Seminar 1 – Recursivitate

1. Verificați dacă un număr natural este norocos. Se consideră că un număr este norocos dacă este alcătuit doar din cifrele 4 și 7.

$$norocos(n) = \begin{cases} adevărat, & n = 4 \; sau \; n = 7 \\ fals, & dacă \; n \,\%\, 10 \; != 7 \; și \; n \,\%\, 10 \; != 4 \\ norocos\left(\frac{n}{10}\right), & altfel \end{cases}$$

```
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 nenul *n*.

- Divizorii se regăsesc din intervalul [1…n]. Vom considera un parametru în plus, care reprezintă valoarea curentă, divizorul candidat, din acest interval pentru care verificăm dacă este într-adevăr divizor.

$$SumaDivizori(n, div_{curent}) = \begin{cases} n, & dacă \; n = div_{curent} \\ div_{curent} + SumaDivizori(n, div_{curent} + 1), & dacă \; n \,\%\, div_{curent} = 0 \\ SumaDivizori(n, div_{curent} + 1), & altfel \end{cases}$$

```
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 în apelul inițial cu valoarea 1. În situațiile în care introducem parametri noi într-o funcție, aceștia trebuind să primească anumite valori la primul apel, este necesar să scriem încă o funcție auxiliară, nerecursivă, care nu face altceva decât să apeleze funcția recursivă cu valorile corespunzătoare pentru parametrii adiționali. În cazul nostru, se va apela funcția *SumaDivizori* inițializând *div_curent* cu valoarea 1.

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

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

Când operăm 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: $l_1 l_2 l_3 ... l_n$.
În descrierea soluțiilor recursive (inclusiv prin modelul matematic), vom ține cont de următoarele:

- Nu avem acces direct la lungimea listei. Dacă dorim sa 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. Așadar, putem efectua verificări de forma:
    - n = 0 (lista vidă)
    - n = 1 (lista cu un singur element)
    - n = 2 (lista cu 2 elemente)
    - n < 2, n > 2 …
    - … etc. (deși, în general, nu verificăm mai mult de 3 elemente)
    - **Nu** putem face verificări de forma : 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 :
    - l_1 este primul element din listă
    - l_2 este al doilea element din listă
    - l_3 este al treilea element din listă
    - **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 sub-lista din care au fost eliminate elementele accesate.

3. Să se calculeze produsul numerelor pare dintr-o listă.

$$ProdPare(l_1 l_2 ... l_n) = \begin{cases} 1, & dacă n = 0 \\ l1 * ProdPare(l_2 ... l_n), & dacă\ l_1\ \%\ 2 == 0 \\ ProdPare(l_2 ... l_n), & altfel \end{cases}$$

- În Python vom lucra cu o listă abstractă, pentru care presupunem următoarele operații:
    - *creazăListăVidă()* – creează o listă vidă și returnează lista creată
    - *eListaVidă(lista)* – verifică dacă *lista* e o listă vidă (adică n=0), returnând *True* sau *False*
    - *primElement(lista)* – returnează primul element din lista *lista* (adică l_1). Dacă *lista* e vidă, returnează None
    - *sublista(lista)* – returnează o copie a listei *lista*, fără primul element (așadar, $l_2 ... l_n$)
    - *adaugaInceput(lista, element)* – adaugă la începutul unei copie a listei *lista* elementul *element* și returnează 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?.

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

```
def prodPareMain(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, (ușor) modificată). Nu vom modifica lista primită ca parametru. În lista rezultat, vom adăuga fiecare element (din lista parametru), dacă este cazul, folosind operația de reuniune. Cu toate acestea, noi putem adăuga la o listă (lista rezultat) doar elemente (deci nu putem reuni două liste (direct)) și doar la început (nu "la mijloc", nu la sfârșit).

4. Să se adauge o valoare *e* pe poziția *m* (m >=1) într-o listă.

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

```
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 >=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 vom opri, fiind necesar să continuăm.

Avem următoarele posibilități:

1. Adăugăm încă un parametru, pentru a reține valoarea originală a lui *m* (pentru că *m* este decrementat pe parcursul apelurilor), iar după adăugare continuăm recursiv revenind la valoarea inițială a lui *m*.

2. Adăugăm încă un parametru, care reține poziția curentă în listă la care ne aflăm. Când poziția curentă este multiplul a lui *m,* adăugăm elementul *e*.

Indiferent de varianta aleasă, din moment ce am introdus un parametru în plus, este necesar să mai scriem o funcție care apelează aceste funcții, inițializând valoarea parametrului auxiliar la cea corespunzătoare.

Varianta 1:

$$adaugaNV1(l_1 l_2 l_3 \ldots l_n, e, m, m_{orig})$$
$$= \begin{cases} \emptyset, & dacă\ n = 0\ și\ m > 1 \\ (e), & dacă\ n = 0\ și\ m = 1 \\ e \cup adaugaNV1(l_1 l_2 l_3 \ldots l_n, e, m_{orig}, m_{orig}), & dacă\ m = 1 \\ l_1 \cup adaugaNV1(l_2 l_3 \ldots l_n, e, m-1, m_{orig}), & altfel \end{cases}$$

```
def adaugaNV1(lista, elem, m, mo):
    if eListaVida(lista) and m > 1:
```

```
        return creazaListaVida()
    elif eListaVida(lista):
        return adaugaInceput(creazaListaVida(), elem)
    elif m == 1:
        return adaugaInceput(adaugaNV1(lista, elem, mo, mo), elem)
    else:
        return adaugaInceput(adaugaNV1(sublista(lista), elem, m-1, mo), primElement(lista))
```

$$adaugaNV1Main(l_1 l_2 l_3 \ldots l_n, e, m) = adaugaNV1(l_1 l_2 l_3 \ldots l_n, e, m, m)$$

```
def adaugaNV1Main(list, e, m):
    return adaugaNV1(list, e, m, m)
```

Varianta 2:

$$adaugaNV2(list, e, m, curent)$$
$$= \begin{cases} \emptyset, & dacă\ n = 0\ și\ curent\ \%\ m\ !\ = 0 \\ (e), & dacă n = 0\ și\ curent\ \%\ m = 0 \\ e\ \cup adaugaNV2\ (list, e, m, curent + 1), & dacă\ curent\ \%\ m = 0 \\ l_1 \cup adaugaNV2(l_2 l_3 \ldots l_n, e, m, curent + 1), & altfel \end{cases}$$

```
def adaugaNV2(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(adaugaNV2(lista, elem, m, curent + 1), elem)
    else:
        return adaugaInceput(adaugaNV2(sublista(lista),elem, m, curent+1), primElement(lista))
```

$$adaugaNV2Main(l_1 l_2 l_3 \ldots l_n, e, m) = adaugaNV2(l_1 l_2 l_3 \ldots l_n, e, m, 1)$$

```
def adaugaNV2Main(list, e, m):
    return adaugaNV2(list, e, m, 1)
```