

Aritmetik på stora heltal – algebra, algoritmer och assembly

Torbjörn Granlund

Avancerade algoritmer 2013

DEL 1: Optimeringsverktyg (för aritmetik på stora heltal)

DEL 2: Multiplikation i GMP

Verktyg 1: Algebra. Exempel: RSA-signering

Vi ska beräkna RSA- n (i tid $O(n^3)$)

$$s = m^d \bmod pq$$

där p och q är primtal, och $n = \log pq \approx \log m \approx \log d$.

Sätt $d_p = d \bmod (p - 1)$ och $d_q = d \bmod (q - 1)$.

Utför sedan de två exponentieringarna:

$$s_p = (m \bmod p)^{d_p} \bmod p$$

$$s_q = (m \bmod q)^{d_q} \bmod q$$

Sedan ges s med CRT från s_p och s_q (i tid $O(n^2)$).

Verktyg 1: Algebra. Exempel: RSA-signering

Vi ska beräkna RSA- n (i tid $O(n^3)$)

$$s = m^d \bmod pq$$

där p och q är primtal, och $n = \log pq \approx \log m \approx \log d$.

Sätt $d_p = d \bmod (p - 1)$ och $d_q = d \bmod (q - 1)$.

Utför sedan de två exponentieringarna:

$$s_p = (m \bmod p)^{d_p} \bmod p$$

$$s_q = (m \bmod q)^{d_q} \bmod q$$

Sedan ges s med CRT från s_p och s_q (i tid $O(n^2)$).

Verktyg 1: Algebra. Exempel: RSA-signering

Vi ska beräkna RSA- n (i tid $O(n^3)$)

$$s = m^d \bmod pq$$

där p och q är primtal, och $n = \log pq \approx \log m \approx \log d$.

Sätt $d_p = d \bmod (p - 1)$ och $d_q = d \bmod (q - 1)$.

Utför sedan de två exponentieringarna:

$$s_p = (m \bmod p)^{d_p} \bmod p$$

$$s_q = (m \bmod q)^{d_q} \bmod q$$

Sedan ges s med CRT från s_p och s_q (i tid $O(n^2)$).

Verktyg 2: Effektiva algoritmer

Exempel:

Karatsubas dekompositionsalgorithm för multiplikation.

$$U = 2^n U_1 + U_0, \quad V = 2^n V_1 + V_0$$

$$UV = (2^{2n} + 2^n)U_1 V_1 - 2^n(U_1 - U_0)(V_1 - V_0) + (2^n + 1)U_0 V_0$$

Verktøy 3: Algoritmvæl etter operandstorlek (1)

Naiv Karatsuba-implementasjon:

```
mul (word *w, word *u, word *v, size_t n)
{
    if (n == 1)
        w[0] = LO (u[0] * v[0]);
        w[1] = HI (u[0] * v[0]);
    else    /* Karatsuba code */
        U1 = u + n/2; U0 = u; V1 = v + n/2; V0 = v;
        mul (P0, U1, V1, n/2);
        mul (P1, U0, V0, n/2);
        sub (Ud, U1, U0, n/2); sub (Vd, V1, V0, n/2);
        mul (Pd, Ud, Vd, n/2);
        copy (w, P0, n);          copy (w + n, P1, n);
        add (w + n/2, w + n/2, P0, n);
        add (w + n/2, w + n/2, P1, n);
        sub (w + n/2, w + n/2, Pd, n);
}
```

Verktøy 3: Algoritmvæl etter operandstorlek (2)

Listigare Karatsuba-implementation:

```
mul (word *w, word *u, word *v, size_t n)
{
    if (n < 17)
        mul_basecase (w, u, v, n);
    else /* Karatsuba code */
        U1 = u + n/2; U0 = u; V1 = v + n/2; V0 = v;
        mul (P0, U1, V1, n/2);
        mul (P1, U0, V0, n/2);
        sub (Ud, U1, U0, n/2); sub (Vd, V1, V0, n/2);
        mul (Pd, Ud, Vd, n/2);
        copy (w, P0, n);          copy (w + n, P1, n);
        add (w + n/2, w + n/2, P0, n);
        add (w + n/2, w + n/2, P1, n);
        sub (w + n/2, w + n/2, Pd, n);
}
```

Verktyg 3: Algoritmval efter operandstorlek (3)

Resultat:

Naiva Karatsuba-koden är enligt mina tester snabbare än basfallsmultiplikation från 8000 bitar (ca 2400 decimaler).

