Primzahlfaktorisierungsalgorithmen - Pollard rho, Pollard (p-1)

Philip Suskin

July 6, 2021

1 Einleitung

Produkte aus Primzahlen sind leicht zu bilden, jedoch ist es danach sehr schwer, allein anhand dieses Produkts auf die ursprünglichen Primfaktoren zurückzuschließen. Primzahlfaktorisierungsalgorithmen befassen sich mit dieser Problematik, und bilden mit besonderem Blick auf zwei spezifische Algorithmen, Pollard rho und Pollard (p-1), die Grundlage dieses Themas.

2 Probedivision

Bevor man einen fortgeschrittenen Algorithmus wie Pollard rho anwendet, ist es hilfreich, möglichst viele "kleine" Primfaktoren unterhalb einer selbst definierten Schranke zu finden, denn für die Erkennung kleiner Primfaktoren gibt es mittels Probedivision effizientere und einfachere Methoden als bei einem vollständigen Algorithmus. Man gehe wie folgt vor:

2.1 Der Algorithmus

1. Man lege eine Schranke S fest, unter der man alle Primfaktoren von N (das zu zerlegende Produkt) zu finden hat (z.B. S = 1699 in MAPLE).
2. Man teste, ob N überhaupt Primfaktoren unter S enthält, mittels g = ggT(P, N), wobei P das Produkt aller Primzahlen unter S ist (den Wert von P kann man fest im Programmcode speichern).

Wenn g > 1:
a. Probedivision über alle Primzahlen unter S durchführen. Dies wird generell so gemacht, dass man die Zahlenmenge $d_1 = 1$, $d_2 = 7$, $d_3 = 11$, $d_4 = 13$, $d_5 = 17$, $d_6 = 19$, $d_7 = 23$, $d_8 = 29$ festlegt, und nach der Teilbarkeitsprobe mit 2, 3 und 5 den Ausdruck $30k + d_i$ für $i = 1, 2, ..., 8$ mit aufsteigendem k (anfangs 0) bis zur Schranke S, N auf Teilbarkeit prüft. Obwohl man mit dem Ausdruck mehr als nur Primzahlen trifft, z.B. $49 = 30 \cdot 1 + d_6$, ist diese Herangehensweise aufgrund des geringeren Speicheraufwands dem Verfahren vorzuziehen, eine immense Liste an Primzahlen im Programm zu laden und durch diese Liste durchzuiterieren.
b. N = N / g neu setzen und zurück nach 2. gehen

Sonst:
3. Prüfe, ob N eine Primzahl ist. Hierfür wird ein seperater Primzahltest angewandt. Wenn N eine Primzahl ist, hat man den letzten Faktor von N gefunden. Sonst geht's weiter.
4. Prüfe, ob N eine Potenz ist. Hierfür wird getestet, ob $\sqrt[k]{N}$ für $k = 2, 3, ..., \lfloor \frac{\log N}{\log S} \rfloor$ eine ganze Zahl ist (nur bis $\lfloor \frac{\log N}{\log S} \rfloor$, da ein größeres k eine Zahl unterhalb der Schranke S aus der Wurzel liefern würde). Wenn N eine Potenz ist, hat man die letzten Faktoren von N gefunden. Wenn nicht, wird es Zeit, einen anspruchsvolleren Algorithmus anzuwenden.

2.2 Laufzeit

Da man durch den Ausdruck $30k + d_{1\text{-}8}$ alle 30 Zahlen 8 Divisionen ($O(1)$) ausführt, beträgt die insgesamte Laufzeit $O(\frac{8}{30} \cdot S)$.

