

Seminar 1 – Recursivitate

Tema seminarului: Algoritmi recursivi și modelul matematic recursiv

I. Algoritmi recursivi cu numere

1. Verificați dacă un număr este norocos (lucky number). Un număr este norocos dacă este alcătuit doar din cifrele 4 și 7.

$$\text{norocos}(n) = \begin{cases} \text{adev\c{a}rat}, & n = 4 \text{ sau } n = 7 \\ \text{fals}, & \text{dac\c{a}} n \% 10 \neq 4 \text{ \c{a}i } n \% 10 \neq 7 \\ \text{norocos}(\lfloor \frac{n}{10} \rfloor), & \text{altfel} \end{cases}$$

```
def norocos(n):  
    if n == 4 or n == 7:  
        return True  
    elif n % 10 != 4 and n % 10 != 7:  
        return False  
    else:  
        return norocos(n // 10)
```

2. Să se calculeze suma divizorilor proprii și improprii ai unui număr natural $n > 1$.
 - Divizorii proprii și improprii ai unui număr sunt din intervalul $[1...n]$. Vom considera un parametru auxiliar, care reprezintă valoarea curentă din acest interval pe care o verificăm dacă este divizor.

$$\begin{aligned} & \text{SumaDivizori}(n, \text{div}_{\text{curent}}) \\ &= \begin{cases} n, & \text{dac\c{a}} n = \text{div}_{\text{curent}} \\ \text{div}_{\text{curent}} + \text{SumaDivizori}(n, \text{div}_{\text{curent}} + 1), & \text{dac\c{a}} n \% \text{div}_{\text{curent}} = 0 \\ \text{SumaDivizori}(n, \text{div}_{\text{curent}} + 1), & \text{altfel} \end{cases} \end{aligned}$$

```
def SumaDivizori(n, div_curent):  
    if div_curent == n:  
        return n  
    elif n % div_curent == 0:  
        return div_curent + SumaDivizori(n, div_curent+1)  
    else:  
        return SumaDivizori(n, div_curent+1)
```

- Parametrul `div_curent` trebuie inițializat la primul apel cu valoarea 1. În situații în care introducem parametri noi într-o funcție, care trebuie să primească o anumită valoare la primul apel, trebuie să mai scriem încă o funcție, principală, care nu face altceva decât să apeleze funcția `SumaDivizori` inițializând `div_curent` cu valoarea 1.

$$SumaDivizoriPrincipal(n) = SumaDivizori(n, 1)$$

```
def SumaDivizoriPrincipal(n):
    return SumaDivizori(n, 1)
```

II. Algoritmi recursivi cu liste

Când lucrăm cu liste, vom porni de la definiția abstractă a listelor: o listă este o secvență de elemente, în care fiecare element are o poziție fixată. În modelele matematice recursive, listele sunt reprezentate enumerând elementele listei: $l_1 l_2 l_3 \dots l_n$. Nu vom discuta despre reprezentarea listei, nu avem operații predefinite pe liste (cum am discutat de operațiile din interfața listei la Structuri de Date și Algoritmi). Totuși, uitându-ne la ce putem face cu aceste liste, ele seamănă cu listele înlănțuite mai degrabă decât cu liste reprezentate pe vectori.

- Nu avem acces la lungimea listei. Dacă vrem să aflăm numărul de elemente din listă, trebuie să facem o funcție (recursivă) separată pentru a număra elementele listei.
 - Putem verifica totuși dacă lungimea listei este egală cu o valoare constantă. Putem face verificări de genul:
 - o $n = 0$ (lista vidă)
 - o $n = 1$ (lista cu un singur element)
 - o $n = 2$ (lista cu 2 elemente)
 - o $n < 2, n > 2 \dots$
 - o ... etc. (deși în general nu verificăm mai mult de 3 elemente)
 - o **Nu** putem face verificări de genul : $n > k$ (unde k nu este o valoare constantă)
 - Nu putem accesa elemente de pe orice poziție, ci doar de la începutul listei, și doar un număr constant de elemente :
 - o l_1 este primul element din listă
 - o l_2 este al doilea element din listă
 - o l_3 este al treilea element din listă
 - o **Nu** avem acces direct la ultimul element (l_n) sau la un element de pe poziția k (l_k), dar se pot face funcții (recursive) pentru a le afla
 - Când accesăm elemente de la începutul listei, avem acces și la restul listei, adică sublista din care au fost eliminate elementele accesate.
3. Să se calculeze produsul numerelor pare dintr-o listă.

