4.1 Complexity of finite field arithmetic

The table below summarizes the costs of various arithmetic operations.

	integers $\mathbb{Z}$	finite field $\mathbb{F}_q$
addition/subtraction	$O(n)$	$O(n)$
multiplication	$\mathsf{M}(n)$	$O(\mathsf{M}(n))$
Euclidean division (reduction)	$O(\mathsf{M}(n))$	$O(\mathsf{M}(n))$
extended gcd (inversion)	$O(\mathsf{M}(n)\log n)$	$O(\mathsf{M}(n)\log n)$
exponentiation		$O(n\mathsf{M}(n))$
root-finding (probabilistic)		$O(\mathsf{M}(d(n+\log d))(n+\log d))$
square-roots (probabilistic)		$O(n\mathsf{M}(n))$

For integers, the parameter n is the bit-length, and for the finite field $\mathbb{F}_q$ we let $n = \log q$. In order to simplify matters, we assume $q = p^e$ where $\log e = O(\log p)$. The bounds above can be achieved more generally, but when the characterstic p is small compared to the degree $e = [\mathbb{F}_q : \mathbb{F}_p]$ one uses different algorithms.

In the case of polynomial root-finding, d is the degree of the polynomial and we list bounds on the expected running time since these operations are most efficiently implemented using probabilistic algorithms. In Lecture 3 we addressed the cost of addition and subtraction in both $\mathbb{Z}$ and $\mathbb{F}_q$, and the cost of multiplication in $\mathbb{Z}$. In this lecture we will fill in the rest of the table.

We saw in Lecture 3 how to use Kronecker substitution to reduce polynomial multiplication to integer multiplications. Thus we can multiply elements of $\mathbb{F}_q$ using integer multiplication, provided that we have a way to reduce products back to our standard representations of $\mathbb{F}_p \simeq \mathbb{Z}/p\mathbb{Z}$ and $\mathbb{F}_q \simeq \mathbb{F}_p[x]/(f)$, using integers in $[0, p-1]$ and polynomials of degree less than $\deg f$. For this we use Euclidean division.

4.2 Euclidean division

Given two positive integers a and b, we wish to compute the unique nonnegative integers q and r for which

$$a = bq + r \qquad (0 \le r < b).$$

We have $q = \lfloor \frac{a}{b} \rfloor$ and $r = a \mod b$. It is enough to compute q, since we can then compute $r = a - bq$. To compute q, we determine a sufficiently precise approximation $c \approx \frac{1}{b}$ and then obtain q by computing ca and rounding down to the nearest integer.

Let us recall Newton's method for finding the root of a function $f(x)$. We start with an initial approximation x_0, and at each step, we take the line tangent to $(x_i, f(x_i))$ and call its intersection with the x-axis x_{i+1}. Thus

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}.$$

To compute $c \approx \frac{1}{b}$, we look for a root of $f(x) = \frac{1}{x} - b$. Thus we use the Netwon iteration

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i - \frac{\frac{1}{x_i} - b}{-\frac{1}{x_i^2}} = 2x_i - bx_i^2.$$

As an example, let us approximate $\frac{1}{123456789}$. For the sake of illustration we work in base 10, but of course an actual implementation would use base 2 (or more likely base 2^w, where w is the word size).

$$\begin{aligned} x_0 &= 1 \times 10^{-8} \\ x_1 &= 2(1 \times 10^{-8}) - (1.2 \times 10^8)(1 \times 10^{-8})^2 \\ &= 0.80 \times 10^{-8} \\ x_2 &= 2(0.80 \times 10^{-8}) - (1.234 \times 10^8)(0.80 \times 10^{-8})^2 \\ &= 0.8102 \times 10^{-8} \\ x_3 &= 2(0.8102 \times 10^{-8}) - (1.2345678 \times 10^8)(0.8102 \times 10^{-8})^2 \\ &= 0.81000002 \times 10^{-8}. \end{aligned}$$

Note that we double the precision we are using at each step, and each x_i is correct up to an error in its last decimal place. The value x_3 already suffices to correctly compute $\lfloor a/b \rfloor$ for $a \leq 10^{15}$.

