Structures and Recursivity

- Pierre-Alain FOUQUE

Array

The arrays allow

- definition under a unique name a set of objects of the same type
- access each objects by its position in the array

Array

The arrays allow

- definition under a unique name a set of objects of the same type
- access each objects by its position in the array

How can we put together objects of different types ?

Structures

Structures allow to put together objects of different types

⇒ new type : **`struct bloc`**

Structures

Structures allow to put together objects of different types

⇒ new type : `struct bloc`

```
struct bloc {
   int number;
   float value;
};
```

Structures

Structures allow to put together objects of different types

⇒ new type : **`struct bloc`**

```
struct bloc {
   int number;
   float value;
};
```

Each object of this type has an integer field, called **`number`**

Structures

Structures allow to put together objects of different types

⇒ new type : **`struct bloc`**

```
struct bloc {
   int number;
   float value;
};
```

Each object of this type has an integer field, called **`number`**

a float field, called **`value`**

How to use it ?

`struct bloc bl;`

define a variable **`bl`**

of type **`struct bloc`**

The field **`number`** can be accessed by **`bl.number`**

and the field **`value`** by **`bl.value`**

```
struct bloc {
   int number;
   float value;
};
```

How to use it ?

`struct bloc bl;`

define a variable **`bl`**

of type **`struct bloc`**

The field **`number`** can be accessed by **`bl.number`**

and the field **`value`** by **`bl.value`**

```
struct bloc {
  int number;
  float value;
};
```

```
struct bloc bl;

bl.number = 10;
bl.value = 3.2;
```

New type with `typedef`

The new type is called
`struct bloc`
⇒ not so easy to use it

```
typedef struct bloc sbloc;
```

define another name : `sbloc`

New type with `typedef`

The new type is called
`struct bloc`
⇒ not so easy to use it

`typedef struct bloc sbloc;`
define another name : **`sbloc`**

```
struct bloc {
   int number;
   float value;
};

typedef struct bloc sbloc;
```

New type with `typedef`

The new type is called

`struct bloc`

⇒ not so easy to use it

`typedef struct bloc sbloc;`

define another name : **`sbloc`**

```
struct bloc {
   int number;
   float value;
};

typedef struct bloc sbloc;
```

ou

```
typedef struct {
   int number;
   float value;
} sbloc;
```

New type with `typedef`

The new type is called

`struct bloc`

⇒ not so easy to use it

```
sbloc bl;

bl.number = 10;
bl.value = 3.2;
```

`typedef struct bloc sbloc;`

define another name : **`sbloc`**

```
struct bloc {
   int number;
   float value;
};

typedef struct bloc sbloc;
```

ou

```
typedef struct {
   int number;
   float value;
} sbloc;
```

Example 1

We can define a point in the plan by its coordinates:

Example 1

We can define a point in the plan by its coordinates:

```
typedef struct {
   float abscisse;
   float ordonnée;
} point2D;
```

Example 1

We can define a point in the plan by its coordinates:

```
typedef struct {
   float abscisse;
   float ordonnée;
} point2D;
```

We can then declare and initialize such an object:

Example I

We can define a point in the plan by its coordinates:

```
typedef struct {
   float abscisse;
   float ordonnée;
} point2D;
```

We can then declare and initialize such an object:

```
   point2D P;

   P.abscisse = 2.5;
   P.ordonnee = 4.3;
```

Example 1 - suite

A structure is then a type like others (int, float, ...) :

Example 1 - suite

A structure is then a type like others (int, float, ...) :

```
point2D translation (point2D a, point2D b)
{
  point2D c;
  c.abscisse = a.abscisse + b.abscisse;
  c.ordonnee = a.ordonnee + b.ordonnee;
  return c;
}
```

Declaration and initialization

`sbloc bl;`

declare a variable **`bl`** of type **`sbloc`**

Its initialization can be made field by field, or globally:

```
sbloc bl2 = { 10, 3.2 };
```

```
typedef struct {
  int number;
  float value;
} sbloc;
```

Declaration and initialization

`sbloc bl;`

declare a variable **`bl`** of type **`sbloc`**

Its initialization can be made field by field, or globally:

`sbloc bl2 = { 10, 3.2 };`

```
typedef struct {
  int number;
  float value;
} sbloc;
```

```
sbloc bl2 = { 10, 3.2 };

sbloc bl;
bl.number = 10;
bl.value = 3.2;
```

Example II - complexes

We can define complexe type by :

Example II - complexes

We can define complexe type by :

```
typedef struct {
   float real;
   float im;
} complexe;
```

Example II - complexes

We can define complexe type by :

```
typedef struct {
   float real;
   float im;
} complexe;
```

We can then declare and initialize such an object:

Example II - complexes

We can define complexe type by :

```
typedef struct {
   float real;
   float im;
} complexe;
```

We can then declare and initialize such an object:

```
complexe c;

c.real = 2.5;
c.im = 4.3;
```

Example II - suite

We can then define the addition :

Example II - suite

We can then define the addition :

```
complexe addition (complexe c1, complexe c2)
{
   complexe c;
   c.real = c1.real + c2.real;
   c.im   = c1.im   + c2.im;
   return c;
}
```

Example II - suite

We can then define the addition :

```
complexe addition (complexe c1, complexe c2)
{
   complexe c;
   c.real = c1.real + c2.real;
   c.im   = c1.im   + c2.im;
   return c;
}
```

```
   complexe c,d;

   c.real = 2.5;
   c.im = 4.3;
   d = addition(c,c);
```

Example II - execution

Example II - execution

```
complexe addition (complexe c1, complexe c2)
{
   complexe c;
   c.real = c1.real + c2.real;
   c.im   = c1.im   + c2.im;
   return c;
}
```

Example II - execution

```
complexe addition (complexe c1, complexe c2)
{
   complexe c;
   c.real = c1.real + c2.real;
   c.im   = c1.im   + c2.im;
   return c;
}
```

```
complexe c,d;

c.real = 2.5;
c.im = 4.3;
d = addition(c,c);
```

Example II - execution

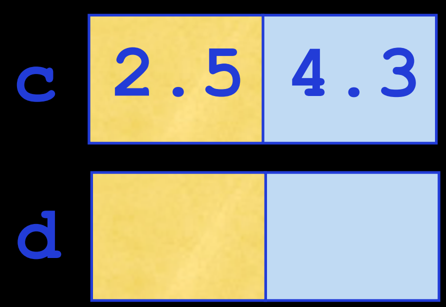
```
complexe addition (complexe c1, complexe c2)
{
   complexe c;
   c.real = c1.real + c2.real;
   c.im   = c1.im   + c2.im;
   return c;
}
```

```
complexe c,d;

c.real = 2.5;
c.im = 4.3;
d = addition(c,c);
```

Example II - execution

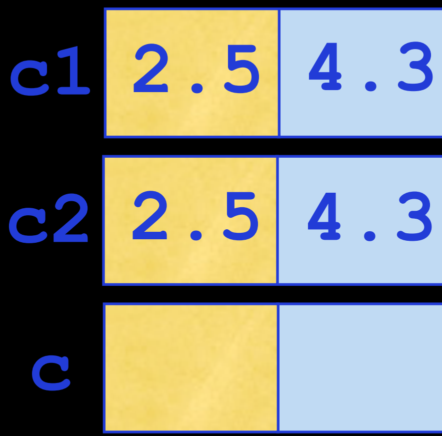
```
complexe addition (complexe c1, complexe c2)
{
   complexe c;
   c.real = c1.real + c2.real;
   c.im   = c1.im   + c2.im;
   return c;
}
```

```
complexe c,d;

c.real = 2.5;
c.im = 4.3;
d = addition(c,c);
```

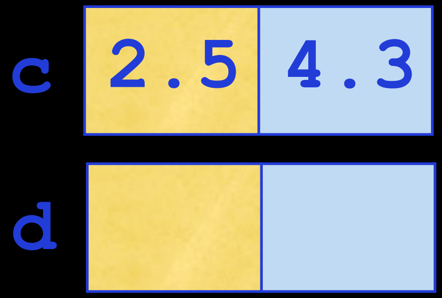


Example II - execution

```
complexe addition (complexe c1, complexe c2)
{
   complexe c;
   c.real = c1.real + c2.real;
   c.im   = c1.im   + c2.im;
   return c;
}
```

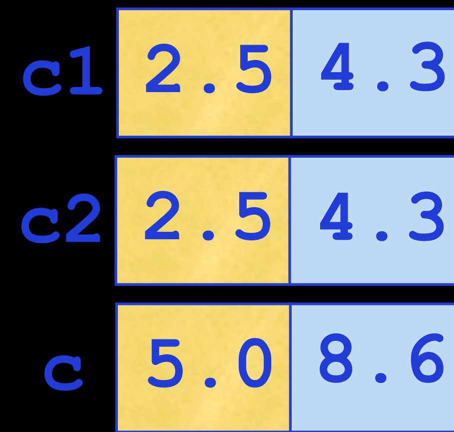



```
complexe c,d;

c.real = 2.5;
c.im = 4.3;
d = addition(c,c);
```

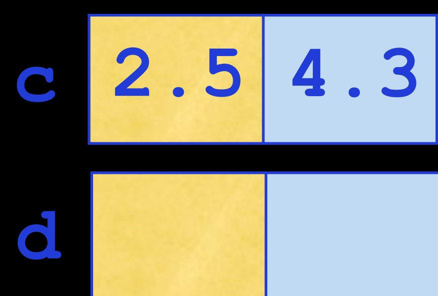


Example II - execution

```
complexe addition (complexe c1, complexe c2)
{
   complexe c;
   c.real = c1.real + c2.real;
   c.im   = c1.im   + c2.im;
   return c;
}
```




```
complexe c,d;

c.real = 2.5;
c.im = 4.3;
d = addition(c,c);
```



Example II - execution

```
complexe addition (complexe c1, complexe c2)
{
   complexe c;
   c.real = c1.real + c2.real;
   c.im   = c1.im   + c2.im;
   return c;
}
```

```
complexe c,d;

c.real = 2.5;
c.im = 4.3;
d = addition(c,c);
```

Display

To print a structure,
one can print the fields one by one
(likes an array)

```
typedef struct {
   int number;
   float value;
} sbloc;
```

Display

To print a structure,
one can print the fields one by one
(likes an array)

```
typedef struct {
   int number;
   float value;
} sbloc;
```

```
void affiche (sbloc bl) {
   printf(``%d - %f \n '',
            bl.number,
            bl.value);
}
```

Example II - display

We remind the complex type :

```
typedef struct {
   float real;
   float im;
} complexe;
```

Example II - display

We remind the complex type :

```
typedef struct {
   float real;
   float im;
} complexe;
```

We can print such an object :

