15 Number Theory

15.6 Computing roots Modulo p

CATAM coursework for Part II of the Mathematical Tripos. Sections have been numbered as they appear in the manual.

2 Computing Legendre Symbols

Question 1 We implement the repeated squaring method for modular exponentiation using a recursive algorithm:

$$a^n \equiv \begin{cases} (a^{\frac{n}{2}})^2, & n \text{ even}, n > 0 \\ a \cdot (a^{\frac{n-1}{2}})^2, & n \text{ odd} \\ 1, & n = 0 \end{cases} \pmod{p}$$

This allows for efficient application of Euler's criterion to compute Legendre symbols. For $p = 30275233$, we compute (a/p) for a taking:

(i) 100 random values between 1 and p. Out of these, 58 numbers were found to be quadratic residues mod p.

(ii) all values between 1 and 100. Out of these, 58 numbers were found to be quadratic residues mod p.

Appendix A contains a record of all output produced.

Question 2 Suppose n is any odd number and $m \in \mathbb{Z}$. If $n = 1$, we have $(m/n) = (m/1) = 1$. Otherwise, we may assume $0 \leq m < n$ by reducing mod n if necessary. Note that $(0/n) = 0$ when $n > 1$. If m is a non-zero even number, we may get rid of all factors of 2 using the following property of the Jacobi symbol:

Lemma For any positive odd n we have

$$\left(\frac{2}{n}\right) = (-1)^{(n^2-1)/8} = \begin{cases} +1, & n \equiv \pm 1 \pmod 8 \\ -1, & n \equiv \pm 3 \pmod 8 \end{cases}$$

A proof of this can be found in [1] (*pp.* 47, Proposition II.2.6). Hence we

may assume m, n are both odd and $1 \leq m < n$. We state the following strengthening of quadratic reciprocity:

Lemma For any two positive odd integers m and n we have

$$\left(\frac{m}{n}\right) = (-1)^{(m-1)(n-1)/4} \left(\frac{n}{m}\right).$$

A proof of this can be found in [1] (*pp.* 47, Proposition II.2.7), but it follows immediately from the law of quadratic reciprocity and the observation that $(m/n) = 0$ whenever m and n are not coprime. Moreover, we can avoid computing the large product $(m-1)(n-1)/4$ by observing that

$$(-1)^{(m-1)(n-1)/4} = \begin{cases} -1, & m, n \equiv 3 \pmod 4 \\ +1, & \text{otherwise} \end{cases}$$

To compute (m/n), it now suffices to compute (n/m). Note that the 'denominator' strictly decreases after each iteration, hence the recursion must halt in finite time.

We compute the complexity of our algorithm which computes (m/n), without loss of generality $m \geq n$. If the numerator at each step is odd, the algorithm operates like Euclid's algorithm for the greatest common divisor. In particular, there are at most $O(\log n)$ reductions of form $(m/n) \mapsto (n \bmod m/m)$. However, with each reduction we also remove all powers of 2 from the numerator. Since the numerator is bounded by m, each reduction takes $O(\log m)$ basic operations hence the entire algorithm has complexity $O(\log m \log n)$.

Reducing mod p, we may assume $0 \leq m, n < p$ so that the algorithm has complexity $O((\log p)^2)$. In contrast, the repeated squaring algorithm of Question 1 computes the Legendre symbol in $O(\log p)$ basic operations. While it may seem that repeated squaring is faster, we must also acknowledge the fact that the 'basic' operations involved are multiplications of numbers containing $\log p$ bits. For large p, this significantly affects the performance of the algorithm from Question 1 whereas the second algorithm remains relatively unaffected.

