Mandatory Problem 1B

Computations in Elementary Number Theory

By Vegard Berge

Task 1

Implement arithmetic operations with polynomials g(x), f(x) modulo a prime number p :

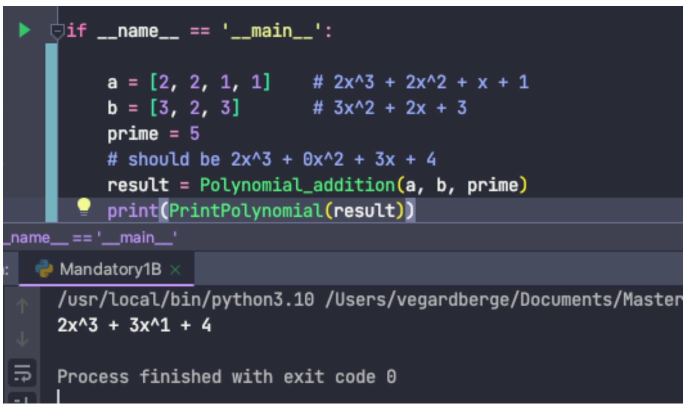
- Addition $f(x) + g(x)$

```
def Polynomial_addition(f_x: list, g_x: list, p: int) -> list:
    """
    Compute the sum of two polynomials f_x & g_x
    h(x) = sum(f_x, g_x)
    :param f_x:
    :param g_x:
    :param p:
    :return polynomial h(x):
    """
    if len(f_x) > len(g_x):
        g_x = [0] * (len(f_x) - len(g_x)) + g_x
    elif len(g_x) > len(f_x):
        f_x = [0] * (len(g_x) - len(f_x)) + f_x
    assert (len(f_x) == len(g_x))
    return [(x + y) % p for x, y in zip(f_x, g_x)]
```

```
if __name__ == '__main__':

    a = [2, 2, 1, 1]    # 2x^3 + 2x^2 + x + 1
    b = [3, 2, 3]       # 3x^2 + 2x + 3
    prime = 5
    # should be 2x^3 + 0x^2 + 3x + 4
    result = Polynomial_addition(a, b, prime)
    print(PrintPolynomial(result))
```

Output of the program:



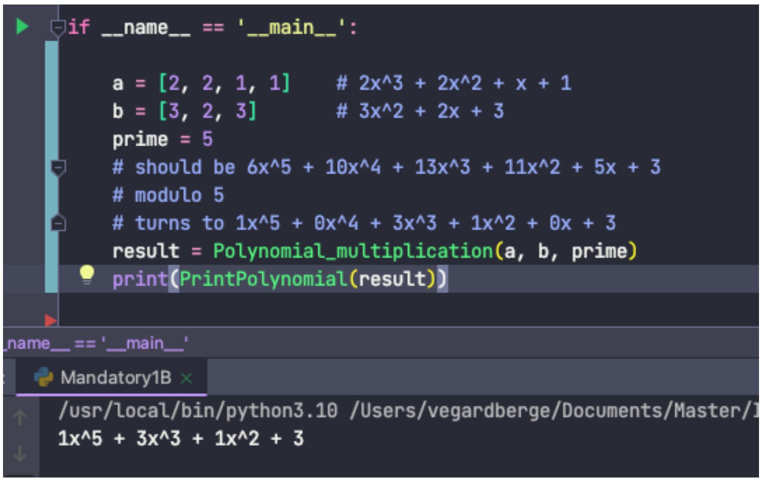
- Multiplication $f(x) * g(x)$

```
def Polynomial_multiplication(f_x: list, g_x: list, p: int) -> list:
    """
    Compute the product of two polynomials f_x & g_x modulo p
    :param f_x:
    :param g_x:
    :param p:
    :return:
    """
    init = [0] * (len(f_x) + len(g_x) - 1)
    for f in range(len(f_x)):
        for g in range(len(g_x)):
            init[f + g] = (init[f + g] + (f_x[f] * g_x[g]) % p) % p
    return init
```

```
if __name__ == '__main__':

    a = [2, 2, 1, 1]    # 2x^3 + 2x^2 + x + 1
    b = [3, 2, 3]       # 3x^2 + 2x + 3
    prime = 5
    # should be 6x^5 + 10x^4 + 13x^3 + 11x^2 + 5x + 3
    # modulo 5
    # turns to 1x^5 + 0x^4 + 3x^3 + 1x^2 + 0x + 3
    result = Polynomial_multiplication(a, b, prime)
    print(PrintPolynomial(result))
```