Example II - display

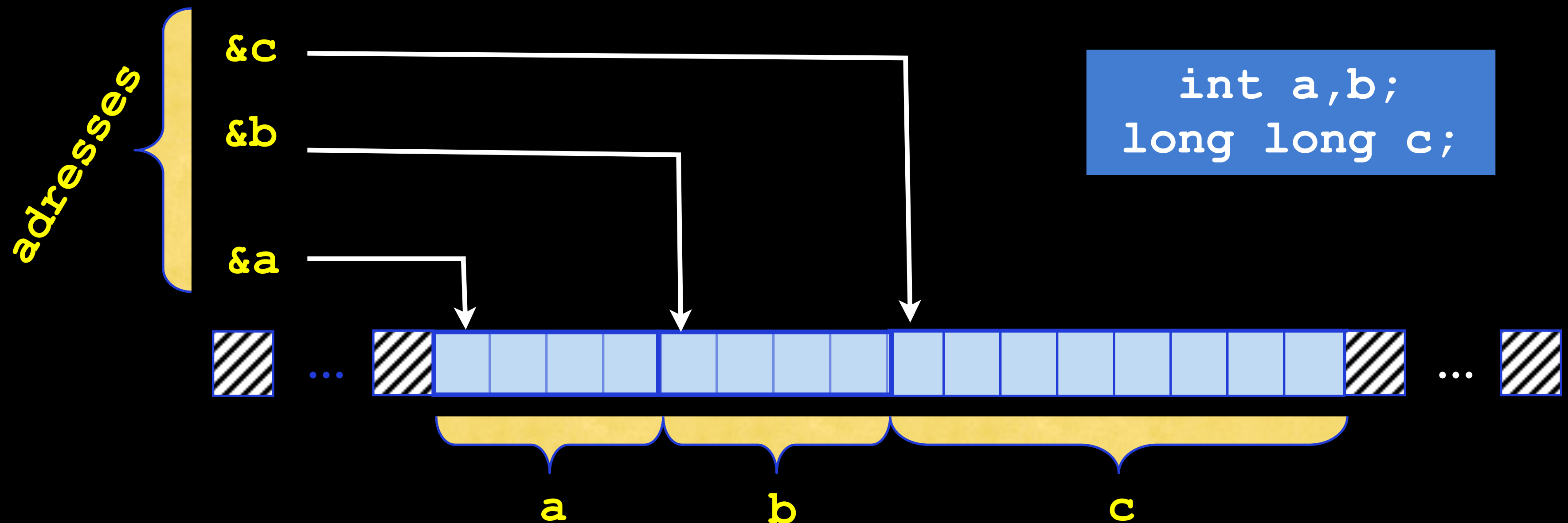
We remind the complex type :

```
typedef struct {
   float real;
   float im;
} complexe;
```

We can print such an object :

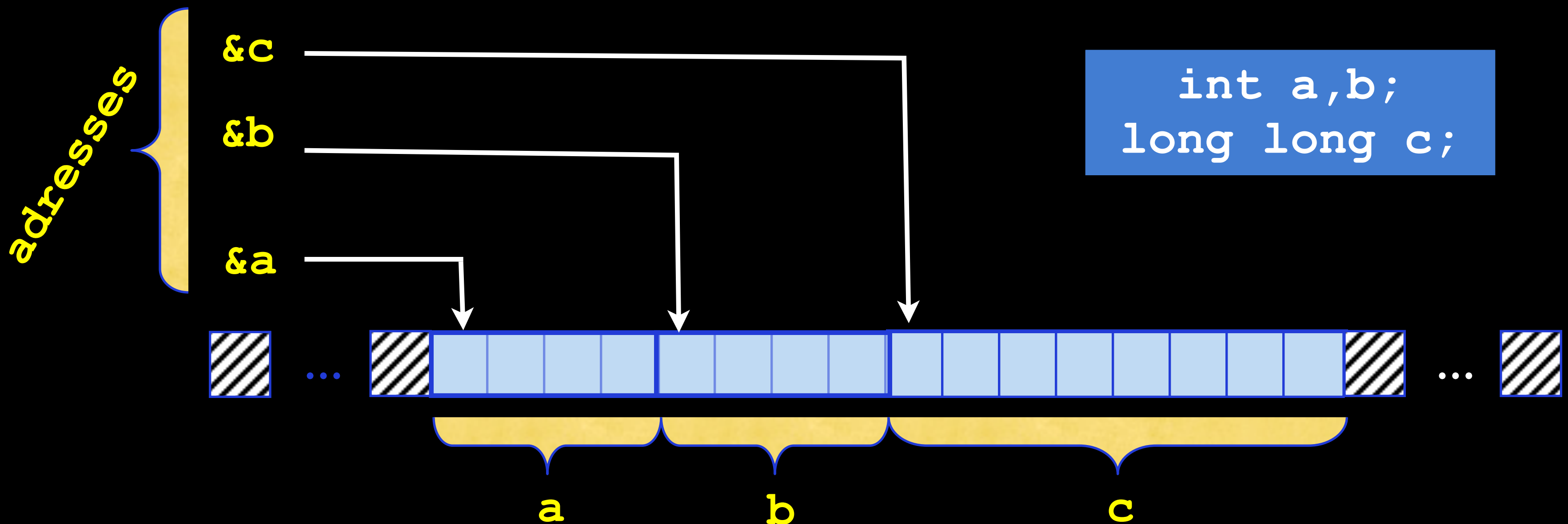
```
void display (complexe c) {
  printf(``%f + i * %f \n '',
            c.real, c.im);
}
```

Pointers



Pointers

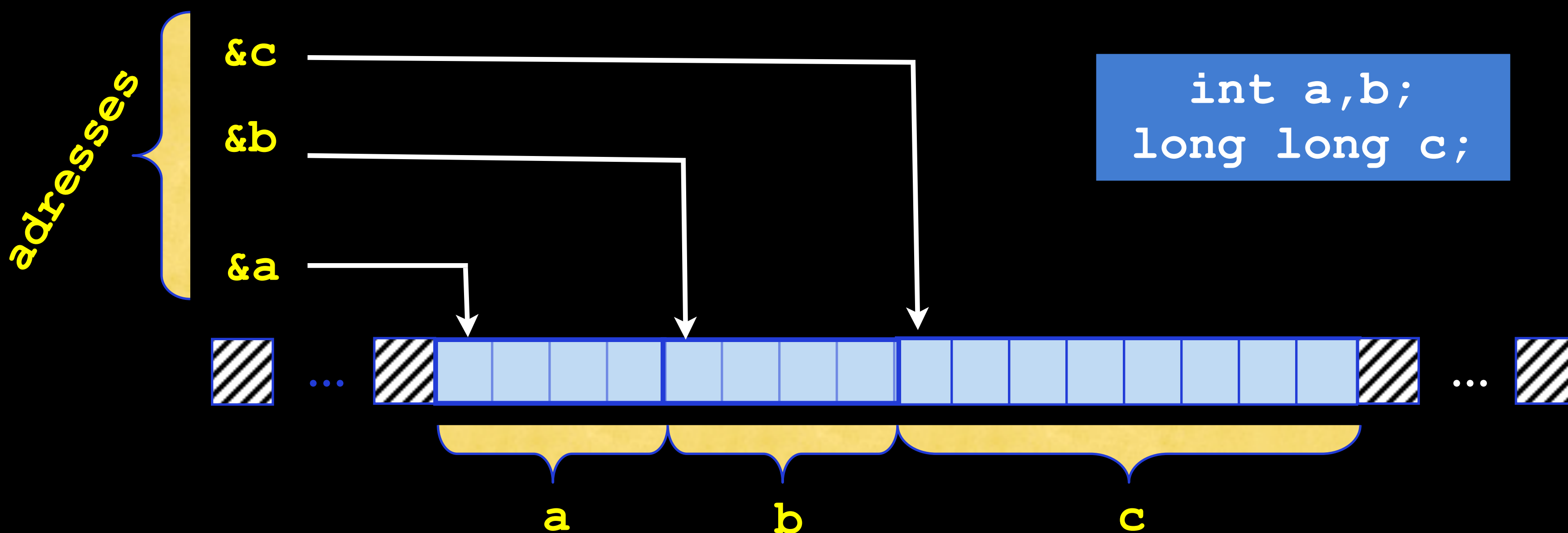
A variable is stored in a memory area reserved during the declaraion



Pointers

A variable is stored in a memory area reserved during the declaraion

`&a` represent the memory address



Pointers and structures

```
int *Pt;
int T;
Pt = &T;
```

Pointers and structures

```
int *Pt;
int T;
Pt = &T;
```

define a pointer `Pt` and points to `T`.

Pointers and structures

```
int *Pt;
int T;
Pt = &T;
```

define a pointer **`Pt`**
and points to **`T`**.
Then **`*Pt`** represent **`T`**

Pointers and structures

```
int *Pt;
int T;
Pt = &T;
```

*** et & converse**

define a pointer **`Pt`**
and points to **`T`**.
Then **`*Pt`** represent **`T`**

Pointers and structures

```
int *Pt;
int T;
Pt = &T;
```

```
typedef struct {
  int number;
  float value;
} sbloc;
```

`*` et `&` converse

define a pointer `Pt` and points to `T`.

Then `*Pt` represent `T`

Pointers and structures

```
int *Pt;
int T;
Pt = &T;
```

```
typedef struct {
   int number;
   float value;
} sbloc;
```

```
sbloc bl;
sbloc *Psb = &bl;

(*Psb).number = 10;
(*Psb).value = 3.2;
```

*** et & converse**

define a pointer `Pt` and points to `T`.

Then `*Pt` represent `T`

Pointers and structures

```
int *Pt;

int T;

Pt = &T;
```

*** et & converse**

define a pointer `Pt` and points to `T`.

Then `*Pt` represent `T`

```
typedef struct {
   int number;
   float value;
} sbloc;
```

```
sbloc bl;
sbloc *Psb = &bl;

(*Psb).number = 10;
(*Psb).value = 3.2;
```

or

```
sbloc bl;
sbloc *Psb = &bl;

Psb->number = 10;
Psb->valur = 3.2;
```

. or -> ?

```
struct bloc {
   int number;
   float value;
};
```

`struct bloc b1;`

define a variable **`b1`** of type **`struct bloc`**

`struct bloc *Pb1;`

define a pointer onto an object of type **`struct bloc`**

. or -> ?

struct bloc b1;
define a variable **b1**
of type **struct bloc**

```
struct bloc {
   int number;
   float value;
};
```

struct bloc *Pb1;
define a pointer onto an object of
type **struct bloc**

The field **number** of **b1** : **b1.number**

. or -> ?

```
struct bloc {
   int number;
   float value;
};
```

`struct bloc b1;`

define a variable **`b1`** of type **`struct bloc`**

`struct bloc *Pb1;`

define a pointer onto an object of type **`struct bloc`**

The field **`number`** of **`b1`** : **`b1.number`**

The field **`value`** of **`*Pb1`** : **`Pb1->value`**

. or -> ?

```c
struct bloc {
   int number;
   float value;
};
```

`struct bloc b1;`
define a variable **`b1`**
of type **`struct bloc`**

`struct bloc *Pb1;`
define a pointer onto an object of
type **`struct bloc`**

The field **`number`** of **`b1`** : **`b1.number`**

The field **`value`** of **`*Pb1`** : **`Pb1->value`**

If **`Pb1`** points to such a structure !!

