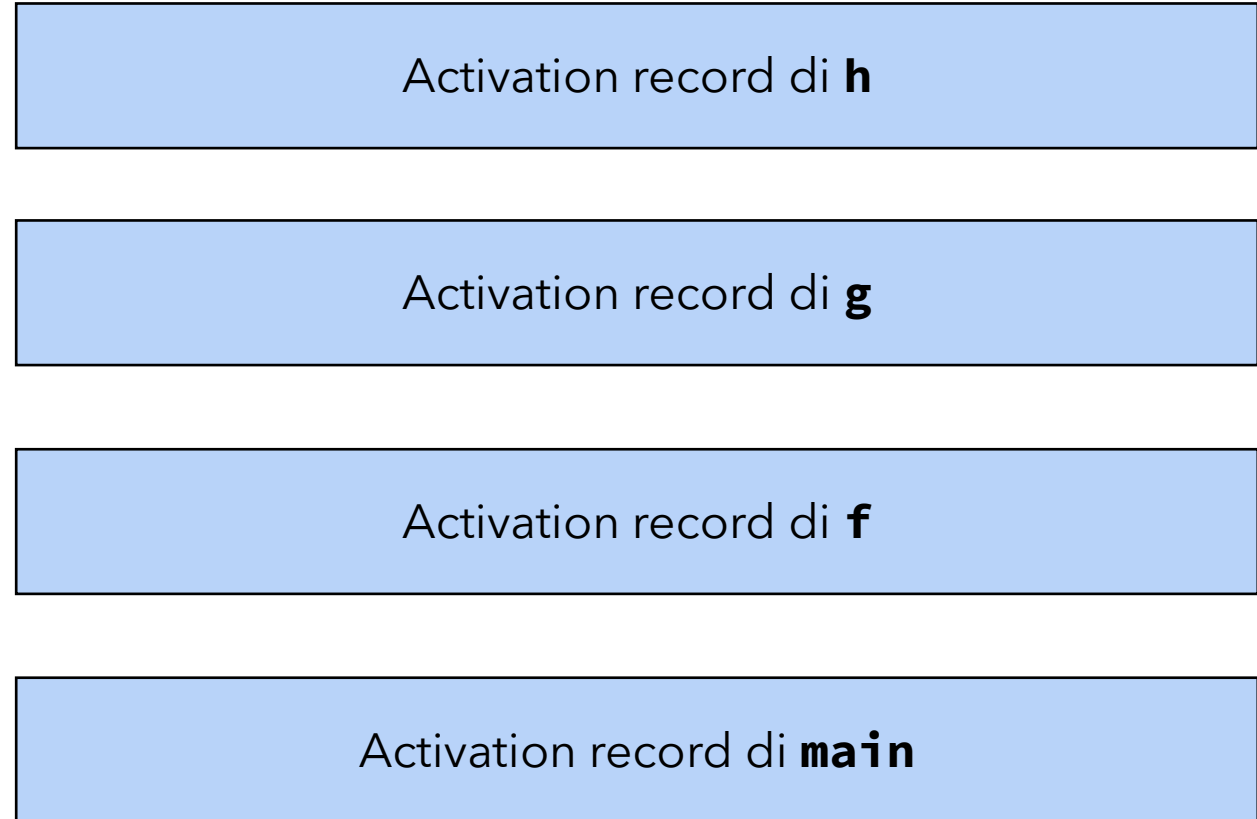


# Call stack Ricorsione

**Liceo G.B. Brocchi - Bassano del Grappa (VI)**  
**Liceo Scientifico - opzione scienze applicate**  
Giovanni Mazzocchin

# Chiamata di funzione

```
double h (double d) {  
    return d * 2;  
}  
  
void g () {  
    double d = h(3.14);  
    return;  
}  
  
int f (int n) {  
    int x = 0;  
    g();  
    return x;  
}  
  
int main () {  
    int val = f(5);  
    return 0;  
}
```



# Call stack (pila delle chiamate)

- Ad ogni chiamata di funzione viene aggiunto un *activation record* (aka *stack frame*) in un'area di memoria organizzata come **stack** (simile ad una pila di piatti: l'inserimento e la rimozione di un elemento avvengono solo in cima alla pila)
- Gli *activation record* contengono le **variabili locali** della funzione (tra cui i **parametri formali**) e altre informazioni necessarie per restituire il controllo al chiamante
- L'istruzione **return** all'interno di una funzione provoca la rimozione dell'*activation record* della stessa

