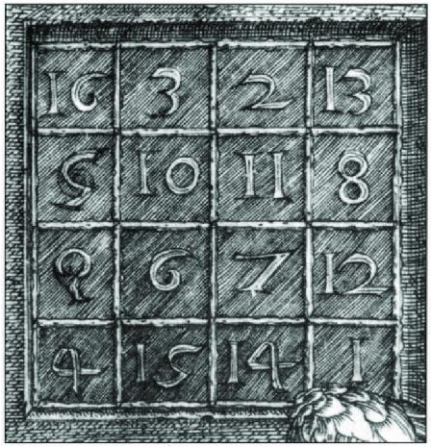
Logic and Discrete Structures -LDS



Course 3

Lecturer Dr. Eng. Cătălin Iapă

e- mail: catalin.iapa@cs.upt.ro

Facebook : Catalin Iapa

CV: Catalin Iapa

Function summary

Functions express calculations in programming.

The definition fields and values correspond to *types* in programming.

In functional languages , functions can be manipulated like any *values* . Functions can be *arguments* and *results* of functions .

Multi-argument (or tuple) functions can be rewritten as single-argument functions that return functions .

What do we know so far?/ What should we know?

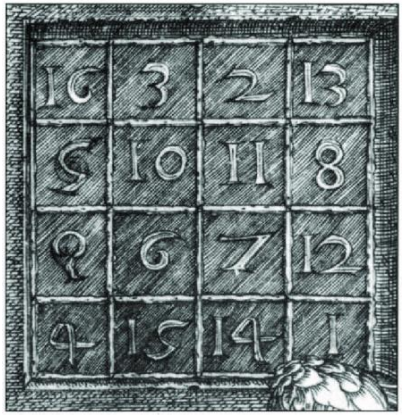
We know *the properties of functions* and how to *use them:* injective, surjective, bijective, invertible functions;

To *build* functions with certain properties;

To *count* functions defined on finite sets (with given properties);

To *compose simple* functions to solve problems;

To identify the type *of* a function.



Inductively defined sets

Recursion

Recursive functions in PYTHON

Pattern matching

Tail recursion

Advantages and disadvantages of recursion

Inductively defined sets

Let the set $A = \{3 , 5 , 7 , 9 , \ldots \}$

We can define it $A = \{ x \mid x = 2k + 3 , k \in \mathbb{N}\}$

Alternatively

- $3 \in A$
- $x \in A \Rightarrow x + 2 \in A$
- an element reaches A only through one of *the* above steps

$\Rightarrow$ we can define *inductively* the set A

Inductively defined sets

$A = \{3, 5, 7, 9, 11, 13, \ldots\}$

$3 \in A$ – the *basic element* : $P(0) : a_0 \in A$

$x \in A \Rightarrow x + 2 \in A$ – *construction of new elements* :
$P(k) \Rightarrow P(k+1) : a_k \in A \Rightarrow a_{k+1} \in A$

an element gets into A *only* through one of the above steps – *closure* (no other element is in the set)

$\Rightarrow$ *the inductive* definition of *A*
$\Rightarrow$ we say that *A* is an *inductive set*

Inductively defined sets

An inductive definition of a set *S* consists from:

- *base*: the *base elements* of *S* (minimum one).
- *induction*: at least one *rule for constructing* new elements of S from elements already existing in S
- *closure* : *S* contains *only the elements* obtained by base and induction steps

The base elements and the rules for constructing new elements constitute *the constructors* of the set.

Inductively defined sets - example

The set of natural numbers N is an inductive set:

- *basis* : $0 \in$
- *induction* : $n \in \mathrm{N} \Rightarrow n + 1 \in \mathrm{N}$

Constructors of N:

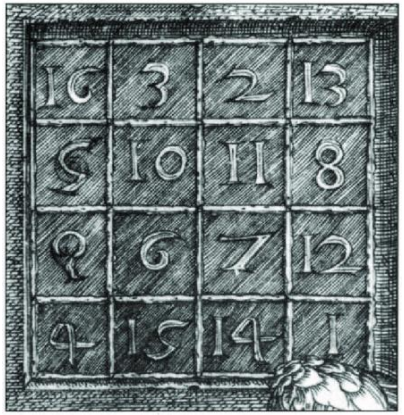
- base 0
- the addition operation by 1

Inductively defined sets - example

$A = \{1 , 3 , 7 , 15 , 31 , \ldots \}$ is an inductive set :

- *base* : $1 \in A$
- *induction* : $x \in A \Rightarrow 2x + 1 \in A$

- Constructors of A :
  - base 1
  - The operation of multiplying by 2 and adding by 1



