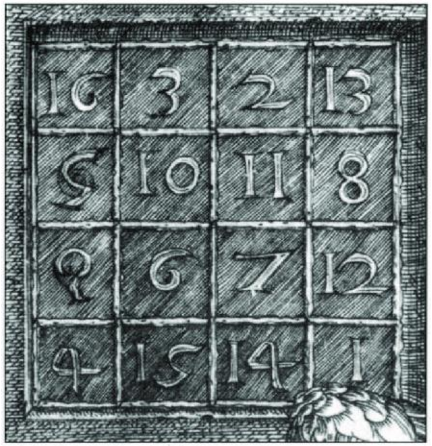
Logică și Structuri Discrete -LSD



Cursul 3

dr. ing. Cătălin Iapă
e-mail: catalin.iapa@cs.upt.ro
facebook: Catalin Iapa
cv: Catalin Iapa

Rezumat funcții

Prin *funcții* exprimăm calcule în programare.

Domeniile de definiție și valori corespund *tipurilor* din programare.

În limbajele funcționale, funcțiile pot fi manipulate ca orice *valori*. Funcțiile pot fi *argumente* și *rezultate* de funcții.

Funcțiile de mai multe argumente (sau de tuple) pot fi rescrise ca funcții de un singur argument care returnează funcții.

Ce știm până acum?/ Ce ar trebui să știm?

Știm *proprietățile funcțiilor* și cum să *ne folosim de ele:* funcții injective, surjective, bijective, inversabile;

Să *construim* funcții cu anumite proprietăți;

Să *numărăm* funcțiile definite pe mulțimi finite (cu proprietăți date);

Să *compunem* funcții simple pentru a rezolva probleme;

Să identificăm *tipul* unei funcții.

Mulțimi definite inductiv

Recursivitate

Funcții recursive în PYTHON

Potrivirea de tipare

Tail recursion

Avantajele și dezavantajele recursivității

Mulțimi definite inductiv

Fie mulțimea $A = \{3, 5, 7, 9, . . . \}$
O putem defini $A = \{x \mid x = 2k + 3, k \in \mathrm{N}\}$

Alternativ, observăm că:
- $3 \in A$
- $x \in A \Rightarrow x + 2 \in A$
- un element ajunge în A doar printr-unul din pașii de mai sus

$\Rightarrow$ putem defini *inductiv* mulțimea A

Mulțimi definite inductiv

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

$3 \in A$ – *elementul de bază*: $P(0) : a_0 \in A$

$x \in A \Rightarrow x + 2 \in A$ – *construcția de noi elemente*:
$P(k) \Rightarrow P(k + 1) : a_k \in A \Rightarrow a_{k+1} \in A$

un element ajunge în *A* doar printr-unul din pașii de mai sus – *închiderea* (niciun alt element nu e în mulțime)

⇒ definiția *inductivă* a lui *A*
⇒ spunem că *A* e o *mulțime inductivă*

Mulțimi definite inductiv

O definiție inductivă a unei mulțimi *S* constă din:

- *bază*: Enumerăm *elementele de bază* din *S* (minim unul).
- *inducția*: Dăm cel puțin o *regulă de construcție* de noi elemente din *S*, pornind de la elemente deja existente in *S*.
- *închiderea*: *S* conține *doar elementele* obținute prin pașii de bază și inducție (de obicei implicită).

Elementele de bază și regulile de construcție de noi elemente constituie *constructorii* mulțimii *S*.

Mulțimi definite inductiv - exemplu

Mulțimea numerelor naturale N e o mulțime inductivă:

- *bază*: 0 ∈ N
- *inducția*: $n \in N \Rightarrow n + 1 \in N$

Constructorii lui N:

- baza 0
- operația de adunare cu 1

Mulțimi definite inductiv - exemplu

$A = \{1, 3, 7, 15, 31, \ldots\}$ e o mulțime inductivă:

- *bază*: $1 \in A$
- *inducția*: $x \in A \Rightarrow 2x + 1 \in A$

- Constructorii lui A:
  - baza 1
  - operat, ia de înmulțire cu 2 și adunare cu 1

