Lecture 15

Chapter 11 *Recursion*

11.3 Functional Recursion

11.4 Binary Search

11.3 Functional Recursion

Functional recursion is a method of defining functions in which the function being defined is applied within its own definition.

Example 1: Fibonacci sequence:

$F(0)=1$ (*base case*)

$F(1)=1$ (*base case*)

$F(n)=F(n-1) + F(n-2)$ for all integers $n>1$ (*recursive def.*)

Example 2: Factorial

Recall that usually we have this definition:

$n! = 1*2*3*4*5*...*(n-2)*(n-1)*n$

11.3 Functional Recursion

Functional recursion is a method of defining functions in which the function being defined is applied within its own definition.

Example 1: Fibonacci sequence:

$F(0)=1$ (*base case*)

$F(1)=1$ (*base case*)

$F(n)=F(n-1) + F(n-2)$ for all integers $n>1$ (*recursive def*)

Example 2: Factorial:

Recall that usually we have this definition:

$n! = \underbrace{1*2*3*4*5*...*(n-2)*(n-1)}_{(n-1)!}*n$

Note

11.3 Functional Recursion

Functional recursion is a method of defining functions in which the function being defined is applied within its own definition.

Example 1: Fibonacci sequence:

$F(0)=1$ (*base case*)

$F(1)=1$ (*base case*)

$F(n)=F(n-1) + F(n-2)$ for all integers $n>1$ (*recursive def.*)

Example 2: Factorial:

Recall that usually we have this definition:

$n! = 1*2*3*4*5*...*(n-2)*(n-1)*n$

Therefore we can give a recursive definition: $\boldsymbol{1! = 1}$

$\boldsymbol{n! = n*(n-1)!}$

11.3 Functional Recursion

Let's see the program that finds factorial of a number, using recursive definition: $1! = 1 \quad n! = n*(n-1)!$

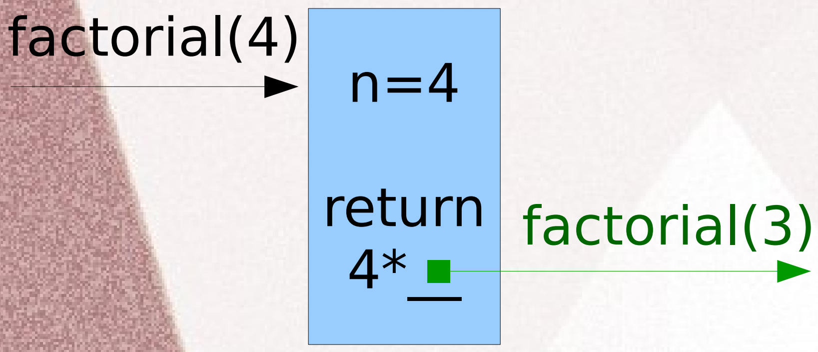
Here is a draft of the program:

```
def factorial(n):
    if n == 1: return 1
     else:
           return n*factorial(n-1)

def main():
      n=input('please, input n:')

      F=factorial(n)

      print("%d! = %d"%(n,F))

main()
```

11.3 Functional Recursion

Let's trace the call of factorial(4):

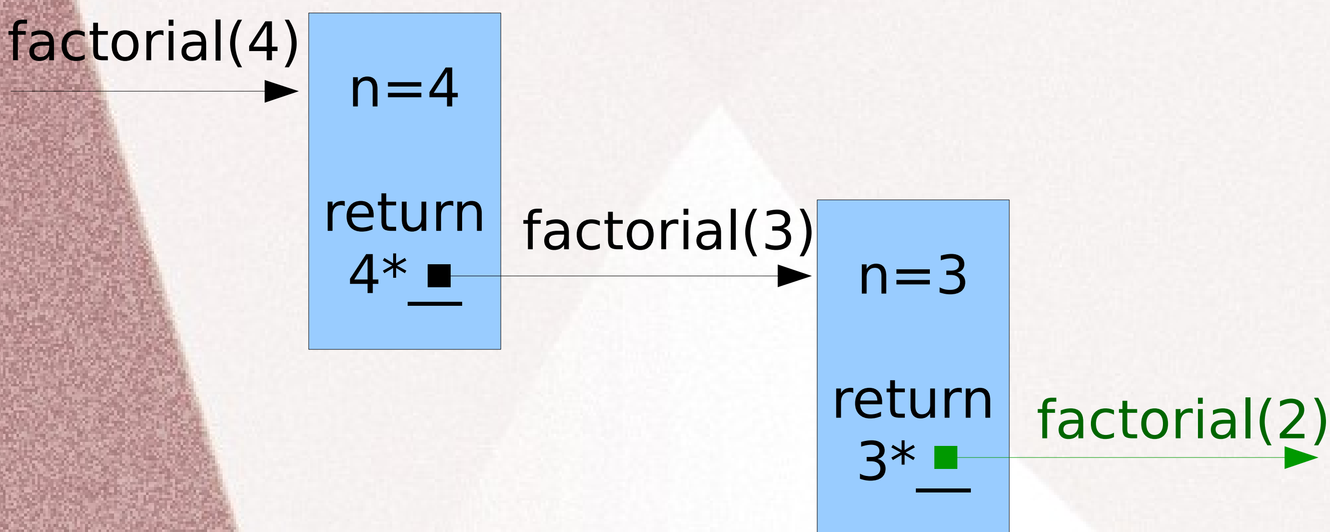
```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
```



