

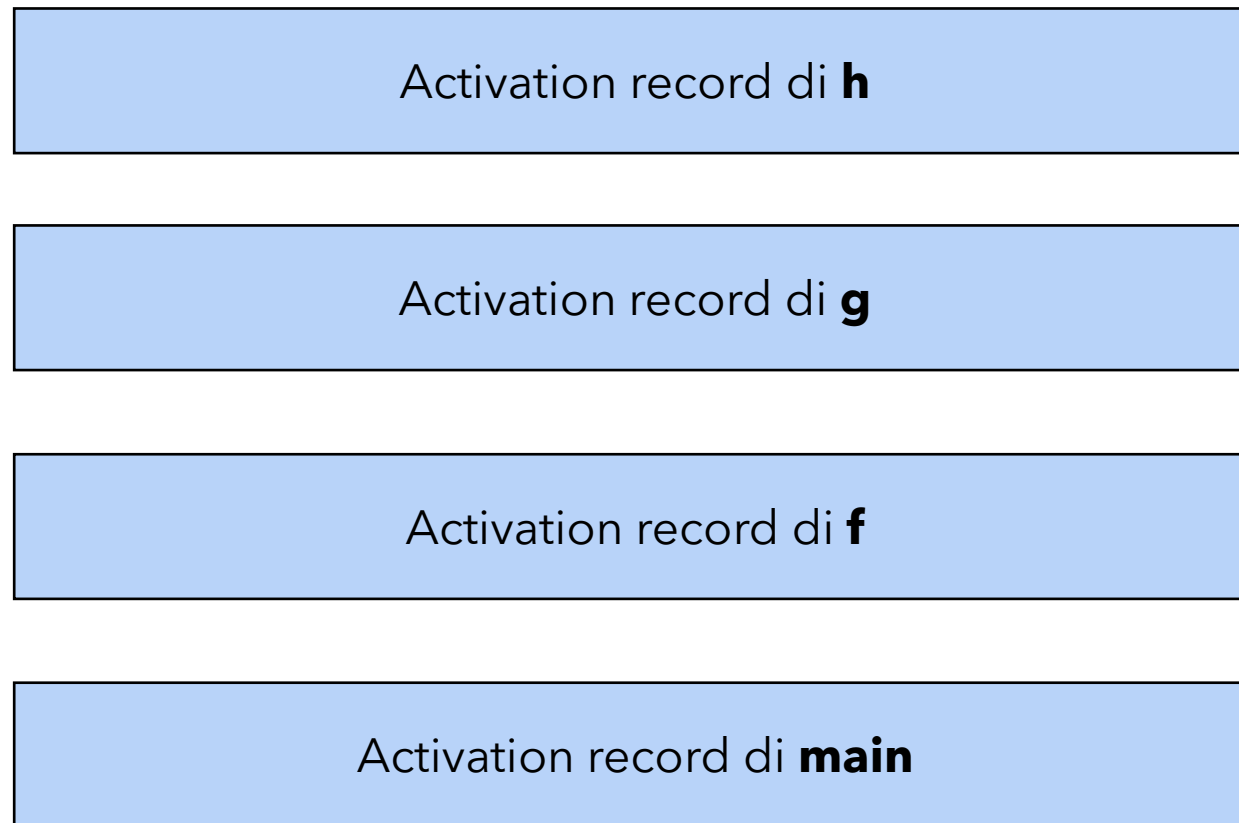
Chiamata di funzione

Ricorsione

Liceo G.B. Brocchi
Classi seconde Scientifico - opzione scienze applicate
Bassano del Grappa, Gennaio 2023

Chiamata di funzione (implementazione)

```
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

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 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 g**

Activation record di **g**

d **informazioni per ritornare 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 **g**

Activation record di **g**

d

informazioni per ritornare f

Activation record di **f**

n

x

informazioni per ritornare a main

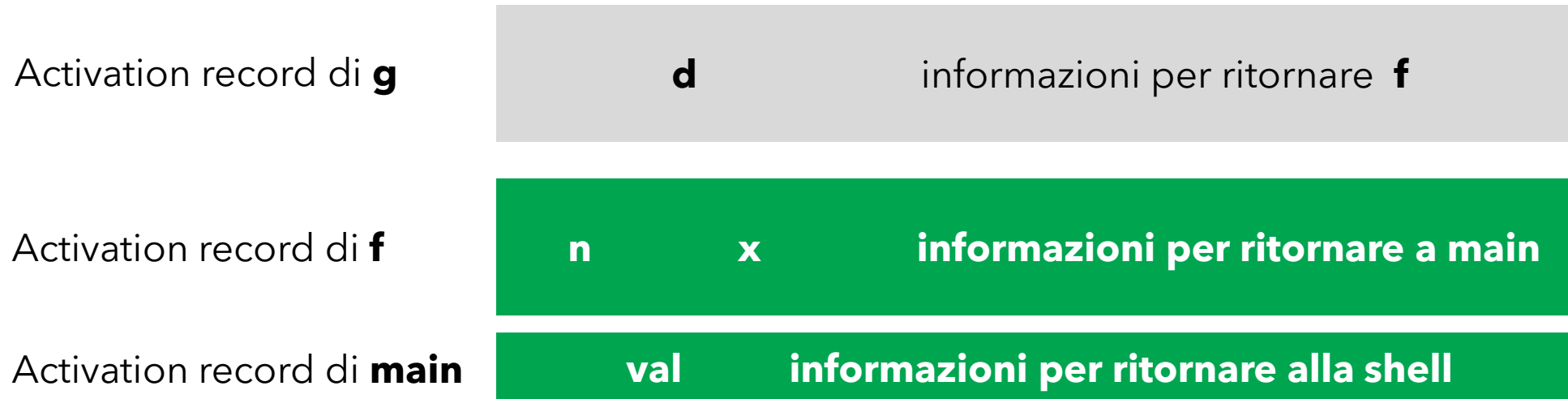
Activation record di **main**

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**

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

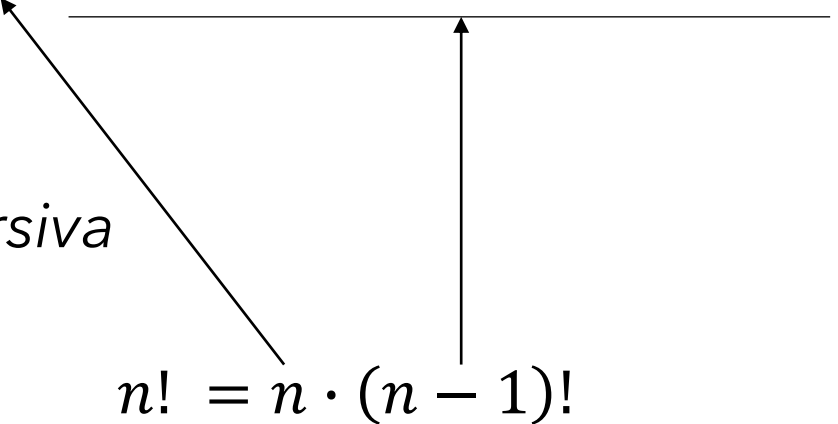
- **Curiosità:** da chi viene invocata la funzione **main**?

Funzioni ricorsive

- **Fattoriale** - definizione *iterativa*

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdot (n - 3) \cdot 2 \cdot 1$$

- **Fattoriale** - definizione *ricorsiva*

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


- **Domanda**: 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 memoria?

Funzioni ricorsive

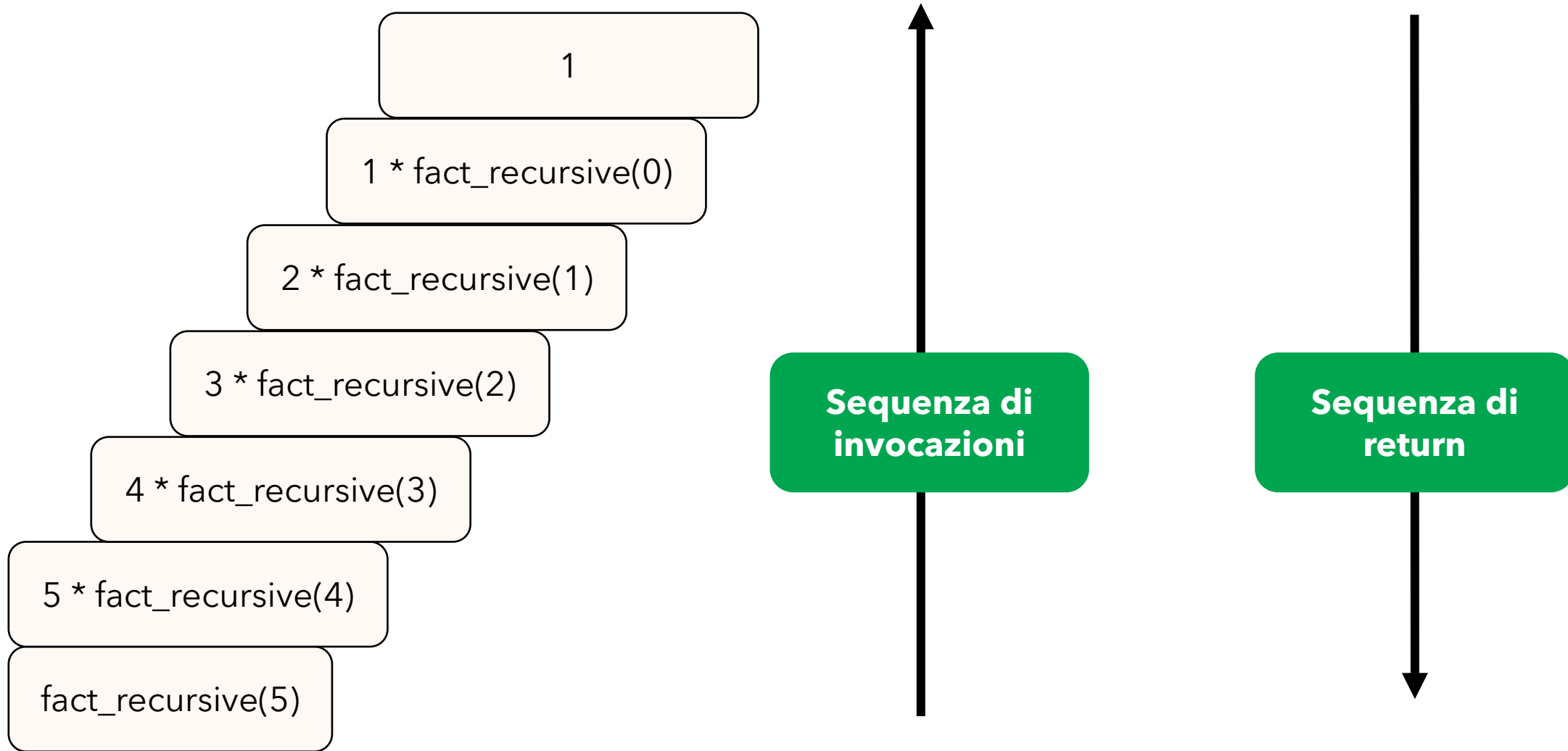
```
int fact_iterative (int n) {  
    if (n < 0) {  
        return -1;  
    }  
    int result = 1;  
    for (int i = 1; i <= n; i++) {  
        result = result * i;  
    }  
    return result;  
}
```

```
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;  
}
```

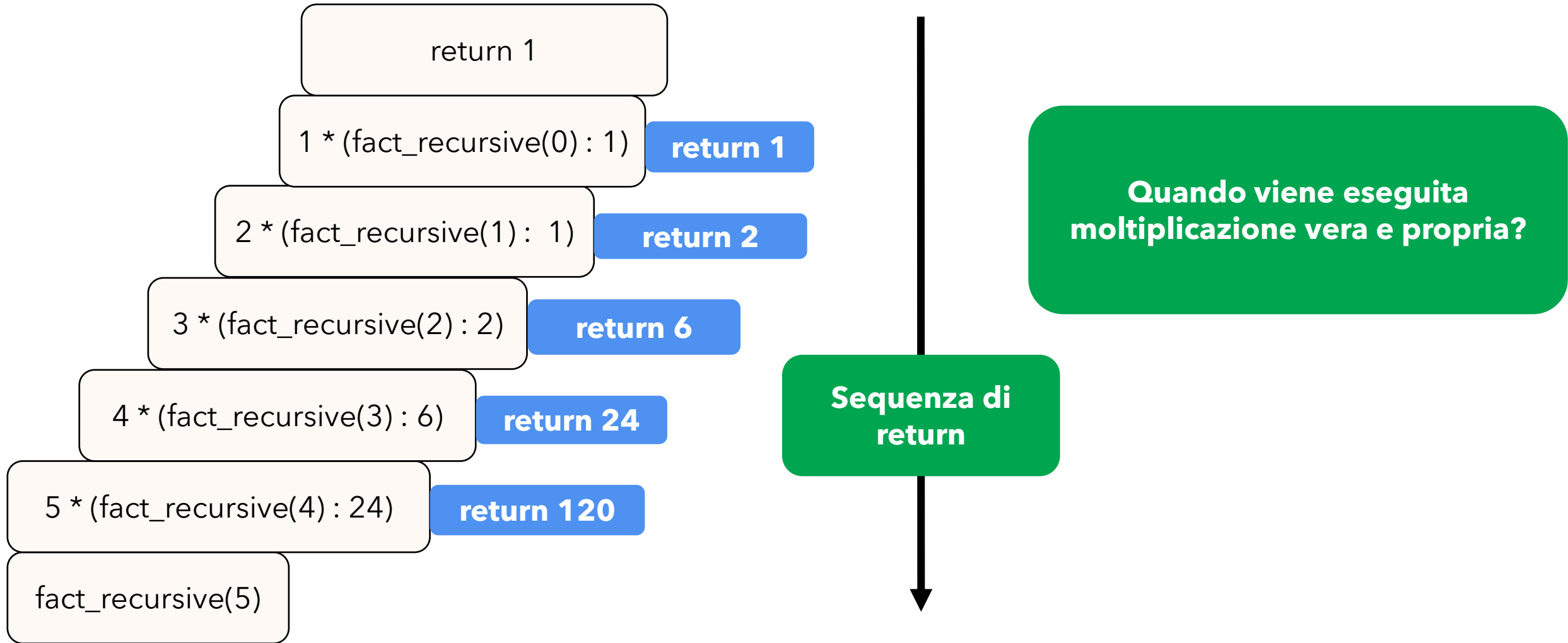
Questo tipo di ricorsione è detto **head recursion** (*ricorsione in testa*).
Prima si invoca la funzione ricorsivamente, e poi si fanno i calcoli tornando indietro (molto inefficiente)

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



Funzioni ricorsive



Funzioni ricorsive – come cambia lo stack a runtime

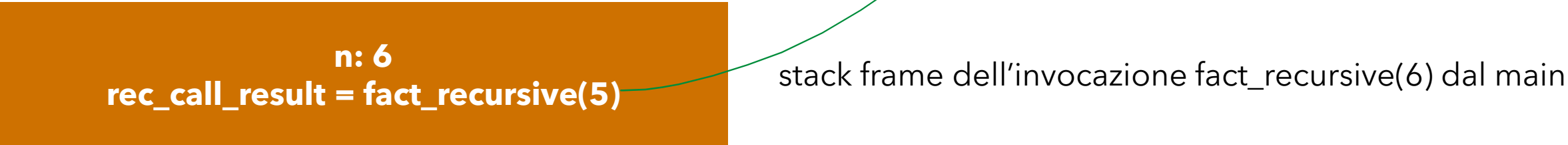
```
int fact_recursive (int n) {  
    if (n < 0) {  
        return -1;  
    }  
  
    if (n == 0) {  
        return 1;  
    }  
  
    int rec_call_result = fact_recursive(n - 1);  
    return n * rec_call_result;  
}  
  