Recursivity

The function power can be expressed recursively :

$$a^e = a * a^{e-1}$$
$$a^0 = 1$$

Recursivity

The function power can be expressed recursively :

$$a^{e} = a * a^{e-1}$$
$$a^{0} = 1$$

⇒ recursive programing :

Recursivity

The function power can be expressed recursively :

```
a^e = a * a^(e-1)
a^0 = 1
```

$$a^e = a * a^{e-1}$$
$$a^0 = 1$$

⇒ recursive programing : the function power calls itself

Recursivity

The function power can be expressed recursively :

$$a^e = a * a^{e-1}$$
$$a^0 = 1$$

Stopping criteria

⇒ recursive programing :
the function power calls itself

Example 1

The function **`power`** can be implemented as follows:

```
double power(double a, int e)
{
   if (e == 0) return 1;
   return power(a,e-1)*a;
}
```

If **`e`**=0 the result is 1 ⇒ **`return 1;`**

otherwise, the result is a^{e-1} **`* a`**

⇒ **`return power(a,e-1)*a;`**

Example I

The function **`power`** can be implemented as follows:

```
double power(double a, int e)
{
   if (e == 0) return 1;
   return power(a,e-1)*a;
}
```

Stopping criteria (points to `if (e == 0) return 1;`)

If **`e`**=0 the result is 1 ⇒ **`return 1;`**

otherwise, the result is a^{e-1} **`* a`**

⇒ **`return power(a,e-1)*a;`**

Example 1 - analyze

At each call to the function **`power`**

- the **`double`** `‘2.0’` is stored into **`a`**
- and the **`int`** `‘2’` is stored into **`e`**

```
double y;

y = power(2.0, 2);
```

```
double power(double a, int e)
{
   if (e == 0) return 1;
   return power(a,e-1)*a;
}
```

Example 1 - execution

```
   double y;

   y = power(2.0, 2);
```

```
double power(double a, int e)
{
   if (e == 0) return 1;
   return power(a,e-1)*a;
}
```

Example 1 - execution

`power(2.0,2)`

```
   double y;

   y = power(2.0, 2);
```

```
double power(double a, int e)
{
   if (e == 0) return 1;
   return power(a,e-1)*a;
}
```

Example 1 - execution

```
power(2.0,2)
   a ← 2.0
```

```
   double y;

   y = power(2.0, 2);
```

```
double power(double a, int e)
{
   if (e == 0) return 1;
   return power(a,e-1)*a;
}
```

Example 1 - execution

```
power(2.0,2)
   a ← 2.0
   e ← 2
```

```
   double y;

   y = power(2.0, 2);
```

```
double power(double a, int e)
{
   if (e == 0) return 1;
   return power(a,e-1)*a;
}
```

Example 1 - execution

```
power(2.0,2)
   a ← 2.0
   e ← 2
   ← power(2.0,1)*2.0
```

```
   double y;

   y = power(2.0, 2);
```

```
double power(double a, int e)
{
   if (e == 0) return 1;
   return power(a,e-1)*a;
}
```

Example 1 - execution

```
power(2.0,2)
    a ← 2.0
    e ← 2
    ← power(2.0,1)*2.0
      power(2.0,1)
```

```
   double y;

   y = power(2.0, 2);
```

```
double power(double a, int e)
{
   if (e == 0) return 1;
   return power(a,e-1)*a;
}
```

Example 1 - execution

```
power(2.0,2)
    a ← 2.0
    e ← 2
    ← power(2.0,1)*2.0
      power(2.0,1)
         a ← 2.0
```

```
   double y;

   y = power(2.0, 2);
```

```
double power(double a, int e)
{
   if (e == 0) return 1;
   return power(a,e-1)*a;
}
```

Example 1 - execution

```
power(2.0,2)
    a ← 2.0
    e ← 2
    ← power(2.0,1)*2.0
      power(2.0,1)
         a ← 2.0
         e ← 1
```

```
   double y;

   y = power(2.0, 2);
```

```
double power(double a, int e)
{
   if (e == 0) return 1;
   return power(a,e-1)*a;
}
```

Example 1 - execution

```
power(2.0,2)
    a ← 2.0
    e ← 2
    ← power(2.0,1)*2.0
        power(2.0,1)
            a ← 2.0
            e ← 1
            ← power(2.0,0)*2.0
```

```
double y;

y = power(2.0, 2);
```

```
double power(double a, int e)
{
   if (e == 0) return 1;
   return power(a,e-1)*a;
}
```

Example 1 - execution

```
power(2.0,2)
    a ← 2.0
    e ← 2
    ← power(2.0,1)*2.0
       power(2.0,1)
          a ← 2.0
          e ← 1
          ← power(2.0,0)*2.0
             power(2.0,0)
```

```
   double y;

   y = power(2.0, 2);
```

```
double power(double a, int e)
{
   if (e == 0) return 1;
   return power(a,e-1)*a;
}
```

Example 1 - execution

```
power(2.0,2)
    a ← 2.0
    e ← 2
    ← power(2.0,1)*2.0
       power(2.0,1)
          a ← 2.0
          e ← 1
          ← power(2.0,0)*2.0
             power(2.0,0)
                a ← 2.0
```

```
   double y;

   y = power(2.0, 2);
```

```
double power(double a, int e)
{
   if (e == 0) return 1;
   return power(a,e-1)*a;
}
```

Example 1 - execution

```
power(2.0,2)
    a ← 2.0
    e ← 2
    ← power(2.0,1)*2.0
      power(2.0,1)
          a ← 2.0
          e ← 1
          ← power(2.0,0)*2.0
            power(2.0,0)
                a ← 2.0
                e ← 0
```

```
double y;

y = power(2.0, 2);
```

```
double power(double a, int e)
{
   if (e == 0) return 1;
   return power(a,e-1)*a;
}
```

Example 1 - execution

```
power(2.0,2)
    a ← 2.0
    e ← 2
    ← power(2.0,1)*2.0
      power(2.0,1)
          a ← 2.0
          e ← 1
          ← power(2.0,0)*2.0
            power(2.0,0)
                a ← 2.0
                e ← 0
                ← 1
```

```
double y;

y = power(2.0, 2);
```

```
double power(double a, int e)
{
   if (e == 0) return 1;
   return power(a,e-1)*a;
}
```

Example 1 - execution

```
power(2.0,2)
    a ← 2.0
    e ← 2
    ← power(2.0,1)*2.0
      power(2.0,1)
        a ← 2.0
        e ← 1
        ← power(2.0,0)*2.0
          power(2.0,0)
            a ← 2.0
            e ← 0
            ← 1
        ← 1 * 2.0 = 2.0
```

```
double y;

y = power(2.0, 2);
```

```
double power(double a, int e)
{
   if (e == 0) return 1;
   return power(a,e-1)*a;
}
```

Example 1 - execution

```
power(2.0,2)
    a ← 2.0
    e ← 2
    ← power(2.0,1)*2.0
      power(2.0,1)
        a ← 2.0
        e ← 1
        ← power(2.0,0)*2.0
          power(2.0,0)
            a ← 2.0
            e ← 0
            ← 1
        ← 1 * 2.0 = 2.0
    ← 2.0 * 2.0 = 4.0
```

```
   double y;

   y = power(2.0, 2);
```

```
double power(double a, int e)
{
   if (e == 0) return 1;
   return power(a,e-1)*a;
}
```

Local Variable

```
double power(double a, int e)
```

Each execution of the function **`power`** has is own local variables **`a`** and **`e`**,
destroyed at the end of the call

Example II

An equivalent implementation is :

```
double power(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = power(a,e-1);
   return z*a;
}
```

Remind : the **`return x`** leaves the function and return the value of the variable **`x`**

Example II

An equivalent implementation is :

```
double power(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = power(a,e-1);
   return z*a;
}
```

Stopping criteria

Remind : the **`return x`** leaves the function and return the value of the variable **`x`**

Example II – execution

`power(2.0,2)`

```
double power(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = power(a,e-1);
   return z*a;
}
```

Example II – execution

```
power(2.0,2)
```

```
puissance(2.0,2)
```

```
double power(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = power(a,e-1);
   return z*a;
}
```

Example II – execution

`power(2.0,2)`

`puissance(2.0,2)`

a	2.0
e	2
z	

```
double power(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = power(a,e-1);
   return z*a;
}
```

Example II – execution

`power(2.0,2)`

`puissance(2.0,2)`

a	2.0
e	2
z	

`puissance(2.0,1)`

```
double power(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = power(a,e-1);
   return z*a;
}
```

Example II – execution

`power(2.0,2)`

`puissance(2.0,2)`

a	2.0
e	2
z	

`puissance(2.0,1)`

a	2.0
e	1
z	

```
double power(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = power(a,e-1);
   return z*a;
}
```

Example II – execution

`power(2.0,2)`

`puissance(2.0,2)`

a	2.0
e	2
z	

`puissance(2.0,1)`

a	2.0
e	1
z	

`puissance(2.0,0)`

```
double power(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = power(a,e-1);
   return z*a;
}
```

Example II – execution

power(2.0,2)

puissance(2.0,2)

a	2.0
e	2
z	

puissance(2.0,1)

a	2.0
e	1
z	

puissance(2.0,0)

a	2.0
e	0
z	

```
double power(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = power(a,e-1);
   return z*a;
}
```

Example II – execution

power(2.0,2)

puissance(2.0,2)

a	2.0
e	2
z	

puissance(2.0,1)

a	2.0
e	1
z	← puissance(2.0,0)

a	2.0
e	0
z	

```
double power(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = power(a,e-1);
   return z*a;
}
```

Example II – execution

power(2.0,2)

puissance(2.0,2)

a	2.0
e	2
z	

puissance(2.0,1)

a	2.0
e	1
z	

puissance(2.0,0)

a	2.0
e	0
z	

```
double power(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = power(a,e-1);
   return z*a;
}
```

Example II – execution

power(2.0,2)

puissance(2.0,2)

