di  $n$  si invoca la funzione ricorsivamente per conoscere il fattoriale di  $n-1$ , cioè per risolvere un problema più semplice

```
else    res=n*fattor(n-1);
```

- il concetto di "problema più semplice" varia di volta in volta: in generale, ***bisogna avvicinarsi ad un caso base***

# Ricorsione e algoritmi

- Le regole appena viste sono fondamentali per poter dimostrare che la soluzione ricorsiva di un problema sia un algoritmo
  - in particolare, che arrivi a conclusione in un numero **finito** di passi
- Le chiamate ricorsive potrebbero succedersi una dopo l'altra, all'infinito

# Ricorsione e algoritmi

Se

- ad ogni invocazione il problema diventa sempre più semplice e si avvicina al caso base
- la soluzione del caso base non richiede ricorsione



allora certamente la soluzione viene calcolata in un numero finito di passi, per quanto complesso possa essere il problema

# Ricorsione infinita

- Non tutti i metodi ricorsivi realizzano algoritmi
  - *se manca il caso base*, il metodo ricorsivo continua ad invocare se stesso all'infinito
  - *se il problema non viene semplificato ad ogni invocazione ricorsiva*, il metodo ricorsivo continua ad invocare se stesso all'infinito
- Dato che la lista dei metodi "in attesa" si allunga indefinitamente, l'ambiente runtime esaurisce la memoria disponibile per tenere traccia di questa lista, e il programma termina con un errore

# Ricorsione Infinita

- Il seguente programma (oltre a non avere senso) presenta ricorsione infinita:

```
if (n == 0) res = 1;  
else      res=n*fattor(n-1);
```

- Il programma terminerà con la segnalazione dell'errore **StackOverflowError**
  - il *runtime stack* è la struttura dati che gestisce le invocazioni in attesa
  - *overflow* significa *trabocco*

# Ricorsione in coda

- Esistono diversi tipi di ricorsione
- Il modo visto fino ad ora si chiama **ricorsione in coda** (*tail recursion*)
  - il metodo ricorsivo esegue *una sola invocazione ricorsiva* e tale invocazione è l'*ultima azione* del metodo

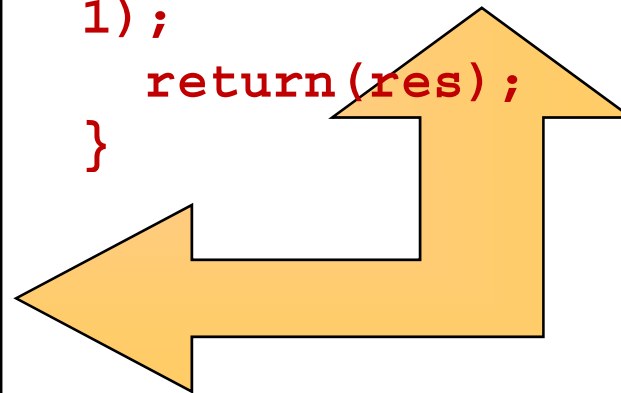
```
if (n == 0) res = 1;  
else    res=n*fattor(n-1);
```

# Iterazione e ricorsione

- La ricorsione in coda può sempre essere agevolmente **eliminata**, trasformando il metodo ricorsivo in un metodo che usa un **ciclo**

```
int fattor(int n) {  
    int res=1;  
    int i;  
    if (n < 0)        res = -1;  
    else              { for  
(i=n;i>0;i--) res=res*i; }  
    return(res);  
}
```

```
int fattor(int n){  
    int res;  
    if (n == 0)      res =  
1;  
    else  
        res=n*fattor(n-  
1);  
    return(res);  
}
```



## Programma ricorsivo per il calcolo dei numeri fattoriali da 0 a 4

```
#include<stdio.h>
int fattor (int n);

main(){
int x;
int n;
for (n=0;n<5;n++){
    x=fattor(n);
    printf("\nFattoriale di %d = %d",n,x);
}

int fattor(int n){
    int res;
    if (n == 0)    res = 1;
    else res=n*fattor(n-1);
    return(res);}
```

## Programma iterativo per il calcolo dei numeri fattoriali da 0 a 4

```
#include<stdio.h>
int fattor (int n);

main(){
int x;
int n;
for (n=0;n<5;n++){
    x=fattor(n);
    printf("\nFattoriale di %d = %d",n,x);}
}

int fattor(int n) {
    int res=1;
    int i;
    if (n < 0)    res = -1;
    else { for (i=n;i>0;i--) res=res*i; }
    return(res);}
```

# Ricorsione: motivazioni

- Allora, a cosa serve la ricorsione in coda?
  - Non è necessaria, però in alcuni casi rende il codice più leggibile
  - È utile quando la soluzione del problema è esplicitamente ricorsiva (es. fattoriale)
  - In ogni caso, la ricorsione in coda è *meno efficiente* del ciclo equivalente, perché il sistema deve gestire le invocazioni sospese

# Ricorsione multipla

- Si parla di ricorsione multipla quando un metodo invoca se stesso più volte durante una sua esecuzione
- Esempio: il calcolo dei numeri di Fibonacci

$$\text{Fib}(n) = \begin{cases} n & \text{se } 0 \leq n < 2 \\ \text{Fib}(n-2) + \text{Fib}(n-1) & \text{se } n \geq 2 \end{cases}$$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, ...