Listiga Karatsuba-koden är snabbare redan vid ca 830 bitar (250 decimaler).

(Tester utförda på Athlon.)

Slutsats 1:

En oavancerad implementation
av en avancerad algoritm kan vara till skada.

Verktyg 4: Minnes- och cache-lokalitet

- ▶ Temporal lokalitet
- ▶ Spatial lokalitet
- ▶ Data layout, padding

Algoritmegenskap: Dekompositionsalgoritmer har god lokalitet.

Verktyg 5: Utrullning av loopar

Istället för:

```
for (i = 0; i < n; i++)  
    jobbenhet
```

Skriver vi:

```
for (i = 0; i < n mod 4; i++)  
    jobbenhet  
for (i = 0; i < n; i += 4)  
    jobbenhet  
    jobbenhet  
    jobbenhet  
    jobbenhet
```

Verktyg 6: Software pipelining (1)

Mål: Hantera fördröjningar (latency) för operationer.

Metod: Vi gör om loopen...

```
for (...)  
{  
    a0 = *ap++;  
    b0 = *bp++;  
    r0 = a0 * b0;  
    *rp++ = r0;  
}
```

Verktøy 6: Software pipelining (2)

... till:

```
for (...)
{
    *rp++ = r0;
    r0 = a0 * b0;
    a0 = *ap++;
    b0 = *bp++;
}
```

Verkt yg 6: Software pipelining (3)

Med feed-in och wind-down:

```
a0 = *ap++;  
b0 = *bp++;  
r0 = a0 * b0;  
a0 = *ap++;  
b0 = *bp++;  
for (...)  
{  
    *rp++ = r0;  
    r0 = a0 * b0;  
    a0 = *ap++;  
    b0 = *bp++;  
}  
*rp++ = r0;  
r0 = a0 * b0;  
*rp++ = r0;
```

Verktyg 5 + 6: Kombinera utrullning och software pipelining

feed-in	pipelined loop	wind-down
-----	-----	-----
a0 = *ap++;	for (...)	
b0 = *bp++;	{	
a1 = *ap++;	*rp++ = r0;	*rp++ = r0;
b1 = *bp++;	r0 = a0 * b0;	r0 = a0 * b0;
	a0 = *ap++;	*rp++ = r1;
r0 = a0 * b0;	b0 = *bp++;	r1 = a1 * b1;
a0 = *ap++;	*rp++ = r1;	
b0 = *bp++;	r1 = a1 * b1;	*rp++ = r0;
r1 = a1 * b1;	a1 = *ap++;	*rp++ = r1;
a1 = *ap++;	b1 = *bp++;	
b1 = *bp++;	}	

Verktyg 7: "Grundifiering" av rekurrenser (1)

Definition: Rekurrens = beroende mellan konsekutiva iterationer.

I aritmetisk kod: Olika varianter av propagering av minnessiffra ("carry")

Påstående: Har vi k operationer för att generera utdata för en rekurrens från det att vi konsumerar indata för rekurrensen, kan ingen processor i världen utföra en iteration på $< k$ steg.

Verktøy 7: "Grundifisering" av rekurrens (2)

Ganska djup rekurrens:

```
add (word *r, word *u, word *v, size_t n)
{
    cy = 0;
    for (i = 0; i < n; i++)
    {
        uword = u[i];
        vword = v[i];
        sum0 = uword + cy;
        cy0 = sum0 < uword;
        sum1 = sum0 + vword;
        cy1 = sum1 < sum0;
        cy = cy0 + cy1;
        r[i] = sum1;
    }
}
```

Verktøy 7: "Grundifisering" av rekurrenser (2b)

Ganska djup rekurrens:

```
add (word *r, word *u, word *v, size_t n)
{
    cy = 0;
    for (i = 0; i < n; i++)
    {
        uword = u[i];
        vword = v[i];
        sum0 = uword + cy;           0         4         8
        cy0 = sum0 < uword;         1         5         ...
        sum1 = sum0 + vword;        1         5
        cy1 = sum1 < sum0;          2         6
        cy = cy0 + cy1;            3         7
        r[i] = sum1;
    }
}
```

Verktøy 7: "Grundifisering" av rekurrenser (3)

