СПЕЦІАЛЬНІ РОЗДІЛИ

ОБЧИСЛЮВАЛЬНОЇ МАТЕМАТИКИ

КОМП’ЮТЕРНИЙ ПРАКТИКУМ No1

Багаторозрядна арифметика

1. Мета роботи

Отримання практичних навичок програмної реалізації багаторозрядної арифметики;

ознайомлення з прийомами ефективної реалізації критичних по часу ділянок програмного коду та

методами оцінки їх ефективності.

2. Завдання до комп’ютерного практикуму

Реалізувати операції:

1) обчислення НСД та НСК двох чисел;

2) додавання чисел за модулем;

3) віднімання чисел за модулем;

4) множення чисел та піднесення чисел до квадрату за модулем;

5) піднесення числа до багаторозрядного степеня d по модулю n.

Перевірити коректність операцій та знайти визначити середній час виконання

3. Код :

# lab 2

    def is\_even(self):

        if self.hex\_charset.index(self.number[-1]) % 2 == 0:

            return True

        return False

    def is\_2\_pow(self):

        if self.to\_binary().count('1') == 1:

            return True

        return False

    def \_\_mod\_\_(self, value):

        if str(value) == '0' or value > self:

            return self

        if value.is\_2\_pow():

            result = self.to\_binary()[-value.elder\_bit():]

            return LongInt(self.from\_binary('0' \* (self.length \* 4 - len(result)) + result))

        if self.elder\_bit() < value.elder\_bit()\*2:

            k = value.elder\_bit() + 1

            r = ((LongInt('1') << (2\*k))/value)[0]

            t = self - (((self\*r) >> (2\*k))\*value)

            if t < value:

                return t

            return t - value

        else:

            return self - (self/value)[0]\*value

    def gcd(self, value):

        if str(self) == '0' or str(value) == '0':

            return max(self, value)

        a = self

        b = value

        d = 0

        while a.is\_even() and b.is\_even():

            a = a >> 1

            b = b >> 1

            d += 1

        while a.is\_even():

            a = a >> 1

        while str(b) != '0':

            while b.is\_even():

                b = b >> 1

            if a > b:

                a, b = b, a

            b = b-a

        return a << d

    def lcm(self, value):

        return ((self \* value) / (self.gcd(value)))[0]

    def add\_mod(self, value, mod):

        return (self+value) % mod

    def sub\_mod(self, value, mod):

        return (self-value) % mod

    def mul\_mod(self, value, mod):

        return ((self)\*(value)) % mod

    def pow\_barrett(self, value, mod):

        if str(value) == '0':

            return LongInt('1')

        if str(value) == '1':

            return self

        result = LongInt('1')

        binary = value.to\_binary().lstrip('0')

        k = mod.elder\_bit() + 1

        r = ((LongInt('1') << (2\*k))/mod)[0]

        for i in binary:

            result = result \* result

            if result > mod:

                if mod.is\_2\_pow():

                    result = result.to\_binary()[-mod.elder\_bit():]

                    result = LongInt(result.from\_binary('0' \* (result.length \* 4 - len(result)) + result))

                elif result.elder\_bit() < mod.elder\_bit()\*2:

                    k = mod.elder\_bit() + 1

                    r = ((LongInt('1') << (2\*k))/mod)[0]

                    t = result - (((result\*r) >> (2\*k))\*mod)

                    if t < mod:

                        result = t

                    else:

                        result = t - mod

                else:

                    result = result - (result/mod)[0]\*mod

            if i == '1':

                result = result\*self

                if result > mod:

                    if mod.is\_2\_pow():

                        result = result.to\_binary()[-mod.elder\_bit():]

                        result = LongInt(result.from\_binary('0' \* (result.length \* 4 - len(result)) + result))

                    elif result.elder\_bit() < mod.elder\_bit()\*2:

                        k = mod.elder\_bit() + 1

                        r = ((LongInt('1') << (2\*k))/mod)[0]

                        t = result - (((result\*r) >> (2\*k))\*mod)

                        if t < mod:

                            result = t

                        else:

                            result = t - mod

                    else:

                        result = result - (result/mod)[0]\*mod

        return result

def randrange(self, start, stop):

        start = int(str(start), base=16)

        stop = int(str(stop), base=16)

        return LongInt(hex(randint(start, stop))[2:])

    def miller\_rabin(self, tries):

        if str(self) == '2':

            return True

        if self.is\_even():

            return False

        r = 0

        d = self - LongInt('1')

        while d.is\_even():

            r += 1

            d = d >> 1

        prime = True

        for \_ in range(tries):

            a = self.randrange(LongInt('2'), self-LongInt('1'))

            if str(a.pow\_barrett(d, self)) == '1':

                break

            for i in range(r):

                print(2\*i)

                if a.pow\_barrett(d << (2\*i), self) == self - LongInt('1'):

                    break

            else:

                prime = False

        return prime

4. Перевірка на коректність:

for i in range(10):

    print(i)

    a = LongInt(''.join(choices('0123456789abcdef', k=128)))

    b = LongInt(''.join(choices('0123456789abcdef', k=128)))

    c = LongInt(''.join(choices('0123456789abcdef', k=3)))

    mod = LongInt(''.join(choices('0123456789abcdef', k=randint(2, 64))))

    if (a % mod) != (a - ((a/mod)[0]\*mod)):

        print('ERROR1')

    if (a.add\_mod(b, mod)) != ((a+b) - ((a+b)/mod)[0]\*mod):

        print('ERROR2')

    if (a.sub\_mod(b, mod)) != ((a-b) - ((a-b)/mod)[0]\*mod):

        print('ERROR3')

    if (a\*b % mod) != (a.mul\_mod(b, mod)):

        print('ERROR4')

Перевірка проходить без помилок

5. Визначення часу роботи:

start\_init = time()

for \_ in range(1000):

    num = LongInt(''.join(choices('0123456789abcdef', k=128)))

end\_init = time()

start\_gcd = time()

for \_ in range(10):

    num1 = LongInt(''.join(choices('0123456789abcdef', k=128)))

    num2 = LongInt(''.join(choices('0123456789abcdef', k=128)))

    num1.gcd(num2)

end\_gcd = time()

start\_lcm = time()

for \_ in range(10):

    num1 = LongInt(''.join(choices('0123456789abcdef', k=128)))

    num2 = LongInt(''.join(choices('0123456789abcdef', k=128)))

    num1.lcm(num2)

end\_lcm = time()

start\_add\_mod = time()

for \_ in range(10):

    num1 = LongInt(''.join(choices('0123456789abcdef', k=128)))

    num2 = LongInt(''.join(choices('0123456789abcdef', k=128)))

    mod = LongInt(''.join(choices('0123456789abcdef', k=randint(2, 64))))

    num1.add\_mod(num2, mod)

end\_add\_mod = time()

start\_sub\_mod = time()

for \_ in range(10):

    num1 = LongInt(''.join(choices('0123456789abcdef', k=128)))

    num2 = LongInt(''.join(choices('0123456789abcdef', k=128)))

    mod = LongInt(''.join(choices('0123456789abcdef', k=randint(2, 64))))

    num1.sub\_mod(num2, mod)

end\_sub\_mod = time()

start\_add\_mod = time()

for \_ in range(10):

    num1 = LongInt(''.join(choices('0123456789abcdef', k=128)))

    num2 = LongInt(''.join(choices('0123456789abcdef', k=128)))

    mod = LongInt(''.join(choices('0123456789abcdef', k=randint(2, 64))))

    num1.add\_mod(num2, mod)

end\_add\_mod = time()

start\_mul\_mod = time()

for \_ in range(10):

    num1 = LongInt(''.join(choices('0123456789abcdef', k=128)))

    num2 = LongInt(''.join(choices('0123456789abcdef', k=128)))

    mod = LongInt(''.join(choices('0123456789abcdef', k=randint(2, 64))))

    num1.mul\_mod(num2, mod)

end\_mul\_mod = time()

start\_pow\_barrett = time()

for \_ in range(10):

    num1 = LongInt(''.join(choices('0123456789abcdef', k=128)))

    num2 = LongInt(''.join(choices('0123456789abcdef', k=3)))

    mod = LongInt(''.join(choices('0123456789abcdef', k=randint(2, 64))))

    num1.pow\_barrett(num2, mod)

end\_pow\_barrett = time()

start\_miller\_rabin = time()

for i in range(10):

    a = LongInt(''.join(choices('0123456789abcdef', k=128))).miller\_rabin(1)

end\_miller\_rabin = time()

init = end\_init - start\_init

gcd = (end\_gcd - start\_gcd - init\*2/100) / 10

lcm = (end\_lcm - start\_lcm - init\*2/100) / 10

add\_mod = (end\_add\_mod - start\_add\_mod - init\*2/100)/10

sub\_mod = (end\_sub\_mod - start\_sub\_mod-init\*2/100)/10

mul\_mod = (end\_mul\_mod - start\_mul\_mod-init\*2/100)/10

pw\_barrett = (end\_pow\_barrett - start\_pow\_barrett-init\*2/100)/10

miller\_rabin = (end\_miller\_rabin-start\_miller\_rabin-init/100)/10

Таблиця з отриманими результатами (час у секундах):

Час виконання тесту Міллера-Рабіна складає 12 секунд для числа довжиною 64 біти.