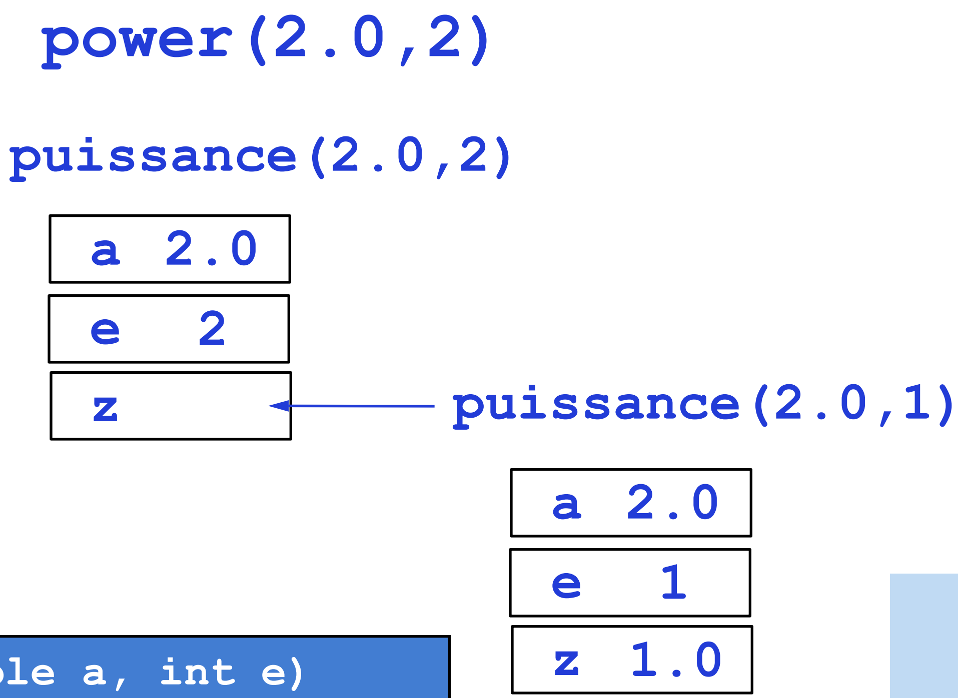
a	2.0
e	2
z	

puissance(2.0,1)

a	2.0
e	1
z	1.0

```
double power(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = power(a,e-1);
   return z*a;
}
```

Example II – execution




```
double power(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = power(a,e-1);
   return z*a;
}
```

Example II – execution

power(2.0,2)

puissance(2.0,2)

a	2.0
e	2
z	

puissance(2.0,1)

a	2.0
e	1
z	1.0

```
double power(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = power(a,e-1);
   return z*a;
}
```

Example II – execution

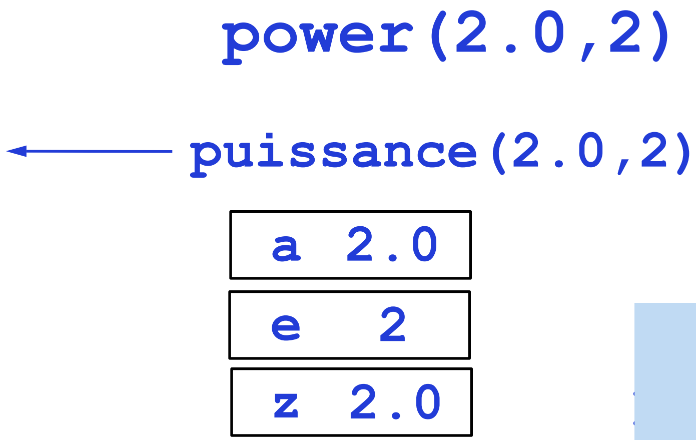
`power(2.0,2)`

`puissance(2.0,2)`

a	2.0
e	2
z	2.0

```
double power(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = power(a,e-1);
   return z*a;
}
```

Example II – execution




```
double power(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = power(a,e-1);
   return z*a;
}
```

Example II – execution

```
power(2.0,2)
```

```
puissance(2.0,2)
```

a	2.0
e	2
z	2.0

```
double power(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = power(a,e-1);
   return z*a;
}
```

Example II – execution

power(2.0,2)

4.0

```
double power(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = power(a,e-1);
   return z*a;
}
```

Example II – stack

power(2.0,2)

a	2.0
e	2
z	2.0

power(2.0,1)

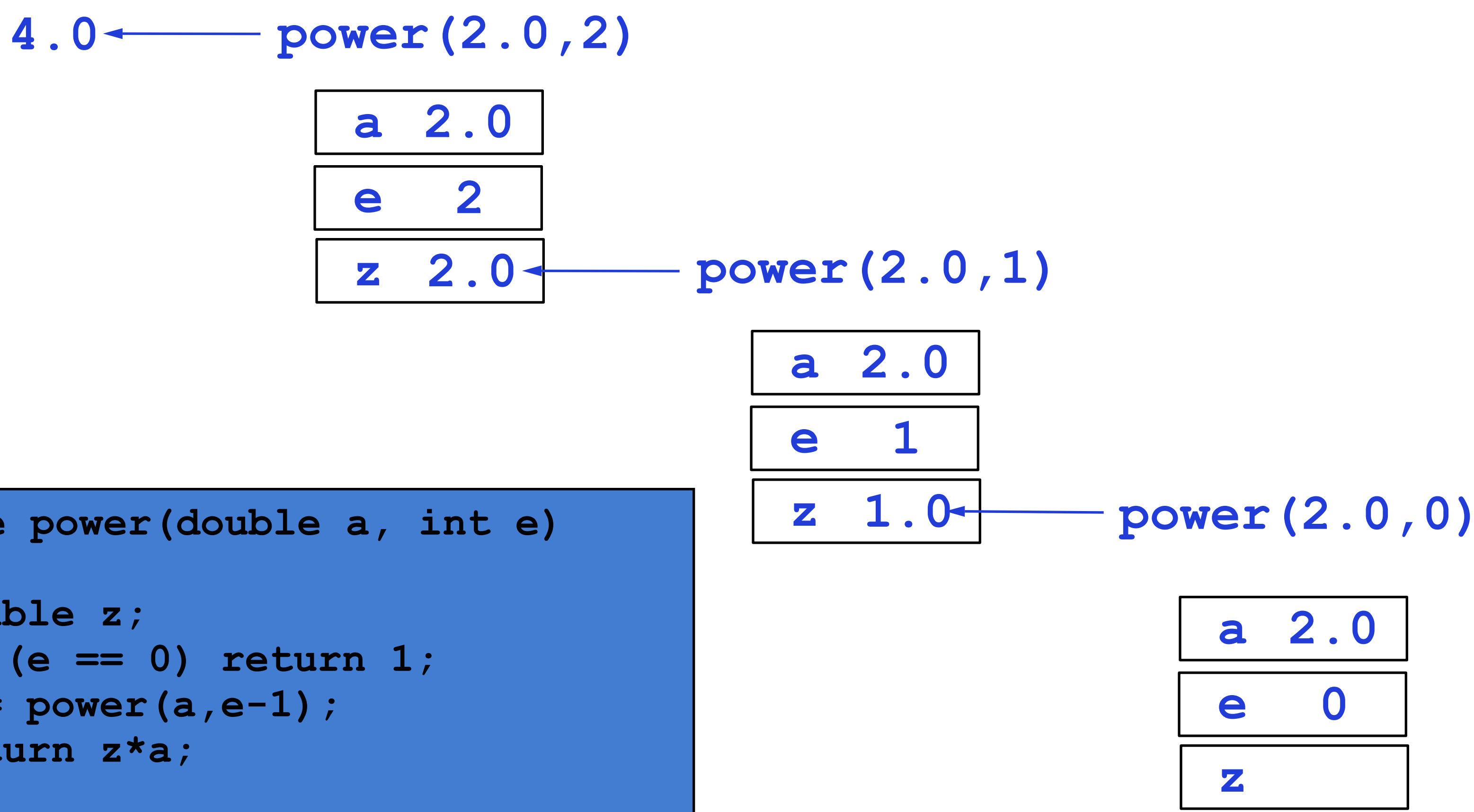
a	2.0
e	1
z	1.0

power(2.0,0)

a	2.0
e	0
z	

```c
double power(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = power(a,e-1);
   return z*a;
}
```

Example II – stack




```
double power(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = power(a,e-1);
   return z*a;
}
```

Non terminal Recursivity

```
double power(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = power(a,e-1);
   return z*a;
}
```

Non terminal Recursivity

```
double power(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = power(a,e-1);
   return z*a;
}
```

The recursive call is not the last instruction of the function

Non terminal Recursivity

```
double power(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = power(a,e-1);
   return z*a;
}
```

The recursive call is not the last instruction of the function
⇒ the operation is performed when « poping » the stack

Example III

The addition of positive integer can be implemented with only increments and decrements

```
a + b = (a+1) + (b-1)
a + 0 = a
```

Example III

The addition of positive integer can be implemented with only increments and decrements

```
a + b = (a+1) + (b-1)
a + 0 = a
```

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Example III

The addition of positive integer can be implemented with only increments and decrements

```
a + b = (a+1) + (b-1)
a + 0 = a
```

← stopping criteria

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Example III – exécution

```
addition(3,2)
```

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Example III – exécution

```
addition(3,2)
```

```
addition(3,2)
```

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Example III – exécution

addition(3,2)

addition(3,2)

a	3
b	2

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Example III – exécution

```
addition(3,2)
```

```
addition(3,2)
```

a	3
b	2

```
addition(4,1)
```

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Example III – exécution

addition(3,2)

addition(3,2)

a	3
b	2

addition(4,1)

a	4
b	1

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Example III – exécution

addition(3,2)

addition(3,2)

a	3
b	2

addition(4,1)

a	4
b	1

addition(5,0)

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Example III – exécution

addition(3,2)

addition(3,2)

a	3
b	2

addition(4,1)

a	4
b	1

addition(5,0)

a	5
b	0

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Example III – exécution

addition(3,2)

addition(3,2)

a	3
b	2

addition(4,1)

a	4
b	1

← addition(5,0)

a	5
b	0

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Example III – exécution

`addition(3,2)`

`addition(3,2)`

a	3
b	2

`addition(4,1)`

a	4
b	1

`addition(5,0)`

a	5
b	0

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Example III – exécution

addition(3,2)

addition(3,2)

a	3
b	2

addition(4,1)

a	4
b	1

5

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Example III – exécution

addition(3,2)

addition(3,2)

a	3
b	2

← **addition(4,1)**

a	4
b	1

5

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Example III – exécution

addition(3,2)

addition(3,2)

a	3
b	2

addition(4,1)

a	4
b	1

5

```c
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Example III – exécution

addition(3,2)

addition(3,2)

a	3
b	2

5

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Example III – exécution

```
addition(3,2)

← addition(3,2)
```

a	3
b	2

5

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Example III – exécution

