Pollard's rho algorithm for logarithms

From Wikipedia, the free encyclopedia

Pollard's rho algorithm for logarithms is an algorithm introduced by John Pollard in 1978 for solving the discrete logarithm problem analogous to Pollard's rho algorithm for solving the Integer factorization problem.

The goal is to compute γ such that $\alpha^\gamma = \beta$, where β belongs to a cyclic group G generated by α. The algorithm computes integers a, b, A, and B such that $\alpha^a\beta^b = \alpha^A\beta^B$. Assuming, for simplicity, that the underlying group is cyclic of order n, we can calculate γ as a solution of the equation $(B-b)\gamma = (a-A) \pmod{n}$.

To find the needed a, b, A, and B the algorithm uses Floyd's cycle-finding algorithm to find a cycle in the sequence $x_i = \alpha^{a_i}\beta^{b_i}$, where the function $f: x_i \mapsto x_{i+1}$ is assumed to be random-looking and thus is likely to enter into a loop after approximately $\sqrt{\frac{\pi n}{2}}$ steps. One way to define such a function is to use the following rules: Divide G into three disjoint subsets of approximately equal size: S_0, S_1, and S_2. If x_i is in S_0 then double both a and b; if $x_i \in S_1$ then increment a, if $x_i \in S_2$ then increment b.

Contents [hide]

Algorithm [edit]

Let G be a cyclic group of order p, and given $\alpha, \beta \in G$, and a partition $G = S_0 \cup S_1 \cup S_2$, let $f: G \to G$ be a map

$$f(x) = \begin{cases} \beta x & x \in S_0 \\ x^2 & x \in S_1 \\ \alpha x & x \in S_2 \end{cases}$$

and define maps $g: G \times \mathbb{Z} \to \mathbb{Z}$ and $h: G \times \mathbb{Z} \to \mathbb{Z}$ by

$$g(x,n) = \begin{cases} n & x \in S_0 \\ 2n \pmod{p} & x \in S_1 \\ n+1 \pmod{p} & x \in S_2 \end{cases}$$

$$h(x,n) = \begin{cases} n+1 \pmod{p} & x \in S_0 \\ 2n \pmod{p} & x \in S_1 \\ n & x \in S_2 \end{cases}$$

Inputs *a* a generator of *G*, *b* an element of *G*

Output An integer *x* such that $a^x = b$, or failure

1. Initialise $a_0 \leftarrow 0$
   $b_0 \leftarrow 0$
   $x_0 \leftarrow 1 \in G$
   $i \leftarrow 1$
2. $x_i \leftarrow f(x_{i-1})$, $a_i \leftarrow g(x_{i-1},a_{i-1})$, $b_i \leftarrow h(x_{i-1},b_{i-1})$
3. $x_{2i} \leftarrow f(f(x_{2i-2}))$, $a_{2i} \leftarrow g(f(x_{2i-2}),g(x_{2i-2},a_{2i-2}))$, $b_{2i} \leftarrow h(f(x_{2i-2}),h(x_{2i-2},b_{2i-2}))$
4. If $x_i = x_{2i}$ then
   1. $r \leftarrow b_i - b_{2i}$
   2. If r = 0 return failure
   3. x $\leftarrow r^{-1}(a_{2i} - a_i) \bmod p$
   4. return x
5. If $x_i \neq x_{2i}$ then $i \leftarrow i+1$, and go to step 2.

