COLLOQUIUM MATHEMATICUM

VOL. 116 2009 NO. 1

WHAT IS THE INVERSE OF REPEATED SQUARE AND MULTIPLY ALGORITHM?

BY

H. GOPALKRISHNA GADIYAR (Chennai),
K. M. SANGEETA MAINI (Chennai), R. PADMA (Chennai)
and MARIO ROMSY (Neubiberg)

Abstract. It is well known that the repeated square and multiply algorithm is an efficient way of modular exponentiation. The obvious question to ask is if this algorithm has an inverse which would calculate the discrete logarithm and what is its time compexity. The technical hitch is in fixing the right sign of the square root and this is the heart of the discrete logarithm problem over finite fields of characteristic not equal to 2. In this paper a couple of probabilistic algorithms to compute the discrete logarithm over finite fields and their time complexity are given by bypassing this difficulty. One of the algorithms was inspired by the famous $3x + 1$ problem.

1. Introduction. Let p be an odd prime number and let a be a primitive root. Then any $b \in (\mathbb{Z}/p\mathbb{Z})^*$ can be expressed as

$$b \equiv a^n \bmod p \tag{1}$$

for a unique integer n with $1 \le n \le p-1$. n is called the *index* or *discrete logarithm* of b to base a modulo p. The *discrete logarithm problem* over prime fields is to find n given a and b modulo p. When p is sufficiently large and random, the discrete logarithm problem is believed to be computationally difficult and hence is the basis of the security of many cryptographic algorithms like the Diffie–Hellman key exchange protocol, El Gamal Cryptosystem, El Gamal signature scheme etc. The well known algorithms to compute the discrete logarithm problem are the baby step – giant step method, Pollard's rho method, Pohlig–Hellman method and the index calculus method [7], [8], [10], [14], [17]. Among these algorithms the first two are square root algorithms. The Pohlig–Hellman method works efficiently for those primes p for which all the prime factors of $p-1$ are small, since it reduces the problem to a number of discrete logarithm problems in (smaller) groups of prime order. So in practice one first applies the Pohlig–Hellman reduction and then for example the Pollard rho method. The index calculus method is a subexponential time algorithm. All these algorithms find n modulo $p-1$.

2000 *Mathematics Subject Classification*: 11Y16, 68Q25, 68W40.
Key words and phrases: discrete logarithm, Legendre symbol, $3x + 1$ problem.

DOI: 10.4064/cm116-1-1

There is an analysis of the discrete logarithm problem using p-adic methods in [4] which links the problem to connections, cocycles and crystalline cohomology. In [13], a straightforward formula for solving the discrete logarithm problem in certain cases is obtained using the Fermat quotient and its generalizations. For an extensive bibliography on this problem look at the website http://www.cs.uwaterloo.ca/~shallit/bib/dlog.bib.

So far, there have been no known polynomial time algorithm to compute the discrete logarithm for a random prime p. There are lower bounds obtained by [9] and [16] for "generic" discrete logarithm problems. Shoup [16] proved that any "generic" algorithm to calculate the discrete logarithm must perform $\Omega(p^{1/2})$ group operations, where p is the largest prime dividing the order of the group.

We know that the modular exponentiation $a^n \bmod p$ $(= b)$ in (1) is performed efficiently using the repeated square and multiply algorithm. In this paper we ask the question of what is the inverse of this algorithm and how much time it takes. As an answer to this question we give a couple of probabilistic algorithms that compute the discrete logarithm.

The Legendre symbol of b determines the least significant bit (l.s.b.) of n which is the index of b to base a in (1). When the l.s.b. of n is 0, the next least significant bit of n is obtained by extracting the "correct" square root. The r least significant bits can be unambiguously and efficiently determined if $p - 1 = 2^r s$, where s is odd and $r \geq 1$ [11]. The ambiguity starts from the $(r+1)$th least significant bit onwards.

In this paper we give two probabilistic algorithms which bypass this ambiguity and compute the discrete logarithm over prime fields. The first algorithm can be thought of as a randomized inverse of the repeated square and multiply algorithm. The second algorithm was inspired by the famous $3x + 1$ problem. The algorithms are immediately extendable to other finite fields (including finite fields of characteristic 2 with a slight modification). In Section 2 we explain how the properties of the Legendre symbol can be used to determine the r least significant bits of the index n and also give a time estimate for computing square roots modulo p. In Section 3 we present our main algorithm and give numerical examples over a prime field. A modification of the algorithm for finite fields of characteristic 2 is also given along with a couple of examples. This algorithm is extendable to the elliptic curve discrete logarithm problem. In Section 4 we state the $3x + 1$ problem and a variant of the algorithm given in Section 3 is presented with an example. In Section 5 we analyze the time complexity of the algorithms.

