PROGRAMMAZIONE 2: SPERIMENTAZIONI

Lezione 6 (parte 1) – Ricorsione in C

Agenda

- Introduzione
  - Esempio: il fattoriale
  - Esempio: la serie di Fibonacci
    - Ordine degli operatori
    - Complessità
- Complessità dei programmi ricorsivi
- La ricorsione rispetto all'iterazione

Video lezione disponibile su YouTube: https://youtu.be/NATfwPQzX1c

Introduzione

- Una **funzione ricorsiva chiama sé stessa** direttamente o indirettamente (attraverso un’altra funzione).
- Una funzione ricorsiva è chiamata per risolvere un problema e, in alcuni casi, è utile avere funzioni che chiamino sé stesse.

Introduzione

- Una funzione ricorsiva "sa", in realtà, risolvere soltanto casi semplici, cioè i cosiddetti **casi base**.
- Se la funzione è chiamata con un caso base, restituisce un risultato.

Introduzione

- Se la funzione è chiamata su un problema più complesso, solitamente divide il programma in due parti:
  1) una parte che sa risolvere (caso base);
  2) una parte che non sa risolvere (caso ricorsivo).
- Per rendere la ricorsione fattibile, la parte che la funzione non sa risolvere assomiglierà al problema originario, ma sarà una versione più semplice o più piccola.

Introduzione

- Semplificando (o riducendo) il problema da risolvere, la funzione chiamerà sé stessa su un problema minore.
- Questo passo è detto **chiamata ricorsiva** o **passo di ricorsione**.
- Ogni passo di ricorsione contiene anche un'istruzione di `return`, perché il risultato sarà combinato con la porzione del problema che la funzione "sa" risolvere, per ottenere un risultato che sarà restituito alla funzione chiamante originaria.

Introduzione

- Il passo di ricorsione viene eseguito mentre la chiamata originaria alla funzione si arresta, in attesa del risultato del passo di ricorsione.

Introduzione

- Perché termini la ricorsione, ogni volta che la funzione chiama sé stessa con una versione più semplice del problema originario, questa sequenza di problemi deve alla fine *convergere al caso base*.

Introduzione

- Quando la funzione riconosce il caso base, restituisce un risultato alla copia precedente della funzione e da qui segue una sequenza di restituzioni in cui si ripercorre *all'indietro* tutta la sequenza delle chiamate fino a che la chiamata originaria della funzione non restituisce il risultato finale.

Introduzione

Una funzione ricorsiva deve **sempre** restituire un valore.

Omettere il caso base o scrivere il passo di ricorsione cosicché non converga al caso base, provocherà una ricorsione infinita (come un ciclo infinito in una soluzione iterativa).

Esempio: il fattoriale

- Il fattoriale di un numero intero non negativo n, scritto come **n!** ("n fattoriale") è il prodotto

  `n · (n-1) · (n-2) · … · 1`

- Es.:
  - 0! → 1
  - 1! → 1
  - 2! → 2 · 1
  - 3! → 3 · 2 · 1
  - 4! → 4 · 3 · 2 · 1
  - 5! → 5 · 4 · 3 · 2 · 1
  - ...
  - 21! → 21 · 20 · 19 · 18 · ... · 1

Esempio: il fattoriale

- Calcolo **iterativo** con l'istruzione `for`.

```
unsigned long long int fatt(unsigned int number)
{
        fact = 1;
        for(c = number; c > = 1; --c)
        {
            fact = fact * c;
        }
        return fact;
}
```