Inductively defined sets

Recursion

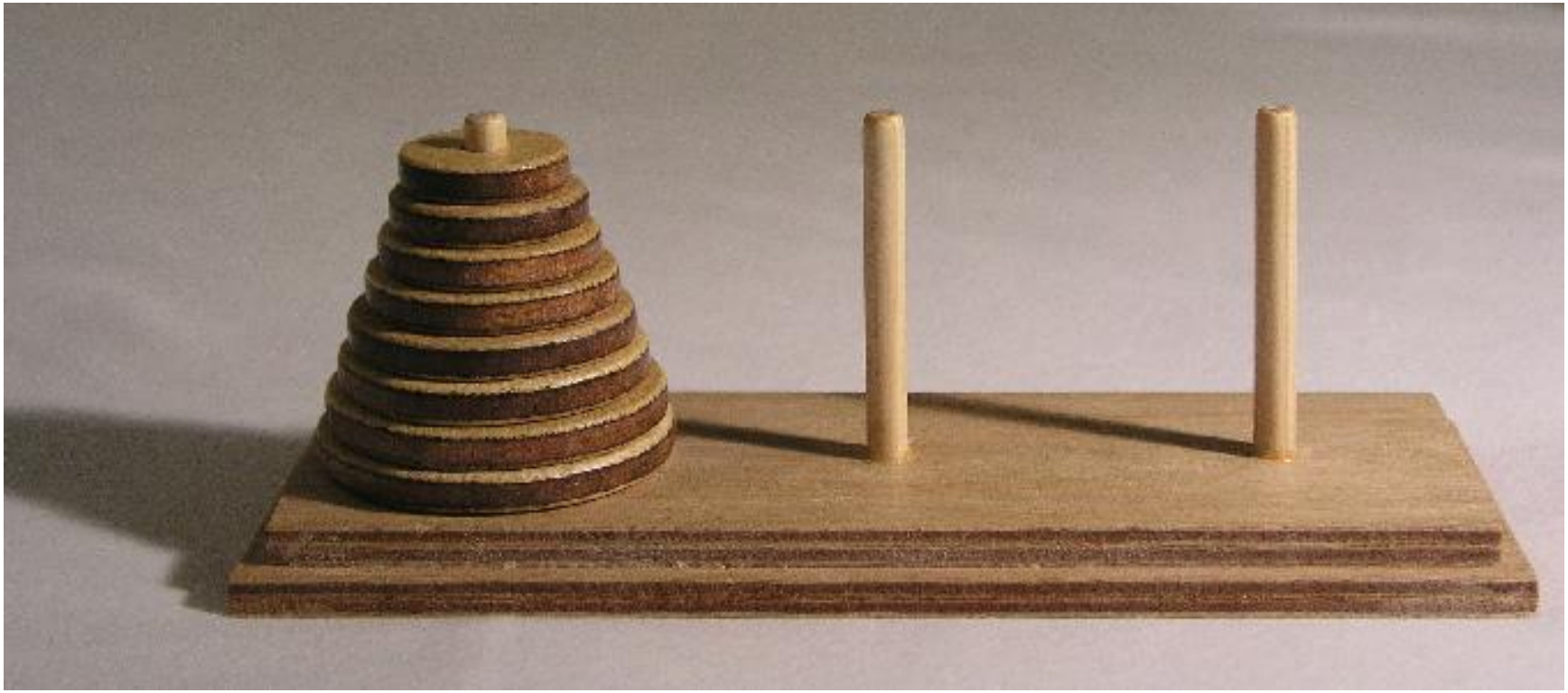
Recursive functions in PYTHON

Pattern matching

Tail recursion

Advantages and disadvantages of recursion

Towers of Hanoi



https://upload.wikimedia.org/wikipedia/commons/thumb/0/07/Tower_of_Hanoi.jpeg/450px-Tower_of_Hanoi.jpeg

Towers of Hanoi

The aim of the game is to move the whole stack from one rod to another , respecting the following rules:

- Only one disk can be moved at a time.
- Each move consists of taking the topmost disc on a rod and sliding it onto another rod, even over other discs already on that rod.
- A larger disc cannot be positioned on top to a smaller disc.

Towers of Hanoi

We need to find the minimum number of moves of the entire stack from one rod to another, based on the initial number of disks

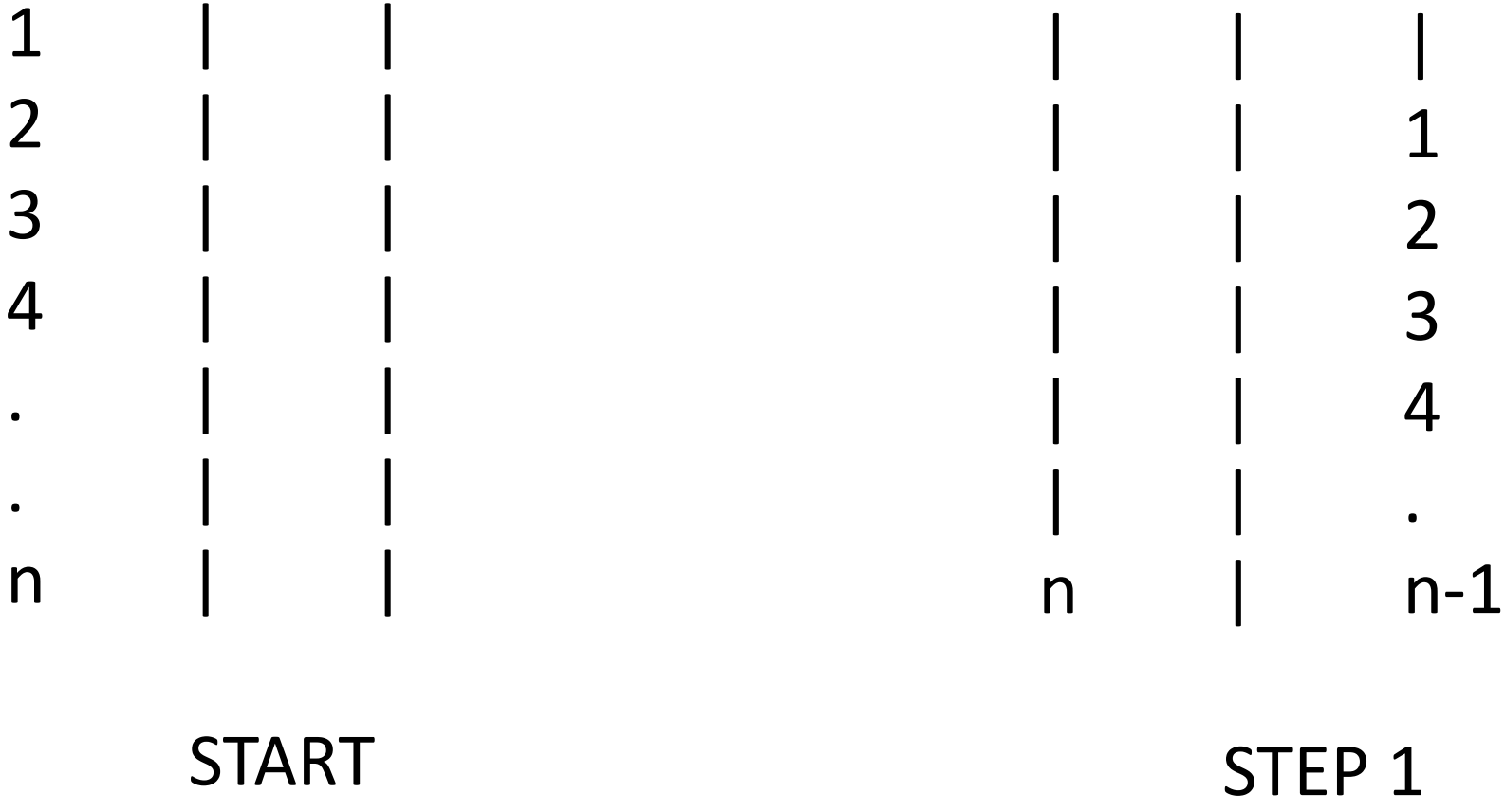
p(n)