11.3 Functional Recursion

Let's trace the call of factorial(4):

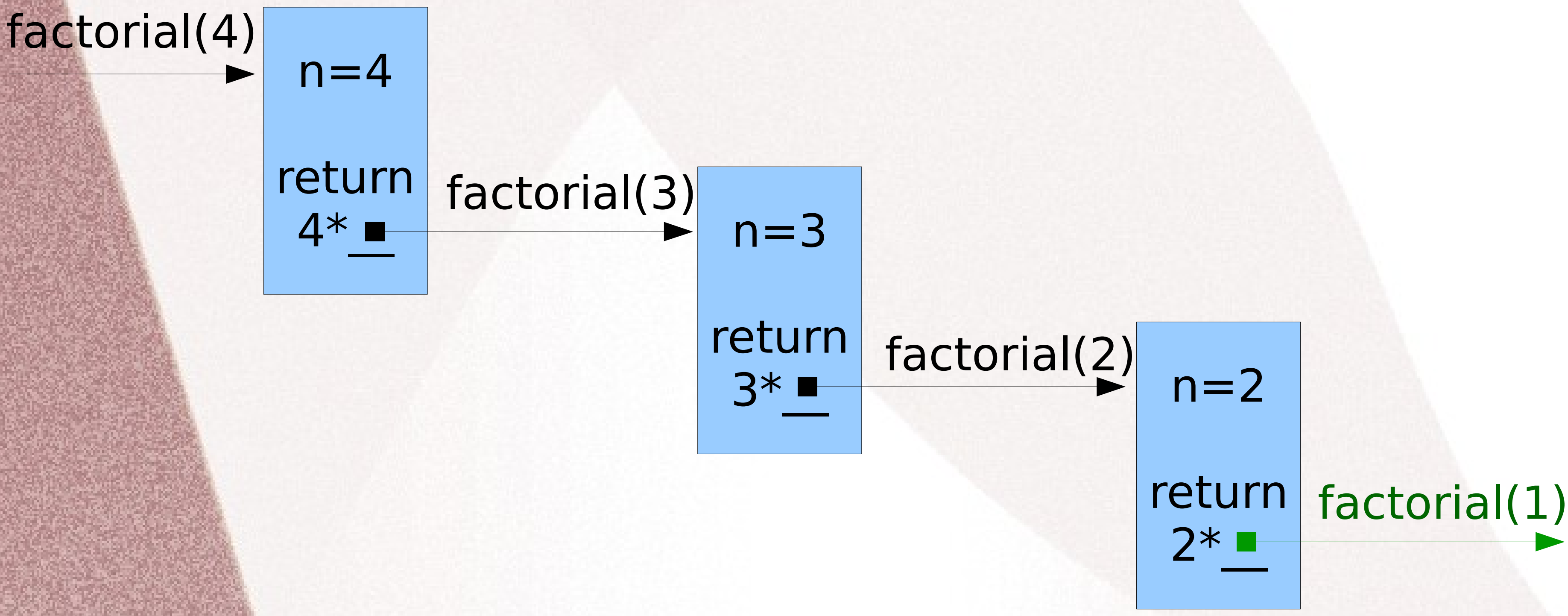
```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
```



11.3 Functional Recursion

Let's trace the call of factorial(4):