3 Computing square roots mod p

Question 3 Suppose $p \equiv 3 \pmod 4$ and a is a residue mod p. In particular, Euler's criterion implies $a^{(p-1)/2} \equiv 1 \pmod p$. Then $x \equiv a^{(p+1)/4}$ is a solution to $x^2 \equiv a \pmod p$ since

$$x^2 \equiv a^{(p+1)/2} \equiv a \cdot a^{(p-1)/2} \equiv a \pmod p.$$

Suppose $p \equiv 5 \pmod 8$. We have $(2/p) \equiv 2^{(p-1)/2} \equiv -1 \pmod p$. If a is a quadratic residue, we have $a^{(p-1)/2} \equiv 1 \Rightarrow a^{(p-1)/4} \equiv \pm 1 \pmod p$. In particular, we can write $2^{k(p-1)/2} \cdot a^{(p-1)/4} \equiv 1 \pmod p$ for some $k \in \{0, 1\}$. But then we have $a \equiv 2^{k(p-1)/2} \cdot a^{(p-1)/4} \cdot a \equiv 2^{k(p-1)/2} \cdot a^{(p+3)/4} \pmod p$. Observe that all the exponents involved are even, so we can read off a solution to $x^2 \equiv a$ as $x \equiv 2^{k(p-1)/4} \cdot a^{(p+3)/8} \pmod p$.

Now suppose p is a prime of the form $2^n + 1$ (we only consider $n > 2$ since $p = 3, 5$ have been covered above). Then $p \equiv (-1)^n + 1 \pmod 3$, hence n must be even. It follows that $p \equiv 1 \pmod 4$ and $p \equiv 2 \pmod 3$, so that $(3/p) = (p/3) = (2/3) = -1$. Let g be any primitive root mod p. The subgroup $\langle g^2 \rangle$ has order $2^{n-1} = \frac{p-1}{2}$ hence contains all the quadratic residues. Moreover, it is the unique multiplicative subgroup of order 2^{n-1}. Since $3 \notin \langle g^2 \rangle$, the multiplicative order of 3 must be 2^n i.e. 3 is also a primitive root mod p.

Question 4 Suppose $p = 65537 = 2^{16} + 1$, and we wish to solve the congruence $x^2 \equiv 18612 \pmod p$. We compute $(18612/65537) = 1$ using the program written for Question 2, so such an x exists. For the purposes of this question, use '=' to denote congruence mod p. For modular exponentiation, we use a program based on the repeated squaring method.

Since 3 is a primitive root mod p, we may write $x = 3^{r_0 + 2r_1 + 2^2 r_2 + \cdots}$ where each $r_j \in \{0, 1\}$. The congruence $x^2 = 18612$ can be written as $\prod_{j \geq 0} 3^{r_j 2^{j+1}} = 18612 \pmod p$.

Raising both the sides to 2^{14}, all the terms for $j \geq 1$ vanish and we are left with $3^{r_0 2^{15}} = 18612^{2^{14}} = 1$. Since 3 is a primitive root, $3^{2^{15}} = -1$ by Euler's criterion and we have $r_0 = 0$. In fact, $18612^{2^n} = 1$ for all $14 \geq n \geq 11$ hence we have $r_0 = ... = r_3 = 0$.

Raising both the sides of $\prod_{j\geq 4} 3^{r_j 2^{j+1}} = 18612$ to the power 2^{10}, we obtain $3^{r_4 2^{15}} = -1$, hence $r_4 = 1$. We may multiply both the sides of the congruence by $3^{-r_4 2^5} = 3^{2^{16}-2^5} = 29606$ to obtain $\prod_{j\geq 5} 3^{r_j 2^{j+1}} = 57313$.

Again, $57313^{2^n} = 1$ for $9 \geq n \geq 6$, so $r_5 = ... = r_8 = 0$, and we have $\sum_{j\geq 9} 3^{r_j 2^{j+1}} = 57313$. Raising both the sides to the power 2^5, we obtain $3^{r_9 2^{15}} = -1$, hence $r_9 = 1$. Multiply both the sides by $3^{-r_9 2^{10}} = 3^{2^{16}-2^{10}} = 64509$ to obtain $\prod_{j\geq 10} 3^{r_j 2^j} = 65536 = -1$. Comparing with $3^{2^{15}} = -1$, we deduce $r_{10} = ... = r_{13} = 0$ and $r_{14} = 1$.

We can now read off the solution to $x^2 = 18612$ as $x = 3^r$ where $r = 2^4 + 2^9 + 2^{14}$. A square root of 18612 (mod p) hence is 45462. The other square root is $-45462 = 20075$ and these are the only solutions to $x^2 = 18612$ since $(\mathbb{Z}/p\mathbb{Z})^*$ is a field.