p(1) = 1

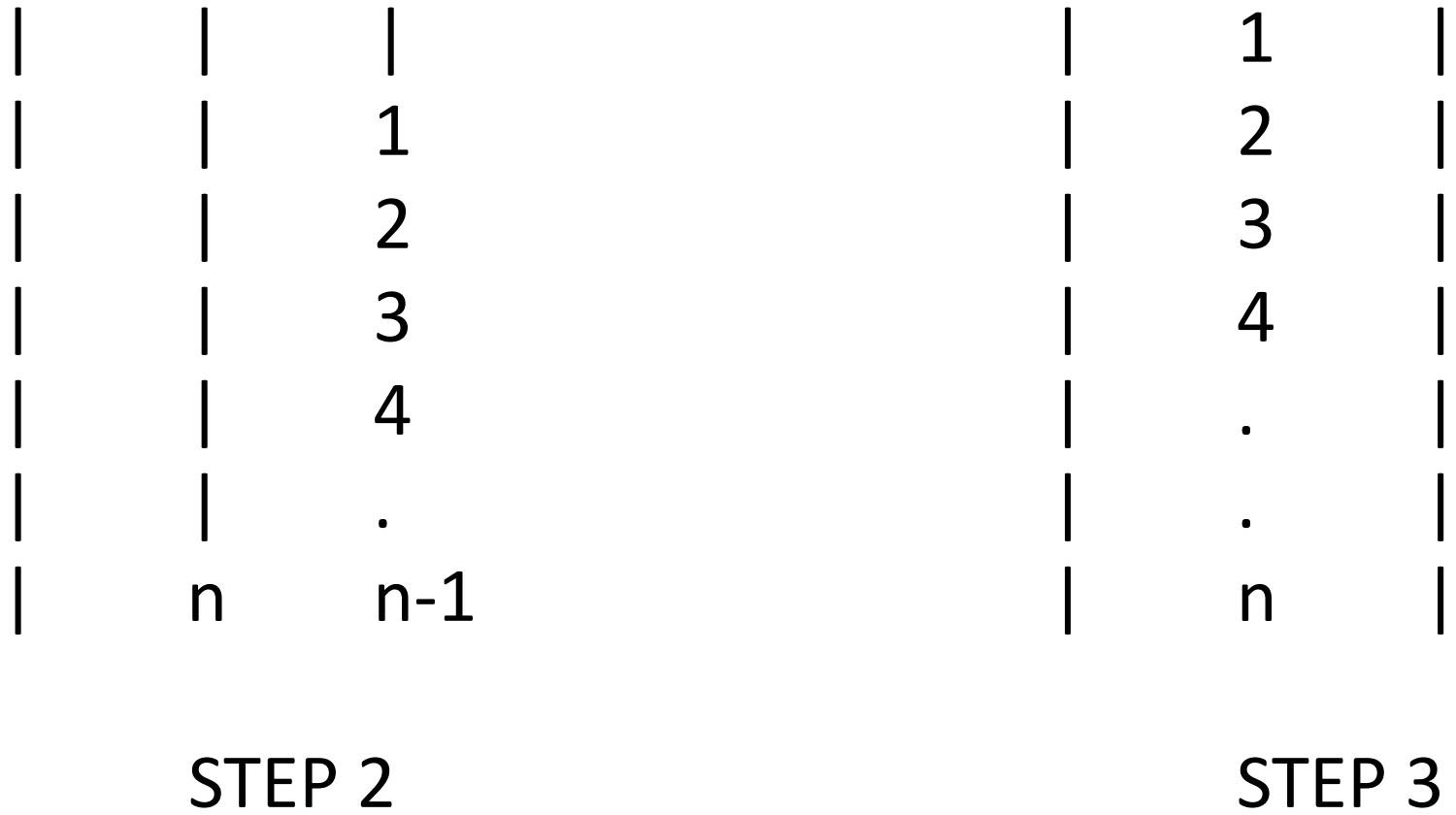
p(2) = 3

P(3) =

Towers of Hanoi



Towers of Hanoi



Towers of Hanoi

Step 1 – move n-1 disks

Step 2 – move the largest disk (1 disk)

Step 3 – move n-1 disks

p(n) = p(n-1) + 1 + p(n-1)

p(n) = 2 * p(n-1) + 1

$p(n)=2^n - 1$

(We can demonstrate through mathematical induction)

The problem of the number of teachers

At the faculty, we have a challenge with the number of teaching staff. We need more *teachers* because the number of students keeps increasing.

