Fast Constant-Time GCD Computation and Modular Inversion

Daniel J. Bernstein[1,2] Bo-Yin Yang[3]

[1]University of Illinois at Chicago

[2]Ruhr Universtät Bochum

[3]Academia Sinica

Monday, August 26, 2019

Summary: Fast, Safe GCD and Inversions

Normally compute $1/x$ in $\mathbf{F}_p$ as x^{p-2}.

$n^{3+o(1)}$ bit ops	using schoolbook multiplication
$n^{2.58\ldots+o(1)}$ bit ops	using Karatsuba multiplication
$n^{2+o(1)}$ bit ops	using FFT-based multiplication

Summary: Fast, Safe GCD and Inversions

Normally compute $1/x$ in $\mathbf{F}_p$ as x^{p-2}.

$n^{3+o(1)}$ bit ops	using schoolbook multiplication
$n^{2.58\ldots+o(1)}$ bit ops	using Karatsuba multiplication
$n^{2+o(1)}$ bit ops	using FFT-based multiplication

Why not use extensions of Euclid's algorithm?

$n^{2+o(1)}$ bit ops	using schoolbook multiplication
$n^{1.58\ldots+o(1)}$ bit ops	using Karatsuba multiplication
$n^{1+o(1)}$ bit ops	using FFT-based multiplication

Summary: Fast, Safe GCD and Inversions

Normally compute $1/x$ in $\mathbf{F}_p$ as x^{p-2}.

$n^{3+o(1)}$ bit ops	using schoolbook multiplication
$n^{2.58\ldots+o(1)}$ bit ops	using Karatsuba multiplication
$n^{2+o(1)}$ bit ops	using FFT-based multiplication

Why not use extensions of Euclid's algorithm?

$n^{2+o(1)}$ bit ops	using schoolbook multiplication
$n^{1.58\ldots+o(1)}$ bit ops	using Karatsuba multiplication
$n^{1+o(1)}$ bit ops	using FFT-based multiplication

Usual answer: Need constant-time algorithm.

Summary: Fast, Safe GCD and Inversions

Normally compute $1/x$ in $\mathbf{F}_p$ as x^{p-2}.

$n^{3+o(1)}$ bit ops	using schoolbook multiplication
$n^{2.58\ldots+o(1)}$ bit ops	using Karatsuba multiplication
$n^{2+o(1)}$ bit ops	using FFT-based multiplication

Why not use extensions of Euclid's algorithm?

$n^{2+o(1)}$ bit ops	using schoolbook multiplication
$n^{1.58\ldots+o(1)}$ bit ops	using Karatsuba multiplication
$n^{1+o(1)}$ bit ops	using FFT-based multiplication

Usual answer: Need constant-time algorithm.

Our algorithm is constant-time; $n^{1+o(1)}$ bit ops; simpler than previous variable-time algorithms. No division subroutine between recursive calls.

Examples of Needing Inversions

NTRU Key generation (where n is prime)

- Find inverse in $\mathbf{F}_3[X]/(X^n - 1)$
- Find inverse in $(\mathbf{Z}/2^k\mathbf{Z})[X]/(X^n - 1)$, which depends on inverse in $\mathbf{F}_2[X]/(X^n - 1)$.

Examples of Needing Inversions

NTRU Key generation (where n is prime)

- Find inverse in $\mathbf{F}_3[X]/(X^n - 1)$
- Find inverse in $(\mathbf{Z}/2^k\mathbf{Z})[X]/(X^n - 1)$, which depends on inverse in $\mathbf{F}_2[X]/(X^n - 1)$.

NTRU Prime Key generation (where n is prime)

- Find inverse in $\mathbf{F}_{4591}[X]/(X^n - X - 1)$ (= a field).
- Find inverse in $\mathbf{F}_3[X]/(X^n - X - 1)$

Examples of Needing Inversions

NTRU Key generation (where n is prime)

- Find inverse in $\mathbf{F}_3[X]/(X^n - 1)$
- Find inverse in $(\mathbf{Z}/2^k\mathbf{Z})[X]/(X^n - 1)$, which depends on inverse in $\mathbf{F}_2[X]/(X^n - 1)$.

NTRU Prime Key generation (where n is prime)

- Find inverse in $\mathbf{F}_{4591}[X]/(X^n - X - 1)$ (= a field).
- Find inverse in $\mathbf{F}_3[X]/(X^n - X - 1)$

Integer Modular Inversions in CSIDH

Needs inverse modulo $p = 4p_1p_2p_3 \cdots p_{73}p_{74} - 1$, where $p_1 \cdots p_{73}$ are the smallest 73 odd primes and $p_{74} = 587$.