# Ricorsione multipla

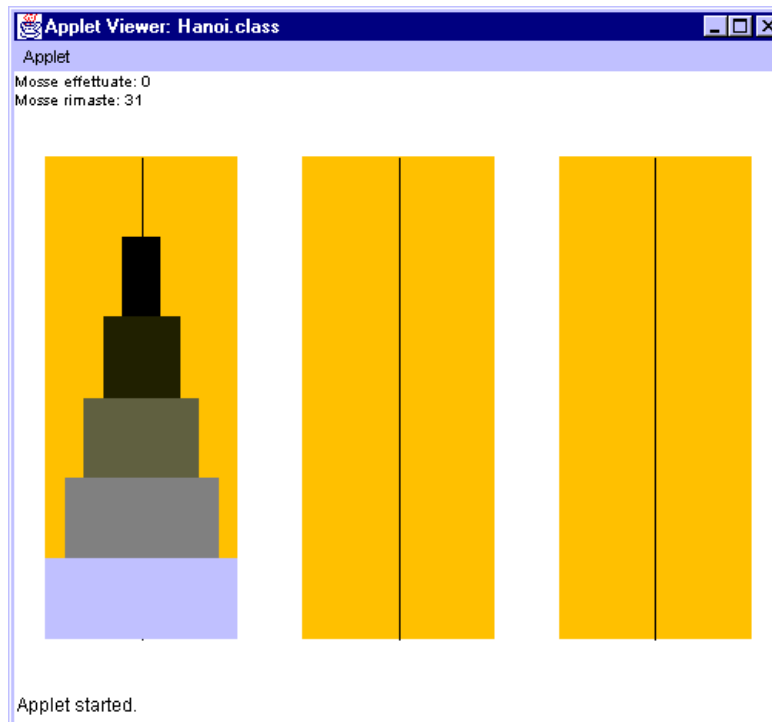
- La ricorsione multipla va usata con molta attenzione, perché può portare a programmi molto inefficienti
- Eseguendo il calcolo dei numeri di Fibonacci di ordine crescente...
  - ... si nota che il tempo di elaborazione cresce MOLTO rapidamente... quasi 3 milioni di invocazioni per calcolare Fib(31) !!!!

# Torre di Hanoi

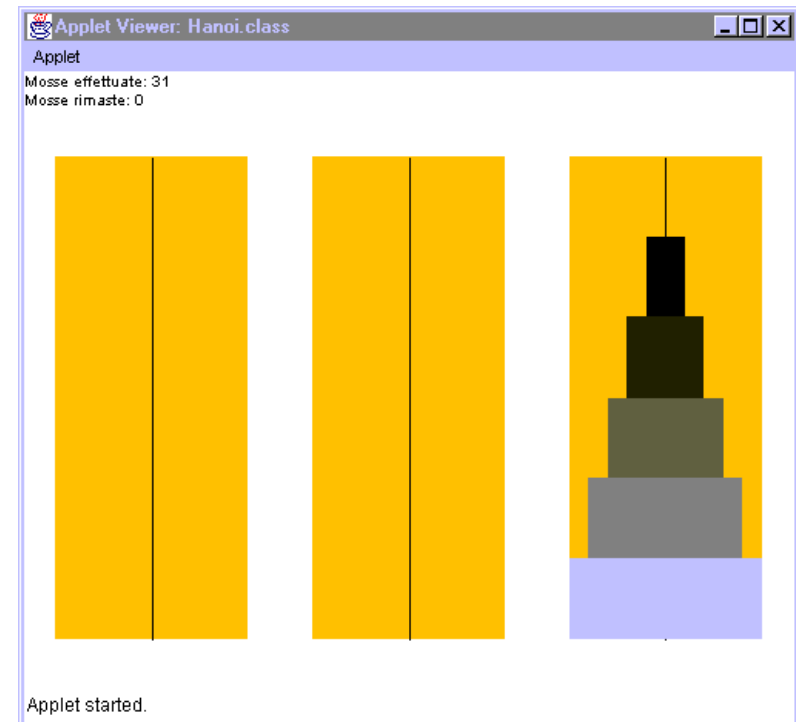
- Il rompicapo è costituito da tre pile di dischi ("torri") allineate
  - all'inizio tutti i dischi si trovano sulla pila di sinistra
  - alla fine tutti i dischi si devono trovare sulla pila di destra
- I dischi sono tutti di dimensioni diverse e quando si trovano su una pila devono rispettare la seguente regola
  - *nessun disco può avere sopra di sé dischi più grandi*