We have a rule that leads to an increase in the number of teachers:

- Every year, a teacher must bring/train *a new teacher*
- The only exception is *the first year* for each teacher, the year in which they do not have to bring/train a teacher

How many teachers will the faculty have after 8 years if we apply these rules and, to simplify the calculation, in year 1 we start with 1 teacher?

The problem of the number of teachers

We define the function:

f(n) = #number of teachers after year n

f(0) = 0
f(1) = 1
f(2) = ? ... 1
f(3) = ? ... 2
f(4) = ? ... 3
f(5) = ? ... 5
f(6) = ? ... 8

The problem of the number of teachers

Year 1: 1 teacher

Year 2: 1 teacher - only last year's teacher

Year 3: 2 teachers – the one from last year + 1 new

Year 4: 3 teachers – the 2 from last year + 1 new

Year 5: 5 teachers – the 3 from last year + 2 new ones

Year 6: 8 teachers – the 5 from last year + 5 new ones

Year 7: 13 teachers – the 8 from last year + 5 new ones

Year 8: 21 teachers – the 13 from last year + 8 new ones

The problem of the number of teachers

f(n) = #number of teachers after year n

- f(n+1) = f(n) + f(n-1)

  Last year's teachers | new teachers

Do you recognize this recurrence?

It's the way to build the famous

Fibonacci string

The problem of the number of teachers

The Fibonacci sequence is the first recurrence known to have been studied mathematically, of all the recurrences studied

It was first published in the journal *"Liber abaci"* by *Leonardo Fibonacci from Pisa* in *1202* . This treatise contained all that was known about mathematics at the time and influenced the development of mathematics for years to come

He studied a real problem of those times, the growth of the rabbit population, which he expressed as follows:

- each month, a pair of rabbits will give birth to an average of 2 more rabbits, except for the first month of life

The Fibonacci sequence

We reduce the problem of the number of teachers mathematically to solving *the equation* :

f(n+1) = f(n) + f(n-1), n≥2 with f(0) = 0 and f(1) = 1

We assume that the string has the form

$$f(n+1) = \lambda^{n+1}, where\ \lambda\ is\ a\ float$$

The equation becomes:

$$\lambda^{n+1} = \lambda^{n} + \lambda^{n-1}$$

$$\lambda^{n+1} - \lambda^{n} - \lambda^{n-1} = 0$$

$$\begin{cases} \lambda^{n-1}(\lambda^2 - \lambda^1 - 1) = 0 \\ f(n) \neq 0, (\forall\ n \in N^*) \end{cases} \Rightarrow \lambda^2 - \lambda - 1 = 0$$

The Fibonacci sequence

Equation of degree 2:

$\lambda^2 - \lambda^1 - 1 = 0$ has the solutions:

$$\lambda_1 = \frac{1+\sqrt{5}}{2}$$

$$\lambda_2 = \frac{1-\sqrt{5}}{2}$$

equation $\lambda^2 - \lambda^1 - 1 = 0$ is called the *associated characteristic equation*, and if it has 2 distinct solutions, then the general solution of the equation from which we started is:

$$f(n+1) = c_1 \left(\frac{1+\sqrt{5}}{2}\right)^{n+1} + c_2 \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}$$

The Fibonacci sequence

We also know that *f(0)= 0* and *f(1) = 1*

$$\begin{cases} f(0) = c_1\left(\frac{1+\sqrt{5}}{2}\right)^0 + c_2\left(\frac{1-\sqrt{5}}{2}\right)^0 = c_1 + c_0 = 0 \\ f(1) = c_1\frac{1+\sqrt{5}}{2} + c_2\frac{1-\sqrt{5}}{2} = 1 \end{cases}$$

With the solutions: $c_1 = \frac{1}{\sqrt{5}}$ and $c_2 = -\frac{1}{\sqrt{5}}$

The Fibonacci sequence

Finally, we proved above that *the n term of the Fibonacci sequence* has the form:

$$f(n) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

This is not something easy to calculate, it took Europeans 6 centuries to find this solution.

The Fibonacci sequence

The truth is that Fibonacci did not discover this string, he only *told Europeans about it* , being used around *200* by *Indian mathematicians* and having applications in grammar and music

Kepler used it in the 16th century to study how *the leaves of a flower are arranged on the stem* , what is the number of leaves on each level

Mathematician *Abraham de Moivre* was the one who discovered *Binet's formula* , in the 17th century, the formula on the previous slide