Mindre djup rekurrens:

```
add (word *r, word *u, word *v, size_t n)
{
    cy = 0;
    for (i = 0; i < n; i++)
    {
        uword = u[i];
        vword = v[i];
        sum0 = uword + vword;
        cy0 = sum0 < uword;
        sum1 = sum0 + cy;
        cy1 = sum1 < sum0;
        cy = cy0 + cy1;
        r[i] = sum1;
    }
}
```

Verktøy 7: "Grundifisering" av rekurrenser (3b)

Mindre djup rekurrens:

```
add (word *r, word *u, word *v, size_t n)
{
    cy = 0;
    for (i = 0; i < n; i++)
    {
        uword = u[i];
        vword = v[i];
        sum0 = uword + vword;
        cy0 = sum0 < uword;
        sum1 = sum0 + cy;           0           3           6
        cy1 = sum1 < sum0;         1           4           ...
        cy = cy0 + cy1;           2           5
        r[i] = sum1;
    }
}
```

Verktøy 7: "Grundifisering" av rekurrenser (4)

Riktig grund rekurrens:

```
add (word *r, word *u, word *v, size_t n)
{
    cy = 0;
    for (i = 0; i < n; i++)
    {
        uword = u[i];
        vword = v[i];
        sum0 = uword + vword;
        cy0 = sum0 < uword;
        sum1 = sum0 + cy;
        cy1 = cy & (sum0 == ~0);
        cy = cy0 + cy1;
        r[i] = sum1;
    }
}
```

Verktøy 7: "Grundifisering" av rekurrenser (4b)

Riktig grund rekurrens:

```
add (word *r, word *u, word *v, size_t n)
{
    cy = 0;
    for (i = 0; i < n; i++)
    {
        uword = u[i];
        vword = v[i];
        sum0 = uword + vword;
        cy0 = sum0 < uword;
        sum1 = sum0 + cy;
        cy1 = cy & (sum0 == ~0);
        cy = cy0 + cy1;
        r[i] = sum1;
    }
}
```

0	2	4
0	2	...
1	3	

Verktyg 8: Assembly

Implementera i assembly!

- ▶ Leta användbara instruktioner
- ▶ Designa mikro-algoritmer efter tillgängliga instruktioner
- ▶ Ta hänsyn till fördröjning för valda instruktioner
- ▶ Vilka instruktioner kan utföras parallellt?
- ▶ Alignment
- ▶ Trial-and-measure
- ▶ Trial-and-measure
- ▶ ...

Verktyg 9: Största möjliga hopplöshet

Villkorliga hopp av två sorter:

1. Prediktabla
2. Slumpmässiga (eller av andra skäl icke prediktabla)

Ett icke prediktabelt hopp kostar som ≈ 30 andra instruktioner.

Optimeringsverktyg för aritmetik på stora heltal

1. Algebra
2. Effektiva algoritmer
3. Algoritmval efter operandstorlek
4. Minnes- och cache-lokalitet
5. Utrullning av loopar
6. Software pipelining
7. "Grundifiering" av rekurrensen
8. Assembly
9. Hopp-prediktabilitet

DEL 2: Stora heltal i GMP

Heltalsmultiplikation

Problem: Beräkna $W \leftarrow U \times V$, $U, V \in \mathbb{Z}$

I GMP $\log_2(U), \log_2(V) < 2^{50}$

Mål: Maximal praktiskt prestanda + lägsta komplexitet

Algoritm-1: Klassisk multiplikation (1)

$$W = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \beta^{i+j} u_i v_j = \sum_{i=0}^{n-1} \left(\beta^i u_i \sum_{j=0}^{n-1} \beta^j v_j \right)$$

Tidskomplexitet: $O(n^2)$

Vår `mul_basecase` kan bli väldigt enkel:

```
mul_basecase (word *w, word *u, size_t un, word *v, size_t vn)
{
    zero (w, un + vn);
    for (i = 0; i < vn; i++)
        w[un + i] = mulladd (w + i, u, un, v[i]);
}
```

Hur ser mul1add ut? Kanske så här:

```
mul1add (word *w, word *u, size_t un, word vword)
{
    cy_word = 0;
    for (j = 0; j < un; j++)
    {
        uword = u[j];
        lo = LO (uword * vword);
        hi = HI (uword * vword);
        wword = w[j];
        w[j] = LO (wword + lo + cy_word);
        cy_word = hi + HI (wword + lo + cy_word);
    }
    return cy_word;
}
```

