



UNIVERSITÀ DI PARMA

Le Funzioni

E Pluribus Unum

Autore Ignoto, Moretum, I sec. A.C.

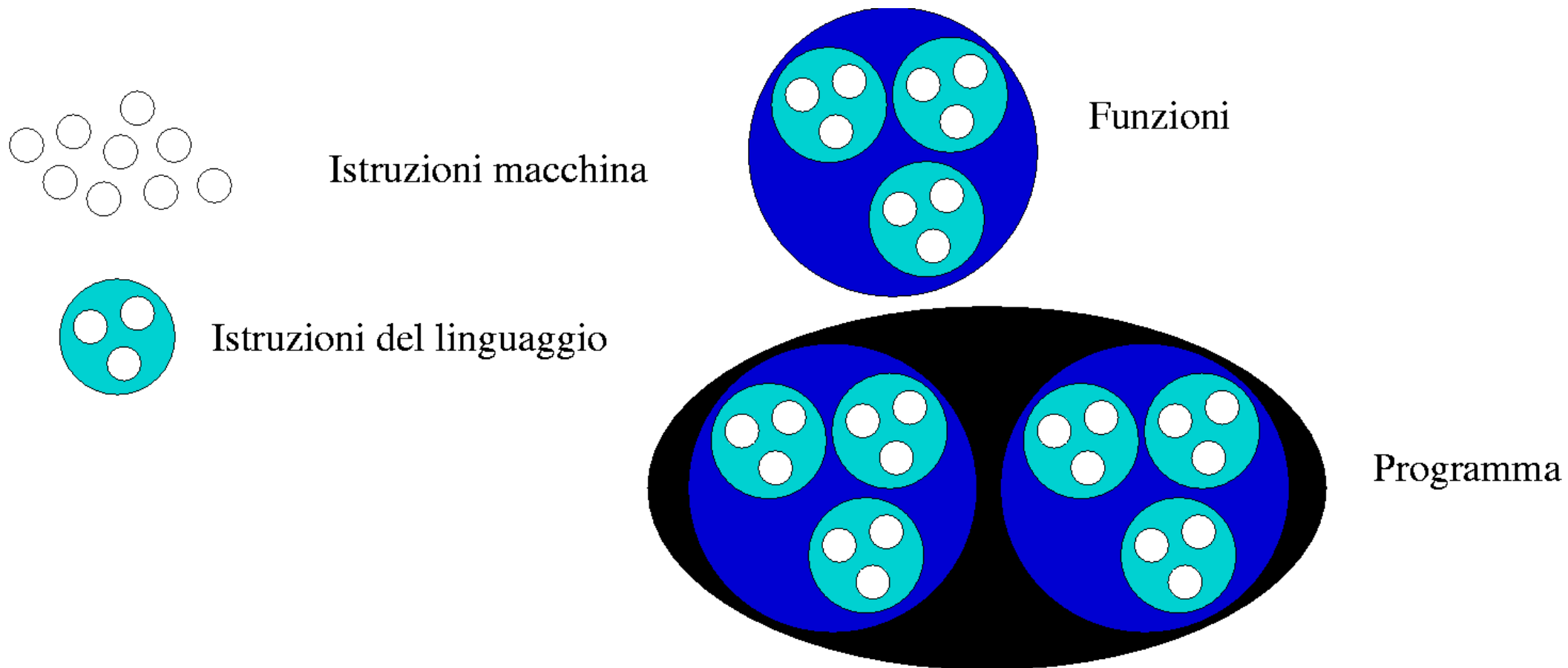
- Modularità
- Le funzioni predefinite
 - La libreria standard del C
- Definire una funzione
 - Tipo restituito
 - Parametri Formali
 - Passaggio dei dati
 - I prototipi
- La Ricorsione

SUMMARY



- Una funzione è un insieme di istruzioni del linguaggio
- Tutti i linguaggi di alto livello permettono l'utilizzo di funzioni o procedure.
- Operazioni più complesse di quelle base del linguaggio

- **Fattorizzazione:** una funzione può essere definita una singola volta ma usata un numero qualunque di volte
 - Riduco dimensione codice → maggior efficienza
- **Facilità di modifica:** viene ridotta la ridondanza
 - Modifico/miglioro/correggo codice in un solo punto
- **Migliore leggibilità:** si possono isolare i dettagli del codice
 - L'uso delle funzioni può essere autoesplicativo