Esempio: il fattoriale

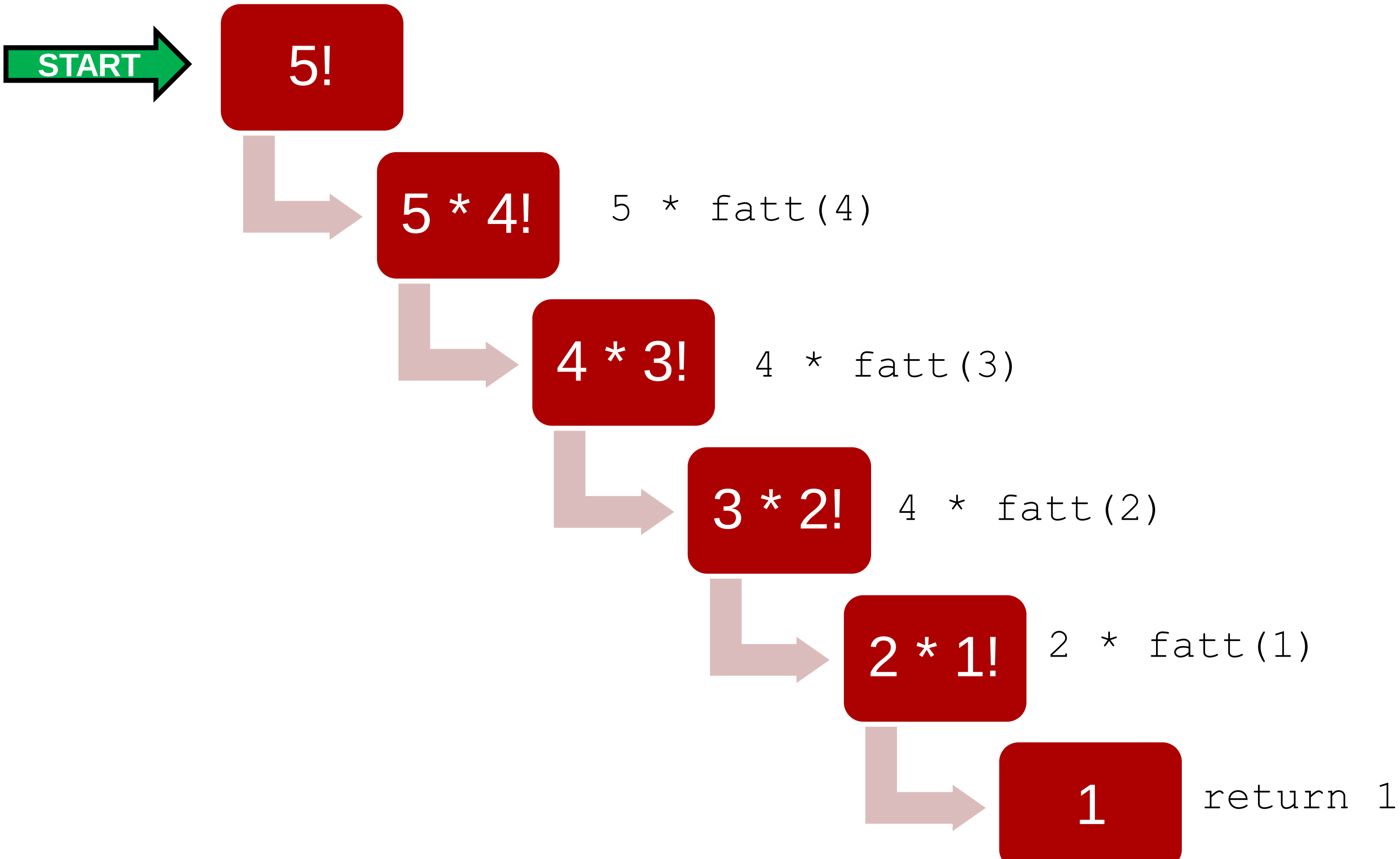
- Si può arrivare a una definizione ricorsiva della funzione fattoriale osservando la relazione:

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

- Ad esempio:
  - 5! è uguale a 5 * 4!
  - 4! è uguale a 4 * 3!
  - 3! è uguale a 3 * 2!
  - 2! è uguale a 2 * 1!
  - 1! è uguale a 1 (caso **base**)