Eller så här (PowerPC64):

```
mul1add:
    mtctr    r5
    li       r9, 0          # cy_word = 0
    addic    r0, r0, 0      # hw cy flag = 0
    addi     r3, r3, -8
    addi     r4, r4, -8
    nop
    nop
    nop                # alignment
    nop                # alignment
    nop                # alignment

L1: ldu      r0, 8(r4)      # r0 = (++u)
    ld       r10, 8(r3)    # r10 = (*w)
    mulld    r7, r0, r6     # low 64 product bits
    mulhdu   r8, r0, r6     # high 64 product bits
    adde     r7, r7, r9     # add old cy_word [in]
    addze    r9, r8         # new cy_word      [out]
    addc     r7, r7, r10    # add loaded (*w)
    stdu     r7, 8(r3)     # ++w = result
    bdnz     L1

    addze    r3, r9
    blr
```

Eller lite mer komplisert på Alpha...

	ldq	r3,	0(r17)	r28	mulq	r19,	r3,	r2		ldq	r17,	32(r17)		stq	r22,	0(r16)		stq	r23,	-24(r16)	
	and	r18,	7,		umulh	r19,	r3,	r6		ldq	r19,	r17,	r25	stq	r23,	8(r16)		bis	r31,	r31,	r31
	ldq	r18,	-8(r18)		ldq	r19,	r1,	r7		ldq	r16,	-16(r16)		bis	r31,	r31,	r31	mulq	r19,	r19,	r28
	cmpeq	r21,	1,	r21	ldq	r0,	16(r17)			umulh	r16,	r8,	r3	mulq	r19,	r1,	r28	bis	r31,	r31,	r31
\$1mod8:	beq	r24,	0(r16)		umulh	r19,	r4,	r24		cmplut	r4,	r2,	r20	bis	r31,	r31,	r31	addq	r8,	r21,	r8
	mulq	r19,	r3,	r7	ldq	r1,	24(r17)			mulq	r19,	r1,	r28	ldq	r4,	r21,	r8	cmplut	r4,	r8,	r22
	umulh	r19,	r3,	r8	mulq	r19,	r0,	r25		cmplut	r4,	r2,	r20	cmplut	r4,	r2,	r20	ldl	r31,	256(r17)	
	addq	r8,	r7,	r23	ldq	r5,	8(r16)			bis	r31,	r31,	r31	ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	cmplut	r23,	r7,	r0	umulh	r19,	r0,	r3		bis	r31,	r31,	r31	ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	stq	r23,	0(r16)		addq	r4,	r2,	r4		ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	bge	r18,	Sent1		ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	ret	r31,	(r26)	1	ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
\$11:	ldq	r8,	0(r31)		ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	ldq	r24,	0(r31)		ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	cmpeq	r20,	2,	r21	ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	bne	r21,	\$2mod8		ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	cmpeq	r20,	3,	r21	ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	bne	r21,	\$3mod8		ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	cmpeq	r20,	4,	r21	ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	bne	r21,	\$4mod8		ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	cmpeq	r20,	5,	r21	ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	bne	r21,	\$5mod8		ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	cmpeq	r20,	6,	r21	ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	bne	r21,	\$6mod8		ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	cmpeq	r20,	7,	r21	ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
\$7mod8:	beq	r21,	\$8mod8		ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	ldq	r17,	8(r17)		ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	mulq	r19,	r3,	r7	ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	umulh	r19,	r3,	r24	ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	addq	r5,	r7,	r23	ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	cmplut	r23,	r7,	r20	ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	addq	r24,	r20,	r24	ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	stq	r23,	0(r16)		ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	ldq	r16,	8(r16)		ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	ldq	r3,	8(r17)		ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
\$6mod8:	ldq	r1,	8(r17)		ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	mulq	r19,	r3,	r25	ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	ldq	r3,	r3		ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	mulq	r19,	r1,	r28	ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	ldq	r0,	16(r17)		ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	ldq	r4,	0(r16)		ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	umulh	r19,	r1,	r8	ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	ldq	r1,	24(r17)		ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	ldq	r17,	48(r17)		ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	mulq	r19,	r0,	r2	ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	ldq	r5,	8(r16)		ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	ldq	r16,	-32(r16)		ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	umulh	r19,	r0,	r6	ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	addq	r4,	r25,	r4	ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	mulq	r19,	r1,	r7	ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	br	r31,	Sent6		ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	ldq	r17,	8(r17)		ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	ldq	r16,	8(r16)		ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	ldq	r8,	0(r0)		ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
	ldq	r3,	0(r17)		ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	
\$8mod8:	ldq	r3,	0(r17)		ldq	r1,	24(r17)			ldq	r17,	64(r17)		ldq	r0,	-32(r17)		ldl	r31,	256(r17)	