int main () {  
    fact_recursive (6) ;  
}
```

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 – evoluzione del 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)

stack frame dell'invocazione `fact_recursive(6)` dal main

Funzioni ricorsive – come cambia lo stack a runtime

n: 5
rec_call_result = fact_recursive(4)

stack frame dell'invocazione ricorsiva fact_recursive(5)
nessun calcolo finora

n: 6
rec_call_result = fact_recursive(5)

stack frame dell'invocazione fact_recursive(6) dal main
nessun calcolo finora

Funzioni ricorsive – come cambia lo stack a runtime

n: 4
rec_call_result = fact_recursive(3)

stack frame dell'invocazione ricorsiva fact_recursive(4)
nessun calcolo finora

n: 5
rec_call_result = fact_recursive(4)

stack frame dell'invocazione ricorsiva fact_recursive(5)
nessun calcolo finora

n: 6
rec_call_result = fact_recursive(5)

stack frame dell'invocazione fact_recursive(6) dal main
nessun calcolo finora

Funzioni ricorsive – come cambia lo stack a runtime

n: 3
rec_call_result = fact_recursive(2)

stack frame dell'invocazione ricorsiva fact_recursive(3)
nessun calcolo finora

n: 4
rec_call_result = fact_recursive(3)

stack frame dell'invocazione ricorsiva fact_recursive(4)
nessun calcolo finora

n: 5
rec_call_result = fact_recursive(4)

stack frame dell'invocazione ricorsiva fact_recursive(5)
nessun calcolo finora

n: 6
rec_call_result = fact_recursive(5)

stack frame dell'invocazione fact_recursive(6) dal main
nessun calcolo finora

Funzioni ricorsive – come cambia lo stack a runtime

n: 2
rec_call_result = fact_recursive(1)

stack frame dell'invocazione ricorsiva fact_recursive(2)
nessun calcolo finora

n: 3
rec_call_result = fact_recursive(2)

stack frame dell'invocazione ricorsiva fact_recursive(3)
nessun calcolo finora

n: 4
rec_call_result = fact_recursive(3)

stack frame dell'invocazione ricorsiva fact_recursive(4)
nessun calcolo finora

n: 5
rec_call_result = fact_recursive(4)

stack frame dell'invocazione ricorsiva fact_recursive(5)
nessun calcolo finora

n: 6
rec_call_result = fact_recursive(5)

stack frame dell'invocazione fact_recursive(6) dal main
nessun calcolo finora

Funzioni ricorsive – come cambia lo stack a runtime

n: 1 rec_call_result = fact_recursive(0)	
n: 2 rec_call_result = fact_recursive(1)	stack frame dell'invocazione ricorsiva fact_recursive(2) nessun calcolo finora
n: 3 rec_call_result = fact_recursive(2)	stack frame dell'invocazione ricorsiva fact_recursive(3) nessun calcolo finora
n: 4 rec_call_result = fact_recursive(3)	stack frame dell'invocazione ricorsiva fact_recursive(4) nessun calcolo finora
n: 5 rec_call_result = fact_recursive(4)	stack frame dell'invocazione ricorsiva fact_recursive(5) nessun calcolo finora
n: 6 rec_call_result = fact_recursive(5)	stack frame dell'invocazione fact_recursive(6) dal main nessun calcolo finora

Funzioni ricorsive – come cambia lo stack a runtime

n: 0
return 1

stack frame dell'invocazione ricorsiva fact_recursive(0)
caso base: viene restituito 1 al chiamante

n: 1
rec_call_result = fact_recursive(0)

stack frame dell'invocazione ricorsiva fact_recursive(1)
nessun calcolo finora

n: 2
rec_call_result = fact_recursive(1)

stack frame dell'invocazione ricorsiva fact_recursive(2)
nessun calcolo finora

n: 3
rec_call_result = fact_recursive(2)

stack frame dell'invocazione ricorsiva fact_recursive(3)
nessun calcolo finora

n: 4
rec_call_result = fact_recursive(3)

stack frame dell'invocazione ricorsiva fact_recursive(4)
nessun calcolo finora

n: 5
rec_call_result = fact_recursive(4)

stack frame dell'invocazione ricorsiva fact_recursive(5)
nessun calcolo finora

n: 6
rec_call_result = fact_recursive(5)

stack frame dell'invocazione fact_recursive(6) dal main
nessun calcolo finora

Funzioni ricorsive – come cambia lo stack a runtime

n: 0
return 1

NB: return 1 significa: *restituisce 1 al chiamante, ossia a chi sta appena sotto sullo stack*

n: 1
rec_call_result = fact_recursive(0)

stack frame dell'invocazione ricorsiva fact_recursive(1)
nessun calcolo finora

n: 2
rec_call_result = fact_recursive(1)

stack frame dell'invocazione ricorsiva fact_recursive(2)
nessun calcolo finora

n: 3
rec_call_result = fact_recursive(2)

stack frame dell'invocazione ricorsiva fact_recursive(3)
nessun calcolo finora

n: 4
rec_call_result = fact_recursive(3)

stack frame dell'invocazione ricorsiva fact_recursive(4)
nessun calcolo finora

n: 5
rec_call_result = fact_recursive(4)

stack frame dell'invocazione ricorsiva fact_recursive(5)
nessun calcolo finora

n: 6
rec_call_result = fact_recursive(5)

stack frame dell'invocazione fact_recursive(6) dal main
nessun calcolo finora

Funzioni ricorsive – come cambia lo stack a runtime

n: 1 rec_call_result = fact_recursive(0): 1 restituisce 1 al chiamante	viene eseguita l'assegnazione, ora rec_call_result: 1; viene poi eseguita l'istruzione successiva che è: return n * 1, ossia return 1 * 1, quindi return 1
n: 2 rec_call_result = fact_recursive(1)	stack frame dell'invocazione ricorsiva fact_recursive(2) nessun calcolo finora
n: 3 rec_call_result = fact_recursive(2)	stack frame dell'invocazione ricorsiva fact_recursive(3) nessun calcolo finora
n: 4 rec_call_result = fact_recursive(3)	stack frame dell'invocazione ricorsiva fact_recursive(4) nessun calcolo finora
n: 5 rec_call_result = fact_recursive(4)	stack frame dell'invocazione ricorsiva fact_recursive(5) nessun calcolo finora
n: 6 rec_call_result = fact_recursive(5)	stack frame dell'invocazione fact_recursive(6) dal main nessun calcolo finora

Funzioni ricorsive – come cambia lo 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 al chiamante

viene eseguita l'assegnazione, ora `rec_call_result`: 1;
viene poi eseguita l'istruzione successiva che è: `return n * 1`, ossia `return 2 * 1`, quindi `return 2`

n: 3
rec_call_result = fact_recursive(2)

stack frame dell'invocazione ricorsiva `fact_recursive(3)`
nessun calcolo finora

n: 4
rec_call_result = fact_recursive(3)

stack frame dell'invocazione ricorsiva `fact_recursive(4)`
nessun calcolo finora

n: 5
rec_call_result = fact_recursive(4)

stack frame dell'invocazione ricorsiva `fact_recursive(5)`
nessun calcolo finora

n: 6
rec_call_result = fact_recursive(5)

stack frame dell'invocazione `fact_recursive(6)` dal main
nessun calcolo finora

Funzioni ricorsive – come cambia lo stack a runtime

n: 3
rec_call_result = fact_recursive(2): 2
restituisce 6 al chiamante

viene eseguita l'assegnazione, ora rec_call_result: 2;
viene poi eseguita l'istruzione successiva che è: return n
* 2, ossia return 3 * 2, quindi return 6

n: 4
rec_call_result = fact_recursive(3)

stack frame dell'invocazione ricorsiva fact_recursive(4)
nessun calcolo finora

n: 5
rec_call_result = fact_recursive(4)

stack frame dell'invocazione ricorsiva fact_recursive(5)
nessun calcolo finora

n: 6
rec_call_result = fact_recursive(5)

stack frame dell'invocazione fact_recursive(6) dal main
nessun calcolo finora

Funzioni ricorsive – come cambia lo stack a runtime

n: 4
rec_call_result = fact_recursive(3): 6
restituisce 24 al chiamante

viene eseguita l'assegnazione, ora rec_call_result: 2;
viene poi eseguita l'istruzione successiva che è: return n
* 6, ossia return 4 * 6, quindi return 24

n: 5
rec_call_result = fact_recursive(4)

stack frame dell'invocazione ricorsiva fact_recursive(5)
nessun calcolo finora

n: 6
rec_call_result = fact_recursive(5)

stack frame dell'invocazione fact_recursive(6) dal main
nessun calcolo finora

Funzioni ricorsive – come cambia lo stack a runtime

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

viene eseguita l'assegnazione, ora rec_call_result: 2;
viene poi eseguita l'istruzione successiva che è: return n
* 6, ossia return 4 * 6, quindi return 24

n: 6
rec_call_result = fact_recursive(5)

stack frame dell'invocazione fact_recursive(6) dal main
nessun calcolo finora

Funzioni ricorsive – come cambia lo stack a runtime

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

viene eseguita l'assegnazione, ora rec_call_result: 120;
viene poi eseguita l'istruzione successiva che è: return n
* 6, ossia return 120 * 6, quindi return 120

Funzioni ricorsive – come cambia lo 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 le cose è estremamente inefficiente a livello di tempo di esecuzione: abbiamo dovuto aspettare un bel po' prima di fare anche soltanto $1 * 1$
- 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: ricorsione infinita e *stack overflow*

- La dimensione dello stack del 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();  
}
```

```
int main() {  
    bad_rec_func();  
}
```

Funzioni ricorsive: 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 (probabile) / distruzione dello stack

```
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 al 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*

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

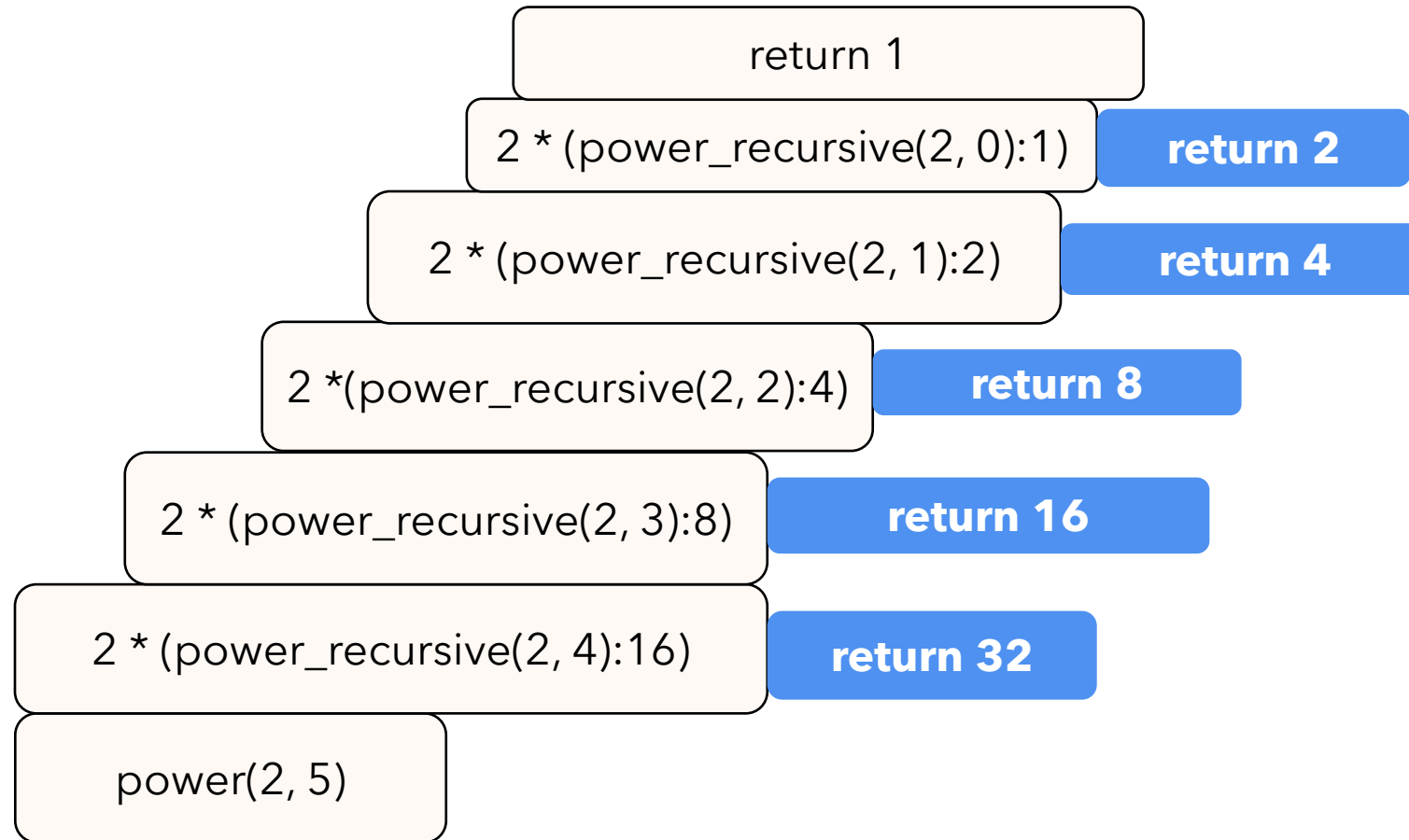
- **Potenza** - definizione *ricorsiva*

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

- **Domanda**: spiegare cosa c'è di «ricorsivo» nell'ultima definizione

Funzioni ricorsive