Esempio: il fattoriale

Sequenza delle chiamate ricorsive



Esempio: il fattoriale

Valori restituiti da ogni chiamata ricorsiva

Valore finale **120** | 120

5 * 24 = **120**
è restituito alla funzione chiamante | 5 * 24

4 * 6 = **24**
è restituito alla funzione chiamante | 4 * 6

3 * 2 = **6**
è restituito alla funzione chiamante | 3 * 2

2 * 1 = **2**
è restituito alla funzione chiamante | 2 * 1

1
è restituito alla funzione chiamante | 1

START

Esempio: il fattoriale

es1.c

```
unsigned long long int fatt(unsigned int number)
{
    // caso base
    if (number <= 1)
    {
        return 1;
    }
    // caso ricorsivo
    else
    {
        return (number * fatt(number-1));
    }
}
```

Esempio: il fattoriale

- La funzione `fatt` verifica dapprima se una condizione di terminazione è vera e cioè se il `number` è minore o uguale a 1.
- Se `number` è davvero minore o uguale a 1, la funzione `fatt` restituisce 1, quindi non è necessaria ulteriore ricorsione.
- È stato quindi schematizzato il **caso base**.

Esempio: il fattoriale

- Se number è maggiore di 1, l'istruzione `return number * fatt(number-1)` esprime il problema come il prodotto di number e una chiamata ricorsiva a `fatt` che calcola il fattoriale di `number - 1`.
- La chiamata a `fatt(number -1)` è un problema più semplice dell'originario.
- È stato quindi schematizzato il **caso ricorsivo**.

Esempio: il fattoriale

- Come si può notare la funzione `fatt` riceve un `unsigned int` (intero senza segno `%u`) e restituisce un `unsigned long long int (%llu)`.
- Essendo `long int` definito su

  [−9.223.372.036.854.775.808
  → +9.223.372.036.854.775.807]
- La versione `unsigned` arriverà fino a 18.446.744.073.709.551.615.

Esempio: il fattoriale

DATA TYPE	MEMORIA (BYTE)	RANGE	FORMATO
`short int`	2	-32.768 to 32.767	%hd
`unsigned short int`	2	0 to 65.535	%hu
`unsigned int`	4	0 to 4.294.967.295	%u
`int`	4	-2.147.483.648 to 2.147.483.647	%d
`long int`	8	-2.147.483.648 to 2.147.483.647	%ld
`unsigned long int`	8	0 to 4.294.967.295	%lu
`long long int`	8	-(2^63) to (2^63)-1	%lld
`unsigned long long int`	8	0 to 18.446.744.073.709.551.615	%llu
`signed char`	1	-128 to 127	%c
`unsigned char`	1	0 to 255	%c
`float`	4		%f
`double`	8		%lf
`long double`	16		%Lf

Esempio: il fattoriale

es1.c

```c
#include <stdio.h>
#define N 21

unsigned long long int fatt(unsigned int number);

int main(void)
{
  unsigned int i;
  unsigned long long int f;

  printf("*** Calcolo dei fattoriali *** \n");

  for(i = 0; i <= N; i++)
  {
    f = fatt(i);
    printf("%u! = %llu\n", i, f);
  }
}
```

Esempio: il fattoriale

es1.c

```
*** Calcolo dei fattoriali ***
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000
21! = 14197454024290336768
```

Esempio: il fattoriale

- Come si può notare, anche usando come fattoriale `unsigned long long int` non è possibile calcolare fattoriali oltre al 21.
- Questo mette in evidenza uno dei limiti del C (e di vari linguaggi procedurali) nel senso che il linguaggio non può essere *esteso* con tipi specifici.
- Altri linguaggi, ad esempio quelli ad oggetti, (come C++ o Java) sono in genere linguaggi **estensibili** che permettono la creazione di nuovi tipi di dati per rappresentare ad esempio interi più grandi (il limite è legato all'architettura hardware).

Esempio: la serie di Fibonacci

- La serie di **Fibonacci**

0, 1, 1, 2, 3, 5, 8, 13, 21 …

- Inizia con 0 e 1 e prevede che **ogni successivo numero sia la somma dei due numeri precedenti**.