$$ProdPare(l_1 l_2 \dots l_n) = \begin{cases} 1, & \text{dacă } n = 0 \\ l_1 * ProdPare(l_2 \dots l_n), & \text{dacă } l_1 \% 2 == 0 \\ ProdPare(l_2 \dots l_n), & \text{altfel} \end{cases}$$

- În Python vom lucra cu o listă abstractă, care presupunem că are următoarele operații:
 - o *creazăListăVidă()* – creează o listă vidă. Returnează lista creată
 - o *eListăVidă(lista)* – verifică dacă *lista* e o listă vidă, returnează *True* sau *False*
 - o *primElement(lista)* – returnează primul element din lista *lista*. Dacă *lista* e vidă, returnează *None*
 - o *sublista(lista)* – returnează o copie a listei *lista*, fără primul element (practic $l_2 \dots l_n$)

- *adaugaInceput(lista, element)* – adaugă la începutul unei copii a listei *lista* elementul *element* si returneaza lista rezultat

```
def prodPare(lista):
    if eListaVida(lista):
        return 1
    elif primElement(lista) % 2 == 0:
        return primElement(lista) * prodPare(sublista(lista))
    else:
        return prodPare(sublista(lista))
```

Ce se întâmplă dacă apelăm funcția pentru lista [1,2,3,4,5,6] ?

```
ProdPare([1,2,3,4,5,6]) = (ramura a 3-a din model matematic recursiv)
  ProdPare([2,3,4,5,6]) = (ramura a 2-a din model)
    2 * ProdPare([3,4,5,6]) = (ramura a 3-a)
      ProdPare([4,5,6]) = (ramura a 2-a)
        4 * ProdPare([5,6]) = (ramura a 3-a)
          ProdPare([6]) = (ramura a 2-a)
            6 * ProdPare([]) = (prima ramură)
              1
            6 * 1 = 6 => ProdPare([6]) = 6
          6 => ProdPare([5,6]) = 6
        4 * 6 = 24 => ProdPare([4,5,6]) = 24
      24 => ProdPare([3,4,5,6]) = 24
    2 * 24 = 48 => ProdPare([2,3,4,5,6]) = 48
  48 => ProdPare([1,2,3,4,5,6]) = 48
```

Cât este rezultatul dacă apelăm ProdPare([])?

Cum trebuie modificat codul dacă dorim ca pentru lista vidă funcția să returneze -1?.

$$ProdParePrincipala(l_1 l_2 l_3 \dots l_n) = \begin{cases} -1, & \text{dacă } n = 0 \\ ProdPare(l_1 l_2 l_3 \dots l_n), & \text{altfel} \end{cases}$$

```
def prodParePrincipala(lista):
    if eListaVida(lista):
        return -1
    else:
        return prodPare(lista)
```

Când rezultatul funcției trebuie să fie o listă, de fiecare dată vom construi o listă nouă (chiar dacă rezultatul funcției trebuie să fie lista dată ca parametru cu niște modificări). Nu putem modifica lista

primită ca parametru. În lista rezultată, vom adăuga fiecare element (din lista parametru), dacă e cazul, folosind operația de reuniune (U). Cu toate acestea, noi putem adăuga la o listă (lista rezultată) doar elemente (deci nu putem reuni două liste) și doar la început (nu la mijloc, nu la sfârșit).