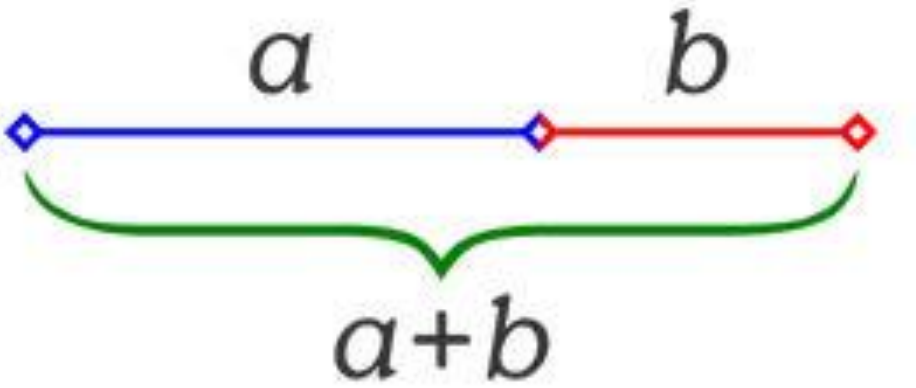
The Fibonacci sequence

Binet's formula connects the terms of the Fibonacci sequence and the power of *the golden number* (golden ratio or golden section), the first irrational number discovered and defined in history

$\varphi = \frac{1+\sqrt{5}}{2} = 1.618033 \ldots$

Euclid defined it first times using the ratio:

$$\frac{a+b}{a} = \frac{a}{b} = \varphi$$



https://ro.wikipedia.org/wiki/Sec%C8%9Biunea_de_aur#/media/Fi%C8%99ier:Numarul_de_aur.jpg

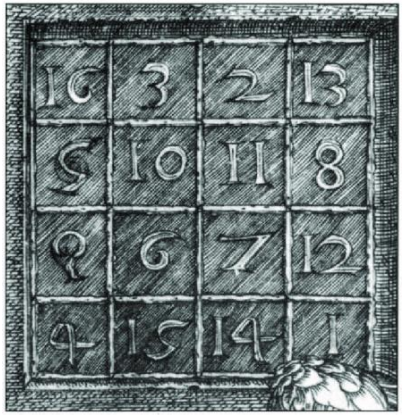
Linear recurrence

Definition: A *recurrence is linear* if it has the form

$$f(n) = a_1 f(n\text{-}1) + a_2 f(n\text{-}2) + \ldots a_d f(n\text{-}d) = \sum_{i=1}^{d} a_i f(n-i), \text{ with fixed numbers } a_i \text{ and d}$$

d is called the order of recurrence

What order does the Fibonacci sequence recurrence?



Inductively defined sets

Recursion

Recursive functions in PYTHON

Pattern matching

Tail recursion

Advantages and disadvantages of recursion

Recursion in computer science

A notion is *recursive* if it is used *in its own definition*.

Recursion is fundamental in computer science:

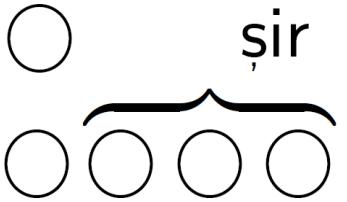
- if a problem has a solution , *it can be solved recursively*
- reducing the problem to a simpler case of *the same problem*

By understanding recursion, we can solve any feasible problem

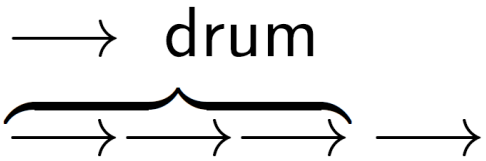
Recursion: examples

Recursion reduces a problem to a simpler case of the same problem

a *string* is { an element / an element followed by a string



a *road* is { a step / *a road followed by a step*



Recurring strings

Arithmetic progression:

$$\begin{cases} x_0 = b & (x_n = b \text{ if } n = 0) \\ x_n = x_{n-1} + r & \text{if } n > 0 \end{cases}$$

Example: 1 , 4 , 7 , 10 , 13 , . . . ($b = 1, r = 3$)

Geometric progression:

$$\begin{cases} x_0 = b & (x_n = b \text{ if } n = 0) \\ x_n = x_{n-1} * \mathrm{r} & \text{if } n > 0 \end{cases}$$

Example: 3 , 6 , 12 , 24 , 48 , . . . ($b = 3, r = 2$)

The definitions above do not compute x_n *directly,* but *from close to close*, depending on x_{n-1}

The elements of a recursive definition

1. *The base case* is the simplest case for the given definition, defined directly (the initial term in a recurring string: x_0)

The base case must not be missing

2. *Recurrence relation* - defines the notion, using a simpler case of the same notion

3. Proof of stopping recursion after a finite number of steps

Recursive functions

A function is *recursive* if it appears in its own definition.

A function f is defined recursively if it exists at least one value $f(x)$ *defined in terms of another value* $f(y)$, where $x \neq y$.

Recursive functions over inductive sets

Many recursive functions have as domain *an inductive set*

If S is an inductive set, we can use its constructors to define a recursive function f with domain S:

- *base* : for each basic element x € S we specify a value f(x)
- *Induction* : we give one or more rules that for any x € S, inductively defined x, that define f(x) in terms of some other previously defined values of f

Recursive functions in PYTHON

The generic form of a recursive function:

```
def recursive_function ():
        ...
        recursive_function()
        ...

recursive_function()
```

Recursive functions in PYTHON

Example:

```
def frec ():
        x=7
        frec ()

frec()
```

- *At each call of the function f, new, distinct memory space is allocated for x*
- In the above example it is wrong that the stop execution condition does not appear

Recursive functions in PYTHON

Example: Display on the screen in descending order all natural numbers less than n.

```
def count (n ):
    print (n)
    if ( n > 0 ):
        count (n - 1 )
count (3)
```

OUTPUTS :

3

2

1

0

Recursive functions in PYTHON

Example: Factorial function

$n! = n \times (n-1) \times (n-2) \times (n-3) \cdots \times 3 \times 2 \times 1$

We can rewrite it recursively:

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

Recursive functions in PYTHON

We detail *how* the factorial function is calculated recursively:

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

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

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

.