To analyze the complexity of this approach, let us assume that b has n bits and that a has at most $2n$ bits; this is precisely the situation we will encounter when we wish to reduce the product of two integers in $[0, p-1]$ that represent elements of $\mathbb{Z}/p\mathbb{Z}$. During the Newton iteration to compute $c \approx \frac{1}{b}$, the size of the integers involved doubles with each step, and the cost of the arithmetic operations grows at least linearly. The total cost is thus at most twice the cost of the last step, which is $\mathsf{M}(n) + O(n)$; note that all operations can be performed using integers by shifting the operands appropriately. Thus we can compute $c \approx \frac{1}{b}$ in time $2\mathsf{M}(n) + O(n)$. We can then compute $ca \approx \frac{a}{b}$, round to the nearest integer, and compute $r = a - bq$ in a total time of at most $4\mathsf{M}(n) + O(n)$.

With a slightly more sophisticated version of this approach it is possible to compute r in time $3\mathsf{M}(n)+O(n)$. and if we expect to repeatedly perform Euclidean division with the same denominator, as when working in $\mathbb{F}_p$, the cost of all subsequent reductions can be reduced to $\mathsf{M}(n) + O(n)$. This uses what is known as *Barret Reduction*; see [3, Algorithm 10.17].[1]

There is an analogous version of the algorithm described above for polynomials that uses the exact same Newton iteration $x_{i+1} = 2x_i - bx_i^2$, where b and the x_i are now polynomials. The precision is controlled by working modulo x^k, where k is doubled at each step. See Algorithms 9.3 and 9.5 in [2] for details.

4.3 Extended Euclidean algorithm

We first recall the standard Euclidean algorithm for computing the greatest common divisor of distinct positive integers a and b. For $a > b$ we repeatedly apply

$$\gcd(a, b) = \gcd(b, a \bmod b),$$

where we take $a \bmod b$ to be the unique integer $r \in [0, b-1]$ congruent to a modulo b.

To compute the multiplicative inverse of an integer modulo a prime, we use the extended Euclidean algorithm, which expresses $\gcd(a, b)$ as a linear combination

$$\gcd(a, b) = as + bt,$$

[1] The algorithm given in [3] for the precomputation step of Barret reduction uses Newton iteration with a fixed precision, which is asymptotically suboptimal; it is better to use the varying precision approach described above. But in practice the precomputation cost is usually not a major concern.

with $s < b$ and $t < a$. In the case that a is prime, we obtain $as + bt = 1$, and t is the inverse of b modulo a. To compute the integers s and t we use the following algorithm. First, let

$$R_1 = \begin{bmatrix} a \\ b \end{bmatrix}, \quad S_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad T_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix},$$

and note that $R_1 = aS_1 + bT_1$. We then compute

$$Q_i = \begin{bmatrix} 0 & 1 \\ 1 & -q_i \end{bmatrix}, \quad R_{i+1} = Q_i R_i, \quad S_{i+1} = Q_i S_i, \quad T_{i+i} = Q_i T_i,$$

where q_i is the quotient $\lfloor R_{i,1}/R_{i,2} \rfloor$ obtained via Euclidean division. Note that applying the linear transformation Q_i to both sides of $R_i = aS_i + bT_i$ ensures $R_{i+1} = aS_{i+1} + bT_{i+1}$. The algorithm terminates when $R_{i,2}$ becomes zero, at which point we have

$$R_k = \begin{bmatrix} d \\ 0 \end{bmatrix}, \quad S_k = \begin{bmatrix} s \\ \pm b \end{bmatrix}, \quad T_k = \begin{bmatrix} t \\ \mp a \end{bmatrix},$$

with $\gcd(a, b) = d = sa + tb$. As an example, with $a = 1009$ and $b = 789$ we have

r	q	s	t
1009		1	0
789	1	0	1
220	3	1	−1
129	1	−3	4
91	1	4	−5
38	2	−7	9
15	2	18	−23
8	1	−43	55
7	1	61	−78
1	7	−104	133
0		789	−1009

Thus 133 is the inverse of 789 modulo 1009.

