Pollard's rho algorithm

Pollard's rho algorithm is an algorithm for integer factorization. It was invented by John Pollard in 1975.[1] It uses only a small amount of space, and its expected running time is proportional to the square root of the size of the smallest prime factor of the composite number being factorized.

Contents

Core ideas

Suppose we need to factorize a number $\boldsymbol{n} = \boldsymbol{pq}$, where $\boldsymbol{p}$ is a non-trivial factor. A polynomial modulo $\boldsymbol{n}$, called $\boldsymbol{g(x)}$ (e.g., $\boldsymbol{g(x) = (x^2 + 1) \bmod n}$), is used to generate a pseudorandom sequence: A starting value, say 2, is chosen, and the sequence continues as $\boldsymbol{x_1 = g(2)}$, $\boldsymbol{x_2 = g(g(2))}$, $\boldsymbol{x_3 = g(g(g(2)))}$, etc. The sequence is related to another sequence $\boldsymbol{\{x_k \bmod p\}}$. Since $\boldsymbol{p}$ is not known beforehand, this sequence cannot be explicitly computed in the algorithm. Yet, in it lies the core idea of the algorithm.

Because the number of possible values for these sequences are finite, both the $\boldsymbol{\{x_k\}}$ sequence, which is mod $\boldsymbol{n}$, and $\boldsymbol{\{x_k \bmod p\}}$ sequence will eventually repeat, even though we do not know the latter. Assume that the sequences behave like random numbers. Due to the birthday paradox, the number of $\boldsymbol{x_k}$ before a repetition occurs is expected to be $\boldsymbol{O(\sqrt{N})}$, where $\boldsymbol{N}$ is the number of possible values. So the sequence $\boldsymbol{\{x_k \bmod p\}}$ will likely repeat much earlier than the sequence $\boldsymbol{\{x_k\}}$. Once a sequence has a repeated value, the sequence will cycle, because each value depends only on the one before it. This structure of eventual cycling gives rise to the name "Rho algorithm", owing to similarity to the shape of the Greek character ρ when the values $\boldsymbol{x_1 \bmod p}$, $\boldsymbol{x_2 \bmod p}$, etc. are represented as nodes in a directed graph.

This is detected by Floyd's cycle-finding algorithm: two nodes $\boldsymbol{i}$ and $\boldsymbol{j}$ (i.e., $\boldsymbol{x_i}$ and $\boldsymbol{x_j}$) are kept. In each step, one moves to the next node in the sequence and the other moves to the one after the next node. After that, it is checked whether $\boldsymbol{\gcd(x_i - x_j, n) \neq 1}$. If it is not 1, then this implies that there is a repetition in the $\boldsymbol{\{x_k \bmod p\}}$ sequence (i.e. $\boldsymbol{x_i \bmod p = x_j \bmod p}$). This works because if the $\boldsymbol{x_i \bmod p}$ is the same as $\boldsymbol{x_j \bmod p}$, the difference between $\boldsymbol{x_i}$ and $\boldsymbol{x_j}$ is necessarily a multiple of $\boldsymbol{p}$. Although this always happens eventually, the resulting GCD is a divisor of $\boldsymbol{n}$ other than 1. This may be $\boldsymbol{n}$ itself, since the two sequences might repeat at the same time. In this (uncommon) case the algorithm fails, and can be repeated with a different parameter.

Algorithm

The algorithm takes as its inputs n, the integer to be factored; and $\boldsymbol{g(x)}$, a polynomial in x computed modulo n. In the original algorithm, $\boldsymbol{g(x) = (x^2 - 1) \bmod n}$, but nowadays it is more common to use $\boldsymbol{g(x) = (x^2 + 1) \bmod n}$. The output is either a non-trivial factor of n, or failure. It performs the following steps:[2]

```
x ← 2; y ← 2; d ← 1
while d = 1:
    x ← g(x)
    y ← g(g(y))
    d ← gcd(|x − y|, n)
if d = n:
    return failure
else:
    return d
```

Here x and y corresponds to $\boldsymbol{x_i}$ and $\boldsymbol{x_j}$ in the section about core idea. Note that this algorithm may fail to find a nontrivial factor even when n is composite. In that case, the method can be tried again, using a starting value other than 2 or a different $\boldsymbol{g(x)}$.

Example factorization

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

i	x	y	$\boldsymbol{\gcd(\lvert x-y\rvert, 8051)}$
1	5	26	1
2	26	7474	1
3	677	871	97
4	7474	1481	1

97 is a non-trivial factor of 8051. Starting values other than $x = y = 2$ may give the cofactor (83) instead of 97. One extra iteration is shown above to make it clear that y moves twice as fast as x. Note that even after a repetition, the GCD can return to 1.