# Call stack (pila delle chiamate)

- il programma viene lanciato da una shell

Activation record di **main**

**val**

**informazioni per ritornare alla shell**

# Call stack (pila delle chiamate)

- **main** invoca **f**

Activation record di **f**

**n**

**x**

**informazioni per ritornare a main**

Activation record di **main**

**val**

**informazioni per ritornare alla shell**

# Call stack (pila delle chiamate)

- **f** invoca **g**

Activation record di **g**

**d**      **informazioni per ritornare a f**

Activation record di **f**

**n**      **x**      **informazioni per ritornare a main**

Activation record di **main**

**val**      **informazioni per ritornare alla shell**

# Call stack (pila delle chiamate)

- **g** invoca **h**

Activation record di **h**

**d**      **informazioni per ritornare a g**

Activation record di **g**

**d**      **informazioni per ritornare a f**

Activation record di **f**

**n**      **x**      **informazioni per ritornare a main**

Activation record di **main**

**val**      **informazioni per ritornare alla shell**

# Call stack (pila delle chiamate)

- **h** esegue l'istruzione **return**

Activation record di **h**

**d**      **informazioni per ritornare a g**

Activation record di **g**

**d**      **informazioni per ritornare a f**

Activation record di **f**

**n**      **x**      **informazioni per ritornare a main**

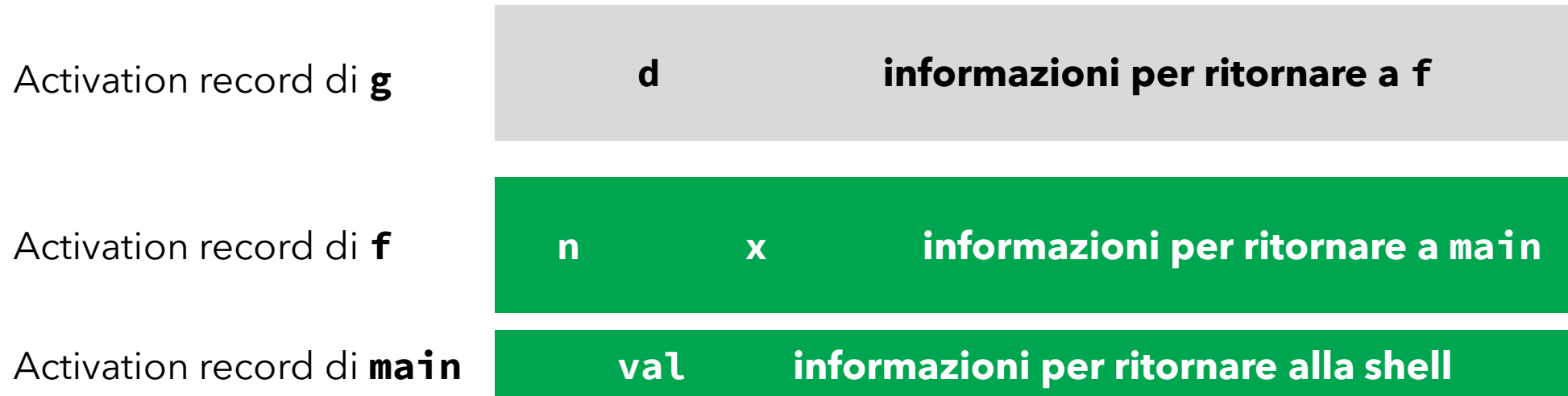
Activation record di **main**

**val**      **informazioni per ritornare alla shell**



# Call stack (pila delle chiamate)

- **g** esegue l'istruzione **return**



# Call stack (pila delle chiamate)

- **f** esegue l'istruzione **return**

Activation record di **f**

**n**

**x**

**informazioni per ritornare a main**

Activation record di **main**

**val**

**informazioni per ritornare alla shell**

# Call stack (pila delle chiamate)

- **f** esegue l'istruzione **return**

Activation record di **main**

**val**

**informazioni per ritornare alla shell**

# Call stack (pila delle chiamate)

- Il flusso di controllo ritorna al **main**, in particolare all'istruzione successiva alla chiamata di **f**, che è un'assegnazione


```
int val = f(5);
```

# Funzioni ricorsive

- **Fattoriale** – definizione *iterativa*

$$n! = n \cdot \underbrace{(n - 1) \cdot (n - 2) \cdot (n - 3) \dots 2 \cdot 1}$$