2. The Legendre symbol and square roots. The Legendre symbol $\left(\frac{x}{p}\right)$, for any integer x, is defined as follows. If $p \,|\, x$, then $\left(\frac{x}{p}\right) = 0$, and

for x with $(x, p) = 1$,

$$\left(\frac{x}{p}\right) = \begin{cases} 1 & \text{if } x^{(p-1)/2} \equiv 1 \bmod p, \\ -1 & \text{if } x^{(p-1)/2} \equiv -1 \bmod p. \end{cases} \tag{2}$$

The definition (2) of the Legendre symbol can be restated as follows. $\left(\frac{x}{p}\right)$ is equal 1 if x is a quadratic residue (that is, a square) modulo p, and is equal to -1 if x is a quadratic nonresidue (that is, a nonsquare) modulo p. Since a is a primitive root,

$$\left(\frac{a}{p}\right) \equiv a^{(p-1)/2} \equiv -1 \bmod p. \tag{3}$$

Using the property of the Legendre symbol

$$\left(\frac{xy}{p}\right) = \left(\frac{x}{p}\right)\left(\frac{y}{p}\right), \tag{4}$$

one has

$$\left(\frac{b}{p}\right) = \left(\frac{a^n}{p}\right) = \left(\frac{a}{p}\right)^n = (-1)^n. \tag{5}$$

Thus the Legendre symbol of b determines whether n is odd or even. In other words, the Legendre symbol of b determines the l.s.b. of n. Let us write n in its binary representation

$$n = n_0 + 2n_1 + 2^2 n_2 + \cdots + 2^k n_k, \tag{6}$$

where each n_i is 0 or 1. If n is even (or odd), then $n_0 = 0$ (or 1). The next bit n_1 is determined by dividing n (or $n-1$) by 2 and checking whether $n/2$ (or $(n-1)/2$) is even or odd. In terms of a and b modulo p, this amounts to finding the "correct" square root of b (or b/a) modulo p.

Let us assume that b is a quadratic residue modulo p. Then the square roots of b are $b^{1/2}$ and $-b^{1/2}$ and hence from (3), the index of the square roots to base a are $n/2$ and $(p-1)/2 + n/2$ modulo $p-1$. If $p \equiv 1 \bmod 4$, then the l.s.b. of $(p-1)/2$ is 0. Hence the l.s.b. of $n/2$ equals the l.s.b. of $(p-1)/2 + n/2$. In other words, the Legendre symbol of $b^{1/2}$ or $-b^{1/2}$ will determine the value of the bit n_1 unambiguously. More generally, if $p - 1 = 2^r s$, where s is odd and $r \geq 1$, then the r least significant bits of n can be unambiguously determined. See for example [11]. The difficulty arises from the $(r+1)$th least significant bit onwards.

Let us explain this with the case of $r = 1$. In this case $p \equiv 3 \bmod 4$. Hence the l.s.b. of $(p-1)/2$ is 1. Now, if the l.s.b. of $n/2$ is 1, then the l.s.b. of $(p-1)/2 + n/2$ is 0 and vice versa. Thus it is not possible to determine the correct value of the bit n_1.

Note that the Legendre symbol can be calculated in polynomial time $(O(\log^2 p))$ [8]. If $p \equiv 3 \bmod 4$, then $r = 1$, and in this case the square roots of a quadratic residue x are $x^{(s+1)/2} \bmod p$ and $-x^{(s+1)/2} \bmod p$. Hence the square root can be calculated in polynomial time $(O(\log^3 p))$. If $p \equiv 1$

mod 4, then there is a polynomial time algorithm ($O(\log^4 p)$) to compute a square root of a quadratic residue, provided we could find a quadratic nonresidue modulo p ([8], [1]). Since a is a primitive root, it is a quadratic nonresidue and hence the square roots of a quadratic residue x modulo p can be calculated deterministically in polynomial time.

3. The repeated square and multiply algorithm and its inverse. In this section we give our main algorithm. In Section 3.1 we explain how modular exponentiation is done using the repeated square and multiply algorithm and describe the difficulty in trying to invert the process. In Section 3.2 we give our main algorithm for computing the discrete logarithm. Section 3.3 explains the algorithm and Section 3.4 gives two examples. In Section 3.5 we generalize the algorithm to all finite fields including finite fields of characteristic 2. This section also contains examples of computing discrete logarithm over finite fields of characteristic 2. Finally, Section 3.6 contains a note on the elliptic curve discrete logarithm problem over binary fields.

