CS102/IT102
Computer Programming I

Lecture 13:
Recursion

Bicol University College of Science
CSIT Department
1st Semester, 2023-2024

Topics

- Recursion
- Unary Recursion
- *N*-ary Recursion

Recursion

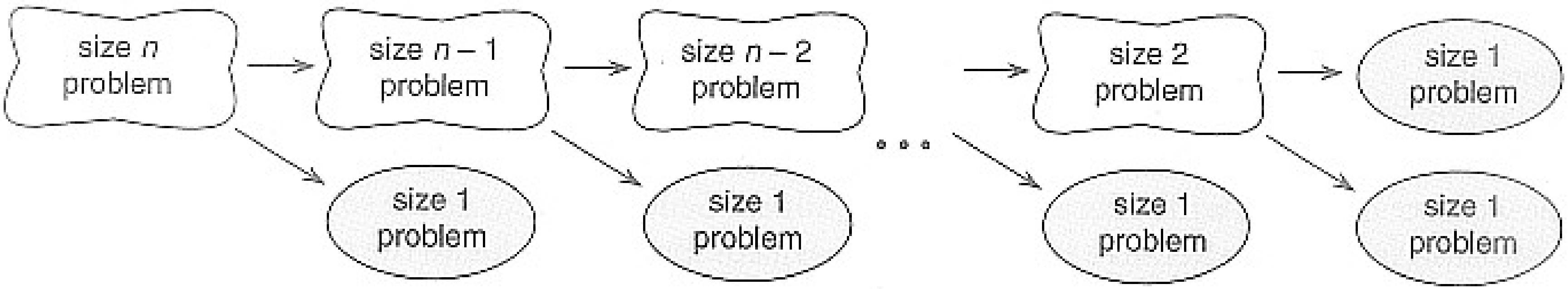
- Queue processing
- Sorting
- Searching

What is Recursion?

- A procedure defined in terms of (simpler versions of) itself
- A process which a function calls itself directly or indirectly
- Components:
  - Base case: One or more simple cases of the problem have a direct and easy answer
  - Recursive definition: The other cases can be re-defined in terms of a similar but smaller problem
  - Convergence to base case: By applying the re-definition process each time, the recursive cases will move closer and eventually reach the base case

What is Recursion?

- The strategy in recursive solutions is called divide-and-conquer. The idea is to keep reducing the problem size until it reduces to the simple case which has an obvious solution.



Format of Recursive Functions

- Recursive functions generally involve an if statement with the following form:

```
if this is a simple case
  solve it
else
  redefine the problem using recursion
```

- The ***if*** branch is the base case, while the ***else*** branch is the recursive case.
- The recursive step provides the repetition needed for the solution and the base step provides the termination

Note: For the recursion to terminate, the recursive case must be moving closer to the base case with each recursive call.

Recursion

If we use iteration, we must be careful not to create an infinite loop by accident:

```
for(int incr=1; incr!=10;incr+=2)
   ...
```

Oops!

```
int result = 1;
while(result >0){
   ...
   result++;
}
```

Oops!

Recursion

Similarly, if we use recursion we must be careful not to create an infinite chain of function calls:

```
int fac(int numb){
   return numb * fac(numb-1);
}
```

Oops!

No termination condition

Or:

```
int fac(int numb){
   if (numb<=1)
      return 1;
   else
      return numb*fac(numb+1);
}
```

Oops!

Recursion

We must always make sure that the recursion *bottoms out*:

- A recursive function must contain at least one non-recursive branch.
- The recursive calls must eventually lead to a non-recursive branch.

Example: Queue processing

```
procedure ProcessQueue ( queue )
{
  if ( queue not empty ) then
  {
    process first item in queue
    remove first item from queue
    ProcessQueue ( rest of queue )
  }
}
```

Example: Queue processing

```
procedure ProcessQueue ( queue )
{
   if ( queue not empty ) then
   {
      process first item in queue
      remove first item from queue
      ProcessQueue ( rest of queue )
   }
}
```

Base case: queue is empty

Example: Queue processing

```
procedure ProcessQueue ( queue )
{
   if ( queue not empty ) then
   {
      process first item in queue
      remove first item from queue
      ProcessQueue ( rest of queue )
   }
}
```

Recursion: call to ProcessQueue

Example: Queue processing

```
procedure ProcessQueue ( queue )
{
  if ( queue not empty ) then
  {
    process first item in queue
    remove first item from queue
    ProcessQueue ( rest of queue )
  }
}
```

Convergence: fewer items in queue

Unary Recursion

- Functions calls itself once (at most)
- Usual format:

```
function RecursiveFunction ( <parameter(s)> )
{
    if ( <base case> ) then
        return <base value>
    else
        return RecursiveFunction ( <expression> )
}
```

- Winding and unwinding the “stack frames”

Example: Factorial

Given $n \geq 0$:

$$n! = n \times (n - 1) \times (n - 2) \times \ldots \times 2 \times 1$$

$$0! = 1$$

Example:

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

Example: Factorial

- Iterative version

```
int factorial (int n)
{
    int i, product=1;
    for (i=n; i>1; --i)
        product=product * i;

    return product;
}
```

Example: Factorial

- *Problem:* Write a recursive function **`Factorial(n)`** which computes the value of $n!$
- *Base Case*:

  If $n = 0$ or $n = 1$:

  Factorial(n) = 1

Example: Factorial

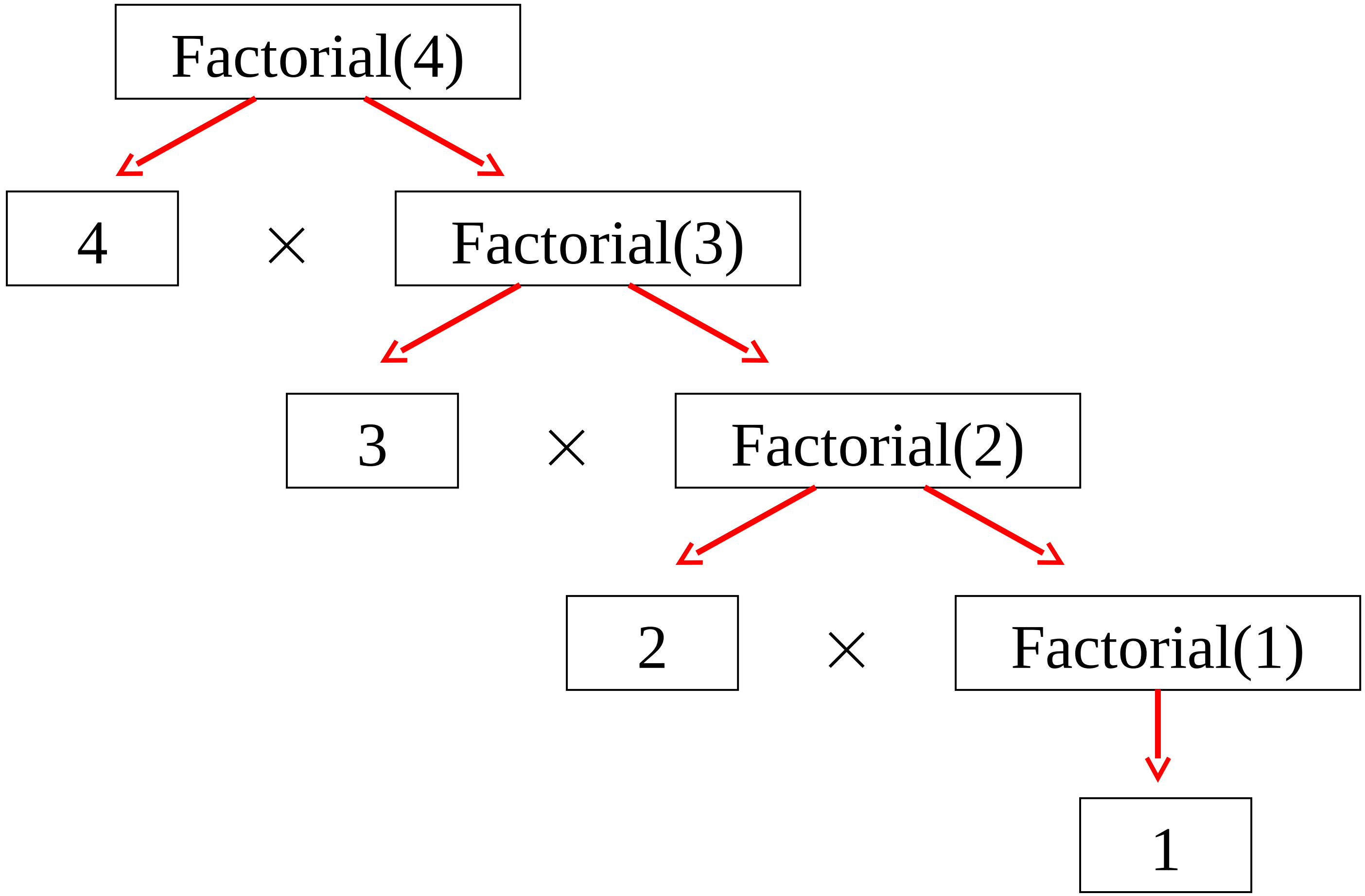
- *Recursion:*

$$n! = n \times \underbrace{(n - 1) \times (n - 2) \times \ldots \times 2 \times 1}_{(n-1)!}$$

If $n > 1$:
  Factorial(n) = $n \times$ Factorial(n - 1)

Example: Factorial

- *Convergence*:



Example: Factorial

The Factorial function can be defined recursively as follows:

$$\text{Factorial}(0) = 1$$

$$\text{Factorial}(1) = 1$$

$$\text{Factorial}(n) = n \times \text{Factorial}(n - 1)$$

Example: Factorial

```
function Factorial ( n )
{
    if ( n is less than or equal to 1 ) then
        return 1
    else
        return n × Factorial ( n - 1 )
}
```

Example: Factorial

Base case

```
function Factorial ( n )
{
    if ( n is less than or equal to 1 ) then
        return 1
    else
        return n × Factorial ( n - 1 )
}
```

Example: Factorial

General Case

```
function Factorial ( n )
{
    if ( n is less than or equal to 1 ) then
        return 1
    else
        return n × Factorial ( n - 1 )
}
```

Example: Factorial

Recursion

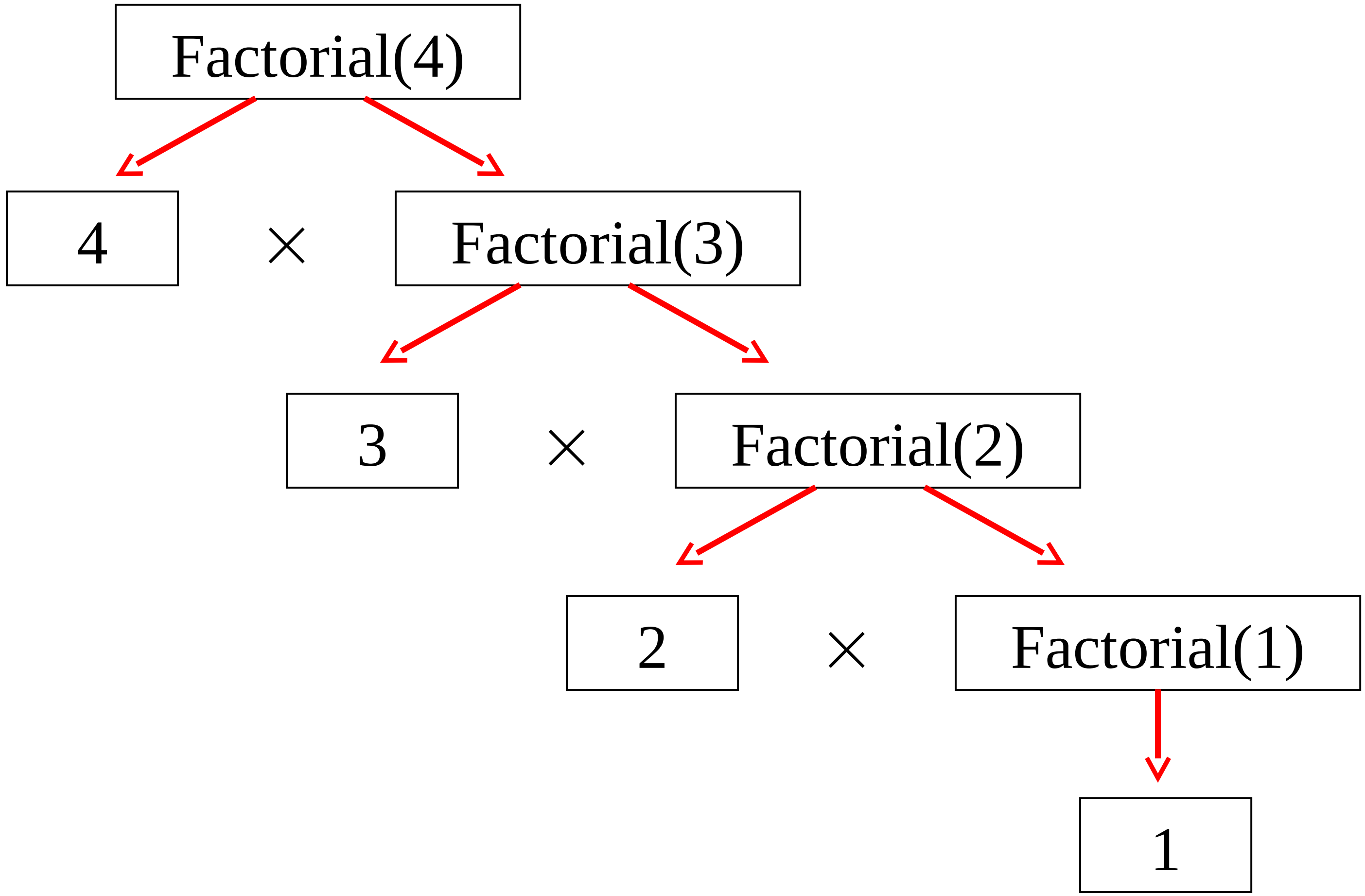
```
function Factorial ( n )
{
    if ( n is less than or equal to 1 ) then
        return 1
    else
        return n × Factorial ( n - 1 )
}
```