- **Fattoriale** – definizione *ricorsiva*

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


- Spiegare cosa c'è di «ricorsivo» nell'ultima definizione
- **NB:**  $0! = 1$

# Funzioni ricorsive

```
int fact_iterative (int n) {  
    if (n < 0) {  
        return -1;  
    }  
    int result = 1;  
    for (int i = 1; i <= n; i++) {  
        result = result * i;  
    }  
    return result;  
}
```

```
int fact_recursive (int n) {  
    if (n < 0) {  
        return -1;  
    }  
    if (n == 0) {  
        return 1;  
    }  
    return n * fact_recursive(n - 1);  
}
```



chiamata ricorsiva

qual è la versione più efficiente in termini di tempo e occupazione di memoria?

# Funzioni ricorsive

```
int fact_iterative(int n) {  
    if (n < 0) {  
        return -1;  
    }  
    int result = 1;  
    for (int i = 1; i <= n; i++) {  
        result = result * i;  
    }  
    return result;  
}
```

```
int fact_recursive(int n) {  
    if (n < 0) {  
        return -1;  
    }  
    if (n == 0) {  
        return 1;  
    }  
    int rec_call_result = fact_recursive(n - 1);  
    return n * rec_call_result;  
}
```

Questo tipo di ricorsione è detto **head recursion** (ricorsione in testa). Prima si invoca la funzione ricorsivamente, e poi si fanno i calcoli tornando indietro (molto inefficiente in questo caso)

**Chiamata ricorsiva.**  
Utilizzando una variabile si capisce ancora meglio che le moltiplicazioni vengono fatte tornando indietro. La moltiplicazione è scritta proprio dopo la chiamata ricorsiva!

# Funzioni ricorsive – call stack a runtime

```
int fact_recursive(int n) {  
    if (n < 0) {  
        return -1;  
    }  
  
    if (n == 0) {  
        return 1;  
    }  
  
    int rec_call_result = fact_recursive(n - 1);  
    return n * rec_call_result;  
}
```

**vediamo come cambia lo stack partendo da questo esempio. Ogni stack frame avrà al suo interno la variabile locale `rec_call_result` e il parametro `n`**



# Funzioni ricorsive – call stack a runtime

non viene assegnato alcun valore a `rec_call_result` fintantoché il flusso di controllo non arriva al caso base

se a destra di un'assegnazione c'è un'invocazione ricorsiva, allora prima di tutto il controllo passa di nuovo alla prima istruzione della funzione

```
n: 6  
rec_call_result = fact_recursive(5)
```

# Funzioni ricorsive – call stack a runtime

```
n: 5  
rec_call_result = fact_recursive(4)
```

```
n: 6  
rec_call_result = fact_recursive(5)
```

# Funzioni ricorsive – call stack a runtime

```
n: 4  
rec_call_result = fact_recursive(3)
```

```
n: 5  
rec_call_result = fact_recursive(4)
```

```
n: 6  
rec_call_result = fact_recursive(5)
```

# Funzioni ricorsive – call stack a runtime

```
n: 3  
rec_call_result = fact_recursive(2)
```

```
n: 4  
rec_call_result = fact_recursive(3)
```

```
n: 5  
rec_call_result = fact_recursive(4)
```

```
n: 6  
rec_call_result = fact_recursive(5)
```

# Funzioni ricorsive – call stack a runtime

```
        n: 2  
rec_call_result = fact_recursive(1)
```

```
        n: 3  
rec_call_result = fact_recursive(2)
```

```
        n: 4  
rec_call_result = fact_recursive(3)
```

```
        n: 5  
rec_call_result = fact_recursive(4)
```

```
        n: 6  
rec_call_result = fact_recursive(5)
```

# Funzioni ricorsive – call stack a runtime

```
n: 1  
rec_call_result = fact_recursive(0)
```

```
n: 2  
rec_call_result = fact_recursive(1)
```

```
n: 3  
rec_call_result = fact_recursive(2)
```

```
n: 4  
rec_call_result = fact_recursive(3)
```

```
n: 5  
rec_call_result = fact_recursive(4)
```

```
n: 6  
rec_call_result = fact_recursive(5)
```