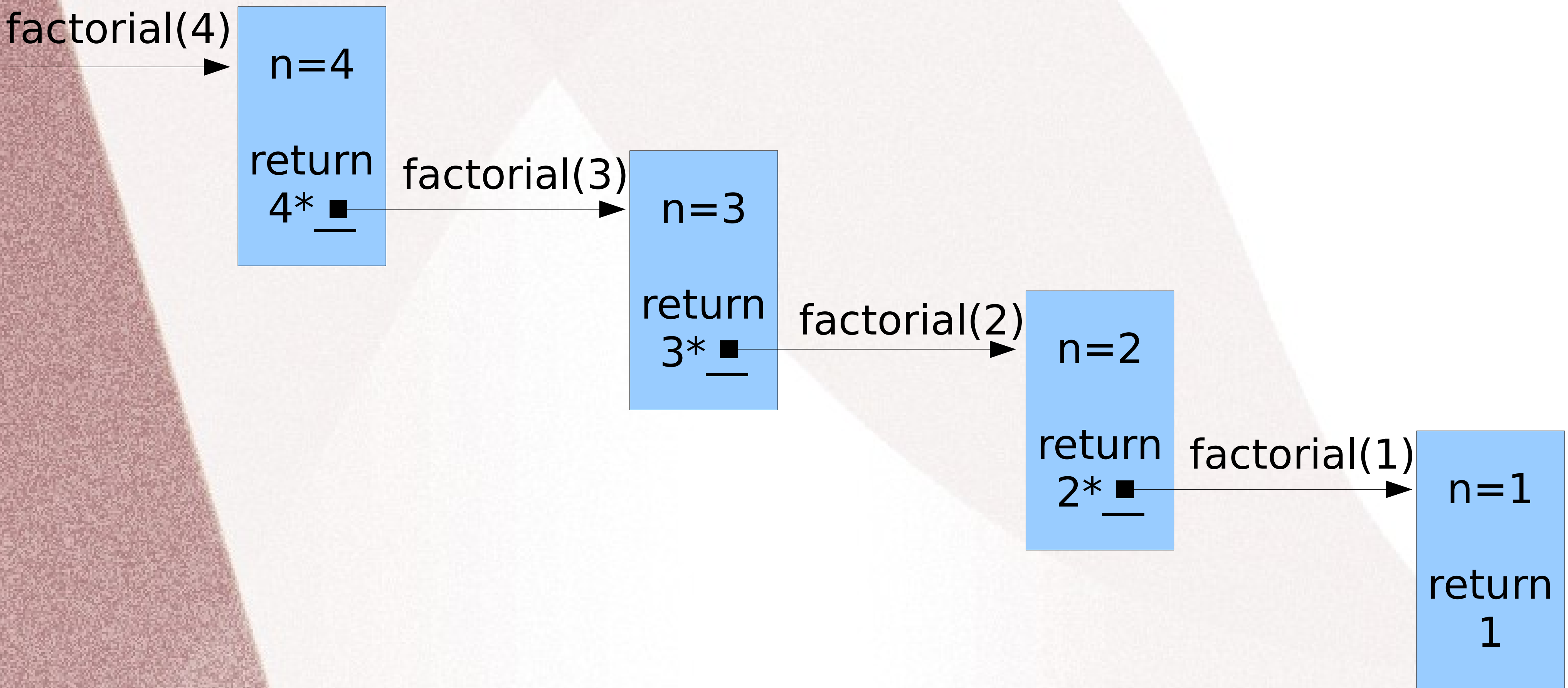
```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
```



11.3 Functional Recursion

Let's trace the call of factorial(4):

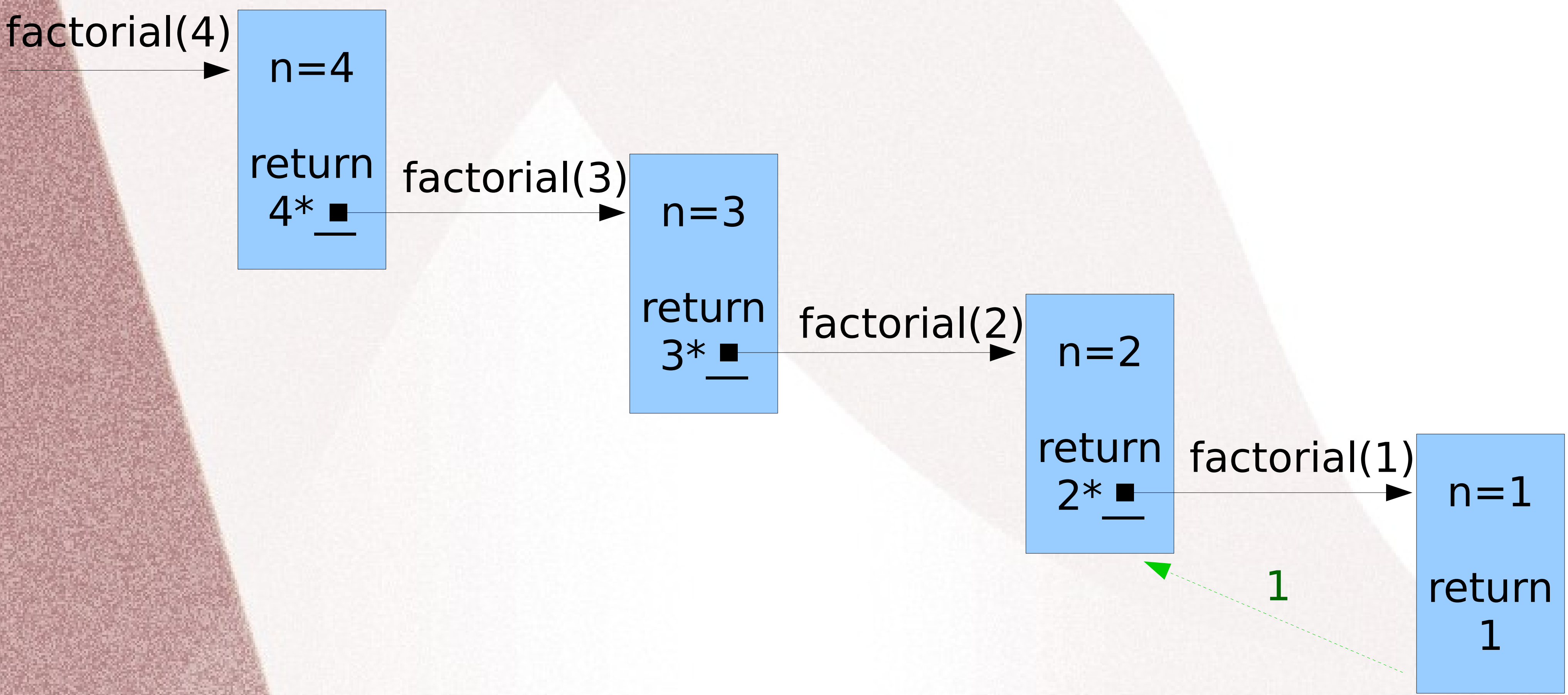
```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
```