# Torre di Hanoi

Situazione iniziale

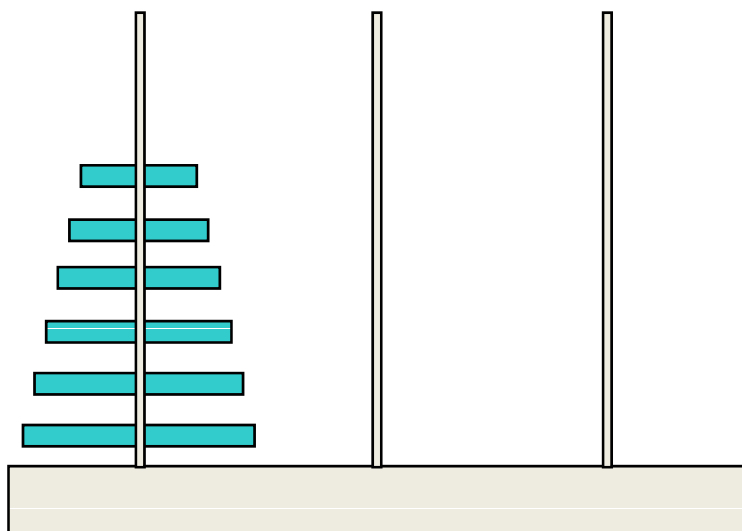


Situazione finale

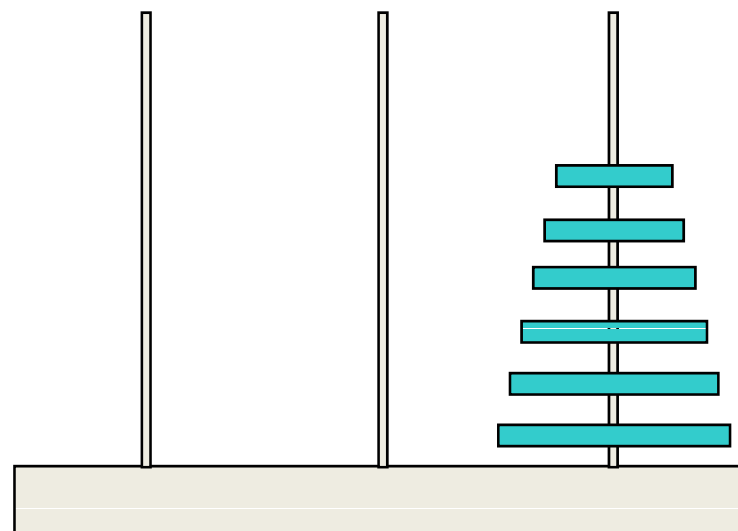


# Torre di Hanoi

Situazione iniziale



Situazione finale



# Torre di Hanoi

- Per risolvere il rompicapo bisogna:
  - *spostare un disco alla volta*
  - un disco può essere rimosso dalla cima della torre ed inserito in cima ad un'altra torre
  - *nessun disco può avere sopra di sé dischi più grandi*