```
double power_recursive (double n, int m) {  
    return m == 0 ? 1 : n * power_recursive(n, m - 1);  
}
```



Definire ricorsivamente oggetti concreti

Def. Un treno è (iterativamente):

il vagone 1 collegato al vagone 2, collegato al vagone 3, collegato al vagone 4 ... collegato all'ultimo vagone

NB: la definizione precedente è iterativa perché specifica come «costruire» il treno passo passo

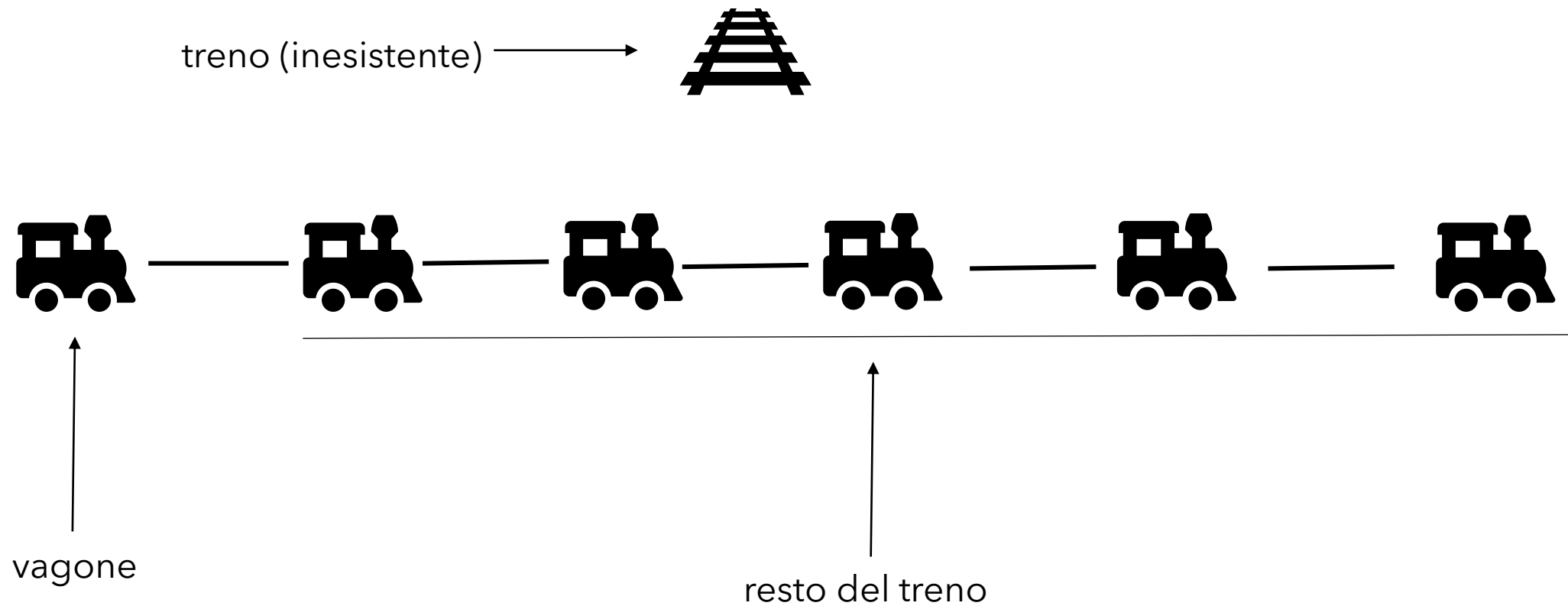
Definire ricorsivamente oggetti concreti

Def. Un treno è (ricorsivamente):

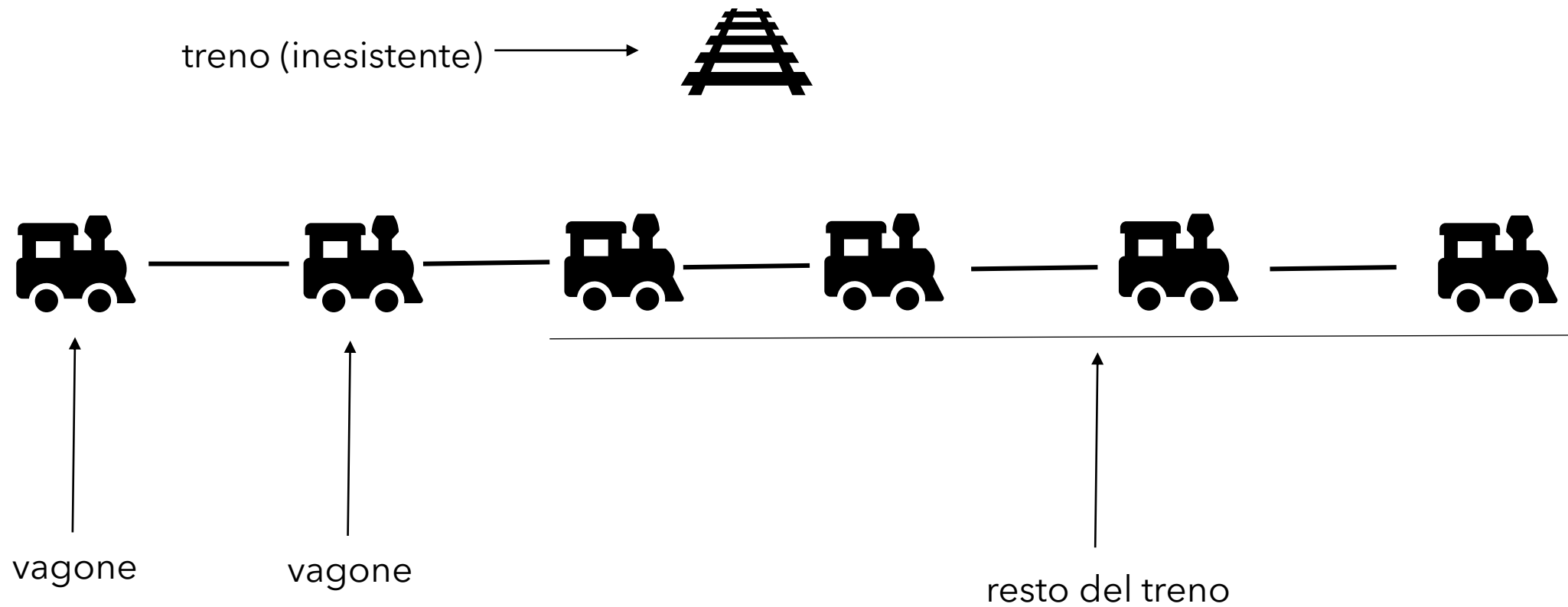
- binari senza niente sopra (caso base)
- un vagone collegato al resto del treno (caso induttivo/ricorsivo)

NB: la definizione precedente è ricorsiva perché utilizza sé stessa su istanze sempre più piccole, e sfrutta l'autosomiglianza del concetto di treno

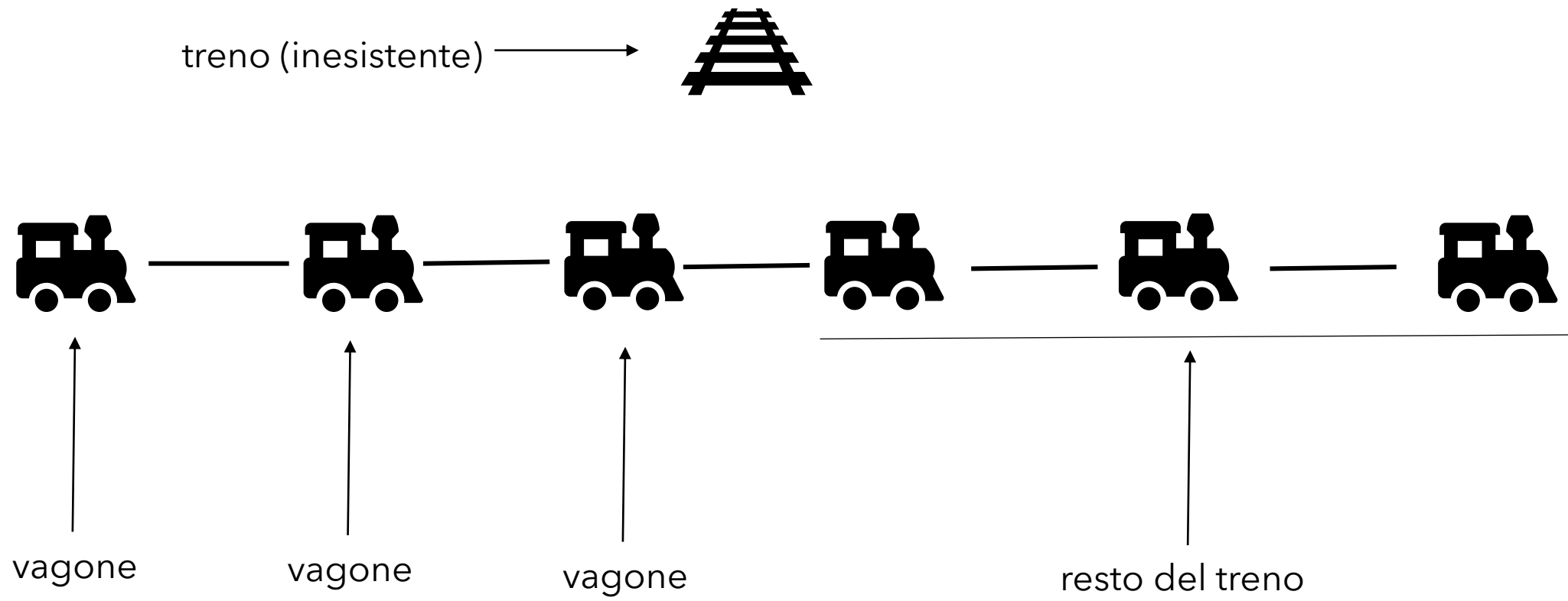
Definire ricorsivamente oggetti concreti



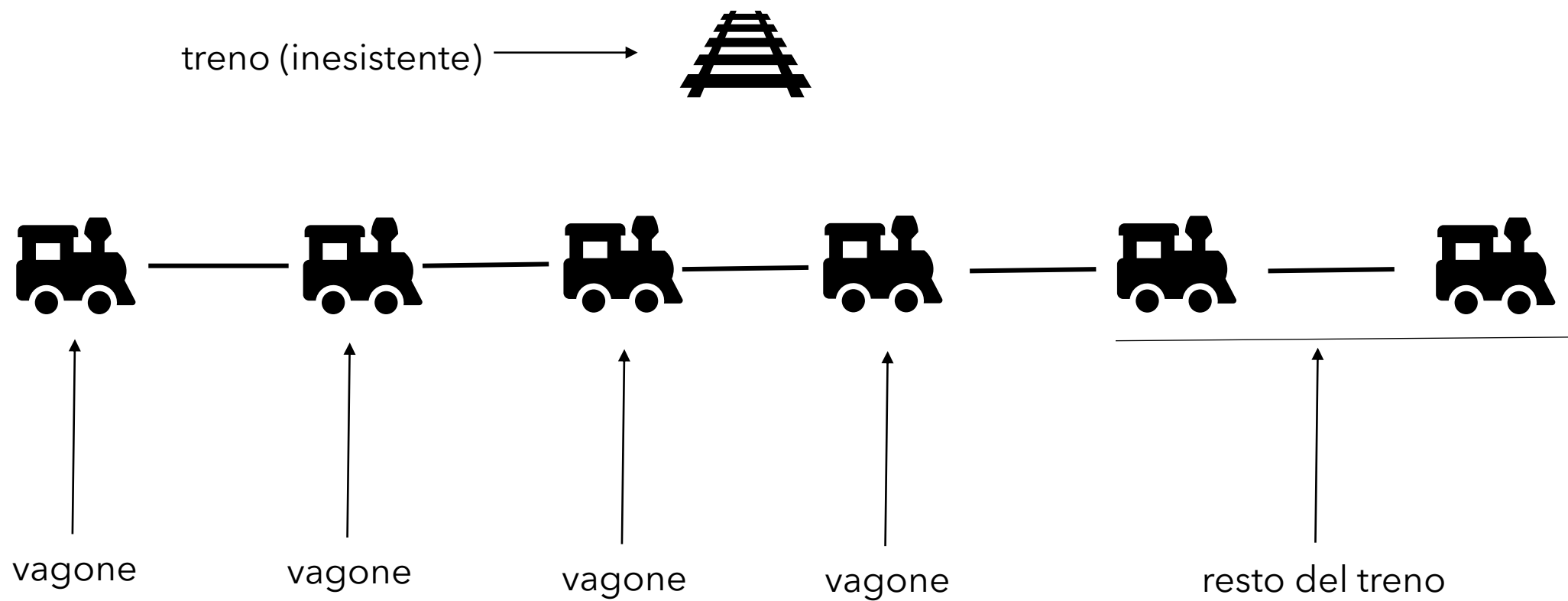
Definire ricorsivamente oggetti concreti



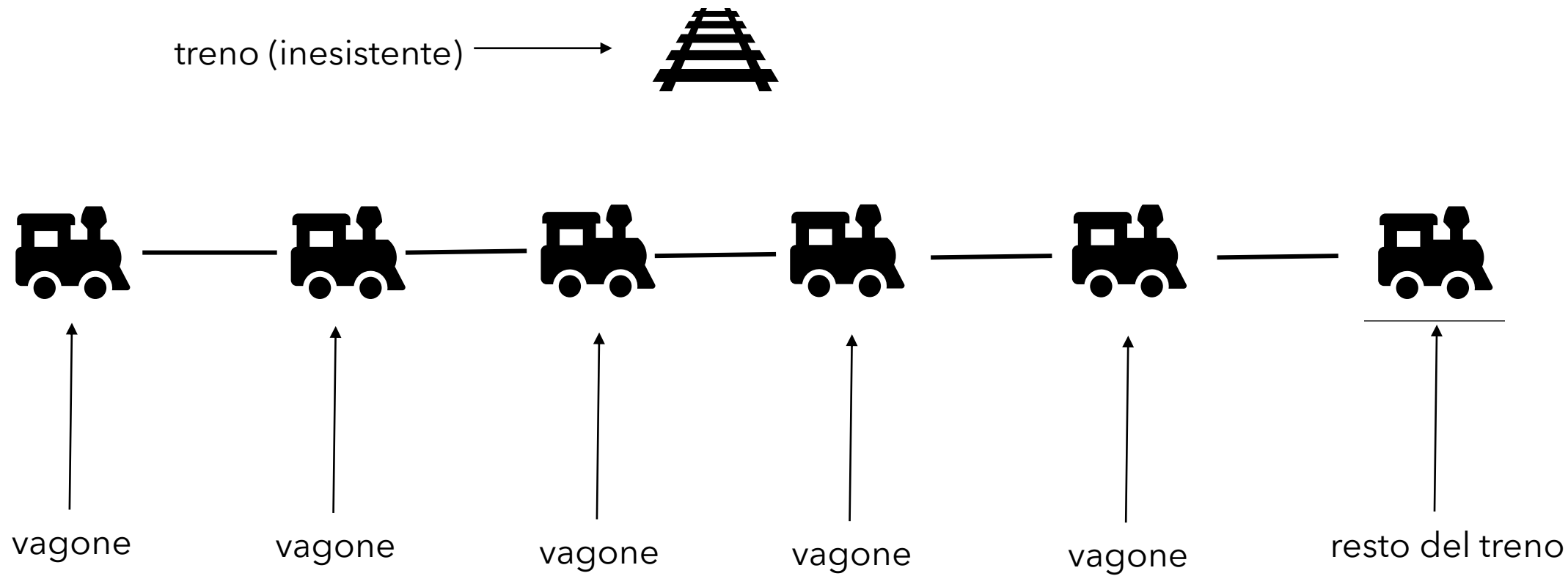
Definire ricorsivamente oggetti concreti



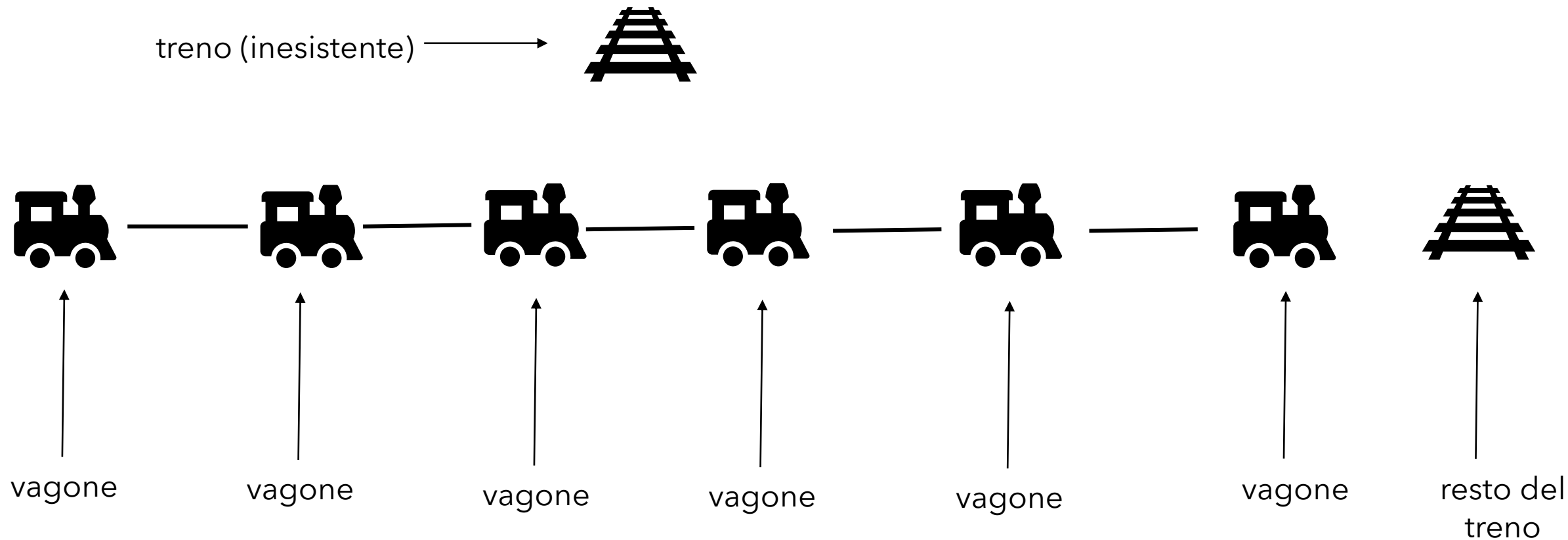
Definire ricorsivamente oggetti concreti



Definire ricorsivamente oggetti concreti



Definire ricorsivamente oggetti concreti



La successione di Fibonacci

(<https://en.wikipedia.org/wiki/Fibonacci>)

$$a_0 = 0$$

$$a_1 = 1$$

$$a_n = a_{n-1} + a_{n-2}$$

Primi termini della serie: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ...

Qual è la differenza rispetto alle funzioni ricorsive precedenti?

La successione di Fibonacci

```
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;  
}
```

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

Fibonacci sullo stack

n: 4
fib_n_1 = fibonacci(3)

stack frame dell'invocazione fibonacci(4) dal main

Fibonacci sullo stack

n: 3
fib_n_1 = fibonacci(2)

stack frame dell'invocazione ricorsiva fibonacci(n-1):
fibonacci(3): nessun calcolo finora

n: 4
fib_n_1 = fibonacci(3)

stack frame dell'invocazione fibonacci(4) dal main

Fibonacci sullo stack

n: 2
fib_n_1 = fibonacci(1)

stack frame dell'invocazione ricorsiva fibonacci(n-1):
fibonacci(3): nessun calcolo finora

n: 3
fib_n_1 = fibonacci(2)

stack frame dell'invocazione ricorsiva fibonacci(n-1):
fibonacci(3): nessun calcolo finora

n: 4
fib_n_1 = fibonacci(3)

stack frame dell'invocazione fibonacci(4) dal main

Fibonacci sullo stack

n: 1
restituisce 1 al chiamante

n: 2
fib_n_1 = fibonacci(1)

n: 3
fib_n_1 = fibonacci(2)

n: 4
fib_n_1 = fibonacci(3)

stack frame dell'invocazione ricorsiva fibonacci(n-1):
fibonacci(1): caso base: restituisce 1 al chiamante

stack frame dell'invocazione ricorsiva fibonacci(n-1):
fibonacci(2): nessun calcolo finora

stack frame dell'invocazione ricorsiva fibonacci(n-1):
fibonacci(3): nessun calcolo finora

stack frame dell'invocazione fibonacci(4) dal main

Fibonacci sullo stack

n: 1
restituisce 1 al chiamante

n: 2
fib_n_1 = fibonacci(1)

n: 3
fib_n_1 = fibonacci(2)

n: 4
fib_n_1 = fibonacci(3)

stack frame dell'invocazione ricorsiva fibonacci(n-1):
fibonacci(1): restituisce 1 al chiamante

stack frame dell'invocazione ricorsiva fibonacci(n-1):
fibonacci(2): nessun calcolo finora

stack frame dell'invocazione ricorsiva fibonacci(n-1):
fibonacci(3): nessun calcolo finora

stack frame dell'invocazione fibonacci(4) dal main

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)
```

stack frame dell'invocazione ricorsiva fibonacci(n-1):
fibonacci(2): fib_n_1 ora vale 1
ora viene invocata fibonacci(n-2): fibonacci(0)

stack frame dell'invocazione ricorsiva fibonacci(n-1):
fibonacci(3): nessun calcolo finora

stack frame dell'invocazione fibonacci(4) dal main

Fibonacci sullo stack

n: 0
restituisce 0 al chiamante

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)

stack frame dell'invocazione ricorsiva fibonacci(n-2):
fibonacci(0)
caso base: restituisce 0 al chiamante

stack frame dell'invocazione ricorsiva fibonacci(n-1):
fibonacci(2): fib_n_1 ora vale 1
ora viene invocata fibonacci(n-2): fibonacci(0)

stack frame dell'invocazione ricorsiva fibonacci(n-1):
fibonacci(3): nessun calcolo finora

stack frame dell'invocazione fibonacci(4) dal main

Fibonacci sullo stack

n: 0
restituisce 0 al chiamante

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)

stack frame dell'invocazione ricorsiva fibonacci(n-2):
fibonacci(0)
caso base: restituisce 0 al chiamante

stack frame dell'invocazione ricorsiva fibonacci(n-1):
fibonacci(2): fib_n_1 ora vale 1
ora viene invocata fibonacci(n-2): fibonacci(0)

stack frame dell'invocazione ricorsiva fibonacci(n-1):
fibonacci(3): nessun calcolo finora

stack frame dell'invocazione fibonacci(4) dal main

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)
```

stack frame dell'invocazione ricorsiva fibonacci(n-1):
fibonacci(2): fib_n_1 ora vale 1
fib_n_2 ora vale 0
restituisce al chiamante fib_n_1 + fib_n_2: 1 + 0: 1

stack frame dell'invocazione ricorsiva fibonacci(n-1):
fibonacci(3): nessun calcolo finora

stack frame dell'invocazione fibonacci(4) dal main

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)
```

stack frame dell'invocazione ricorsiva fibonacci(n-1):
fibonacci(2): fib_n_1 ora vale 1
fib_n_2 ora vale 0
restituisce al chiamante la somma fib_n_1 + fib_n_2

stack frame dell'invocazione ricorsiva fibonacci(n-1):
fibonacci(3): nessun calcolo finora

stack frame dell'invocazione fibonacci(4) dal main

Fibonacci sullo stack

n: 3
fib_n_1 = fibonacci(2): 1
invoca fibonacci(n-2): fibonacci(1)

n: 4
fib_n_1 = fibonacci(3)

stack frame dell'invocazione ricorsiva fibonacci(n-1):
fibonacci(3), ora fib_n_1 vale 1
ora viene invocata fibonacci(n-2): fibonacci(1)

stack frame dell'invocazione fibonacci(4) dal main

Fibonacci sullo stack

n: 1
restituisce 1 al chiamante

stack frame dell'invocazione ricorsiva fibonacci(n-2):
fibonacci(1)
caso base: restituisce 1 al chiamante

n: 3
fib_n_1 = fibonacci(2): 1
invoca fibonacci(n-2): fibonacci(1)

stack frame dell'invocazione ricorsiva fibonacci(n-1):
fibonacci(3), ora fib_n_1 vale 1
ora viene invocata fibonacci(1)

n: 4
fib_n_1 = fibonacci(3)

stack frame dell'invocazione fibonacci(4) dal main

Fibonacci sullo stack

n: 1
restituisce 1 al chiamante

stack frame dell'invocazione ricorsiva fibonacci(n-2):
fibonacci(1)
caso base: restituisce 1 al chiamante

n: 3
fib_n_1 = fibonacci(2): 1
invoca fibonacci(n-2): fibonacci(1)

stack frame dell'invocazione ricorsiva fibonacci(n-1):
fibonacci(3), ora fib_n_1 vale 1
ora viene invocata fibonacci(1)

n: 4
fib_n_1 = fibonacci(3)

stack frame dell'invocazione fibonacci(4) dal main

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)
```

stack frame dell'invocazione ricorsiva fibonacci(n-1):
fibonacci(3), ora fib_n_1 vale 1
ora fib_n_2 vale 1

stack frame dell'invocazione fibonacci(4) dal main

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

stack frame dell'invocazione ricorsiva fibonacci(n-1)
restituisce fib_n_1 + fib_n_2: 1 + 2 al chiamante

n: 4

fib_n_1 = fibonacci(3)

stack frame dell'invocazione fibonacci(4) dal main

Fibonacci sullo stack

n: 4
fib_n_1 = fibonacci(3): 2

stack frame dell'invocazione fibonacci(4) dal main
ora fib_n_1 vale 2
invoca fibonacci(n-2): fibonacci(2)

Fibonacci sullo stack

n: 2
fib_n_1 = fibonacci(1)

n: 4
fib_n_1 = fibonacci(3): 2

stack frame dell'invocazione ricorsiva fibonacci(n-2):
fibonacci(2)

stack frame dell'invocazione fibonacci(4) dal main
ora fib_n_1 vale 2
invoca fibonacci(n - 2): fibonacci(2)

Fibonacci sullo stack

n: 1
restituisce 1 al chiamante

n: 2
fib_n_1 = fibonacci(1)

n: 4
fib_n_1 = fibonacci(3): 2

stack frame dell'invocazione ricorsiva fibonacci(n-2)
caso base: restituisce 1 al chiamante

stack frame dell'invocazione ricorsiva fibonacci(n-2)
restituisce 2 al chiamante

stack frame dell'invocazione fibonacci(4) dal main
ora fib_n_1 vale 2
invoca fibonacci(n - 2): fibonacci(2)

Fibonacci sullo stack

n: 1
restituisce 1 al chiamante

n: 2
fib_n_1 = fibonacci(1)

n: 4
fib_n_1 = fibonacci(3): 2

stack frame dell'invocazione ricorsiva fibonacci(n-2)
caso base: restituisce 1 al chiamante

stack frame dell'invocazione ricorsiva fibonacci(n-2)
restituisce 2 al chiamante

stack frame dell'invocazione fibonacci(4) dal main
ora fib_n_1 vale 2
invoca fibonacci(n - 2): fibonacci(2)

Fibonacci sullo stack

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

stack frame dell'invocazione ricorsiva fibonacci(n-2)
ora fib_n_1 vale 1

n: 4
fib_n_1 = fibonacci(3): 2

stack frame dell'invocazione fibonacci(4) dal main
ora fib_n_1 vale 2
invoca fibonacci(n - 2): fibonacci(2)

Fibonacci sullo stack

n: 0
restituisce 0 al chiamante

stack frame dell'invocazione ricorsiva fibonacci(n-2)
restituisce 0 al chiamante

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

stack frame dell'invocazione ricorsiva fibonacci(n-2)
ora fib_n_1 vale 1

n: 4
fib_n_1 = fibonacci(3): 2

stack frame dell'invocazione fibonacci(4) dal main
ora fib_n_1 vale 2
invoca fibonacci(n - 2): fibonacci(2)

Fibonacci sullo stack

n: 0
restituisce 0 al chiamante

stack frame dell'invocazione ricorsiva fibonacci(n-2)
restituisce 0 al chiamante

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

stack frame dell'invocazione ricorsiva fibonacci(n-2)
ora fib_n_1 vale 1

n: 4
fib_n_1 = fibonacci(3): 2

stack frame dell'invocazione fibonacci(4) dal main
ora fib_n_1 vale 2
invoca fibonacci(n - 2): fibonacci(2)

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: 4  
fib_n_1 = fibonacci(3): 2
```

stack frame dell'invocazione ricorsiva fibonacci(n-2)
ora fib_n_1 vale 1
ora fib_n_2 vale 0
restituisce 1 al chiamante

stack frame dell'invocazione fibonacci(4) dal main
ora fib_n_1 vale 2
invoca fibonacci(n - 2): fibonacci(2)

