Seminar 1
Recursive algorithms and the recursive mathematical model

1. Check if a number is a lucky number. A number is considered lucky if it contains only the digits 4 and 7.

Recursive mathematical model:

$$lucky(n) = \begin{cases} true, & n = 4 \; or \; n = 7 \\ false, & if \; n \,\%\, 10 \,! = 7 \; and \; n \,\%\, 10 \,! = 4 \\ lucky\left(\frac{n}{10}\right), & otherwise \end{cases}$$

Implementation (currently in Python, from the next seminar in Prolog):

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

2. Compute the sum of the proper divisors of a number *n*. For example if n = 20, we want to compute 2 + 4 + 5 + 10 = 21.

- The proper divisors of a number are included in the interval [2…n-1] (more precisely in [2…n/2]).We will consider an extra parameter which represents the current value from this interval, which will be checked to see if it is a divisor.

Recursive mathematical model:

$$DivisorsSum(n, div_{current}) = \begin{cases} 0, & if \; div_{current} > \frac{n}{2} \\ div_{current} + DivisorsSum(n, div_{current} + 1), & if \; n \,\%\, div_{current} = 0 \\ DivisorsSum(n, div_{current} + 1), & otherwise \end{cases}$$

Implementation:

```
Def DivisorsSum(n, div_current):
    if div_current > n/2:
        return 0
    elif n % div_current == 0:
        return div_current + DivisorsSum(n, div_current+1)
    else:
```

```
        return DivisorsSum(n, div_current+1)
```

- The parameter div_current has to be initialized at the first call with the value 2.
- If we add new parameters to a function, they have to receive a certain value at the first call, so we have to write an auxiliary function, and this auxiliary function will call the function DivisorsSum initializing the parameter div_current with 2.

Recursive mathematical model:

$$DivisorsSumMain(n) = DivisorsSum(n, 2)$$

Implementation:

```
def DivisorsSumMain(n):
    return DivisorsSum(n, 2)
```

For most problems, we are going to work with LISTS.
We will start from the abstract definition: a list is a sequence of elements, in which every element has a fixed position.

In the recursive mathematical model lists are represented through an enumeration of the list elements: $l_1l_2l_3 \ldots l_n$.

We will not discuss the representation of the lists and we do not have predefined operations.
However, if we analyze what we can do with these lists, these are more similar to linked lists than to lists represented on arrays/vectors.

- We don't have access to the length of the list. If we want to find the number of the elements from a list, we have to write a recursive function for counting the elements of the list.
- However, we can check if the length of the list is equal to a constant value.
  - We can check the following:
    - n = 0 (the list is empty)
    - n = 1 (the list has just an element)
    - n = 2 (the list have two elements)
    - n < 2, n >= 2 …
    - … etc. (although, generally, we do not check more than 3 elements)
  - We cannot check:
    - n > p (where p is a parameter, i.e., not a constant value)
    - or if we have two lists: $l_1l_2l_3 \ldots l_n$ and $m_1m_2 \ldots m_k$ we cannot compare their lengths
- We cannot access the elements from any position, we can access only the beginning of the list and only a constant number of elements:
  - We can access:
    - l_1 is the first element from list
    - l_2 is the second element from the list
    - l_3 is the third element from a list
  - We cannot access directly (but we can write recursive functions to do the following):
    - the last element (l_n)
    - an element from position k (l_k)

- When we access the elements from the beginning of the list, we actually divide the list into the first element(s) and the rest of the list, so we have access and to the rest of the list as well, in other words, the sub-list from which the accessed elements have been eliminated.
    - We can divide into:
        - l_1 and we automatically have $l_2 \ldots l_n$
        - l_2 and we automatically have $l_3 \ldots l_n$
        - l_3 and we automatically have $l_4 \ldots l_n$
    - We cannot divide into:
        - $l_1 \ldots l_{n-1}$ and l_n (we cannot just remove the last element)
        - $l_1 \ldots l_k$ and $l_{k+1} \ldots l_n$
- If the result of the function has to be a list, every time we will create a new list (even if the result of the function has to be the initial list transmitted as parameter, with some modification). We cannot modify the list received as a parameter. In the resulting list, we will add every element (from the parameter list if that's the case) using the reunion operation.
    - We can add to a list (the resulted list) just elements and only to the beginning
        - $e \cup l_1 \ldots l_n$
        - $e_1 \cup e_2 \cup l_1 \ldots l_n$
        - $l_1 \cup e \cup l_2 \ldots l_n$
    - We cannot:
        - $l_1 \ldots l_n \cup e$ (add element to the end of the list)
        - $l_1 \ldots l_n \cup m_1 \ldots m_k$ (add/concatenate two lists)

3. Compute the product of the even numbers from a list

Recursive mathematical model:

$$EvenProduct(l_1 l_2 \ldots l_n) = \begin{cases} 1, & if\ n = 0 \\ l1 * EvenProduct(l_2 \ldots l_n), & if\ l_1 \,\%\, 2 == 0 \\ EvenProduct(l_2 \ldots l_n), & otherwise \end{cases}$$

Implementation:

While Python has a list container, for the implementation of the problem, we will assume that we have our own list implementation which has the following operations:
- isEmpty(list) -> returns True or False
- sublist(list) -> returns the list without the first element
- firstElem (list) -> returns the first element of the list
- createEmpty() -> creates and returns an empty list
- addFirst(elem, list) -> adds the elem to the beginning of the list and returns the resulting list

```
def EvenProduct(listt):
    if isEmpty(listt):
        return 1
    elif firstElem(listt) % 2 == 0:
        return firstElem(listt) * EvenProduct(sublist(listt))
    else:
        return EvenProduct(sublist(listt))
```

What is the result, if we call the function for the list [1,2,3,4,5,6] ?