It is clear that the r is reduced by at least a factor of 2 every two steps, thus the total number of iterations is $O(n)$, and each step involves a Euclidean division whose cost is bounded by $O(\mathsf{M}(n))$. This yields a complexity of $O(n\mathsf{M}(n))$, but a more careful analysis shows that it is actually $O(n^2)$, even if schoolbook multiplication is used (the key point is that the total size of all the q_i is $O(n)$ bits).

This can be further improved using the *fast Euclidean algorithm*, which uses a divide-and-conquer approach to compute the product $Q = Q_{k-1} \cdots Q_1$ by splitting the product in half and recursively computing each half using what is known as a *half-gcd* algorithm. One can then compute $R_k = QR_1$, $S_k = QS_1$, and $T_k = QT_q$. The details are somewhat involved (care must be taken when determining how to split the product in a way that balances the work evenly and k is not known ahead of time), but this yields a recursive running time of

$$T(n) = 2T(n/2) + O(\mathsf{M}(n)) = O(\mathsf{M}(n) \log n).$$

See [2, Chapter 11] for further information.

The extended Euclidean algorithm works in any Euclidean ring, that is, a ring with a norm function that allows us to use Euclidean division to write $a = qb + r$ with r of norm

strictly less than b (for $b \neq 0$). In particular, this includes polynomial rings, in which the norm of a polynomial is simply its degree. Thus we can compute the inverse of a polynomial modulo another polynomial, provided the two polynomials are relatively prime. With the fast Euclidean algorithm the time to compute the gcd of two polynomials over a finite field $\mathbb{F}_q$, and the time to compute the inverse of one such polynomial modulo another, is $O(\mathsf{M}(d(n+\log d))\log d)$, where $n = \log q$ and d is the maximum degree of the polynomials. This bound assumes Kronecker substitution is used to convert polynomial multiplication to integer multiplication.

We note that the gcd of two elements in a Euclidean ring is unique only up to multiplication by a unit, but if the gcd is itself a unit (as will be the case if the inputs are relatively prime), we can always normalize so that the gcd is 1.

4.4 Exponentiation (scalar multiplication)

Let a be a positive integer. In a multiplicative group, the computation

$$g^a = \underbrace{gg\cdots g}_{a}$$

is known as *exponentiation*. In an additive group, this is equivalent to

$$ag = \underbrace{g+g+\cdots+g}_{a},$$

and is called *scalar multiplication*. The same algorithms are used in both cases, and most of these algorithms were first developed in a multiplicative setting (the multiplicative group of a finite field), so we call them exponentiation algorithms, even though we will often be applying them to groups written additively. It is more convenient to describe the algorithms using additive notation (fewer superscripts), so we do so. We use P, Q, and R to denote group elements and use a to denote a a positive integer exponent (scalar).

The oldest and most commonly used exponentiation algorithm is the "double-and-add" method, also known as left-to-right binary exponentiation. Given an element P and a positive integer a with binary representation $a = \sum 2^i a_i$ with $a_i \in \{0,1\}$, we compute the scalar multiple $Q = aP$ as follows:

```
def double_and_add (P, a):
    a=a.digits(2); n=len(a)       # represent a in binary using n bits
    Q=P; i=n-2                    # start 1 bit below the high bit
    while i >= 0:
        Q=Q+Q                     # double
        if a[i]==1: Q = Q + P     # add
        i -= 1
    return Q
```

Alternatively, we may use an "add-and-double" method, also known as right-to-left binary exponentiation.

```
def add_and_double (P, a):
    a=a.digits(2); n=len(a)       # represent a in binary using n bits
    Q=0; R=P; i=0                 # start with the low bit
    while i < n:
        if a[i]==1: Q = Q + R     # add
```

```
        R=R+R                       # double
        i += 1
    return Q
```