Examples of Needing Inversions

NTRU Key generation (where n is prime)

- Find inverse in $\mathbf{F}_3[X]/(X^n - 1)$
- Find inverse in $(\mathbf{Z}/2^k\mathbf{Z})[X]/(X^n - 1)$, which depends on inverse in $\mathbf{F}_2[X]/(X^n - 1)$.

NTRU Prime Key generation (where n is prime)

- Find inverse in $\mathbf{F}_{4591}[X]/(X^n - X - 1)$ (= a field).
- Find inverse in $\mathbf{F}_3[X]/(X^n - X - 1)$

Integer Modular Inversions in CSIDH

Needs inverse modulo $p = 4p_1p_2p_3 \cdots p_{73}p_{74} - 1$, where $p_1 \cdots p_{73}$ are the smallest 73 odd primes and $p_{74} = 587$.

An Example in $\mathbf{F}_7[X]$

Euclid-Stevin Algorithm

$$
\begin{aligned}
R_0 &= 2y^7 + 7y^6 + y^5 + 8y^4 + 2y^3 + 8y^2 + y + 8 \\
R_1 &= 3y^6 + y^5 + 4y^4 + y^3 + 5y^2 + 9y + 2 \\
R_2 &= R_0 - (3y + 6)R_1 = 4y^5 + 2y^4 + 2y^3 + 4y + 3 \\
R_3 &= R_1 - (6y + 6)R_2 = y^4 + 3y^3 + 2y^2 + 2y + 5 \\
R_4 &= R_2 - (4y + 4)R_3 = 3y^3 + 5y^2 + 4y + 4 \\
R_5 &= R_3 - (5y + 2)R_4 = 2y + 4 \\
R_6 &= R_4 - (5y^2 + 3y + 3)R_5 = 6 \\
R_7 &= R_5 - (5y + 3)R_6 = 0
\end{aligned}
$$

Non-Constant-Time

An "ideal" Euclidean step has dividend of degree 1 higher than the divisor, resulting in a remainder of degree 1 lower than the divisor.

An Example in $\mathbf{F}_7[X]$

Euclid-Stevin Algorithm

$$
\begin{aligned}
R_0 &= 2y^7 + 7y^6 + y^5 + 8y^4 + 2y^3 + 8y^2 + y + 8 \\
R_1 &= 3y^6 + y^5 + 4y^4 + y^3 + 5y^2 + 9y + 2 \\
R_2 &= R_0 - (3y + 6)R_1 = 4y^5 + 2y^4 + 2y^3 + 4y + 3 \\
R_3 &= R_1 - (6y + 6)R_2 = y^4 + 3y^3 + 2y^2 + 2y + 5 \\
R_4 &= R_2 - (4y + 4)R_3 = 3y^3 + 5y^2 + 4y + 4 \\
R_5 &= R_3 - (5y + 2)R_4 = 2y + 4 \\
R_6 &= R_4 - (5y^2 + 3y + 3)R_5 = 6 \\
R_7 &= R_5 - (5y + 3)R_6 = 0
\end{aligned}
$$

Non-Constant-Time

An "ideal" Euclidean step has dividend of degree 1 higher than the divisor, resulting in a remainder of degree 1 lower than the divisor. From R_4 to R_5 is non-ideal!

#Subtractions = #Coeffs. − 1 − #Skips

15 coefficients to start, 1 to end = 14 steps?

$$
\begin{aligned}
R_0 &= 2y^7 + 7y^6 + y^5 + 8y^4 + 2y^3 + 8y^2 + y + 8 \\
R_1 &= 3y^6 + y^5 + 4y^4 + y^3 + 5y^2 + 9y + 2 \\
& \quad R_0 - 3yR_1 = 4y^6 + 3y^5 + 5y^4 + y^3 + 2y^2 + 2y + 1 \\
R_2 &= R_0 - (3y + 6)R_1 = 4y^5 + 2y^4 + 2y^3 + 4y + 3 \\
& \quad R_1 - 6yR_2 = 3y^5 + 6y^4 + y^3 + 2y^2 + 5y + 2 \\
R_3 &= R_1 - (6y + 6)R_2 = y^4 + 3y^3 + 2y^2 + 2y + 5 \\
& \quad R_2 - 4yR_3 = 4y^4 + y^3 + 6y^2 + 5y + 3 \\
R_4 &= R_2 - (4y + 4)R_3 = 3y^3 + 5y^2 + 4y + 4 \\
& \quad R_3 - 5yR_4 = 6y^3 + 3y^2 + 3y + 5 \\
R_5 &= R_3 - (5y + 2)R_4 = 2y + 4 \\
& \quad R_4 - 5y^2R_5 = 6y^2 + 4y + 4 \\
& \quad R_4 - (5y^2 + 3y)R_5 = 6y + 4 \\
R_6 &= R_4 - (5y^2 + 3y + 3)R_5 = 6 \\
& \quad R_5 - 5yR_6 = 4 \\
R_7 &= R_5 - (5y + 3)R_6 = 0
\end{aligned}
$$

A Euclidean Subtraction stage

Starting from a Dividend of higher degree than Divisor

"Regular" Subtraction Stage

- Subtract from Dividend correct multiple of Divisor.