3 Pollard rho

3.1 Funktionsweise

Man stelle sich zu Beginn eine Zahlenfolge aus ganzen Zahlen vor, die sich aus der rekursiven diskreten Funktion $f(x) = x^2 + c \mod p$ mit $c \in \mathbb{Z} \setminus \{0, -2\}$ ergibt. Die Folge wird sich pseudozufällig im Bereich von 0 bis $p - 1$ verhalten, **bis sich eine beliebige Zahl zum ersten Mal wiederholt (dies wird als Kollision bezeichnet)**. Ab der ersten Kollision verhält sich die Zahlenfolge nur noch zyklisch mit allen Zahlen zwischen dem ersten und dem wiederholten Vorkommen der Zahl. Dies lässt sich leicht anhand der Definition von der Funktion $f(x)$ erklären. Hier ein Beispiel:

> $p = 59, c = 1, x_0 = 2$: 2, 5, 26, 28, 30, 16, 21, 29, 16, 21, 29, 16, 21, 29, 16, 21, 29
> Hier ist das zyklische Verhalten ab der ersten Kollision der Zahl 16 zu sehen.

Somit weiß man, dass es (unendlich viele) Kollisionen in der Zahlenfolge gibt, sprich unendlich viele y_i, y_j mit $y_i \equiv y_j \mod p, i \neq j$. Die minimale Differenz $l(p)$ zwischen i und j ist außerdem für jede Zahlenfolge fest bestimmt und wird als Periode bezeichnet, die Zahlen vor der ersten Kollision bilden somit den ergänzenden aperiodischen Teil $k(p)$ der Folge. Zusammen bilden diese beiden Komponenten den Hintergrund des "rho" im Namen Pollard rho, da der griechische Buchstabe rho (rechts abgebildet) von unten beginnend das Sich-in-einen-Zyklus-begeben der auf $f(x)$ basierenden Zahlenfolgen darstellt. "Pollard" ist dabei der Name des Entwicklers des Algorithmus (John M. Pollard, 1941 in London geboren).

ρ

Mit dem Wissen, dass sich eine Zahlenfolge $x_0, x_1, ...$ aus einer rekursiven Funktion $f(x)$ modulo dem Faktor p der zusammengesetzten Zahl $N = pq$ ab der ersten Kollision zyklisch verhält, kann man mit zwei Iteratoren i und j durch dieselbe Zahlenfolge **modulo N** durchiterieren (der Faktor p steht einem nämlich natürlich nicht zur Verfügung, er wird gesucht). Inkrementiert man i nach jeder Iteration um eine Stelle und j um zwei Stellen der Zahlenfolge, so wird garantiert nach einer endlichen Menge an Iterationen der Ausdruck $ggT(x_i - x_j, N) > 1$ erfüllt. Genauer wird dies eintreten, wenn $x_i \equiv x_j \mod p$, denn somit ist $x_i - x_j$ logischerweise ein Vielfaches von p und $ggT(x_i - x_j, N) = p$, wenn nicht auch $x_i \equiv x_j \mod N$ gilt, in welchem Fall $ggT(x_i - x_j, N) = N$ und der Algorithmus mit neuer Konstante c oder neuem Initalwert x_0 erneut angewandt werden muss. Man kann sich aufgrund des bereits erklärten zyklischen Verhaltens von der Zahlenfolge mod p sicher sein, dass die Iteratoren i und j nach einer endlichen Menge an Iterationen zwei kongruente Zahlen modulo p finden werden, da der Iterator j eine Stelle mehr als i pro Iteration voranschreitet und ihn daher garantiert nach maximal $l(p)$ Iterationen "einholen" würde, sobald sich beide Iteratoren im zyklischen Bereich befinden (nach $k(p)$ Iterationen). Insgesamt tritt also $ggT(x_i - x_j, N) > 1$ nach $r(p) < k(p) + l(p)$ Iterationen ein.