11.3 Functional Recursion

Let's trace the call of factorial(4):

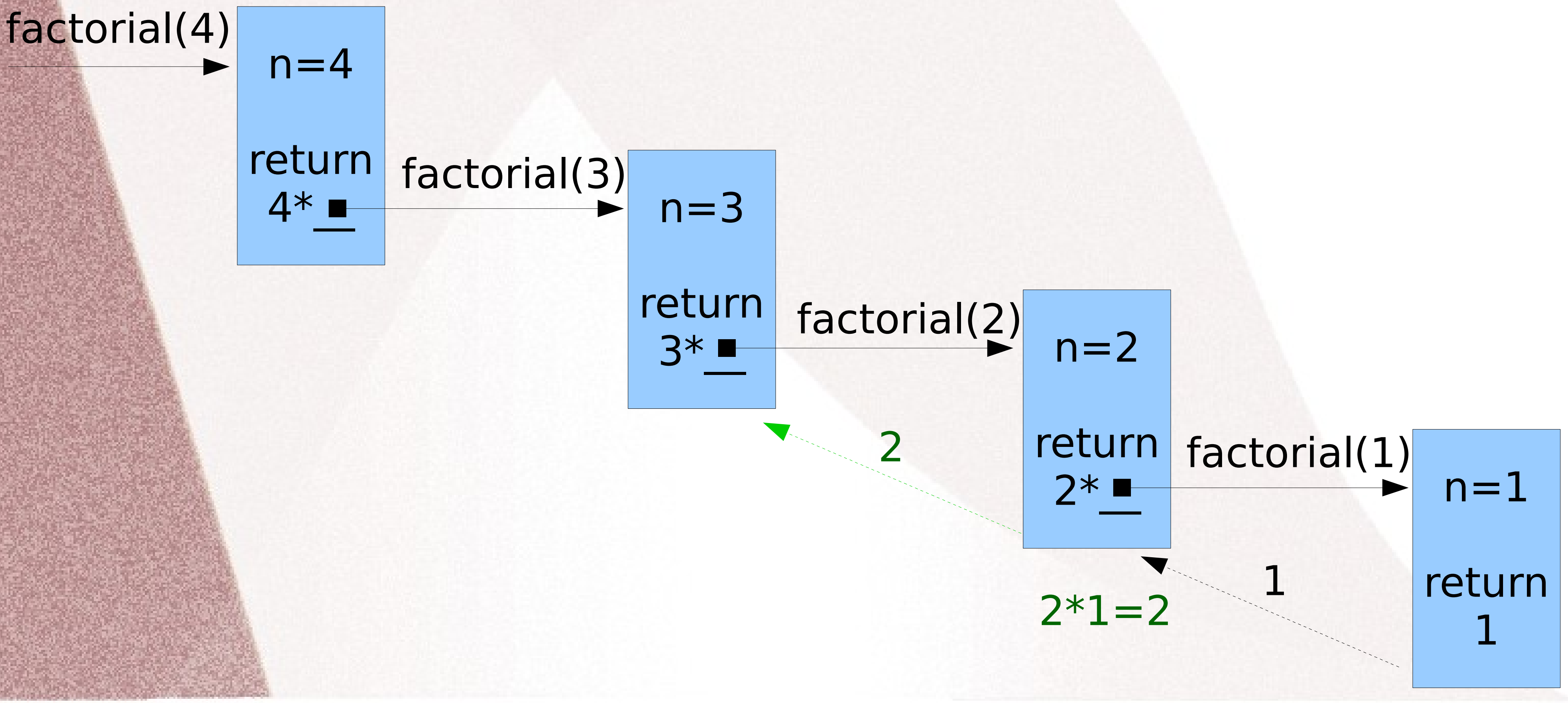
```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
```



11.3 Functional Recursion

Let's trace the call of factorial(4):