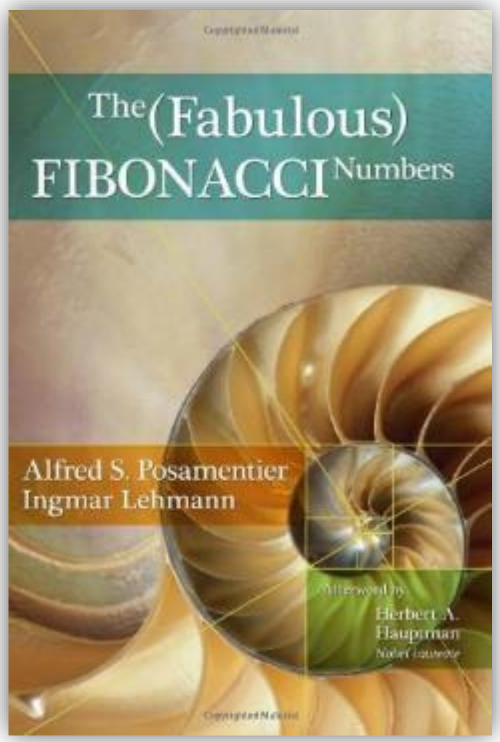
Leonardo Pisano detto il Fibonacci

Esempio: la serie di Fibonacci

- La serie di Fibonacci ricorre in natura e, in particolare, descrive una forma a spirale. Il rapporto dei numeri successivi di Fibonacci converge a un valore pari a 1,6180339887...
- Anche il numero 1,6180339887... ricorre ripetutamente in natura ed è chiamato **proporzione** (o **sezione**) **aurea**.
- La proporzione aurea si può trovare in vari aspetti del mondo reale.

Esempio: la serie di Fibonacci

- Per ulteriori approfondimenti:
  - **The Fabulous Fibonacci Numbers**
  - di Posamentier e Lehmann



Esempio: la serie di Fibonacci

- La serie di Fibonacci può essere definita ricorsivamente come segue:
- Fibonacci(0) = 0
- Fibonacci(1) = 1
- Fibonacci(2) = Fibonacci(1) + Fibonacci(0)
- Fibonacci(3) = Fibonacci(2) + Fibonacci(1)
- Fibonacci(4) = Fibonacci(3) + Fibonacci(2)
- ...
- Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)

Esempio: la serie di Fibonacci

- Come si potrà facilmente intuire, i numeri della serie tendono a diventare grandi velocemente.
- Pertanto è stato scelto:
  - `unsigned int` come numero su cui calcolare la serie;
  - `unsigned long long int` come valore di ritorno della funzione `fibonacci()`.