Suppose p is any odd prime and a is a quadratic residue mod p. In this section, use '=' to denote congruence mod p. We may find $\alpha > 0$ and s odd such that $p - 1 = 2^\alpha s$. Since s is odd, we define $z = a^{(s+1)/2}$ and observe that if $y^2 = a^s$, then zy^{-1} is a square root of a mod p (where y^{-1} is the multiplicative inverse of y in $(\mathbb{Z}/p\mathbb{Z})^\times$.)

Now $(a^s)^{2^{\alpha-1}} = a^{(p-1)/2} = (a/p) = 1$, so y is an element of the cyclic multiplicative group $G = \{g \in (\mathbb{Z}/p\mathbb{Z})^\times \mid g^{2^\alpha} = 1\}$. Suppose n is any non-residue mod p, and let $b = n^s$. Then we have $b^{2^\alpha} = n^{p-1} = 1$ hence $b \in G$, and moreover b generates the group since $b^{2^{\alpha-1}} = n^{(p-1)/2} = (n/p) = -1$. We can then write $y = b^r$, and solve for r algorithmically by considering its binary expansion. A square root of a, then, is $zb^{2^\alpha - r}$.

Question 5 We implement an algorithm that uses the above method to compute square roots mod p when $p \equiv 1 \pmod 8$, using the more direct computations from Question 3 to handle other cases. The second solution to the congruence $x^2 \equiv a \pmod p$ can be computed as -1 times the first solution.

Here are some test cases, in the form `(a, root a)`. The values of a (other than the first) have been chosen randomly subject to being quadratic residues. The primes have been chosen[1] to cover all possible congruence classes mod 8.

[1]The first prime is the largest known Fermat prime, while the latter four are relatively

The generated roots can be confirmed to be accurate by squaring them. In particular, the first test case agrees with what we found in Question 4.

```
mod 65537:   (18612, 45462)  (14131, 5917)  (2761, 58554)
mod 10501:   (6761, 4425)    (7888, 5247)   (10146, 6935)
mod 10601:   (5182, 5389)    (172, 3957)    (8087, 3958)
mod 11311:   (728, 6770)     (3929, 5167)   (10263, 4552)
mod 11411:   (6915, 551)     (2374, 10717)  (7003, 7705)
```

Additionally, we compute the roots of all quadratic residues in $\{1, 2, ..., 20\}$ mod 30275233 and present the results in Appendix B.

The complexity of this algorithm can two sources, the first of which is finding b. We run an exhaustive search, computing (n/p) for $n = 1, 2, ...$ until a non-residue is hit. In the worst case this has complexity $O(p(\log p)^2)$, but assuming that quadratic residues are uniformly distributed in $(\mathbb{Z}/p\mathbb{Z})^\times$ i.e. if $\mathbb{P}((a/p) = 1) = \frac{1}{2}$, we expect to succeed in $\sum_{n=1}^{\infty} n(\frac{1}{2})^n = 2$ steps. Hence this part of the algorithm effectively has complexity $O(1)$ (since $n \approx 1$).

Hence all of the complexity comes from computing r, which we can bound by p. Then we need to compute the $O(\log p)$ digits of r, each of which takes $O(\log p)$ steps to compute (since the computation involves exponentiation). Hence we estimate the entire algorithm to have complexity $O((\log p)^2)$.

4 Computing roots of polynomials mod p

Let p be a prime.

Question 6 Given two polynomials f and g in $(\mathbb{Z}/p\mathbb{Z})[x]$, note that $\mathbb{Z}/p\mathbb{Z}$ is a field hence the leading coefficient of g is a unit– moreover, we can use Fermat's little theorem to explicitly compute its inverse using the repeated squaring method. With this, we implement a recursive algorithm to eliminate terms in f of degree higher than $\deg(g)$ and find polynomials q, r such that $f = qg + r$, $\deg(r) < \deg(g)$. Write $r = \mathrm{rem}(f, g)$.

