Pollard's rho algorithm

From Wikipedia, the free encyclopedia

This article is about the integer factorization algorithm. For the discrete logarithm algorithm, see Pollard's rho algorithm for logarithms.

Pollard's rho algorithm is a special-purpose integer factorization algorithm. It was invented by John Pollard in 1975.[1] It is particularly effective for a composite number having a small prime factor.

Contents [hide]

1 Core ideas
2 Algorithm
3 Variants
4 Application
5 Example factorization
6 The Example n = 10403 = 101 . 103
  6.1 C++ Pseudo code
  6.2 The Results
7 Complexity
8 References
9 Additional reading
10 External links

Core ideas [edit]

The ρ algorithm is based on Floyd's cycle-finding algorithm and on the observation that (as in the birthday problem) t random numbers $x_1, x_2, \ldots, x_t$ in the range [1, n] will contain a repetition with probability P > 0.5 if t > $1.177n^{1/2}$. The constant 1.177 comes from the more general result that if P is the probability that t random numbers in the range [1, n] contain a repetition, then $P > 1 - \exp\{ -t^2/2n \}$. Thus P > 0.5 provided $1/2 < \exp\{ -t^2/2n \}$, or $t^2 > 2n\ln 2$, or $t^2 > 2n \ln 2$, or $t > (2\ln 2)^{1/2} n^{1/2} = 1.177n^{1/2}$.

The ρ algorithm uses g(x), a polynomial modulo *n*, as a generator of a pseudo-random sequence. (The most commonly used function is $g(x) = x^2 \bmod n$.) Let's assume n = pq. The algorithm generates the sequence $x_1 = g(2)$, $x_2 = g(g(2))$, $x_3 = g(g(g(2)))$, and so on. Two different sequences will in effect be running at the same time —the sequence $\{x_k\}$ and the sequence $\{x_k \bmod p\}$. Since $p < n^{1/2}$, the latter sequence is likely to repeat earlier than the former sequence. The repetition of the mod p sequence will be detected by the fact that $\gcd(x_k \bmod p - x_m \bmod p, n) = p$, where k < m. Once a repetition occurs, the sequence will cycle, because each term depends only on the previous one. The name **ρ algorithm** derives from the similarity in appearance between the Greek letter ρ and the directed graph formed by the values in the sequence and their successors. Once it is cycling, Floyd's cycle-finding algorithm will eventually detect a repetition. The algorithm succeeds whenever the sequence $\{x_k \bmod p\}$ repeats before the sequence $\{x_k\}$. The randomizing function g(x) must be a polynomial modulo n, so that it will work both modulo p and modulo n. That is, so that $g(x \bmod p) \equiv g(x) \pmod p$.

Algorithm [edit]

The algorithm takes as its inputs *n*, the integer to be factored; and *g(x)*, a polynomial p(x) computed modulo n. This will ensure that if p|n, and $x \equiv y \bmod p$, then $g(x) \equiv g(y) \bmod p$. In the original algorithm, $g(x) = x^2 - 1 \bmod n$, but nowadays it is more common to use $g(x) = x^2 + 1 \bmod n$. The output is either a non-trivial factor of n, or failure. It performs the following steps:[2]

1. x ← 2; y ← 2; d ← 1;
2. While d = 1:
    1. x ← g(x)
    2. y ← g(g(y))
    3. d ← gcd(|x - y|, n)
3. If d = n, return failure.
4. Else, return d.

Note that this algorithm may fail to find a nontrivial factor even when n is composite. In that case, you can try again, using a starting value other than 2 or a different g(x). The name **ρ algorithm** comes from the fact that the values of x (mod d) eventually repeat with period d, resulting in a ρ shape when you graph the values.

Variants [edit]

In 1980, Richard Brent published a faster variant of the rho algorithm. He used the same core ideas as Pollard but a different method of cycle detection, replacing Floyd's cycle-finding algorithm with the related Brent's cycle finding method.[3]

A further improvement was made by Pollard and Brent. They observed that if $\gcd(a,n) > 1$, then also $\gcd(ab,n) > 1$ for any positive integer *b*. In particular, instead of computing $\gcd(|x-y|,n)$ at every step, it suffices to define *z* as the product of 100 consecutive $|x-y|$ terms modulo *n*, and then compute a single $\gcd(z,n)$. A major speed up results as 100 *gcd* steps are replaced with 99 multiplications modulo *n* and a single *gcd*. Occasionally it may cause the algorithm to fail by introducing a repeated factor, for instance when *n* is a square. But it then suffices to go back to the previous gcd term, where $\gcd(z,n) = 1$, and use the regular ρ algorithm from there.