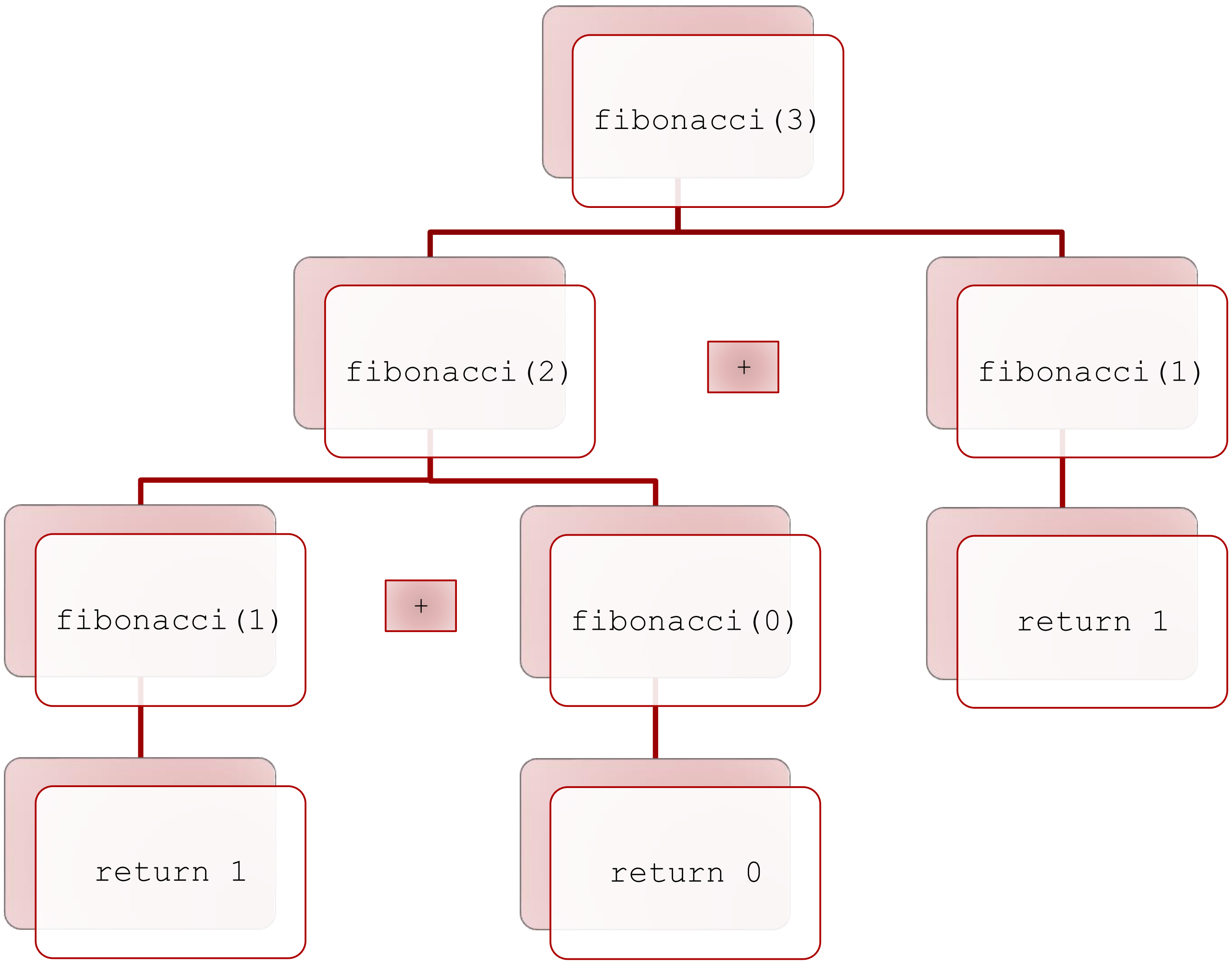
Esempio: la serie di Fibonacci

```
unsigned long long int fibonacci(unsigned int n)
{
  // caso base
  if (n==0 || n == 1)
  {
      return n; // o 0 o 1
  }
  // caso ricorsivo
  else
  {
      return fibonacci(n-1) + fibonacci(n-2);
  }
}
```

Esempio: la serie di Fibonacci

- Come si può notare nel codice precedente, ogni volta che la funzione `fibonacci` è richiamata, essa verifica il **caso base**, ovvero se `n` è uguale a 0 o 1. In caso affermativo, viene restituito `n` (quindi o 0 o 1).
- È interessante notare che, se `n` è maggiore di 1, il **caso ricorsivo** genera **due** chiamate ricorsive, ognuna delle quali risolve un problema leggermente più semplice dell'originario.

Esempio: la serie di Fibonacci



Ordine degli operatori

- Come si è potuto notare, il passo ricorsivo della funzione genera una somma tra due funzioni ricorsive.
- È bene sapere che, per ragioni di ottimizzazione, il C non specifica l'ordine in cui vengono *valutati* gli operandi degli operatori (+ incluso).
- Quindi non è possibile, ad esempio, stabilire nell'operazione

```
fibonacci(2) + fibonacci(1)
```

quale delle due funzioni venga eseguita prima.

