Jumping for Bernstein-Yang Inversion

Li-Jie Jian, **Dean Wang**, Bo-Yin Yang, Ming-Shing Chen

Institute of Information Science, Academia Sinica

2024.07.16

Motivation: Bernstein-Yang Algorithm [TCHES'19]

- NTRU
- NTRU Prime
- BIKE

Bernstein-Yang GCD algorithm

Bernstein-Yang GCD algorithm uses a matrix to keep track of changes in the process of GCD.

Definition

The algorithm determines a transition matrix $\mathcal{T}$ from the degree-0 coefficients of inputs f, g and their degree difference δ as

$$\mathcal{T}(\delta, f, g) = \begin{cases} \begin{bmatrix} 0 & 1 \\ \frac{g(0)}{x} & \frac{-f(0)}{x} \end{bmatrix} & \text{if } \delta > 0 \text{ and } g(0) \neq 0, \\ \\ \begin{bmatrix} 1 & 0 \\ \frac{-g(0)}{x} & \frac{f(0)}{x} \end{bmatrix} & \text{otherwise.} \end{cases}$$

Bernstein-Yang GCD algorithm

We compute reciprocal of polynomial g in $\mathbb{F}_q[x]/(x^p - x - 1)$ by performing a number of consecutive divsteps on $(x^p - x - 1, g)$ to obtain the transition matrix.

$$\begin{bmatrix} f_0 \\ g_0 \end{bmatrix} = \begin{bmatrix} u & v \\ q & r \end{bmatrix} \cdot \begin{bmatrix} x^p - x - 1 \\ g \end{bmatrix}$$

where

$$f_0 = u \cdot (x^p - x - 1) + v \cdot g \quad \to \quad f_0 \equiv v \cdot g \mod (x^p - x - 1)$$

we get $g^{-1} = v/f_0$ in $\mathbb{F}_q[x]/(x^p - x - 1)$.

Bernstein-Yang GCD algorithm

Algorithm 1 divsteps $(n,\ \delta,\ f,\ g)$

Input: $n \geq 0, \delta \in \mathbb{Z}$

Output: $\delta, f, g, M \in \mathbf{R}_q[x]^{2\times 2}$

$\begin{bmatrix} u & v \\ q & r \end{bmatrix} \in \mathbf{R}_q[x]^{2\times 2} \leftarrow \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

for $i \leftarrow 1$ to n **do**

    if $\delta > 0$ and $g_0 \neq 0$ **then** ▷ swap

        $\delta \leftarrow -\delta$

        $f, g, u, v, q, r \leftarrow g, f, q, r, u, v$

    end if

    $\delta \leftarrow \delta + 1$

    $g \leftarrow (g \cdot f_0 - f \cdot g_0)/x$

    $q, r \leftarrow (q \cdot f_0 - u \cdot g_0), (r \cdot f_0 - v \cdot g_0)$

    $u, v \leftarrow u \cdot x, v \cdot x$ ▷ Raise degree

end for

return $\delta, f, g, \begin{bmatrix} u & v \\ q & r \end{bmatrix}$

Algorithm 2 jumpdivstep $(n,\ \delta,\ f,\ g)$

Input: $n \geq 0, \delta \in \mathbb{Z}$

Output: $\delta, f, g, M \in \mathbf{R}_q[x]^{2\times 2}$

if $n < n_{threshold}$ **then**

    return divsteps(n, δ, f, g)

end if

$j \leftarrow \lfloor n/2 \rfloor$

$k \leftarrow n - j$

$\delta, f', g', M_1 \leftarrow$ jumpdivstep(j, δ, f, g)

$\begin{bmatrix} f \\ g \end{bmatrix} \leftarrow x^{-j} \cdot M_1 \cdot \begin{bmatrix} f \\ g \end{bmatrix} + \begin{bmatrix} f' \\ g' \end{bmatrix}$

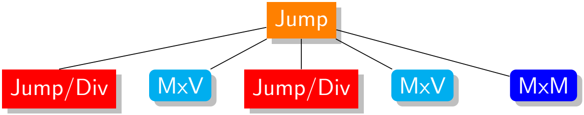
$\delta, f', g', M_2 \leftarrow$ jumpdivstep(k, δ, f, g)

$\begin{bmatrix} f \\ g \end{bmatrix} \leftarrow x^{-k} \cdot M_2 \cdot \begin{bmatrix} f \\ g \end{bmatrix} + \begin{bmatrix} f' \\ g' \end{bmatrix}$