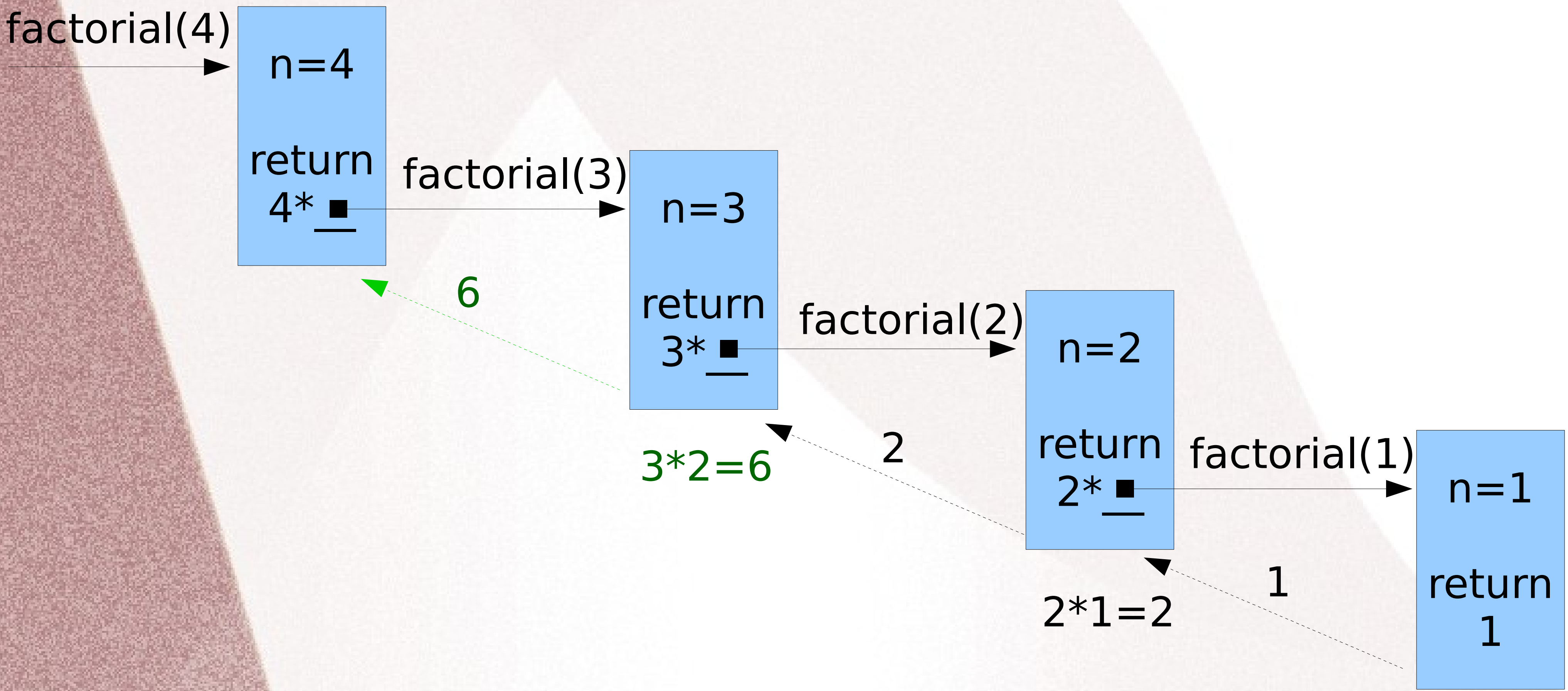
```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
```



11.3 Functional Recursion

Let's trace the call of factorial(4):

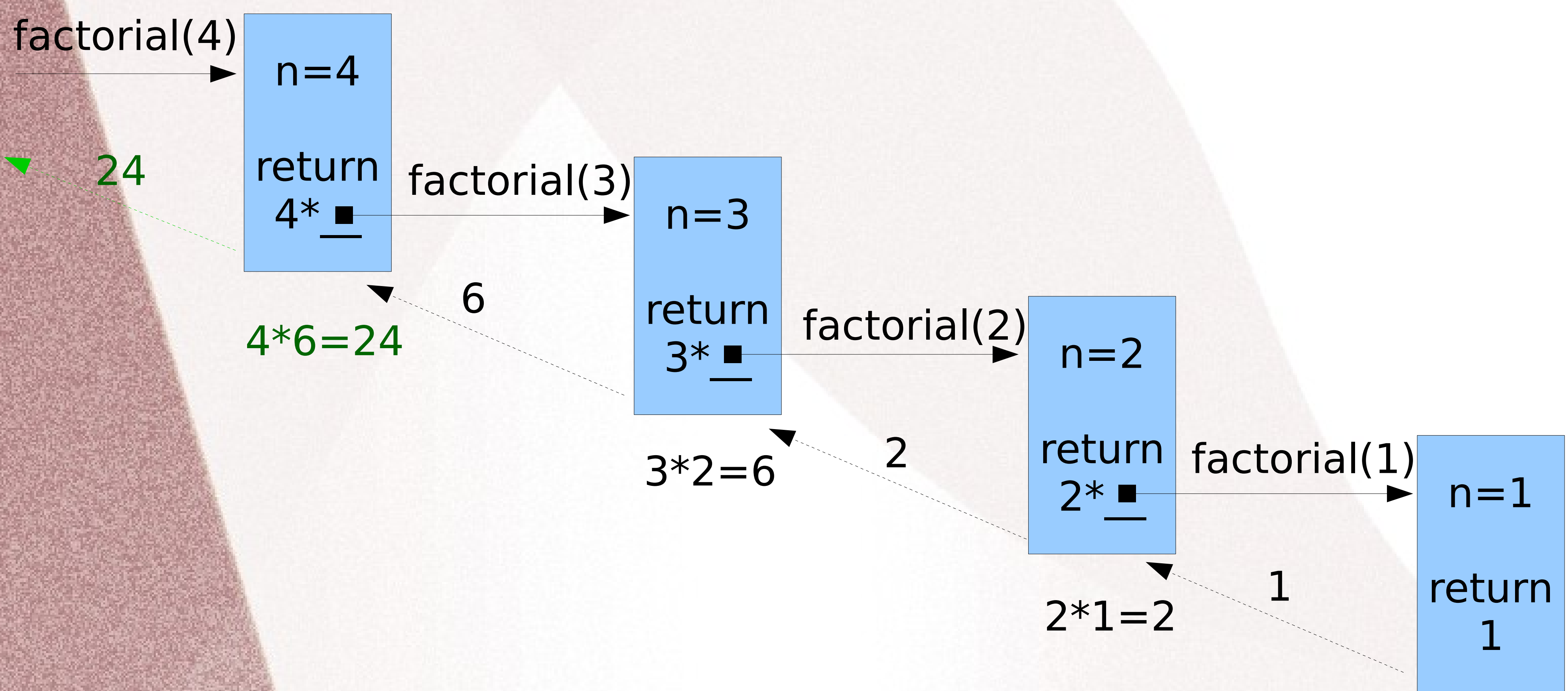
```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
```