Application [edit]

The algorithm is very fast for numbers with small factors, but slower in cases where all factors are large. The ρ algorithm's most remarkable success was the factorization of the eighth Fermat number, F_8 = 1238926361552897 * 93461639715357977769163558199606896584051237541638188580280321. The ρ algorithm was a good choice for F_8 because the prime factor p = 1238926361552897 is much smaller than the other factor. The factorization took 2 hours on a UNIVAC 1100/42.

Example factorization [edit]

Let n = 8051 and $g(x) = (x^2 + 1) \bmod 8051$.

i	x_i	y_i	GCD($\|x_i - y_i\|$, 8051)
1	5	26	1
2	26	7474	1
3	677	871	97

97 is a non-trivial factor of 8051. Starting values other than x = y = 2 may give the cofactor (83) instead of 97.

The Example n = 10403 = 101 . 103 [edit]

Here we introduce another variant, where only a single sequence is computed, and the gcd is computed inside the loop that detects the cycle.

C++ Pseudo code [edit]

The following pseudo code finds the factor 101 of 10403 with a starting value of x = 2.

```
int gcd( int a, int b) {
 int remainder;
 while (b != 0) {
  remainder = a % b;
  a = b;
  b = remainder;
 }
 return a;
}

int main () {

 int number = 10403,x_fixed = 2,cycle_size = 2,x = 2,factor = 1;

 while (factor == 1) {

  for (int count=1;count <= cycle_size && factor == 1;count++) {
   x = (x*x+1)%number;
   factor = gcd(x - x_fixed, number);
```

```
    }

    cycle_size *= 2;
    x_fixed = x;
  }
  cout << "\nThe factor is  " << factor;
}
```

The Results [edit]

In the following table the third and fourth columns contain secret information not known to the person trying to factor pq = 10403. They are included to show how the algorithm works. If we start with x = 2 and follow the algorithm, we get the following numbers:

x	x_{fixed}	x mod 101	x_{fixed} mod 101	step
2	2	2	2	0
5	2	5	2	1
26	2	26	2	2
677	26	71	26	3
598	26	93	26	4
3903	26	65	26	5
3418	26	85	26	6
156	3418	55	85	7
3531	3418	**97<--**	85	8
5168	3418	17	85	9
3724	3418	88	85	10
978	3418	69	85	11
9812	3418	15	85	12
5983	3418	24	85	13
9970	3418	72	85	14
236	9970	34	72	15
3682	9970	46	72	16
2016	9970	**97<--**	72	17
7087	9970	17	72	18
10289	9970	88	72	19
2594	9970	69	72	20
8499	9970	15	72	21
4973	9970	24	72	22
2799	9970	**72<--**	**72**	23

The first repetition modulo 101 is 97 which occurs in step 17. The repetition is not detected until step 23, when x = x_{fixed} (mod 101). This causes gcd(x - x_fixed, n) = gcd(2799 - 9970, n) to be p = 101, and a factor is found.

Complexity [edit]

If the pseudo random number x = g(x) occurring in the Pollard ρ algorithm were an actual random number, it would follow that success would be achieved half the time, by the Birthday paradox in $O(\sqrt{p}) \leq O(n^{1/4})$ iterations. It is believed that the same analysis applies as well to the actual rho algorithm, but this is a heuristic claim, and rigorous analysis of the algorithm remains open.[4]

References [edit]

1. ^ Pollard, J. M. (1975), "A Monte Carlo method for factorization", *BIT Numerical Mathematics* **15** (3): 331–334, doi:10.1007/bf01933667
2. ^ Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L. & Stein, Clifford (2001), "Section 31.9: Integer factorization", *Introduction to Algorithms* (Second ed.), Cambridge, MA: MIT Press, pp. 896–901, ISBN 0-262-

03293-7 (this section discusses only Pollard's rho algorithm).
3. ^ Brent, Richard P. (1980), "An Improved Monte Carlo Factorization Algorithm", *BIT* **20**: 176–184, doi:10.1007/BF01933190
4. ^ Galbraith, Steven D. (2012), "14.2.5 Towards a rigorous analysis of Pollard rho", *Mathematics of Public Key Cryptography*, Cambridge University Press, pp. 272–273, ISBN 9781107013926.

Additional reading [edit]

- Katz, Jonathan; Lindell, Yehuda (2007), "Chapter 8", *Introduction to Modern Cryptography*, CRC Press

External links [edit]

- Weisstein, Eric W., "Pollard rho Factorization Method", *MathWorld*.
- Java Implementation

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: Integer factorization algorithms