3.2 Pseudocode

Der Grundalgorithmus Pollard rho ist recht kompakt und in wenigen Zeilen Pseudocode darstellbar:

Algorithm 1 Pollard rho Algorithmus

```
f(x) = x^2 + c mod p mit c ∈ ℤ \ {0, −2}, wähle x_0 ∈ ℤ
x_i = x_0
x_j = x_0
d = 1
while d = 1 do
    x_i = f(x_i) mod N
    x_j = f(f(x_j)) mod N
    d = ggT(x_i − x_j, N)
end while
return d
```

3.3 Beispiel

Table 1: Wie der Algorithmus im Hintergrund funktioniert

	mod p			mod q			mod N			
i	y_i	y_{2i}	$y_i - y_{2i}$	z_i	z_{2i}	$z_i - z_{2i}$	x_i	x_{2i}	$x_i - x_{2i}$	$ggT(x_i - x_{2i}, N)$
1	3	8	-5	3	8	-5	3	8	-5	1
2	8	15	-7	8	26	-18	8	3968	-3960	1
3	4	25	-21	63	31	32	63	615	-552	1
4	15	34	-19	26	47	-21	3968	2748	1220	1
5	47	34	13	18	31	-13	2938	4046	-1108	1
6	25	34	-9	31	47	-16	615	2748	-2133	1
7	34	34	0	11	31	-20	3515	4046	-531	59
8	34	34	0	47	47	0	2748	2748	0	4307
9	34	34	0	18	31	-13	1332	4046	-2714	59

In dieser Tabelle ist die Funktionsweise des Algorithmus für ein besipielhaftes $N = pq, p = 59, q = 73$ dargestellt. Man sieht, dass der Faktor $p = 59$ nach 7 Iterationen entdeckt wird. Als Referenz gilt für die Zahlenfolge mod p: $k(p) = 7$ und $l(p) = 1$ und für die Zahlenfolge mod q: $k(q) = 5$ und $l(q) = 4$. Wichtiger Hinweis: wäre eine Kollision von p und eine von q in der genau gleichen Iteration aufgetreten, hätte man das triviale Ergebnis $ggT(x_i - x_{2i}, N) = N$ erzielt (z.B. in der Tabelle bei Iteration 8 zu sehen).

3.4 Laufzeit

Unter der Annahme, dass die Zahlenfolge aus $f(x)$ sich zufällig verhält, kann mittels des "Geburtstagsproblems" angenommen werden, dass eine Kollision mod p in $O(\sqrt{p})$ Zeit zum ersten Mal auftritt. Der hauptsächliche Aufwand im Pollard Rho Algorithmus sind die ggT-Berechnungen, die eine maximale Laufzeit von $O(\log^2 N)$ haben, weshalb ein Faktor von N insgesamt in $O(\sqrt{p} \cdot \log^2 N)$ Zeit erwartet werden kann. Hierdurch wird klar, dass Pollard rho sich besonders für zusammengestzte Zahlen eignet, bei denen eine der beiden Faktoren klein ist.

4 Pollard (p-1)

4.1 Funktionsweise

Pollard (p-1) ist ein nicht so universaler Algorithmus wie Pollard rho, er wird meist nach der Probedivision als Methode zur Faktorisierung von "kryptographisch schwachen" Primzahlen p_i aus N genutzt, deren Vorgänger $p_i - 1$ ausschließlich aus Primpotenzen unter einer selbst bestimmten Schranke B besteht (man nennt so eine Primzahl B-potenzglatt). Die Ermittlung eines Faktors p_i beruht auf dem kleinen Fermatschen Satz, der besagt, dass für eine ganze Zahl a und eine Primzahl p die Kongruenz $a^{p-1} \equiv 1 \bmod p$ gilt. Da grundsätzlich $1^k \equiv 1 \bmod p$ gilt, ist $a^m \equiv 1 \bmod p$, sobald alle Primfaktoren von $p - 1$ in m enthalten sind. Um hiervon die Chancen für ein unbekanntes p zu erhöhren, bildet man das Produkt m aller Primzahlpotenzen $< B$ und berechnet schließlich die Potenz $R = a^m$, wobei a eine zu N teilerfremde Zahl ist. Da $R \equiv 1 \bmod p$ (falls m tatsächlich alle Primfaktoren von $p - 1$ enthält), muss der größte gemeinsame Teiler aus **R - 1** und N bestimmt werden: $ggT(R - 1, N) = p$.