Prestanda nu

n	Base
10^0	3 ns
10^1	84 ns
10^2	7,5 μ s
10^3	740 μ s
10^4	75 ms
10^5	7,5 s
10^6	997 s
10^7	1,2 dgr
10^8	220 dgr
10^9	60 år

(Tider på en 2.9 GHz Haswell PC, GMP 5.2.)

Bas- β heltal vs polynom (1)

Integer:

$$U = \sum_{i=0}^{n-1} u_i \beta^i, \quad u_i < \beta$$

Motsvarande polynom:

$$u(x) = \sum_{i=0}^{n-1} u_i x^i$$

Bas- β heltal vs polynom (1)

Integer:

$$U = \sum_{i=0}^{n-1} u_i \beta^i, \quad u_i < \beta$$

Motsvarande polynom:

$$u(x) = \sum_{i=0}^{n-1} u_i x^i$$

Bas- β heltal vs polynom (2)

Om vi har ett tal i bas-10

18 5054 0856 8445 1320 8201

och låter $\beta = 10^4$, så motsvarar detta polynomet

$$18x^5 + 5054x^4 + 0856x^3 + 8445x^2 + 1320x + 8201$$

Algorithm-2: Karatsubas “magiska formel” (1)

Givet heltal $U, V < 2^{2n}$. Låt $\beta = 2^n$.

$$U = \beta U_1 + U_0, \quad V = \beta V_1 + V_0$$

$$\begin{aligned} UV &= (\beta^2 + \beta)U_1 \times V_1 + \\ &\quad - \beta(U_1 - U_0) \times (V_1 - V_0) + \\ &\quad + (\beta + 1)U_0 \times V_0 \end{aligned}$$

Tidskomplexitet:

$$T(n) = 3T(n/2) + O(n)$$

$$T(n) \in O(n^{\log 3 / \log 2}) \subset O(n^{1.59})$$

Prestanda nu

n	Base	Kara
10^0	3 ns	n/a
10^1	84 ns	115 ns (bad!)
10^2	7,5 μ s	4,7 μ s
10^3	740 μ s	193 μ s
10^4	75 ms	7,7 ms
10^5	7,5 s	0,3 s
10^6	997 s	11 s
10^7	1,2 dgr	7,4 min
10^8	220 dgr	4 tim
10^9	60 år	8 dgr

Algorithm-3: Tooms generalisering av Karatsuba (1)

Låt heltalet U representeras av polynomet $u(x)$, dvs $u(\beta) = U$ för ett β^n vi väljer. Analogt för V , $v(x)$.

Tooms observation: Vi kan evaluera $u(x)$ and $v(x)$ i några punkter $x_0, x_1 \dots x_k$, sedan multiplicera $u(x_0)$ med $v(x_0)$, $u(x_1)$ med $v(x_1)$ etc. Produkten $w(x)$ får man fram med interpolation.

Om $u(x)$ och $v(x)$ har grad k , $w(x)$ kommer ha graden $2k$, och vi behöver $2k + 1$ punkter för att bestämma koefficienterna hos $w(x)$.

I Toom-språk, så har Karatsubas algoritm $k = 1$ och evaluerar i punkterna $0, -1$ och ∞ .

Algorithm-3: Tooms generalisering av Karatsuba (1)

Låt heltalet U representeras av polynomet $u(x)$, dvs $u(\beta) = U$ för ett β^n vi väljer. Analogt för V , $v(x)$.

Tooms observation: Vi kan evaluera $u(x)$ and $v(x)$ i några punkter $x_0, x_1 \dots x_k$, sedan multiplicera $u(x_0)$ med $v(x_0)$, $u(x_1)$ med $v(x_1)$ etc. Produkten $w(x)$ får man fram med interpolation.

Om $u(x)$ och $v(x)$ har grad k , $w(x)$ kommer ha graden $2k$, och vi behöver $2k + 1$ punkter för att bestämma koefficienterna hos $w(x)$.