A Euclidean Subtraction stage

Starting from a Dividend of higher degree than Divisor

"Regular" Subtraction Stage

- Subtract from Dividend correct multiple of Divisor.
  - If "Dividend lead term" $= 0$, no problem!
- Decrement "Dividend" degree.
- If Divisor has higher degree than Dividend, Swap.

What if "the Divisor lead term" $= 0$?

- Decrement Divisor Degree, do dummy Subtraction

How did existing constant-time GCD do it?

- Do GCD in rising order from Constant term up
- Keep polynomial as arrays and track the degrees.

A Better Subtraction Stage

What we do differently

- Start the known (bigger) polynomial as "Divisor" !!
  - We can ensure that its lead term is non-zero!
- Track $\delta = \deg \text{Divisor} - \deg \text{Dividend}$.
- Can reverse polynomials (lead term $=$ "constant").

Our Subtraction Stage: "divstep"

- If δ is positive, and Dividend has a non-zero lead (constant) term, then Swap & negate δ.
- Take appropriate linear combination of Divisor and Dividend. Shift Dividend (divide by x), increment δ.

What do we do exactly

Details of computation with $R_0, R_1 \in k[x]$, $d = \deg R_0 > \deg R_1$

Setting up

- "Divisor" $f = x^d R_0(1/x)$, "Dividend" $g = x^{d-1} R_1(1/x)$, "Degree Difference" $\delta = 1$.
- Do $2d - 1$ divstep's (and collect return values).

divstep : $\mathbb{Z} \times k[[x]]^* \times k[[x]] \to \mathbb{Z} \times k[[x]]^* \times k[[x]]$,

$\mathsf{divstep}(\delta, f, g) :=$

$$\begin{cases} (1-\delta, g, (g(0)f - f(0)g)/x) & \text{if } \delta > 0 \text{ and } g(0) \neq 0, \\ (1+\delta, f, (f(0)g - g(0)f)/x) & \text{otherwise.} \end{cases}$$

n	δ_n	f_n x^0	x^1	x^2	x^3	x^4	x^5	x^6	x^7	x^8	x^9	...	g_n x^0	x^1	x^2	x^3	x^4	x^5	x^6	x^7	x^8	x^9	...
0	1	2	0	1	1	2	1	1	1	0	0	...	3	1	4	1	5	2	2	0	0	0	...
1	0	3	1	4	1	5	2	2	0	0	0	...	5	2	1	3	6	6	3	0	0	0	...
2	1	3	1	4	1	5	2	2	0	0	0	...	1	4	4	0	1	6	0	0	0	0	...
3	0	1	4	4	0	1	6	0	0	0	0	...	3	6	1	2	5	2	0	0	0	0	...
4	1	1	4	4	0	1	6	0	0	0	0	...	1	3	2	2	5	0	0	0	0	0	...
5	0	1	3	2	2	5	0	0	0	0	0	...	1	2	5	3	6	0	0	0	0	0	...
6	1	1	3	2	2	5	0	0	0	0	0	...	6	3	1	1	0	0	0	0	0	0	...
7	0	6	3	1	1	0	0	0	0	0	0	...	1	4	4	2	0	0	0	0	0	0	...
8	1	6	3	1	1	0	0	0	0	0	0	...	0	2	4	0	0	0	0	0	0	0	...
9	2	6	3	1	1	0	0	0	0	0	0	...	5	3	0	0	0	0	0	0	0	0	...
10	−1	5	3	0	0	0	0	0	0	0	0	...	4	5	5	0	0	0	0	0	0	0	...
11	0	5	3	0	0	0	0	0	0	0	0	...	6	4	0	0	0	0	0	0	0	0	...
12	1	5	3	0	0	0	0	0	0	0	0	...	2	0	0	0	0	0	0	0	0	0	...
13	0	2	0	0	0	0	0	0	0	0	0	...	6	0	0	0	0	0	0	0	0	0	...
14	1	2	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0	...
15	2	2	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0	...
16	3	2	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0	...
17	4	2	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0	...
18	5	2	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0	...
19	6	2	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0	...
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.