$M \leftarrow M_2 \cdot M_1$

return δ, f, g, M

Jumpdivsteps



MxV:

$$\begin{bmatrix} f' \\ g' \end{bmatrix} = x^{-n} \times \begin{bmatrix} u_1 & v_1 \\ q_1 & r_1 \end{bmatrix} \times \begin{bmatrix} f \\ g \end{bmatrix}$$

MxM:

$$\mathcal{T}_2 \cdot \mathcal{T}_1 = \begin{bmatrix} u_2 & v_2 \\ q_2 & r_2 \end{bmatrix} \times \begin{bmatrix} u_1 & v_1 \\ q_1 & r_1 \end{bmatrix}$$

Matrix multiplication by NTT

Normally, NTT polynomial multiplication requires 2x input transforms, **1x point-wise multiplication** 1x output transform.

$$
\begin{array}{cccccccc}
\textbf{Normal} & -- & \begin{bmatrix} u_2 & v_2 \\ q_2 & r_2 \end{bmatrix} & & \begin{bmatrix} f' \\ g' \end{bmatrix} & & \begin{bmatrix} f' \\ g' \end{bmatrix} & \\
 & & \textbf{4x}\downarrow & & \textbf{2x}\downarrow & & \uparrow\textbf{2x} & \\
\textbf{NTT} & -- & \begin{bmatrix} u_2 & v_2 \\ q_2 & r_2 \end{bmatrix} & \times & \begin{bmatrix} f \\ g \end{bmatrix} & = & \begin{bmatrix} f' \\ g' \end{bmatrix} & \\
 & & \downarrow & & & & & \\
 & & \begin{bmatrix} u_2 & v_2 \\ q_2 & r_2 \end{bmatrix} & \times & \begin{bmatrix} u_1 & v_1 \\ q_1 & r_1 \end{bmatrix} & = & \begin{bmatrix} u' & v' \\ q' & r' \end{bmatrix} & \xrightarrow{\textbf{4x}} \begin{bmatrix} u' & v' \\ q' & r' \end{bmatrix}
\end{array}
$$

Saturated divsteps

- Sufficiently utilizes all storage of vector registers while keeping coefficients aligned as possible.
- Multiply x by rotating storage space to prevent overflow.

$$\mathcal{T} = \begin{bmatrix} u & v \\ q & r \end{bmatrix} \text{ or } \begin{bmatrix} u/x^n & v/x^n \\ q & r \end{bmatrix}$$

Saturated divsteps

$$u, v, q, r \rightarrow \boxed{x_0}\boxed{x_1}\boxed{x_2}\boxed{x_3}\boxed{x_4}\boxed{x_5}\boxed{\ldots}\boxed{x_{n-1}}$$

If degree of $g = 0$, the lift operations will only apply to the same pair, we denote u and v as:

$$u, v \rightarrow \boxed{\color{red}{x_n}}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{\ldots}\boxed{0}$$

Before NTT matrix multiplication, conditional multiplications are required to address this ***special case***.

Sheared divsteps

- Skips degree raising in the last step to prevent overflow.

$$\mathcal{T} = \begin{bmatrix} u/x & v/x \\ q & r \end{bmatrix}$$

$u, v \rightarrow$	x_1	x_2	x_3	x_4	x_5	x_6	$\dots$	x_n

$q, r \rightarrow$	x_0	x_1	x_2	x_3	x_4	x_5	$\dots$	x_{n-1}

Sheared divsteps

$$\begin{bmatrix} u_2/x & v_2/x \\ q_2 & r_2 \end{bmatrix} \begin{bmatrix} u_1/x & v_1/x \\ q_1 & r_1 \end{bmatrix} = \begin{bmatrix} u_2u_1/x^2 + v_2q_1/x & u_2v_1/x^2 + v_2r_1/x \\ q_2u_1/x + r_2q_1 & q_2v_1/x + r_2r_1 \end{bmatrix}$$

In M×M, since the degree of augend and addend are inconsistent, we can't add them in Toom/NTT form, taking additional output transforms.