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: 4  
    fib_n_1 = fibonacci(3): 2
```

stack frame dell'invocazione ricorsiva fibonacci(n-2)
ora fib_n_1 vale 1
ora fib_n_2 vale 0
restituisce 1 al chiamante

stack frame dell'invocazione fibonacci(4) dal main
ora fib_n_1 vale 2
invoca fibonacci(n - 2): fibonacci(2)

Fibonacci sullo stack

```
n: 4  
fib_n_1 = fibonacci(3): 2  
fib_n_2 = fibonacci(2): 1  
restituisce fib_n_1 + fib_n_2: 2 + 1: 3 al  
chiamante
```

stack frame dell'invocazione fibonacci(4) dal main
ora fib_n_1 vale 2
ora fib_n_2 vale 1
restituisce fib_n_1 + fib_n_2: 2 + 1: 3 al chiamante

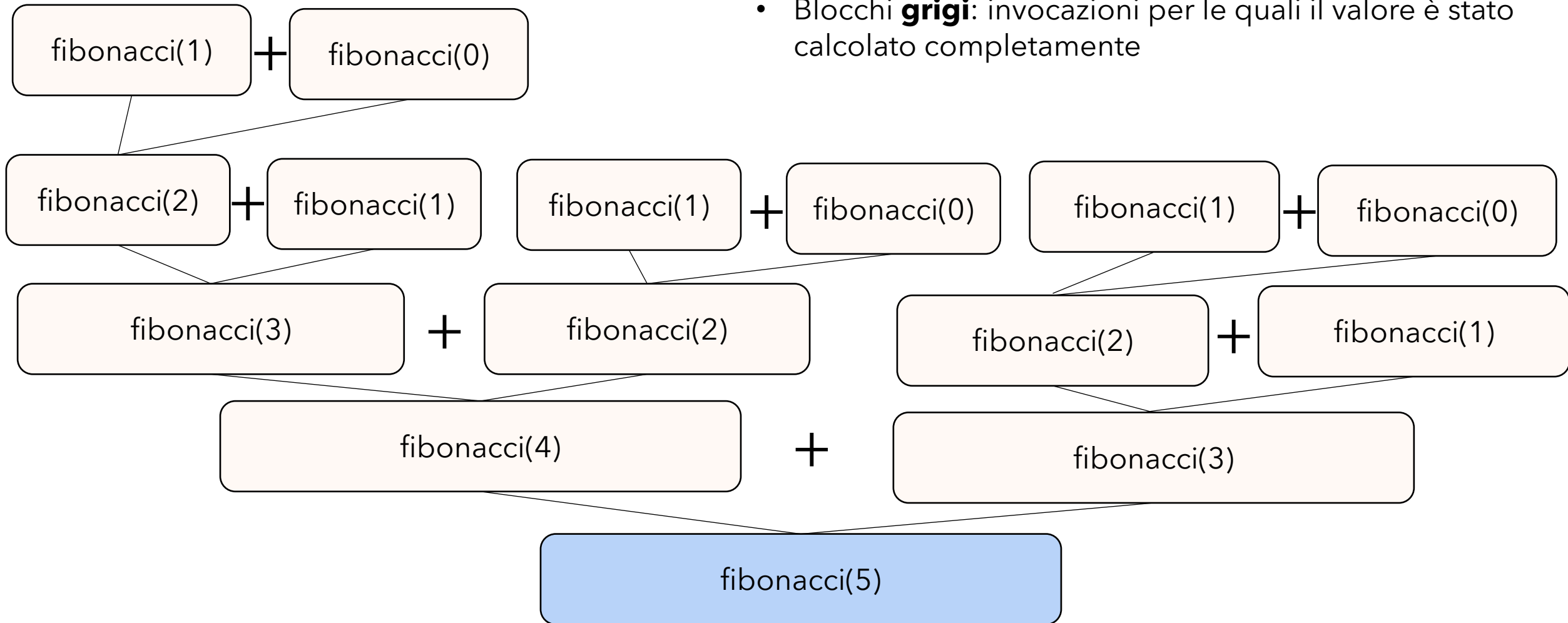
Fibonacci sullo stack