This allows for the application of Euclid's algorithm to find the greatest common divisor of f and g, by

$$\gcd(f, g) = \begin{cases} f, & g = 0 \\ \gcd(g, \mathrm{rem}(f, g)), & \text{otherwise} \end{cases}.$$

large palindromic primes [OEIS: A055578].

The answer is generated up to a unit, so we divide by the leading coefficient to normalise. To test the algorithm, we compute the greatest common divisors for the following pairs of polynomials:

$$\begin{aligned}
&\gcd(x^3+6x^2+5x+5, x^3+13x^2+6x+3) = x^2+78x+62 && (p=109).\\
&\gcd(x^3+2x^2+9x+4, x^3+3x^2+7x+9) = 1 && (p=131).\\
&\gcd(x^3+3x^2+9x+12, x^3+6x^2+12x+4) = x+83 && (p=157).
\end{aligned}$$

If f, g, h are polynomials, write $f \equiv g \pmod{h}$ to mean 'there exists a polynomial q such that $f = qh + g$.' It is straightforward to check that this is an equivalence relation, and $f_1 \equiv g_1 \pmod{h}$, $f_2 \equiv g_2 \pmod{h}$ implies $f_1 + f_2 \equiv g_1 + g_2 \pmod{h}$ and $f_1 f_2 \equiv g_1 g_2 \pmod{h}$. Moreover, $\mathrm{rem}(f, h)$ is the unique g such that $f \equiv g \pmod{h}$, $\deg(g) < \deg(h)$. It follows that $f \equiv f' \pmod{h}$ implies $\mathrm{rem}(f, h) = \mathrm{rem}(f', h)$.

Question 7 Let $f \in (\mathbb{Z}/p\mathbb{Z})[x]$ be any polynomial. The polynomial $\Phi(x) = x^p - x$ factorises as $\Phi(x) = \prod_{i=0}^{p-1}(x-i)$ hence $g = \gcd(f, \Phi)$ is a product of distinct linear factors. Moreover, $(x-i) \mid f$ if and only if $(x-i) \mid g$, hence to compute the roots of f it suffices to compute the roots of g.

Now the computation of g using Euclidean algorithm takes at most $\deg(f)$ steps, and in each step except for the first one, the degree of polynomials involved is less than $\deg(f)$. Naively computing $\mathrm{rem}(\Phi, f)$ would take $O(p)$ steps; however we observe that

$$\mathrm{rem}(x^p - x, f) \equiv \mathrm{rem}(x^p, f) - \mathrm{rem}(x, f) \pmod{h}$$

The second term is straightforward to compute. For the first term, we can use a repeated squaring algorithm similar to that for modular exponentiation of numbers. This speeds up the first step of Euclidean algorithm to take $O(\log p)$ steps, and moreover the polynomials involved in each computation have degree at most $2\deg(f)$.

Having reduced f to the case where it is a product of distinct linear factors, we fix a $v \in \mathbb{Z}/p\mathbb{Z}$ and compute $\gcd((x+v)^{(p-1)/2} - 1, f)$. Everything that has been said about avoiding large powers in the computation applies here. The greatest common divisor is f if $\alpha + v$ is a quadratic residue for every root α of f, it is 1 if $\alpha + v$ is a non-residue for every root α of f. Otherwise,

we have arrived at a non-trivial factor of f, and can repeat the process with these till we have found all linear factors (and hence the roots).

How many tries will this algorithm require before successfully factorizing a quadratic? For simplicity assume that quadratic residues are uniformly distributed in $(\mathbb{Z}/p\mathbb{Z})^\times$, i.e. $\mathbb{P}((n/p) = 1) = \frac{1}{2}$ for n sampled randomly. Then in this simplified model, the probability of obtaining a successful factorisation $(x-\alpha)(x-\beta)$ with a randomly chosen v is $\frac{1}{2}$ (i.e. probability that both $v+\alpha$ and $v+\beta$ are residues or non-residues). Hence the expected number of tries before success is $\sum_{n=1}^{\infty} n(\frac{1}{2})^n = 2$.

We employ this method to find square roots by using $f = x^2 - a$. The roots of all quadratic residues between 21 and 99 are computed and recorded in Appendix B. The averate number of tries until success is 1.83 which is very close to the expected value predicted by our model.