Table: Iterates $(\delta_n, f_n, g_n) = \text{divstep}^n(\delta, f, g)$ for $k = \mathbf{F}_7$, $\delta = 1$, $f = 2 + 7x + 1x^2 + 8x^3 + 2x^4 + 8x^5 + 1x^6 + 8x^7$, and $g = 3 + 1x + 4x^2 + 1x^3 + 5x^4 + 9x^5 + 2x^6$.

Time-Constant divstep

- first half $(\delta, f, g) \to (\delta', f', g')$,

$$\begin{aligned}
\mathit{swap} &= \begin{cases} -1 & \text{if } \delta > 0 \text{ and } g(0) \neq 0, \\ 0 & \text{otherwise.} \end{cases} \\
\mathit{mask} &= (f \text{ xor } g) \text{ and } \mathit{swap} \\
f' &= f \text{ xor } \mathit{mask} \\
g' &= g \text{ xor } \mathit{mask} \\
\delta' &= \delta \text{ xor } ((\delta \text{ xor } -\delta) \text{ and } \mathit{swap})
\end{aligned}$$

  (equivalent vector instructions are available).

- second half:

$$(\delta, f, g) \to (1 + \delta, f, (f(0)g - g(0)f)/x).$$

Results using divsteps

NTRU and NTRU Prime Rings

Inverting in $\mathbf{F}_3[x]/(x^{700} + x^{699} + \cdots + x + 1)$

- NTRU HRSS original: 150000 Haswell cycles
  - Tracks two extra indices compared to ours
  - Requires a scaling by variable x^r at the end
- Our Method: 90000 Haswell cycles

NTRU Prime Keygen

Mostly an Inversion in $\mathbf{F}_{4591}[x]/(x^{761} - x - 1)$

- Originally: 6 million cycles (Haswell)
- Ours: 0.94 million cycles (Haswell)

Radix-2 divstep for Integers case

divstep : $\mathbf{Z} \times \mathbf{Z}_2^* \times \mathbf{Z}_2 \to \mathbf{Z} \times \mathbf{Z}_2^* \times \mathbf{Z}_2, (\delta, f, g) \mapsto$

$$\begin{cases} (1-\delta, g, (g-f)/2) & \text{if } \delta > 0 \text{ and } g \text{ is odd,} \\ (1+\delta, f, (g+(g \bmod 2)f)/2) & \text{otherwise.} \end{cases}$$

2-adic divstep Split in Two Halves

- Conditional Swap: $(\delta, f, g) \to (-\delta, g, -f)$ if $g \bmod 2 = 1$ and $\delta > 0$, otherwise no change.
- Eliminate: $\delta \to \delta + 1$, $g = (g + (g \bmod 2)f)/2)$.

Termination Theorem

For k-bit f and g, $2.883k$ 2-adic divsteps result in a zero.

Sub-Quadratic GCD/Modular Inversion

Simpler Structure, no Middle Step

The transition matrix of $\text{divstep}^n(\delta, f, g)$ depends only on bottom n bits or coefficients of f, g.

- $n/2$ steps, update (f, g) with mults, $n/2$ more steps.
- n divsteps takes time $n \log^{2+o(1)} n$ with FFT mults.

Time-Constancy

- Prior: two recursive steps sandwiching a division. The latter to ensure progress.
  - A division is not naturally time-constant.
  - The split is not necessarily even.
- Ours: two equivalent recursive steps, even split.

Better on smaller CPUs, larger sizes

Integer Inversion Results

- Intel CPUs, $p = 2^{255} - 19$: 10050, 8778, and 8543 cycles on Haswell, Skylake, and Kaby Lake; Nath-Sarkar: 11854, 9301, and 8971 cycles (resp.)
- ARM Cortex A7 CPUs, $p = 2^{255} - 19$: 35277 cycles. Fujii-Aranha: 62648 cycles.
- Intel CPUs, $p = 2^{511} - 187$: Under 30000 cycles for all. Nath-Sarkar: 72804, 47062, and 45014 cycles on Haswell, Skylake, and Kaby Lake (resp.).

Polynomial $(\mathbf{F}_{4591}[X]/(X^{761} - X - 1))$ Inversion

752k Haswell/662k Skylake cycles $\rightarrow$ 707k/611k

What's Left

More Usage Cases

- SIDH/CSIDH integer inversions 1.5×–2× speedup
- LEDACrypt polynomial inversions 2×–4× speedup

A Prettier Theorem?

Find a direct proof (no exhaustive search) that k-bit integer divsteps terminate in $\propto k$ steps.

Future Work

- Use more Inversions if I/M small enough?
- Verification of Code