# Funzioni ricorsive – call stack a runtime

n: 0  
return 1

n: 1  
rec\_call\_result = fact\_recursive(0)

n: 2  
rec\_call\_result = fact\_recursive(1)

n: 3  
rec\_call\_result = fact\_recursive(2)

n: 4  
rec\_call\_result = fact\_recursive(3)

n: 5  
rec\_call\_result = fact\_recursive(4)

n: 6  
rec\_call\_result = fact\_recursive(5)

# Funzioni ricorsive – call stack a runtime

n: 0  
return 1

n: 1  
rec\_call\_result = fact\_recursive(0)

n: 2  
rec\_call\_result = fact\_recursive(1)

n: 3  
rec\_call\_result = fact\_recursive(2)

n: 4  
rec\_call\_result = fact\_recursive(3)

n: 5  
rec\_call\_result = fact\_recursive(4)

n: 6  
rec\_call\_result = fact\_recursive(5)



# Funzioni ricorsive – call stack a runtime

```
n: 1  
rec_call_result = fact_recursive(0): 1  
restituisce 1*1=1 al chiamante
```

```
n: 2  
rec_call_result = fact_recursive(1)
```

```
n: 3  
rec_call_result = fact_recursive(2)
```

```
n: 4  
rec_call_result = fact_recursive(3)
```

```
n: 5  
rec_call_result = fact_recursive(4)
```

```
n: 6  
rec_call_result = fact_recursive(5)
```

# Funzioni ricorsive – call stack a runtime

**ATTENZIONE:** le istruzioni successive alla chiamata ricorsiva sono:

- l'assegnazione a `rec_call_result`
- `return n*rec_call_result`

**Quando avete chiaro questo concetto, avete capito tutto**

```
n: 2
rec_call_result = 1
restituisce 2*1=2 al chiamante
```

```
n: 3
rec_call_result = fact_recursive(2)
```

```
n: 4
rec_call_result = fact_recursive(3)
```

```
n: 5
rec_call_result = fact_recursive(4)
```

```
n: 6
rec_call_result = fact_recursive(5)
```

# Funzioni ricorsive – call stack a runtime

n: 3  
rec\_call\_result = fact\_recursive(2): 2  
restituisce  $3 \times 2 = 6$  al chiamante

n: 4  
rec\_call\_result = fact\_recursive(3)

n: 5  
rec\_call\_result = fact\_recursive(4)

n: 6  
rec\_call\_result = fact\_recursive(5)

# Funzioni ricorsive – call stack a runtime

```
n: 4  
rec_call_result = fact_recursive(3): 6  
  restituisce  $6 \times 4 = 24$  al chiamante
```

```
n: 5  
rec_call_result = fact_recursive(4)
```

```
n: 6  
rec_call_result = fact_recursive(5)
```

# Funzioni ricorsive – call stack a runtime

```
n: 5  
rec_call_result = fact_recursive(4): 24  
restituisce 5*24=120 al chiamante
```

```
n: 6  
rec_call_result = fact_recursive(5)
```

# Funzioni ricorsive – call stack a runtime

```
        n: 6  
rec_call_result = fact_recursive(5): 120  
    restituisce 6*120=720 al chiamante
```

# Funzioni ricorsive – call stack a runtime

- Lo stack delle chiamate a `fact_recursive` è stato svuotato, e il chiamante (`main`) si ritrova in mano il valore giusto, ossia  $6! = 720$
- Si vede bene che questo modo di calcolare è estremamente inefficiente a livello di tempo di esecuzione: abbiamo dovuto aspettare un bel po' prima di fare anche soltanto  $1 \cdot 1$
- Anche in termini di spazio occupato in memoria questa implementazione è un disastro: se viene chiesto il fattoriale di un numero molto grande, il numero di chiamate molto probabilmente farà traboccare lo stack (**stack overflow**)

# Funzioni ricorsive – call stack a runtime

- La dimensione dello stack di un programma non è infinita
- Se si va oltre un certo numero di chiamate (che dipende dal sistema hardware e software su cui viene eseguito il programma), si può provocare uno **stack overflow** (traboccamento dello stack)
- Proviamo a vedere cosa succede con questa funzione ricorsiva, nella quale il caso base non viene mai raggiunto. Il programma va avanti all'infinito? Teoricamente dovrebbe andare avanti all'infinito, in pratica **crasha** perché occupa troppa memoria

```
void bad_rec_func() {  
    if (1 == 0) {  
        return;  
    }  
    return bad_rec_func();  
}
```



# Ricorsione infinita e *stack overflow*

```
cyofanni@LAPTOP-I0S1RKRC:~/Desktop/high-school-cs-class/cpp_lectures$ ./recursion  
Segmentation fault (core dumped)
```

`Segmentation fault` significa «Errore di segmentazione»: è un messaggio dato dal sistema operativo per segnalare che un programma ha provato ad accedere ad aree di memoria a cui non doveva accedere.