```
        n: 4
    fib_n_1 = fibonacci(3): 2
    fib_n_2 = fibonacci(2): 1
    restituisce fib_n_1 + fib_n_2: 2 + 1: 3 al
        chiamante
```

l'ultimo stack frame viene poppato e il controllo viene restituito a chi ha chiamato fibonacci(4)

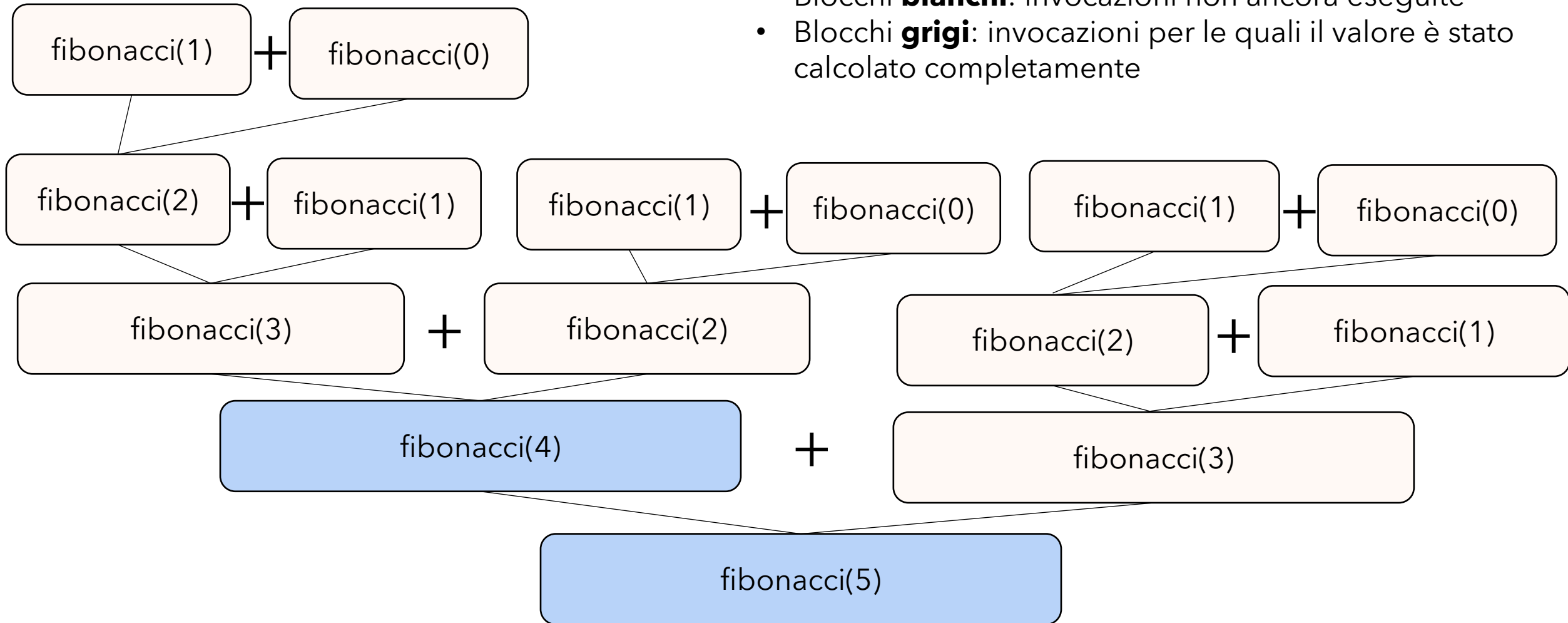
Fibonacci: visualizzazione del calcolo con l'albero delle chiamate

- Blocchi **azzurri**: invocazione attiva
- Blocchi **bianchi**: invocazioni non ancora eseguite
- Blocchi **grigi**: invocazioni per le quali il valore è stato calcolato completamente



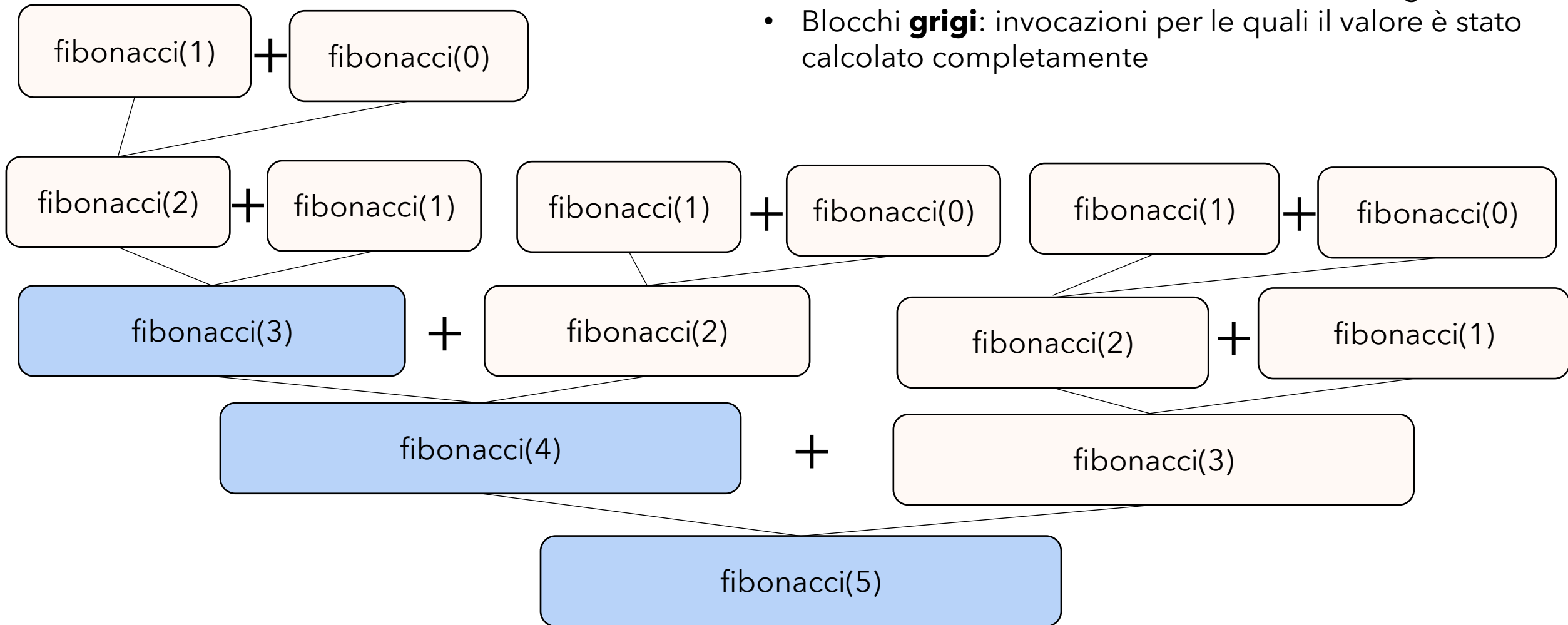
Fibonacci: visualizzazione del calcolo con l'albero delle chiamate

- Blocchi **azzurri**: invocazione attiva
- Blocchi **bianchi**: invocazioni non ancora eseguite
- Blocchi **grigi**: invocazioni per le quali il valore è stato calcolato completamente



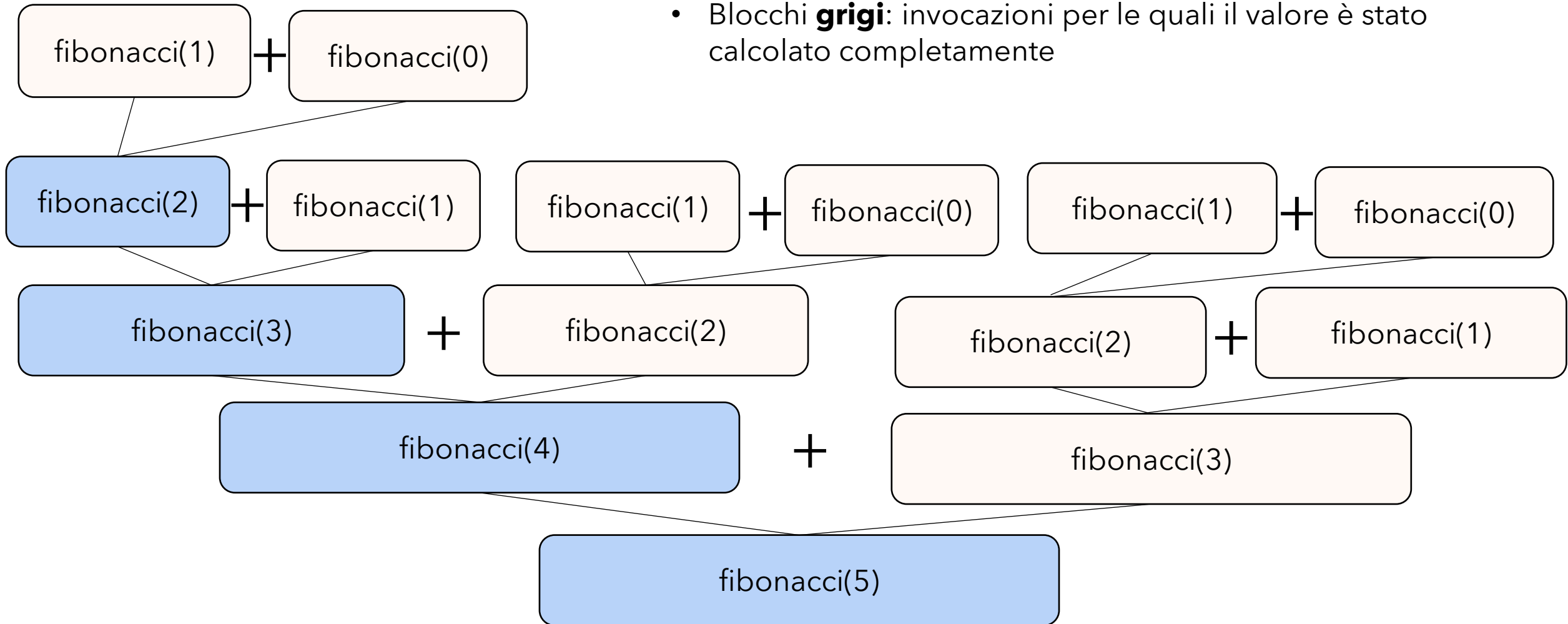
Fibonacci: visualizzazione del calcolo con l'albero delle chiamate

- Blocchi **azzurri**: invocazione attiva
- Blocchi **bianchi**: invocazioni non ancora eseguite
- Blocchi **grigi**: invocazioni per le quali il valore è stato calcolato completamente



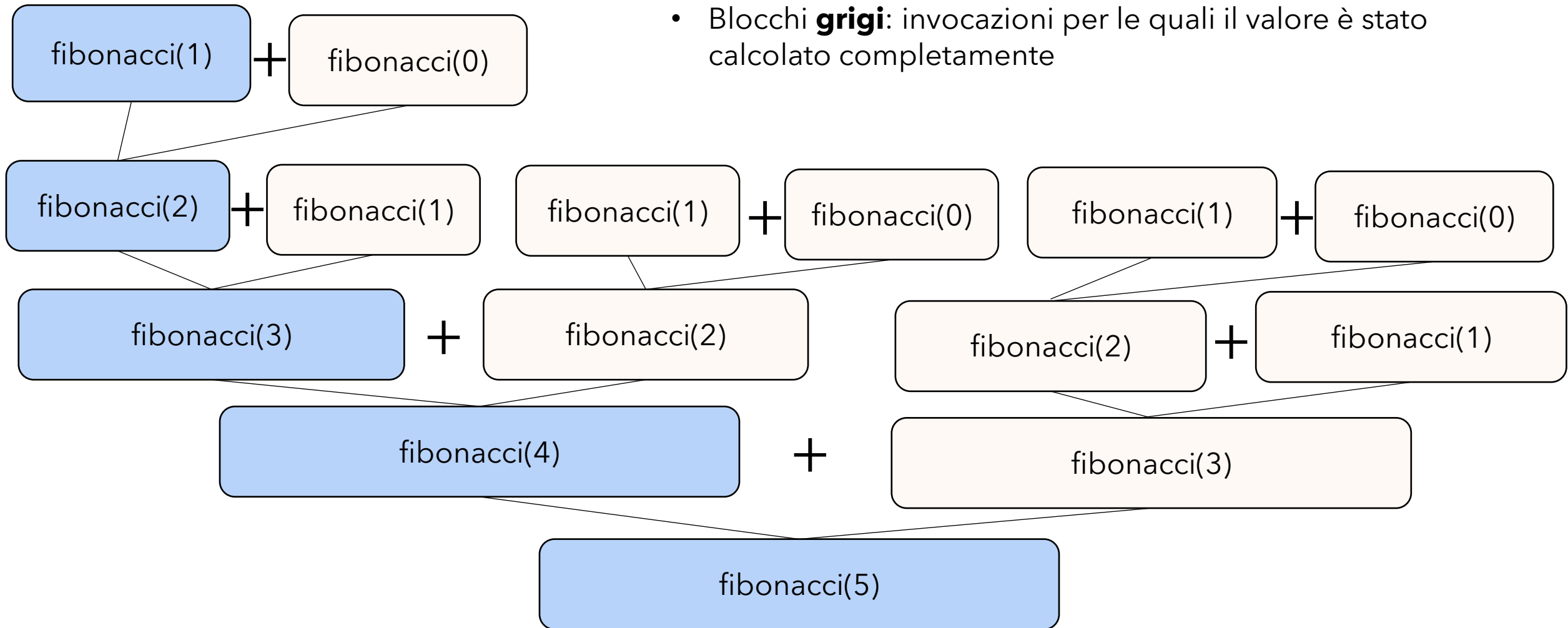
Fibonacci: visualizzazione del calcolo con l'albero delle chiamate

- Blocchi **azzurri**: invocazione corrente
- Blocchi **bianchi**: invocazioni non ancora eseguite
- Blocchi **grigi**: invocazioni per le quali il valore è stato calcolato completamente



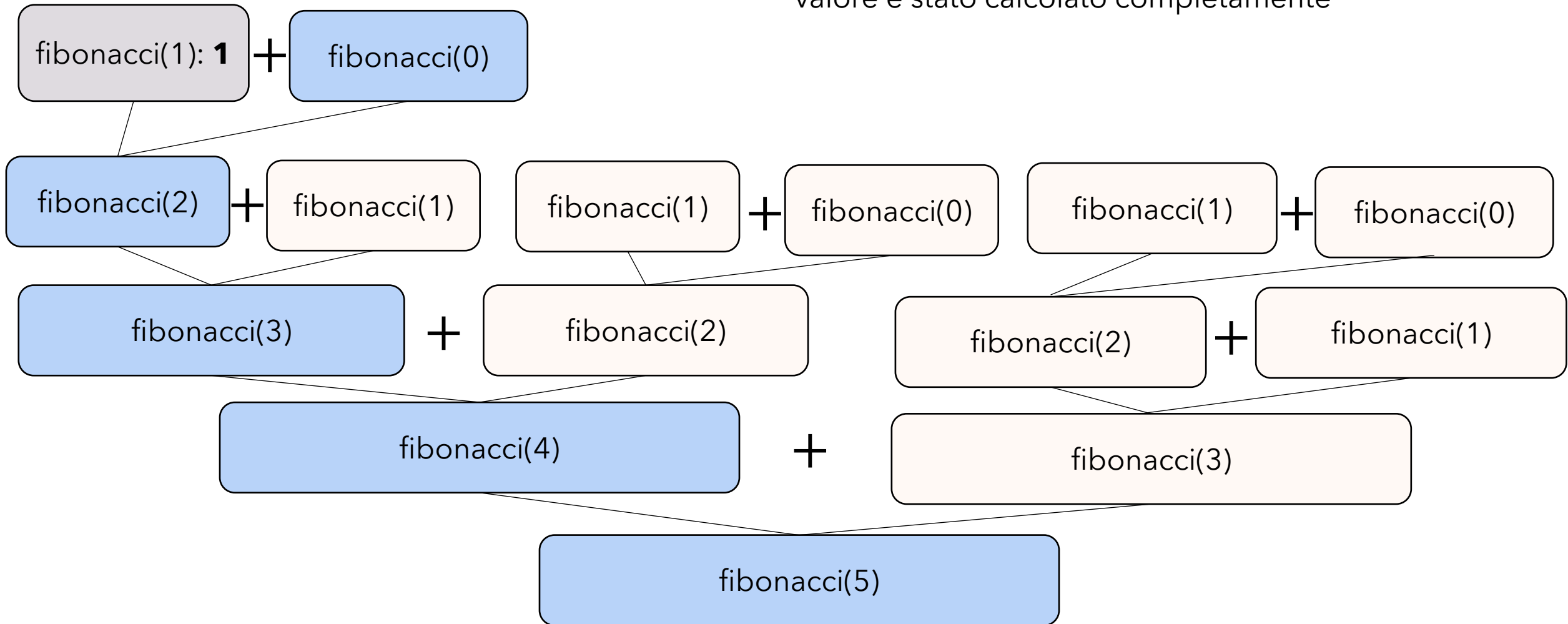
Fibonacci: visualizzazione del calcolo con l'albero delle chiamate

- Blocchi **azzurri**: invocazione corrente
- Blocchi **bianchi**: invocazioni non ancora eseguite
- Blocchi **grigi**: invocazioni per le quali il valore è stato calcolato completamente



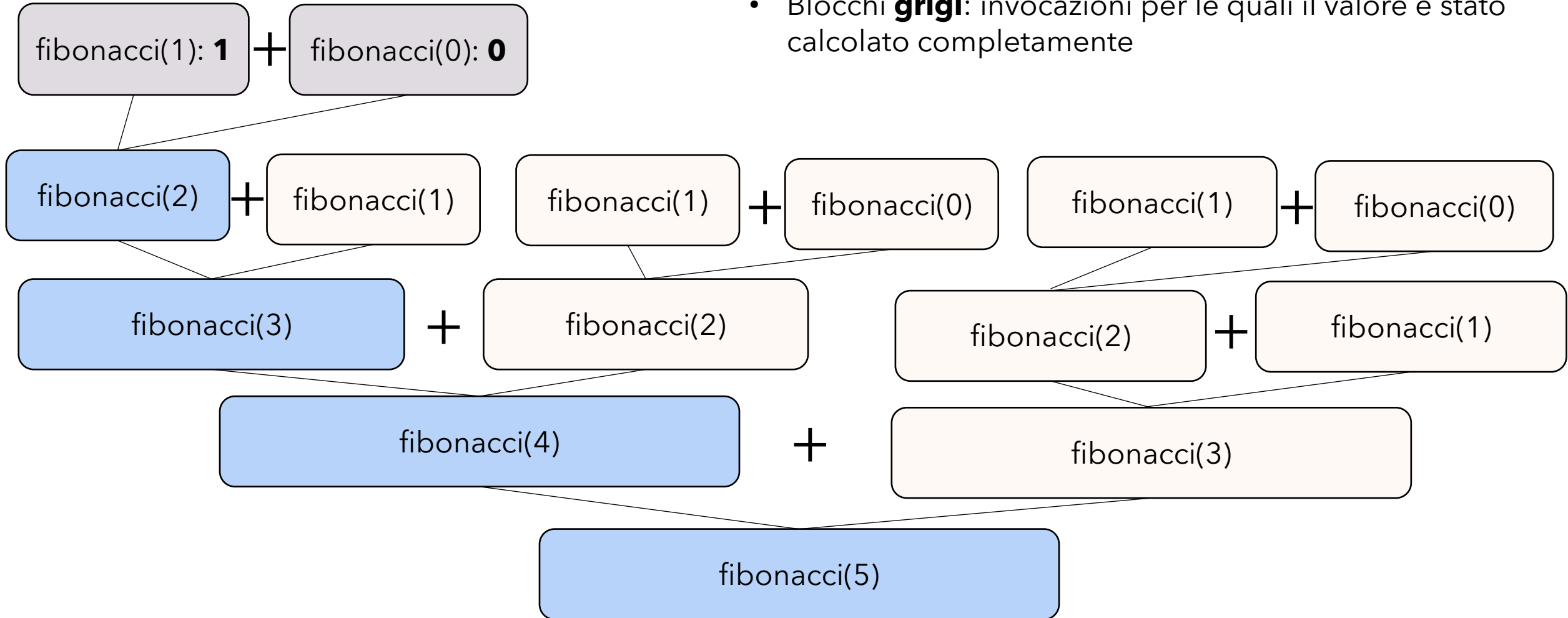
Funzioni ricorsive

- Blocchi **azzurri**: invocazione corrente
- Blocchi **bianchi**: invocazioni non ancora eseguite
- Blocchi **grigio scuro**: invocazioni per le quali il valore è stato calcolato completamente



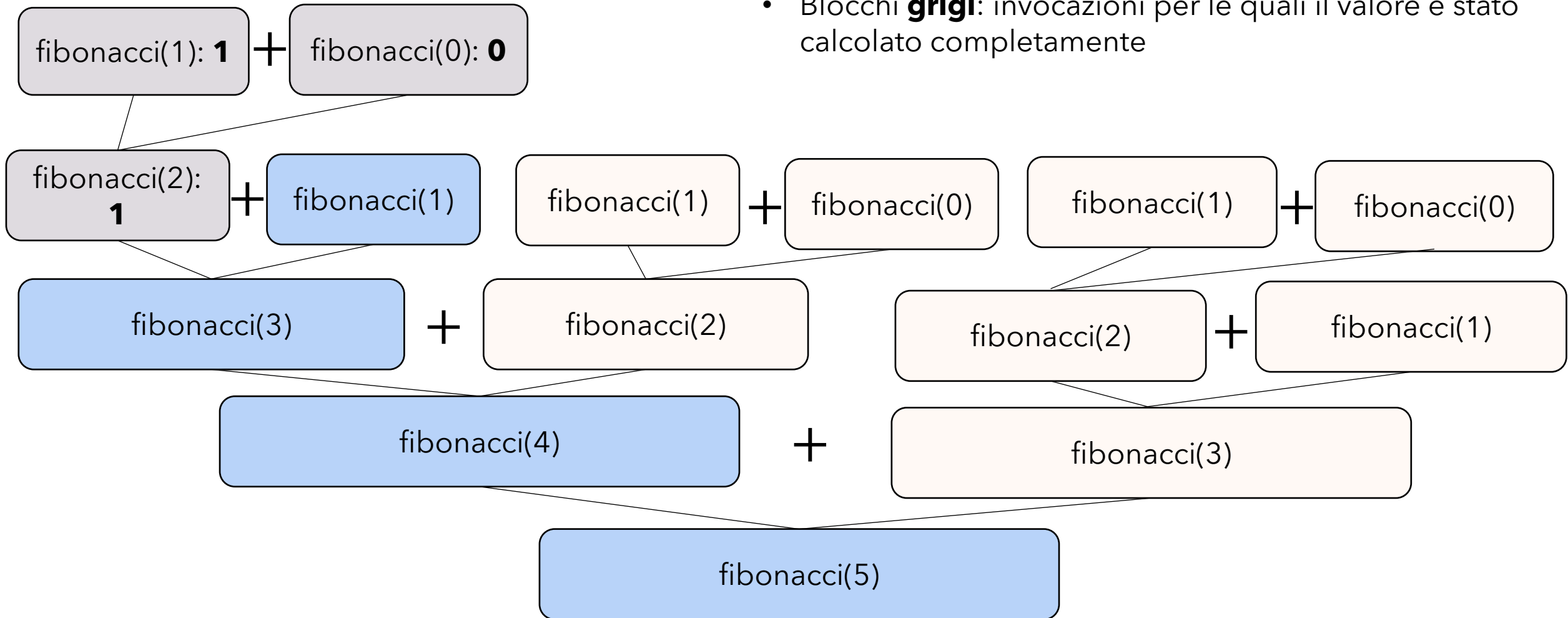
Fibonacci: visualizzazione del calcolo con l'albero delle chiamate