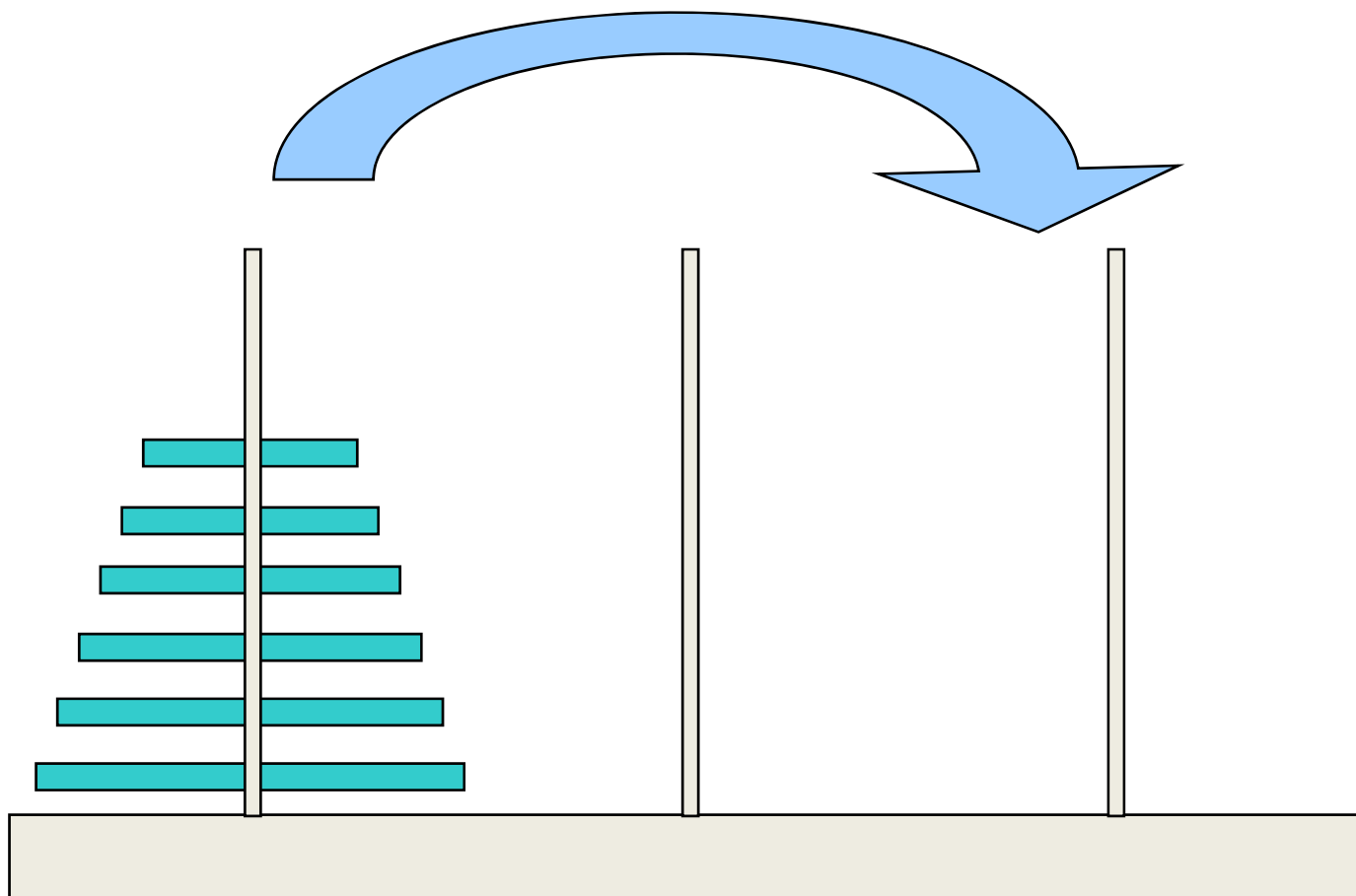
# Torre di Hanoi

- Per il rompicapo delle Torri di Hanoi è noto un algoritmo di soluzione ricorsivo:
  - Il problema generale consiste nello spostare  $n$  dischi da una torre ad un'altra, usando la terza torre come deposito temporaneo
  - Per spostare  $n$  dischi da una torre all'altra si suppone, come sempre si fa nella ricorsione, di saper spostare  $n-1$  dischi da una torre all'altra

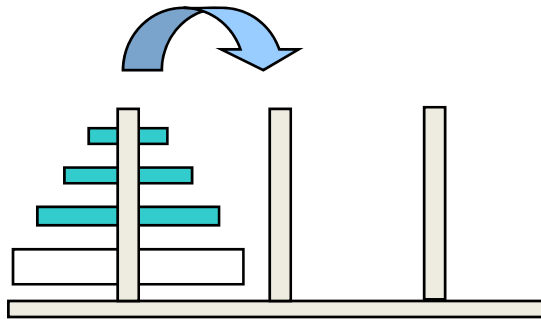
# Torre di Hanoi

- Il **caso base** si ha quando  **$n$**  vale 1, in questo caso possiamo spostare liberamente il disco da una torre ad un'altra
- Per spostare  **$n$**  dischi dalla torre 1 alla torre 3
  - 1) gli  **$n-1$**  dischi in cima alla torre 1 vengono spostati sulla torre 2, usando la torre 3 come deposito temporaneo (si usa una chiamata ricorsiva, al termine della quale la torre 3 rimane vuota)
  - 2) il disco rimasto nella torre 1 viene portato nella torre 3
  - 3) gli  **$n-1$**  dischi in cima alla torre 2 vengono spostati sulla torre 3, usando la torre 1 come deposito temporaneo (si usa una chiamata ricorsiva, al termine della quale la torre 1 rimane vuota)

