МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н.Г. ЧЕРНЫШЕВСКОГО»

Кафедра теоретических основ компьютерной безопасности и криптографии

Расширенный алгоритм Евклида для полиномов над полем

Обобщённый алгоритм Евклида для полиномов над целыми числами

ЛАБОРАТОРНАЯ РАБОТА

студента 4 курса 431 группы

специальности 10.05.01 Компьютерная безопасность

факультета компьютерных наук и информационных технологий

Серебрякова Алексея Владимировича

Научный руководитель

доцент, к. п. н. ___________ А. С. Гераськин

подпись, дата

Саратов 2022

XEA-P. Расширенный алгоритм Евклида для полиномов над полем (Extended Euclidean Algorithm for Polynomials over a Field)

Вход: $p_1(x), p_2(x) \in J[x]$, $p_2(x) \neq 0$, $m = \deg[p_1(x)] \geqslant \deg[p_2(x)] = n$; J — поле.

Выход: $p_h(x), f(x), g(x) \in J[x]$, такие, что $\deg[f(x)] < \deg[p_1(x)] - \deg[p_h(x)]$, $\deg[g(x)] < \deg[p_2(x)] - \deg[p_h(x)]$ и $p_h(x) = \gcd[p_1(x), p_2(x)] = p_1(x)g(x) + p_2(x)f(x)$.

1. [Инициализация] $[p_0(x), p_1(x)] := [p_1(x), p_2(x)]$;
$[g_0(x), g_1(x)] := (1, 0)$;
$[f_0(x), f_1(x)] := (0, 1)$.

2. [Основной цикл] Пока $p_1(x) \neq 0$ выполнять
$\{q(x) := \mathbf{PDF}[p_0(x), p_1(x)]$;
$[p_0(x), p_1(x)] := [p_1(x), p_0(x) - p_1(x)q(x)]$;
$[g_0(x), g_1(x)] := [g_1(x), g_0(x) - g_1(x)q(x)]$;
$[f_0(x), f_1(x)] := [f_1(x), f_0(x) - f_1(x)q(x)]\}$.

3. [Выход] Вернуть $[p_h(x), g(x), f(x)] := [p_0(x), g_0(x), f_0(x)]$.

Анализ времени работы алгоритма ***XEA-P***. Ясно, что время работы этого алгоритма доминируется временем выполнения шага 2.

Поскольку мы работаем в поле, первое выполнение алгоритма **PDF** требует времени $O[n(m - n + 1)]$, где $m = \deg[p_1(x)]$, $n = \deg[p_2(x)]$, $m \geqslant n$, и $m - n = \deg[q(x)]$; кроме того, первое выполнение каждого из полиномиальных умножений $p_1(x)q(x)$, $g_1(x)q(x)$ и $f_1(x)q(x)$ также происходит за время $O[n(m - n + 1)]$, и оно явно доминирует время выполнения каждого из соответствующих полиномиальных вычитаний. Поэтому время первого выполнения шага 2 равно $O[n(m - n + 1)]$, что доминирует и время всех его последующих выполнений (проверьте это).

Итак, мы можем сказать, что в худшем случае каждое выполнение шага 2 происходит за время $O[n(m - n + 1)]$, а поскольку может быть не более n выполнений этого шага, мы имеем

$$t_{\mathbf{XEA\text{-}P}}[p_1(x), p_2(x)] = O[n^2(m - n + 1)].$$

```
[alse0722@alse0722-ms7a34 coding_teory]$ /bin/python /home/alse0722/Desktop/univer/coding_teory/fin/n14.py
DIV
[1, 5, 10, 10, 5, 1] / [1, 2, 1] ->
quot: [1.0, 3.0, 3.0, 1.0], rem: [0]

DIV
[5, 16, 10, 22, 7, 11, 1, 3] / [1, 2, 1, 3] ->
quot: [5.0, 1.0, 3.0, 0.0, 1.0], rem: [0.0, 5.0]

DIV
[1, 0, -1] / [1, -1] ->
quot: [1.0, 1.0], rem: [0]

DIV
[1, -2, 1] / [1, -1] ->
quot: [1.0, -1.0], rem: [0]

GCD [1, -2, 1] / [1, -1]
[1, -1]
GCDEX [1, 7, 6, 6] / [1, 1, 2]
([1, 1], [2, 2, 2], [1])
[alse0722@alse0722-ms7a34 coding_teory]$
```