The number of group operations required is effectively the same for both algorithms. If we ignore the first addition in the `add_and_double` algorithm (which could be replaced by an assignment, since initially $Q = 0$), both algorithms use precisely

$$n + \mathrm{wt}(a) - 2 \le 2n = O(n)$$

group operations, where $\mathrm{wt}(a)$ is the *Hamming weight* of a, the number of 1's in its binary representation. Up to the constant factor 2, this is asymptotically optimal, and it implies that exponentiation in a finite field has complexity $O(n\mathsf{M}(n))$.

Nevertheless, researchers have gone to great lengths to make the constant factor 2 as close to the optimal value 1 as possible, and there are a wide variety of optimizations that are used in practice. Here we give just one example, windowed exponentiation, which is able to reduce the constant factor to $1 + o(1)$.

4.4.1 Fixed-window exponentiation

Let the positive integer s be a *window size* and write a as

$$a = \sum a_i 2^{si}, \quad (0 \le a_i < 2^s).$$

This is equivalent to writing a in base 2^s. With fixed-window exponentiation, one first precomputes multiples dP for each of the "digits" $d \in [0, 2^s - 1]$ that may appear in the base-2^s expansion of a. One then uses a left-to-right approach as in the double-and-add algorithm, except now we double s times and add the appropriate multiple $a_i P$.

```
def fixed_window (P, a, s):
    a=a.digits(2^s)                             # write a in base 2^s
    R = [0*P, P]
    for i in range(2,2^s): R.append(R[-1]+P)    # precompute digits
    Q = R[a[-1]]                                # copy the top digit
    i = len(a)-2
    while i >= 0:
        for j in range(0,s): Q = Q + Q          # double s times
        Q = Q + R[a[i]]                         # add the next digit
        i -= 1
    return Q
```

In the algorithm above we precompute multiples of P for every possible digit that might occur. As an optimization one could examine the digits a_i and only precompute the multiples of P that are actually necessary.

Let n be the number of bits in a and let $m = \lceil n/s \rceil$ be the number of base-2^s digits a_i. The precomputation step uses $2^s - 2$ additions (we get $0P$ and $1P$ for free), there are $m - 1$ additions of multiples of P corresponding to digits a_i (when $a_i = 0$ these cost nothing), and there are a total of $(m - 1)s$ doublings. This yields an upper bound of

$$2^s - 2 + m - 1 + (m - 1)s \approx 2^s + n/s + n$$

group operations. If we choose $s = \lg n - \lg\lg n$, we obtain the bound

$$n/\lg n + n/(\lg n - \lg\lg n) + n = n + O(n/\log n),$$

which is $(1 + o(1))n$ group operations.

4.4.2 Sliding-window exponentiation

The sliding-window algorithm modifies the fixed-window algorithm by "sliding" over blocks of 0s in the binary representation of a. There is still a window size s, but a is no longer treated as an integer written in a fixed base 2^s. Instead, the algorithm scans the bits of the exponent from left to right, assembling "digits" of at most s bits with both high and low bits set: with a sliding window of size 3 the bit-string 110100010010100 could be broken up as 11|0|1|000|1|00|101|00 with 4 nonzero digits, where as a fixed window approach would use 110|100|010|010|100 with 5 nonzero digits.

This improves the fixed-window approach in two ways: first, it is only necessarily to precompute odd digits, and second, depending on the pattern of bits in a, sliding over the zeros may reduce the number of digits used, as in the example above. In any case, the sliding-window approach is never worse than the fixed-window approach, and for $s > 2$ it is always better.