C Functions



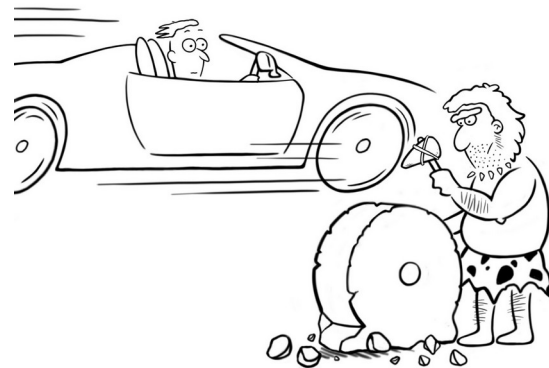
```
graph TD; A[C Functions] --> B[Built-in Functions]; A --> C[User-defined Functions];
```

Built-in Functions

```
printf()  
scanf()  
malloc()  
rand()  
...
```

User-defined Functions

- Non reinventare la ruota!
- La *C Standard Library* fornisce numerose funzioni predefinite
 - Conoscere la loro esistenza...
 - Includere header opportuno
 - Usarle!



- Principali header
 - `stdlib.h` → utilità generale (I/O, memoria, tipi di dato, ricerca...)
 - `string.h` → funzioni di stringa
 - `math.h` → funzioni matematiche
 - `time.h` → gestione ora, data...
 - `ctype.h` → caratteri

Definire una nuova funzione

- Già fatto... → la main()!

Tipo di dato restituito
(int, float, char...)

Nome della funzione

Elenco parametri
formali

↓ ↓ ↓

`<return type>` `<function name>(<formal param1>, ...)`

`{`
`<function body>`
`}`

← Corpo Funzione
(dichiarazioni e statement)

- Tipo di dato restituito
 - Quelli già visti... e quelli che vedremo!
 - Nel caso di puntatori $\rightarrow *$
 - E se non devo restituire nessun dato?
 - Void
- Nome
 - Regole già viste per le variabili
 - Opportunamente mnemonico!

- Elenco parametri formali
 - Lista separata da “,”
 - Specificare tipo e nome
 - Se non necessari \rightarrow void

- Corpo funzione
 - Racchiuso tra { }
 - Codice che realizza funzione
 - Può essere anche vuoto
 - Utile in fasi iniziali sviluppo
 - Se restituisco qualcosa obbligo `return`

- Il compilatore quando incontra una chiamata a funzione deve sapere di che funzione si tratta
 - Il codice viene letto dall'alto in basso
 - Non sempre possibile definire una funzione prima di una sua chiamata
- Soluzione → riga di definizione della funzione (prototipo)

- L'invocazione di una funzione definita da noi segue le stesse regole già viste per le funzioni predefinite
 - Devo passare un numero e un tipo di argomenti coerenti con i parametri della funzione
 - Attenzione alle conversioni di tipo!

- Quando invoco una funzione → argomenti
 - O anche “parametri attuali” o “effettivi”
 - Valori passati ad una funzione
- Nella definizione di una funzione → parametri formali
 - Di fatto variabili locali alla funzione
 - A loro viene assegnato il valore degli argomenti