.

$n! = n \times (n-1) \times (n-2) \times (n-3) \cdots \times 3!$

$n! = n \times (n-1) \times (n-2) \times (n-3) \cdots \times 3 \times 2!$

$n! = n \times (n-1) \times (n-2) \times (n-3) \cdots \times 3 \times 2 \times 1!$

Recursive functions in PYTHON

Mathematically, *the recursive form* of the factorial function is:

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

Recursive functions in PYTHON

In PYTHON we can write the factorial function like this:

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

print(factorial(4))
```

OUTPUTS:

24

Recursive functions in PYTHON

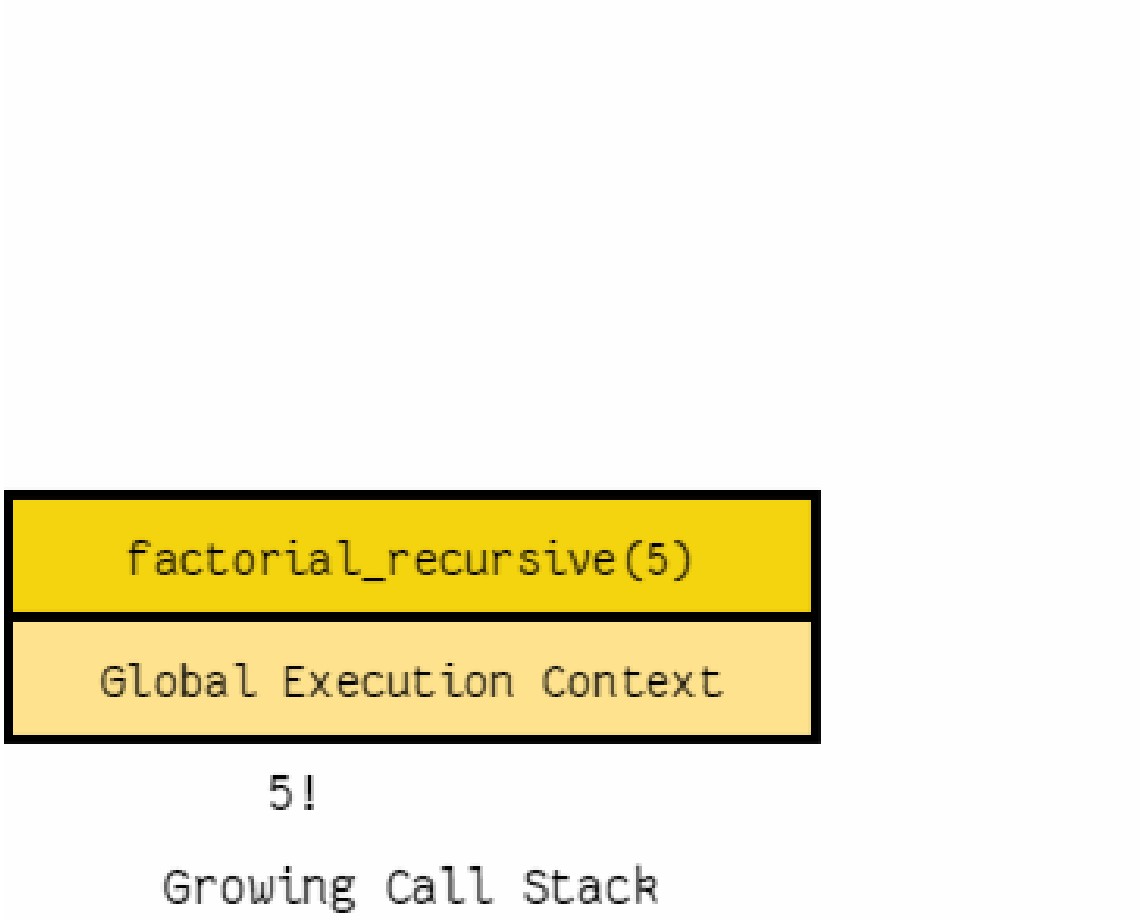
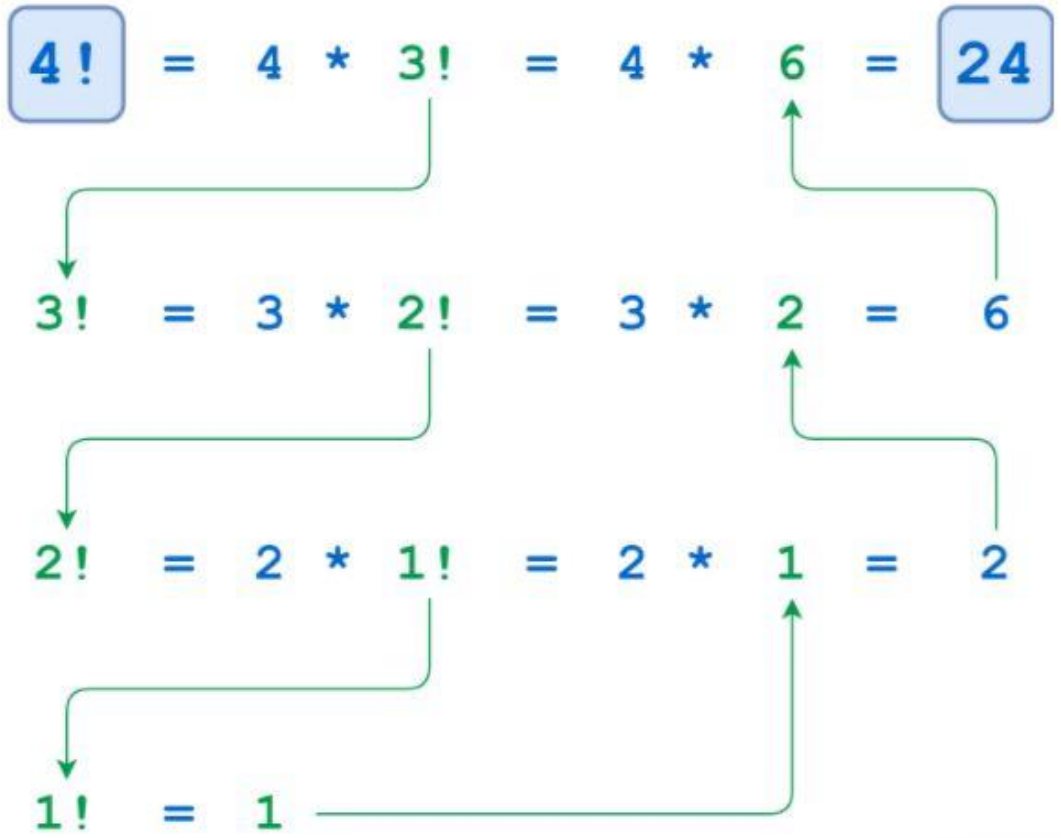


Image : https://files.realpython.com/media/stack.9c4ba62929cf.gif

Recursive functions in PYTHON

The steps the program takes to calculate:



https://files.realpython.com/media/jsturtz-factorial-example.496c01139673.png

Recursive functions in PYTHON

- How the factorial recursive function works:

factorial (4) will execute the following steps:

factorial(4) = factorial(3) * 4 – the function remains in execution
factorial(3) = factorial(2) * 3 – the function remains in execution
factorial(2) = factorial(1) * 2 – the function remains in execution
factorial(1) = 1
factorial(2) = factorial(1) * 2 = 1 * 2 = 2
factorial(3) = factorial(2) * 3 = 2 * 3 = 6
factorial(4) = factorial(3) * 4 = 6 * 4 = 24

An unsolved problem : "problem 3 · n + 1"

Collatz conjecture(1937),

Let be a positive number n:

- if it is even, we divide it by 2: *n/* 2