Код программы:

```
import warnings
from sympy.polys.galoistools import gf_monic
from sympy.polys.galoistools import gf_div
from sympy.polys.galoistools import gf_sub_mul
from sympy.polys.galoistools import gf_mul_ground
from sympy.polys.domains import ZZ
import numpy.core.numeric as NX
from numpy.lib.twodim_base import diag
from numpy.linalg import eigvals
from numpy.core import (hstack)
from sympy import rem
from sympy import poly
from sympy.abc import x
import numpy as np
from sympy import *
x = symbols('x')


def normalize(poly):
    while poly and poly[-1] == 0:
        poly.pop()
    if poly == []:
        poly.append(0)


def poly_divmod(num, den):
    # Create normalized copies of the args
    num = num[:]
    normalize(num)
    den = den[:]
    normalize(den)

    if len(num) >= len(den):
        # Shift den towards right so it's the same degree as num
        shiftlen = len(num) - len(den)
        den = [0] * shiftlen + den
    else:
        return [0], num

    quot = []
    divisor = float(den[-1])
    for i in range(shiftlen + 1):
        # Get the next coefficient of the quotient.
        mult = num[-1] / divisor
        quot = [mult] + quot

        # Subtract mult * den from num, but don't bother if mult == 0
        # Note that when i==0, mult!=0; so quot is automatically normalized.
        if mult != 0:
            d = [mult * u for u in den]
            num = [u - v for u, v in zip(num, d)]

        num.pop()
        den.pop(0)

    normalize(num)
    return quot, num

def degree(f):
    try:
        return len(f)-1
```

```
    except:
        return 0

def copy_list(f):
    new_poly = []
    for i in range(len(f)):
        new_poly.append(f[i])
    return new_poly

def lead(f):
    try:
        return f[len(f) - 1]
    except:
        return 0

def check(f):
    new_poly = copy_list(f)
    if len(new_poly) == 0:
        return 0
    elif new_poly[len(new_poly)-1] == 0:
        new_poly.pop()
        return check(new_poly)
    else:
        return f

def multiply_polynomials(f, g):
    new_poly = []
    for i in range(len(f) * len(g)):
        new_poly.append(0)
    for i in range(len(f)):
        for j in range(len(g)):
            index = i+j
            new_poly[index] += f[i] * g[j]
    new_poly = check(new_poly)
    return new_poly

def convert_int(x):
    return [x]

def polynomial_subtraction(f, g):
    # subtracting g from f
    new_poly = []
    if type(f) == int:
        f = convert_int(f)
    if type(g) == int:
        g = convert_int(g)
    if len(f) > len(g):
        for i in range(len(g)):
            new_poly.append(f[i] - g[i])
        for j in range(len(g), len(f)):
            new_poly.append(f[j])
    elif len(f) == len(g):
        for i in range(len(f)):
            new_poly.append(f[i] - g[i])
    else:
        for i in range(len(f)):
            new_poly.append(f[i] - g[i])
        for j in range(len(f), len(g)):
            new_poly.append(-g[j])
    return new_poly

def divide_polynomials(f, g):
    if degree(g) == 0 and g[0] == 0:
```

```
        warnings.warn("g cannot be the 0 polynomial")
    q = [0]
    r = f
    while r != 0 and degree(r) >= degree(g):
        ti = []
        t = float(lead(r)) / float(lead(g))
        x = degree(r) - degree(g)
        while degree(q) < x:
            q.append(0)
        while degree(ti) < x:
            ti.append(0)
        if x == 0:
            q.append(0)
            ti.append(0)
        q[x] = t
        ti[x] = t
        r = polynomial_subtraction(r, multiply_polynomials(ti, g))

        r = check(r)
    return (q, r)


def convert_to_monic(polynomial):
    if lead(polynomial) != 1:
        new_poly = []
        for i in range(len(polynomial)):
            new_poly.append(polynomial[i]/lead(polynomial))
        return new_poly
    else:
        return polynomial


def gcd(f, g):
    if degree(g) == 0 and g[0] == 0:
        warnings.warn("g cannot be the 0 polynomial")
    if degree(f) >= degree(g):
        ma = f
        mi = g
    else:
        ma = g
        mi = f
    q, r = divide_polynomials(ma, mi)
    if r == 0:
        return mi
    else:
        r1 = convert_to_monic(r)
        return gcd(mi, r1)


def gcdex(f, g, p, K):

    if not (f or g):
        return [K.one], [], []

    p0, r0 = gf_monic(f, p, K)
    p1, r1 = gf_monic(g, p, K)

    if not f:
        return [], [K.invert(p1, p)], r1
    if not g:
        return [K.invert(p0, p)], [], r0

    s0, s1 = [K.invert(p0, p)], []
```

```
    t0, t1 = [], [K.invert(p1, p)]

    while True:
        Q, R = gf_div(r0, r1, p, K)

        if not R:
            break

        (lc, r1), r0 = gf_monic(R, p, K), r1

        inv = K.invert(lc, p)

        s = gf_sub_mul(s0, s1, Q, p, K)
        t = gf_sub_mul(t0, t1, Q, p, K)

        s1, s0 = gf_mul_ground(s, inv, p, K), s1
        t1, t0 = gf_mul_ground(t, inv, p, K), t1

    return s1, t1, r1


def test(num, den):
    print("%s / %s ->" % (num, den))
    q, r = poly_divmod(num, den)
    print("quot: %s, rem: %s\n" % (q, r))
    return q, r

print("DIV")
num = [1, 5, 10, 10, 5, 1]
den = [1, 2, 1]
test(num, den)

print("DIV")
num = [5, 16, 10, 22, 7, 11, 1, 3]
den = [1, 2, 1, 3]
test(num, den)

print("DIV")
test([1, 0, -1], [1, -1])

print("DIV")
test([1, -2, 1], [1, -1])

print("GCD [1, -2, 1] / [1, -1]")
print(gcd([1, -2, 1], [1, -1]))
f = [1, 7, 6, 6]
g = [1, 1, 2]

print("GCDEX [1, 7, 6, 6] / [1, 1, 2]")
print(gcdex(f, g, 3, ZZ))
```