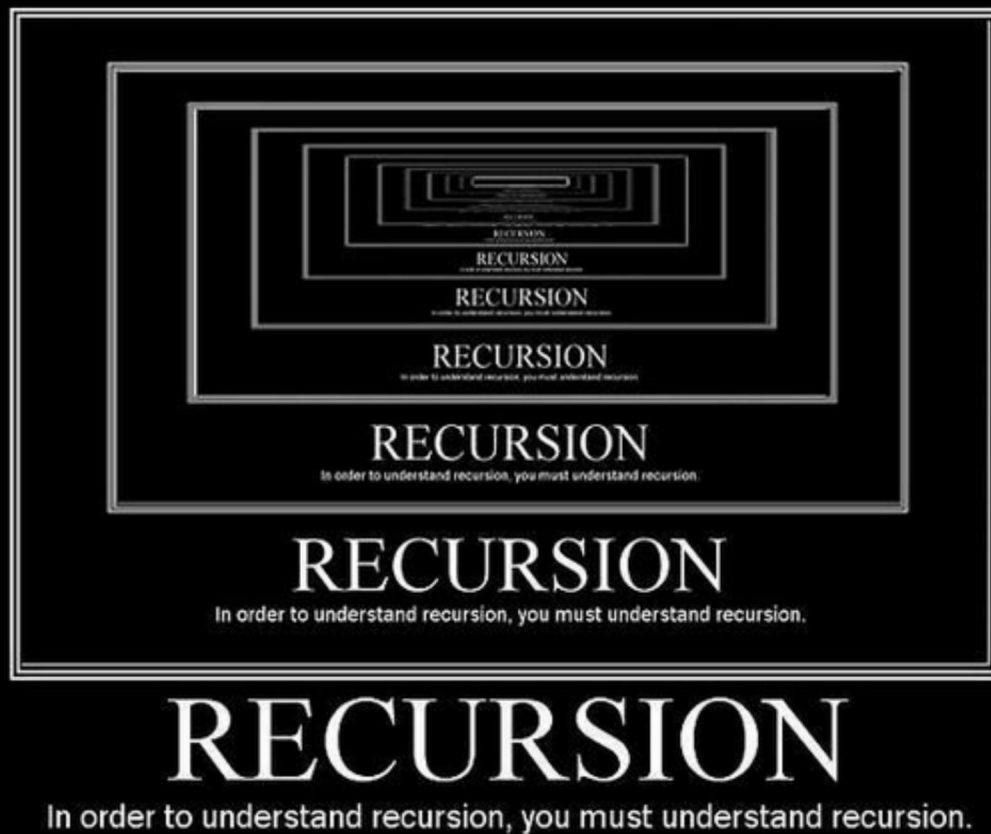
- Passaggio argomenti, due modalità
 - Per valore
 - Per indirizzo

- Passaggio argomenti, per valore
 - È il default per tutti i dati ad eccezione degli array
 - Viene creata una copia dell'argomento
 - La funzione quindi lavora su copie, non sul dato originale
 - Impossibile agire su variabili passate alla funzione

- Passaggio argomenti per indirizzo
 - È il default per gli array
 - Viene passato alla funzione l'indirizzo del dato
 - Attenzione al tipo di dato!
 - La funzione può modificare il dato originale
 - Effetti collaterali

- Permette
 - Uscita funzione e
 - Restituzione valore per funzioni non void
- Non obbligatorio per funzioni void
- Può restituire uno e un solo valore

`return (<espressione>);`



- Approccio molto elegante e semplice che permette di risolvere problemi anche complessi
- La ricorsione si può applicare a tutti quei problemi in cui la risoluzione dipende dalla risoluzione dello stesso problema però di ordine di grandezza inferiore ovvero più semplice
- Esempio: il fattoriale

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

- Esempio: fattoriale
 - Il fattoriale di n lo posso calcolare come funzione del fattoriale di $n-1$
 - Esistono inoltre casi specifici

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

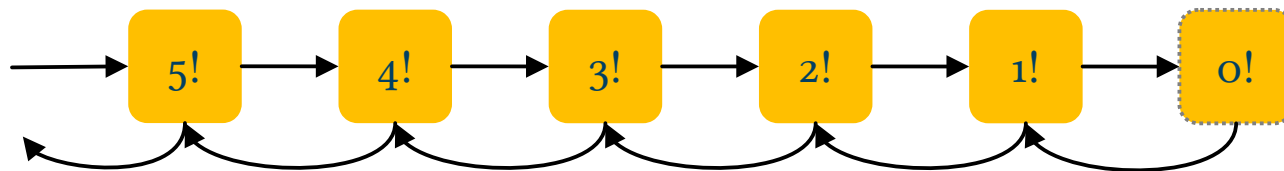
- Una soluzione ricorsiva di un problema è facilmente trasponibile in codice
- **Funzione che chiama se stessa!**