Vi starete chiedendo cosa significa «segmentazione»: lo vedremo più avanti.

Nei sistemi GNU/Linux, `Segmentation fault` è quasi un sinonimo di «crash del programma» (definizione non esatta, ma utile per iniziare a capirci qualcosa)

# Altro caso di Segmentation fault

```
char str[] = "hello, world";  
for (int i = 0; i < 1000; i++) {  
    str[i] = 'q';  
}
```

**str è un array di 13 caratteri, ma stiamo provando ad accedere a indici ben superiori a 12**

```
cyofanni@LAPTOP-I0S1RKRC:~/Desktop/high-school-cs-class/cpp_lectures$ g++ -o recursion recursion.cpp  
cyofanni@LAPTOP-I0S1RKRC:~/Desktop/high-school-cs-class/cpp_lectures$ ./recursion  
*** stack smashing detected ***: terminated  
Aborted (core dumped)  
cyofanni@LAPTOP-I0S1RKRC:~/Desktop/high-school-cs-class/cpp_lectures$ |
```

# Funzioni ricorsive

- **Potenza** – definizione *iterativa*

$$n^0 = 1$$
$$n^m = \underbrace{n \cdot n \cdot n \cdot n \dots}_{(m \text{ volte, con } m \text{ naturale diverso da } 0)}$$

- **Potenza** – definizione *ricorsiva*

$$n^m = n \cdot n^{m-1}$$

- Spiegare cosa c'è di «ricorsivo» nell'ultima definizione e scrivere una funzione che la implementa

# Fibonacci - ricorsivamente

$$f_0 = 0$$

$$f_1 = 1$$

$$f_n = f_{n-1} + f_{n-2}$$

primi termini della successione: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

**qual è la differenza rispetto alle funzioni  
ricorsive precedenti?**

# Fibonacci - ricorsivamente

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

---

```
unsigned long long fibonacci(unsigned int n) {  
    if (n == 0 || n == 1) {  
        return n;  
    }  
    unsigned long long fib_n_1 = fibonacci(n - 1);  
    unsigned long long fib_n_2 = fibonacci(n - 2);  
    return fib_n_1 + fib_n_2;  
}
```

**salviamo i risultati delle 2  
chiamate ricorsive in 2  
variabili**

# Fibonacci sullo stack

```
n: 4  
fib_n_1 = fibonacci(3)
```

# Fibonacci sullo stack

```
        n: 3  
fib_n_1 = fibonacci(2)
```

```
        n: 4  
fib_n_1 = fibonacci(3)
```

# Fibonacci sullo stack

```
        n: 2  
fib_n_1 = fibonacci(1)
```

```
        n: 3  
fib_n_1 = fibonacci(2)
```

```
        n: 4  
fib_n_1 = fibonacci(3)
```



# Fibonacci sullo stack

**n: 1**  
**restituisce 1**

**n: 2**  
**fib\_n\_1 = fibonacci(1)**

**n: 3**  
**fib\_n\_1 = fibonacci(2)**

**n: 4**  
**fib\_n\_1 = fibonacci(3)**

# Fibonacci sullo stack

**n: 1**  
**restituisce 1**

**n: 2**  
**fib\_n\_1 = fibonacci(1)**

**n: 3**  
**fib\_n\_1 = fibonacci(2)**

**n: 4**  
**fib\_n\_1 = fibonacci(3)**

# Fibonacci sullo stack

```
        n: 2  
fib_n_1 = fibonacci(1): 1  
fib_n_2 = fibonacci(0)
```

```
        n: 3  
fib_n_1 = fibonacci(2)
```

```
        n: 4  
fib_n_1 = fibonacci(3)
```

# Fibonacci sullo stack

```
n: 0  
restituisce 0
```

```
n: 2  
fib_n_1 = fibonacci(1): 1  
fib_n_2 = fibonacci(0)
```

```
n: 3  
fib_n_1 = fibonacci(2)
```

```
n: 4  
fib_n_1 = fibonacci(3)
```