4. Să se adauge o valoare e pe poziția m ($m \geq 1$) într-o listă.

$$adauga(l_1 l_2 l_3 \dots l_n, e, m) = \begin{cases} \emptyset, & \text{dacă } n = 0, m > 1 \\ (e), & \text{dacă } n = 0, m = 1 \\ e \cup l_1 l_2 l_3 \dots l_n, & \text{dacă } m = 1 \\ l_1 \cup adauga(l_2 l_3 \dots l_n, e, m - 1), & \text{altfel} \end{cases}$$

```
def adauga(lista, elem, m):
    if eListaVida(lista) and m > 1:
        return creazaListaVida()
    elif eListaVida(lista):
        return adaugaInceput(creazaListaVida(), elem)
    elif m == 1:
        return adaugaInceput(lista, elem)
    else:
        return adaugaInceput(adauga(sublista(lista), elem, m-1), primElement(lista))
```

5. Să se adauge o valoare e dată din m în m ($m \geq 2$). De exemplu pentru lista $[1,2,3,4,5,6,7,8,9,10]$, $e = 111$ și $m = 4$, rezultatul este $[1,2,3,111,4,5,6,111,7,8,9,111,10]$.

Ce trebuie modificat dacă avem nevoie de adăugare repetată? În momentul adăugării (ramura a 3-a) nu ne oprim ci trebuie să continuăm. Avem 2 variante:

1. Adăugăm încă un parametru, pentru a reține valoarea originală a lui m (pentru că m scade la 1 pe parcursul apelurilor), iar după adăugare continuăm recursiv revenind la valoarea originală a lui m .
2. Adăugăm încă un parametru, care reține poziția curentă în listă la care ne aflăm. Când poziția curentă este multiplu de m adăugăm elementul e .

Indiferent de varianta aleasă, din moment ce am introdus un parametru în plus, trebuie să mai scriem o funcție care face apelul acestor funcții, inițializând corespunzător parametrul auxiliar.

Varianta 1:

$$adaugaMV1(l_1 l_2 l_3 \dots l_n, e, m, m_{orig}) = \begin{cases} \emptyset, & \text{dacă } n = 0 \text{ și } m > 1 \\ (e), & \text{dacă } n = 0 \text{ și } m = 1 \\ e \cup adaugaMV1(l_1 l_2 l_3 \dots l_n, e, m_{orig}, m_{orig}), & \text{dacă } m = 1 \\ l_1 \cup adaugaMV1(l_2 l_3 \dots l_n, e, m - 1, m_{orig}), & \text{altfel} \end{cases}$$

```

def adaugaMV1(lista, elem, m, mo):
    if eListaVida(lista) and m > 1:
        return creazaListaVida()
    elif eListaVida(lista):
        return adaugaInceput(creazaListaVida(), elem)
    elif m == 1:
        return adaugaInceput(adaugaMV1(lista, elem, mo, mo), elem)
    else:
        return adaugaInceput(adaugaMV1(sublista(lista), elem, m-1, mo), primElement(lista))

```

$$adaugaMV1Principal(l_1 l_2 l_3 \dots l_n, e, m) = adaugaMV1(l_1 l_2 l_3 \dots l_n, e, m, m)$$

```

def adaugaMV1Principal(list, e, m):
    return adaugaMV1(list, e, m, m)

```

Varianta 2:

$$\begin{aligned}
 & adaugaMV2(l_1 l_2 \dots l_n, e, m, curent) \\
 &= \begin{cases} \emptyset, & \text{dacă } n = 0 \text{ și } curent \% m \neq 0 \\ (e), & \text{dacă } n = 0 \text{ și } curent \% m = 0 \\ e \cup adaugaMV2(l_1 l_2 \dots l_n, e, m, curent + 1), & \text{dacă } curent \% m = 0 \\ l_1 \cup adaugaMV2(l_2 l_3 \dots l_n, e, m, curent + 1), & \text{altfel} \end{cases}
 \end{aligned}$$

```

def adaugaMV2(lista, elem, m, curent):
    if eListaVida(lista) and curent % m != 0:
        return creazaListaVida()
    elif eListaVida(lista):
        return adaugaInceput(creazaListaVida(), elem)
    elif curent % m == 0:
        return adaugaInceput(adaugaMV2(lista, elem, m, curent + 1), elem)
    else:
        return adaugaInceput(adaugaMV2(sublista(lista), elem, m, curent+1), primElement(lista))

```

$$adaugaMV2Principal(l_1 l_2 l_3 \dots l_n, e, m) = adaugaMV2(l_1 l_2 l_3 \dots l_n, e, m, 1)$$

```

def adaugaMV2Principal(list, e, m):
    return adaugaMV2(list, e, m, 1)

```