4.5 Root-finding in finite fields

Let $f(x)$ be a polynomial in $\mathbb{F}_q[x]$. We wish to find a solution to $f(x) = 0$ that lies in $\mathbb{F}_q$. As an important special case, this will allow us to compute square roots using $f(x) = x^2 - a$, and, more generally, rth roots.[2]

The algorithm we give here was originally proposed by Berlekamp for prime fields [1], and then refined and extended by Rabin [4], whose presentation we follow here. The algorithm is probabilistic, and is one of the best examples of how randomness can be exploited in a number-theoretic setting. As we will see, it is quite efficient, with an expected running time that is quasi-quadratic in the size of the input. By contrast, no deterministic polynomial-time algorithm for root-finding is known, even for the special case of computing square roots.[3]

4.5.1 Randomized algorithms

Probabilistic algorithms are typically classified as one of two types: *Monte Carlo* or *Las Vegas*. Monte Carlo algorithms are randomized algorithms whose output may be incorrect, depending on random choices that are made, but whose running time is bounded by a function of its input size, independent of any random choices. The probability of error is required to be less than $1/2 - \epsilon$, for some $\epsilon > 0$, and can be made arbitrarily small be running the algorithm repeatedly and using the output that occurs most often. In contrast, a Las Vegas algorithm always produces a correct output, but its running time may depend on random choices. The running time of a Las Vegas algorithm need not be bounded, but we do require its expected running time to be finite. As a trivial example, consider an algorithm to compute $a + b$ that first flips a coin repeatedly until it gets a head and then computes $a + b$ and outputs the result. The running time of this algorithm may be arbitrarily long, even when computing $1 + 1 = 2$, but its *expected* running time is $O(n)$, where n is the size of the inputs.

[2] An entirely different approach to computing rth roots using discrete logarithms is explored in Problem Set 2. It has better constant factors when the r-power torsion subgroup of $\mathbb{F}_q^*$ is small (which is usually the case), but is asymptotically slower then the algorithm presented here in the worst case.

[3] Deterministic polynomial-time bounds for root-finding can be proven in various special cases, including the computation of square-roots, if one assumes a generalization of the Riemann hypothesis.

Las Vegas algorithms are generally preferred, particularly in mathematical applications. Note that any Monte Carlo algorithm whose output can be verified can always be converted to a Las Vegas algorithm (just run the algorithm repeatedly until you get an answer that is verifiably correct). The root-finding algorithm we present here is of the Las Vegas type.

4.5.2 Factoring with gcd's

Recall that the Frobenuis automorphism π of finite field $\mathbb{F}_q$ is defined by $\pi(x) = x^q$. It has the property that if x is any element of $\overline{\mathbb{F}}_q$, then $\pi(x) = x$ if and only if x lies in $\mathbb{F}_q$. It follows that the roots of the polynomial $x^q - x$ are precisely the elements of $\mathbb{F}_q$. Thus if $f(x)$ is an arbitrary polynomial in $\mathbb{F}_q[x]$, then the polynomial

$$g = \gcd(f, x^q - x)$$

has the form $\prod_i (x - \alpha_i)$, where the α_i range over all the distinct roots of f that lie in $\mathbb{F}_q$. If f has no roots in $\mathbb{F}_q$ then g will have degree 0, and otherwise we can reduce the problem of finding a root of f to the problem of finding a root of g, a polynomial whose roots are distinct and known to lie in $\mathbb{F}_q$.

In order to compute $g = \gcd(f, x^q - x)$ efficiently, one does *not* compute $x^q - x$ and then take the gcd with f; this would take time exponential in $n = \log q$. Instead, one computes $x^q \bmod f$ by exponentiating the polynomial x in the ring $\mathbb{F}_q[x]/(f)$, whose elements are uniquely represented by polynomials of degree less than $d = \deg f$. Each multiplication in this ring involves the computation of a product in $\mathbb{F}_q[x]$ followed by a reduction modulo f. This reduction is achieved using Euclidean division, and can be accomplished using two polynomial multiplications once an approximation to $1/f$ has been precomputed, see §4.2, and is within a constant factor of the time to multiply two polynomials of degree d in any case. The total cost of each multiplication in $\mathbb{F}_q[x]/(f)$ is thus $O(\mathsf{M}(d(n + \log d)))$, assuming that we use Kronecker substitution to multiply polynomials. The time to compute $x^q \bmod f$ using any of the exponentiation algorithms described in §4.4 is then $O(n\mathsf{M}(d(n + \log d)))$.