Ordine degli operatori

- Il C specifica l'ordine di calcolo degli operandi di soli **quattro operatori**:
  1) `&&` (AND)
  2) `||` (OR)
  3) `,` (concatenazione)
  4) `?:` (condizionale, unico operatore ternario del C)
- Solo le operazioni svolte con gli operatori di cui sopra valutano gli operandi da **sinistra verso destra**.
  - es.:
    - `if (x && y)` → valuta prima la `x`
    - `if (f(1) || f(2))` → valuta prima la `f(1)`
    - `int x = 0, y = 10;` → valuta prima la `x`
    - `max = (a > b) ? a : b;` → valuta prima `(a > b)` ma non si sa se valuta prima `a` o `b`.

Ordine degli operatori

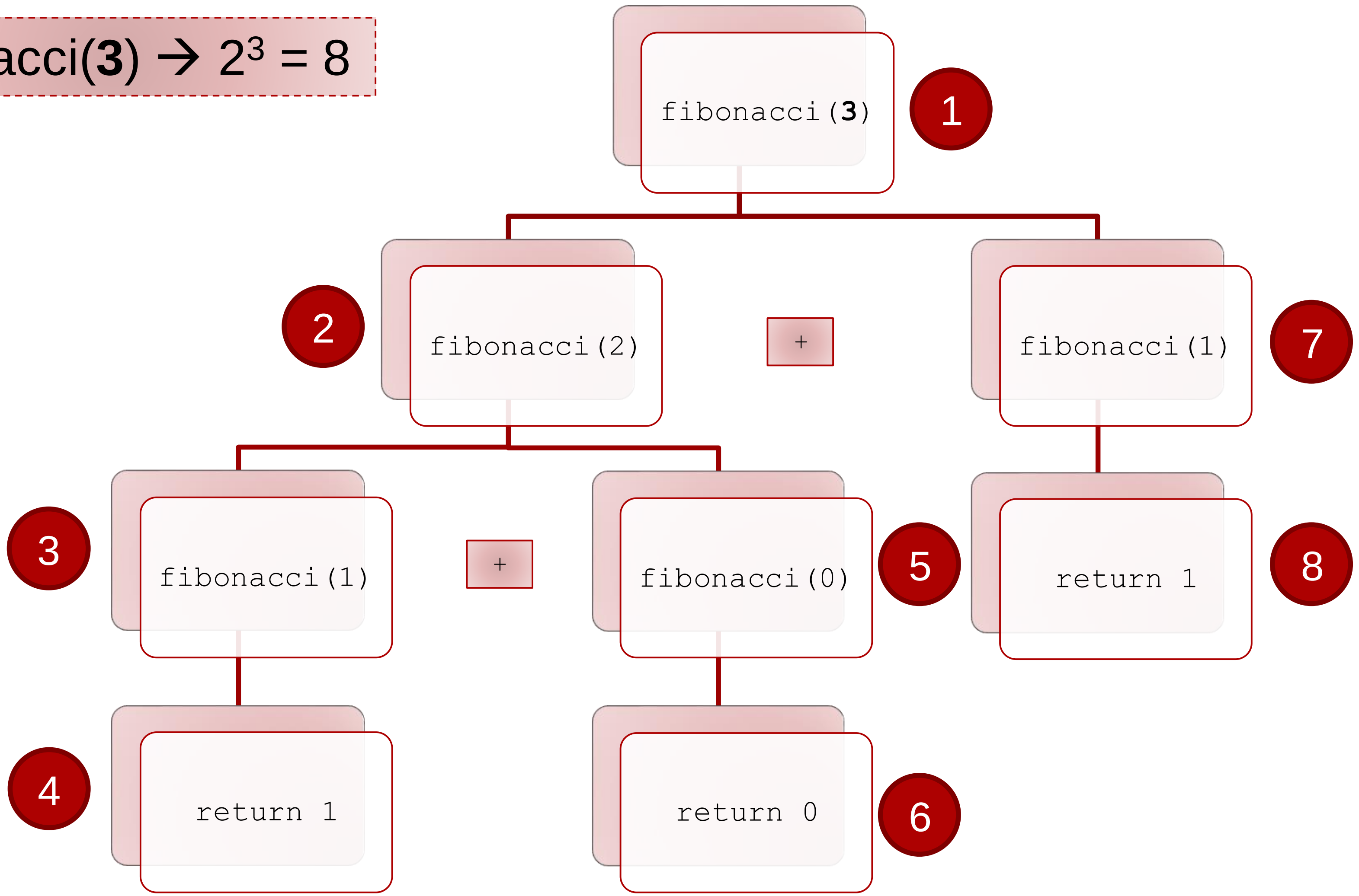
Scrivere programmi che dipendono dall'ordine di calcolo degli operandi può provocare errori, perché è possibile che i compilatori non calcolino gli operandi nell'ordine atteso.