- Blocchi **azzurri**: invocazione corrente
- Blocchi **bianchi**: invocazioni non ancora eseguite
- Blocchi **grigi**: invocazioni per le quali il valore è stato calcolato completamente



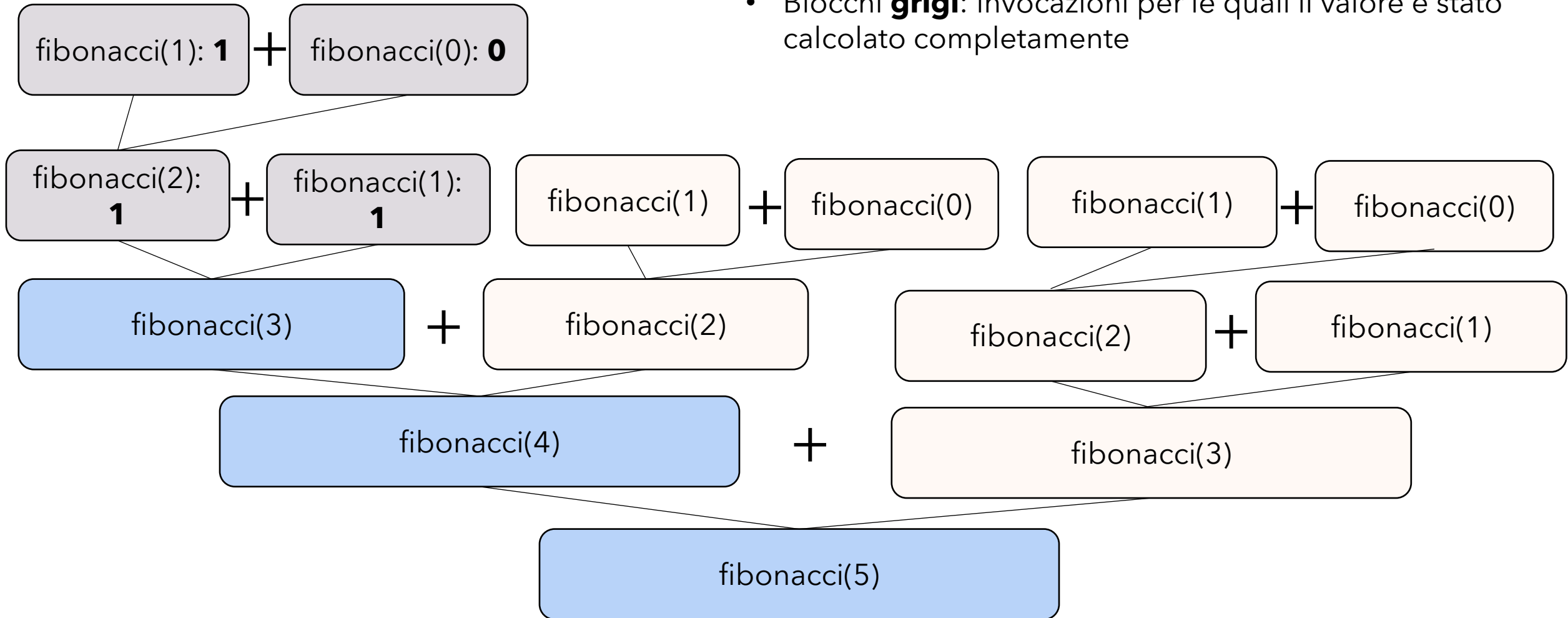
Fibonacci: visualizzazione del calcolo con l'albero delle chiamate

- Blocchi **azzurri**: invocazione corrente
- Blocchi **bianchi**: invocazioni non ancora eseguite
- Blocchi **grigi**: invocazioni per le quali il valore è stato calcolato completamente



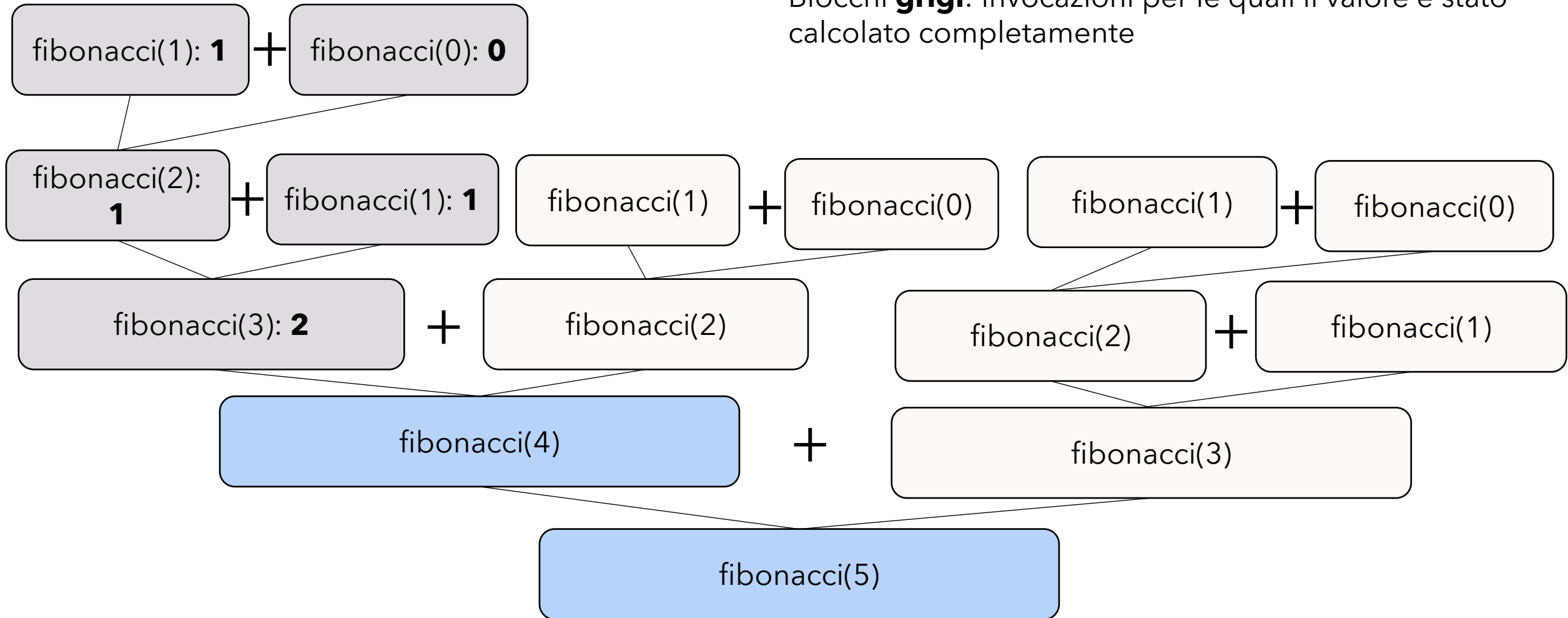
Fibonacci: visualizzazione del calcolo con l'albero delle chiamate

- Blocchi **azzurri**: invocazione corrente
- Blocchi **bianchi**: invocazioni non ancora eseguite
- Blocchi **grigi**: invocazioni per le quali il valore è stato calcolato completamente



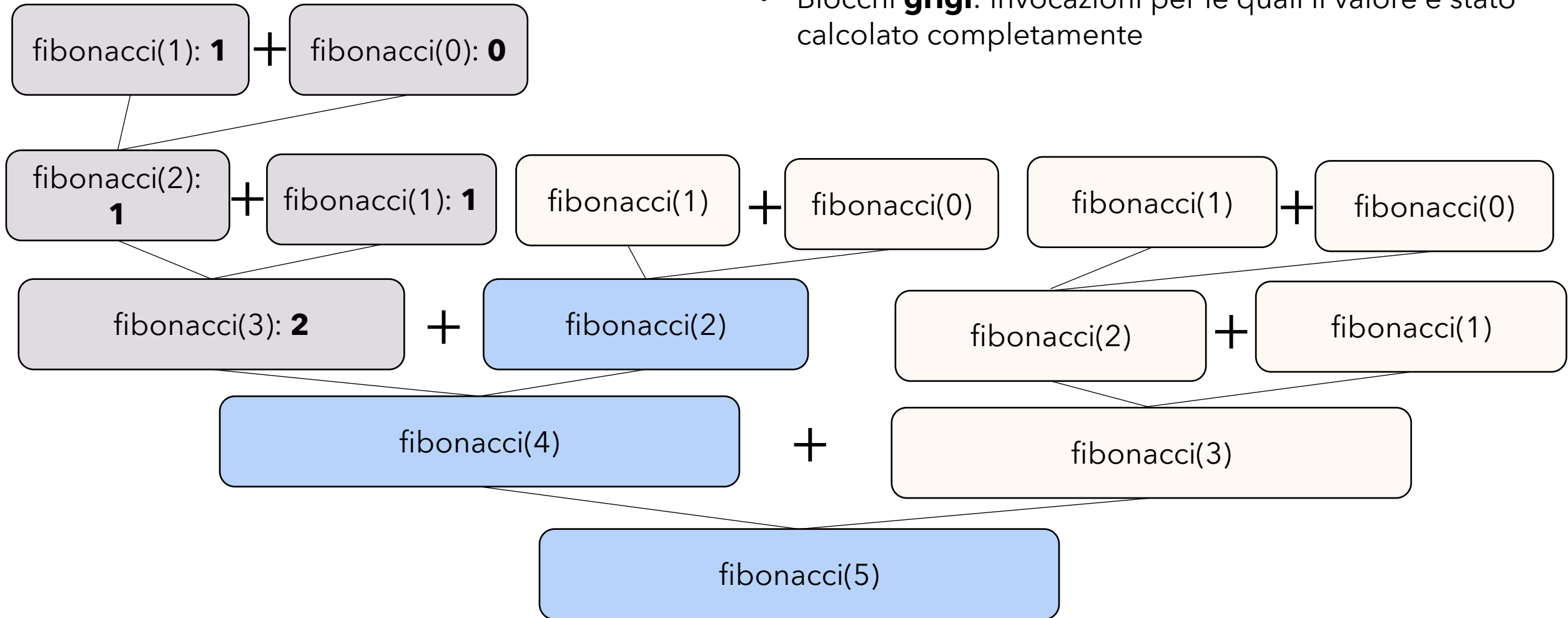
Fibonacci: visualizzazione del calcolo con l'albero delle chiamate

- Blocchi **azzurri**: invocazione corrente
- Blocchi **bianchi**: invocazioni non ancora eseguite
- Blocchi **grigi**: invocazioni per le quali il valore è stato calcolato completamente



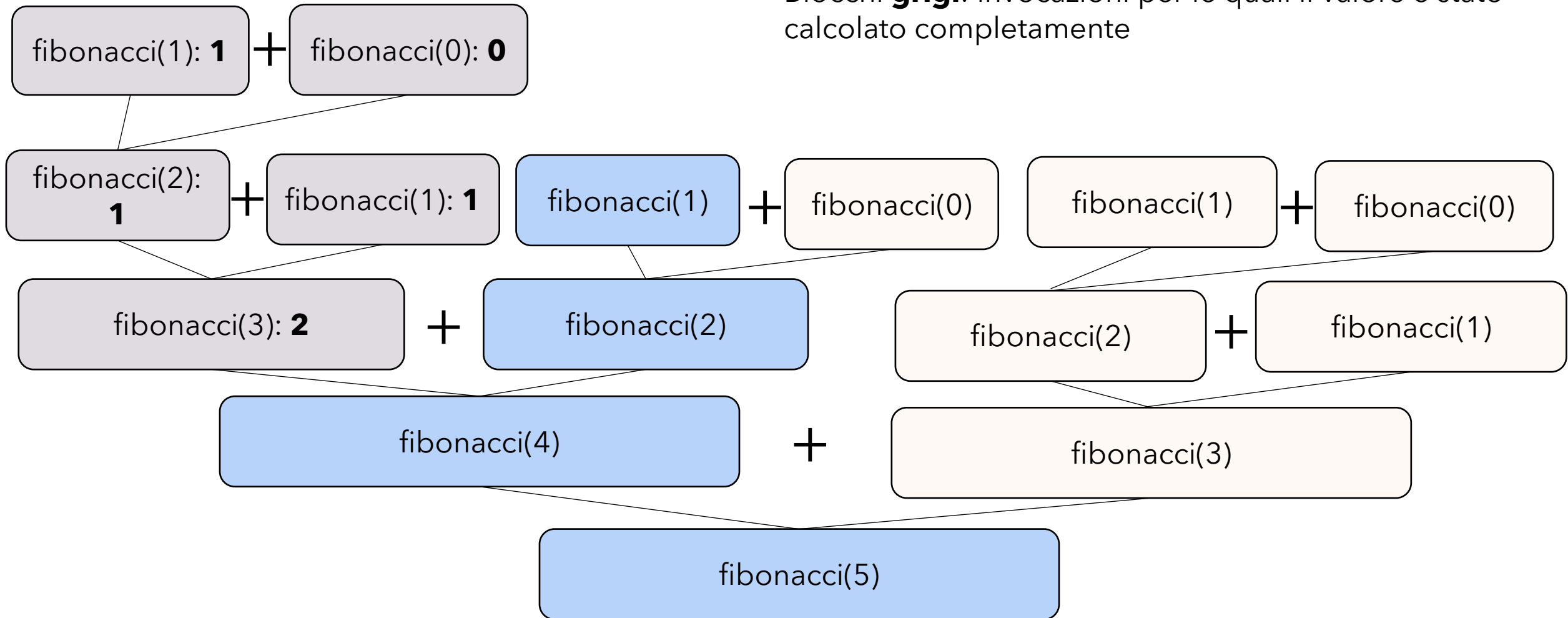
Fibonacci: visualizzazione del calcolo con l'albero delle chiamate

- Blocchi **azzurri**: invocazione corrente
- Blocchi **bianchi**: invocazioni non ancora eseguite
- Blocchi **grigi**: invocazioni per le quali il valore è stato calcolato completamente



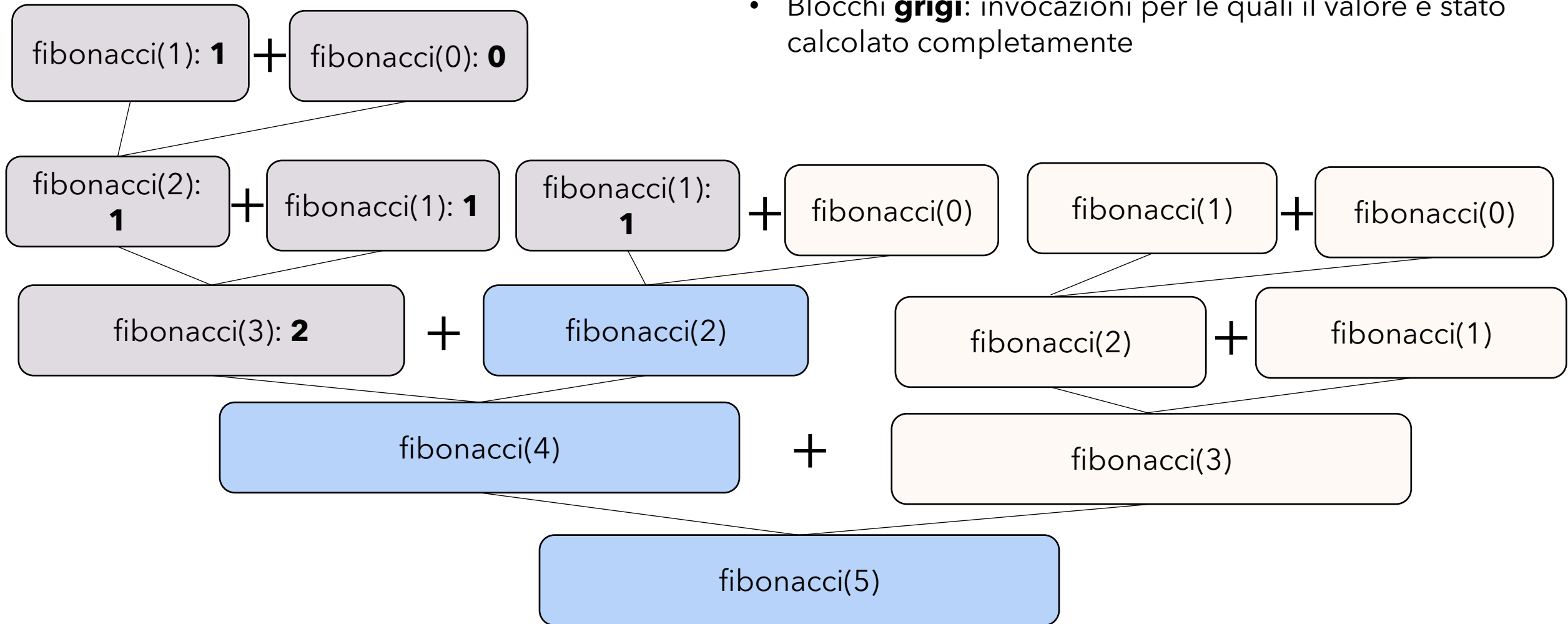
Fibonacci: visualizzazione del calcolo con l'albero delle chiamate

- Blocchi **azzurri**: invocazione corrente
- Blocchi **bianchi**: invocazioni non ancora eseguite
- Blocchi **grigi**: invocazioni per le quali il valore è stato calcolato completamente



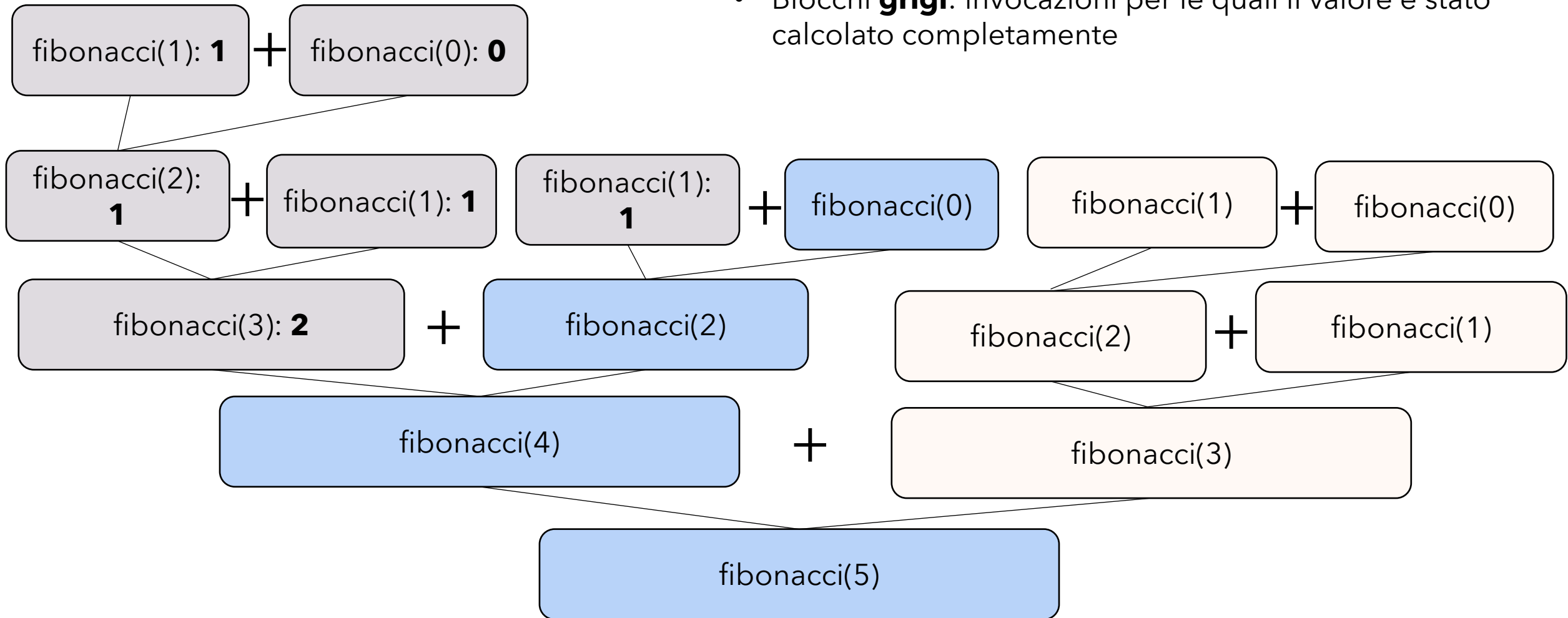
Fibonacci: visualizzazione del calcolo con l'albero delle chiamate

- Blocchi **azzurri**: invocazione corrente
- Blocchi **bianchi**: invocazioni non ancora eseguite
- Blocchi **grigi**: invocazioni per le quali il valore è stato calcolato completamente



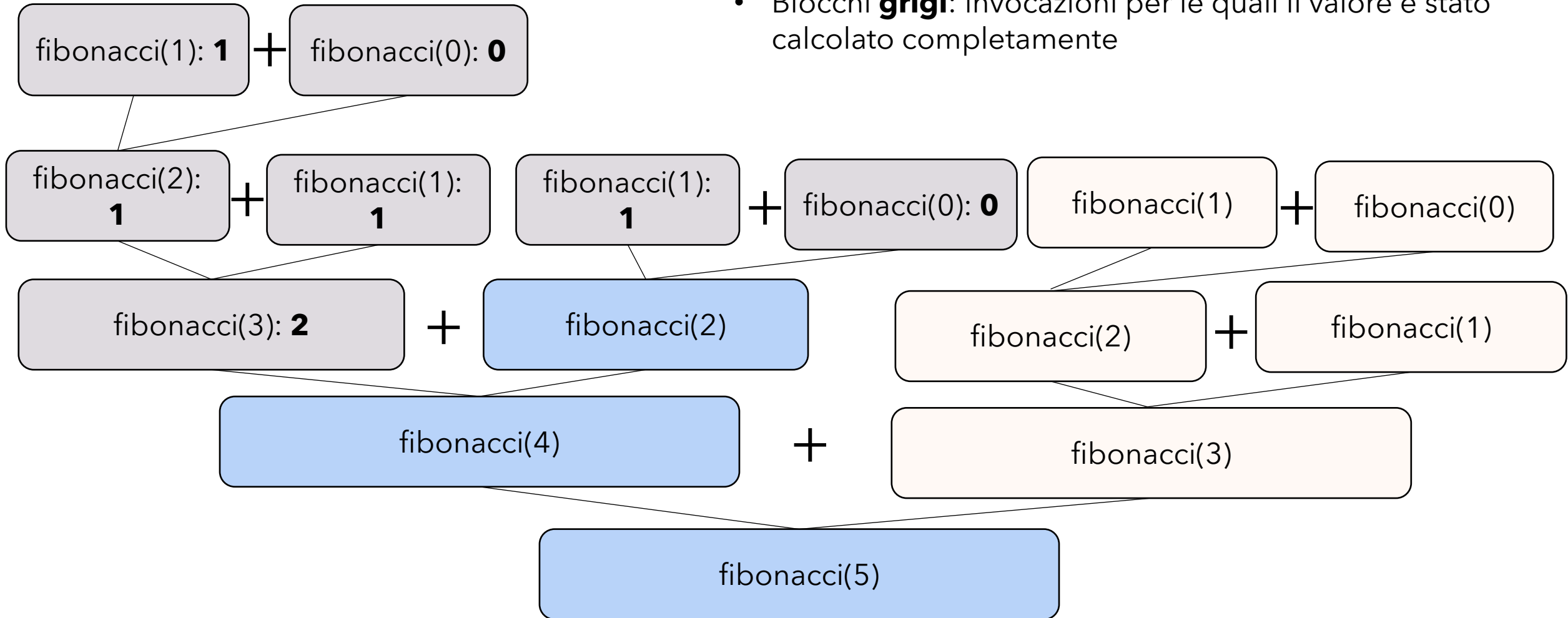
Fibonacci: visualizzazione del calcolo con l'albero delle chiamate

- Blocchi **azzurri**: invocazione corrente