```
addition(3,2)

addition(3,2)
```

a	3
b	2

5

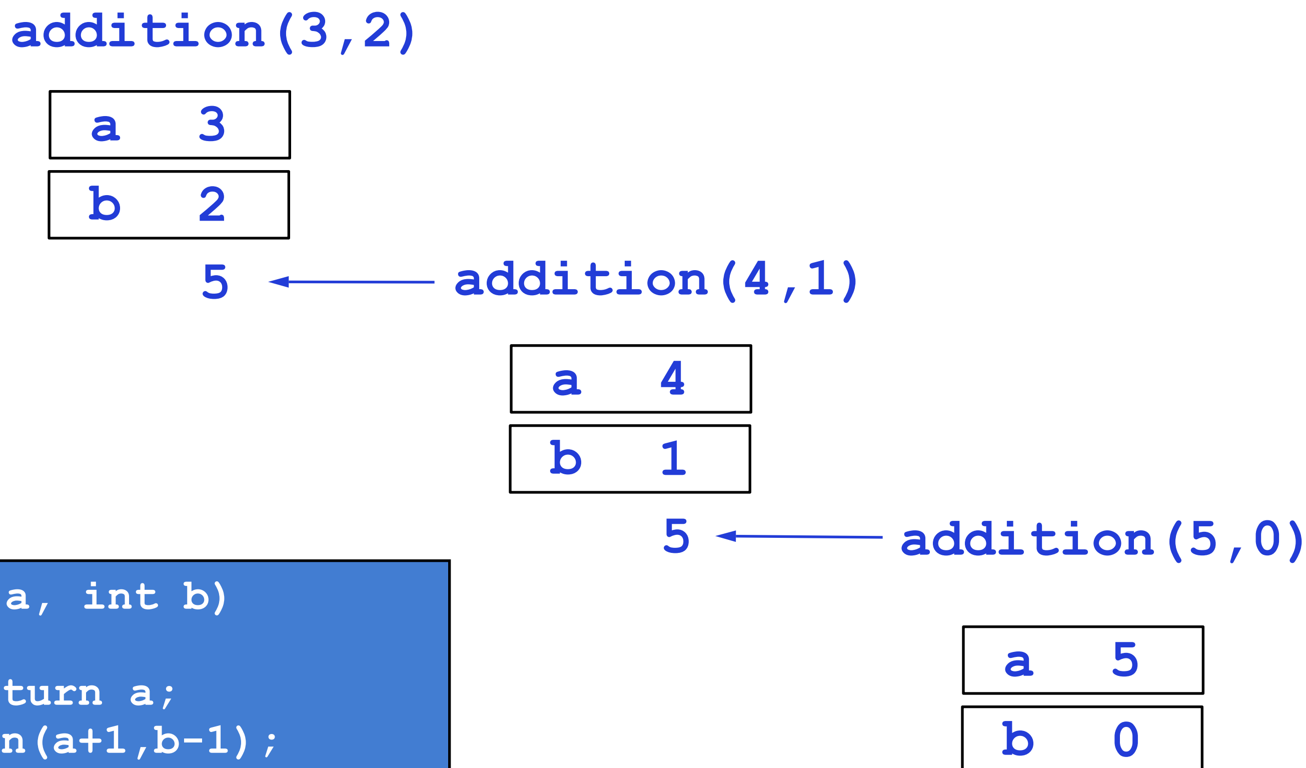
```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Example III – exécution

`addition(3,2)` → `5`

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Example III – stack




```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Example III – stack

5 ← addition(3,2)

a	3
b	2

5 ← addition(4,1)

a	4
b	1

5 ← addition(5,0)

a	5
b	0

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Terminal Recursivity

```
int addition(int a, int b)
{
  if (b == 0) return a;
  return addition(a+1,b-1);
}
```

Terminal Recursivity

```
int addition(int a, int b)
{
  if (b == 0) return a;
  return addition(a+1,b-1);
}
```

The recursive call is the last instruction of the function

Terminal Recursivity

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

The recursive call is the last instruction of the function
⇒ when the result is obtained (**b=0**) it is sent when poping the stack

Terminal Recursivity

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

The recursive call is the last instruction of the function

⇒ when the result is obtained (`b=0`) it is sent when poping the stack

there is no need to store the locals a, b

Multiple Recursion

To evaluate a function, some function need to evaluate itself many times
Eg : Fibonacci

```
F(n) = F(n-2) + F(n-1)
F(0) = 1
F(1) = 1
```

Multiple Recursion

To evaluate a function, some function need to evaluate itself many times
Eg : Fibonacci

```
F(n) = F(n-2) + F(n-1)
F(0) = 1
F(1) = 1
```

Stopping rules

Example IV

The Fibonacci sequence can be implemented as follows :

```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Example IV

The Fibonacci sequence can be implemented as follows :

```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Stopping rules

Example IV – execution

```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Example IV – execution

fibonacci(4)

```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Example IV – execution

`fibonacci(4)`

`fibonacci(2)`

```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Example IV – execution

`fibonacci(4)`

`fibonacci(2)`

`fibonacci(0)`

```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Example IV – execution

fibonacci(4)

fibonacci(2)

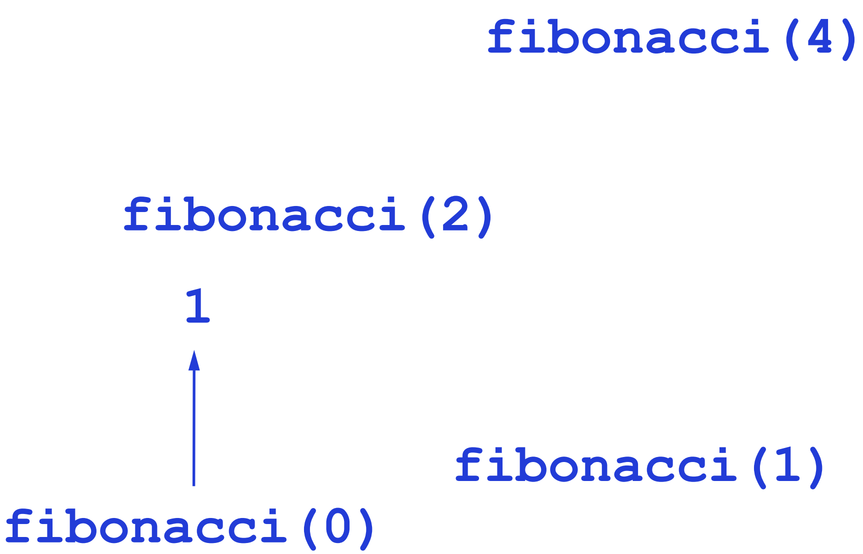
1

↑

fibonacci(0)

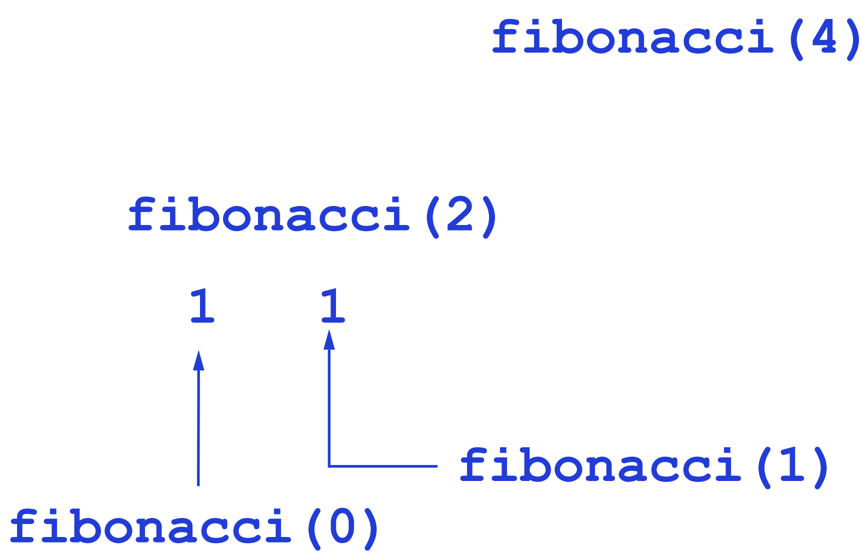
```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Example IV – execution



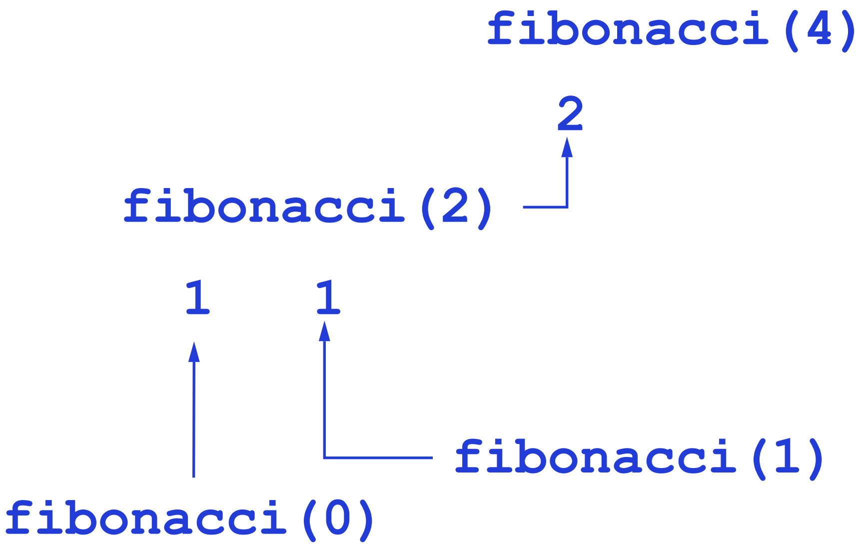

```c
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Example IV – execution



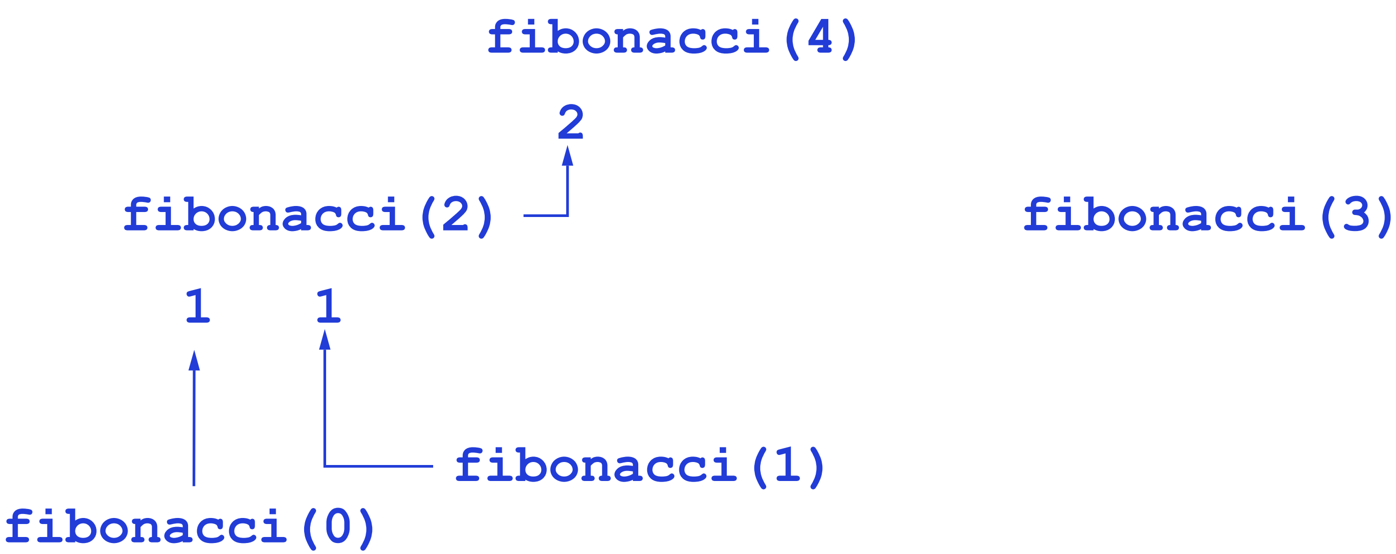