I Toom-språk, så har Karatsubas algoritm $k = 1$ och evaluerar i punkterna $0, -1$ och ∞ .

Algorithm-3: Tooms generalisering av Karatsuba (2)

Exempel:

Kapa operanderna U and V in 3 delar vardera för att forma två grad-2 polynom $u(x)$ and $v(x)$. Vi behöver evaluera i $k + 1 = 5$ punkter, t ex $-1, 0, +1, +2, \infty$.

Tidskomplexitet:

$$T(n) = 5T(n/3) + O(n)$$

$$T(n) \in O(n^{\log 5 / \log 3}) \subset O(n^{1.47})$$

Kapa operanderna i 4 delar och evaluera i $k + 1 = 7$ punkter, t ex $-1, -1/2, 0, +1/2, +1, +2, \infty$.

Tidskomplexitet:

$$T(n) = 7T(n/4) + O(n)$$

$$T(n) \in O(n^{\log 7 / \log 4}) \subset O(n^{1.41})$$

Prestanda nu

n	Base	Kara	Toom 3,4
10^0	3 ns	n/a	n/a
10^1	84 ns	115 ns	n/a
10^2	7,5 μ s	4,7 μ s	4,6 μ s
10^3	740 μ s	193 μ s	147 μ s
10^4	75 ms	7,7 ms	4,1 ms
10^5	7,5 s	0,3 s	107 ms
10^6	997 s	11 s	2,8 s
10^7	1,2 dgr	7,4 min	1,17 min
10^8	220 dgr	4 tim	30 min
10^9	60 år	8 dgr	12 tim

Algoritm-4: FFT-familjen

FFT är en algoritm som beräknar vissa DFTs effektivt. Indata är ett $\text{grad-}2^k$ -polynom, utdata är ett annat $\text{grad-}2^k$ -polynom.

FFT behöver koefficienter i en ring R med *principalenhetsrötter* av ordning 2^k

En FFT-baserad heltalsmultiplikation ser ut så här:

1. $u(x) \leftarrow \text{SPLIT}(U), v(x) \leftarrow \text{SPLIT}(V),$
2. $u'(x) \leftarrow \text{FFT}(u(x)), v'(x) \leftarrow \text{FFT}(v(x))$
3. $p'(x) \leftarrow u'(x) \cdot v'(x)$ punktvis multiplikation
4. $p(x) \leftarrow \text{FFT}^{-1}(p'(x))$
5. $P = p(\beta)$

Prestanda nu

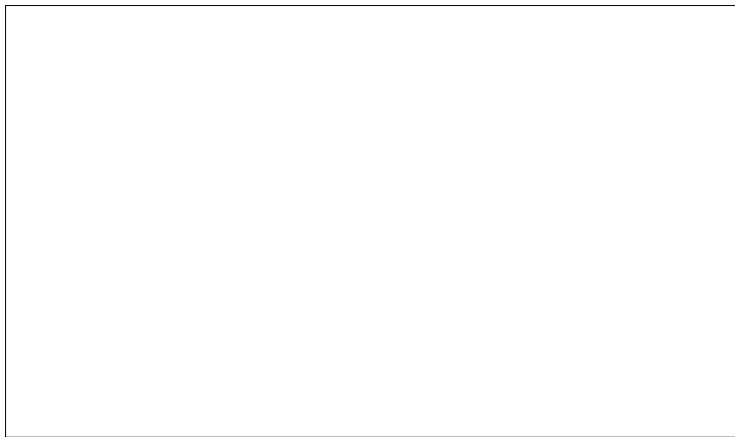
n	Base	Kara	Toom 3,4	SS FFT
10^0	3 ns	n/a	n/a	
10^1	84 ns	115 ns	n/a	
$10^{1,5}$	750 ns	661 ns	970 ns	
10^2	7,5 μ s	4,7 μ s	4,6 μs	8.3 μ s
10^3	740 μ s	193 μ s	147 μs	187 μ s
10^4	75 ms	7,7 ms	4,1 ms	2,8 ms
10^5	7,5 s	0,3 s	107 ms	44 ms
10^6	997 s	11 s	2,8 s	0,58 s
10^7	1,2 dgr	7,4 min	1,17 min	7,1 s
10^8	220 dgr	4 tim	30 min	100 s
10^9	60 år	8 dgr	12 tim	20 min