Unsaturated divsteps

- Execute fewer steps of divsteps than storage size.

$$\mathcal{T} = \begin{bmatrix} u & v \\ q & r \end{bmatrix}$$

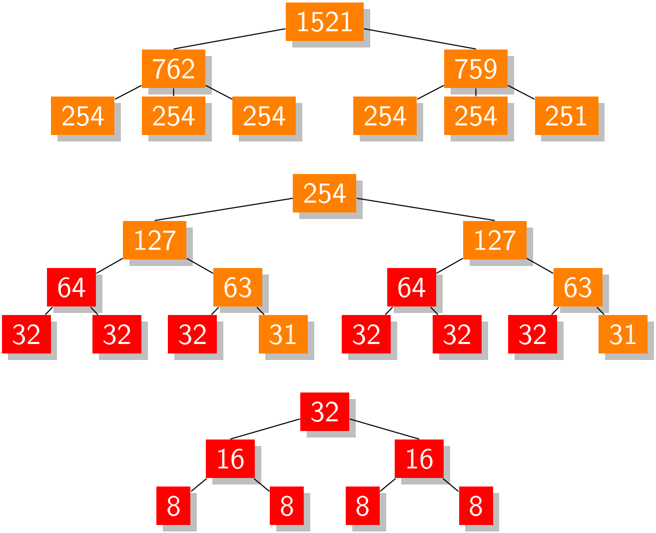
$$u, v, q, r \to \boxed{x_0} \boxed{x_1} \boxed{x_2} \boxed{x_3} \boxed{x_4} \boxed{x_5} \boxed{\dots} \boxed{x_{n-1}}$$

This method can eliminate all overhead present in previous versions.

Comparison

1. Use unsaturated divsteps as much as possible.
2. If the structure still lacks steps, we use some sheared divsteps evenly to gain extra steps.

	Operation	In	Mul	Out	ceq	dup	and	or	mvn	ext
MxV	**Saturated**	6	4	2	2	2	2m	2m	0	0
	Sheared	6	4	2	0	0	0	0	0	2m
	Unsaturated	6	4	2	0	0	0	0	0	4m
MxM	**Saturated**	0	8	4	0	0	10m	8m	2(m-1)	0
	Sheared	0	8	8	0	0	0	0	0	8m
	Unsaturated	0	8	4	0	0	0	0	0	0


1521
762
759
254
254
254
254
254
251
254
127
127
64
63
64
63
32
32
32
31
32
32
32
31
32
16
16
8
8
8
8

Matrix multiplication

Length	Algorithm	In	Mul	Out	PxP	MxV	MxM	Jump
8x8	Schoolbook	0	94	0	94	376	752	1,504
	Karatsuba	0	56	0	56	224	448	896
	Extend	0	50	0	50	200	400	800
	Batched(x8)	0	360	0	360	-	-	-
16x16	Schoolbook	0	231	0	231	924	1,848	3,696
	Karatsuba	0	182	0	182	728	1,456	2,912
32x32	Schoolbook	0	760	0	760	3,040	6,080	12,160
	Toom	114	374	462	950	2,762	5,296	10,364
	Karatsuba	0	614	0	614	2,456	4,912	9,824
64x64	Schonhage	367	2,319	521	3207	11419	22,104	43,474
	Karatsuba	0	1,999	0	1,999	7,996	15,992	31,984
	Toom	207	1,295	944	2,446	7,689	14,964	29,514
	Rader**[Hwang24]**	1,228	411	570	2,209	6,468	10,480	18,504
128x128	Karatsuba	0	6,998	0	6,998	27,992	55,984	111,968
	Schonhage	1,691	4,903	1,521	8,115	27,727	52,072	100,762
	Toom	454	3,096	1,896	5,446	17,538	34,168	67,428
	Bruun	1,982	2,443	1,764	6,189	19,246	34,528	65,092
	Rader	2,908	828	1,240	4,976	14,516	23,216	40,616
768x768	Good-3	11,022	2,494	5,349	18,865	53,740	85,436	222,520

Jumping for $\mathbb{Z}_3[x]/\langle x^{761} - x - 1\rangle$

We perform Divsteps for steps less than 128. Since it only requires 2 bits to store the coefficients in F_3, we divide them into 2 vectors, sign-bits and value-bits, to achieve further acceleration.

value :	v_0	v_1	v_2	v_3	v_4	v_5	$\dots$	v_{n-1}
sign :	s_0	s_1	s_2	s_3	s_4	s_5	$\dots$	s_{n-1}

After that, we use 8 bits to store the coefficients, and start to perform Jumpdivsteps.

Result

Benchmark for key generation in **sntrup761**.

sntrup761	**Supercop/Jumpdivsteps**	**Divsteps/Jumpdivsteps**
Cortex-A53	13x	2.5x
Cortex-A72	12x	**2.8x**
Cortex-A76	12x	2.1x
M1	**29x**	2.2x

Takeaway

- We exploit the structure of Jumpdivsteps, and are the first to make it faster than Divsteps in practical.
- We implement fast key generation for NTRU-Prime.
- We optimize matrix multiplications in various length by revising NTT algorithms.