CSI 32

Lecture 21

Chapter 11 Recursion

11.3 Functional Recursion

11.4 Binary Search

11.3 Functional Recursion

From the previous lecture:

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 definition*)

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

From the previous lecture:

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*)

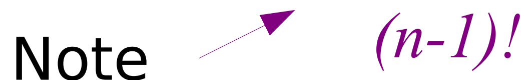
$F(1)=1$ (*base case*)

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

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  $(n-1)!$

11.3 Functional Recursion

From the previous lecture:

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*)

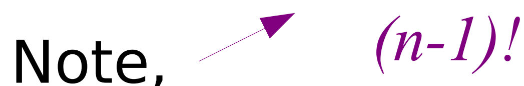
$F(1)=1$ (*base case*)

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

Example 2: Factorial:

Recall that usually we have this definition:

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

Note,  $(n-1)!$

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

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

11.3 Functional Recursion

Let's see the program that finds factorial of a number, using recursive definition:

$1! = 1$

$n! = n*(n-1)!$

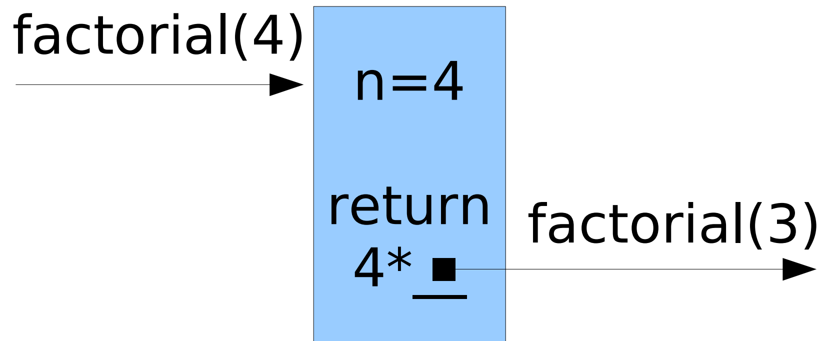
Here is a draft of the program:

```
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):