Output of the program:



- Division with remainder $g(x) = q(x) * f(x) + r(x), 0 \leq deg\ r(x) < deg\ f(x)$

```
def PolynomialDivision(f_x: list, g_x: list, p: int):
    """
    Compute the quotient q and remainder r of polynomial f
    divided by polynomial g(x) !=0 modulo n
    :param f_x:
    :param g_x:
    :param p:
    :return quotient & remainder:
    """

    f_x, g_x, tmp_G = format_lists(f_x, g_x)

    # Copy input arr. so we can change them
    f_x = copy.deepcopy(f_x)
    g_x = copy.deepcopy(g_x)

    quotient = []
    _round = 0
    while True:
        # find index of element with the highest order != 0
        highest_order_elemt_index = get_highest_order_elem(f_x)
        # order_of_highest_elemt = len(f_x) - highest_order_elemt_index

        if getDegre(f_x) == 0 and f_x[-1] == 0:
            return quotient, f_x
        if getDegre(f_x) < getDegre(g_x):
            # can't divide this element, rest is remainder
            return quotient, f_x
        # Find mult. inverse of cg mod n
        factor = [x for x in range(p) if ((x * g_x[get_highest_order_elem(g_x)]) % p) == f_x[
            highest_order_elemt_index]][0]

        # Add quotient to arr. tuple of (coefficient, order)
        quotient.append((factor, abs(get_highest_order_elem(f_x) - get_highest_order_elem(g_x))))

        # multiply factor to f_x
        factor_order = quotient[_round][1]
        y = 0
        for g_i in range(len(tmp_G)):
            g_i_z = -(tmp_G[g_i] * factor) % p
            order_g_i = len(tmp_G) - g_i
            f_x[len(f_x) - (order_g_i + factor_order)] = (f_x[len(f_x) - (order_g_i + factor_order)] + g_i_z) % p
            y += 1
        _round += 1
```

```
if __name__ == '__main__':

    a = [2, 2, 1, 1]    # 2x^3 + 2x^2 + x + 1
    b = [3, 2, 3]       # 3x^2 + 2x + 3
    prime = 5
    # 2x^3 + 2x^2 + x + 1 \div 3x^2 + 2x + 3
    # Should be: (4x+1) + ((3x +2) / (2x^3 + 2x^2 + x + 1))
    quot, rem = PolynomialDivision(a, b, prime)
    print(f"Quotients: {PrintQuotient(quot)} remainders: {PrintPolynomial(rem)}")
```

Output of the program:

- $gcd(f(x), g(x))$ with Euclidean Algorithm

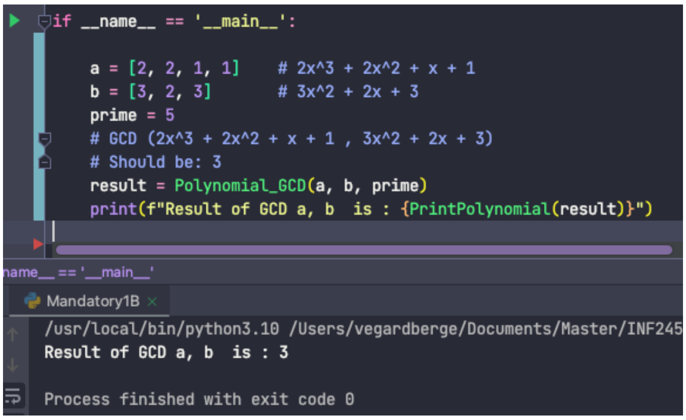
```
def Polynomial_GCD(f_x: list, g_x: list, p: int) -> list:
    """
    GCD for polynomials
    :param f_x:
    :param g_x:
    :param p:
    :return Returns  the gcd of the two polynomials f(x) & g(x) :
    """
    f = copy.deepcopy(f_x)
    g = copy.deepcopy(g_x)

    q, r = PolynomialDivision(f, g, p)
    g = [0] * (len(r) - len(g)) + g
    while True:
        if getDegre(r) == 0 and r[-1] == 0:
            break
        q, r_prime = PolynomialDivision(g, r, p)
        g = r
        r = r_prime
    return g
```

```
if __name__ == '__main__':

    a = [2, 2, 1, 1]     # 2x^3 + 2x^2 + x + 1
    b = [3, 2, 3]        # 3x^2 + 2x + 3
    prime = 5
    # GCD (2x^3 + 2x^2 + x + 1 , 3x^2 + 2x + 3)
    # Should be: 3
    result = Polynomial_GCD(a, b, prime)
    print(f"Result of GCD a, b  is : {PrintPolynomial(result)}")
```