# Torre di Hanoi



# Torre di Hanoi



**Algoritmo** Hanoi( $n, da, a, int$ )

**input:** il numero di dischi  $n$ , il perno di partenza, di arrivo e intermedio

**output:** le mosse necessarie

**if**  $n = 1$  **then**

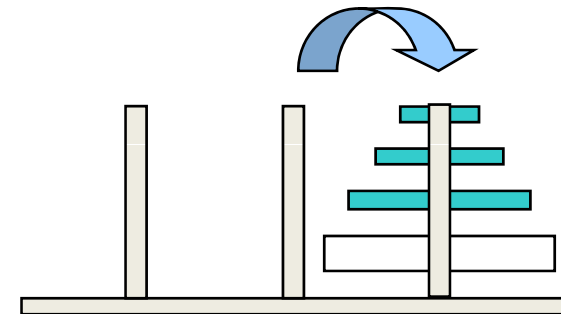
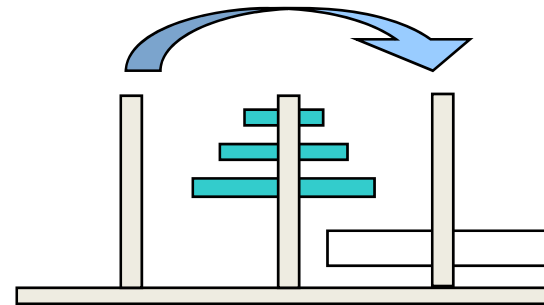
    muovi ( $1, da, a$ )

**else**

    hanoi ( $n-1, da, int, a$ )

    muovi ( $n, da, a$ )

    hanoi ( $n-1, int, a, da$ )



# Torre di Hanoi

- Si può dimostrare che il numero di mosse necessarie per risolvere il rompicapo con l'algoritmo proposto è pari a:

$$2^n - 1$$

- Il tempo necessario alla soluzione è proporzionale al numero di mosse (ogni mossa richiede un tempo costante)

# Torre di Hanoi

- Una leggenda narra che alcuni monaci buddisti in un tempio dell'Estremo Oriente siano da sempre impegnati nella soluzione del rompicapo, spostando fisicamente i loro 64 dischi da una torre all'altra, consapevoli che quando avranno terminato il mondo finirà
- Sono necessarie  $2^{64}$  mosse, che sono circa 16 miliardi di miliardi di mosse
- Supponendo che i monaci facciamo una mossa ogni minuto, essi fanno circa 500000 mosse all'anno, quindi il mondo finirà tra circa 30 mila miliardi di anni
- Un processore ad 1GHz che fa una mossa ad ogni intervallo di clock (un miliardo di mosse al secondo...) impiega 16 miliardi di secondi, che sono circa 500 anni...

# Appendice sulla complessità

# Attributi degli algoritmi: correttezza

- La prima caratteristica di un algoritmo è la correttezza, cioè deve terminare e fornire una **soluzione corretta** del problema.

# Attributi degli algoritmi: efficienza

- Tempo e spazio di memoria sono risorse limitate di un calcolatore, per cui dovendo scegliere fra due algoritmi corretti, si preferirà quello che usa meno risorse.
- L'efficienza di un algoritmo indica quanto parsimoniosamente esso utilizza le risorse a disposizione

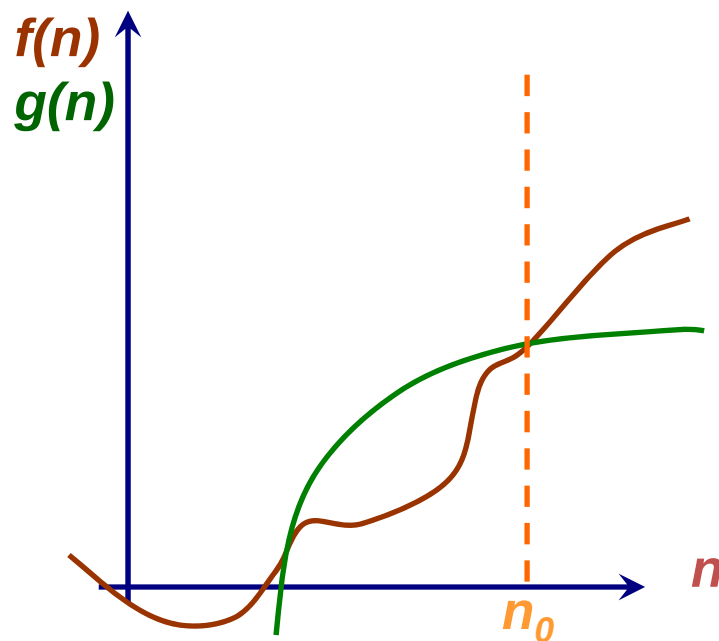
# Notazione $o()$ , $O()$ , $\Theta()$

- Per quantificare l'efficienza di un algoritmo al variare della numerosità dei dati su cui agisce si usa cercare di esprimere la sua **complessità computazionale** in termini di funzioni della numerosità dei dati  $n$