Example: Factorial

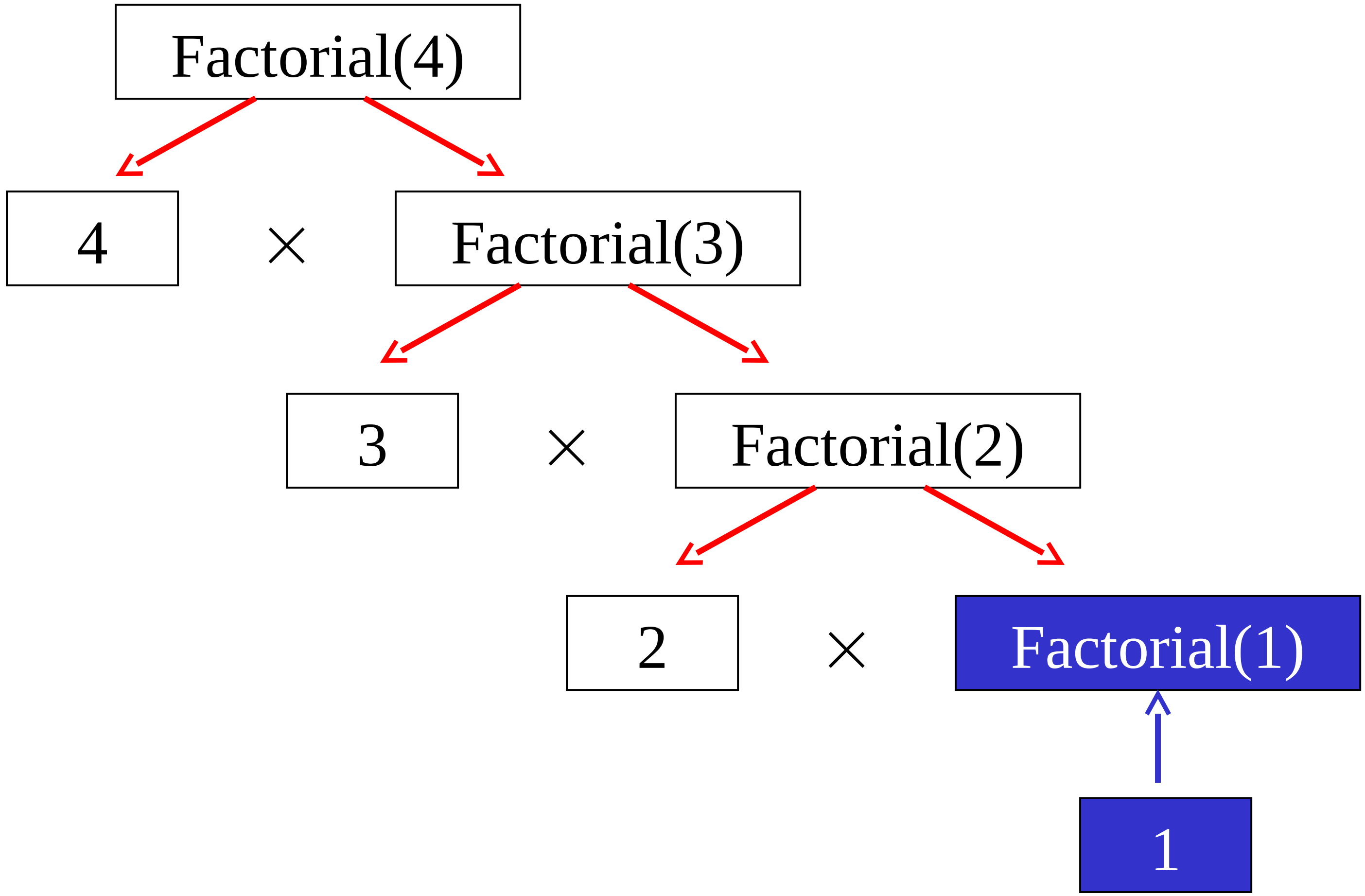
Convergence

```
function Factorial ( n )
{
    if ( n is less than or equal to 1 ) then
        return 1
    else
        return n × Factorial ( n - 1 )
}
```

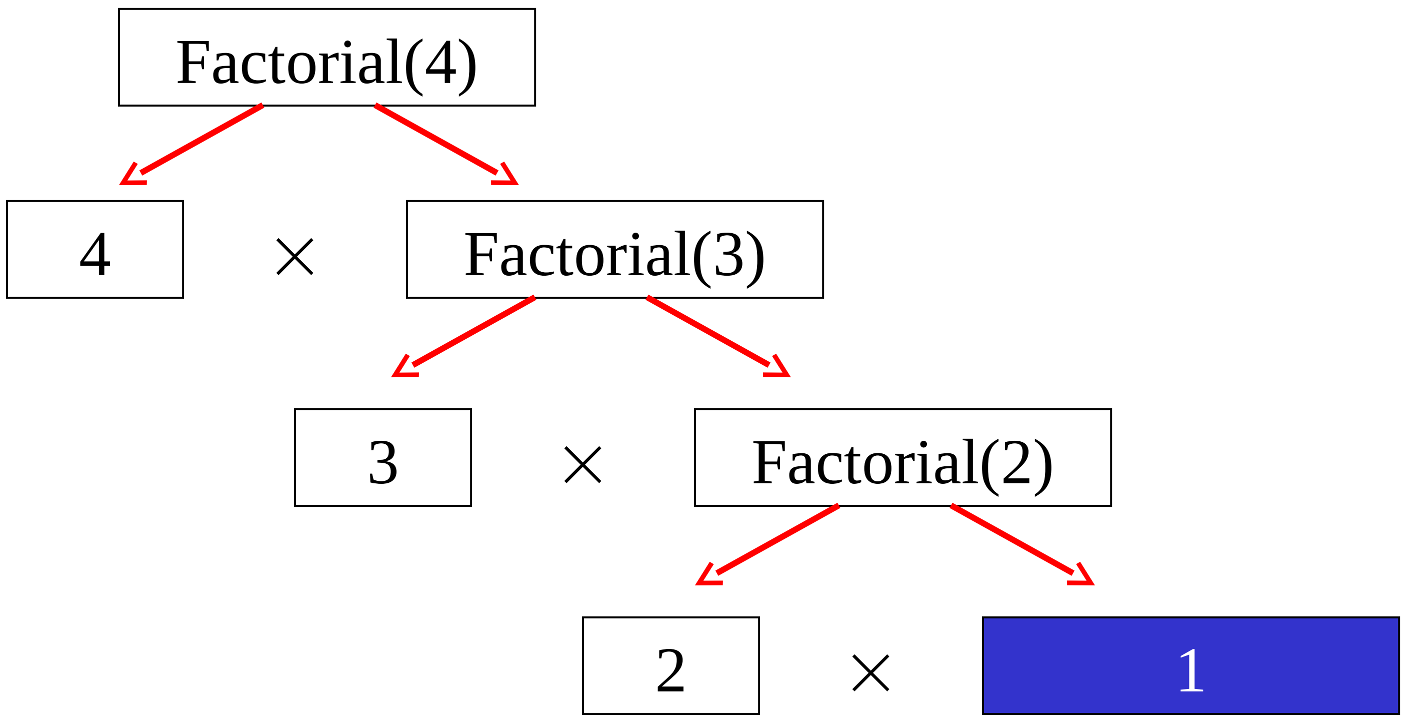
Example: Factorial



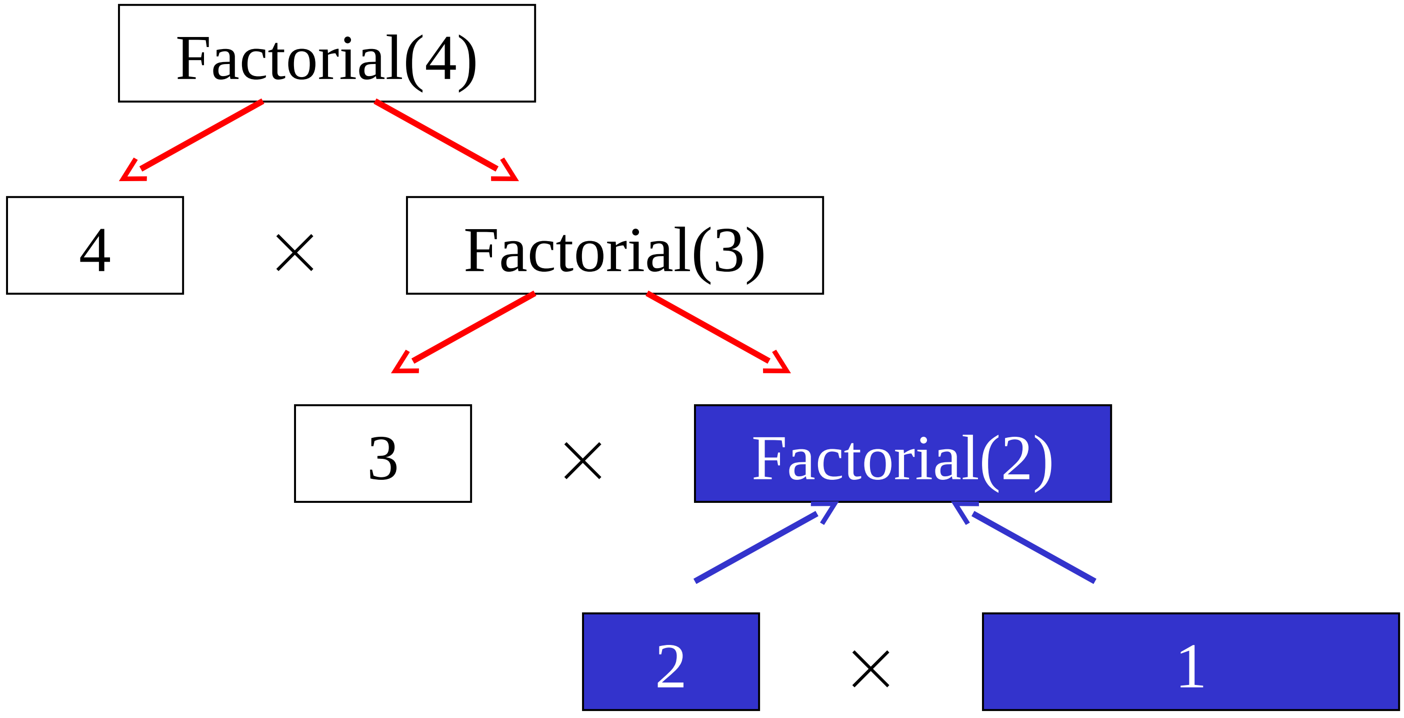
Example: Factorial



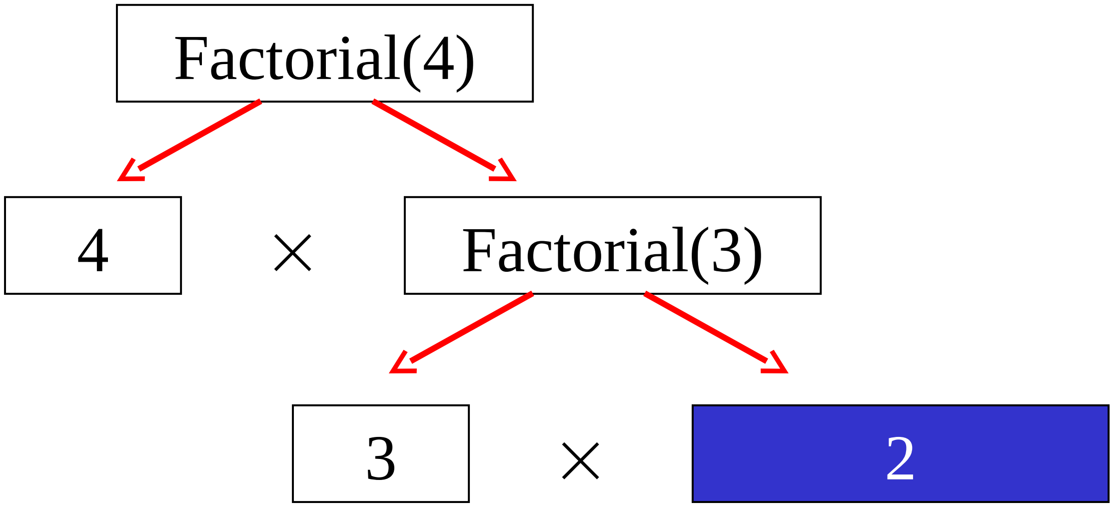
Example: Factorial



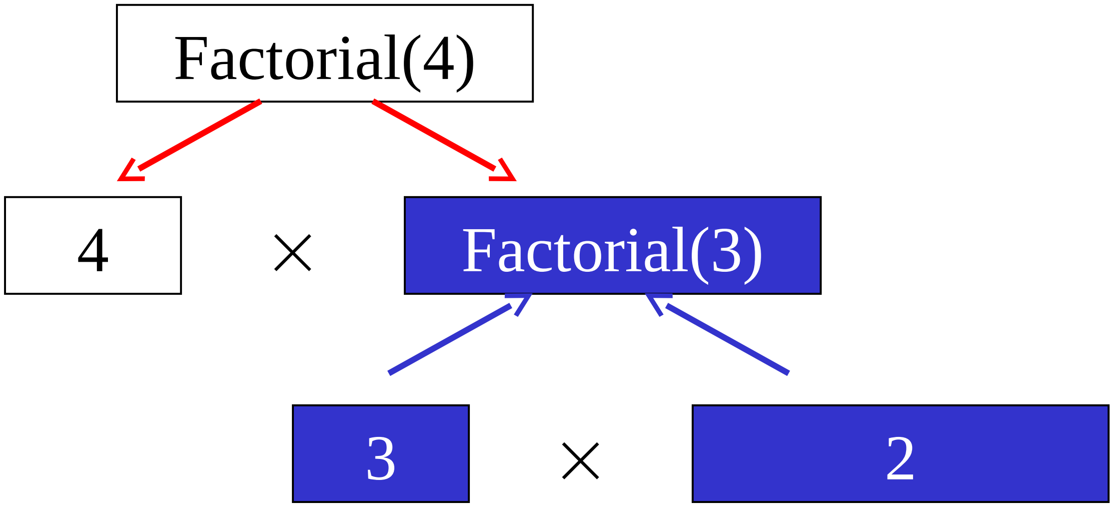
Example: Factorial



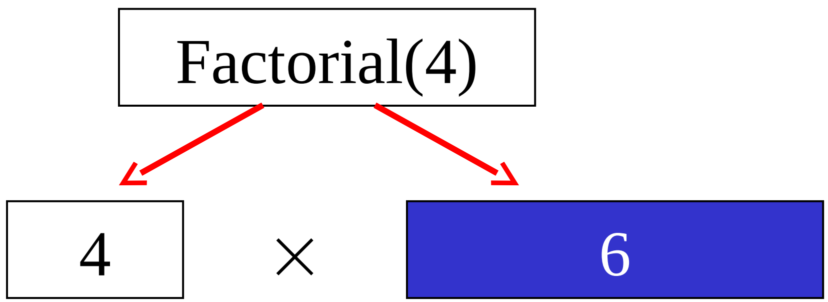
Example: Factorial



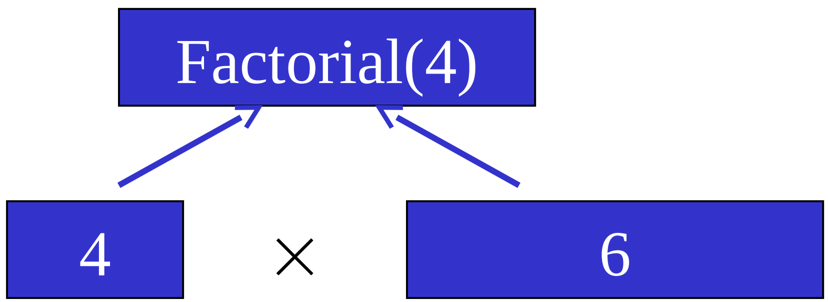
Example: Factorial



Example: Factorial



Example: Factorial



Example: Factorial

24

Example: factorl.c

Computes the factorial of a number
 	
function Factorial (n)
{
if (n is less than or equal to 1)
then
return 1
else
return n $\times$ Factorial (n - 1)
}
	`/* Compute the factorial of n */`
	