3.1. *Modular exponentiation and its inverse.* The repeated square and multiply algorithm is used to compute modular exponentiation in polynomial time. Let us quickly recall how we compute b given a, n and p as in (1) using this algorithm. If n has the binary representation as in (6), then let $a_0 = a$, $b_0 = 1$ and inductively compute $a_j = a_{j-1}^2 \bmod p$ and $b_j = b_{j-1}a_j \bmod p$ if $n_j = 1$, and $b_j = b_{j-1}$ if $n_j = 0$, for $j = 1$ to k. Then b_k is the value of $a^n \bmod p$. That is,

$$b \equiv a^n \equiv (a^{2^k} \bmod p)^{n_k} \cdots (a^{2^1} \bmod p)^{n_1} (a^{2^0} \bmod p)^{n_0} \bmod p. \tag{7}$$

It is clear that the inverse of this algorithm is to divide and repeatedly extract square root. Division is done when the bit n_i is 1, just as multiplication is done in the repeated square and multiply algorithm when n_i is 1. Note that knowing the bits n_i is equivalent to knowing the value of n. Also, if the "correct" square root can be taken every time, it will fix the correct value of the bit n_i and hence n can be calculated in polynomial time. But the difficulty is in fixing the correct square root. In fact, deciding which is the "correct" square root is a much harder problem and allowing even a weak oracle for finding the "correct" square root would make the discrete logarithm problem "easy" [2].

We know from Section 2 that the r least significant bits n_i can be unambiguously determined in polynomial time by dividing by a if the Legendre symbol is -1 and extract any square root or by just extracting any square root if the Legendre symbol is 1. From the $(r+1)$th least significant bit onwards, we do not know which is the right bit. Now we give our probabilistic algorithm which bypasses this problem.

3.2. *Algorithm to compute the discrete logarithm*

INPUT: $a,\ b,\ p$, where $a^n \equiv b \bmod p$.
OUTPUT: n.

STEP 1. Choose an integer B and create Table I consisting of the pairs $(a^{k_j} \bmod p,\ k_j)$ where $j \leq B$. Here $\{k_j\}$ is any subsequence of integers. For example, $k_j = j$ or $k_j = 2^j$.
STEP 2. Initialize $i \leftarrow 1$, $l \leftarrow 1$, $b[1] \leftarrow b$, $m_1[1] \leftarrow n$, $c_1[1] \leftarrow b$, $c_2[1] \leftarrow b$ and $m_2[1] \leftarrow n$. Table II will consist of $(b[i], m_1[i])$ and Table III will consist of $(c_1[l], c_2[l], m_2[l])$. Also initialize $k \leftarrow 0$, $m \leftarrow n$.

STEP 3. (i) If $\left(\frac{b}{p}\right) = -1$ then goto Step 4.
(ii) If $\left(\frac{b}{p}\right) = 1$ then goto Step 6.

STEP 4. (i) $b \leftarrow b/a \bmod p$ and $m \leftarrow m - 1$.
(ii) Goto Step 5.
(iii) If Step 5 does not solve for n, $i \leftarrow i + 1$,
store $b[i] \leftarrow b$ and $m_1[i] \leftarrow m$ in Table II.
(iv) Goto Step 6.

STEP 5. (i) If $b \equiv a^{k_j} \bmod p$ for any $j \leq B$ in Table I, Solve(m, k_j, k).
(ii) If $b \equiv b[j] \bmod p$ for any j in Table II, Solve$(m, m_1[j], k)$.
(iii) If $b \equiv c_1[j]$ or $c_2[j] \bmod p$ for any j in Table III,
Solve$(m, m_2[j], k)$.

STEP 6. (i) $b \leftarrow b^{1/2} \bmod p$ and $m \leftarrow (m/2)$, $k \leftarrow k + 1$. Goto Step 5.
(ii) If Step 5 does not solve for n, $b \leftarrow p - b \bmod p$, Goto Step 5.
(iii) If Step 5 does not solve for n, $l \leftarrow l + 1$,
store $c_1[l] \leftarrow b$, $c_2[l] \leftarrow p - b$ and $m_2[l] \leftarrow m$ in Table III.
(iv) $b \leftarrow c_1[l]$ or $c_2[l]$ randomly.
(v) Goto Step 3.

SOLVE()
Solve(x, y, t): Solve the linear congruence:

$$2^t x \equiv 2^t y \bmod p - 1. \tag{8}$$

Return n

3.3. *Explanation of the algorithm.* Table I consists of B precomputed powers of a and the corresponding discrete logarithms. Table II consists of intermediate values of b and the corresponding values of m as a function of n when the Legendre symbol is -1. Table III consists of the intermediate values of b which are square roots and the corresponding values of m as a function of n when the Legendre symbol is 1. The current value of b in the loop equals $a^m \bmod p$ up to some signs and/or roots of unity or order 2^k. Since m is replaced by $m - 1$ or $m/2$, m is always a linear function of n.