```c
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Example IV – execution



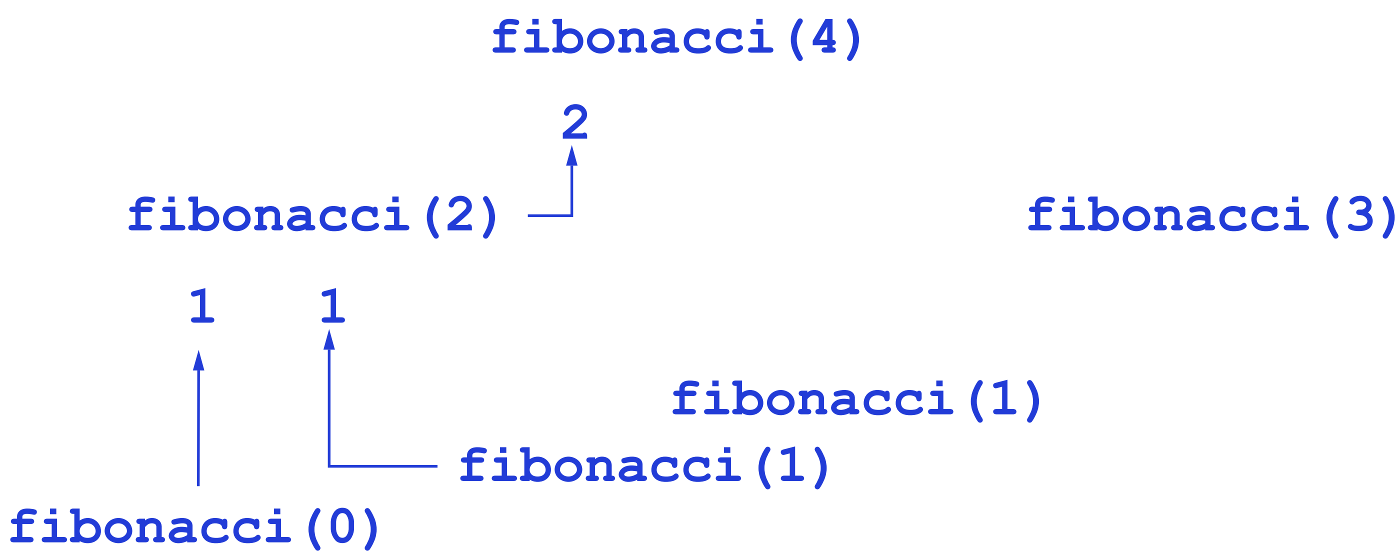

```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Example IV – execution



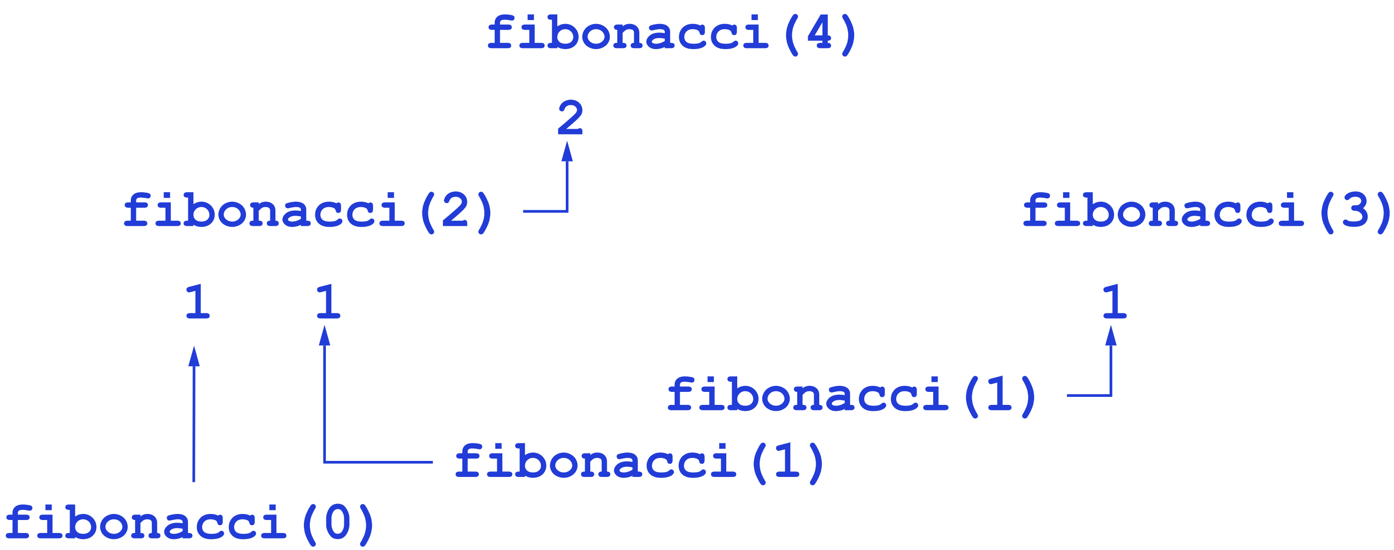

```c
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Example IV – execution



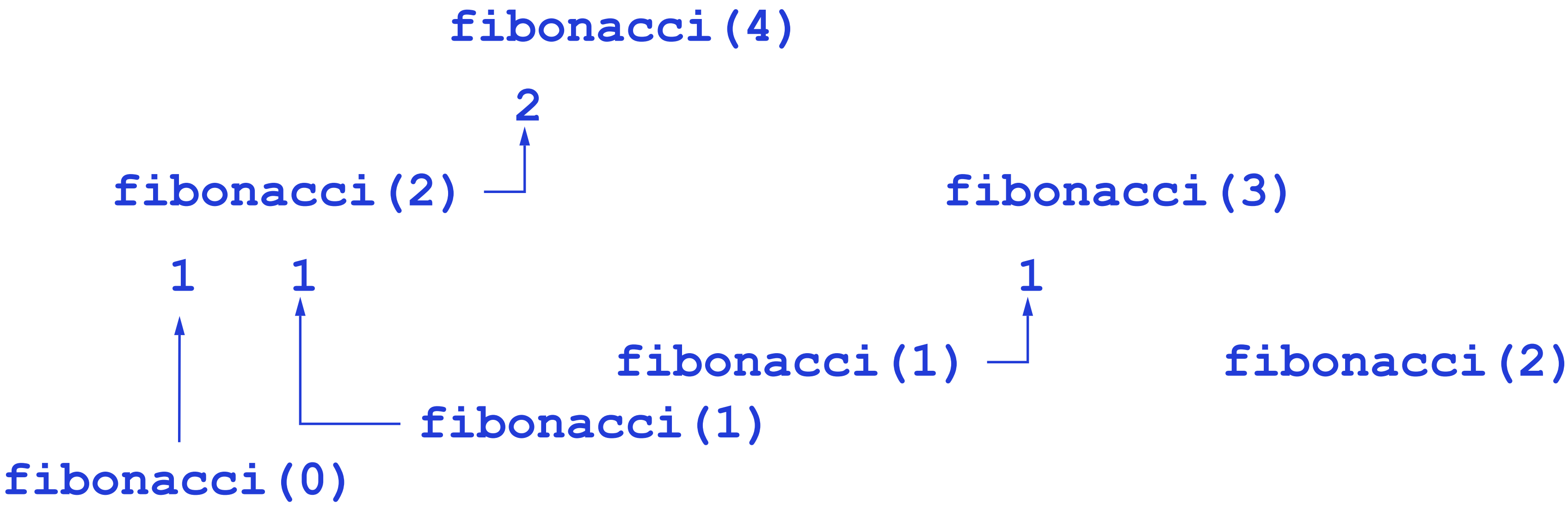

```c
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Example IV – execution



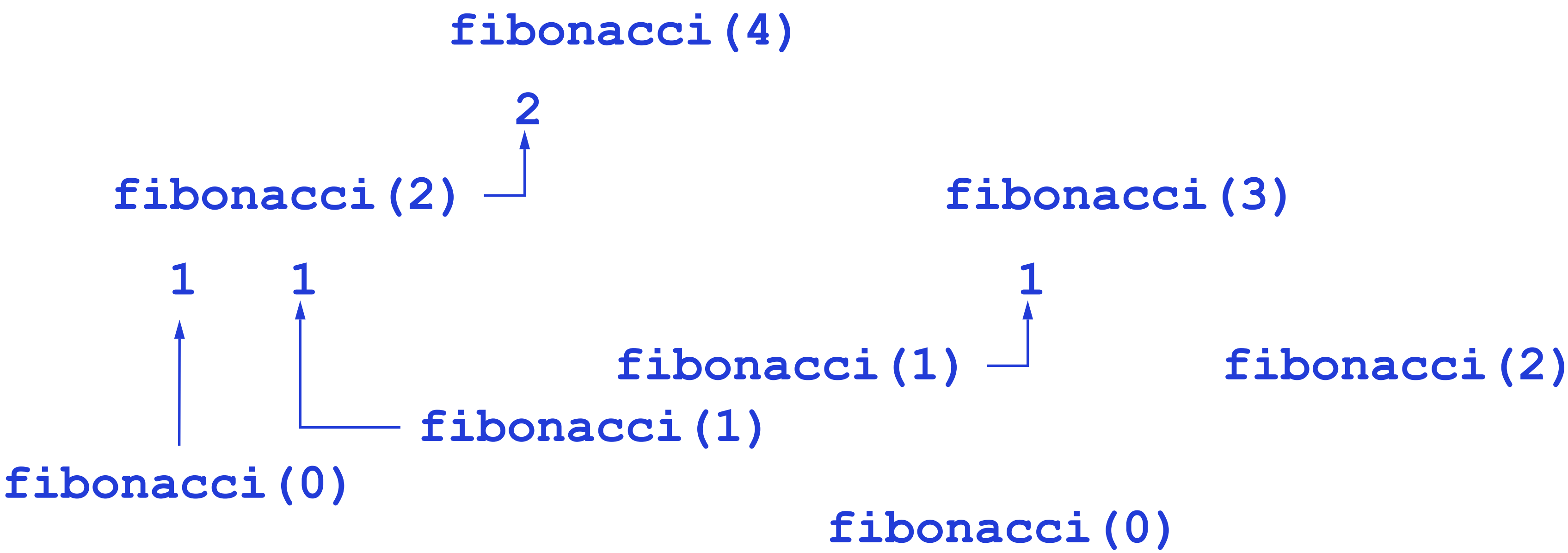