Output of the program:



- Exponentiation $g(x)^a \; mod \; f(x)$ for a positive integer a with binary method

```
def Polynomial_BinaryExponentiation(f_x: list, alfa: int, g_x: list, p: int) -> list:
    """
    Calculates for this polynomial f, some modulo polynomial g and a positive integer a
    (f(x))^a mod g(x)
    :param f_x:
    :param alfa:
    :param g_x:
    :param p:
    :return polynomial h(x) = (f(x))^a mod g(x):
    """
    res = [1]

    if alfa == 0:
        return res
    if alfa == 1:
        return PolynomialDivision(f_x, g_x, p)[1]

    q, h = PolynomialDivision(f_x, g_x, p)

    while alfa > 0:
        if alfa % 2 == 1:
            tmp = Polynomial_multiplication(res, h, p)
            tmp = [0] * (len(g_x) - len(tmp)) + tmp
            q, res = PolynomialDivision(tmp, g_x, p)

        alfa = alfa // 2
        h = Polynomial_multiplication(h, h, p)
        h = [0] * (len(g_x) - len(h)) + h
        q, h = PolynomialDivision(h, g_x, p)

    return res
```

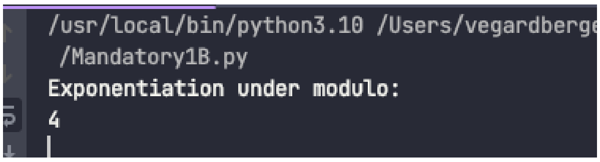
Demonstration & output of this program comes in the next point:

- Compute $(x+1)^{15399} \; mod \; x^2 + x + 1$ and modulo $p = 5$

```
f = [1, 1, 1]  # x^2 + x + 1
g = [1, 1]  # x + 1
a = 15399
PRIME = 5
print("Exponentiation under modulo: ")
```

```
residue = Polynomial_BinaryExponentiation(g, a, f, PRIME)
print(PrintPolynomial(residue))
```

Output of the program:



Task 2

- Implement the algorithm to find a solution $a(mod\ p)$ to the congruence f(x) $\equiv 0(mod\ p)$. With this method find all solutions to the congruence for
$$p = 113,\ f(x) = x^4 + 70x^3 + 89x^2 + 81x + 96$$

```
def Polynomial_find_root(f_x: list, prime):
    """
    Find the solutions fo the congruence f(x) = 0 mod p of degree d

    Output: Residue r mod p, s.t f(r) congruent 0 mod p
    :param prime:
    :param f_x:
    :param p:
    :return:
    """
    res = None
    ###########################
    # STAGE 1
    # compute x^p mod f(x)
    # create polynomial x of degree p
    # h(x) congruent with x^P - x mod f(x)
    # simply: h_prime = x^P mod f(x) wiht bin. exp
    h_prime = Polynomial_BinaryExponentiation([1, 0], prime, f_x, prime)
    # now:
    # h(x) congruent x^P - X mod f(x)
    h_x = Polynomial_addition(h_prime, [-1, 0], prime)

    # g(x) <- gcd(h(x), f(x)
    g_x = Polynomial_GCD(h_x, f_x, prime)

    #############################
    # STAGE 2
    if g_x[getDegre(g_x)] != 1:
        # make g(x) monic
        inv_largest = pow(g_x[get_highest_order_elem(g_x)], -1, prime)
        g_x = [(g * inv_largest) % prime for g in g_x]

    if getDegre(g_x) == 0:
        # has no solution
        return res
    if getDegre(g_x) == 1:
        # g(x) = x - r, r is solution
        return -(g_x[-1]) % prime

    while True:
```

```
        assert getDegre(g_x) >= 2

        # new random elem
        b = random.randint(2, prime)
        # v(x) <- gcd(g(x), (x+b)^ ((p - 1)/2)) - 1

        # simplify expression gcd(g(x), theta) where
        # theta = (x+b)^ ((p - 1)/2)) - 1
        theta = [1, (b + (prime - 1)) % prime]
        # Binary Exponentiation to calculate expression
        tmp_x = Polynomial_BinaryExponentiation(theta, ((prime - 1) // 2), f_x, prime)
        v_x = Polynomial_GCD(g_x, tmp_x, prime)

        # if v(x) = 1 or g(x) repeat, new b
        if v_x == [1, 0] or v_x == g_x:
            continue
        # if v(x) = x - r, then r is sol => return
        elif getDegre(v_x) == 1:
            return -(pow(v_x[-2], -1, prime) * v_x[-1]) % prime

        # if 2 <= deg v(x) < deg g(x), set g(x) <- v(x) or g(x) / v(x) of smallest deg
        elif 2 <= getDegre(v_x) < getDegre(g_x):
            q, r = PolynomialDivision(g_x, v_x, prime)
            if q[0][1] < getDegre(v_x):
                g_x = [quot[0] for quot in q]
            else:
                g_x = v_x
```