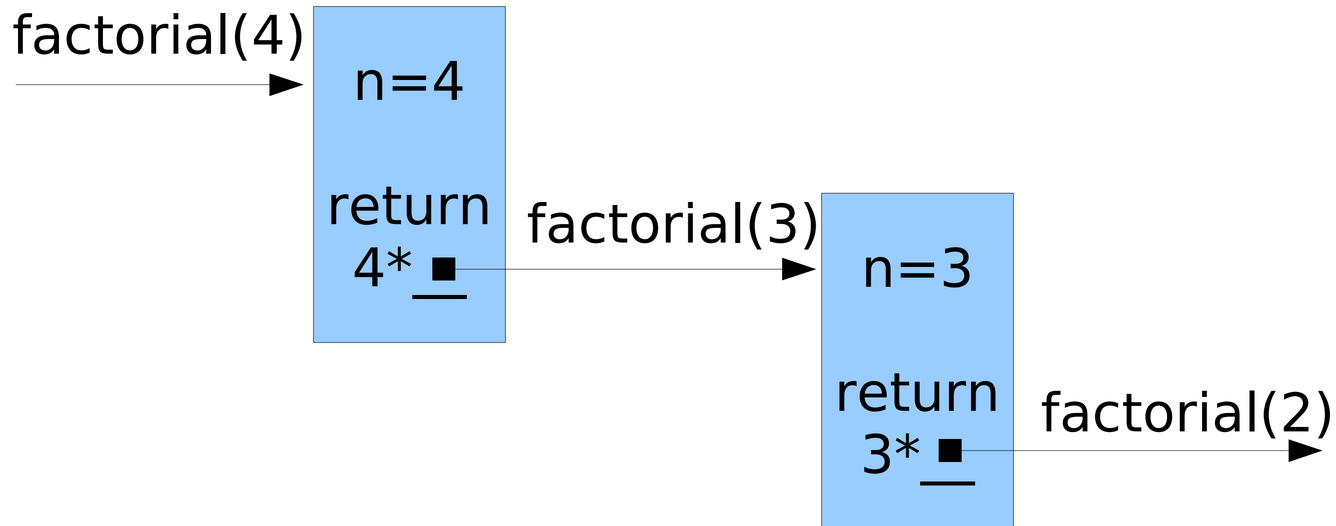
```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
```



11.3 Functional Recursion

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

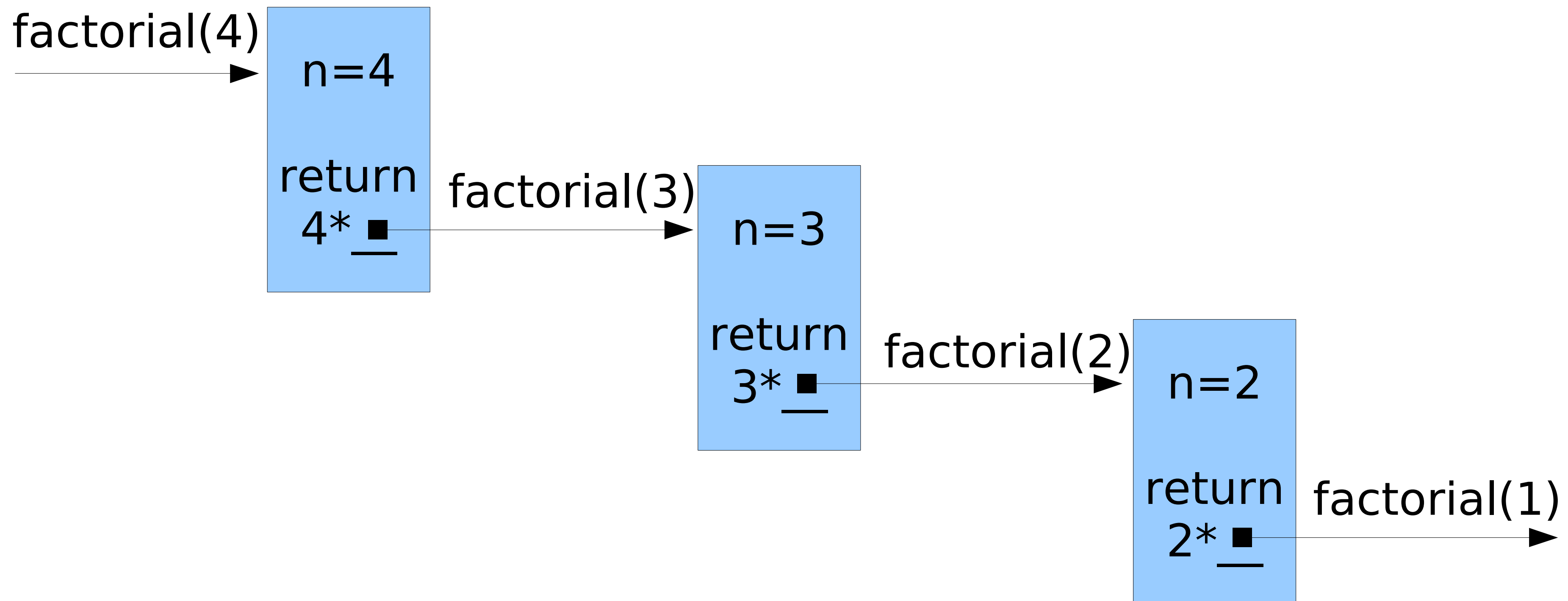
```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
```



11.3 Functional Recursion

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

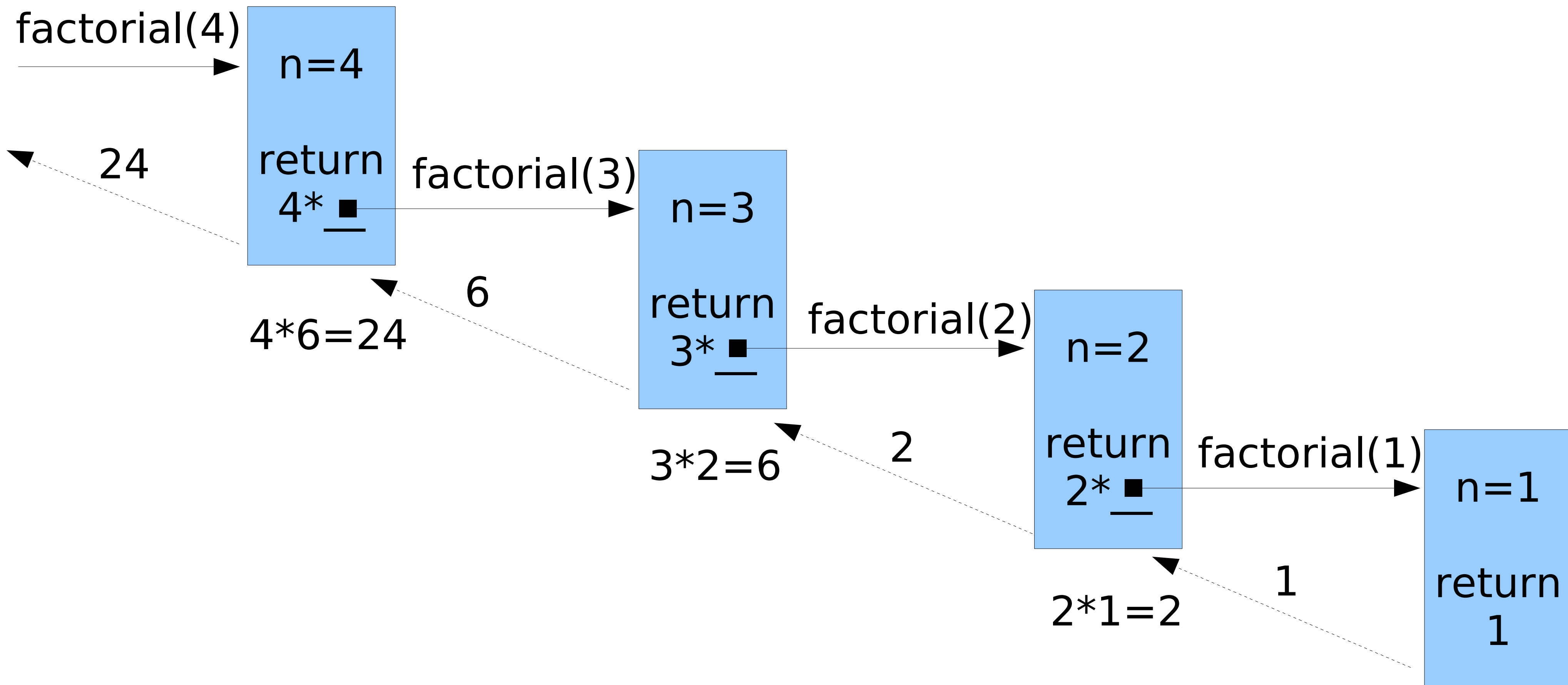
```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
```



11.3 Functional Recursion

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

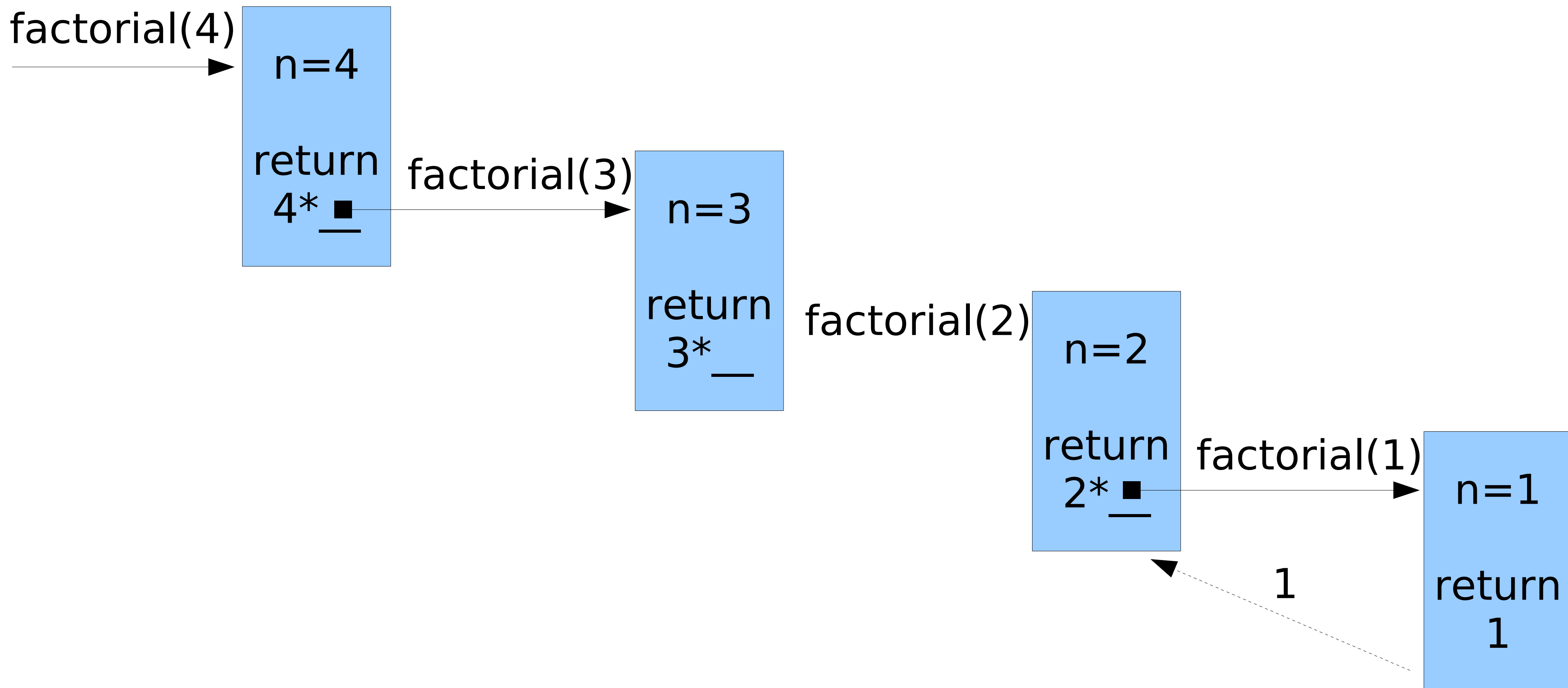
```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