```c
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Example IV – execution



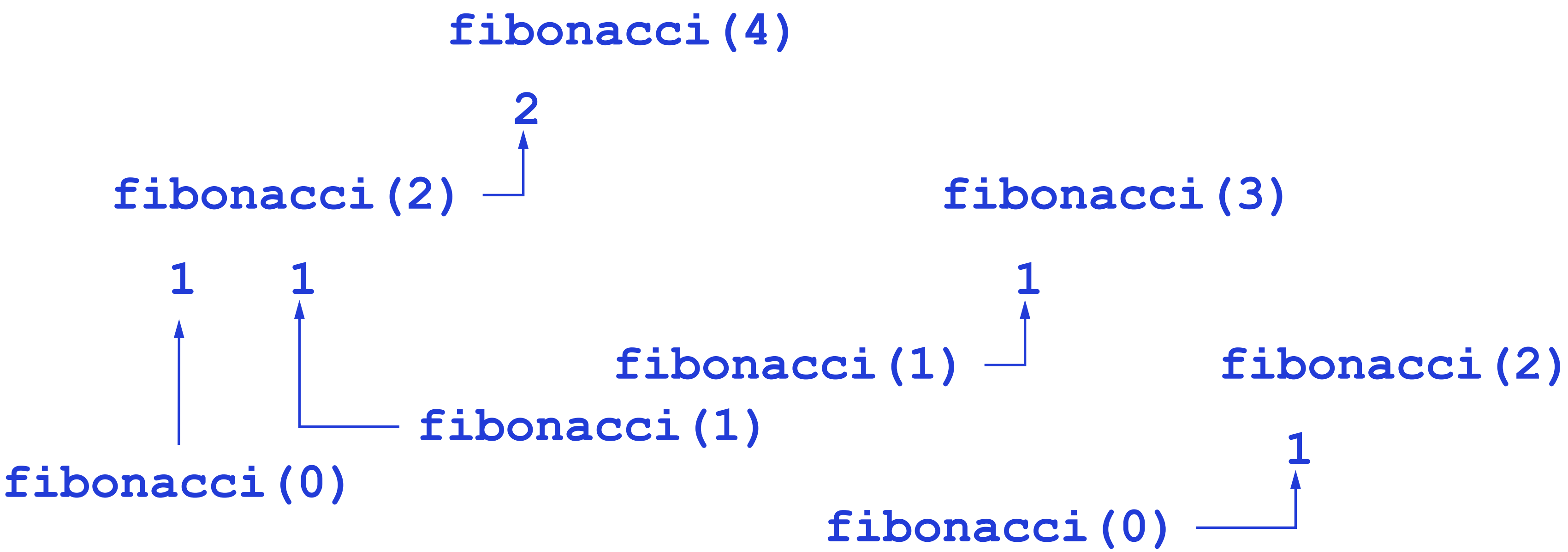

```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Example IV – execution




```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Example IV – execution




```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Example IV – execution

fibonacci(4)

2

fibonacci(2)

fibonacci(3)

1 1

1

fibonacci(1)

fibonacci(2)

fibonacci(1)

1

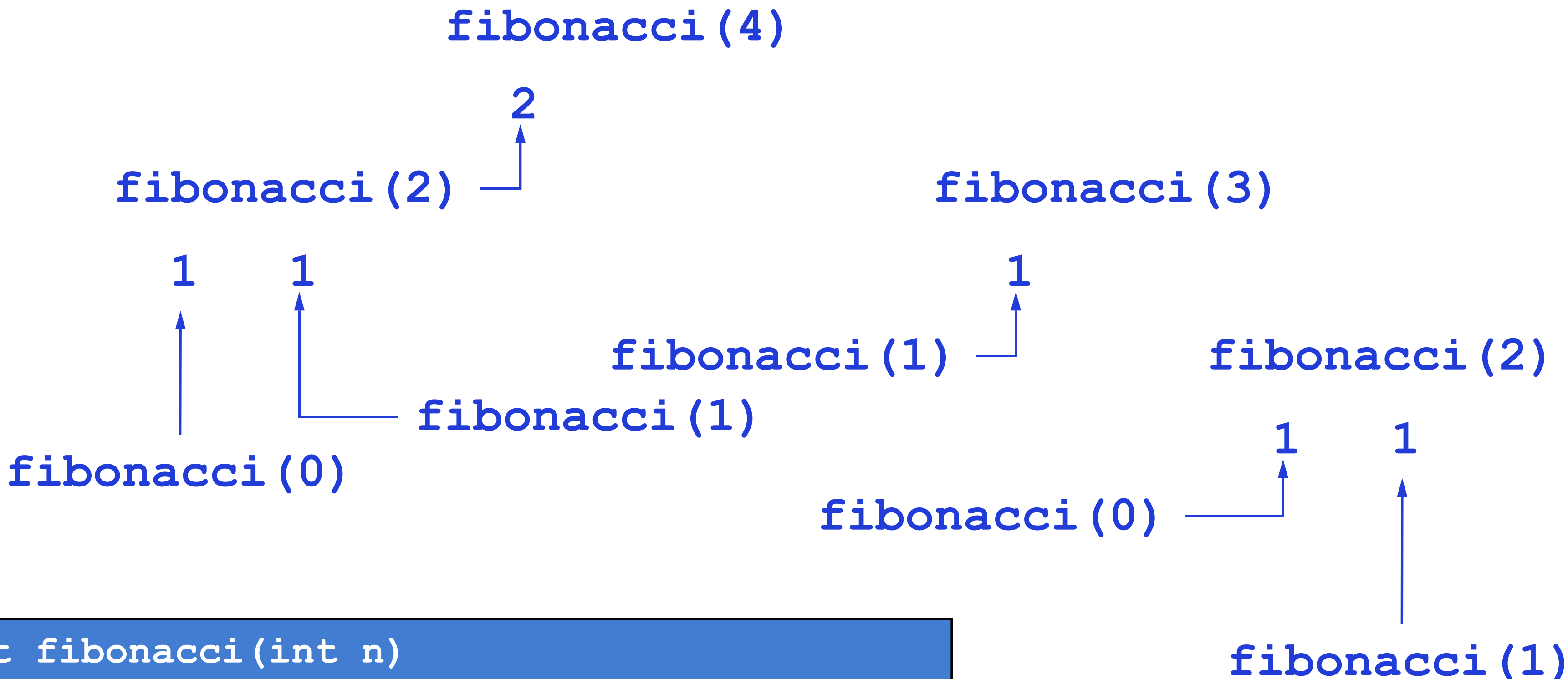
fibonacci(0)

fibonacci(0)

```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

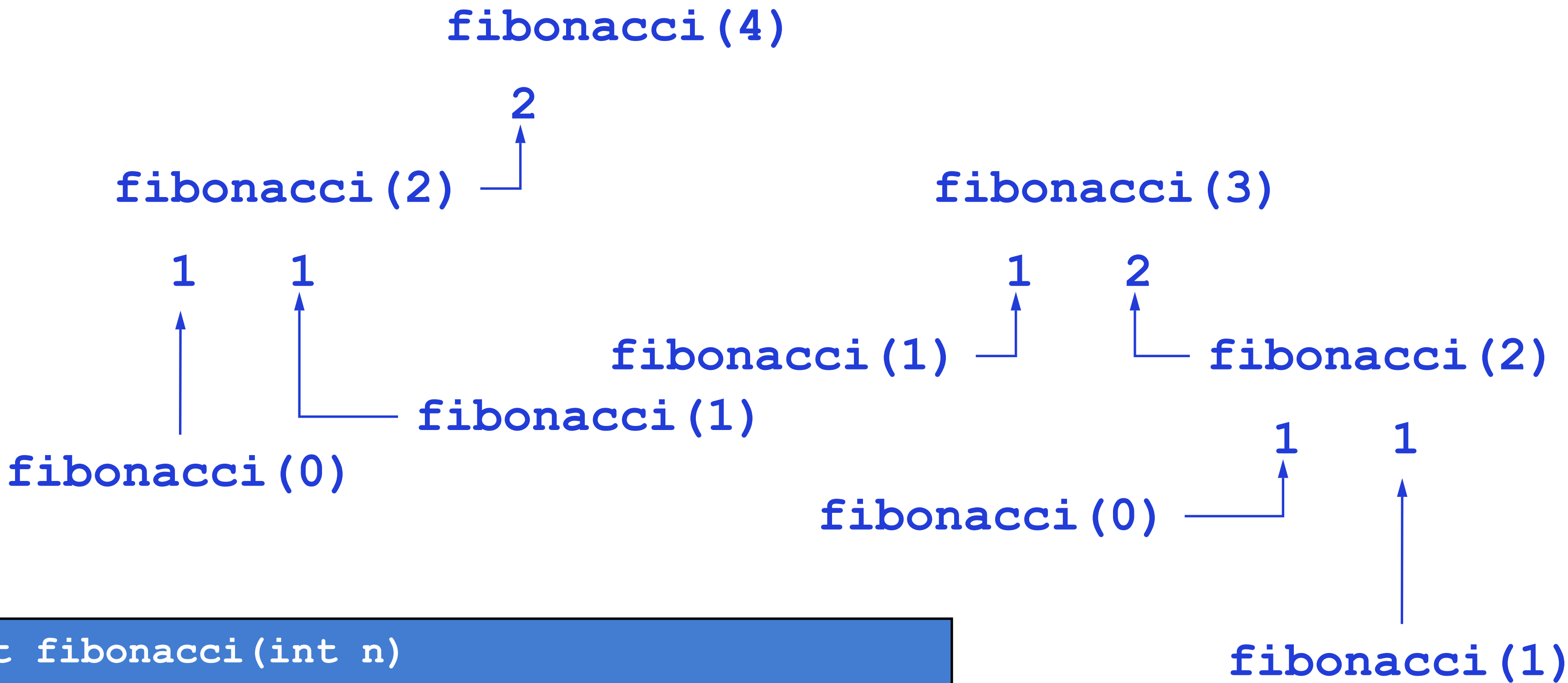
fibonacci(1)

