```
# Se considera o lista de numere intregi. Sa se scrie o functie care
  construuieste un dictionar in care fiecare numar din lista initiala
  are asociat multimea divizorilor sai:
# Exemplu:
  [12, 13, 14, 15, 16]
=> {12: {1, 2, 3, 4, 6, 12}, 13: {1, 13}, 14: {1, 2, 7, 14}, 15: {1, 3, 5, 15}, 16 : {1, 2, 4, 8, 16}}

def build_divizor_set (example_number):
    result = set()
    copy = example_number

    def traverse_index (example_number):
        if example_number == 0
            return 0
        if copy % example_number == 0:
            result.add(example_number)
        example_number = example_number - 1
        traverse_index (example_number)

    traverse_index (example_number)
    return result


def build_dict (example_list):
    result = dict()

    def traverse_list (example_list):
        if not example_list:
            return 0
        result[example_list[0]] = build_divizor_set (example_list[0])
        traverse_list (example_list[1:])

    traverse_list (example_list)
    return result


example_list = [12, 13, 14, 15, 16]
print (build_dict (example_list))

--------------------------------------------------------------
# Implement the function PARTITION, which takes as parameters a
   condition as a function and a list and returns a pair of lists, with
   the elements that satisfy and do not satisfy the condition:

def partition (example_list, condition):
    result = (list(), list())

    def traversal (example_list, condition):
        if not example_list:
            return 0
        if condition (example_list[0]):
            result[0].append(example_list[0])
        else:
            result[1].append(example_list[0])
       travesral (example_list[1:]

    traversal (example_list)
    return result

example_list = [4, 6, 7, 5, 4, 8, 9]
```

```
condition = lambda x: x >= 5
print (partition (example_list, condition))

---------------------------------------------------------------------
# Write a function that takes as parameters 2 lists (the first list has
   all distinc elements) and returns a dictionary that has keys from
   the first list and values from the second list.
   If the lists are of different lengths, the dictionary will have a
   number of elements equal to the number of elements in the
   shorter list.

# Example:
# Input: [1, 18, 118], [0, 1, 1, 1]
# Output: {1: 0, 18: 1, 118: 1}

def build_dict (list_1, list_2):
    result = dict()

        def traversal (list_1, list_2):
            if not list_1:
                return 0
            if not list_2:
                return 0
            result[list_1[0]] = list_2[0]
            traversal (list_1[1:], list_2[1:])

    traversal (list_1, list_2)
    return result

list_1 = [1, 18, 118]
list_2 = [0, 1, 1, 1]
print (build_dict (list_1, list_2))

------------------------------------------------------------------
# Given a dictionary, take from it the keys and values and put each in a different list.

# Example:
# Input: {1: 0, 18: 1, 118: 1}
# Output: [1, 18, 118], [0, 1, 1, 1]

example_dict = {1: 0, 18: 1, 118: 1}
first_list = list (example_dict.keys())
second_list = list (example_dict.values())
print (first_list, second_list)

--------------------------------------------------------------------
# Deleting a node from a given tree as an argument to a specific
   function:

tree_data = {
  'value': 2,
  'left': None
  'right': {
    'value': -6,
    'left': {
      'value': 5,
      'left': None,
      'right': None
             },
     'right': {
        'value': -11,
        'left': None,
        'right': None
                  }
               }
            },
   'right': {
      'value': 10,
      'left': None,
      'right': None
              }
           }

def rsd (tree):    # preorder traversal
    if tree is not None:
        return [tree['value']] + rsd (tree['left']) + rsd (tree['right'])
    else:
        return []

def srd (tree):    # inorder traversal
    if tree is not None:
        return srd (tree['left']) + [tree['value']] + srd (tree['right'])
    else:
        return []

def sdr (tree):    # postorder traversal
    if tree is not None:
        return sdr (tree['left']) + sdr (tree['right']) + [tree['value']]
    else:
        return []

def delete_node (parent, value):
    if (parent['left']['value'] == value):
        parent ['left'] = None
    elif (parent['right']['value'] == value):
        parent ['right'] = None

delete_node (tree_data, 7)
print (rsd(tree_data))

-----------------------------------------------------------------
# Write a function that takes a binary tree and returns the list of
   nodes that have a single child. The order of the nodes in the list
   will be that of the inordal traversal:

def sigle_child (tree):
    result = list()

    def traversal (node):
        if node:
            if (node['right'] is not None and node['left'] is None) or
               (node['right'] is None and node['left'] is not None):
                result.append(node['value'])
             if node['right'] is not None or node['left'] is not None:
                 traverse (node['right'])
                 traverse (node['left'])

    traverse (tree)
    return result
```

```
tree_data = {
  'value': 2,
  'left': None
  'right': {
    'value': -6,
    'left': {
      'value': 5,
      'left': None,
      'right': None
            },
     'right': {
       'value': -11,
       'left': None,
       'right': None
               }
             }
          },
   'right': {
     'value': 10,
     'left': None,
     'right': None
            }
         }

print (single_child (tree_data))

----------------------------------------------------------------
# Write a function that takes a binary tree and returns the total
   number of nodes in the tree:

def count_nodes (tree):
    result = list()

    def traverse (node):
        if node:
            result.append(node['value'])

----------------------------------------------------------------
# Scrieti o functie care interschimba recursiv subarborele stang cu subarborele drept intr-un arbore binar:

def swap (tree):
    if tree:
        aux = tree['left']
        tree['left'] = tree['right']
        tree['right'] = aux

        swap (tree['left'])
        swap (tree['right'])

def rsd (tree):
    if tree is not None:
        return [tree['value']] + rsd (tree['left']) + rsd (tree['right'])
    else:
        return []

tree_data = {
  'value': 2,
  'left': None
  'right': {
    'value': -6,
     'left': {
       'value': 5,
       'left': None,
       'right': None
              },
      'right': {
        'value': -11,
        'left': None,
        'right': None
                  }
               }
            },
   'right': {
      'value': 10,
      'left': None,
      'right': None
              }
           }

swap (tree_data)
print (rsd (tree_data))

  ----------------------------------------------------------
# Write a function that returns a list of all non-leaf nodes that are
   positive numbers in a binary tree:

def positive_number (tree):
    result = list()

    def traversal (tree):
        if tree:
            if tree['left'] is not None or tree['right'] is not None:
                if tree['value'] >= 0:
                    result.append (tree['value'])
                traversal (tree['right'])
                traversal (tree['left'])
        traversal (tree)
        return result

tree_data = {
  'value': 2,
  'left': None
  'right': {
     'value': -6,
     'left': {
       'value': 5,
       'left': None,
       'right': None
              },
      'right': {
        'value': -11,
        'left': None,
        'right': None
                  }
               }
            },
   'right': {
      'value': 10,
      'left': None,
      'right': None
```

```
              }
           }

print (positive_number (tree_data))

-------------------------------------------------------------
# Write a function that returns a set of all odd-numbered leaves in
   a binary tree:

def odd_numbered (tree):
    result = set()

    def traversal (tree):
        if tree:
            if tree['left'] is None and tree['right'] is None:
                if tree['value'] % 2 == 1:
                    result.add (tree['value'])
            traversal (tree['right'])
            traversal (tree['left'])
        traversal (tree)
        return result

tree_data = {
  'value': 2,
  'left': None
  'right': {
     'value': -6,
     'left': {
       'value': 5,
       'left': None,
       'right': None
              },
      'right': {
        'value': -11,
        'left': None,
        'right': None
                  }
               }
            },
   'right': {
      'value': 10,
      'left': None,
      'right': None
              }
           }

print (odd_numbered (tree))

-----------------------------------------------------------
# Implement a function resMax, which takes as parameters a list of
   functions and a list of numbers of the same length and returns
   the result with the maximum value between the call of each
   function and the number at the same index as it is:

# Example
# resMax ([lambda x : x + 1, lambda x : x * 10, lambda x : x - 3],
                 [4, -1, 10])     will return 7

def resMax (function_list, number_list):
    result = list()
    def traverse (function_list, number_list):
        if not function list:
            return 0
        result.append (function_list[0] (number_list[0]))
        traverse (function_list[1:], number_list[1:])
    traverse (function_list, number_list)
    return max (result)

print ( resMax ([lambda x : x + 1, lambda x : x * 10, lambda x : x - 3],
                         [4, -1, 10]))

------------------------------------------------------------
#  Varianta 2:
def resMax (example_functions, example_numbers):
    result = list (map (lambda example_functions,
                example_numbers:
                example_functions (example_numbers),
                example_functions, example_numbers))
    return max (result)

example_functions = [lambda x : x + 1, lambda x : x * 10,
                                    lambda x : x - 3]
example_numbers =  [1, -22, 3]
print (resMax (example_functions, example_numbers))

---------------------------------------------------------------
# Implement an applyFunction function, which takes as
   parameters a list of functions and a number x and returns a list
   containing the value of the functions in point x:

# Example: f(x) = 2 * x - 4
                  g(x) = x + 6
   applyFunction ([f, g], 7)    will return [10, 13]

def applyFunction (function_list, x):
    result = list()

    def traverse (function_list, x):
        if not function_list:
            return 0
        result.append (function_list[0](x))
        traverse (function_list[1:], x)

     traverse (function_list, x)
     return result

f = lambda x : 2 * x - 4
g = lambda x : x + 6
print (applyFunction ([f, g], 7))

-------------------------------------------------------------------
# That movie problem from exam:

from functools import reduce

def number_films (dictionary_rating, min_rating, max_rating):
    result = reduce (lambda acc, items: {**acc, items[0] : items[1]}
                if min_rating <= items[1] <= max_rating else acc,
               dictionary_rating.items(), {})
    return len (list (result.keys()))
```

```
def get_movies (movie_dictionary, dictionary_rating, min_rating,
                            maxrating):
    result = reduce (lambda acc, items: {**acc, items[0] : items[1]}
                 if min_rating <= items[1] <= max_rating else acc,
                dictionary_rating.items(), {})
    key_list = list (result.keys())
    new_result = reduce (lambda acc, items: {**acc, items[0] :
                          items[1]} if items[0] in key_list else acc,
                          movie_dictionary.items(), {})
    result_list = list (new_result.values())
    return result_list

movie_dictionary = {1: "Avatar 2", 2: "The Godfather", 3: "The Dark
                                   Knight"}
dictionary_rating = {1: 6.75, 2: 8.80, 3: 7.67}
print (number_films (dictionary_rating, 7.50, 10.00))
print (get_movies (movie_dictionary, dictionary_rating, 7.50,
          10.00))

----------------------------------------------------------------------
# Write a function that receives a dictionary of strings to integers
   and a list of strings and returns a set containing all values in
   dictionary that match the strings in the list:

# Input : {'aa': 5, 'bb': 7, 'ca': 6}, ['aa', 'bb', 'c']
# Output: {5, 7}

from functools import reduce

def check_occurrence (example_dict, example_list):
    result = reduce (lambda acc, items: {**acc, items[0] : items[1]}
                 if items[0] in example_list else acc,
                 example_dict.items(), {})
    return set (result.values())

example_dict = {'aa': 5, 'bb': 7, 'ca': 6}
example_list = ['aa', 'bb', 'c']
print (check_occurrence (example_dict, example_list))

-----------------------------------------------------------------------
# Using the reduce function, implement the map function that
   builds a dictionary where all values have been transformed using
   a given function as a parameter:

# Input : {'a': 5, 'b': 6, 'c': 7}
               lambda x : x + 1
# Output: {'a': 6, 'b': 7, 'c': 8}

def custom_map (example_dict, function):
    result = reduce (lambda acc, items: {**acc,
                 items[0] : function(items[1])}, example_dict.items(), {})
   return result

example_dict = {'a': 5, 'b': 6, 'c': 7}
function = lambda x : x + 1
print (custom_map (example_dict, function))

------------------------------------------------------------
# Implement exists and for_all using reduce for dictionaries. They
```

```
   take as parameters a boolean function (condition) of key and
  value (expressing the condition) and the dictionary to be searched:

# Input: dictionary: {'a': 5, 'b': 7, 'c': 1}
             condition: value >= 5
# Output: exists: True
               for_all: False

from functools import reduce

def exists (example_dict, condition):
    result = reduce (lambda acc, items: {**acc, items[0] : items[1]}
                 if condition (items[1]) else acc, example_dict.items(), {})
    if not result:
        return False
    else:
        return True

def for_all (example_dict, condition):
    result = reduce (lambda acc, items: {**acc, items[0] : items[1]}
                 if condition (itmes[1]) else acc, example_dict.items(), {})
    if result == example_dict:
       return True
    else:
       return False

example_dict = {'a': 5, 'b': 7, 'c': 1}
condition = lambda x : x >= 5
print ("exists:", exists (example_dict, condition))
print ("for all:", for_all (example_dict, condition))

-----------------------------------------------------------------
# Using the reduce function, implement the filter function that
   creates a new dictionary with only the pairs from the given
   dictionary which satisfy a given condition:

# Input: dictionary: {'a': 5, 'b': 7, 'c': 1}
             condition: value >= 5
# Output: {'a': 5, 'b': 7}

from functools import reduce

def custom_filter (example_dict, condition):
    result = reduce (lambda acc, items: {*acc, items[0] : items[1]}
                 if condition (items[1]) else acc, example_dict.items(), {})

example_dict = {'a': 5, 'b': 7, 'c': 1}
condition = lambda x : x >= 5
print (custom_filter (example_dict, condition))

--------------------------------------------------------------------
# Implement a standard partition function which takes as
   parameters a boolean function f and a set s and returns a pair of
   sets, with the elements of s satisfying the condition f:

# Input: lambda x : x % 2 == 0
              {1, 2, 3, 4}
# Output: ({2, 4}, {1, 3})

def partition (condition, example_set):
    result = (set(), set())
    copy_list = list (example_set)

    def traverse (condition, example_list):
        if not example_list:
            return 0
        if (condition (example_list[0])):
            result[0].add (example_list[0])
        else:
            result[1].add (example_list[0])
       traverse (condition, example_list[1:])
    traverse (condition, copy_list)
    return result

condition = lambda x : x % 2 == 0
example_set = {1, 2, 3, 4}
print (partition (condition, example_set))

-------------------------------------------------------------
# Cake problem from exam:

def number_cakes (price_dictionary, first_limit, second_limit):
    result = len (list (filter (lambda x: first_limt <= x <= second_limit,
                 price_dictionary.values())))
   return result

def get_cakes (dictionary_cakes, price_dictionary, first_limit,
                         second_limit):
   result = list (map (lambda x: dictionary_cakes [x],
                list (filter (lambda x: first_limit <= price_dictionary[x]
                <= second_limit, price_dictionary.keys()))))
   return result

dictionary_cakes = {1: "Amandina", 2: "Savarina", 3: "Fruit cake",
                                 4: "Chocolate cake"}
price_dictionary = {1: 16.25, 2: 12.50, 3: 14.00, 4: 13.70}
print (number_cakes (price_dictionary, 12.00, 13.99))
print (get_cakes (dictionary_cakes, price_dictionary, 12.00, 13.99))

-----------------------------------------------------------------
# Using reduce, implement a function called count, which takes a
   function f and a list as a parameter and returns the number of
   elements for which the function f is true:

from functools import reduce

def count (condition, example_list):
    result = reduce (lambda acc, items: acc + 1 if condition (items)
                 else acc, example_list, 0)
   return result

example_list = [1, 2, 3, 4, 5]
condition = lambda x : x % 2 == 0
print (count (condition, example_list))

-------------------------------------------------------------------
# Implement a function sum which calculates the sum of all
   elements (assumed integers) for which the function f it's true:

from functools import reduce
```

```
def sum (condition, example_list):
    result = reduce (lambda sum, element: sum + element
                if condition (element) else sum, example_list, 0)
   return result

example_list = [1, 2, 3, 4, 5]
condition = lambda x : X % 2 == 0
print (sum (condition, example_list))

-----------------------------------------------------------------------
# Write a function that removes consecutive duplicates: takes a
   list as parameter and constructs a list in which all sequences of
   equal consecutive elements have been replaced by a single
   element:

def remove_consecutive_elem (example_list):
    result = list()
    def traverse_list (example_list):
        if len (example_list) == 1:
            return 0
        if not (example_list[0] == example_list [1]):
            result.append (example_list[0])
        traverse_list (example_list[1:])
    traverse_list (example_list)
    return result

example_list = [1, 2, 3, 3, 4, 4, 4, 5]
print (remove_consecutive_elem (example_list))

------------------------------------------------------------------------
# Same problem using reduce:

from functools import reduce

def remove_consecutive_elem (example_list):
    result = reduce (lambda acc, items: acc + [items]
                if item not in acc else acc, example_list, [])
   return result

example_list = [1, 2, 3, 3, 4, 4, 4, 5]
print (remove_consecutive_elem (example_list))

------------------------------------------------------------------------
# Write a function that takes a list of pairs (of a specified type) and
   returns a set containing the elements of the first position in each
  pair:

# Input: [(1, 2), (3, 4)]
# Output: {1, 3}

from functools import reduce

def build_set (example_list):
    result = reduce (lambda acc, items: acc.add(items[0]) or acc,
                example_list, set())
    return result

example_lits = [(1, 2), (3, 4)]
print (build_set (example_list))
```

```
-----------------------------------------------------------------
# Implement the standard filter function that takes as parameters
  a boolean function f and a set s and returns the set of elements
  in s that satisfy the condition f:

# Input: lambda x : x % 2 == 0
         {1, 2, 3, 4}
# Output: {2, 4}

from functools import reduce

def set_filter (f, example_set):
    result = reduce (lambda acc, items: acc.add(items) or acc
                if f(items) else acc, example_set, set())
    return result

f = lambda x : x % 2 == 0
example_set = {1, 2, 3, 4}
print (set_filter (f, example_set))

----------------------------------------------------------------------
# Implement the standard partition function which takes as
  parameters a boolean function f and a set s and returns a pair
  of sets, with the elements of s satisfying and not satisfying
  the condition f:

# Input: lambda x : x % 2 == 0
         {1, 2, 3, 4}
# Output: ({2, 4}, {1, 3})

from functools import reduce

def split_filter (f, example_set):
    result = reduce (lambda acc, items: (acc[0].add (items) or acc[0],
                acc[1]) if condition (items) else (acc[0],
               acc[1].add (items) or acc[1]), example_set, (set(), set()))
    return result

f = lambda x : x % 2 == 0
example_set = {1, 2, 3, 4}
print (split_filter (f, example_set))

---------------------------------------------------------------------
# Same problem, without using reduce:

def split_filter (condition, example_list):
    result = (set(), set())
    set_list = list (example_set)

    def traverse_set (condition, set_list):
        if not set_list:
            return 0
        if condition (set_list[0]):
            result[0].add (set_list[0])
        else:
            result[1].add (set_list[0])
        traverse_set (condition, set_list[1:])
    traverse_set (condition, set_list)
    return result
```