```



11.3 Functional Recursion

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

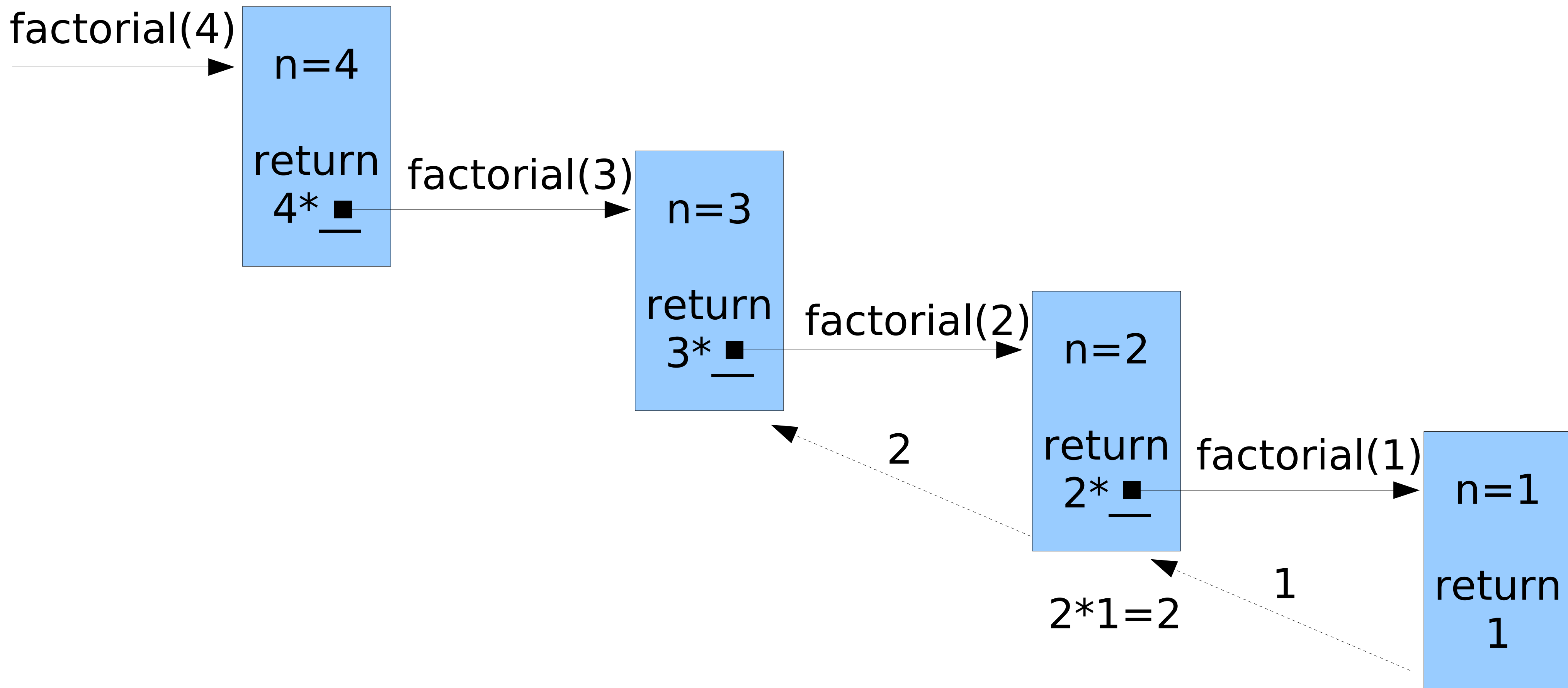
```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
```



11.3 Functional Recursion

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

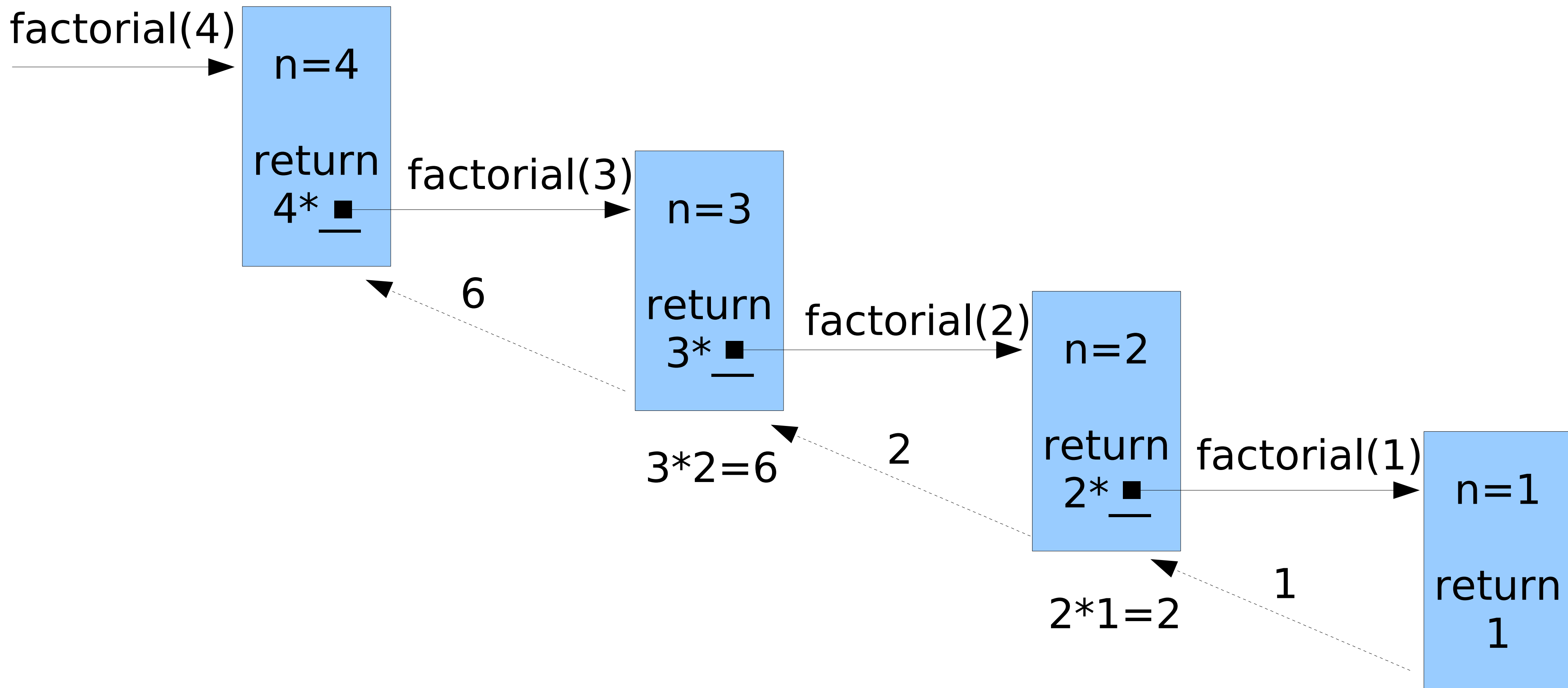
```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
```



11.3 Functional Recursion

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

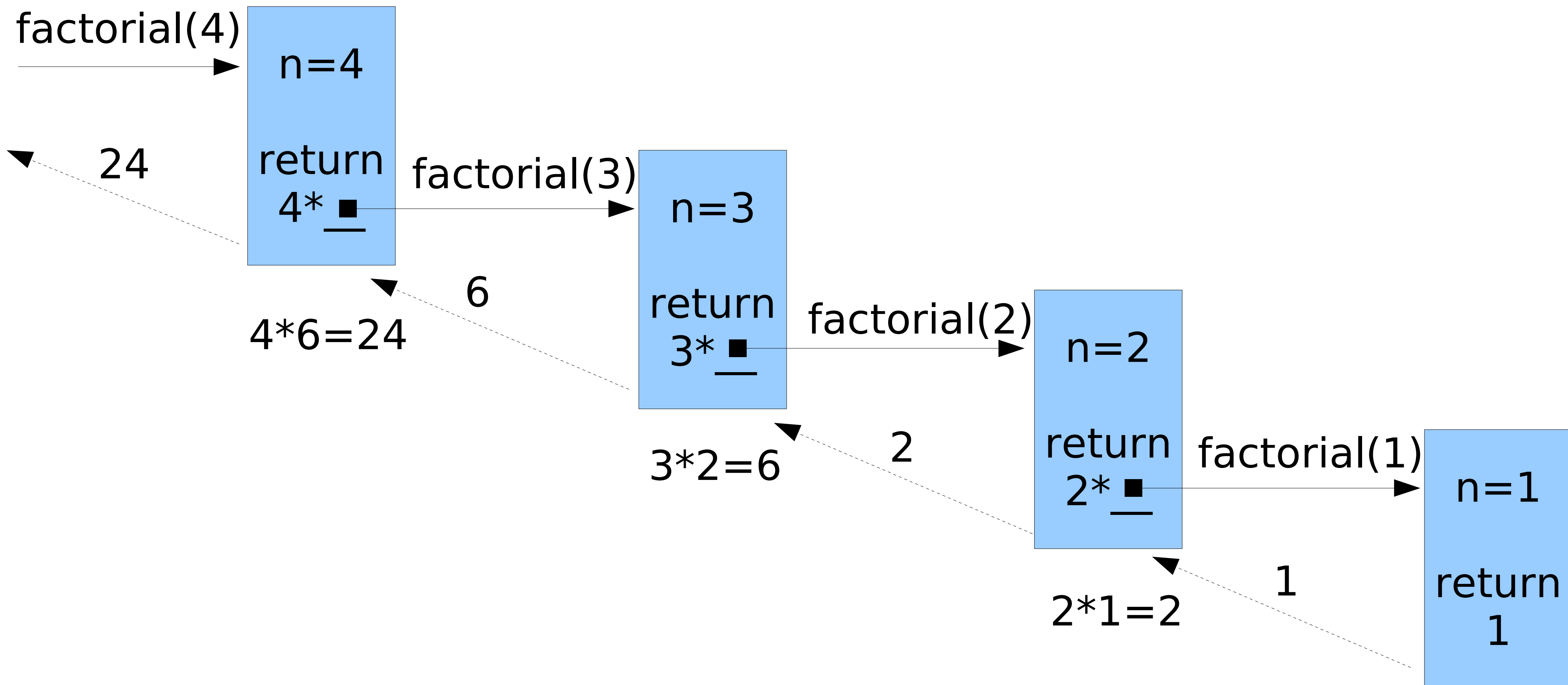
```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
```



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) = 6 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,A(1,1))=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 is an algorithm for locating 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 $log_2 n$ steps.

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)

Example of the binary search algorithm on the following input:

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

Number of elements if the array: 11
Middle element: 6th
19 > 7   True

therefore take the left part

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)

Example of the binary search algorithm on the following input:

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

Number of elements if the array: 5
Middle element: 3rd
9 > 7 True

therefore take the left part

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)

Example of the binary search algorithm on the following input:

[1,7]   find: 7

Number of elements if the array: 2
Middle element: 1st
1 > 7 False

therefore take the right part

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)

Example of the binary search algorithm on the following input:

[7]   find: 7

Number of elements if the array: 1
7=7

Found it!
Return the index of 7 (in the original list)

11.4 Binary Search

See the program binary_search.py