Variants

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

Application

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 ninth Fermat number, $F_9 = 1238926361552897 \times 93461639715357977769163558199606896584051237541638188580280321$. The ρ algorithm was a good choice for F_9 because the prime factor $p = 1238926361552897$ is much smaller than the other factor. The factorization took 2 hours on a UNIVAC 1100/42.

The example $n = 10403 = 101 \cdot 103$

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 code sample

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

```
#include <stdio.h>
#include <stdlib.h>

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

int main (int argc, char *argv[])
{
  int n = 10403, loop = 1, count;
  int x_fixed = 2, x = 2, size = 2, factor;

  do {
    printf("----  loop %4i  ----\n", loop);
    count = size;
    do {
      x = (x * x + 1) % n;
      factor = gcd(abs(x - x_fixed), n);
      printf("count = %4i  x = %6i  factor = %i\n", size - count + 1, x, factor);
    } while (--count && (factor == 1));
    size *= 2;
    x_fixed = x;
    loop++;
  } while (factor == 1);
  printf("factor is %i\n", factor);
  return factor == n ? EXIT_FAILURE : EXIT_SUCCESS;
}
```

The above code will show the algorithm progress as well as intermediate values. The final output line will be "factor is 101".

Python code sample

```
def gcd(a, b):
    while a % b != 0:
        a, b = b, a % b
    return b

number = 10403
x_fixed = 2
cycle_size = 2
x = 2
factor = 1

while factor == 1:
    count = 0
    while count <= cycle_size and factor <= 1:
        x = (x*x + 1) % number
        factor = gcd(x - x_fixed, number)
        count += 1
    cycle_size *= 2
    x_fixed = x

print(factor)
```

The results

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

$\boldsymbol{x}$	$\boldsymbol{x_{fixed}}$	$\boldsymbol{x \bmod 101}$	$\boldsymbol{x_{fixed} \bmod 101}$	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 $\boldsymbol{x \equiv x_{fixed} \pmod{101}}$. This causes $\boldsymbol{\gcd(x - x_{fixed}, n) = \gcd(2799 - 9970, n)}$ to be $\boldsymbol{p = 101}$, and a factor is found.

Complexity

If the pseudorandom number $\boldsymbol{x = g(x)}$ 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 $\boldsymbol{O(\sqrt{p})} \leq \boldsymbol{O(n^{1/4})}$ 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

1. Pollard, J. M. (1975), "A Monte Carlo method for factorization", *BIT Numerical Mathematics*, **15** (3): 331–334, doi:10.1007/bf01933667 (https://doi.org/10.1007%2Fbf01933667)
2. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L. & Stein, Clifford (2009), "Section 31.9: Integer factorization", *Introduction to Algorithms* (third ed.), Cambridge, MA: MIT Press, pp. 975–980, ISBN 978-0-262-03384-8 (this section discusses only Pollard's rho algorithm).
3. Brent, Richard P. (1980). "An Improved Monte Carlo Factorization Algorithm" (http://maths-people.anu.edu.au/~brent/pub/pub051.html), *BIT*, **20**: 176–184, doi:10.1007/BF01933190 (https://doi.org/10.1007%2FBF01933190)
4. Galbraith, Steven D. (2012), "14.2.5 Towards a rigorous analysis of Pollard rho", *Mathematics of Public Key Cryptography* (https://books.google.com/books?id=owd768EIvosC&pg=PA272), Cambridge University Press, pp. 272–273, ISBN 9781107013926.

Further reading

- Katz, Jonathan; Lindell, Yehuda (2007), "Chapter 8", *Introduction to Modern Cryptography*, CRC Press
- Samuel S. Wagstaff, Jr. (2013). *The Joy of Factoring* (http://www.ams.org/bookpages/stml-68). Providence, RI: American Mathematical Society. pp. 135–138. ISBN 978-1-4704-1048-3.

External links

- Comprehensive article on Pollard's Rho algorithm aimed at an introductory-level audience (https://sofdstand.wixsite.com/persuasnp/single-post/2018/08/27/Factorising-part-2—Pollards-Rho-algorithm)
- Weisstein, Eric W. "Pollard rho Factorization Method" (http://mathworld.wolfram.com/PollardRhoFactorizationMethod.html). *MathWorld*.
- Java Implementation (http://introcs.cs.princeton.edu/java/99crypto/PollardRho.java.html)