Given the discussion above, for calculation of complexity we assume the algorithm succeeds on the first trial. Since f is quadratic, the complexity of polynomial division arises from having to invert leading coefficients which takes $O(\log p)$ steps. The faster algorithm for computing greatest common divisors takes $O(\log p)$ steps to compute $\gcd((x+v)^{\frac{p-1}{2}} - 1, f)$ hence the entire algorithm has complexity $O(\log p)$.the entire algorithm has complexity $O(\log p)$.the entire algorithm has complexity $O(\log p)$.the entire algorithm has complexity $O(\log p)$.the entire algorithm has complexity $O(\log p)$.the entire algorithm has complexity $O(\log p)$.the entire algorithm has complexity $O(\log p)$.the entire algorithm has complexity $O(\log p)$. This is significantly faster than the algorithm of Question 5 for large p.

Question 8 We use the algorithm discussed above to compute the roots of the following biquadratic polynomials modulo 35564117:

$x^4 + 5x^3 + 12x^2 + 6$	$\{7174009, 9335487, 21485344, 33133389\}$
$x^4 + x^3 + 3x^2 + 7x + 4$	$\{22043805, 13520313, 35564116\}$
$x^4 + 4x^3 + 15x^2 + 3x + 8$	$\{16498240, 3842901\}$

Appendix A: Computed Legendre symbols

Using Euler's criterion, Legendre symbols were computed for 100 random values between 1 and $p = 30275233$. The results are recorded here in the

format `a: (a/p)`.

```
24141936: -1     29578235: -1      6027797: -1      4399811: -1
20332888: -1      8682668: -1     13809005:  1      5091412:  1
11576664: -1      7896904:  1      9157099:  1     20348411:  1
 4117541:  1     15944328:  1       654042:  1     20561326: -1
11575859: -1      6943447:  1     27916149:  1      5504001:  1
15002063: -1      3532513: -1     28710072: -1     17259301:  1
23365236:  1      5930395:  1      9001591:  1     26866135: -1
10214255: -1     13947469: -1     15677694:  1     26060209: -1
13784763:  1     21689417:  1       485366: -1     11893055:  1
26469353:  1     21860157: -1     13229681: -1     13923851:  1
18558515: -1     14559517:  1      7424651: -1      7982684: -1
 9845602: -1     18803726: -1      9773489:  1     11324071: -1
10223823: -1       971442:  1     23485997:  1     18253324:  1
 9413550:  1     26309092:  1      2722011:  1      8253929:  1
24917551: -1     12956198:  1      4633362:  1     16725381: -1
10224020:  1     14575687: -1     11616773: -1      6718462:  1
 5225404:  1      7626498:  1      6241372:  1     18460242:  1
 4236700:  1     10253395: -1      6578488: -1     12141336: -1
 6940169: -1     27691476: -1     12249683: -1      2980379: -1
22984728:  1     24957651: -1     29082462:  1     10337564: -1
11326793:  1     22856609: -1     24041680: -1     17566555: -1
10789636: -1     19899519:  1     22833624: -1       636904: -1
18111573:  1      1749387:  1      2455622:  1     10199970: -1
17759897:  1      8759666:  1     10291713:  1      7107328: -1
21706118:  1      8048139:  1      9420873:  1      8160687: -1
```

`Number of quadratic residues encountered: 52`

We also perform a similar calculation for all a between 1 and 100.

```
  1:  1       2:  1       3:  1       4:  1       5: -1
  6:  1       7:  1       8:  1       9:  1      10: -1
 11: -1      12:  1      13:  1      14:  1      15: -1
 16:  1      17:  1      18:  1      19:  1      20: -1
 21:  1      22: -1      23: -1      24:  1      25:  1
 26:  1      27:  1      28:  1      29:  1      30: -1
 31: -1      32:  1      33: -1      34:  1      35: -1
```

```
 36:  1      37: -1      38:  1      39:  1      40: -1
 41: -1      42:  1      43: -1      44: -1      45: -1
 46: -1      47:  1      48:  1      49:  1      50:  1
 51:  1      52:  1      53:  1      54:  1      55:  1
 56:  1      57:  1      58:  1      59: -1      60: -1
 61: -1      62: -1      63:  1      64:  1      65: -1
 66: -1      67:  1      68:  1      69: -1      70: -1
 71: -1      72:  1      73:  1      74: -1      75:  1
 76:  1      77: -1      78:  1      79: -1      80: -1
 81:  1      82: -1      83:  1      84:  1      85: -1
 86: -1      87:  1      88: -1      89: -1      90: -1
 91:  1      92: -1      93: -1      94:  1      95: -1
 96:  1      97: -1      98:  1      99: -1     100:  1

Number of quadratic residues encountered: 58
```