Example [edit]

Consider, for example, the group generated by 2 modulo $N = 1019$ (the order of the group is $n = 1018$, 2 generates the group of units modulo 1019). The algorithm is implemented by the following C++ program:

```
#include <stdio.h>

const int n = 1018, N = n + 1;  /* N = 1019 -- prime     */
const int alpha = 2;            /* generator             */
const int beta = 5;             /* 2^{10} = 1024 = 5 (N) */

void new_xab( int& x, int& a, int& b ) {
  switch( x%3 ) {
  case 0: x = x*x     % N;  a =  a*2  % n;  b =  b*2  % n;  break;
  case 1: x = x*alpha % N;  a = (a+1) % n;                  break;
  case 2: x = x*beta  % N;                  b = (b+1) % n;  break;
  }
}

int main(void) {
  int x=1, a=0, b=0;
  int X=x, A=a, B=b;
  for(int i = 1; i < n; ++i ) {
    new_xab( x, a, b );
    new_xab( X, A, B ); new_xab( X, A, B );
    printf( "%3d  %4d %3d %3d  %4d %3d %3d\n", i, x, a, b, X, A, B );
    if( x == X ) break;
  }
  return 0;
}
```

The results are as follows (edited):

```
 i     x   a   b     X   A   B
------------------------------
 1     2   1   0    10   1   1
 2    10   1   1   100   2   2
 3    20   2   1  1000   3   3
 4   100   2   2   425   8   6
 5   200   3   2   436  16  14
 6  1000   3   3   284  17  15
 7   981   4   3   986  17  17
 8   425   8   6   194  17  19
..............................
48   224 680 376    86 299 412
49   101 680 377   860 300 413
50   505 680 378   101 300 415
51  1010 681 378  1010 301 416
```

That is $2^{681}5^{378} = 1010 = 2^{301}5^{416} \pmod{1019}$ and so $(416 - 378)\gamma = 681 - 301 \pmod{1018}$, for which $\gamma_1 = 10$ is a solution as expected. As $n = 1018$ is not prime, there is another solution $\gamma_2 = 519$, for which $2^{519} = 1014 = -5 \pmod{1019}$ holds.

Complexity [edit]

The running time is approximately $\mathcal{O}(\sqrt{n})$. If used together with the Pohlig-Hellman algorithm, the running time of the combined algorithm is $\mathcal{O}(\sqrt{p})$, where p is the largest prime factor of n.

References [edit]

- Pollard, J. M. (1978). "Monte Carlo methods for index computation (mod *p*)". *Mathematics of Computation* **32** (143): 918–924. doi:10.2307/2006496.
- Menezes, Alfred J.; van Oorschot, Paul C.; Vanstone, Scott A. (2001). "Chapter 3" (PDF). *Handbook of Applied Cryptography*.

v · t · e	Number-theoretic algorithms [hide]
Primality tests	AKS test · APR test · Baillie–PSW · ECPP test · Elliptic curve · Pocklington · Fermat · Lucas · *Lucas–Lehmer* · *Lucas–Lehmer–Riesel* · *Proth's theorem* · *Pépin's* · Quadratic Frobenius test · Solovay–Strassen · Miller–Rabin
Prime-generating	Sieve of Atkin · Sieve of Eratosthenes · Sieve of Sundaram · Wheel factorization
Integer factorization	Continued fraction (CFRAC) · Dixon's · Lenstra elliptic curve (ECM) · Euler's · Pollard's rho · $p - 1$ · $p + 1$ · Quadratic sieve (QS) · General number field sieve (GNFS) · *Special number field sieve (SNFS)* · Rational sieve · Fermat's · Shanks' square forms · Trial division · Shor's
Multiplication	Ancient Egyptian · Long · Karatsuba · Toom–Cook · Schönhage–Strassen · Fürer's
Discrete logarithm	Baby-step giant-step · **Pollard rho** · Pollard kangaroo · Pohlig–Hellman · Index calculus · Function field sieve
Greatest common divisor	Binary · Euclidean · Extended Euclidean · Lehmer's
Modular square root	Cipolla · Pocklington's · Tonelli–Shanks
Other algorithms	Chakravala · Cornacchia · Integer relation · Integer square root · Modular exponentiation · Schoof's
Italics indicate that algorithm is for numbers of special forms · Smallcaps indicate a deterministic algorithm	

Categories: Logarithms | Number theoretic algorithms