# Limite inferiore: $o()$

- Si dice che una funzione  $f(n)$  è "o piccolo" di una funzione  $g(n)$ , scrivendo  $f(n)=o(g(n))$ , se:

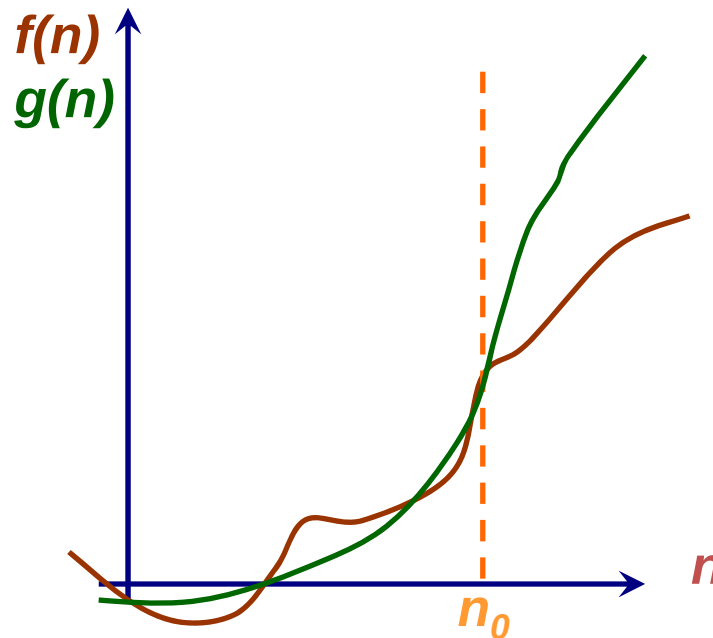
$$\exists n_0, \exists M : \forall n > n_0 \Rightarrow M \cdot g(n) \leq f(n)$$



# Limite superiore: $O()$

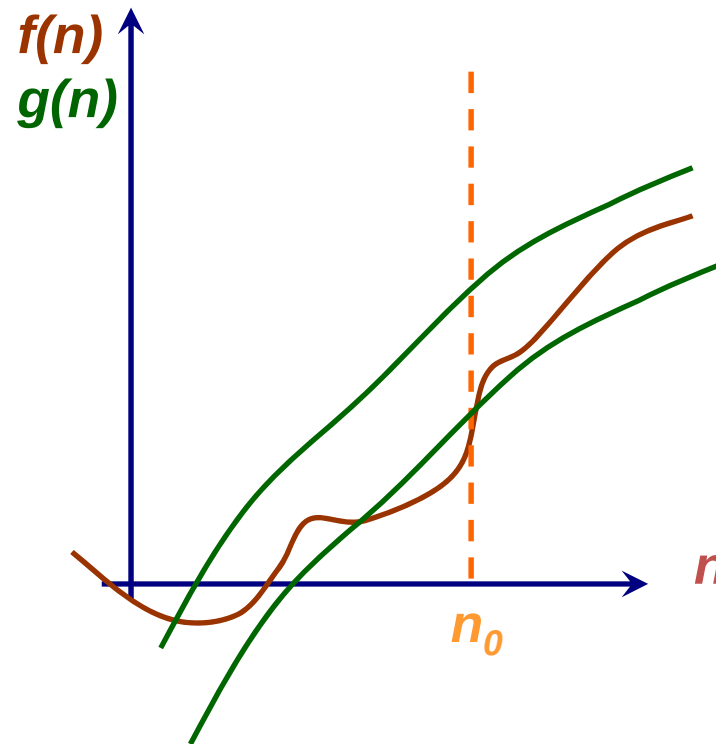
- Si dice che una funzione  $f(n)$  è "o grande" di una funzione  $g(n)$ , scrivendo  $f(n) = O(g(n))$ , se:

$$\exists n_0, \exists M : \forall n > n_0 \Rightarrow f(n) \leq M \cdot g(n)$$



# Andamento asintotico: $\Theta()$

- Si dice che una funzione  $f(n)$  è "theta grande" di una funzione  $g(n)$ , scrivendo  $f(n) = \Theta(g(n))$ , se la funzione  $f(n)$  è contemporaneamente  $o(g(n))$  ed  $O(g(n))$ :



# Complessità computazionale Torre di Hanoi

- Se il numero di mosse necessarie per risolvere il rompicapo della torre di Hanoi con l'algoritmo proposto è pari a:  $2^n - 1$ , allora l'algoritmo ha complessità computazionale pari a  $\Theta(2^n)$