4.2 Beispiel

Für eine zusammengesetzte Zahl $N = 299$ würde bei der Wahl der Schranke $B = 5$ der Exponent $m = 2^2 \cdot 3^1 \cdot 5^1$ ermittelt werden. Da N ungerade ist kann trivialerweise die zu N teilerfremde Basis $a = 2$ bestimmt werden. Mit $ggT(a^m - 1, N) = 13$ erhält man den Primfaktor 13. Man kann im Nachhinein verstehen, warum 13 gefunden wurde, da $13 - 1 = 12$ die Primfaktorezerlegung $2^2 \cdot 3^1$ besitzt, und alle diese Primpotenzen in m vorkommen.

4.3 Laufzeit

Es werden $O(\frac{S}{\log S})$ Primzahlen evaluiert, wobei pro Primzahl $O(\log S)$ Multiplikationen modulo N durchgeführt werden. Da die Laufzeit von Multiplikation modulo N $O(N^2)$ beträgt, beträgt die Gesamtlaufzeit $O(S \cdot N^2)$.

5 Zusatz für C++ Liebhaber

Vermutlich kennen sich viele aus diesem Seminar mit C++ aus. Ich habe mithilfe der GMP Library die hier dokumentierten Algorithmen mit zusätzlichen Angaben bezüglich der Laufzeitanalyse (Iterationen, ggT-Anwendungen, Gesamtlaufzeit in Microsekunden) implementiert. Speziell habe ich die C++ Wrapper-Klassen `mpz_class` aus gmpxx für ganzzahlige Berechnungen (der Großteil) und `mpfr::mpreal` aus mpreal für reelle Berechnungen verwendet. Außerdem habe ich bei verschiedenen Tests/Laufzeitanalysen die gnuplot-iostream Library verwendet (sehr empfehlenswert!), die das Plotten in C++ mittels gnuplot ermöglicht. Hier ist nun ein kurzer Überblick über die Abfolge von Algorithmen, die beim Aufruf von `Factorize::findFactors()` verwendet werden:

1. `_removeSmallFactors(std::vector<mpz_class>& factors, mpz_class& n, const mpz_class& b)`: Entfernt mittels Probedivision alle Primfaktoren aus `n` unterhalb der Schranke `b` und speichert sie in `factors`.

2. `pollardPOne(const mpz_class& n, const mpz_class& s)`: Liefert mittels Pollard (p-1) das Produkt aller Faktoren p_i, deren Vorgänger $p_i - 1$ `s`-potenzglatt ist.

3. `_getAllFactors(std::vector<mpz_class>& factors, mpz_class& n, const mpz_class& b, mpz_class& x0, const mpz_class& c, PollardRho pRho)`: Findet rekursiv alle übrigen Faktoren aus `n` (somit auch aus dem Ergebnis von `pollardPOne`, falls es kein Primfaktor ist) mittels eines Primalitätstests, Potenztests und schließlich der Anwendung von einer der drei Varianten des Pollard rho Algorithmus, angegeben mit `pRho`. Die drei bisher implementierten Varianten sind Floyd's Algorithmus (in dieser Doku beschrieben), Floyd's verbesserter Algorithmus und Brent's Algorithmus (die letzteren beiden werden im Vortrag erläutert).

Zum Schluss stehen alle Primfaktoren (ggf. mehrfach bei Primpotenzen) unsortiert im Vektor `factors`.

Leider bin ich noch nicht dazu gekommen, eine CMakeLists.txt zu erstellen, und der Code ist ohne die nötigen Anpassung sehr schwer zu compilieren, aber für den Anfang könnte der Code an sich für den einen oder anderen von Interesse sein: `https://github.com/classix-ps/Pollard-Rho`.