- if it is odd, we multiply it by 3 and add 1: 3 · *n* + 1

Is 1 reached from any positive number?

(unsolved problem in math...)

$$f(n)=\begin{cases} n/2, & if\ n\ is\ even \\ 3\cdot n+1, & if\ n\ is\ odd \end{cases}$$

Examples:

3 →10→5→16→8→4→2→ 1

11→34→17→52→26→13→40→20→10→5→16→8→4→2→ 1

An unsolved problem : "problem 3 · n + 1"

How many steps does it take to get to #1?

- We define the function $p : \mathbb{N}* \rightarrow \mathbb{N}$ that counts the steps until stopping:
  - for 3→ 10→5→16→8→4→2→1 we have 7 steps
- We do not have a formula with which to calculate p(n) directly.
- But if the string n , $f(n)$, $f(f(n))$, . . . reaches 1, then the number of steps taken **from** n is **one more** than continuing **from** $f(n)$
- $$p(n) = \begin{cases} 0, & if\ n = 1 \\ 1 + p(f(n)), & if\ n > 1 \end{cases}$$

An unsolved problem : "problem 3 · n + 1"

How many steps does it take to get to #1?

- We define the function p : N∗ → N that counts the steps until stopping:
  - for 3→ 10→5→16→8→4→2→1 we have 7 steps
- We do not have a formula with which to calculate p(n) directly.
- But if the string n , f (n), f (f (n)) , . . . reaches 1, then the number of steps taken **from** n is **one more** than continuing **from** f (n)
- $$p(n) = \begin{cases} 0, & if\ n = 1 \\ 1 + p(f(n)), & if\ n > 1 \end{cases}$$
- The function p is used in its own definition, so it is recursively defined.

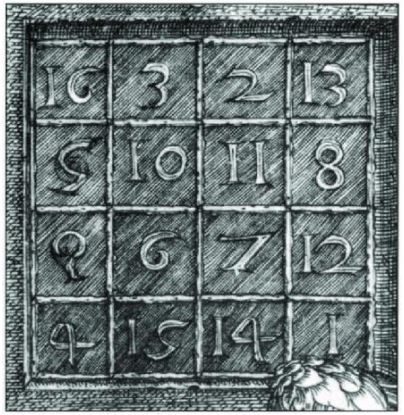
An unsolved problem : "problem 3 · n + 1"