# Ordinamento di una lista: bubble-sort

- Vediamo un'altro esempio: vogliamo ordinare una lista di numeri
- L'idea è di far "galleggiare" il minimo della lista nelle prima posizione
- Riduco la parte di lista da considerare, escludendo le prime posizioni già ordinate

# Ordinamento: bubble-sort

Passo 1: parto dalla fine dell'array e scambio il valore di due elementi se quello con l'indice più alto è minore dell'altro.

|   |    |    |    |    |
|---|----|----|----|----|
| 1 | 87 | 14 | 27 | 22 |
|---|----|----|----|----|



|   |    |    |    |    |
|---|----|----|----|----|
| 1 | 87 | 14 | 22 | 27 |
|---|----|----|----|----|

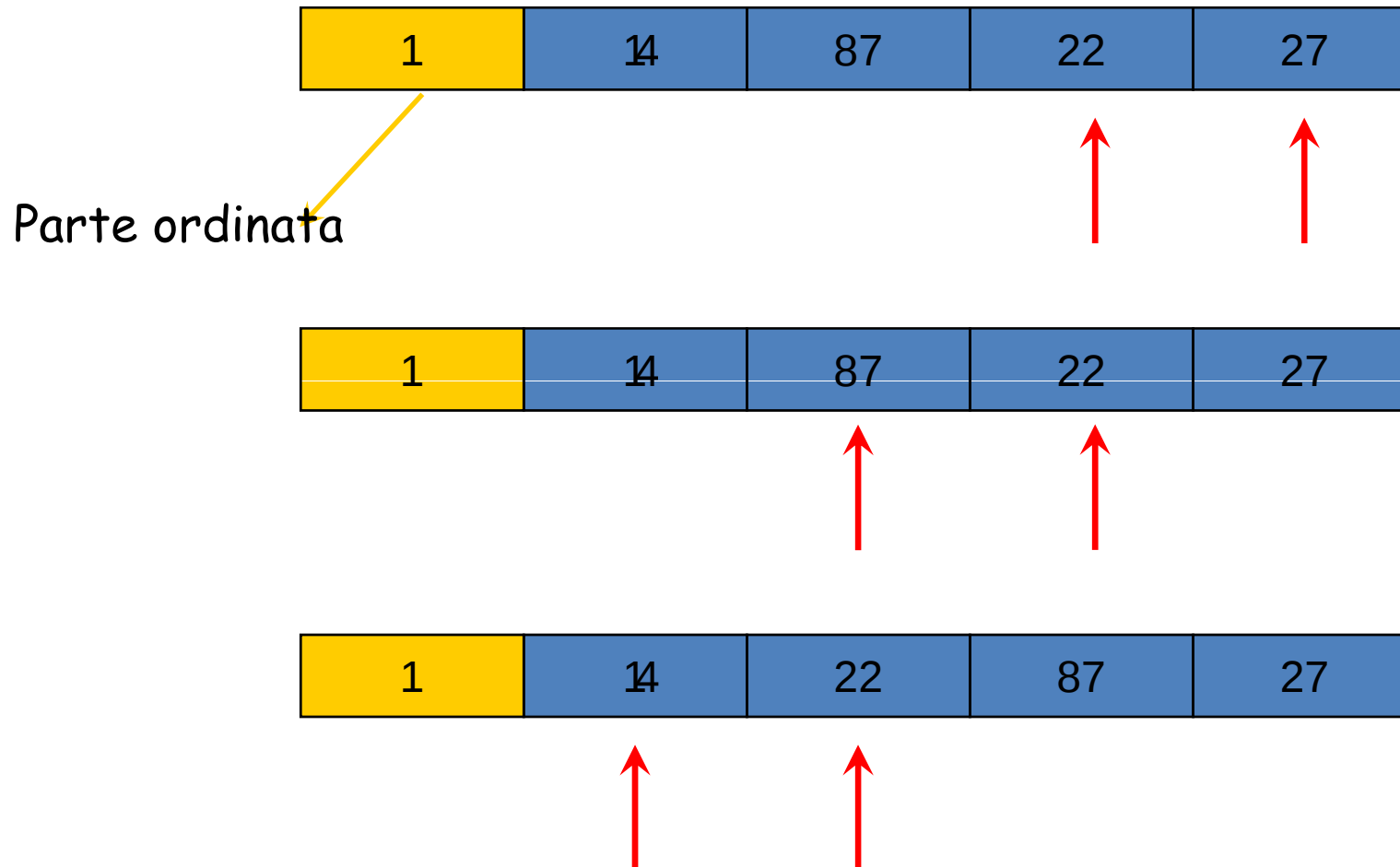


|   |    |    |    |    |
|---|----|----|----|----|
| 1 | 14 | 87 | 22 | 27 |
|---|----|----|----|----|