11.3 Functional Recursion

Let's trace the call of factorial(4):

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
```



11.3 Functional Recursion

See the program factorial_rec.py

11.3 Functional Recursion

Another example of recursive function:
Ackermann function or Ackermann–Péter function.

$$A(m,n) = \begin{cases} n+1 & \text{if } m=0 \\ A(m-1,1) & \text{if } m>0 \text{ and } n=0 \\ A(m-1,A(m,n-1)) & \text{if } m>0 \text{ and } n>0 \end{cases}$$

This function has only recursive definition, and it grows very fast.

Let's see how it works:
A(0,4) = 5 A(0,7) = 8 A(1,0) = A(0,1) = 2

11.3 Functional Recursion

Another example of recursive function:
Ackermann function or Ackermann–Péter function.

$$A(m,n) = \begin{cases} n+1 & \text{if } m=0 \\ A(m-1,1) & \text{if } m>0 \text{ and } n=0 \\ A(m-1,A(m,n-1)) & \text{if } m>0 \text{ and } n>0 \end{cases}$$

This function has only recursive definition, and it grows very fast.

Let's see how it works:
A(0,4) = 5 A(0,7) = 8 A(1,0) = A(0,1) = 2
A(2,0) = A(1,1) = A(0, A(1,0))=A(0,2)=3

11.3 Functional Recursion

Another example of recursive function:
Ackermann function or Ackermann–Péter function.

$$A(m,n) = \begin{cases} n+1 & \text{if } m=0 \\ A(m-1,1) & \text{if } m>0 \text{ and } n=0 \\ A(m-1,A(m,n-1)) & \text{if } m>0 \text{ and } n>0 \end{cases}$$

This function has only recursive definition, and it grows very fast.

Let's see how it works:

A(0,4) = 5 A(0,7) = 8 A(1,0) = A(0,1) = 2

A(2,0) = A(1,1) = A(0, A(1,0))=A(0,2)=3

A(1,2) = A(0, A(1,1))=A(0,3)=4

11.3 Functional Recursion

Another example of recursive function:
Ackermann function or Ackermann–Péter function.

$$A(m,n) = \begin{cases} n+1 & \text{if } m=0 \\ A(m-1,1) & \text{if } m>0 \text{ and } n=0 \\ A(m-1,A(m,n-1)) & \text{if } m>0 \text{ and } n>0 \end{cases}$$

This function has only recursive definition, and it grows very fast.

Let's see how it works:

A(0,4) = 5 A(0,7) = 8 A(1,0) = A(0,1) = 2

A(2,0) = A(1,1) = A(0, A(1,0))=A(0,2)=3

A(1,2) = A(0, A(1,1))=A(0,3)=4

A(2,1) = A(1, A(2,0))=A(1,3)=A(0, A(1,2))=A(0,4)=5

11.3 Functional Recursion

Another example of recursive function:
Ackermann function or Ackermann–Péter function.

$$A(m,n) = \begin{cases} n+1 & \text{if } m=0 \\ A(m-1,1) & \text{if } m>0 \text{ and } n=0 \\ A(m-1,A(m,n-1)) & \text{if } m>0 \text{ and } n>0 \end{cases}$$

This function has only recursive definition, and it grows very fast.

Let's see how it works:

A(0,4) = 5 A(0,7) = 8 A(1,0) = A(0,1) = 2

A(2,0) = A(1,1) = A(0, A(1,0))=A(0,2)=3

A(1,2) = A(0, A(1,1))=A(0,3)=4

A(2,1) = A(1, A(2,0))= A(1,3)=A(0, A(1,2))=A(0,4)=5

A(3,2) = A(2,A(3,1))=A(2,A(2,A(3,0)))=A(2,A(2,A(2,1)))=
A(2,A(2,5))=A(2,A(1,A(2,4)))=A(2,A(1,A(1,A(2,3))))=
A(2,A(1,A(1,A(1,A(2,2)))))=A(2,A(1,A(1,A(1,A(1,A(2,1))))))=
A(2,A(1,A(1,A(1,A(1,5)))))=A(2,A(1,A(1,A(1,A(0,A(1,4))))))=
A(2,A(1,A(1,A(1,A(0,A(0,A(1,3)...)=
A(2,A(1,A(1,A(1,A(0,A(0,A(0,A(1,2))...)=

=A(2,A(1,A(1,A(1,A(0,A(0,A(0,4)...)=A(2,A(1,A(1,A(1,A(0,A(0,5)...)=
A(2,A(1,A(1,A(1,A(0,6)...)=A(2,A(1,A(1,A(1,7))))=A(2,A(1,A(1,A(0,A(1,6))
)))=A(2,A(1,A(1,A(0,A(0,A(1,5)...)=A(2,A(1,A(1,A(0,A(0,A(0,A(1,4)...)=
A(2,A(1,A(1,A(0,A(0,A(0,A(0,A(1,3)...)=
A(2,A(1,A(1,A(0,A(0,A(0,A(0,A(0,A(1,2)...)=
A(2,A(1,A(1,A(0,A(0,A(0,A(0,A(0,4)...)=A(2,A(1,A(1,A(0,A(0,A(0,A(0,5)...)
=A(2,A(1,A(1,A(0,A(0,A(0,6)...)=A(2,A(1,A(1,A(0,A(0,7)...)=
A(2,A(1,A(1,A(0,8)...)=A(2,A(1,A(1,9)...)=A(2,A(1,A(0,A(1,8))))=