The algorithm is probabilistic as we select one of the square roots randomly in Step 6.

The final step is to solve a linear congruence modulo $p-1$, if the new value of b matches any of the integers in Table I, II or III. If b matches a value in Table I, then x will be a linear function of n, and y will be a constant. In other cases, both x and y will be linear functions of n. Note that a linear congruence can be solved in polynomial time.

When b coincides with a value in Table I, the value of n can be uniquely obtained by solving the linear congruence.

When b coincides with a value in Table II or III, then the corresponding value of n can be found modulo $\left(\frac{p-1}{d}\right)$, for a divisor d of $p-1$. Hence there will be d solutions modulo $p-1$ and we have to choose the correct value of n modulo $p-1$ by trial and error. If d is too large, then one can start the algorithm again from somewhere in the middle of the tree where we can choose the other square root.

k counts the number of times we take square roots modulo p. Note that while solving the linear congruence, we multiply both sides by 2^k, so that the denominator of m gets cleared (as 2 is not invertible modulo $p-1$). This also takes care of the fact that in Table III, though we store two square roots, the exponent m is taken to be $m/2$, as whether we take $m/2$ or $m/2+(p-1)/2$, in Solve(), multiplication by 2^k would remove this ambiguity.

3.4. *Examples.* In this section we explain the algorithm in Section 3.2 with a small prime. Let $p = 103$. Then $a = 5$ is a primitive root of p.

Example 1. This is an example of collision with an element in Table I. Let $b = 84$. Let $B = 7$ and $k_j = 2^{j-1} \bmod p$ for $j = 1, \ldots 7$.

Table I

j	0	1	2	3	4	5	6
$5^{2^j} \bmod 103$	5	25	7	49	32	97	36

Discrete logarithm calculation for $b = 84$

b	$(\frac{b}{103})$	$\frac{b}{a} \bmod 103$	$b^{1/2},\ -b^{1/2} \bmod 103$	random sqrt	m
84	-1	58	–	–	$n-1$
58	1	–	26, 77	77	$(n-1)/2$
77	-1	36	–	–	$(n-1)/2-1$

Since $36 \equiv 5^{2^6} \bmod 103$ and $k = 1$ as we have taken square root only once, after multiplying both sides by 2 we get the congruence

(9) $$n - 3 \equiv 2^7 \bmod 102$$

and thus $n \equiv 29 \bmod 102$.

Note that the binary digits of 29 are given by (11101). Comparing these bits and the second column of the above table, we find that we have wrongly chosen the square root 77 in the second row, yet we are lucky to find the discrete logarithm at the third step itself, while if we had taken the correct path, it would have taken us 6 steps. There is a trading off between extraction of square roots and the precomputation of powers of a.

Note that Table II in the algorithm corresponds to the third and sixth columns of the above table, and Table III corresponds to the fourth and sixth columns.

EXAMPLE 2. This example gives a collision in Table II or III.

Discrete logarithm calculation for $b = 99$

b	$(\frac{b}{103})$	$b/a \bmod 103$	$b^{1/2},\ -b^{1/2} \bmod 103$	random sqrt	m
99	-1	61	–	–	$n-1$
61	1	–	24, 79	24	$(n-1)/2$
24	-1	46	–	–	$(n-3)/2$
46	1	–	47, 56	56	$(n-3)/4$
56	1	–	46, 57	–	$(n-3)/8$

Note that 46 in the fourth column matches the 46 in the third column. Equating the corresponding values of m, and multiplying both sides by 2^3 gives

(10) $$n - 3 \equiv 4(n - 3) \bmod 102.$$

Solving the linear congruence gives $n \equiv 3 \bmod 34$. Hence there are three possible values for n mod 102, namely, 3, 37 and 71. One can easily check that 37 is the correct value of n.

3.5. *Discrete logarithm over finite fields.* It is clear that the algorithm given in Section 3.2 is, just as it is, extendable to finite fields of characteristic $p \geq 3$, as the analogue of Legendre symbol and efficient computation of square roots exist in these fields [1].

When the characteristic of the finite field is 2, every element in the field is a square and every element has exactly one square root. [3] gives an efficient algorithm for computing square roots over finite fields of characteristic 2. Note that if we choose a normal basis, then the square root operation reduces to a mere cyclic shift [8]. Hence, our algorithm in Section 3.2 can be modified in this case as follows. Step 3 should randomly decide whether the l.s.b. of m is 1 or 0. That is,

STEP 3. (o) Randomly choose a bit 0 or 1.
(i) If the bit is 1 then goto Step 4.
(ii) If the bit is 0 then goto Step 6.