Appendix B: Roots of residues mod 30275233

For every integer $1 \leq a \leq 20$, we check if a is a quadratic residue mod $p = 30275233$ and if so, compute a solution to $x^2 \equiv a \pmod{p}$ using the method of Question 5. The computed roots are presented in format `(a, root a)`. Note that the second root is simply the additive inverse of the given one.

```
(1, 1)           (2, 1149953)     (3, 2513663)
(4, 2)           (6, 5886698)     (7, 2907008)
(8, 2299906)     (9, 3)           (12, 5027326)
(13, 11851235)   (14, 22168463)   (16, 30275229)
(17, 18030218)   (18, 3449859)    (19, 4272041)
```

We perform a similar exercise for integers $21 \leq a < 100$, this time using the method of Question 7 instead i.e. computing the roots of $x^2 - a$. The computed square roots are presented in format `(a, root a)`. We also record the average number of values of v used before arriving at a successful factorisation.

```
(21, 22061809)   (24, 18501837)   (25, 5)
(26, 27657471)   (27, 7540989)    (28, 5814016)
(29, 8124419)    (32, 25675421)   (34, 29489830)
```

```
(36, 6)          (38, 5406095)    (39, 29358329)
(42, 3695637)    (47, 14785760)   (48, 20220581)
(49, 7)          (50, 24525468)   (51, 19443699)
(52, 23702470)   (53, 21857170)   (54, 17660094)
(55, 8318960)    (56, 16213540)   (57, 27902481)
(58, 5300371)    (63, 21554209)   (64, 8)
(67, 7229599)    (68, 24490030)   (72, 23375515)
(73, 12449718)   (75, 12568315)   (76, 8544082)
(78, 29309412)   (81, 9)          (83, 26415583)
(84, 16426848)   (87, 16157654)   (91, 24388229)
(94, 25188917)   (96, 23546792)   (98, 22225562)

Average number of tries: 1.8333333333333333
```

Appendix C: Programs

jacobi.py

```
def modexp(a,n,p):
    result = 1
    x = a % p
    while n:
        if n % 2 != 0:
            result = (result * x) % p
        x = (x * x) % p
        n = n // 2
    return result

def invmod(a, p):
    return modexp(a, p-2, p)

def legendre(a,p):
    result = modexp(a, (p-1) // 2, p)
    if result <= p//2:
        return result
    else:
        return (result-p)

def jacobi(m,n):
    m = m % n
    if n == 1:
        return 1
    elif m == 0:
        return 0
    else:
        sgn = 1
        tmp = n % 8
        while m % 2 == 0:
            m = m // 2
            if tmp == 3 or tmp == 5:
                sgn = - sgn
        if m % 4 == 3 and n % 4 == 3:
            sgn = - sgn
        return sgn * jacobi(n,m)
```