	`int factorial ( int n )`
	`{`
	`if ( n <= 1 )`
	`{`
	`return 1;`
	`}`
	`else`
	`{`
	`return n * factorial(n-1);`
	`}`
	`}`

Tracing Recursive Functions

- Executing recursive algorithms goes through two phases:
  - Expansion in which the recursive step is applied until hitting the base step
  - "Substitution" in which the solution is constructed backwards starting with the base step

Expansion phase

$$
\begin{aligned}
\text{factorial}(4) \quad &= 4 * \text{factorial } (3) \\
&= 4 * (3 * \text{factorial } (2)) \\
&= 4 * (3 * (2 * \text{factorial } (1))) \\
\hline
&= 4 * (3 * (2 * 1)) \\
&= 4 * (3 * 2) \\
&= 4 * 6 \\
&= 24
\end{aligned}
$$

Substitution phase

Example: “Frames” during calculation of `factorial(4)`

```
printf(“%d”, factorial(4));
```

Example: “Frames” during calculation of `factorial(4)`

```
printf("%d", factorial(4));
```

```
int factorial ( int n )            n: 4
{
    if ( 4 <= 1 )
    {
        return 1;
    }
    else
    {
        return 4 * factorial( 4 - 1 );
    }
}
```

Example: "Frames" during calculation of `factorial(4)`

```
printf("%d", factorial(4));
```

```
int factorial ( int n )        n: 4
{
    if ( 4 <= 1 )
    {
        return 1;
    }
    else
    {
        return 4 * factorial( 3 );
    }
}
```

Example: “Frames” during calculation of `factorial(4)`

```
printf(“%d”, factorial(4));
```

```
int
{

}
```

n: 3

```
int factorial ( int n )
{
   if ( 3 <= 1 )
   {
      return 1;
   }
   else
   {
      return 3 * factorial( 3 - 1 );
   }
}
```

Example: “Frames” during calculation of `factorial(4)`

```
printf("%d", factorial(4));
```

```
int
{

}
```

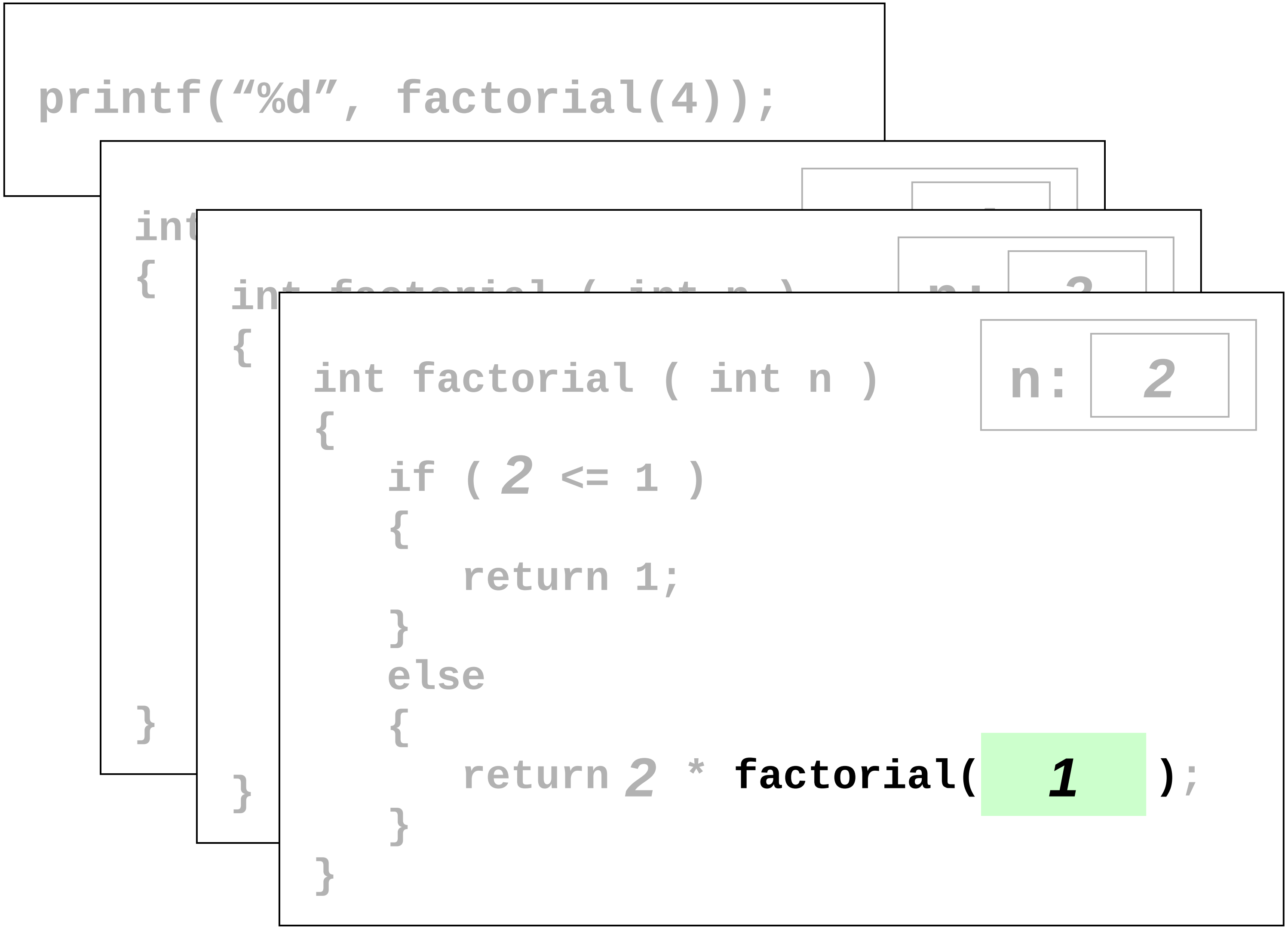
```
int factorial ( int n )                n: 3
{
    if ( 3 <= 1 )
    {
        return 1;
    }
    else
    {
        return 3 * factorial( 2 );
    }
}
```

Example: "Frames" during calculation of `factorial(4)`

```
printf("%d", factorial(4));
```

```
int factorial ( int n )                  n: 2
{
   if ( 2 <= 1 )
   {
      return 1;
   }
   else
   {
      return 2 * factorial( 2 - 1 );
   }
}
```

Example: "Frames" during calculation of `factorial(4)`



Example: “Frames” during calculation of `factorial(4)`

```
printf(“%d”, factorial(4));
```

```
int factorial ( int n )
{
   if ( 1 <= 1 )
   {
      return 1;
   }
   else
   {
      return n * factorial( n - 1 );
   }
}
```

n: 1

Example: "Frames" during calculation of **`factorial(4)`**

```
printf("%d", factorial(4));
```

```
int factorial ( int n )          n: 1
{
    if ( 1 <= 1 )
    {
        return 1 ;
    }
    else
    {
        return n * factorial( n - 1 );
    }
}
```

Base case:

`factorial(1) is 1`

Example: "Frames" during calculation of **`factorial(4)`**

```
printf("%d", factorial(4));
```

```
int factorial ( int n )
{
    if ( 2 <= 1 )
    {
        return 1;
    }
    else
    {
        return 2 * factorial( 1 );
    }
}
```

n: 2

1

Example: "Frames" during calculation of **`factorial(4)`**

```
printf("%d", factorial(4));
```

```
int factorial ( int n )
{
    if ( 2 <= 1 )
    {
        return 1;
    }
    else
    {
        return 2 * factorial( 2 - 1 );
    }
}
```

n: 2

$\underbrace{\texttt{factorial(2 - 1)}}_{1}$

Example: "Frames" during calculation of **factorial(4)**

```
printf("%d", factorial(4));
```

```
int factorial ( int n )                n: 2
{
    if ( 2 <= 1 )
    {
        return 1;
    }
    else
    {
        return         2         ;
    }
}
```

factorial(2) is 2

Example: "Frames" during calculation of `factorial(4)`

```
printf("%d", factorial(4));
```

```
int
{

}
```

```
int factorial ( int n )        n: 3
{
    if ( 3 <= 1 )
    {
        return 1;
    }
    else
    {
        return 3 * factorial( 2 );
    }
}
```

2

Example: "Frames" during calculation of `factorial(4)`

```
printf("%d", factorial(4));
```

```
int
{

}
```

```
int factorial ( int n )
{
   if ( 3 <= 1 )
   {
      return 1;
   }
   else
   {
      return 3 * factorial( 3 - 1 );
   }
}
```

n: 3

2

Example: "Frames" during calculation of `factorial(4)`

```
printf("%d", factorial(4));
```

```
int
{
}
```

```
int factorial ( int n )
{
   if ( 3 <= 1 )
   {
      return 1;
   }
   else
   {
      return 6;
   }
}
```

n: 3

factorial(3) is 6

Example: “Frames” during calculation of `factorial(4)`

```
printf(“%d”, factorial(4));
```

```
int factorial ( int n )                n: 4
{
   if ( 4 <= 1 )
   {
      return 1;
   }
   else
   {
      return 4 * factorial( 3 );
   }
}
```

$$\underbrace{\texttt{factorial(3)}}_{6}$$

Example: “Frames” during calculation of `factorial(4)`

```
printf(“%d”, factorial(4));
```

n: **4**

```
int factorial ( int n )
{
   if ( 4 <= 1 )
   {
      return 1;
   }
   else
   {
      return 4 * factorial( 4 - 1 );
   }
}
```

6

Example: "Frames" during calculation of `factorial(4)`

```
printf("%d", factorial(4));
```

```
int factorial ( int n )
{
   if ( 4 <= 1 )
   {
      return 1;
   }
   else
   {
      return 24;
   }
}
```

n: 4

factorial(4) is 24

Example: "Frames" during calculation of `factorial(4)`

```
printf("%d", factorial(4));
```

$\underbrace{\texttt{factorial(4)}}_{24}$

***Output:* 24**

Example: testprog.c

```
#include <stdio.h>
#include “factorl.c”

/* Main program for testing factorial() function */

int main(void)
{
    int n;

    printf("Please enter n: ");
    scanf("%d", &n);

    printf("%d! is %d\n", n, factorial(n));

    return 0;
}
```

Example: Multiplication

- Suppose we wish to write a recursive function to multiply an integer m by another integer n using addition. [We can add, but we only know how to multiply by 1].
- The best way to go about this is to formulate the solution by identifying the base case and the recursive case.
- The base case is if n is 1. The answer is m.
- The recursive case is: m*n = m + m (n-1).

$$m*n \begin{cases} m, & n = 1 \\ m + m*(n-1), & n>1 \end{cases}$$

Example: Multiplication

```
#include <stdio.h>

int multiply(int m, int n);

int main(void) {
    int num1, num2;

    printf("Enter two integer numbers to multiply: ");
    scanf("%d%d", &num1, &num2);

    printf("%d x %d = %d\n", num1, num2, multiply(num1, num2));

    return 0;
}

int multiply(int m, int n) {
      if (n == 1)
            return m;     /* simple case */
      else
            return m + multiply(m, n  - 1);  /* recursive step */
}
```

Example: Multiplication

Expansion phase

$$
\begin{aligned}
\text{multiply}(5,4) \quad &= 5 + \text{multiply}(5, 3) \\
&= 5 + (5 + \text{multiply}(5, 2)) \\
&= 5 + (5 + (5 + \text{multiply}(5, 1))) \\
&= 5 + (5 + (5 + 5)) \\
&= 5 + (5 + 10) \\
&= 5 + 15 \\
&= 20
\end{aligned}
$$

Substitution phase

Example: Power function

- Suppose we wish to define our own power function that raise a double number to the power of a non-negative integer exponent. x^n , $n>=0$.
- The base case is if n is 0. The answer is 1.
- The recursive case is: $x^n = x * x^{n-1}$.

$$x^n \begin{cases} 1, & n = 0 \\ x * x^{n-1}, & n>0 \end{cases}$$

Example: Power function

```
#include <stdio.h>

double pow(double x, int n);

int main(void) {
    double x;
    int n;

    printf("Enter double x and integer n to find pow(x,n): ");
    scanf("%lf%d", &x, &n);

    printf("pow(%f, %d) = %f\n", x, n, pow(x, n));

    return 0;
}

double pow(double x, int n) {
      if (n == 0)
          return 1;     /* simple case */
     else
          return x * pow(x, n  - 1);  /* recursive step */
}
```

Example: Power function

Expansion phase

pow(5,3) = 5 * pow(5, 2)

= 5 * (5 * pow(5, 1))

= 5 * (5 * (5 * pow(5, 0)))

= 5 * (5 * (5 * 1))

= 5 * (5 * 5)

= 5 * 25

= 125

Substitution phase

N-ary Recursion

- Sometimes a function can only be defined in terms of two or more calls to itself.
- Efficiency is often a problem.