- Blocchi **bianchi**: invocazioni non ancora eseguite
- Blocchi **grigi**: invocazioni per le quali il valore è stato calcolato completamente



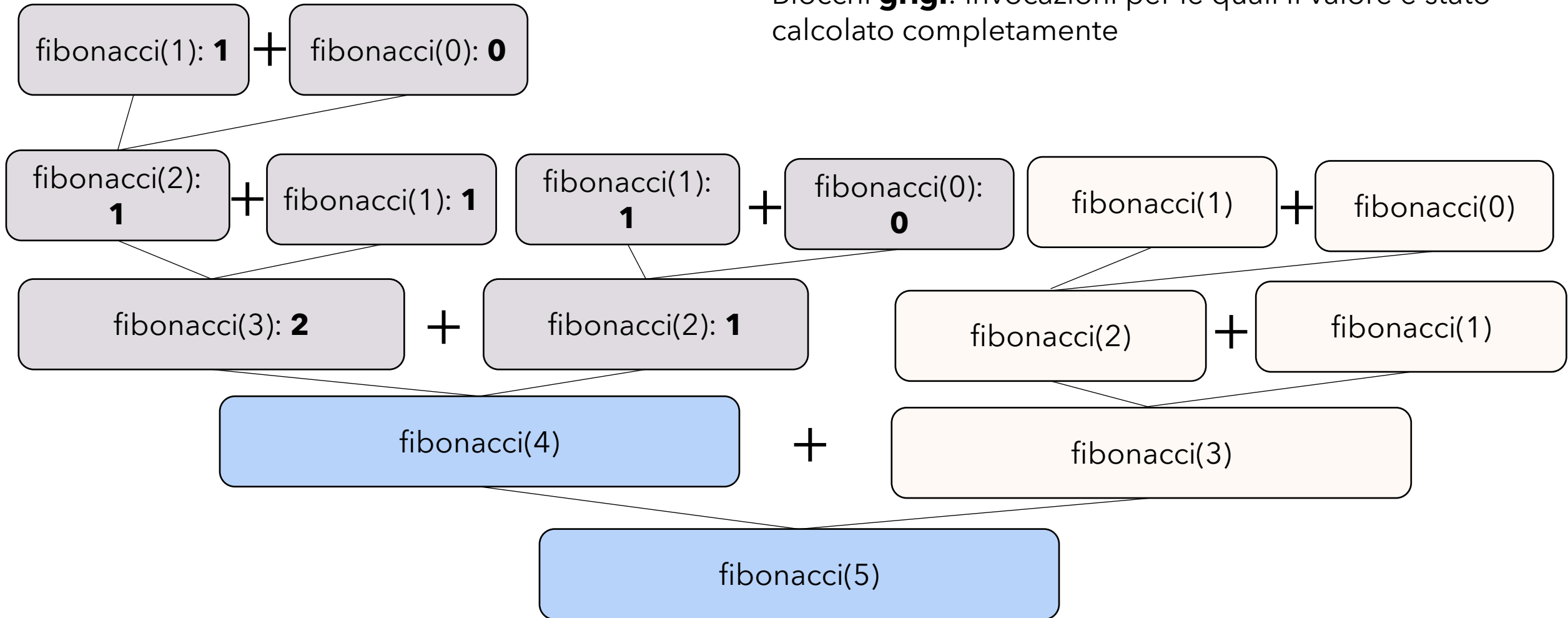
Fibonacci: visualizzazione del calcolo con l'albero delle chiamate

- Blocchi **azzurri**: invocazione corrente
- Blocchi **bianchi**: invocazioni non ancora eseguite
- Blocchi **grigi**: invocazioni per le quali il valore è stato calcolato completamente



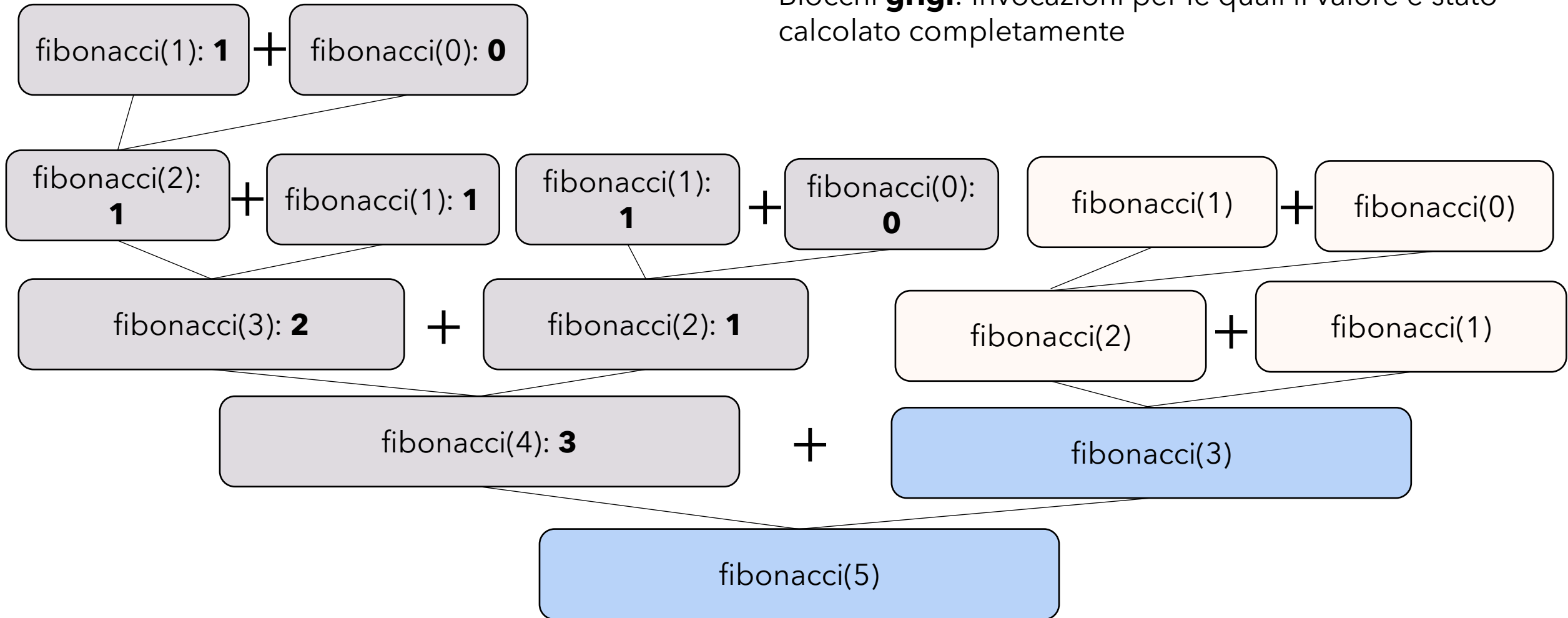
Fibonacci: visualizzazione del calcolo con l'albero delle chiamate

- Blocchi **azzurri**: invocazione corrente
- Blocchi **bianchi**: invocazioni non ancora eseguite
- Blocchi **grigi**: invocazioni per le quali il valore è stato calcolato completamente



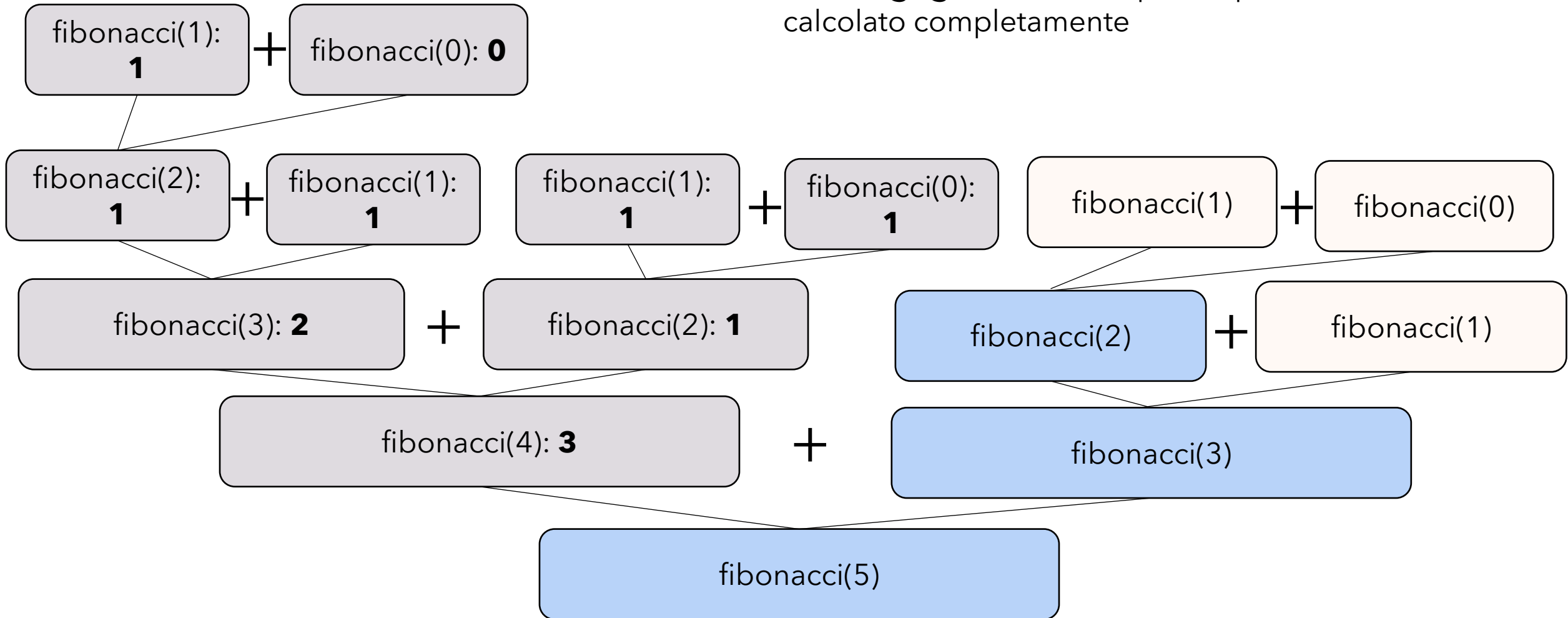
Fibonacci: visualizzazione del calcolo con l'albero delle chiamate

- Blocchi **azzurri**: invocazione corrente
- Blocchi **bianchi**: invocazioni non ancora eseguite
- Blocchi **grigi**: invocazioni per le quali il valore è stato calcolato completamente



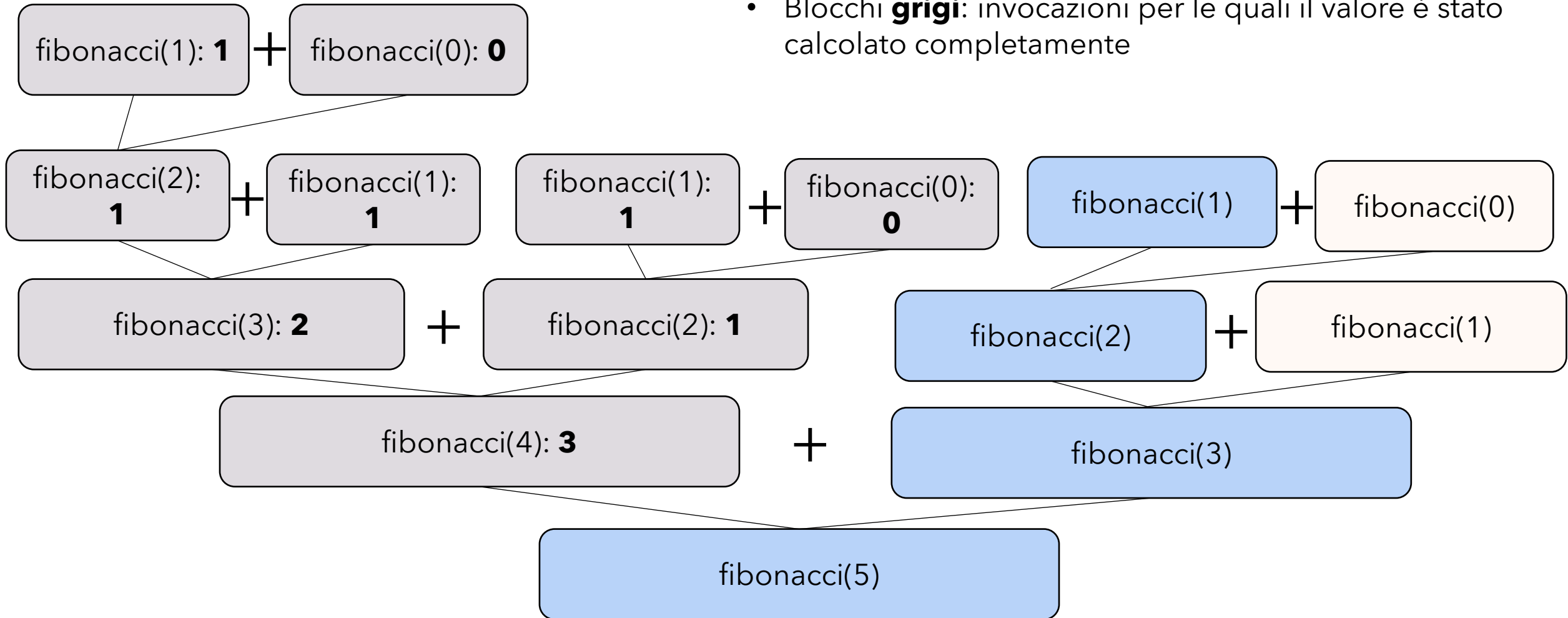
Fibonacci: visualizzazione del calcolo con l'albero delle chiamate

- Blocchi **azzurri**: invocazione corrente
- Blocchi **bianchi**: invocazioni non ancora eseguite
- Blocchi **grigi**: invocazioni per le quali il valore è stato calcolato completamente



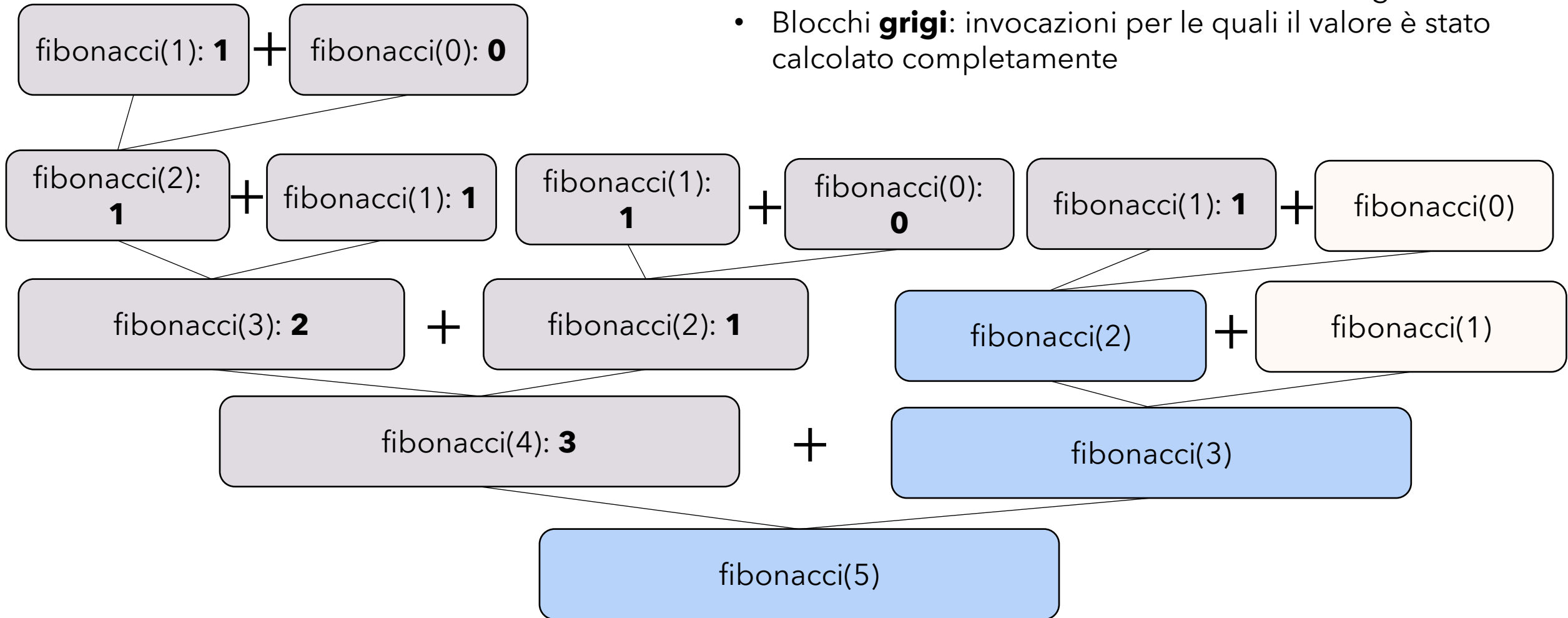
Fibonacci: visualizzazione del calcolo con l'albero delle chiamate

- Blocchi **azzurri**: invocazione corrente
- Blocchi **bianchi**: invocazioni non ancora eseguite
- Blocchi **grigi**: invocazioni per le quali il valore è stato calcolato completamente



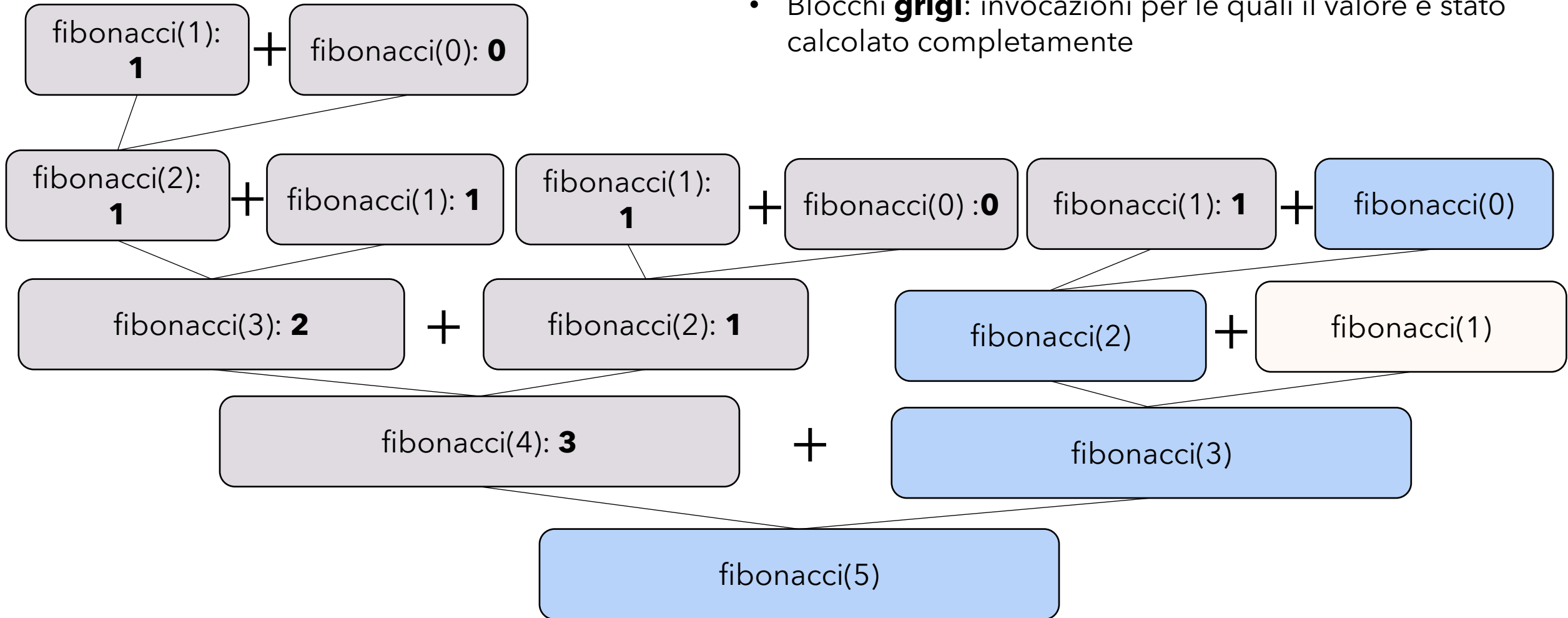
Fibonacci: visualizzazione del calcolo con l'albero delle chiamate

- Blocchi **azzurri**: invocazione corrente
- Blocchi **bianchi**: invocazioni non ancora eseguite
- Blocchi **grigi**: invocazioni per le quali il valore è stato calcolato completamente



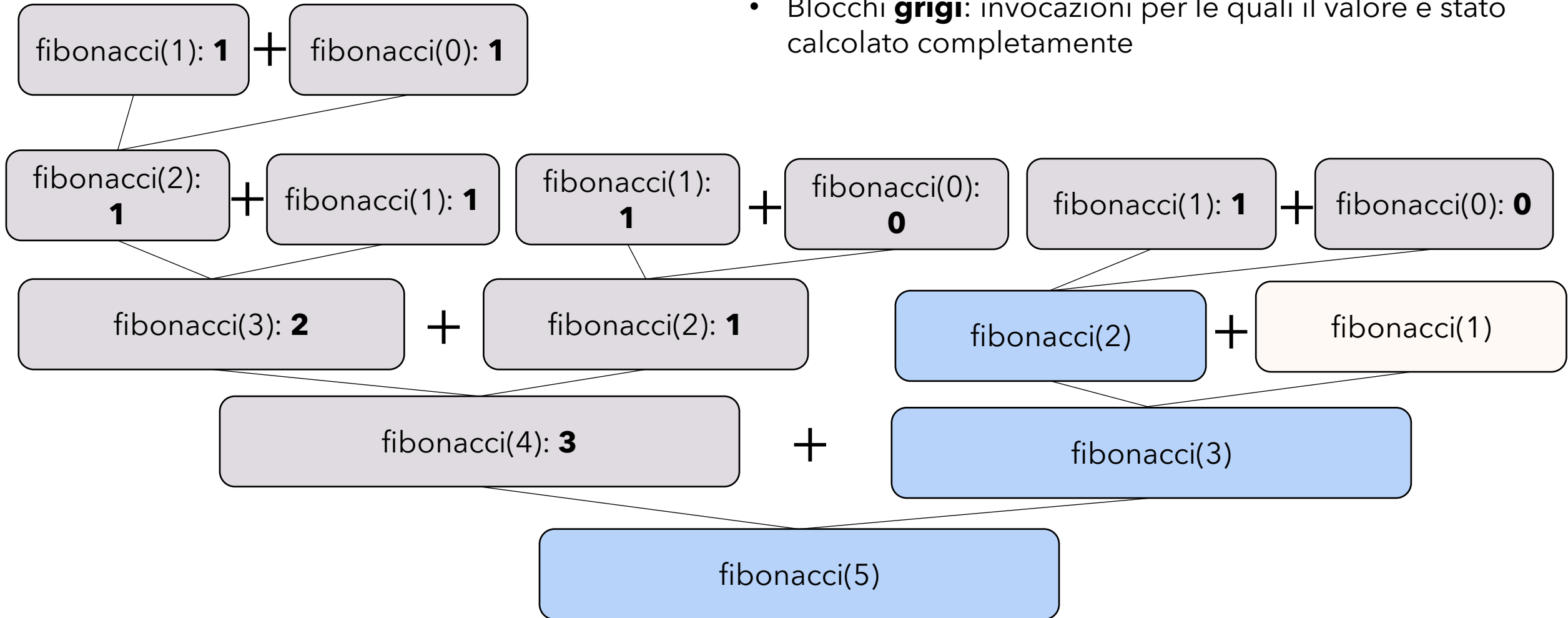
Fibonacci: visualizzazione del calcolo con l'albero delle chiamate

- Blocchi **azzurri**: invocazione corrente