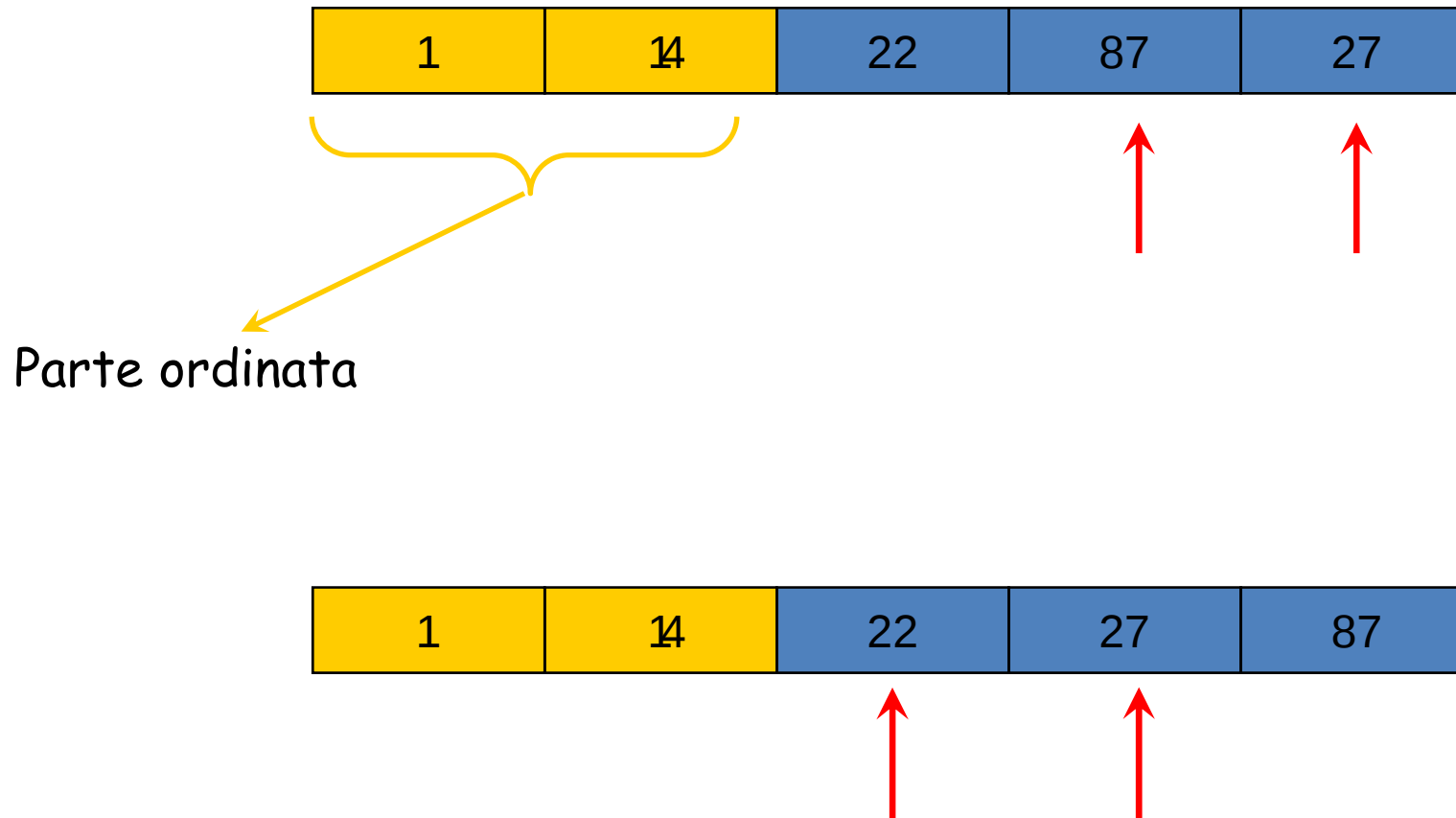
# Ordinamento: bubble-sort

Passo 2: riduco l'array da considerare per l'ordinamento e ripeto il passo 1



# Ordinamento: bubble-sort

Passo 3: riduco l'array da considerare per l'ordinamento e ripeto il passo 1



# Bubble-sort: efficienza

- L'idea è di far "galleggiare" il minimo della lista nelle prima posizione
- Riduco la parte di lista da considerare, escludendo le prime posizioni già ordinate
- la parte disordinata dell'array si riduce di un solo elemento ad ogni passo, e tutti gli elementi della parte disordinata devono essere analizzati

# Bubble sort: efficienza

Bisogna sommare la complessità computazionale di tutti i passi:

$$n + (n-1) + (n-2) + \dots + 2 + 1$$

$$\sum_{i=1}^n (n-i) = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2)$$