Example: Fibonacci

- A series of numbers which
  - begins with 0 and 1
  - every subsequent number is the sum of the previous two numbers
- 0, 1, 1, 2, 3, 5, 8, 13, 21,...
- Write a recursive function which computes the *n*-th number in the series (n = 0, 1, 2,...)

Example: Fibonacci

The Fibonacci series can be defined recursively as follows:

Fibonacci(0) = 0

Fibonacci(1) = 1

Fibonacci(n) = Fibonacci(n - 2) + Fibonacci(n - 1)

Example: fibonacc.c

```
function Fibonacci ( n )
{
  if ( n is less than or equal to 1 ) then
    return n
  else
    return Fibonacci ( n - 2 ) + Fibonacci ( n - 1 )
}
```

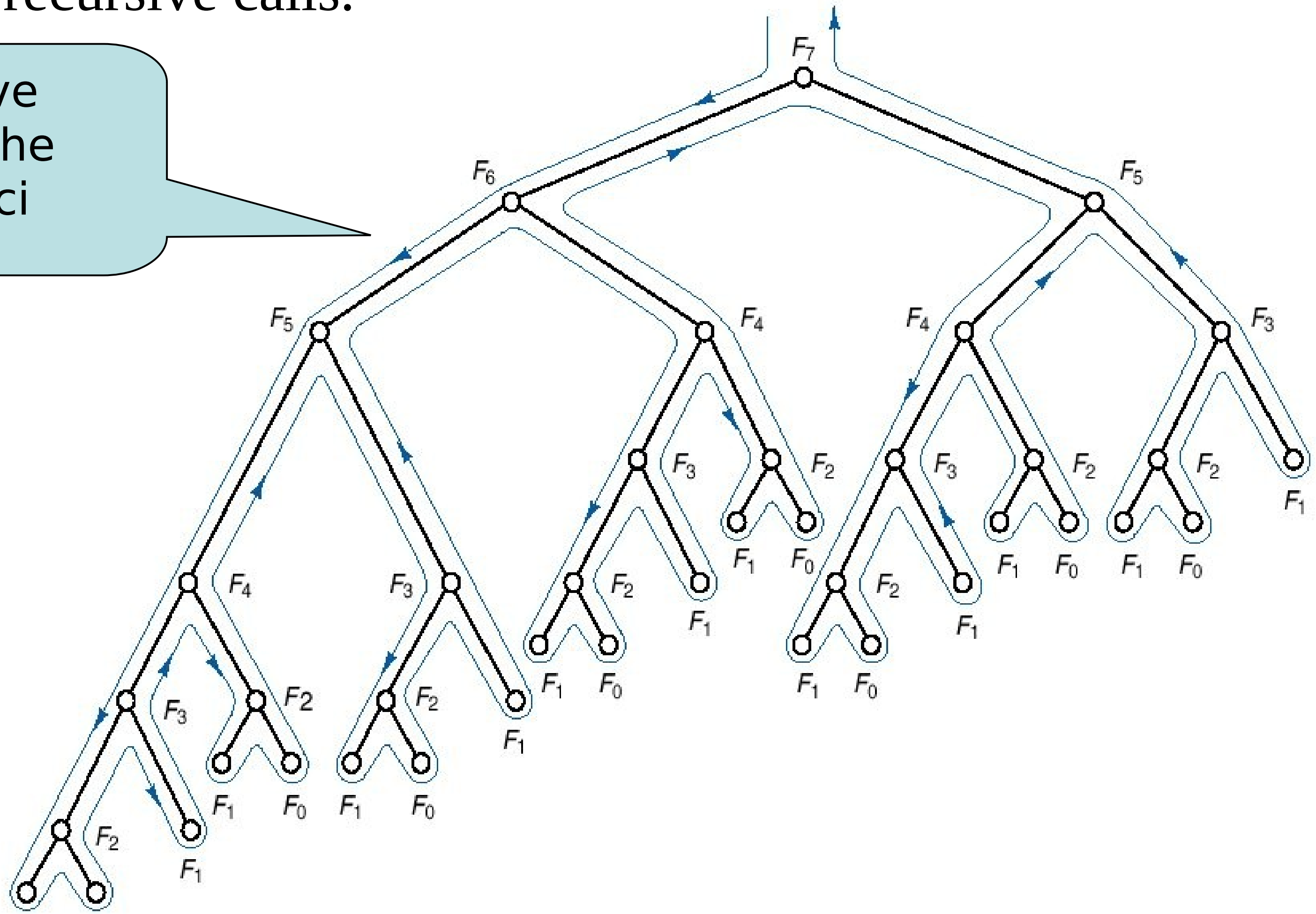
```c
/* Compute the n-th Fibonacci number,
   when=0,1,2,... */

long fib ( long n )
{
   if ( n <= 1 )
      return n ;
   else
      return fib( n - 2 ) + fib( n - 1 );
}
```

Tracing using Recursive Tree

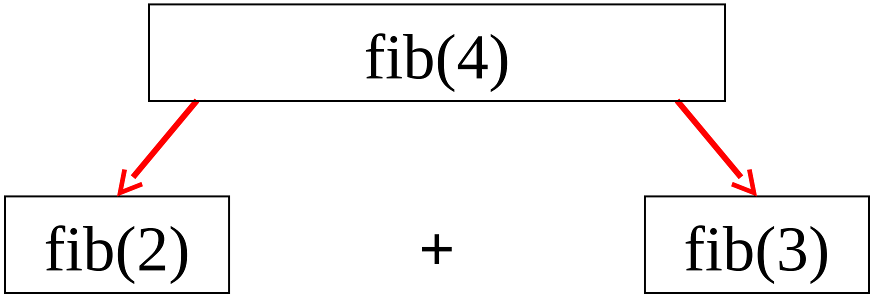
- Another way to trace a recursive function is by drawing its recursive tree.
- This is usually better if the recursive case involves more than one recursive calls.

Recursive tree of the Fibonacci function



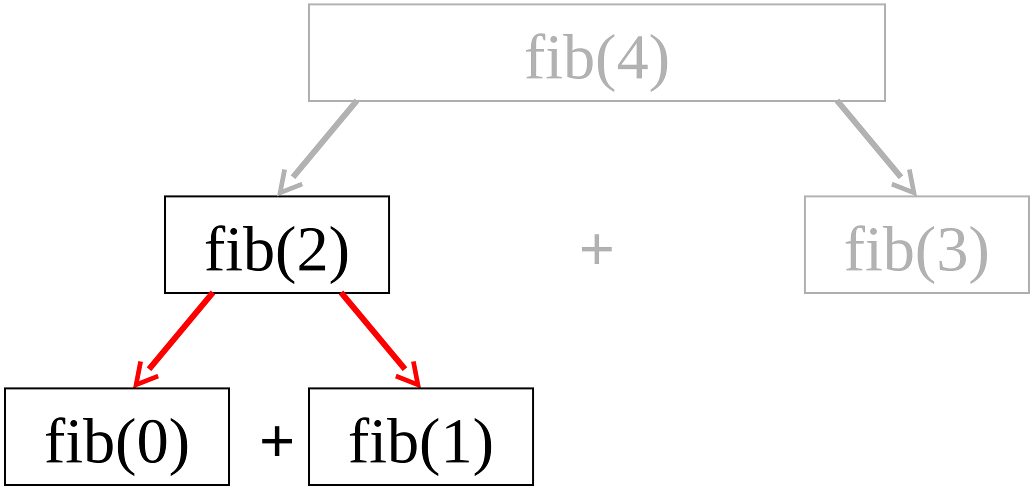
Example: Computation of `fib(4)`

```
long fib ( long 4 )
{
    if ( 4 <= 1 )
        return n ;
    else
        return fib( 4 - 2 ) + fib( 4 - 1 );
}
```



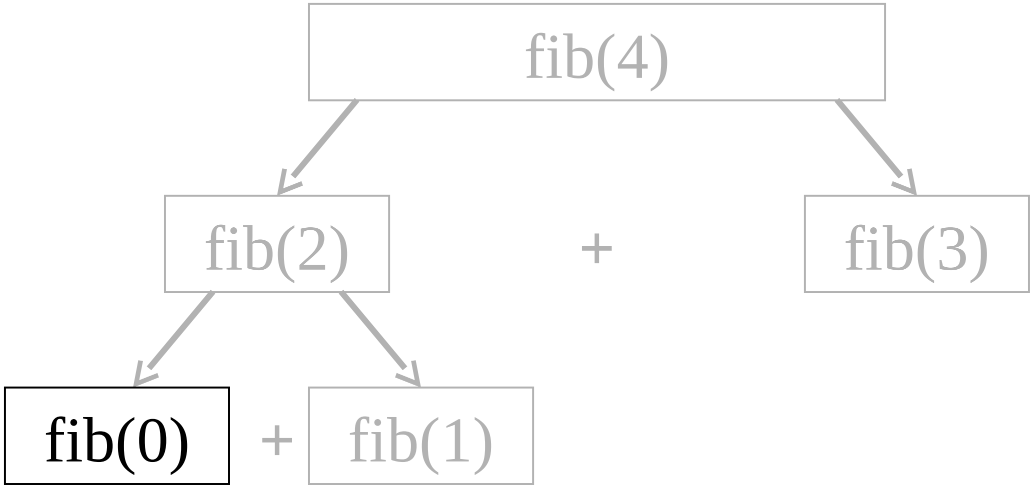
Example: Computation of `fib(4)`

```
long fib ( long 2 )
{
   if ( 2 <= 1 )
      return n ;
   else
      return fib( 2 - 2 ) + fib( 2 - 1 );
}
```



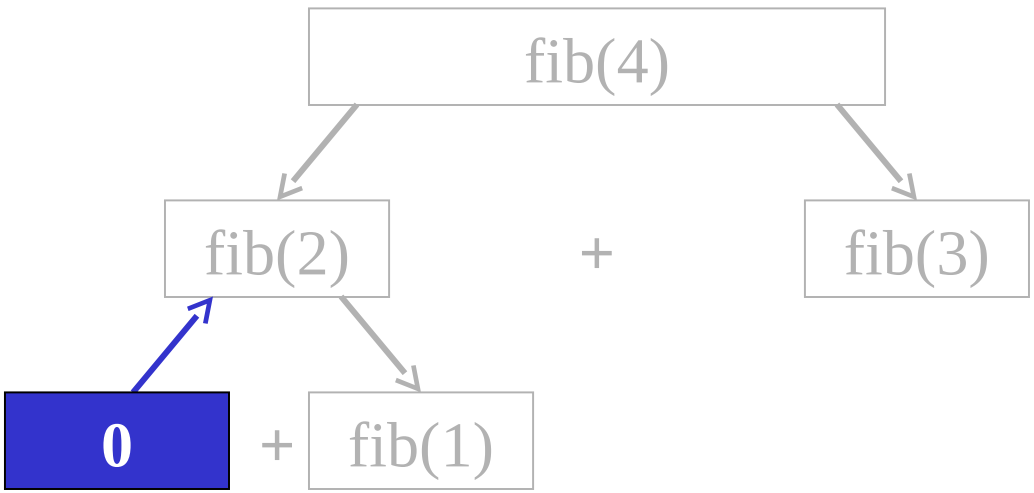
Example: Computation of **`fib(4)`**

```
long fib ( long 0 )
{
    if ( 0 <= 1 )
        return 0 ;
    else
        return fib( n - 2 ) + fib( n - 1 );
}
```



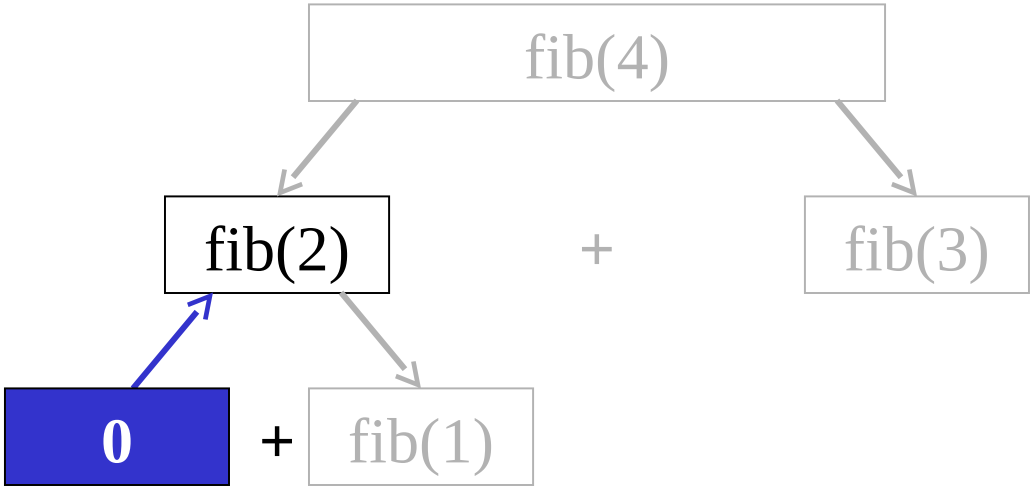
Example: Computation of `fib(4)`

```
long fib ( long 0 )
{
   if ( 0 <= 1 )
      return 0 ;
   else
      return fib( n - 2 ) + fib( n - 1 );
}
```



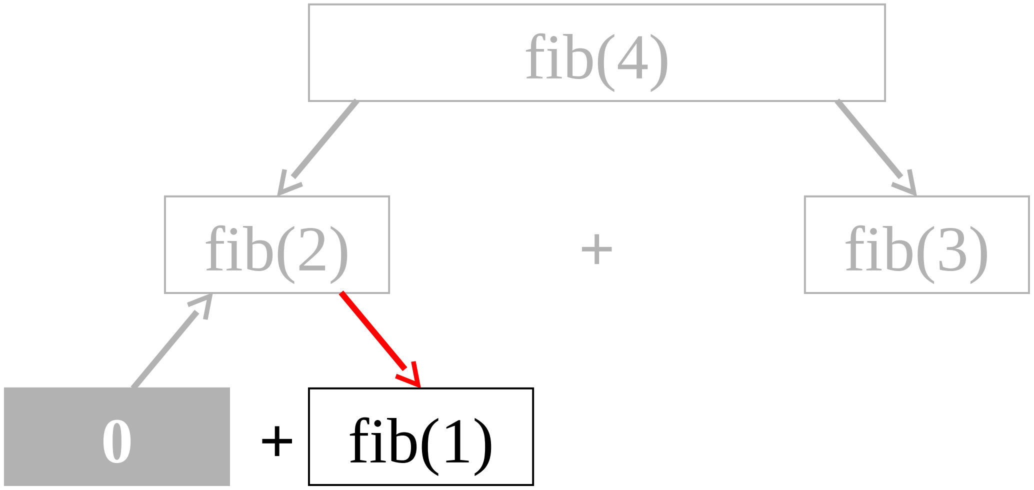
Example: Computation of `fib(4)`

```
long fib ( long 2 )
{
   if ( 2 <= 1 )
      return n ;
   else
      return       0       + fib( 2 - 1 );
}
```