Example IV – execution



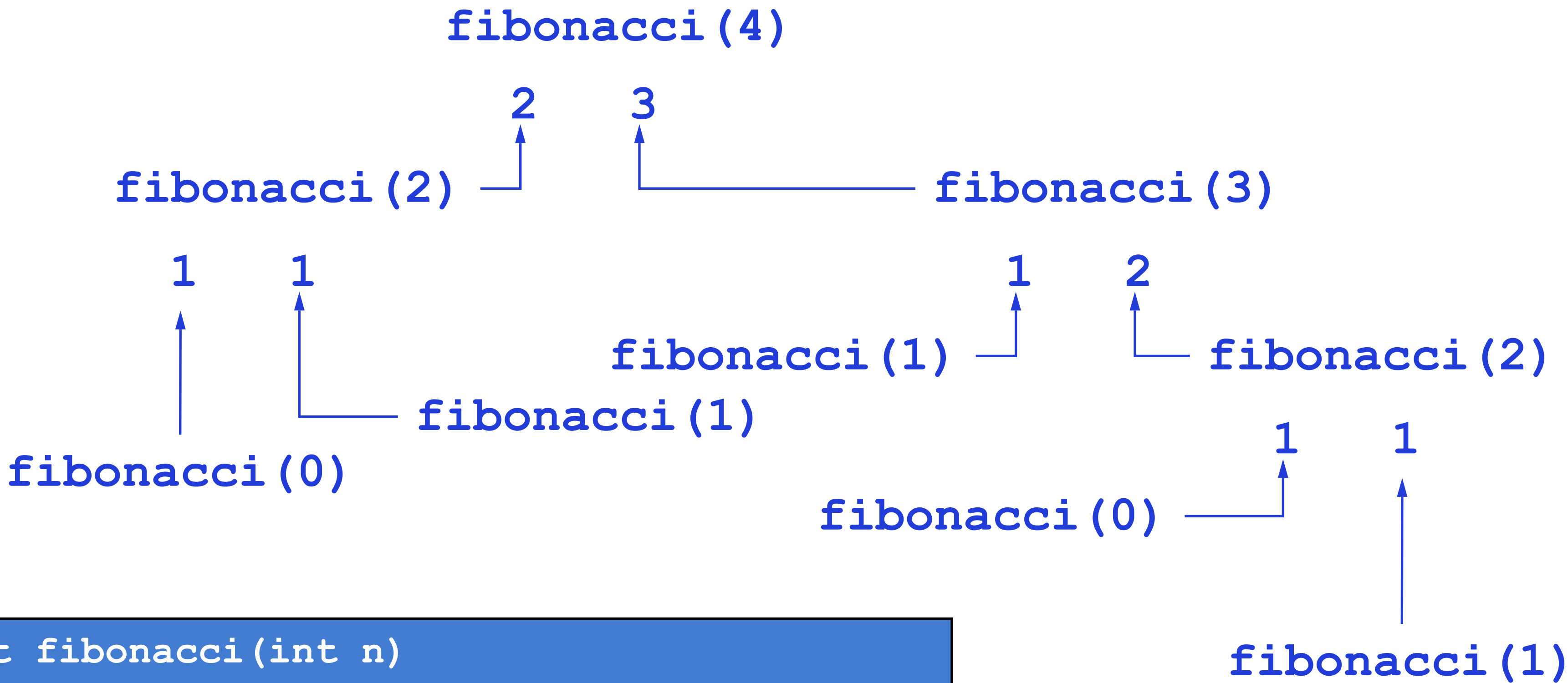

```c
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Example IV – execution



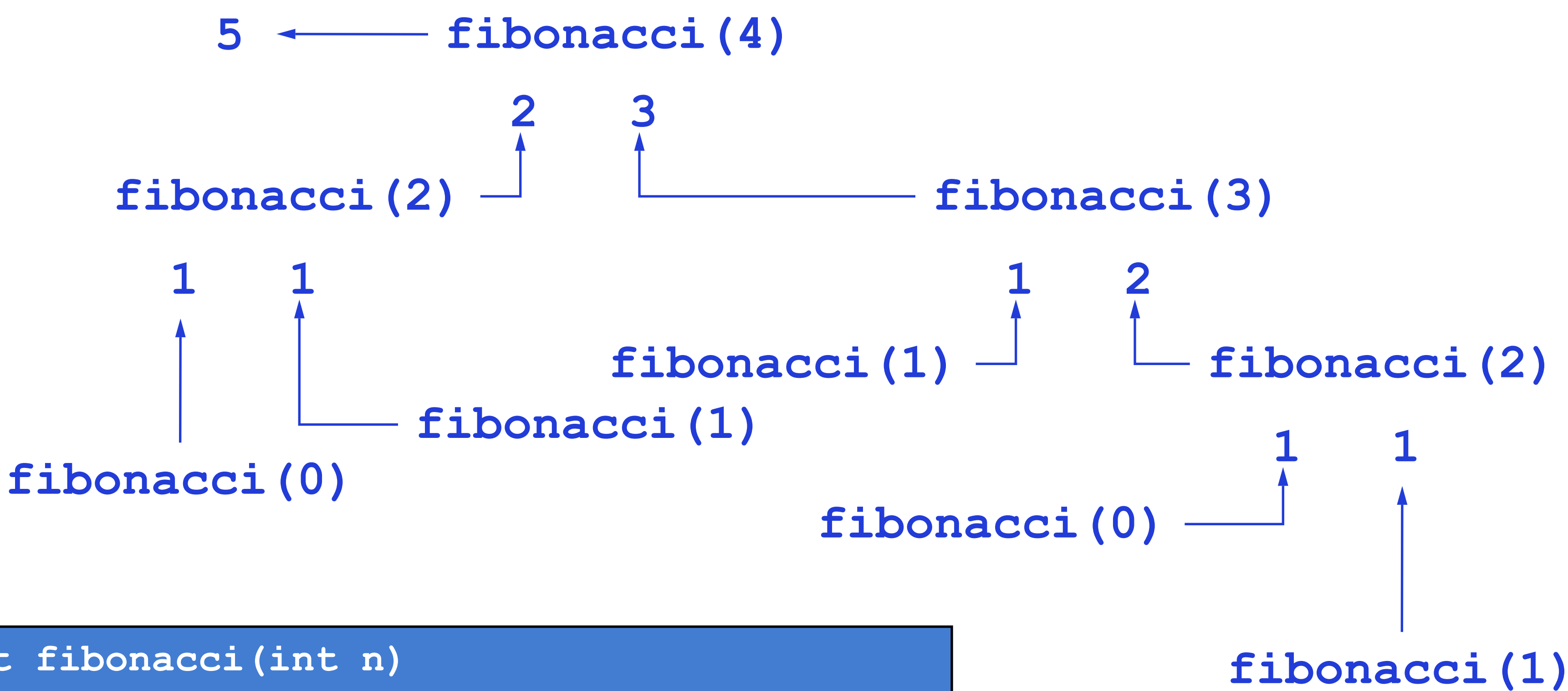

```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Example IV – execution



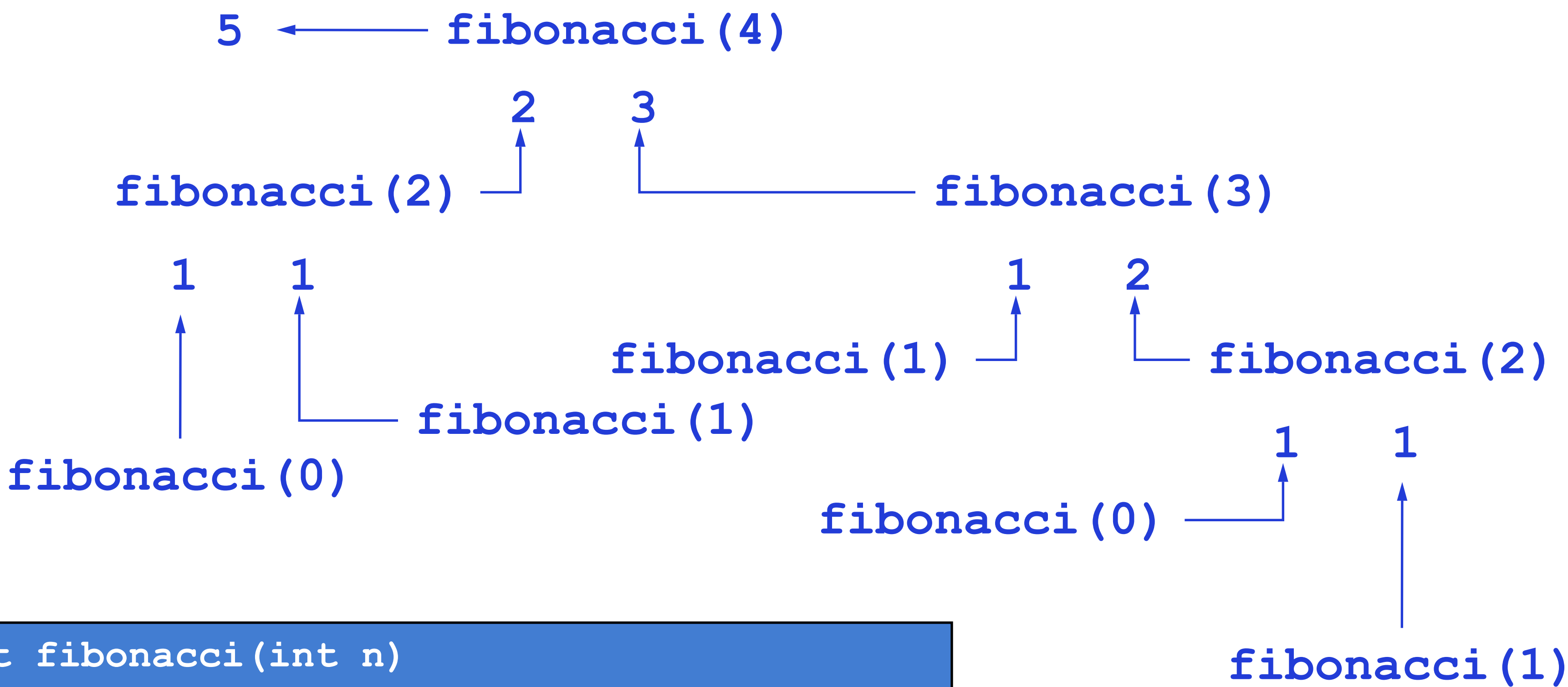

```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Example IV – execution




```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Example IV – execution




```c
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

⇒ not efficient!

Advantage of Recursivity

Programming recursively is close to the mathematical definition
$\Rightarrow$ easy to implement

But, do not forget the **stopping rules**, otherwise you have an infinite loop !

Conclusion

Structures :

- allow to put together objects of different types (eg a function can return many values of different types)
- definition of new types than can be used as other basic types