In Step 6, as there is only one square root, we need not perform (ii) and Table III will consist of only $c_1[l]$ and the corresponding value of m. Also Step (iv) should be skipped.

To be precise, the randomness in our algorithm for $p > 3$ in the selection of the square root in (iv) of Step 6 has been shifted to Step 3 where we randomly fix the l.s.b. of m for characteristic 2 fields.

We give two toy examples below.

Let us consider the finite field F_{2^7} with the primitive polynomial $f(x) = x^7 + x + 1$. Thus, x is the generator of the multiplicative group $F_{2^7}^*$ of the finite field F_{2^7}.

EXAMPLE 1. Let us create Table I with $B = 7$ and $k_j = 2^j$ for $j = 0, \ldots, 6$. Let

$$b = x^4 + x^3 + x^2 + 1.$$

Table I

j	0	1	2	3	4	5	6
$x^{2^j} \bmod f(x)$	x	x^2	x^4	$x(x+1)$	$x^2(x^2+1)$	$x(x^3+x+1)$	$x(x^3+1)$

Discrete logarithm calculation for $b = x^4 + x^3 + x^2 + 1$

b	random bit	$b/x \bmod f(x)$	$b^{1/2} \bmod f(x)$	m
$x^4 + x^3 + x^2 + 1$	0	–	$x^5 + x + 1$	$n/2$
$x^5 + x + 1$	1	$x^4(x^2 + 1)$	–	$n/2 - 1$
$x^4(x^2 + 1)$	1	$x^3(x^2 + 1)$	–	$n/2 - 2$
$x^3(x^2 + 1)$	1	$x^2(x^2 + 1)$	–	$n/2 - 3$

From Table I, $x^2(x^2 + 1) \equiv x^{2^4} \bmod f(x)$. Hence,

$$\frac{n}{2} - 3 \equiv 2^4 \bmod 127, \tag{11}$$

which gives $n \equiv 38 \bmod 127$.

EXAMPLE 2. Let

$$b = x^6 + x^5 + x^3 + x + 1.$$

Discrete logarithm calculation for $b = x^6 + x^5 + x^3 + x + 1$

b	random bit	$b/x \bmod f(x)$	$b^{1/2} \bmod f(x)$	m
$x^6+x^5+x^3+x+1$	0	–	$x^6+x^5+x^4+x^2+x+1$	$n/2$
$x^6+x^5+x^4+x^2+x+1$	1	$x^6+x^5+x^4+x^3+x$	–	$n/2-1$
$x^6+x^5+x^4+x^3+x$	1	$x^5+x^4+x^3+x^2+1$	–	$n/2-2$
$x^5+x^4+x^3+x^2+1$	0	–	$x^6+x^5+x^3+x+1$	$(n/2-2)/2$

The value of the original b and the square root in the fourth row are the same. Equating the corresponding values of m, we get

$$n \equiv \frac{1}{2}\left(\frac{n}{2} - 2\right) \bmod 127. \tag{12}$$

Solving this linear congruence gives $n \equiv 41 \bmod 127$.

3.6. *Elliptic curve discrete logarithm over binary fields.* A natural question to ask now is if our algorithm can be extended to elliptic curves as well. Since there is no analogue of the Legendre symbol, it is not possible to extend this algorithm to all elliptic curves. Note that in any group of odd order (written multiplicatively) every element has exactly one square root. So if we can calculate this square root efficiently, the modified algorithm of this section can be applied. When the elliptic curve is defined over finite fields of characteristic 2, there is an efficient algorithm for point-halving [5], [15] if the cardinality of the curve is odd. Hence in this case we can generalize the algorithm given in Section 3.5 to compute the discrete logarithm on such elliptic curves.

4. The $3x+1$ problem and the discrete logarithm problem. In this section we give a variant of the main algorithm given in Section 3.2. This was inspired by the famous $3x+1$ problem. The $3x+1$ problem, which was posed by L. Collatz, states that if

$$T(x) = \begin{cases} 3x+1 & \text{if } x \equiv 1 \bmod 2, \\ x/2 & \text{if } x \equiv 0 \bmod 2, \end{cases} \tag{13}$$

then for any positive integer x there exists an integer $k > 0$ with $T^k(x) = 1$. This problem remains open since 1937. For an annotated bibliography of this problem, see [6].

Note that if x is odd, the function T converts it into an even integer, while if x is even, it divides x by 2. The iteration will terminate once $T^k(x) = 2^l$ for some integers k and l.