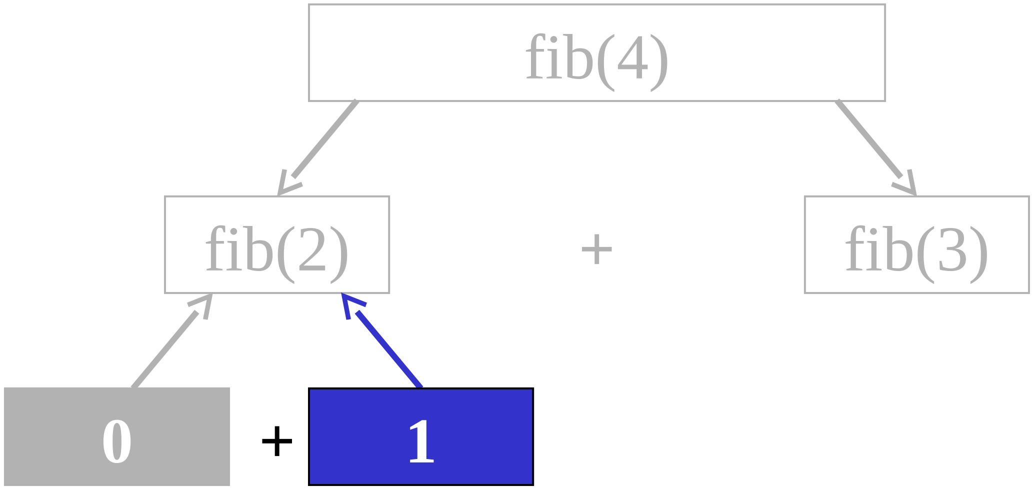
Example: Computation of `fib(4)`

```
long fib ( long 1 )
{
   if ( 1 <= 1 )
      return 1 ;
   else
      return fib( n - 2 ) + fib( n - 1 );
}
```



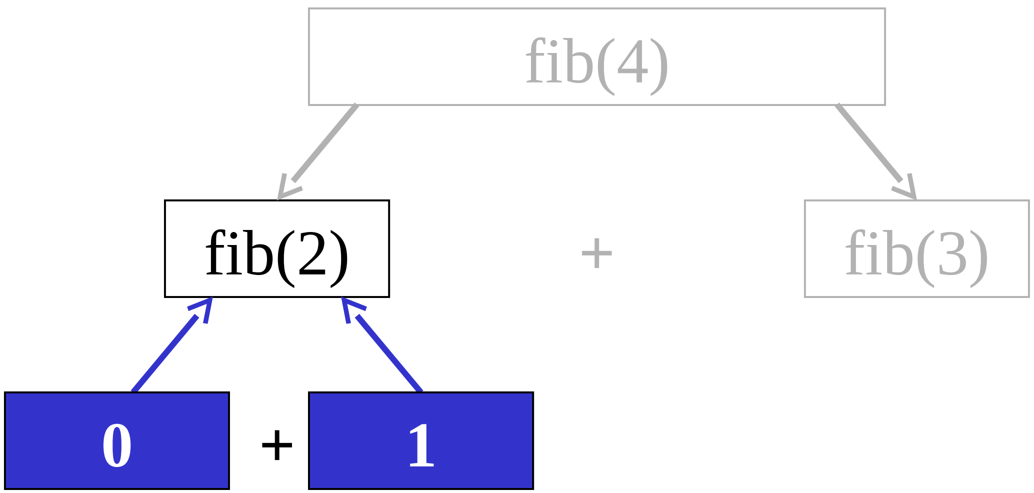
Example: Computation of `fib(4)`

```
long fib ( long 1 )
{
    if ( 1 <= 1 )
        return 1 ;
    else
        return fib( n - 2 ) + fib( n - 1 );
}
```



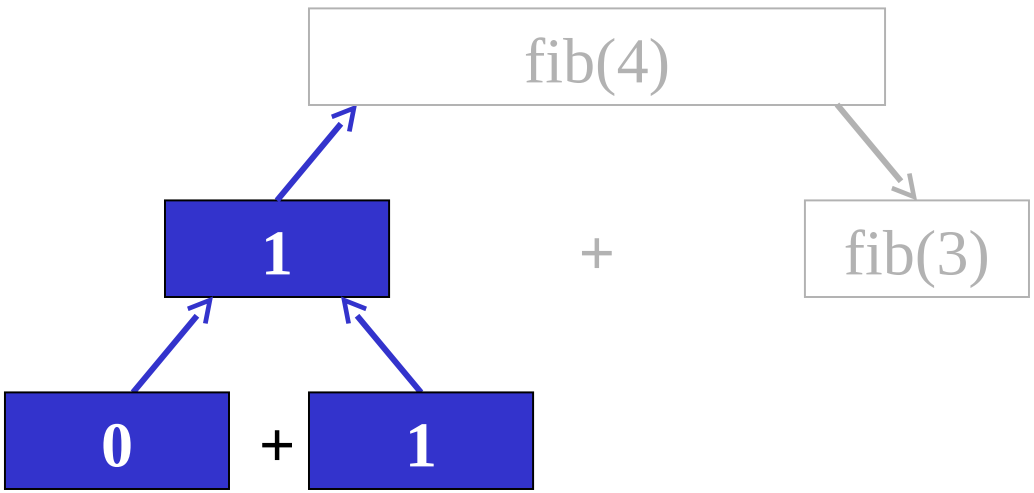
Example: Computation of `fib(4)`

```
long fib ( long 2 )
{
    if ( 2 <= 1 )
        return n ;
    else
        return 0 + 1 ;
}
```



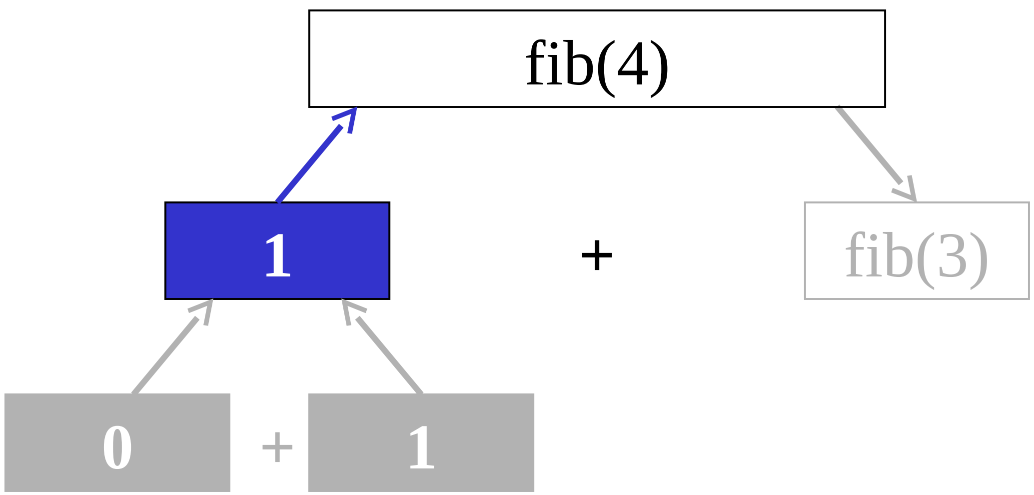
Example: Computation of `fib(4)`

```
long fib ( long 2 )
{
   if ( 2 <= 1 )
      return n ;
   else
      return       0        +       1       ;
}
```



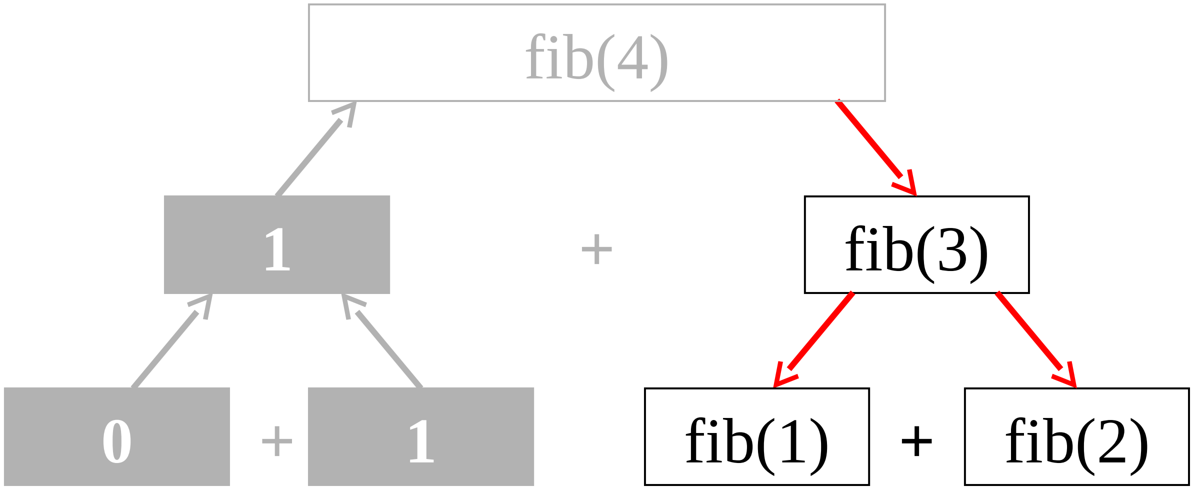
Example: Computation of **`fib(4)`**

```
long fib ( long 4 )
{
   if ( 4 <= 1 )
      return n ;
   else
      return     1      + fib( 4 - 1 );
}
```



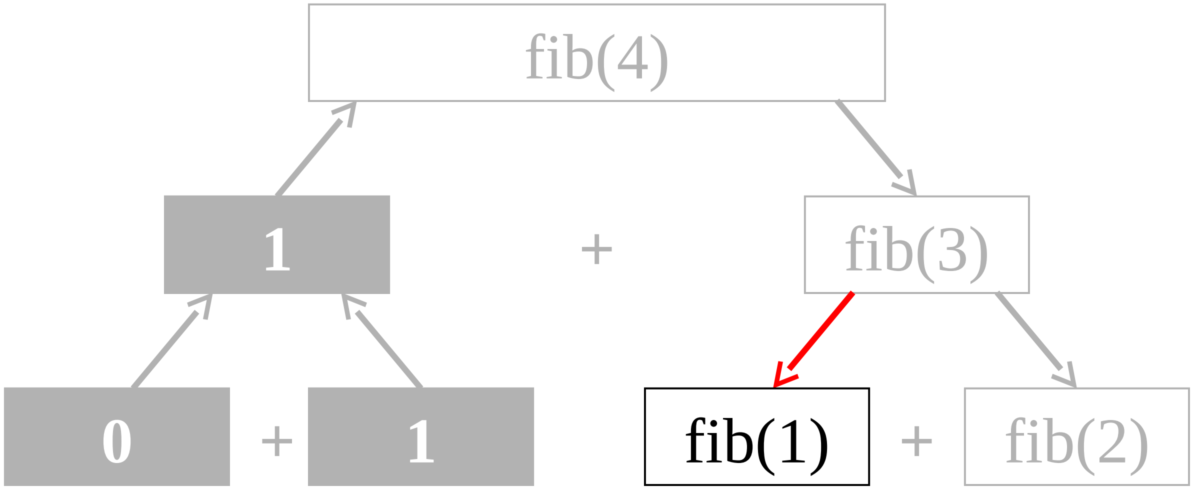
Example: Computation of `fib(4)`

```
long fib ( long 3 )
{
   if ( 3 <= 1 )
      return n ;
   else
      return fib( 3 - 2 ) + fib( 3 - 1 );
}
```



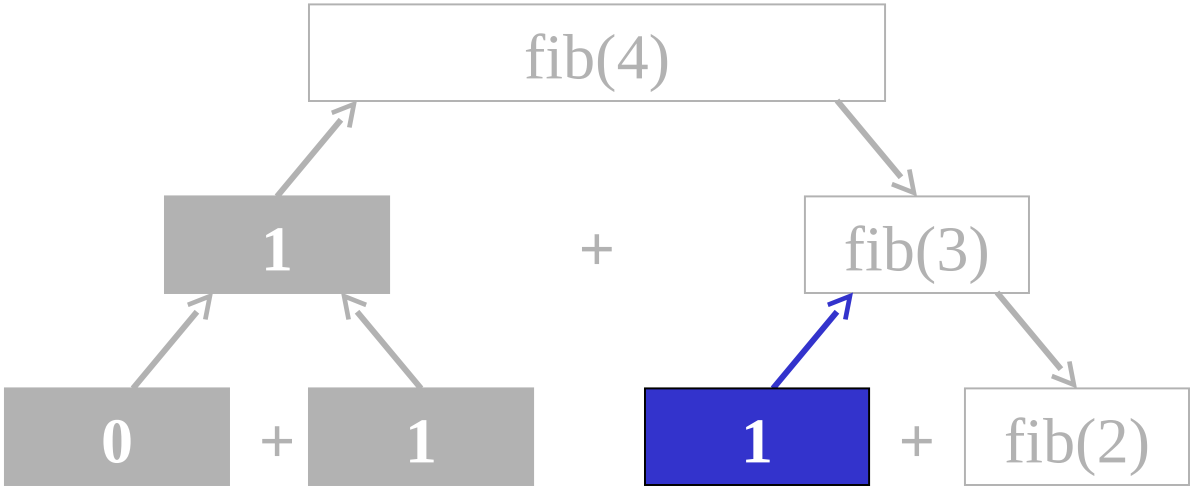
Example: Computation of `fib(4)`

```
long fib ( long 1 )
{
    if ( 1 <= 1 )
        return 1 ;
    else
        return fib( n - 2 ) + fib( n - 1 );
}
```