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

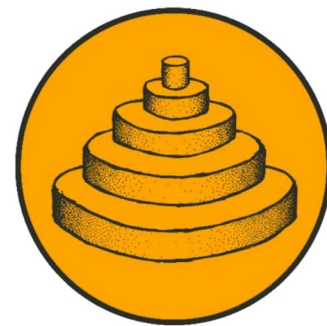
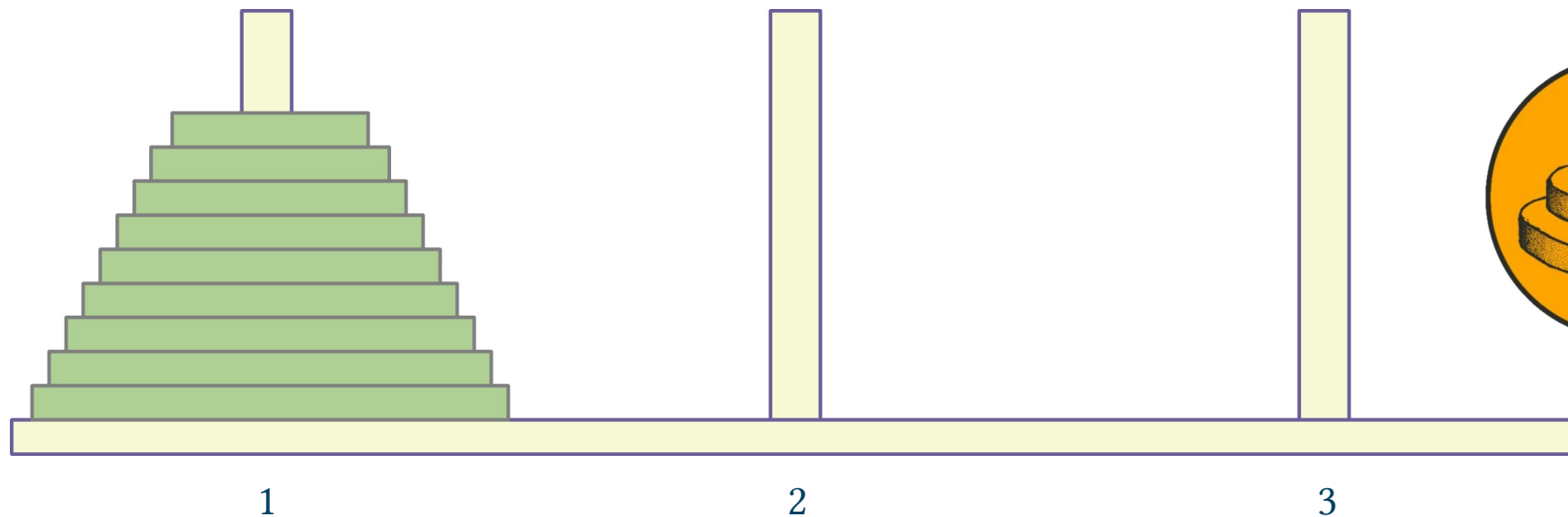


```
unsigned long long fact(unsigned int n)
{
    if(n==0) return 1;
    return n*fact(n-1);
}
```

- Chiamate ricorsive implicano avere tante “istanze” della stessa funzione
 - Ognuna con le sue variabili locali



- Torre di Hanoi



Obbiettivo: spostare n dischi dal piolo 1 al piolo 3

- Torre di Hanoi, definiamo
 - $T(n,x,y)$ → sposta n dischi da piolo x a piolo y
 - $D(x,y)$ → sposta un singolo disco da piolo x a piolo y

$$T(n,1,3) = \begin{cases} \text{per } n=0 & \text{niente} \\ \text{per } n>1 & T(n-1,1,2) + D(1,3) + T(n-1,2,3) \end{cases}$$

- Errori Comuni
 - Mancanza condizione di uscita
 - La semplicità può mascherare inefficienza