In the algorithm we gave in Section 3.2, if the Legendre symbol is -1 (that is, the index of b is odd), we divided b by a so that the index of the

new value of b becomes even and if the Legendre symbol is 1 (that is, the index of b is even), we calculated the square roots of b so that the index is halved.

Now it is clear how we are going to modify the algorithm in Section 3.2. We will assume for the sake of simplicity that $(3, p-1) = 1$. If the Legendre symbol is -1, then compute $b^3 a \bmod p$. That is, in Step 4(i), we do

$$b \leftarrow b^3 a \bmod p \quad \text{and} \quad m \leftarrow 3m + 1 \tag{14}$$

and the rest of the algorithm goes as before.

4.1. *Example.* We explain our algorithm with an example, again with a small prime. Let us take $p = 101$. Then $a = 2$ is a primitive root of 101. Let $b = 72$. Let $B = 7$ and $k_j = 2^j$ for $j = 0, \ldots, 6$.

Table I

j	0	1	2	3	4	5	6
$2^{2^j} \bmod 101$	2	4	16	54	88	68	79

Discrete logarithm calculation for $b = 72$

b	$(\frac{b}{101})$	$b^3 a \bmod 101$	$b^{1/2}$, $-b^{1/2} \bmod 101$	random sqrt	m
72	-1	5	–	–	$3n+1$
5	1	–	45, 56	56	$(3n+1)/2$
56	1	–	37, 64	37	$(3n+1)/4$
37	1	–	21, 80	80	$(3n+1)/8$
80	1	–	22, 79	–	$(3n+1)/16$

Since $79 \equiv 2^{2^6} \bmod 101$, and $k = 4$, we have

$$3n + 1 \equiv 1024 \equiv 24 \bmod 100. \tag{15}$$

The solution of this linear congruence is given by $n \equiv 41 \bmod 100$.

5. Time complexity of the algorithms. Let us first look at the algorithm in Section 3.5 over finite fields of characteristic 2. Here we randomly choose to do division or extract square root. Let $\varrho : \mathcal{N} \to \{0, 1\}$ be the random decision function. Starting with $r_0 = b = a^n$, we have for $i \in \mathcal{N}$,

$$r_i = \begin{cases} \sqrt{r_{i-1}} & \text{if } \varrho(i) = 0, \\ r_{i-1}/a & \text{if } \varrho(i) = 1. \end{cases} \tag{16}$$

Let Table I consist of $B \in \mathcal{N}$ precomputed values and label them $r_{-B}, \ldots, r_{-1}$. Since we need to keep track of the exponents we store pairs of the form (a^k, k), hence write $(r_{-B}, k_{-B}), \ldots, (r_{-1}, k_{-1})$, where $\{k_j\}$ is the chosen subsequence of integers (as in Section 3.2). So starting with (r_0, m_0), $m_0 = n$,

we calculate a random walk by

$$(r_i, m_i) = \begin{cases} (\sqrt{r_{i-1}}, m_{i-1}/2) & \text{if } \varrho(i) = 0, \\ (r_{i-1}/a, m_{i-1} - 1) & \text{if } \varrho(i) = 1. \end{cases} \tag{17}$$

Then we look for a collision $r_i = r_j$ for $i \neq j$. Since n is unknown, the m_i are linear functions in n.

Note that in the algorithm given in Section 3.2, the random function will decide which square root will be taken. Since we store both square roots, we add a control bit b_i in each step where $b_i = 1$ means a square root was taken ($b_i = 0$ means a division). In step i, when we look for collisions of r_i with some previous element, we test $r_i = r_j$ if $b_j = 0$ and $r_i = \pm r_j$ if $b_j = 1$. (Note that if r_j is one square root then $-r_j$ is the other and that the associated linear functions are the same.) Hence in this case, we have

$$(r_i, m_i, b_i) = \begin{cases} (\min(\sqrt{r_{i-1}}, p - \sqrt{r_{i-1}}), m_{i-1}/2, 1) & \text{if } \left(\frac{r_{i-1}}{p}\right) = 1 \text{ and } \varrho(i) = 0, \\ (\max(\sqrt{r_{i-1}}, p - \sqrt{r_{i-1}}), m_{i-1}/2, 1) & \text{if } \left(\frac{r_{i-1}}{p}\right) = 1 \text{ and } \varrho(i) = 1, \\ (r_{i-1}/a, m_{i-1} - 1, 0) & \text{if } \left(\frac{r_{i-1}}{p}\right) = -1. \end{cases}$$

Compared with the algorithm given in Section 4, only the iteration function is changed.

We first recall both algorithms of Pollard in order to compare our algorithms with them. Here we give the versions from [12] to calculate the discrete logarithm mod p (it is clear how to modify them to work in a general group).