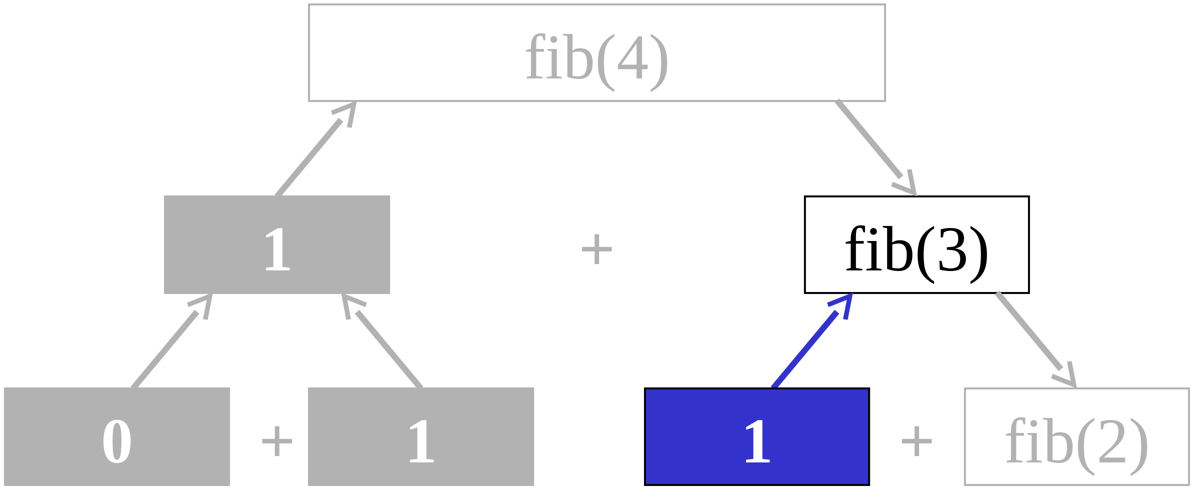
Example: Computation of `fib(4)`

```
long fib ( long 1 )
{
   if ( 1 <= 1 )
      return 1 ;
   else
      return fib( n - 2 ) + fib( n - 1 );
}
```



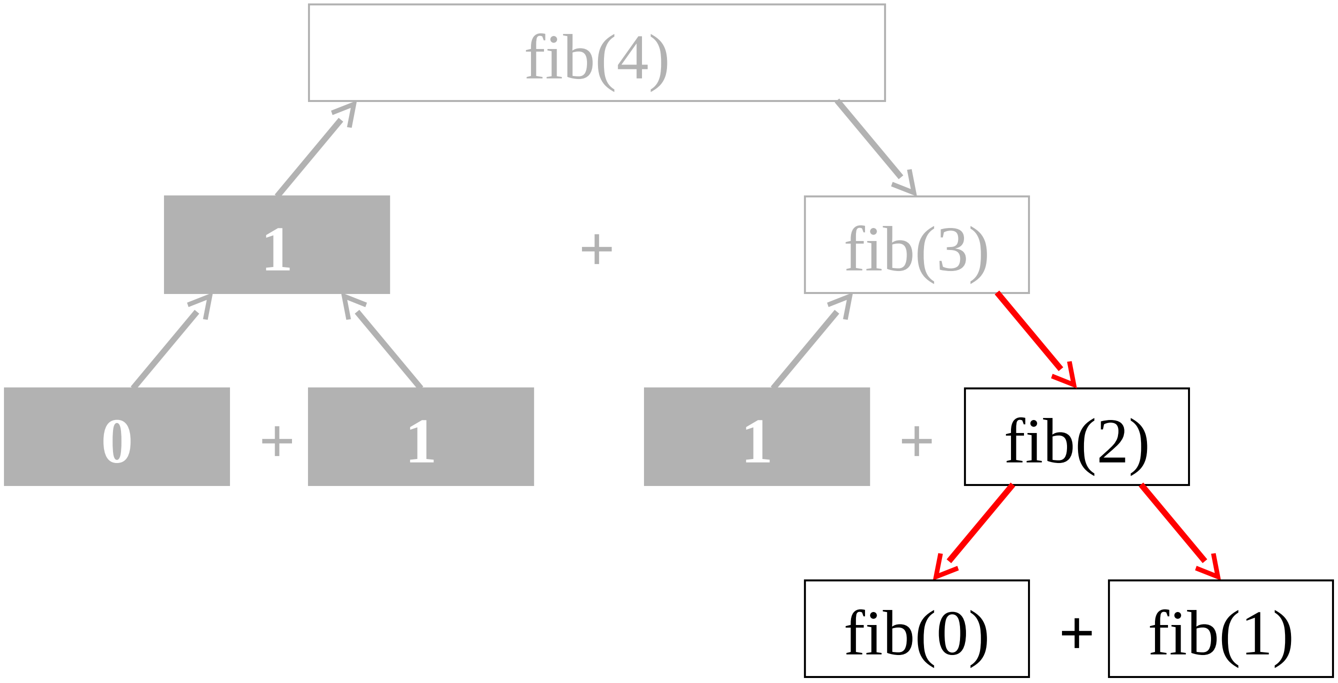
Example: Computation of `fib(4)`

```
long fib ( long 3 )
{
    if ( 3 <= 1 )
        return n ;
    else
        return     1     + fib( 3 - 1 );
}
```



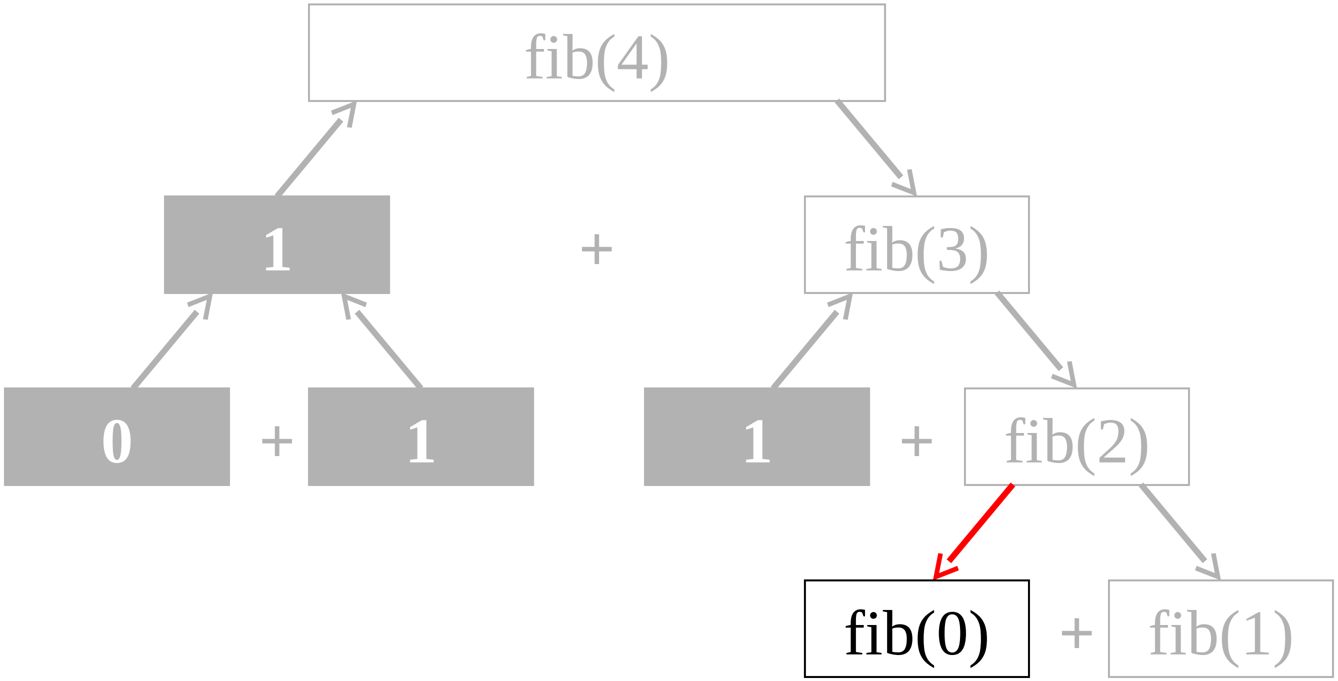
Example: Computation of `fib(4)`

```
long fib ( long 2 )
{
   if ( 2 <= 1 )
      return n ;
   else
      return fib( 2 - 2 ) + fib( 2 - 1 );
}
```



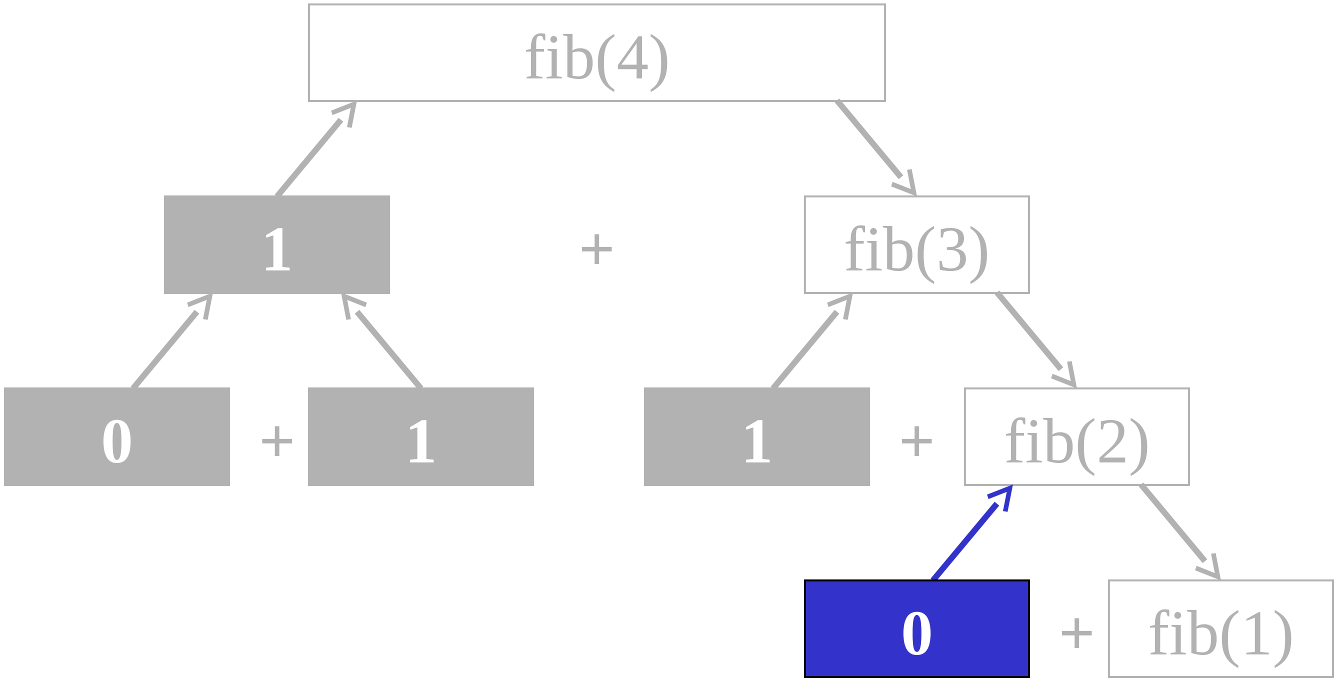
Example: Computation of **fib(4)**

```
long fib ( long 0 )
{
   if ( 0 <= 1 )
      return 0 ;
   else
      return fib( n - 2 ) + fib( n - 1 );
}
```



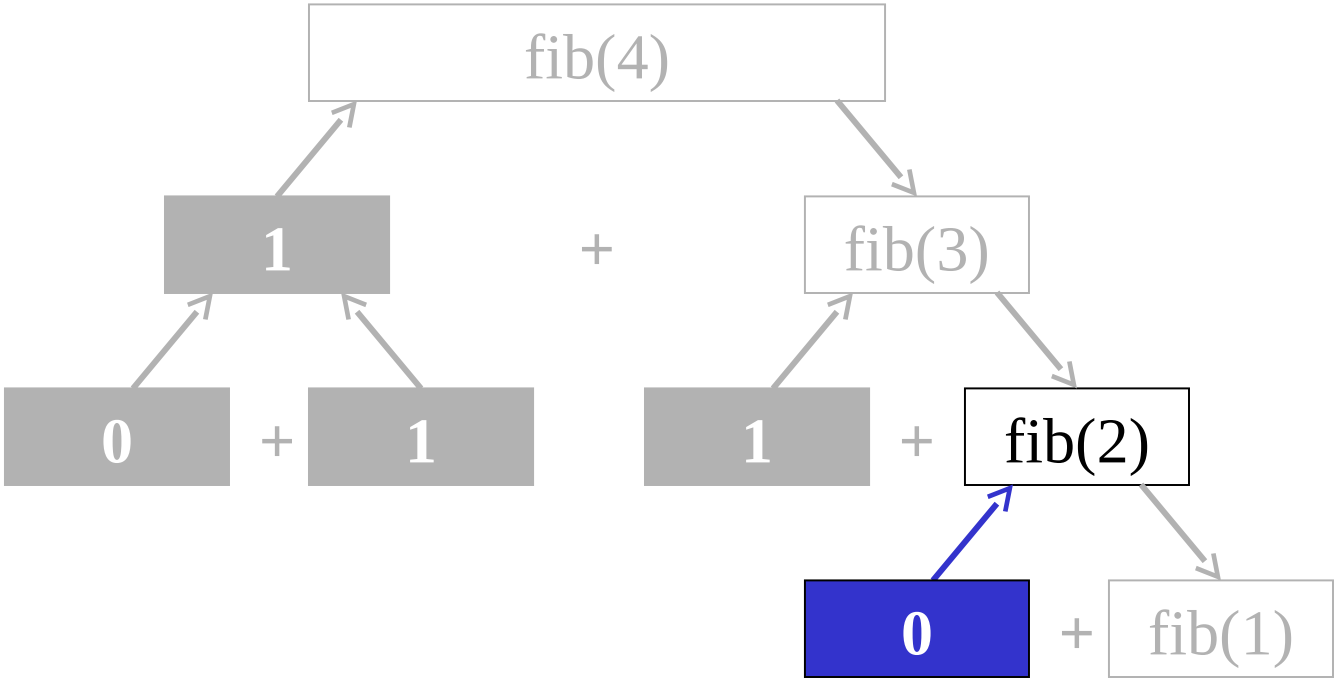
Example: Computation of `fib(4)`

```
long fib ( long 0 )
{
    if ( 0 <= 1 )
        return 0 ;
    else
        return fib( n - 2 ) + fib( n - 1 );
}
```