A(2,A(1,A(0,A(0,A(1,7)...)=A(2,A(1,A(0,A(0,A(0,A(1,6)...)=
A(2,A(1,A(0,A(0,A(0,A(0,A(1,5)...)=
A(2,A(1,A(0,A(0,A(0,A(0,A(0,A(1,4)...)=
A(2,A(1,A(0,A(0,A(0,A(0,A(0,A(0,A(1,3)...)=
A(2,A(1,A(0,A(0,A(0,A(0,A(0,A(0,A(0,A(1,2)...)=
A(2,A(1,A(0,A(0,A(0,A(0,A(0,A(0,A(0,4)...)=
A(2,A(1,A(0,A(0,A(0,A(0,A(0,A(0,5)...)=
A(2,A(1,A(0,A(0,A(0,A(0,A(0,6)...)=A(2,A(1,A(0,A(0,A(0,A(0,7)...)=
A(2,A(1,A(0,A(0,A(0,8)...)=A(2,A(1,A(0,A(0,9))))=A(2,A(1,A(0,10)))=
A(2,A(1,11))=A(2,A(0,A(1,10)))=A(2,A(0,A(0,A(1,9))))=
A(2,A(0,A(0,A(0,A(1,8)...)=A(2,A(0,A(0,A(0,A(1,8)...)=
A(2,A(0,A(0,A(0,A(0,A(1,7)...)=A(2,A(0,A(0,A(0,A(0,A(0,A(1,6)...)=
A(2,A(0,A(0,A(0,A(0,A(0,A(0,A(1,5)...)=
A(2,A(0,A(0,A(0,A(0,A(0,A(0,A(0,A(1,4)...)=
A(2,A(0,A(0,A(0,A(0,A(0,A(0,A(0,A(0,A(1,3)...)=

A(2,A(0,A(0,A(0,A(0,A(0,A(0,A(0,A(0,5)...)=
A(2,A(0,A(0,A(0,A(0,A(0,A(0,A(0,6)...)=
A(2,A(0,A(0,A(0,A(0,A(0,A(0,7)...)=A(2,A(0,A(0,A(0,A(0,A(0,8)...)=
A(2,A(0,A(0,A(0,A(0,9)...)=A(2,A(0,A(0,A(0,10)...)=A(2,A(0,A(0,11)...)=
A(2,A(0,12)...)=A(2,13)...)=A(1,A(2,12))=A(1,A(1,A(2,11)))=
A(1,A(1,A(1,A(2,10))))=A(1,A(1,A(1,A(1,A(2,9)))))=
A(1,A(1,A(1,A(1,A(1,A(2,8)...)=A(1,A(1,A(1,A(1,A(1,A(1,A(2,7)...)=
A(1,A(1,A(1,A(1,A(1,A(1,A(1,A(2,6)...)= ... = 29

11.4 Binary Search

A binary search algorithm locates the position of an element (a number, a letter, a word, ...) in a **sorted list**.

It inspects the middle element of the sorted list:

if equal to the sought value,

then the position has been found;

otherwise,

the lower(left) half or upper(right) half is chosen for further searching *based on*:

whether the sought value is less than or greater than the middle element.

This method reduces the number of elements needed to be checked by a factor of two each time, and finds the sought value if it exists in the list or if not determines "not present".

Complexity: logarithmic

i.e. if we start with n elements, the algorithm will terminate in at most $k \times log_2 n$ steps, where k is a constant.

11.4 Binary Search

Our book has a nice example with lexicon (please, take a look at it). We will deal with numbers in class.

Input:
- a sorted list of numbers (integers of floating point numbers),
- a number to find (may be not present in the list)

Output: location of the element in the list (if present), 'Not in the list' (if not present)

11.4 Binary Search

Example: binary search algorithm on the following input:

[1,7,9,12,17,19,23,45,67,123,167] find: 7

11.4 Binary Search

Example: binary search algorithm on the following input:

[1,7,9,12,17,19,23,45,67,123,167] find: 7

start stop

Number of elements if the array: 11,
start index = 0,
stop index = 10

11.4 Binary Search

Example: binary search algorithm on the following input:

[1,7,9,12,17,19,23,45,67,123,167]   find: 7

start   stop

Number of elements if the array: 11,
start index = 0,
stop index = 10

Middle element: $\lfloor \frac{(start+stop)}{2} \rfloor = \lfloor \frac{(0+10)}{2} \rfloor = 5$

or (0+10) // 2 =5th

11.4 Binary Search

Example: binary search algorithm on the following input:

[1,7,9,12,17,19,23,45,67,123,167]   find: 7

start   stop

Number of elements if the array: 11,
start index = 0,
stop index = 10

Middle element: $\lfloor \frac{(start + stop)}{2} \rfloor = \lfloor \frac{(0+10)}{2} \rfloor = 5$

or (0+10) // 2 =5^{th}

5^{th} element → *19* = *7* ← the number we are looking for ? No

11.4 Binary Search

Example: binary search algorithm on the following input:

[1,7,9,12,17,19,23,45,67,123,167]   find: 7