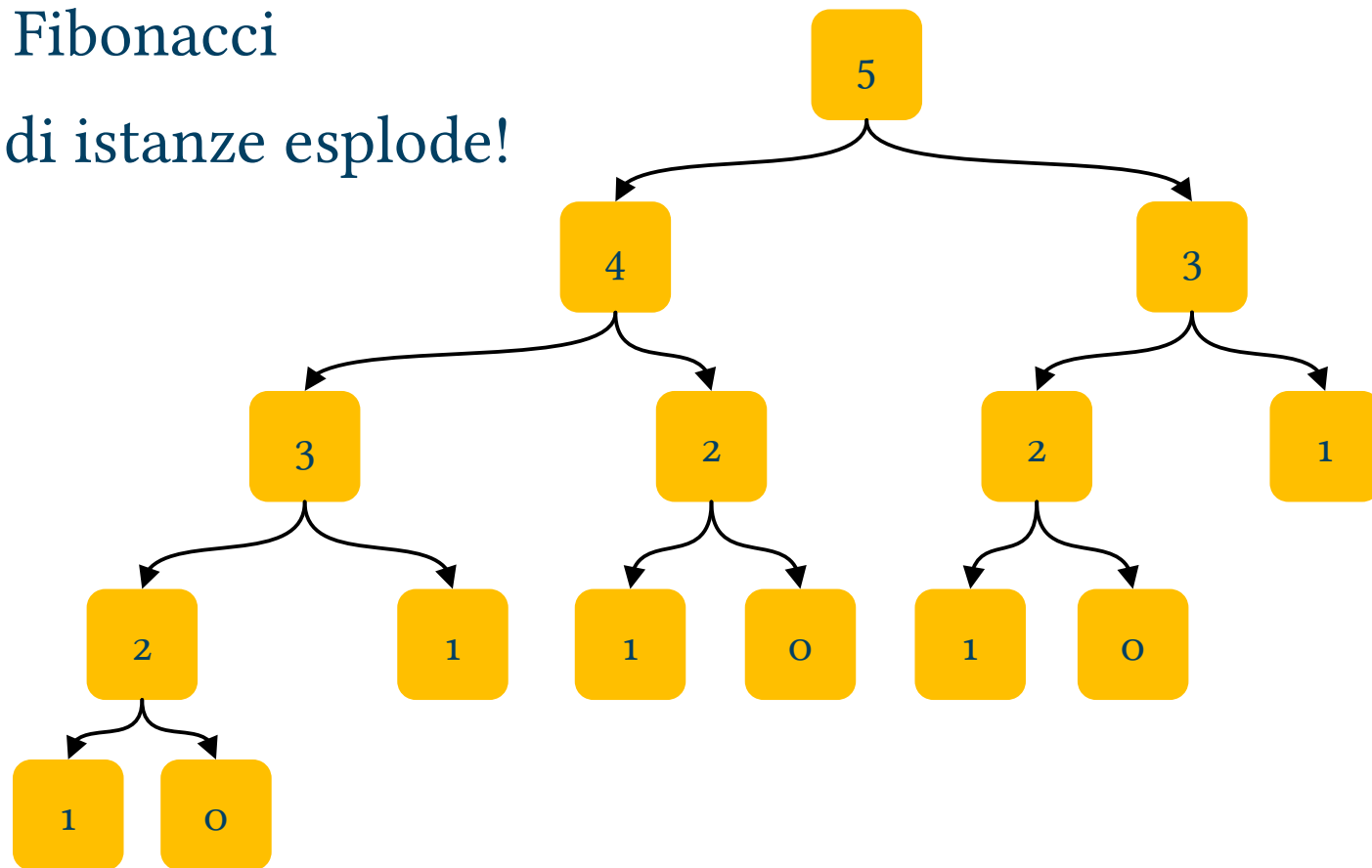
- Numeri di Fibonacci
 - 0, 1, 1, 2, 3, 5, 8, 13, 21

$$f(n) = \begin{cases} \text{per } n=0 & 0 \\ \text{per } n=1 & 1 \\ \text{per } n>1 & f(n-1) + f(n-2) \end{cases}$$



```
unsigned long fibo(unsigned long n){  
    if(n==0) return 0;  
    if(n==1) return 1;  
  
    return fibo(n-1)+fibo(n-2);  
}
```

- Numeri di Fibonacci
- Il numero di istanze esplode!



- L'implementazione ricorsiva vista in precedenza non è efficiente
- Si può dimostrare che se S è il tempo necessario al calcolo del numero di fibonacci di indice $n \rightarrow f(n)$, il calcolo di $f(n+1)$ richiede circa $1,6 \times S$
- Ad esempio se il calcolo di $f(n)$ richiede 1 s, il calcolo di $f(n+18)$ richiederà un'ora
- Soluzione \rightarrow dynamic programming (vedi esempio)

- Entrambe sono di fatto ripetizioni
 - Ciclo esplicito vs Ciclo implicito
 - Tutto ciò che posso risolvere con la ricorsione è risolvibile iterativamente
 - Entrambe vogliono condizione di uscita
 - Rischio ciclo infinito
- Ricorsione → codice più semplice
 - Spesso più facile e codice più compatto
 - Contro: overhead chiamate di funzione



UNIVERSITÀ DI PARMA

Le Funzioni



E Pluribus Unum

Autore Ignoto, Moretum, I sec. A.C.