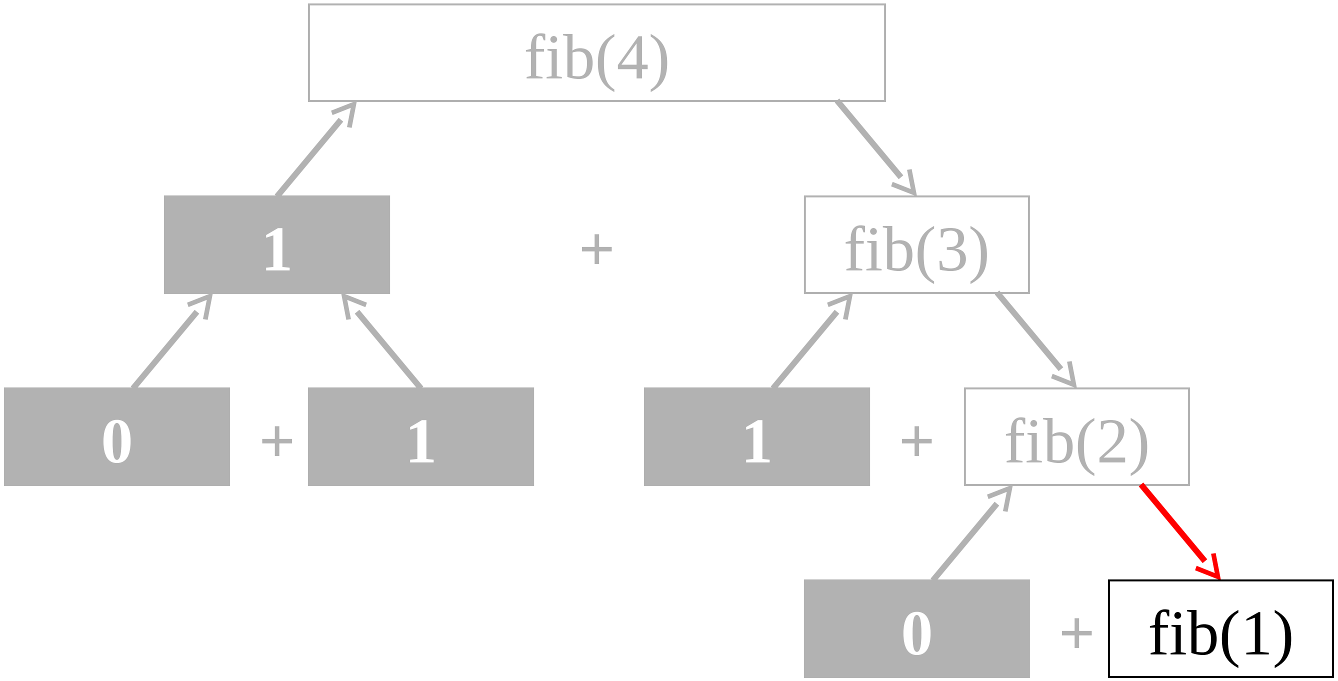
Example: Computation of `fib(4)`

```
long fib ( long 2 )
{
    if ( 2 <= 1 )
        return n ;
    else
        return      0      + fib( 2 - 1 );
}
```



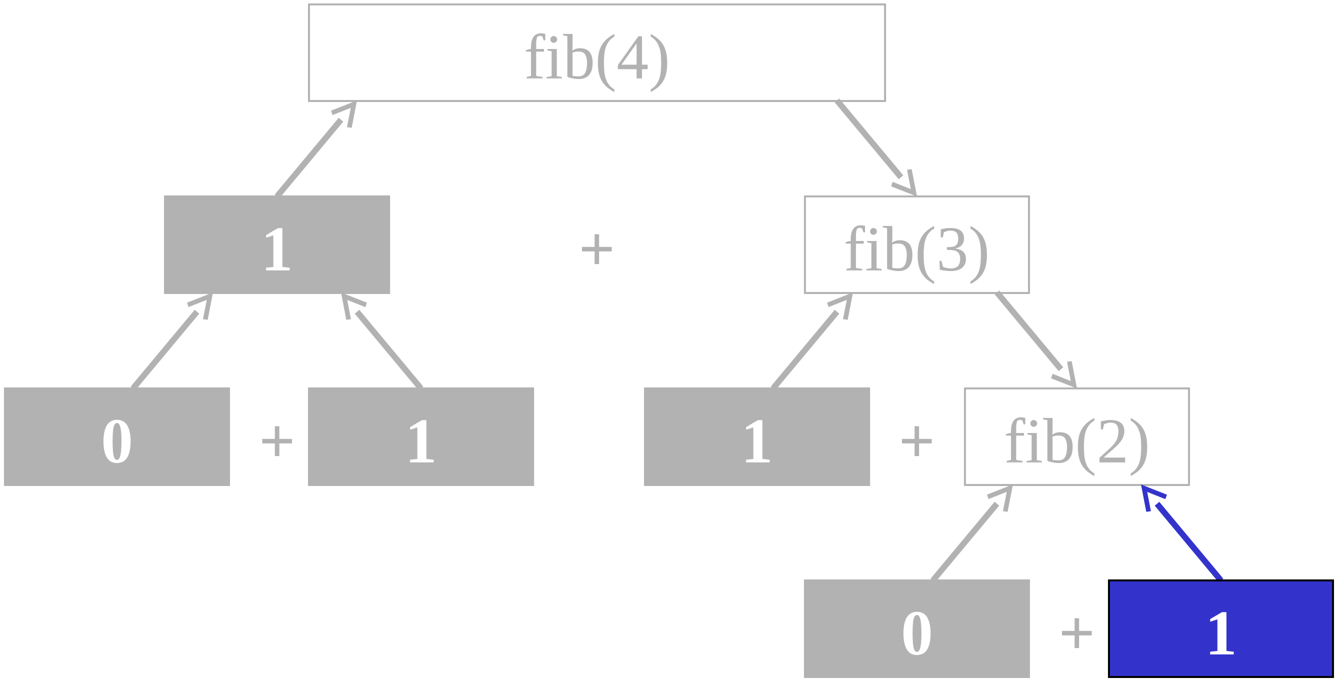
Example: Computation of `fib(4)`

```
long fib ( long 1 )
{
   if ( 1 <= 1 )
      return 1 ;
   else
      return fib( n - 2 ) + fib( n - 1 );
}
```



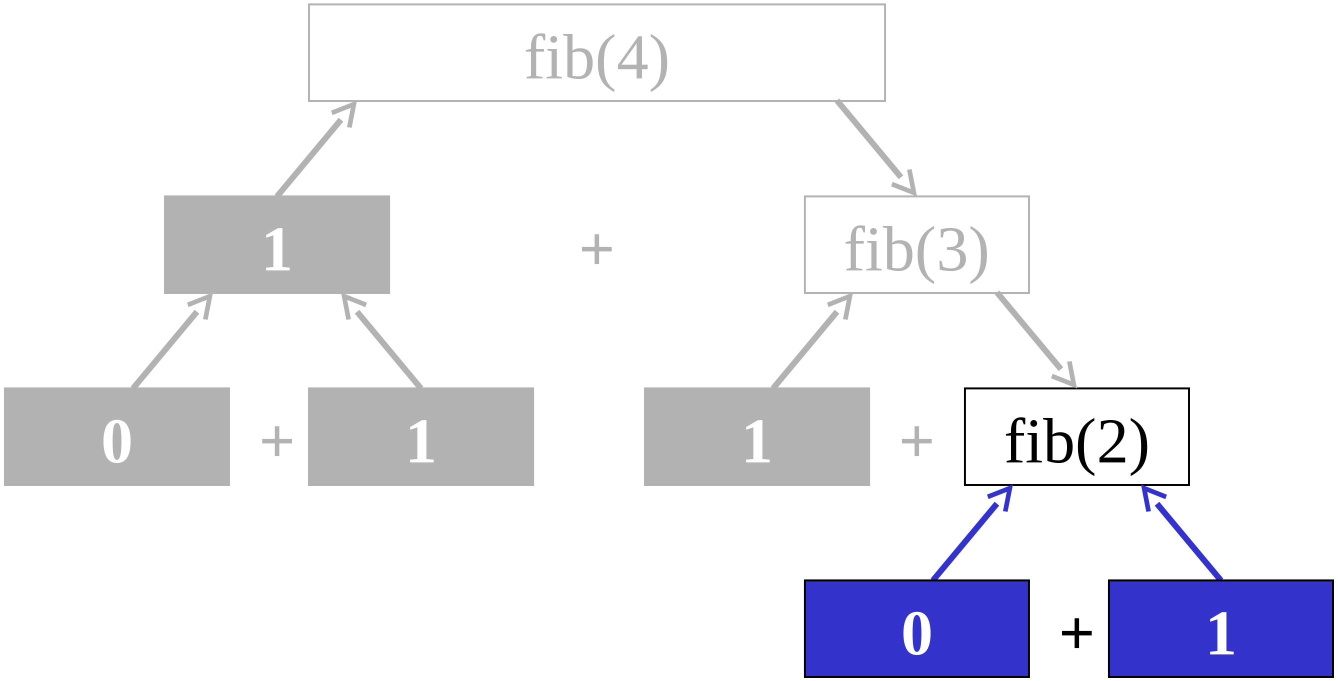
Example: Computation of `fib(4)`

```
long fib ( long 1 )
{
   if ( 1 <= 1 )
      return 1 ;
   else
      return fib( n - 2 ) + fib( n - 1 );
}
```



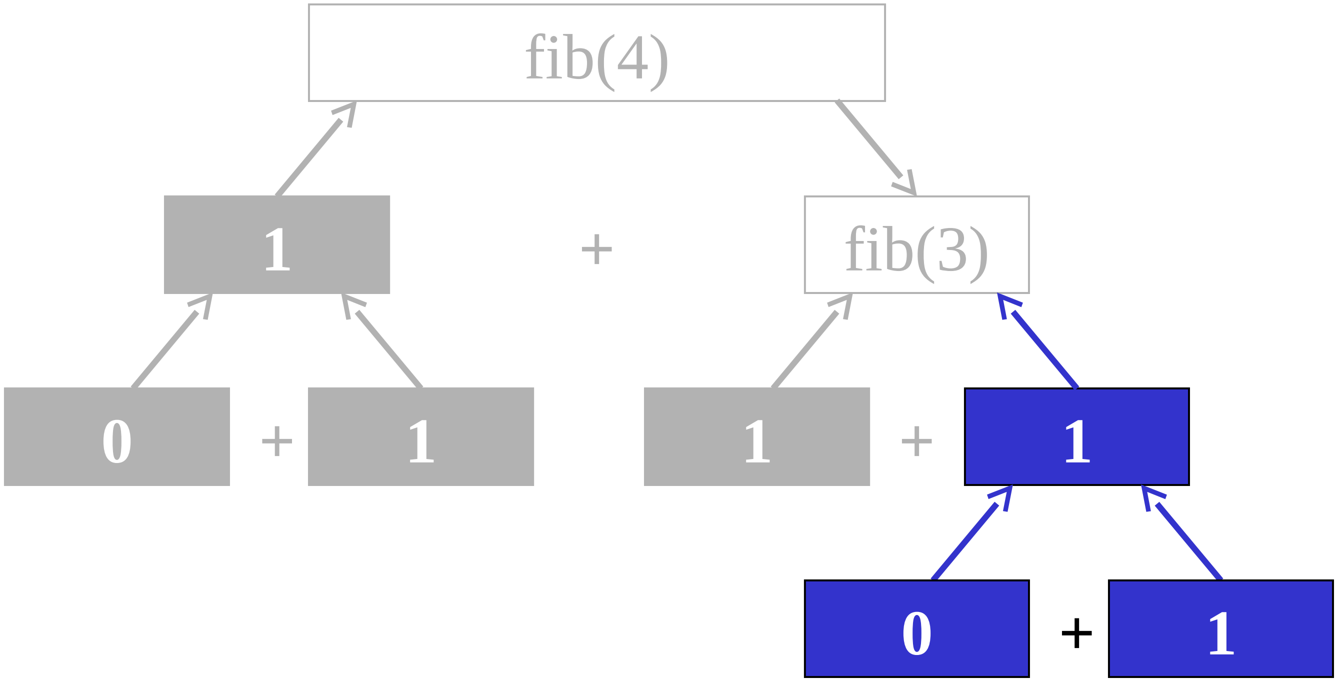
Example: Computation of `fib(4)`

```
long fib ( long 2 )
{
   if ( 2 <= 1 )
      return n ;
   else
      return         0          +          1          ;
}
```



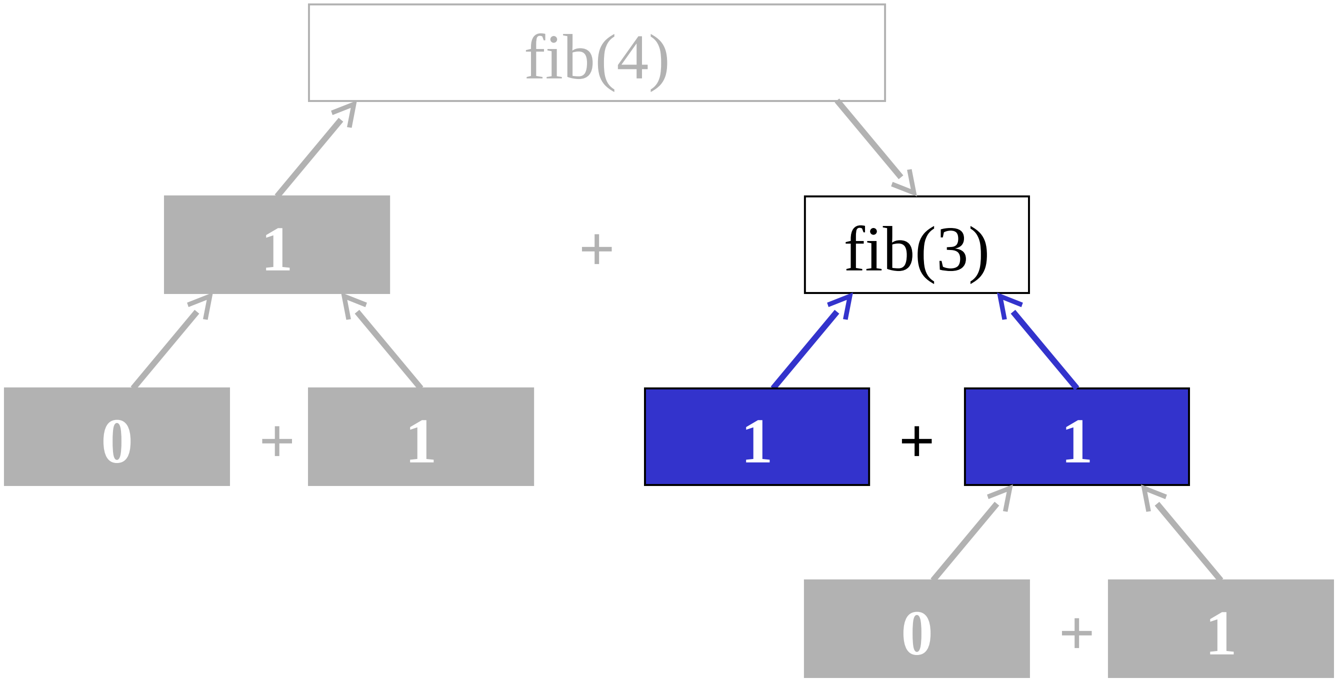
Example: Computation of **`fib(4)`**

```
long fib ( long 2 )
{
   if ( 2 <= 1 )
      return n ;
   else
      return    0    +    1    ;
}
```