sqroots.py

```
import jacobi as j

def rootmod(a,p):
    if p % 4 == 3:
        root = j.modexp(a, (p+1)//4, p) % p
    elif p % 8 == 5:
        if j.modexp(a, (p-1)//4, p) == 1:
            root = j.modexp(a, (p+3)//8, p) % p
        else:
            u = j.modexp(2, (p-1)//4, p)
            v = j.modexp(a, (p+3)//8, p)
            root = (u * v) % p
    else:
```

```
        # decompose p-1 = (2^alpha) * s
        s = p - 1
        alpha = 0
        while s % 2 == 0:
            s = s // 2
            alpha = alpha + 1
        order = 2 ** alpha
        # find non-residue n and generator b =n^s of roots of unity
        n = 1
        while j.jacobi(n,p) == 1:
            n = n + 1
        b = j.modexp(n, s, p)
        # find binary expansion of r
        r = []
        num = j.modexp(a, s, p)
        i = 0
        while num != 1:
            if j.modexp(num, 2 ** (alpha - i - 2), p) == 1:
                r.append(0)
            else:
                r.append(1)
                u = j.modexp(b, order - 2**(i+1), p)
                num = (u * num) % p
            i += 1
        exp = order
        for i in range(len(r)):
            exp = exp - (2 ** i) * r[i]
        yinv = j.modexp(b, exp, p)
        z = j.modexp(a, (s+1)//2, p)
        root = (z * yinv) % p
    return root
```

polyroots.py

```
import jacobi as j
from polynomial import Polynomial
import random

x = Polynomial(1,0)

def stdize(f,p):
    if f == Polynomial(0):
        return f
    else:
        g = f
        for i in range(f.degree + 1):
            g[i] = f[i] % p
        return g

def monic(f, p):
    if f == Polynomial(0):
        return f
    else:
        a = f[f.degree]
        g = f * j.invmod(a, p)
        return stdize(g, p)
```

```
23
24 def dividemod(f, g, p):
25     f = stdize(f, p)
26     g = stdize(g, p)
27     if g.degree < 0:
28         raise Exception("division by 0")
29     else:
30         quo = Polynomial(0)
31         rem = f
32         while rem.degree >= g.degree:
33             deg = rem.degree
34             tmp = g * (x ** (deg - g.degree))
35             coeff = (j.invmod(tmp[deg],p) * rem[deg]) % p
36             quo += coeff * (x ** (deg - g.degree))
37             tmp = tmp * coeff
38             rem = stdize(rem - tmp, p)
39         return (quo, rem)
40
41 def gcdmod(f, g, p):
42     if g == Polynomial(0):
43         return monic(f, p)
44     else:
45         h = dividemod(f, g, p)[1]
46         return gcdmod(g, h, p)
47
48 def polymodexp(f, n, g, p): # Computes f^n (mod g) in Z/pZ[X]
49     result = Polynomial(1)
50     tmp = dividemod(f, g, p)[1]
51     while n:
52         if n % 2 != 0:
53             result = dividemod(result * tmp, g, p)[1]
54         tmp = dividemod(tmp * tmp, g, p)[1]
55         n = n // 2
56     return result
57
58 def speedygcd(f, n, h, g, p): # gcd(a^n+b, g) when n is large
59     f1 = polymodexp(f, n, g, p) + h
60     return gcdmod(f1, g, p)
61
62 def polyrootsmod(f, p):
63     f = speedygcd(x, p, -x, f, p)
64     if f.degree <= 0:
65         return ([], -1)
66     elif f.degree == 1:
67         solution = (-1 * f[0] * j.invmod(f[1], p)) % p
68         return ([solution], -1)
69     else:
70         fails = []
71         while True:
72             v = random.randint(0,p-1)
73             if v not in fails:
74                 v = Polynomial(v)
75                 factor1 = speedygcd(x + v, p//2, Polynomial(-1), f, p)
76                 if factor1.degree <= 0 or factor1.degree == f.degree:
77                     fails.append(v[0])
78                     continue
79                 else:
```

```
                    break
        factor2 = dividemod(f, factor1, p)[0]
        roots = polyrootsmod(factor1, p)[0] + polyrootsmod(factor2, p)[0]
        return (roots, len(fails)+1)
```