- Blocchi **bianchi**: invocazioni non ancora eseguite
- Blocchi **grigi**: invocazioni per le quali il valore è stato calcolato completamente



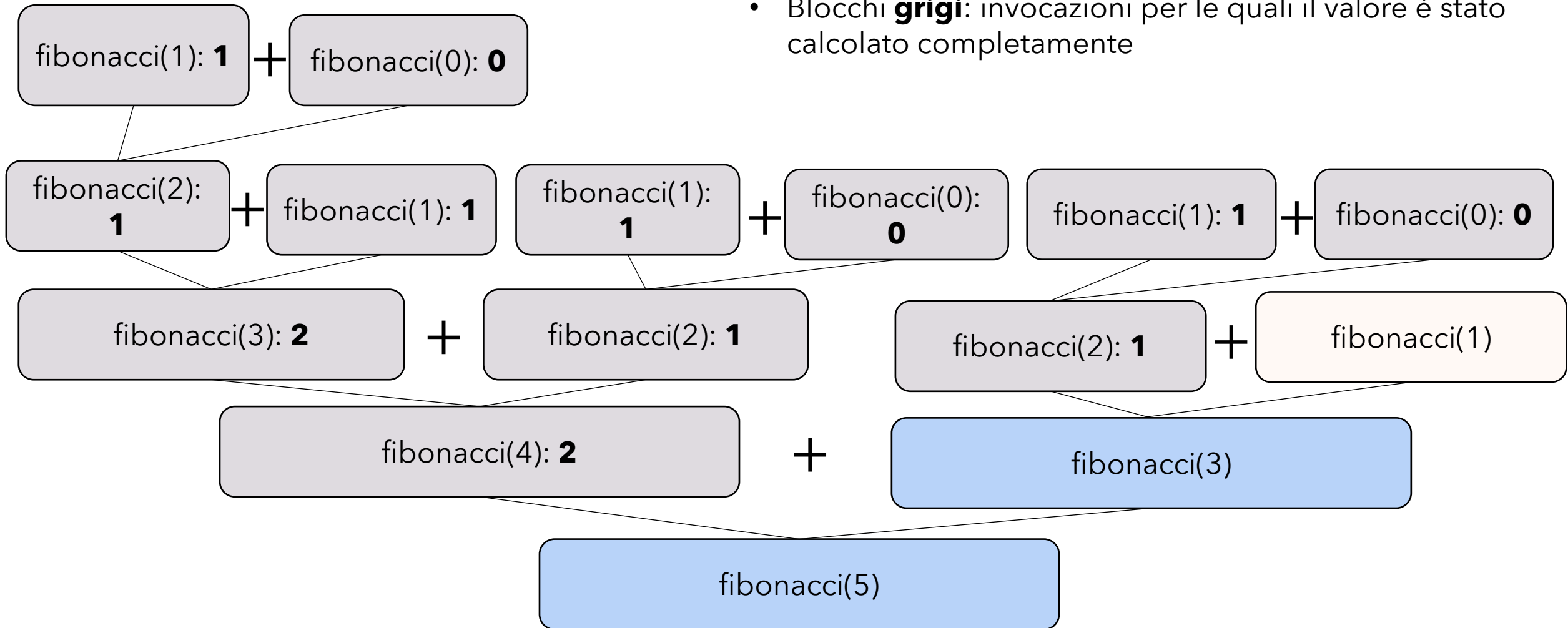
Fibonacci: visualizzazione del calcolo con l'albero delle chiamate

- Blocchi **azzurri**: invocazione corrente
- Blocchi **bianchi**: invocazioni non ancora eseguite
- Blocchi **grigi**: invocazioni per le quali il valore è stato calcolato completamente



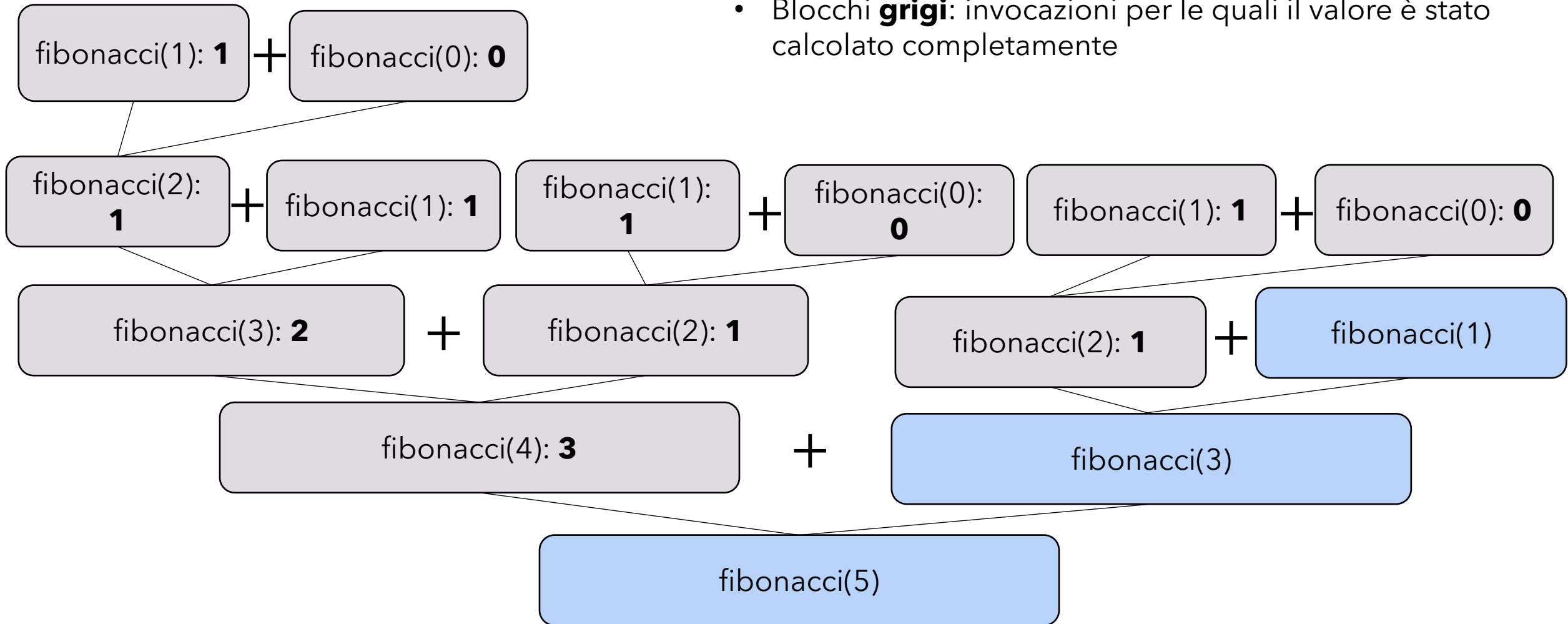
Fibonacci: visualizzazione del calcolo con l'albero delle chiamate

- Blocchi **azzurri**: invocazione corrente
- Blocchi **bianchi**: invocazioni non ancora eseguite
- Blocchi **grigi**: invocazioni per le quali il valore è stato calcolato completamente



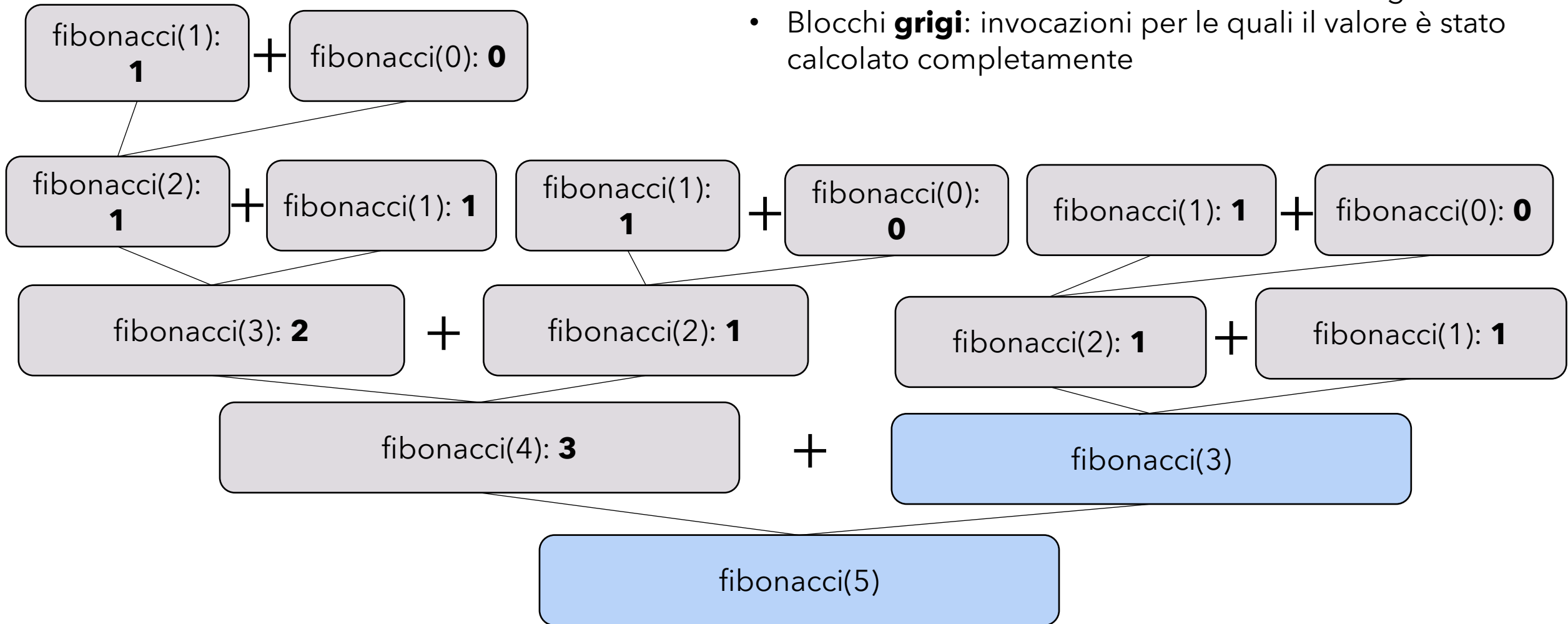
Fibonacci: visualizzazione del calcolo con l'albero delle chiamate

- Blocchi **azzurri**: invocazione corrente
- Blocchi **bianchi**: invocazioni non ancora eseguite
- Blocchi **grigi**: invocazioni per le quali il valore è stato calcolato completamente



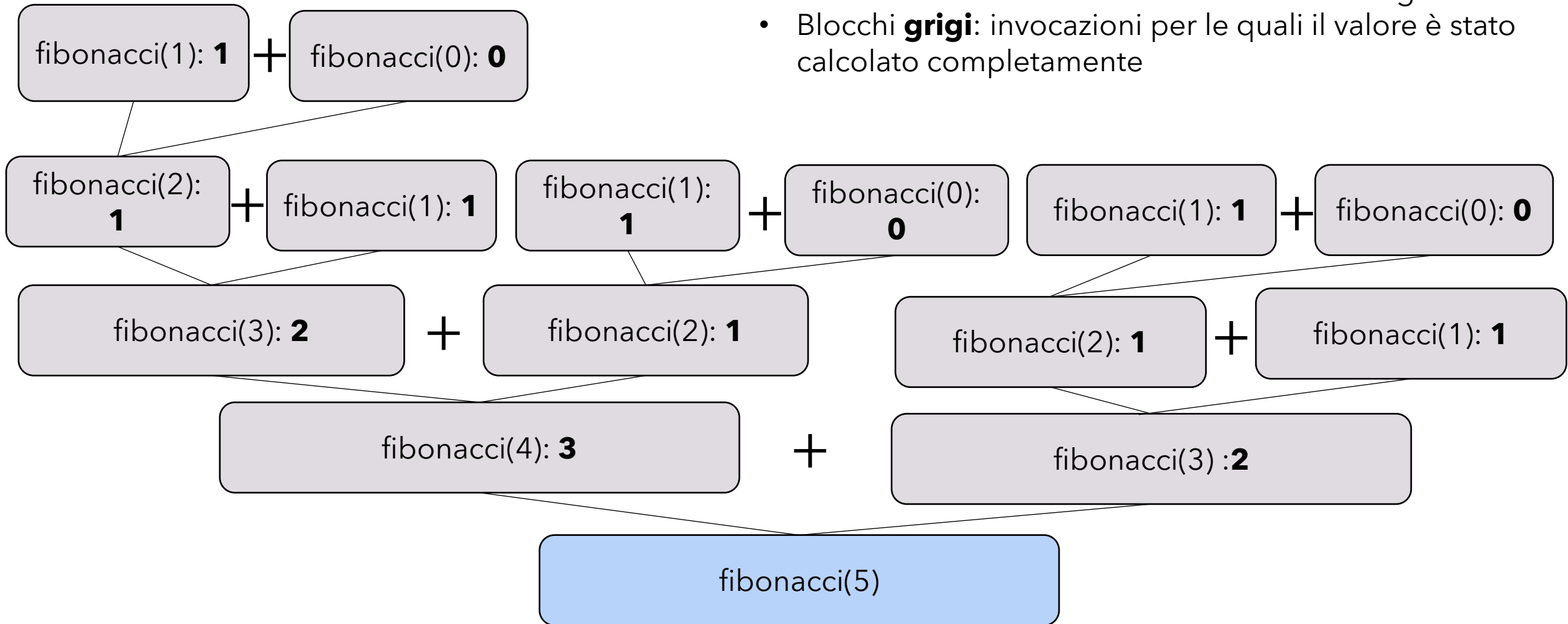
Fibonacci: visualizzazione del calcolo con l'albero delle chiamate

- Blocchi **azzurri**: invocazione corrente
- Blocchi **bianchi**: invocazioni non ancora eseguite
- Blocchi **grigi**: invocazioni per le quali il valore è stato calcolato completamente



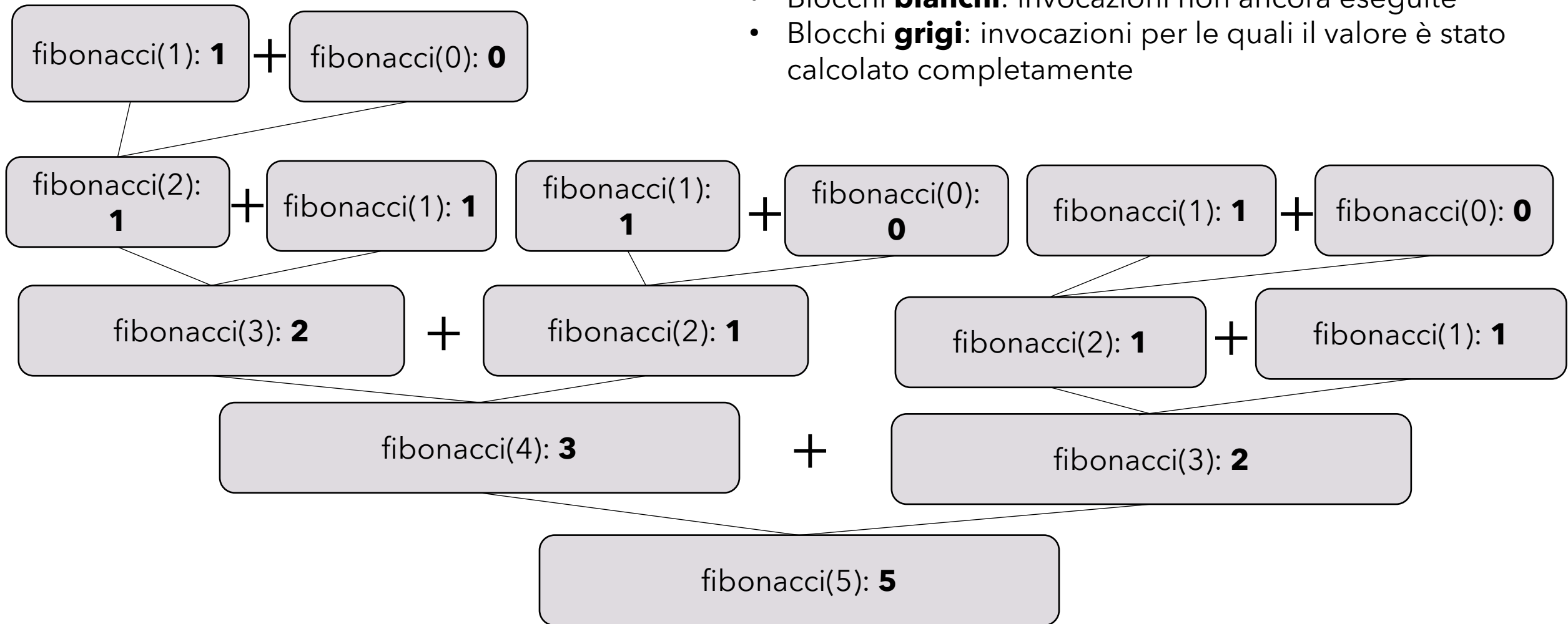
Fibonacci: visualizzazione del calcolo con l'albero delle chiamate

- Blocchi **azzurri**: invocazione corrente
- Blocchi **bianchi**: invocazioni non ancora eseguite
- Blocchi **grigi**: invocazioni per le quali il valore è stato calcolato completamente



Fibonacci: visualizzazione del calcolo con l'albero delle chiamate

- Blocchi **azzurri**: invocazione corrente
- Blocchi **bianchi**: invocazioni non ancora eseguite
- Blocchi **grigi**: invocazioni per le quali il valore è stato calcolato completamente



Fibonacci: visualizzazione del calcolo con l'albero delle chiamate

non vi sembra di aver calcolato più volte le stesse cose?
Guardate ad esempio quante volte viene risolto il
«sottoproblema» fibonacci(2)