Mulțimi definite inductiv

Recursivitate

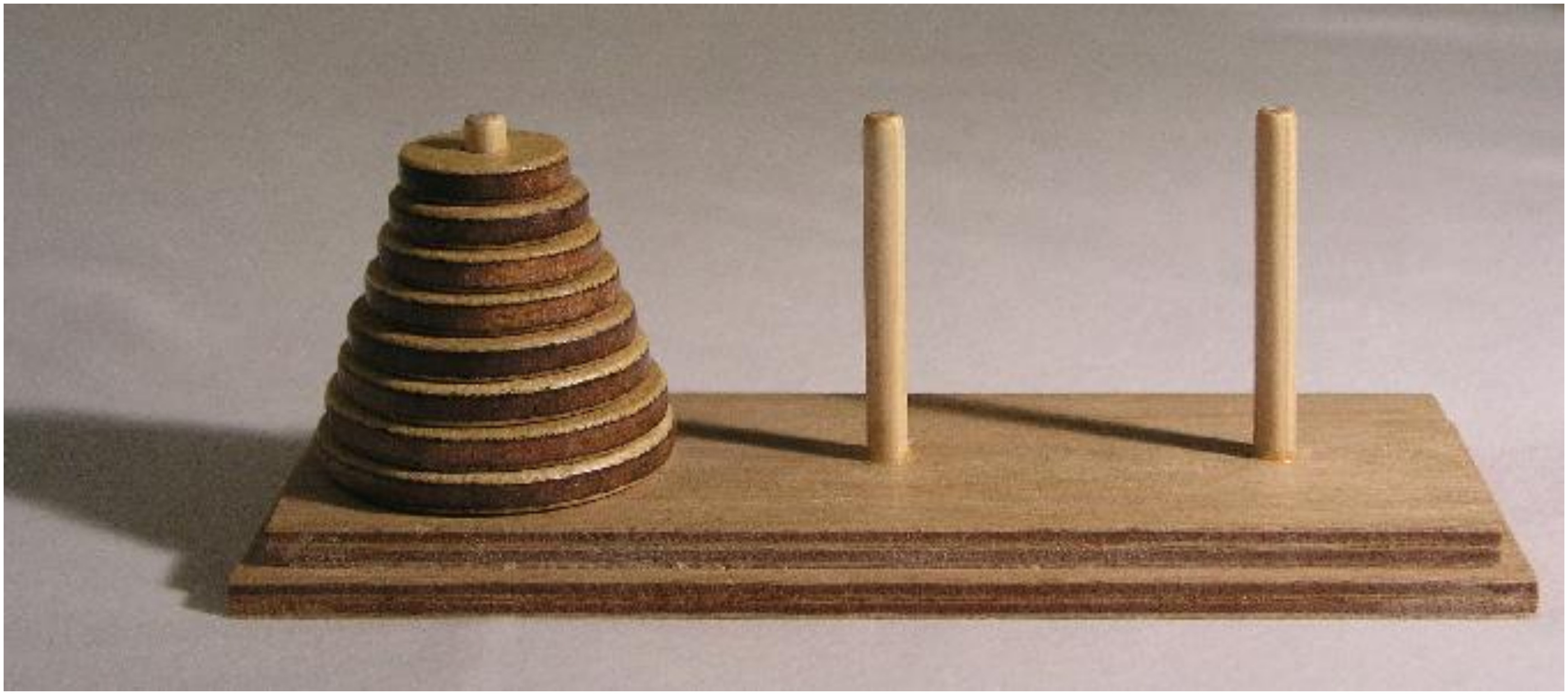
Funcții recursive în PYTHON

Potrivirea de tipare

Tail recursion

Avantajele și dezavantajele recursivității

Turnurile din Hanoi



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

Turnurile din Hanoi

Scopul jocului este acela de a muta întreaga stivă de pe o tijă pe alta, respectând următoarele reguli:

- Doar un singur disc poate fi mutat, la un moment dat.
- Fiecare mutare constă în luarea celui mai de sus disc de pe o tija și glisarea lui pe o altă tijă, chiar și deasupra altor discuri care sunt deja prezente pe acea tijă.
- Un disc mai mare nu poate fi poziționat deasupra unui disc mai mic.