# Fibonacci sullo stack

**n: 0**  
restituisce 0

**n: 2**  
**fib\_n\_1 = fibonacci(1): 1**  
**fib\_n\_2 = fibonacci(0)**

**n: 3**  
**fib\_n\_1 = fibonacci(2)**

**n: 4**  
**fib\_n\_1 = fibonacci(3)**

# Fibonacci sullo stack

```
        n: 2  
    fib_n_1 = fibonacci(1): 1  
    fib_n_2 = fibonacci(0): 0  
    restituisce fib_n_1 + fib_n_2: 1 + 0: 1
```

```
        n: 3  
    fib_n_1 = fibonacci(2)
```

```
        n: 4  
    fib_n_1 = fibonacci(3)
```

# Fibonacci sullo stack

```
        n: 2  
    fib_n_1 = fibonacci(1): 1  
    fib_n_2 = fibonacci(0): 0  
    restituisce fib_n_1 + fib_n_2: 1 + 0: 1
```

```
        n: 3  
    fib_n_1 = fibonacci(2)
```

```
        n: 4  
    fib_n_1 = fibonacci(3)
```

# Fibonacci sullo stack

```
        n: 3  
fib_n_1 = fibonacci(2): 1  
  fib_n_2 = fibonacci(1)
```

```
        n: 4  
fib_n_1 = fibonacci(3)
```



# Fibonacci sullo stack

```
n: 1  
restituisce 1
```

```
n: 3  
fib_n_1 = fibonacci(2): 1  
fib_n_2 = fibonacci(1)
```

```
n: 4  
fib_n_1 = fibonacci(3)
```

# Fibonacci sullo stack

```
n: 1  
restituisce 1
```

```
n: 3  
fib_n_1 = fibonacci(2): 1  
fib_n_2 = fibonacci(1)
```

```
n: 4  
fib_n_1 = fibonacci(3)
```

# Fibonacci sullo stack

```
        n: 3  
    fib_n_1 = fibonacci(2): 1  
    fib_n_2 = fibonacci(1): 1  
    restituisce fib_n_1 + fib_n_2: 1 + 1: 2
```

```
        n: 4  
    fib_n_1 = fibonacci(3)
```

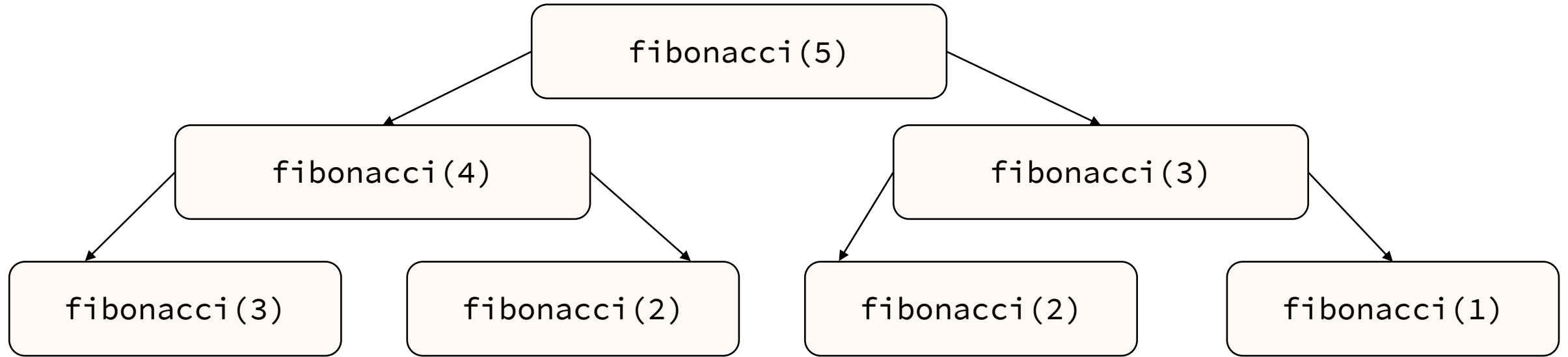
# Fibonacci sullo stack

```
        n: 3  
    fib_n_1 = fibonacci(2): 1  
    fib_n_2 = fibonacci(1): 1  
    restituisce fib_n_1 + fib_n_2: 1 + 1: 2
```

```
        n: 4  
    fib_n_1 = fibonacci(3)
```

e così via...  
si noti l'inefficienza del calcolo

# Fibonacci: albero delle chiamate



completare l'albero per esercizio

# Da vedere a casa

- [Fibonacci Mystery - Numberphile](#)