EvenProduct([1,2,3,4,5,6]) = (the 3-rd branch from recursive mathematical model)
EvenProduct([2,3,4,5,6]) = (the 2-nd branch from recursive mathematical model)
2 * EvenProduct([3,4,5,6]) = (the 3-rd branch from recursive mathematical model)
EvenProduct([4,5,6]) = (the 2-nd branch)
4 * EvenProduct([5,6]) = (the 3-rd branch)
EvenProduct([6]) = (the 2-nd branch)
6 * EvenProduct([]) = (first brunch)
1
6 * 1 = 6 =>EvenProduct([6]) = 6
6 =>EvenProduct([5,6]) = 6
4 * 6 = 24 =>EvenProduct([4,5,6]) = 24
24 =>EvenProduct([3,4,5,6]) = 24
2 * 24 = 48 =>EvenProduct([2,3,4,5,6]) = 48
48 =>EvenProduct([1,2,3,4,5,6]) = 48

Determine the result, if we call EvenProduct([])
Modify the code, so that the result of the function for the empty list to be -1.

Recursive mathematical model:

$$EvenProductMain(l_1 l_2 l_3 \dots l_n) = \begin{cases} -1, & if\ n = 0 \\ EvenProduct(l_1 l_2 l_3 \dots l_n), & otherwise \end{cases}$$

Implementation:

```
def EvenProductMain(listt):
    if isEmpty(listt):
        return -1
    else:
        return EvenProduct(listt)
```

4.

a. Add a value e on position *m* (m >=1) in a list (indexing starts from 1). For example, if we add in list [1,2,3,4,5,6], e = 11 on m = 4 => [1,2,3, 11, 4, 5, 6]

Recursive mathematical model:

$$add(l_1 l_2 l_3 \dots l_n, e, m) = \begin{cases} \emptyset, & if\ n = 0, m > 1 \\ (e), & if\ n = 0, m = 1 \\ e \cup l_1 l_2 l_3 \dots l_n, & if\ m = 1 \\ l_1 \cup add(l_2 l_3 \dots l_n, e, m - 1), & otherwise \end{cases}$$

Implementation:

```
def add(listt, e, m):
    if isEmpty(listt) and m >1:
        return createEmpty()
```

```
	elif isEmpty(listt) and m==1:
		return addFirst(e, createEmpty())
	elif m==1:
		return addFirst(e, listt)
	else:
		return addFirst(firstElem(listt), add(sublist(listt), e, m-1))
```

b. Add a value *e* in a list from *m* to *m(m >=2)*.

Eg: for the list: [1,2,3,4,5,6,7,8,9,10], *e* = 11 and *m* = 4, the result is [1,2,3,11,4,5,6,11,7,8,9,11,10].

Compared to point a), when we add (the 3-rd branch), we don't stop, we have to continue.
In this case, we have two options:

1. Adding another parameter, to save the original value of *m* (because m decreases to *1*, during the recursive calls) and after inserting we have to return to the original value of *m*.
2. Adding another parameter to save the current position in list. When the current position is multiple of *m*, we add the element *e*.

Whichever option we choose, since we have inserted an extra parameter, we have to write another function which will call these functions, setting the value of the extra parameter to an appropriate value.
Option 1:

<u>Recursive mathematical model:</u>

$$addNV1(l_1 l_2 l_3 \dots l_n, e, m, m_{orig}) = \begin{cases} \emptyset, & if\ n = 0\ and\ m > 1 \\ (e), & if\ n = 0\ and\ m = 1 \\ \quad e \cup addNV1\,(l_1 l_2 l_3 \dots l_n, e, m_{orig}, m_{orig}), & if\ m = 1 \\ l_1 \cup addNV1(l_2 l_3 \dots l_n, e, m-1, m_{orig}), & otherwise \end{cases}$$

```
def addNV1(listt, e, m, m_orig):
	if isEmpty(listt) and m >1:
		return createEmpty()
	elif isEmpty(listt) and m == 1:
		return addFirst(e, createEmpty())
	elif m == 1:
		return addFirst(e, addNV1(list, e, m_orig, m_orig))
	else:
		return addFirst(firstElem(listt), addNV1(sublist(listt), e, m-1, m_orig))
```

<u>Recursive mathematical model:</u>

$$addNV1Main(l_1 l_2 l_3 \dots l_n, e, m) = addNV1(l_1 l_2 l_3 \dots l_n, e, m, m)$$

<u>Implementation:</u>

```
def addNV1Main(listt, e, m):
        return addNV1(listt, e, m, m)
```

Option2:

Recursive mathematical model:

$$addNV2(list, e, m, current) = \begin{cases} \emptyset, & if\ \ n = 0\ și\ current\ \%\ m\ ! = 0 \\ (e), & if\ n = 0\ și\ current\ \%\ m = 0 \\ e\ \cup addNV2\ (list, e, m, current + 1), & if\ current\ \%\ m = 0 \\ l_1 \cup addNV2(l_2 l_3 \dots l_n, e, m, current + 1), & otherwise \end{cases}$$

Implementation:

```
def addNV2(listt, e, m, current):
        if isEmpty(listt) and current % m != 0:
                return createEmpty()
        elif isEmpty(listt) and current % m == 0:
                return addFirst(e, createEmpty())
        elif current % m == 0:
                return addFirst(e, addNV2(listt, e, m, current+1))
        else:
                return addFirst(firstElem(listt), addNV2(sublist(listt), e, m, current + 1))
```

Recursive mathematical model:

$$addNV2Main(l_1 l_2 l_3 \dots l_n, e, m) = \ addNV2(l_1 l_2 l_3 \dots l_n, e, m, 1)$$

Implementation

```
def addNV2Main(listt, e, m):
        return addNV2(listt, e, m, 1)
```