Turnurile din Hanoi

Trebuie să găsim numărul minim de mutări a întregii stive de pe un disc pe altul în funcție de numărul de discuri inițial

p(n)

p(1) = 1

p(2) = 3

P(3) = 7

Putem găsii o regulă generală?

Turnurile din Hanoi

Pasul 1 – mutăm n-1 discuri
Pasul 2 – mutăm discul cel mai mare (1 disc)
Pasul 3 – mutăm n-1 discuri

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

p(n)= $2^n - 1$
(Putem demonstra prin inducție matematică)

Problema numărului de profesori

La facultate avem o provocare cu numărul de cadre didactice. Avem nevoie de mai mulți *profesori* pentru că numărul de studenți tot crește.

Avem o regulă care duce la creșterea numărului de profesori:

- În fiecare an, un profesor trebuie să aducă/ să formeze *un nou profesor*
- Excepție face doar *primul an* pentru fiecare profesor, an în care nu trebuie să aducă/ să formeze un profesor

Câți profesori o să aibă facultatea după 8 ani dacă aplicăm aceste reguli și, pentru a simplifica calculul, în anul 1 pornim de la 1 profesor?

Problema numărului de profesori

Definim funcția:

f(n) = #numărul de profesori după anul n

f(0) = 0

f(1) = 1

f(2) = ? ... 1

f(3) = ? ... 2

f(4) = ? ... 3

f(5) = ? ... 5

f(6) = ? ... 8

Problema numărului de profesori

Anul 1: 1 profesor

Anul 2: 1 profesor - doar profesorul de anul trecut

Anul 3: 2 profesori – cel de anul trecut + 1 nou

Anul 4: 3 profesori – cei 2 de anul trecut + 1 nou

Anul 5: 5 profesori – cei 3 de anul trecut + 2 noi

Anul 6: 8 profesori – cei 5 de anul trecut + 5 noi

Anul 7: 13 profesori – cei 8 de anul trecut + 5 noi

Anul 8: 21 profesori – cei 13 de anul trecut + 8 noi

Problema numărului de profesori

f(n) = #numărul de profesori după anul n

- $f(n+1) = \underbrace{f(n)}_{\text{Profesorii de anul trecut}} + \underbrace{f(n-1)}_{\text{profesori noi}}$

Recunoașteți această recurență?

E modul de construire a renumitului

Șir al lui Fibonacci

Problema numărului de profesori

Șirul lui Fibonacci e prima recurență despre care se știe că s-a studiat matematic, dintre toate recurențele studiate

A fost publicat pentru prima dată în tratatul *"Liber abaci"* de *Leonardo Fibonacci din Pisa* în anul *1202*. Acest tratat cuprindea tot ce se știa despre matematică la acele vremuri și a influențat dezvoltarea matematicii în anii următori

A studiat cu ajutorul lui o problemă reală a acelor vremuri, creșterea populației de iepuri, pe care a exprimat-o astfel:

- în fiecare lună, o pereche de iepuri vor naște în medie alți 2 iepuri, cu excepția primei luni de viață

Șirul lui Fibonacci

Problema numărului de profesori o reducem matematic la rezolvarea *ecuației*:

f(n+1) = f(n) + f(n-1), n≥2 cu f(0) = 0 și f(1) = 1

Presupunem că șirul are forma

$$f(n+1) = \lambda^{n+1}, unde\ \lambda\ parametru\ real$$

Ecuația devine:

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

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

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

Șirul lui Fibonacci

Ecuația de gradul 2:

$\lambda^2 - \lambda^1 - 1 = 0$ are soluțiile:

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

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

Ecuația $\lambda^2 - \lambda^1 - 1 = 0$ poartă denumirea de *ecuația caracteristică asociată*, iar dacă aceasta are 2 soluții distince, atunci soluția generală a ecuației de la care am plecat este:

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

Șirul lui Fibonacci

Mai știm că *f(0)= 0* și *f(1) = 1*

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

Cu soluțiile: $c_1 = \frac{1}{\sqrt{5}}$ și $c_2 = -\frac{1}{\sqrt{5}}$

Șirul lui Fibonacci

În final, am demonstrat mai sus că *termenul n din șirul lui Fibonacci* are forma:

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

Nu e ceva ușor de calculat, europenilor le-a luat 6 secole să ajungă la această soluție

Șirul lui Fibonacci

Adevărul e că Fibonacci nu a descoperit acest șir, el doar *le-a spus europenilor despre el*, fiind folosit în jurul anului *200* de către *matematicienii indieni* și având aplicații în gramatică și muzică

Kepler l-a folosit în secolul 16 pentru a studia cum sunt dispuse *frunzele unei flori pe tulpină*, care e numărul de frunze de la fiecare nivel

Matematicianul *Abraham de Moivre* a fost cel care a descoperit *formula lui Binet*, în secolul 17, formula de pe slide-ul precedent

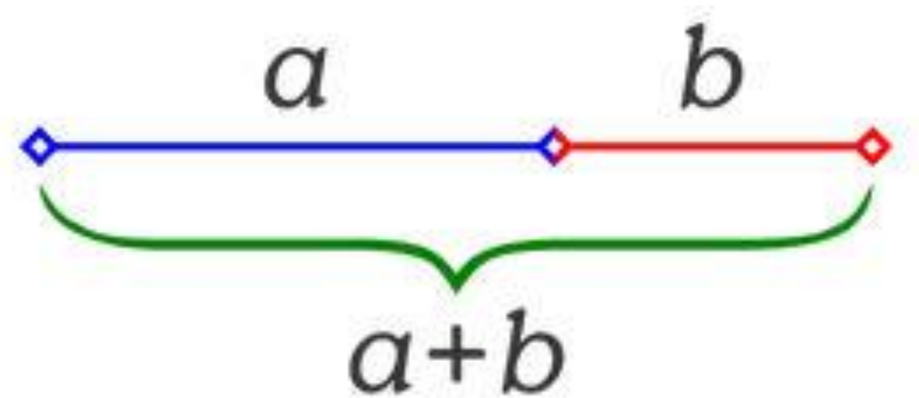
Șirul lui Fibonacci

Formula lui Binet face legătura între termenii șirului lui Fibonacci și puterea n a *numărului de aur* (golden ratio sau secțiunea de aur, raportul de aur, proporția de aur), primul număr irațional descoperit și definit în istorie

$\varphi = \frac{1+\sqrt{5}}{2} = 1{,}618033...$

Euclid l-a definit prima oară folosind raportul:

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



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

Recurență liniară

Definiție: O *recurență este liniară* dacă are forma

$$f(n) = a_1 f(n\text{-}1) + a_2 f(n\text{-}2) + \dots\ a_d f(n\text{-}d)$$
$$= \sum_{i=1}^{d} a_i f(n-i), \text{ cu numere fixe } a_i \text{ și d}$$

d se numește ordinul recurenței

Ce ordin are recurența șirul lui Fibonacci?

Mulțimi definite inductiv

Recursivitate

Funcții recursive în PYTHON

Potrivirea de tipare

Tail recursion

Avantajele și dezavantajele recursivității

Recursivitate în informatică

O noțiune e *recursivă* dacă e *folosită în propria sa definiție*.

Recursivitatea e fundamentală în informatică:

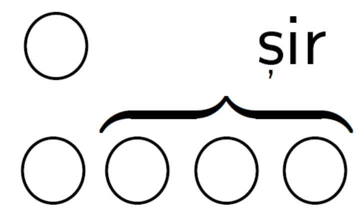
- dacă o problemă are soluție, *se poate rezolva recursiv*
- reducând problema la un caz mai simplu al *aceleiași probleme*

Înțelegând recursivitatea, putem rezolva orice problemă fezabilă

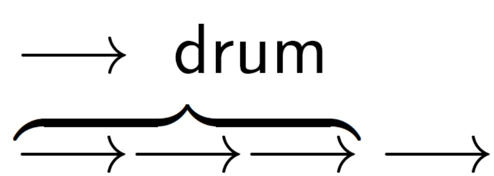
Recursivitate: exemple

Recursivitatea reduce o problemă la un caz mai simplu al aceleiași probleme

$$\text{un } \textit{șir} \text{ e } \begin{cases} \text{un singur element} \\ \text{un element urmat de un } \textit{șir} \end{cases}$$



$$\text{un } \textit{drum} \text{ e } \begin{cases} \text{un pas} \\ \text{un } \textit{drum} \text{ urmat de un pas} \end{cases}$$