Once we have computed $x^q \bmod f$, we subtract x and compute $g = \gcd(f, x^q - x)$. Using the fast Euclidean algorithm, this takes $O(\mathsf{M}(d(n + \log d)) \log d)$ time. Thus the total time to compute g is $O(\mathsf{M}(d(n + \log d))(n + \log d))$, which we will see dominates the cost of root-finding. Typically $\log d = O(n)$ and this simplifies to $O(n\mathsf{M}(dn))$. Having computed g, we seek to factor it into two polynomials of lower degree by again applying a gcd, with the goal of eventually obtaining a linear factor, which will yield a root.

Assuming that q is odd (which we do), we may factor the polynomial $x^q - x$ as

$$x^q - x = x(x^s - 1)(x^s + 1),$$

where $s = (q - 1)/2$. Ignoring the root 0 (which we can easily check separately), this factorization splits $\mathbb{F}_q^*$ precisely in half: the roots of $x^s - 1$ are the elements of $\mathbb{F}_q^*$ that are quadratic residues, and the roots of $x^s + 1$ are the elements of $\mathbb{F}_q^*$ that are not. If we compute

$$h = \gcd(g, x^s - 1),$$

we obtain a divisor of g whose roots are precisely the roots of g that are quadratic residues. If we suppose that the roots of g are as likely as not to be quadratic residues, we should expect the degree of h to be approximately half the degree of g, and so long as the degree of h is strictly between 0 and $\deg g$, one of h or g/h is a polynomial of degree at most half the degree of g and whose roots are all roots of our original polynomial f.

To make further progress, and to obtain an algorithm that is guaranteed to work no matter how the roots of g are distributed in $\mathbb{F}_q$, we take a probabilistic approach. Rather than using the fixed polynomial $x^s - 1$, we consider random polynomials of the form

$$(x + \delta)^s - 1,$$

where δ is uniformly distributed over $\mathbb{F}_q$. We claim that if α and β are any two nonzero roots of g, then with probability $1/2$, exactly one of these is a root $(x + \delta)^s - 1$. It follows from this claim that so long as g has at least 2 distinct nonzero roots, the probability that the polynomial $h = \gcd(g, (x + \delta)^s + 1)$ is a proper divisor of g is at least $1/2$.

Let us say that two elements $\alpha, \beta \in \mathbb{F}_q$ are of *different type* if they are both nonzero and $\alpha^s \neq \beta^s$. Our claim is an immediate consequence of the following theorem from [4].

Theorem 4.1 (Rabin)**.** *For every pair of distinct $\alpha, \beta \in \mathbb{F}_q$ we have*

$$\#\{\delta \in \mathbb{F}_q : \alpha + \delta \text{ and } \beta + \delta \text{ are of different type}\} = \frac{q-1}{2}.$$

Proof. Consider the map $\phi(\delta) = \frac{\alpha+\delta}{\beta+\delta}$, defined for $\delta \neq -\beta$. We claim that ϕ is a bijection form the set $\mathbb{F}_q \backslash \{-\beta\}$ to the set $\mathbb{F}_q \backslash \{1\}$. The sets are the same size, so we just need to show surjectivity. Let $\gamma \in \mathbb{F}_q - \{1\}$, then we wish to find a solution $x \neq -\beta$ to $\gamma = \frac{\alpha+x}{\beta+x}$. We have $\gamma(\beta + x) = \alpha + x$ which means $x - \gamma x = \gamma\beta - \alpha$. This yields $x = \frac{\gamma\beta-\alpha}{1-\gamma}$, which is not equal to $-\beta$, since $\alpha \neq \beta$. Thus ϕ is surjective.