Pollard rho method. Given a, p and $b \equiv a^n \pmod{p}$ one starts with $x_0 = 1$ and calculates the sequence

$$x_i = \begin{cases} bx_{i-1} \bmod p & \text{if } 0 < x_{i-1} < \frac{1}{3}p, \\ x_{i-1}^2 \bmod p & \text{if } \frac{1}{3}p < x_{i-1} < \frac{2}{3}p, \\ ax_{i-1} \bmod p & \text{if } \frac{2}{3}p < x_{i-1} < p. \end{cases} \tag{18}$$

and looks for collisions $x_i = x_j$ for $i \neq j$.

Pollard kangaroo method. Given $a, p, b \equiv a^n \pmod{p}$ and an interval $[A, B]$ with $n \in [A, B]$, we choose a bound N, e.g. $N = \lceil \sqrt{B - A} \rceil$. Starting with $t_0 = a^B$ we calculate $t_1, \ldots, t_N$ by

$$t_i = t_{i-1} * a^{f(t_{i-1})},$$

where f is a function that takes random integer values of mean $\sqrt{B - A}$. With $w_0 = b$ one computes a second sequence $(w_i)_i$ by

$$w_i = w_{i-1} * a^{f(w_{i-1})}$$

until $w_j = t_N$ for some j (or $\sum_i f(w_i) > B - A + \sum_i^N f(t_i)$, where one has to start again).

The algorithms of Section 3 and 4 are kind of random walks and look quite similar to the standard Pollard rho method. Hence the expected number of steps should be about $O(\sqrt{p})$ (experimental results seem to confirm this).

Of course, the motivation of our algorithms, inverting the repeated square and multiply algorithm, is different to Pollard's algorithms. From the algorithmic point of view the difference lies in the iteration functions.

To illustrate the different "jump behaviours" we adopt the scheme of [18, Section 3] where every element from the cyclic group $\langle a \rangle$ is viewed as lying on a circle and a^{i+1} is one step to the right (counterclockwise) from a^i.

Figure 1 (cf. [18, Section 3, Figure 2]) illustrates the rho method (characterized by random jumps within the whole group $\langle a \rangle$) and the kangaroo method (characterized by small jumps).

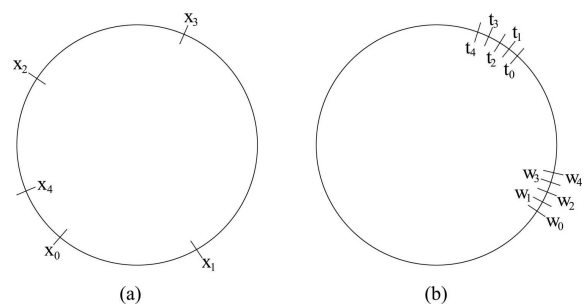

Fig. 1. (a) Pollard rho and (b) Pollard kangaroo methods

The real inverse of the repeated square and multiply algorithm would consist of tiny steps (division by a) and jumps (taking the square root, i.e. halving the index) of decreasing size (see Figure 2(a)). Our algorithms still consist of tiny steps and jumps, but the jumpsize will vary, depending on the random choice of the square root. There will be a jump to the right if and only if a wrong square root is chosen (see Figure 2(b)).

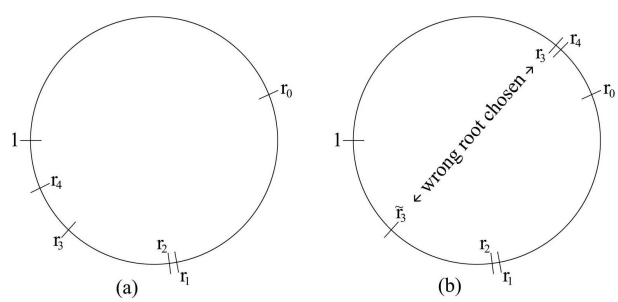

Fig. 2. (a) Real inverse of repeated square and multiply algorithm and (b) probabilistic version

Finally, we want to remark that our algorithms are probably slower than both algorithms of Pollard, since taking square roots and division are (in general) slower than squaring and multiplication. On the one hand, ϱ being a "random decision function" gives the walk a random pattern. On the other hand, Floyd's cycle detection method, which keeps the memory requirements of the rho method constant, will not apply here as the function ϱ depends only on the number of steps, i.e. the sequence will not fall into a cycle at all. One could modify our algorithms so that the decision function ϱ is dependent on the previous element of the sequence as it is the case in Pollard's algorithms. Consequently, the sequence would fall into a cycle after the first collision and therefore Floyd's method could be applied. Using this modification we would not be able to check whether a collision with an "unchosen" square root occurs.