Șiruri recurente

Progresia aritmetică:

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

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

Progresia geometrică:

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

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

Definițiile de mai sus nu calculeaă x_n *direct* ci *din aproape în aproape*, în funcție de x_{n-1}

Elementele unei definiții recursive

1. *Cazul de bază* este cel mai simplu caz pentru definiția dată, definit direct (termenul inițial dintr-un șir recurent: x_0)
Cazul de bază nu trebuie să lipsească

2. *Relația de recurență* propriu-zisă – definește noțiunea, folosind un caz mai simplu al aceleiași noțiuni

3. Demonstrația de *oprire a recursivității* după un număr finit de pași

Funcții recursive

O funcție e *recursivă* dacă apare în propria sa definiție.

O funcție f e definită recursiv dacă există cel puțin o valoare $f(x)$ *definită în termenii altei valori* $f(y)$, unde $x \neq y$.

Funcții recursive peste mulțimi inductive

Multe funcții recursive au ca domeniu *mulțimi inductive*

Dacă S este o mulțime inductivă, putem folosi *constructorii* săi pentru a defini *o funcție recursivă f cu domeniul S:*

- *baza*: pentru fiecare element de bază x € S specificăm o valoare f (x)
- *Inducția*: dăm una sau mai multe reguli care pentru orice x € S, x definit inductiv, definesc f (x) în termenii unor alte valori ale lui f, definite anterior

Funcții recursive în PYTHON

Forma generică a unei funcții recursive:

```
def functie_recursiva ():
        ...