We now note that

$$\phi(\delta)^s = \frac{(\alpha + \delta)^s}{(\beta + \delta)^s}$$

is -1 if and only if $\alpha + \delta$ and $\beta + \delta$ are of different type. The elements $\gamma = \phi(\delta)$ for which $\gamma^s = -1$ are precisely the non-residues in $\mathbb{F}_q \backslash \{1\}$, of which there are exactly $(q-1)/2$. □

We now give the algorithm.

Root-finding Algorithm
Input: A polynomial $f \in \mathbb{F}_q[x]$.
Output: An element $r \in \mathbb{F}_q$ such that $f(r) = 0$, or `null` if no such r exists.

1. If $f(0) = 0$ then return 0.
2. Compute $g = \gcd(f, x^q - x)$.
3. If $\deg g = 0$ then return `null`.
4. While $\deg g > 1$:
   a. Pick a random $\delta \in \mathbb{F}_q$.
   b. Compute $h = \gcd(g, (x + \delta)^s - 1)$.
   c. If $0 < \deg h < \deg g$ then replace g by h or g/h, whichever has lower degree.
5. Return $r = -b/a$, where $g(x) = ax + b$.

It is clear that the output of the algorithm is always correct, since every root of the polynomial g computed in step 2 is a root of f, and when g is updated in step 4c it is always replaced by a proper divisor. We now consider its complexity.

4.5.3 Complexity

It follows from Theorem 4.1 that the polynomial h computed in step 4b is a proper divisor of g with probability at least $1/2$, since g has at least two distinct nonzero roots $\alpha, \beta \in \mathbb{F}_q$. Thus the expected number of iterations needed to obtain a proper factor h of g is bounded by 2, and the expected cost of obtaining such an h is $O(\mathsf{M}(e(n + \log e))(n + \log e))$, where $n = \log q$ and $e = \deg g$, and this dominates the cost of the division in step 4c.

Each time g is updated in step 4c its degree is reduced by at least a factor of 2. It follows that the expected total cost of step 4 is within a constant factor of the expected time to obtain a proper divisor of the initial polynomial g, which is $O(\mathsf{M}(d(n + \log d))(n + \log d))$, where $d = \deg f$. Step 2 satisfies the same complexity bound, and the total expected running time is $O(\mathsf{M}(d(n + \log d))(n + \log d))$, which is $O(n\mathsf{M}(dn))$ when $\log d = O(n)$.

The algorithm can easily be modified to find all the distinct roots of f, by modifying step 4c to recursively find the roots of both h and g/h. In this case the amount of work done at each level of the recursion tree is bounded by $O(\mathsf{M}(d(n + \log d))(n + \log d))$. Bounding the depth of the recursion is somewhat more involved, but one can show that with very high probability the degrees of h and g/h are approximately equal and that the expected depth of the recursion is $O(\log d)$. Thus we can find all the distinct roots of f in

$$O(\mathsf{M}(d(n + \log d))(n + \log d) \log d)$$

expected time. When $\log d = O(n)$ this becomes $O(n\mathsf{M}(dn) \log d)$.

Assuming that d is less than the charcteristic of $\mathbb{F}_q$, one can easily determine the multiplicity of each root of f: a root α of f occurs with multiplicity k if and only if α is a root of $f^{(k)}$ but not a root of $f^{(k+1)}$, where $f^{(k)}$ denotes the kth derivative of f. The time to perform this computation is negligible compared to the time to find the distinct roots.

References

[1] Elwyn R. Berlekamp, *Factoring polynomials over large finite fields*, Mathematics of Computation **24** (1970), 713–735.

[2] Joachim von zur Gathen and Jürgen Garhard, *Modern Computer Algebra*, second edition, Cambridge University Press, 2003.

[3] Henri Cohen and Gerhard Frey, *Handbook of Elliptic and Hyperelliptic Curve Cryptography*, CRC Press, 2006.

[4] Michael O. Rabin, *Probabilistic algorithms in finite fields*, SIAM Journal of Computing **9** (1980), 273–280.