In order to produce all roots with some probability I implemented this function. This also checks that the possible root found, actually is a root of $f(x)$

For a given finite polynomial $f(x)$ in Z/p, where p is prime, we calculate the roots r_i such that $f(r_i) = 0$. For a polynomial with degree d , there are at least d roots counting multiplicity.

```
def Polynomial_FindAll_roots(f_x: list, p: int, probability=10) -> list:
    assert not all(e == 0 for e in f_x)
    POLY_F = copy.deepcopy(f_x)
    ROOTS = []
    # Run root test deg(f) * 10 times if otherwise not specified
    for _ in range(len(f_x) * probability):
        un_verified_root = Polynomial_find_root(POLY_F, p)
        if un_verified_root:
            POLY_F = Polynomial_scale(POLY_F, pow(POLY_F[get_highest_order_elem(POLY_F)], -1, p), p)
            root = (p - un_verified_root) % p
            q, r = PolynomialDivision(POLY_F, [1, root], p)
            assert all(e == 0 for e in r)
            if un_verified_root not in ROOTS:  ROOTS.append(un_verified_root)
    return ROOTS
```

```
if __name__ == '__main__':
    f_x = [1, 70, 89, 81, 96]   # x^4 + 70x^3 + 89x^2 + 81x + 96
    prime = 113
    roots = Polynomial_FindAll_roots(f_x, prime)
    print(roots)
```

```
if __name__ == '__main__':
    f_x = [1, 70, 89, 81, 96]    # x^4 + 70x^3 + 89x^2 + 81x + 96
    prime = 113
    roots = Polynomial_FindAll_roots(f_x, prime)
    print(roots)
```

```
Mandatory1B
/usr/local/bin/python3.10 /Users/vegardberge/Documents/Master/INF245/
[71, 13]
```

Here are some of the extra functions that I implemented along the way. These are more for the “python” functionality than the algorithms.

```
import copy, random

def Polynomial_scale(f_x: list, c, prime):
    assert (isinstance(c, int))
    return [(f * c) % prime for f in f_x]


def get_highest_order_elem(l: list) -> int:
    if len([x for x in l if x == 0]) == len(l):
        return len(l)
    else:
        return [i for i, e in enumerate(l) if e != 0][0]


def getDegre(l: list):
    x = 0
    for i, e in enumerate(l):
        if e != 0:
            x = (len(l)) - i - 1
            break
    return x

def PrintPolynomial(h_x: list):
    out = ""
    for i, e in enumerate(h_x):
        if i == len(h_x) - 1:
            out += str(e)
        elif e == 0:
            continue
        else:
            out += f"{e}x^{(len(h_x) - 1) - i} + "
    return out

def PrintQuotient(q_x: list):
    out = ""
    for i, e in enumerate(q_x):
        out += f"{e[0]}x^{(len(q_x)-(i+1))} + "
    return out

def format_lists(f_x, g_x):
    strip_index = getDegre(g_x) + 1 if getDegre(g_x) > getDegre(f_x) else getDegre(f_x) + 1
    g_x = g_x[-strip_index:]
    f_x = f_x[-strip_index:]
    tmp_G = g_x
    g_x = [0] * (len(f_x) - len(g_x)) + g_x
    return f_x, g_x, tmp_G
```