```
def next(n):
    if (n%2 == 0):
        return n/2
    else :
        return 3 * n + 1

def steps (n):
    if (n == 1):
          return 0
    else :
        return 1 + steps ( next (n))
```

Fibonacci string in PYTHON

```
def fibonacci (n):
    if (n ==0):
        return 0
    elif (n==1):
        return 1
    else:
        return fibonacci (n-1) + fibonacci (n-2)

print( fibonacci (7))
```

OUTPUT: 21



Inductively defined sets

Recursion

Recursive functions in PYTHON

Pattern matching

Tail recursion

Advantages and disadvantages of recursion

Pattern matching

We can also write the function this way, using pattern matching:

```
def fibonacci(n):
    match n:
        case 0:
            return 1
        case 1:
            return 1
        case _:
            return fibonacci(n-1) + fibonacci(n-2)
```

Pattern matching

We can write *the last condition in these ways, they are equivalent:*

```
case _:
        return fibonacci(n-1) + fibonacci(n-2)
```

or

```
case other:
        return fibonacci(n-1) + fibonacci(n-2)
```

or

```
case n:
        return fibonacci(n-1) + fibonacci(n-2)
```

Pattern matching

How *match case is executed* :

It is checked, one by one, *from the first* case to the last, if the value matches.

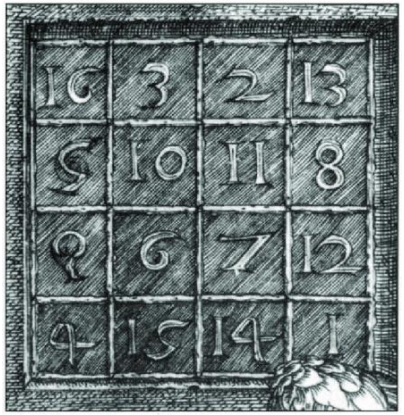
- if *it doesn't match,* go to the next case

- if *it matches,* the instruction from the respective case is executed, and the rest of the cases are no longer checked

Pattern matching

Example : To display if *a point is on the* axes :

```
def points (x, y):
    match (x, y):
        case (0, 0):
            print(“The point is at origin")
        case (0, _):
            print("The point is on the Ox axis")
        case (_, 0):
            print(“The point is on the Oy axis")
        case (_, _):
            print("The point is not on any axis")
puncte(0,3)     #The point is on the Ox axis
```



Inductively defined sets

Recursion

Recursive functions in PYTHON

Pattern matching

Tail recursion

Advantages and disadvantages of recursion

Limitation in PYTHON

One of the disadvantages of recursion is that each function call that remains in execution uses *memory space on the stack*

PYTHON by default *limits* the number of calls of the same expression to 1000 (10**3) times

The error that occurs when calling the same expression more than 1000 times generates the error: *maximum recursion depth exceeded error*

In problems where we need more than 1000 iterations we can change this limit using the *setrecursionlimit () function* from the *sys module*

Limitation in PYTHON

To *increase the limit* from a maximum of 1,000 calls to a maximum of 100,000 calls :

```
import sys
sys.setrecursionlimit (10 **5 )
```

Tail recursion

Factorial, *tail recursion* :

```
def fact(n, a=1):
    if (n <= 1):
        return a
    else:
        return fact(n - 1, n * a)

print(factorial(4))
```

Factorial, classical recursion :

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

print(factorial(4))
```

Tail recursion

- How the factorial recursive function works:

factorial (4) will execute the following steps:

factorial(4, 1) = factorial(3, 4 * 1)
factorial(3, 4) = factorial(2, 3 * 4)
factorial(2, 12) = factorial(1, 12 * 2)
factorial(1, 24) = 24
factorial(2, 12) = 24
factorial(3, 4) = 24
factorial(4, 1) = 24

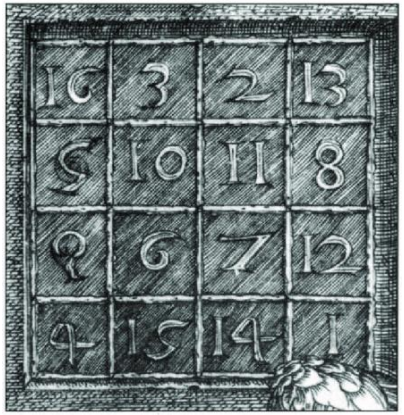
Example of using if - else

PYTHON also allows a more compact writing of the if-else statement like this:

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

print(factorial(4))
```

OUTPUTS:

24



Inductively defined sets

Recursion

Recursive functions in PYTHON

Pattern matching

Tail recursion

Advantages and disadvantages of recursion

Advantages of recursion in programming

- The code is *more short* and easy to follow, elegant, clean
- Complex problems can be broken down into *simpler subproblems* and thus easier to solve
- *Generating strings* is done more simply recursively

Disadvantages of recursion in programming

- It is harder to follow step by step *the logic behind* a code written recursively
- Repeated recursive calls use *a lot of memory*
- Errors that occur in recursive functions are *more difficult to correct*

To know

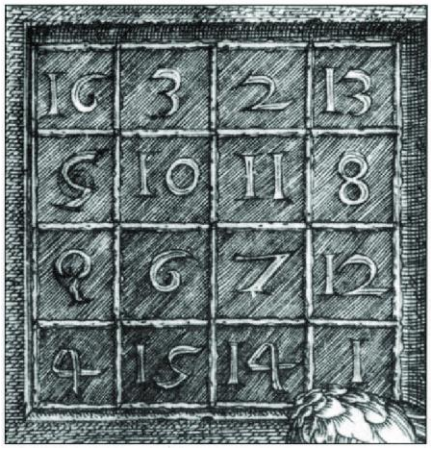
To recognize and define recursive notions

To recognize if a recursive definition is correct

- has the base case? does recursion stop?

To solve problems by writing recursive functions

- the base case + the reduction step to a simpler problem



Thank you!

Bibliography

- The full math induction game was inspired by ***the Mathemati cs for Computer Science course*** from Massachusetts Institute of Technology (from https://ocw.mit.edu /)
- The content of the course is mainly based on the materials of the past years from the LSD course, taught by Prof. Dr. Marius Minea et al. Dr. Eng. Casandra Holotescu (http://staff.cs.upt.ro/~marius/curs/lsd/index.html)