main.py

```
import jacobi as j
import sqroots as s
import polyroots as pr
import random
from polynomial import Polynomial

# Pretty printing
def monostr(number, digitcount):
    n = number
    if n == 0:
        count = 1
    else:
        count = 0
        while(n > 0):
            n = n // 10
            count = count + 1
    return (" "*(digitcount - count) + str(number))

def legstr(n):
    if n == -1:
        return ("-1")
    else:
        return (" "+str(n))

# Question 1
p1 = 30275233
f1res = 0
f2res = 0

# clear contents before proceeding
f1 = open("../output/Legendre-100.txt", "w"); f1.close()
f2 = open("../output/Legendre-random.txt", "w"); f2.close()

f1 = open("../output/Legendre-100.txt", "a")
f2 = open("../output/Legendre-random.txt", "a")
for i in range(100):
    if i % 5 == 0 and i > 0:
        f1.write("\n")
    if i % 4 == 0 and i > 0:
        f2.write("\n")
    leg1 = j.legendre(i+1,p1)
    if leg1 == 1:
        f1res += 1
    f1.write(monostr(i+1, 3) + ": " + legstr(leg1) + " "*5)
    rnum = random.choice(range(p1)) + 1
    leg2 = j.legendre(rnum,p1)
    if leg2 == 1:
        f2res += 1
    f2.write(monostr(rnum, 8) + ": " + legstr(leg2) + " "*5)
```

```
f1.write("\n\nNumber of quadratic residues encountered: "+ str(f1res))
f2.write("\n\nNumber of quadratic residues encountered: "+ str(f2res))
f1.close(); f2.close()

# Question 2
f = open("../output/Test-roots.txt", "w"); f.close()

def gentests(prime, count):
    i = 0
    while i < count:
        a = random.choice(range(prime))
        if j.jacobi(a, prime) == 1:
            f.write("("+str(a)
                    + ", "
                    + str(s.rootmod(a,prime)) + ")  ")
            i += 1
        else:
            continue
    f.write("\n")

# Question 5
f = open("../output/Test-roots.txt", "a")

f.write("mod 65537:   ")
f.write("(18612, " + str(s.rootmod(18612,65537)) + ")  ")
gentests(65537, 2)
f.write("mod 10501:   ")
gentests(10501, 3)
f.write("mod 10601:   ")
gentests(10601, 3)
f.write("mod 11311:   ")
gentests(11311, 3)
f.write("mod 11411:   ")
gentests(11411, 3)

f.close()

f = open("../output/Twenty-roots.txt", "w"); f.close()
f = open("../output/Twenty-roots.txt", "a")
p = 30275233

count = 0
for a in range(20):
    if j.jacobi(a, p) != 1:
        continue
    else:
        count += 1
        f.write("("+str(a)+", "+str(s.rootmod(a,p))+")   ")
        if count % 3 == 0:
            count = 0
            f.write("\n")
f.close()


# Question 6
print("\nGCDs of polynomials:")
print("gcd [1,6,5,5], [1,13,6,3] mod 109")
```

```
print(pr.gcdmod(Polynomial(1,6,5,5),
                Polynomial(1,13,6,3),
                109))
print("gcd [1,2,9,4], [1,3,7,9] mod 131")
print(pr.gcdmod(Polynomial(1,2,9,4),
                Polynomial(1,3,7,9),
                131))
print("gcd [1,3,9,12], [1,6,12,4] mod 157")
print(pr.gcdmod(Polynomial(1,3,9,12),
                Polynomial(1,6,12,4),
                157))

# Question 7
f = open("../output/Twenty-roots-2.txt", "w"); f.close()
f = open("../output/Twenty-roots-2.txt", "a")
p = 30275233

totaltries = []
count = 0
for a in range(21, 100):
    if j.jacobi(a, p) != 1:
        continue
    else:
        count += 1
        (roots,tries) = pr.polyrootsmod(Polynomial(1,0,-1 * a), p)
        totaltries.append(tries)
        f.write("("+str(a)+", "+str(roots[0])+")   ")
        if count % 3 == 0:
            count = 0
            f.write("\n")
f.write("\nAverage number of tries: ")
f.write(str(sum(totaltries)/len(totaltries)))
f.close()

# Question 8
p = 35564117
polys = [Polynomial(1,5,12,0,6),
         Polynomial(1,1,3,7,4),
         Polynomial(1,4,15,3,8)]
for poly in polys:
    print("\nRoots of "+str(poly))
    print(pr.polyrootsmod(poly, p)[0])
```

References

[1] Koblitz, N. *A course in Number Theory and Cryptography*, Graduate Texts in Mathematics 114, Springer, 1987.