start   stop

Number of elements if the array: 11,
start index = 0,
stop index = 10

Middle element: $\lfloor \frac{(start+stop)}{2} \rfloor = \lfloor \frac{(0+10)}{2} \rfloor = 5$

or (0+10) // 2 =5th

5th element $\rightarrow$ *19* = 7 $\leftarrow$ the number we are looking for ? No

19 $\geq$ 7 ?   True

11.4 Binary Search

Example: binary search algorithm on the following input:

[1,7,9,12,17,19,23,45,67,123,167]   find: 7

start   stop

Number of elements if the array: 11,
start index = 0,
stop index = 10

Middle element: $\lfloor \frac{(start+stop)}{2} \rfloor = \lfloor \frac{(0+10)}{2} \rfloor = 5$

or (0+10) // 2 =5th

5th element $\rightarrow$ *19* = *7* $\leftarrow$ the number we are looking for ? No

19 $\geq$ 7 ?   True   therefore take the left part (with 19)

11.4 Binary Search

[1,7,9,12,17,19]   find: 7

start   stop

Number of elements in the "new" array (sub-array): 6,
start = 0,
stop = 5

11.4 Binary Search

[1,7,9,12,17,19]   find: 7

start   stop

Number of elements in the “new” array (sub-array): 6,
start = 0,
stop = 5

Middle element: $\left\lfloor \frac{(0+5)}{2} \right\rfloor = 2$ or   (0+5)//2 = 2nd

2nd element → *9* = *7* ← the number we are looking for? No

11.4 Binary Search

[1,7,9,12,17,19]   find: 7

start ... stop

Number of elements in the “new” array (sub-array): 6,
start = 0,
stop = 5

Middle element: $\lfloor \frac{(0+5)}{2} \rfloor = 2$ or   (0+5)//2 = 2^{nd}

2^{nd} element $\rightarrow$ *9* = *7* $\leftarrow$ the number we are looking for? No

9 $\geq$ 7 ? True

11.4 Binary Search

[1,7,9,12,17,19]   find: 7

start   stop

Number of elements in the "new" array (sub-array): 6,
start = 0,
stop = 5

Middle element: $\lfloor \frac{(0+5)}{2} \rfloor = 2$ or   (0+5)//2 = 2^{nd}

2^{nd} element $\rightarrow$ *9* = *7* $\leftarrow$ the number we are looking for? No

9 $\geq$ 7 ? True   therefore take the left part (with 9)

11.4 Binary Search

[1,7,9]   find: 7

start   stop

Number of elements in the sub-array: 3,
start = 0,
stop = 2

11.4 Binary Search

[1,7,9]   find: 7

start   stop

Number of elements in the sub-array: 3,
start = 0,
stop = 2

Middle element: $\lfloor \frac{(0+2)}{2} \rfloor = 1$ or   (0+2) // 2 = 1st

11.4 Binary Search

[1,7,9]   find: 7

start   stop

Number of elements in the sub-array: 3,
start = 0,
stop = 2

Middle element: $\lfloor \frac{(0+2)}{2} \rfloor = 1$ or   (0+2) // 2 = 1^{st}

1^{st} element $\rightarrow$ 7 = 7 $\leftarrow$ *the number we are looking for*? Yes

Stop, return index 1.

11.4 Binary Search

See the program binary_search.py

Homework Assignment

Recall **Fibonacci numbers/sequence :**

$F(0)=1$ (*base case*)
$F(1)=1$ (*base case*)
$F(n)=F(n-1) + F(n-2)$ for all integers $n>1$ (*recursive definition*)

Write a program, that for a given non-negative integer *n*, displays *n first Fibonacci numbers*. Print all the Fibonacci function calls.

Here is a sketch of the program (Fibonacci calls are not printed):

```
def Fibonacci(n):
   if n == 0:
      ...
   elif n == 1:
      ...
   else:
      ...

def main():
   n = eval(input("Please, input any non-negative
                                          integer:"))

   for i in range(n):
      print(Fibonacci(i))
```

next slide →

Homework Assignment

Similar to example with Factorial Function (recursive definition) (slides 6 – 13) draw a figure of Fibonacci Function call on n=5

Suggestion: it is probably worth doing it as a tree: