Tartalomjegyzék

- 1 Problem 1: Variadic Intersect
- 2 Problem 2: Quadratic
- 3 Problem 3: Matrix Error
- 4 Problem 4: Modulus Matrix
- 5 Note:

Problem 1: Variadic Intersect

Write a variadic python function which calculates the intersection of arbitrary many sets (lists). The function can have any number of arguments, all of which is a list. Return the intersection of the input lists as an other list.

You will have at least one list in the input so you don't have to calculate the intersection of zero lists. But any of the input lists can be an empty list.

Problem 2: Quadratic

Write a python function that can solve any quadratic equation in the form: ax^2 + bx + c = 0, also handle the special case when some of the coefficients are zero. Such an equation is defined by the coefficients of the quadratic polynomial on the left-hand-side (a, b,c).

From these coefficients, calculate every real solution and return them as a list, not longer than 2.

Return None if all the coefficients are zero, because in that case you would have infinite number of solutions.

If there is no solution or only complex ones, then return an empty list.

If there is a real root with multiplicity 2, then return a one long list.

The only problem is that mathematicians cannot use a uniform notation. Some write the polynomial terms in ascending order, some write it the other way around:

```
ax^2 + bx + c
c + bx + ax^2
```

And also the zero coefficients can be left out:

```
ax^2 + c = 0
```

To this end, you have to write the function with 3 optional keyword arguments, with the default value of 0. In this way, one can use this function with any order of coefficients as long as the quadratic term is a, the linear is b and the constant is c. Also, one can leave out coefficients if they are 0.

Problem 3: Matrix Error

Implement the matrix operations of the Matrix class: __add__, __sub__ and __mul__, but mind the size of the matrices!

If the matrices do not fit, then raise (throw) a ValueError exception. Otherwise evaluate the operations as in the previous matrix exercise.

Use the previous __str__ method also. CODE class Matrix:

```
def __init__(self, m):
    pass

def __add__(self, other):
    pass

def __sub__(self, other):
    pass

def __mul__(self, other):
    pass

def __str__(self):
    pass
```

m1 = Matrix([[1, 2], [3, 4]])

m2 = Matrix([[1, 0, 1], [0, 2, 0]])

print(a + b) # ValueError

print(a - b) # ValueError

print(a * b)

```
1   4   1
3   8   3
```

Problem 4: Modulus Matrix

Combine two earlier classes to implement a class which stores and calculates with matrices where the elements are Modulus objects. Use inheritance: the Matrix class should be the parent of the new ModulusMatrix class.

Depending on the Matrix class you have to re-implement one or more methods, but you have to re-implement the __init__ method for sure (use super).

If you write the Matrix class cleverly then you can use the inherited methods and you don't have to re-implement much. But you can rewrite the whole class, too.

If the dimension of the matrices don't match then raise a ValueError. Also, if any of the matrix element's modulo doesn't match, then raise a ValueError too.

For example:

```
m5 = ModulusMatrix([[Modulus(5, 2), Modulus(5, 1)], [Modulus(5, 3), Modulus(5, 2)]])
m6 = ModulozMatrix([[Modulus(5, 1), Modulus(5, 5)], [Modulus(5, 2), Modulus(5, 1)]])

print(m5)
print(m6)
print(m5 + m6)
```

```
print(m5 - m6)
print(m5 * m6)
```

should print:

```
 2   1
 3   2

 1   0
 2   1

 3   1
 0   3

 1   1
 1   1

 4   1
 2   2
```

Hint: Use exceptions in the Modulus class too. Raise an exception if the modulo of the operands are not the same. If you do that then you only have to mind the size of the matrices.

Note:

Problem 1-3: 1 point

Problem 4: 2 point