        functie_recursiva()
        ...

functie_recursiva()
```

Funcții recursive în PYTHON

Exemplu:

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

frec()
```

- La fiecare apel, se alocă *spațiu de memorie nou*, distinct pentru x
- În exemplu de mai sus e greșit faptul că *nu apare condiția de oprire* din execuție

Funcții recursive în PYTHON

Exemplu: Să se afișeze pe ecran în ordine descrescătoare toate numerele naturale mai mici ca n.

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

OUTPUT:

3

2

1

0

Funcții recursive în PYTHON

Exemplu: Funcția factorial

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

Putem să o rescriem recursiv:

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

Funcții recursive în PYTHON

Detaliem *modul în care se calculează recursiv* funcția factorial:

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

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

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

.

.

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

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

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

Funcții recursive în PYTHON

Matematic, *forma recursivă* a funcției factorial este:

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

Funcții recursive în PYTHON

În PYTHON putem scrie funcția factorial astfel:

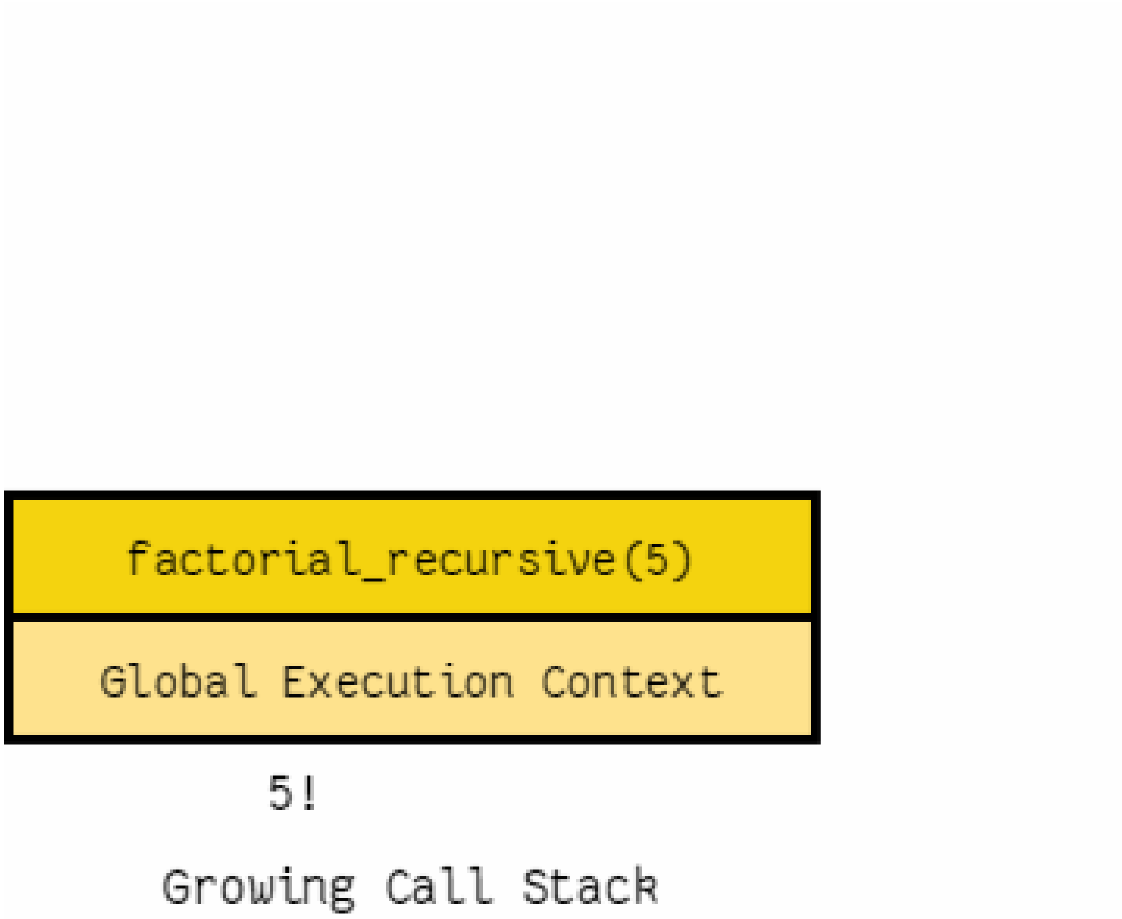
```
def factorial (x):
        if(x ==1):
                return 1
        else:
                return x * factorial (x-1)