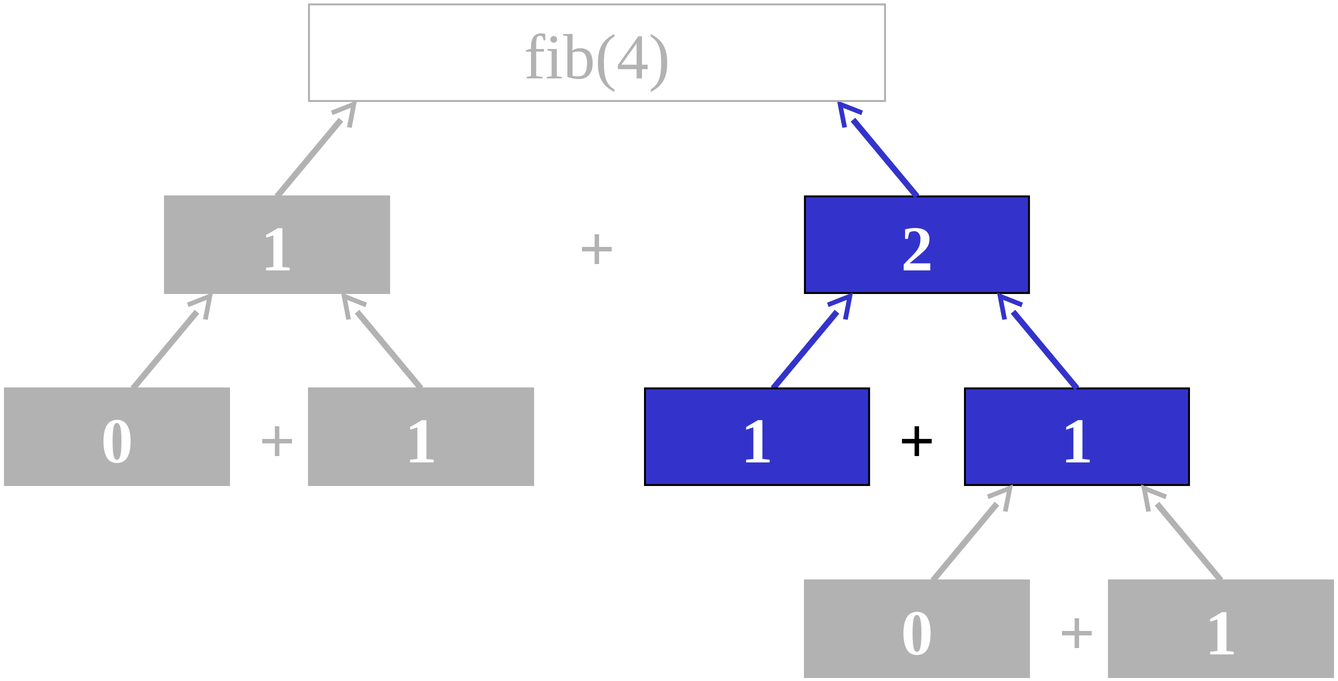
Example: Computation of `fib(4)`

```
long fib ( long 3 )
{
   if ( 3 <= 1 )
      return n ;
   else
      return       1       +       1       ;
}
```



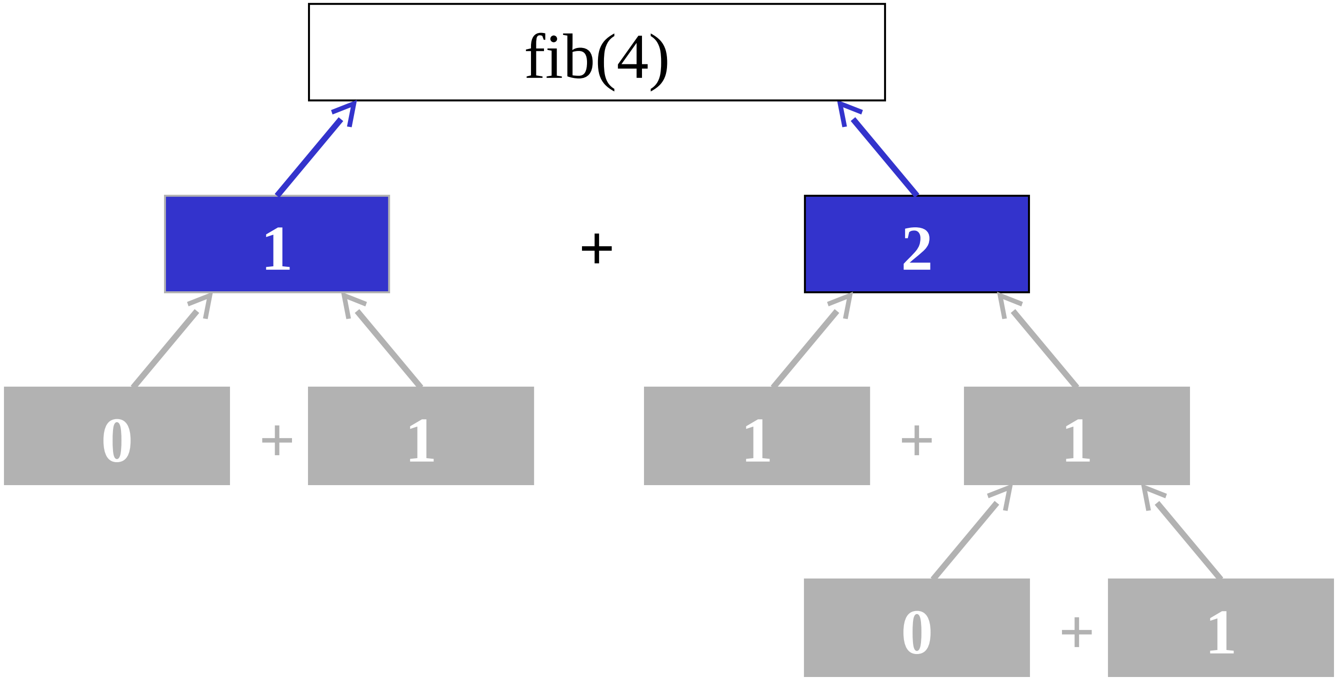
Example: Computation of **`fib(4)`**

```
long fib ( long 3 )
{
    if ( 3 <= 1 )
        return n ;
    else
        return       1       +       1       ;
}
```



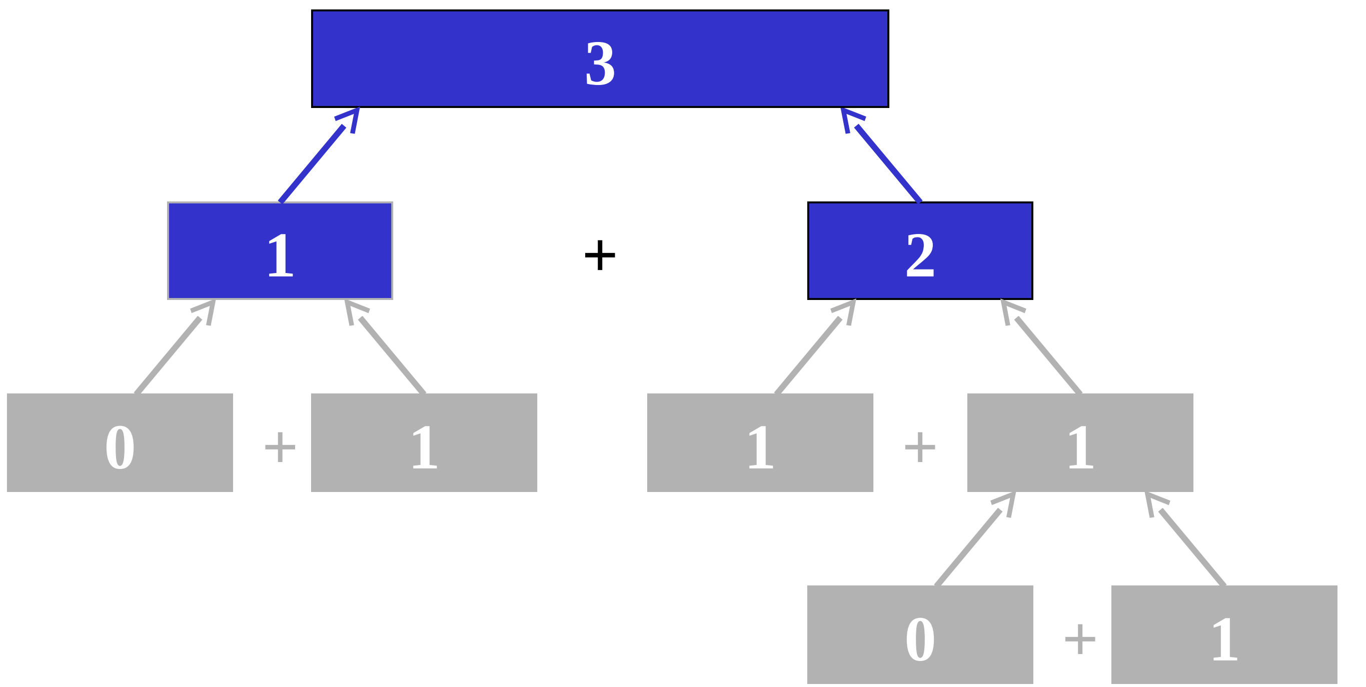
Example: Computation of `fib(4)`

```
long fib ( long 4 )
{
    if ( 4 <= 1 )
        return n ;
    else
        return      1      +      2      ;
}
```



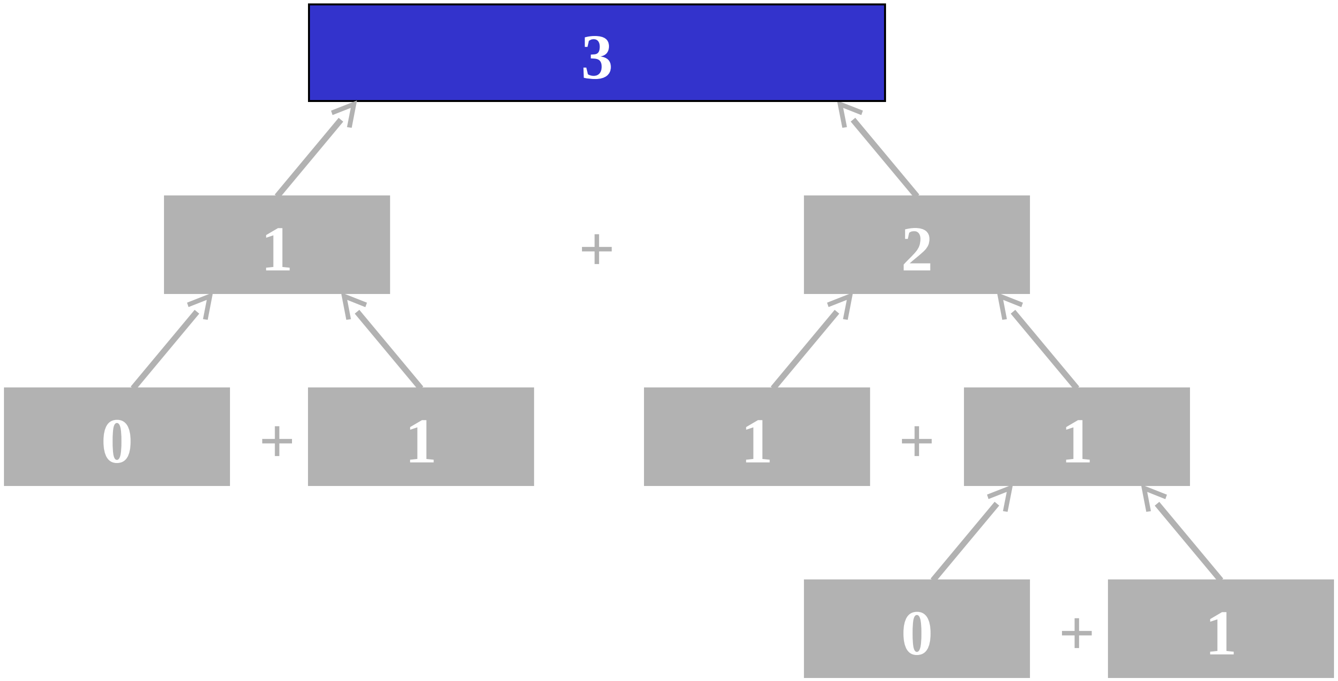
Example: Computation of `fib(4)`

```
long fib ( long 4 )
{
    if ( 4 <= 1 )
        return n ;
    else
        return 1 + 2 ;
}
```



Example: Computation of `fib(4)`

Thus, **`fib(4)`** returns the value 3.



Example: fibonacc.c

Sample **`main()`** for testing the **`fib()`** function:

```
int main(void)
{
   long number;

   printf("Enter number: ");
   scanf("%ld", &number);

   printf("Fibonacci(%ld) = %ld\n",
                    number, fib(number));

   return 0;
}
```

Reading

- King Chapter 9

  Section 9.6
- D&D Chapter 5

  Sections 5.13 to 5.14
- Kernighan & Ritchie Chapter 4

  Section 4.10