6. Conclusion and future directions. In this paper we have asked what is the inverse of repeated square and multiply algorithm and given a couple of probabilistic algorithms to compute the discrete logarithm. The algorithms are parallelizable. It is noted that the algorithm given for binary finite fields can also be extended to elliptic curves over such fields. Analysis of the algorithms shows that these algorithms are of square root type. Though the algorithms do not appear to be more efficient than the already existing ones, both the original algorithm and the $3x+1$ version might be subsumed under a general scheme, in which one uses an affine map $x \mapsto ax + b$, with $(a, p-1) = 1$ if x is odd, and $x \mapsto x/2$ if x is even, and it would be worthwhile to see if some interesting algebra could be uncovered.

REFERENCES

[1] E. Bach and J. Shallit, *Algorithmic Number Theory*, *Vol. 1*, *Efficient Algorithms*, The MIT Press, Cambridge, MA, 1996.

[2] M. Blum and S. Micali, *How to generate cryptographically strong sequences of pseudo-random bits*, SIAM J. Comput. 13 (1984), 850–864.

[3] K. Fong, D. Hankerson, J. Lopez, and A. Menezes, *Field inversion and point halving revisited*, IEEE Trans. Comput. 53 (2004), 1047–1059.

[4] H. Gopalkrishna Gadiyar, K. M. Sangeeta Maini and R. Padma, *Cryptography*, *connections*, *cocycles and crystals*: *A p-adic exploration of the discrete logarithm problem*, in: Progress in Cryptology—Indocrypt 2004, Lecture Notes in Comput. Sci. 3348, Springer, 2004, 305–314.

[5] E. Knudsen, *Elliptic scalar multiplication using point halving*, in: Advances in Cryptology—ASIACRYPT '99, Lecture Notes in Comput. Sci. 1716, Springer, 1999, 135–149.

[6] J. C. Lagarias, *The* $3x + 1$ *problem*: *An annotated bibliography*, http://www.arxiv.org/math.NT/0309224.

[7] K. S. McCurley, *The discrete logarithm problem*, in: Cryptology and Computational Number Theory, Proc. Sympos. Appl. Math. 42, Amer. Math. Soc., 1990, 49–74.

[8] A. J. Menezes, P. C. van Oorschot and S. A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, New York, 1996.

[9] V. I. Nechaev, *On the complexity of a deterministic algorithm for a discrete logarithm*, Mat. Zametki 55 (1994), 91–101; English transl.: Math. Notes 55 (1994), 165–172.

[10] A. M. Odlyzko, *Discrete logarithms*: *The past and the future*, Designs Codes Cryptogr. 19 (2000), 129–145; reprinted in: Towards a Quarter-Century of Public Key Cryptography, N. Koblitz (ed.), Kluwer, 2000, 59–75.

[11] R. Peralta, *Simultaneous security of bits in the discrete log*, Advances in Cryptology—EUROCRYPT '85, Lecture Notes in Comput. Sci. 219, Springer, 1986, 62–72.

[12] J. M. Pollard, *Monte Carlo methods for index computations* (mod p), Math. Comp. 32 (1978), 918–924.

[13] H. Riesel, *Some soluble classes of the discrete logarithm problem*, BIT 28 (1988), 839–851.

[14] O. Schirokauer, D. Weber, and T. Denny, *Discrete logarithms*: *The effectiveness of the index calculus method*, in: Algorithmic Number Theory (Talence, 1996), H. Cohen (ed.), Lecture Notes in Math. 1122, Springer, 1996, 337–362.

[15] R. Schroeppel, *Elliptic curve point halving wins big*, in: 2nd Midwest Arithmetical Geometry in Cryptography Workshop (Urbana, IL, 2000).

[16] V. Shoup, *Lower bounds for discrete logarithms and related problems*, in: Advances in Cryptology—EUROCRYPT '97, Lecture Notes in Comput. Sci. 1233, Springer, 1997, 256–266.

[17] E. Teske, *Square-root algorithms for the discrete logarithm problem* (*a survey*), in: Public-Key Cryptography and Computational Number Theory, de Gruyter, Berlin, 2001, 283–301.

[18] —, *Computing discrete logarithms with the parallelized kangaroo method*, Discrete Appl. Math. 130 (2003), 61–82.

AU-KBC Research Centre
M. I. T. Campus of Anna University
Chromepet, Chennai 600 044, India
E-mail: gadiyar@au-kbc.org
onlykmsm@gmail.com
padma@au-kbc.org

Institut für theoretische Informatik
und Mathematik
Universität der Bundeswehr München
85577 Neubiberg, Germany
E-mail: mario.romsy@unibw.de

Received 4 July 2007 (4944)