print(factorial(4))
```

OUTPUT:

24

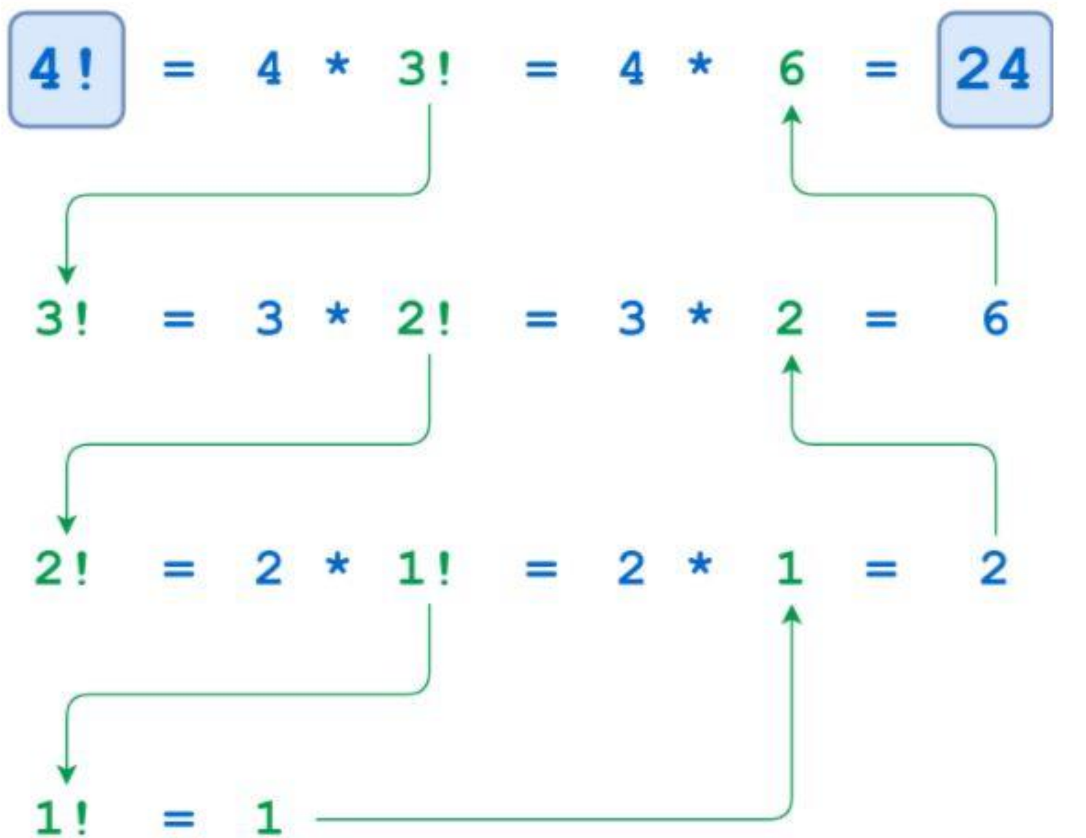
Funcții recursive în PYTHON



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

Funcții recursive în PYTHON

Pașii pe care îi face programul pentru a calcula:



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

Funcții recursive în PYTHON

- Cum funționează funcția recursivă factorial:

factorial (4) va avea la execuție următorii pași:

factorial(4) = factorial(3) * 4 – funcția rămâne în execuție
factorial(3) = factorial(2) * 3 – funcția rămâne în execuție
factorial(2) = factorial(1) * 2 – funcția rămâne în execuție
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

O problemă nerezolvată: “problema $3 \cdot n + 1$”

Conjectura lui Collatz (1937),

Fie un număr pozitiv *n*:

- dacă e par, îl împărțim la 2: $n/2$
- dacă e impar, îl înmulțim cu 3 și adunăm 1: $3 \cdot n + 1$

Se ajunge la 1 pornind de la orice număr pozitiv ?

(problemă nerezolvată în matematică...)

$$f(n)=\begin{cases} n/2, & dacă\ n\ e\ par \\ 3 \cdot n + 1, & dacă\ n\ e\ impar \end{cases}$$

Exemple:

3→10→5→16→8→4→2→1

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

O problemă nerezolvată: “problema 3 · n + 1”

Câți pași sunt necesari până ajungem la #1?

- Definim funcția *p* : N∗ → N care numără pașii până la oprire:
  - pentru 3→10→5→16→8→4→2→1 avem 7 pași
- Nu avem o formulă cu care să definim *p*(*n*) direct.
- Dar dacă șirul *n*, *f* (*n*), *f* (*f* (*n*)), . . . ajunge la 1, atunci numărul de pași parcurși **de la** *n* e **cu unul mai mare** decât continuând **de la** *f* (*n*)
- $p(n) = \begin{cases} 0, & dacă\; n = 1 \\ 1 + f(n), & altfel\; (n > 1) \end{cases}$
- Funcția *p* e folosită în propria definiție, deci a fost definită *recursiv*.

O problemă nerezolvată: “problema $3 \cdot n + 1$”

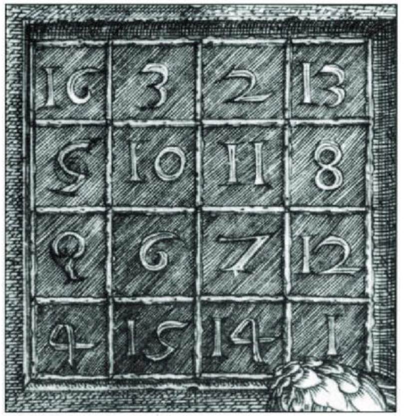
```
def urmatorul(n):
   if(n%2 == 0):
      return n/2
   else:
      return 3 * n + 1