Complessità

- È d'obbligo un avvertimento riguardo ai programmi ricorsivi come quello usato per generare i numeri di Fibonacci.
- Ogni livello di ricorsione nella funzione `fibonacci` ha un effetto di **raddoppio** sul numero delle chiamate.
- Il numero di chiamate ricorsive per calcolare l'ennesimo numero di Fibonacci è dell'ordine di $\mathbf{2^n}$ dove **2** è il numero di chiamate (costante), mentre **n** è il numero in input di cui si vuole il valore nella serie Fibonacci:
  - Es.:
    - n = 10 → 2^{10} → un migliaio di chiamate
    - n = 20 → 2^{20} → un milione di chiamate
    - n = 30 → 2^{30} → un miliardo di chiamate

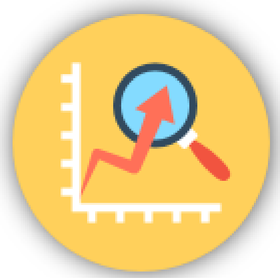
Complessità

fibonacci(**3**) → $2^3 = 8$



Complessità

- Com’è facilmente intuibile, problemi di questo genere rendono umili anche i computer più potenti.
- Gli *informatici* chiamano questo fenomeno **complessità esponenziale**.



- I problemi di complessità in generale e, la complessità esponenziale in particolare, sono esaminati nei vari corsi d'Informatica universitari riguardanti gli algoritmi.

La ricorsione rispetto all'iterazione

- Qualsiasi problema risolvibile in maniera ricorsiva si può anche risolvere in maniera iterativa.
- Il difetto della ricorsione è che invocando ripetutamente la stessa funzione, si crea un **aggravio di calcolo** (**overhead**). Ogni chiamata ricorsiva fa sì che venga creata **un'altra copia della funzione** (in realtà solo le variabili) portando ad un considerevole consumo di quantità di memoria. Ciò può essere dispendioso sia in termini di tempo di elaborazione che di spazio in memoria.
- Perché scegliere quindi la ricorsione?
  - Un approccio ricorsivo si preferisce normalmente ad un approccio iterativo quando rispecchia in maniera più naturale il problema e produce un programma più facile da capire e correggere.

La ricorsione rispetto all'iterazione

- Una buona ingegneria del software è importante, come lo sono le alte prestazioni.
- A volte i due obiettivi sono spesso in contrasto tra di loro; una buona ingegneria del software rende più facile e fattibile lo sviluppo e l'aggiornamento dei software grandi e complessi.
- Funzionalizzare i programmi (ovvero suddividerli in funzioni) favorisce una buona ingegneria del software, ma ha un prezzo: un programma molto *funzionalizzato* consuma tempo d'esecuzione.
- Programmi più *monolitici* (con poche funzioni) possono avere prestazioni migliori, ma saranno più difficili da scrivere e mantenere.

La ricorsione rispetto all'iterazione

- Le odierne architetture hardware sono ottimizzate per rendere efficienti le chiamate di funzioni.
- Gli odierni processori (CPU) sono sempre più veloci.
- Per la maggior parte delle applicazioni e dei sistemi software che si andranno a costruire, concentrarsi su una buona ingegneria del software sarà più importante che ottenere elevate prestazioni.

FINE PRESENTAZIONE