def pasi(n):
   if (n == 1):
      return 0
   else:
      return 1 + pasi(urmatorul(n))
```

Șirul lui Fibonacci în 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
```



Mulțimi definite inductiv

Recursivitate

Funcții recursive în PYTHON

Potrivirea de tipare

Tail recursion

Avantajele și dezavantajele recursivității

Potrivirea de tipare (pattern matching)

Putem scrie funcția și în acest mod, folosind potrivirea de tipare:

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

Potrivirea de tipare (pattern matching)

Ultima condiție putem să o scriem în aceste forme, sunt echivalente:

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

Potrivirea de tipare (pattern matching)

Modul de în care se execută *match case*:

Se verifică, pe rând, *de la primul* case la ultimul, dacă valoarea se potrivește.

- dacă *nu se potrivește* se merge la următorul case
- dacă *se potrivește* se execută instrucțiunea de la case-ul respectiv, iar restul case-urilor nu se mai verifică

Potrivirea de tipare (pattern matching)

Exemplu: Să se afișeze dacă *un punct e pe axe*:

```
def puncte (x, y):
  match (x, y):
    case (0, 0):
      print("Punctul e în origine")
    case (0, _):
      print("Punctul e pe axa Ox")
    case (_, 0):
      print("Punctul e pe axa Oy")
    case (_, _):
      print("Punctul nu e pe nici o axa")
puncte(0,3)    # Punctul e pe axa Ox
```

- Mulțimi definite inductiv
- Recursivitate
- Funcții recursive în PYTHON
- Potrivirea de tipare
- **Tail recursion**
- Avantajele și dezavantajele recursivității

Limitare în PYTHON

Unul dintre dezavantajele recursivității este că fiecare apel de funcție care rămâne în execuție folosește *spațiu de memorie pe stivă*

PYTHON *limitează* implicit numărul de apeluri ale aceleiași expresii la 1000 (10**3) de ori

Eroarea care apare la apelarea de mai mult de 1000 de ori a aceleiași expresii generează eroarea: *maximum recursion depth exceeded error*

În probleme în care avem nevoie de iterații mai mari de 1000 putem modifica această limită folosind funcția *setrecursionlimit()* din modulul *sys*

Limitare în PYTHON

Pentru a *crește limita* de la maxim 1.000 de apeluri la maxim 100.000 de apeluri:

```python
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, recursivitate clasică:

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


print(factorial(4))
```

Tail recursion

- Cum funționează funcția recursivă factorial:

factorial (4) va avea la execuție următorii pași:

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

Exemplu de folosire if - else

PYTHON permite și o scriere mai compactă a instrucțiunii if-else astfel:

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

print(factorial(4))
```

OUTPUT:

24

Mulțimi definite inductiv

Recursivitate

Funcții recursive în PYTHON

Potrivirea de tipare

Tail recursion

Avantajele și dezavantajele recursivității

Avantajele recursivității în programare

- Codul e *mai scurt* și ușor de urmărit, elegant, curat
- Problemele complexe pot fi împărțite în *subprobleme mai simple* și astfel mai ușor de rezolvat
- *Generarea de șiruri* se face mai simplu recursiv

Dezvantajele recursivității în programare

- E mai greu de urmărit pas cu pas *logica din spatele* unui cod scris recursiv
- Apelurile recursive repetate folosesc *multă memorie*
- Erorile care apar la funcțiile recursive sunt *mai greu de corectat*

De știut

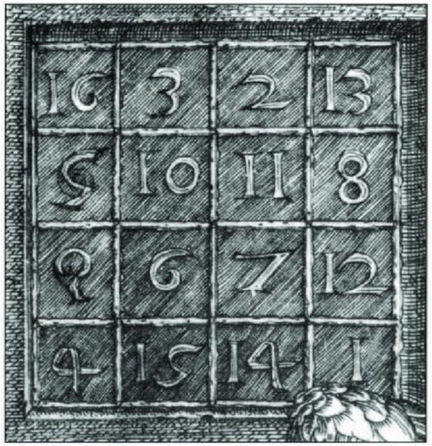
Să recunoaștem și să definim noțiuni recursive

Să recunoaștem dacă o definiție recursivă e corectă

- are caz de bază? se oprește recursivitatea?

Să rezolvăm probleme scriind funcții recursive

- cazul de bază + pasul de reducere la o problemă mai simplă



Vă mulțumesc!

Bibliografie

- Exemplele cu Turnurile din Hanoi și problema numărului de profesori au fost inspirate din cursul ***Mathematics for Computer Science*** de la Massachusetts Institute of Technology (de pe https://ocw.mit.edu/)
- Conținutul cursului se bazează preponderent pe materialele de anii trecuți de la cursul de LSD, predat de conf. dr. ing.Marius Minea și ș.l. dr. ing. Casandra Holotescu (http://staff.cs.upt.ro/~marius/curs/lsd/index.html)