Likstora vs olikstora operands (1)

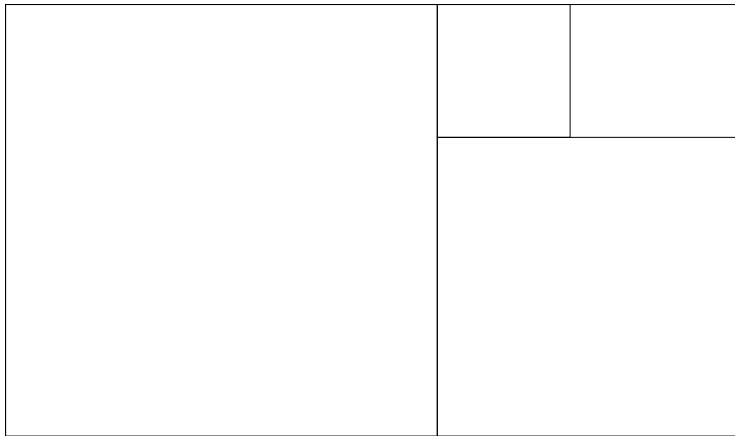
Vi har hittills bara berört likstora operander, och mappat dessa på två polynom av samma gradtal.

Hur kan vi hantera olikstora operander?

Likstora vs olikstora operander (2)



Likstora vs olikstora operander (3)



Anpassning av algoritmer till olikstora operander

- ▶ Basfallsalgoritmen fungerar
- ▶ Karatsuba-Toom inte så uppenbar
- ▶ Karatsuba-Toom-Bodrato-Zanoni hjälper
- ▶ FFT är "enkel" (men blir söligare)

Bodrato-Zanonis generalisering av Toom

År 2006 generaliserade M.Bodrato and A.Zanoni Tooms algoritmen att använda polynom $u(x)$, $v(x)$ av **olika** gradtal.

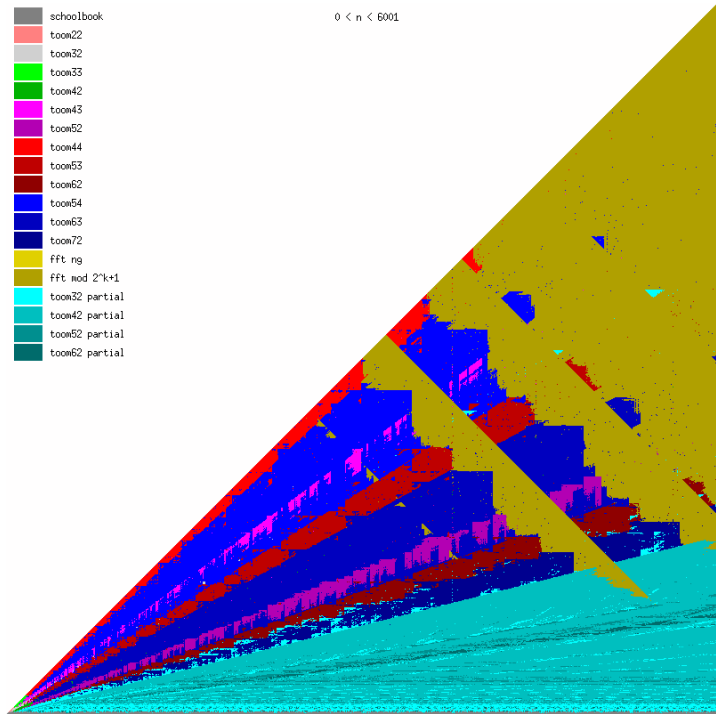
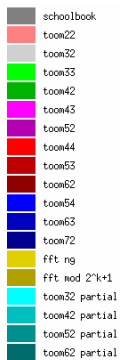
Detta är användbart för multiplikation av *olikstora* heltalsoperander.

Exempel: Om storleken av U och V är i relationen 3:2, kan vi mappa U på ett grad-2-polynom $u(x)$ och V på ett grad-1-polynom $v(x)$. Vi behöver evaluera i 4 punkter.

Toom-Bodrato-Zanoni primitiv i GMP

deg(u)	deg(v)	k	punkter	namn
1	1	3	$-1, 0, \infty$	toom22_mul
2	1	4	$-1, 0, +1, \infty$	toom32_mul
2	2	5	$-1, 0, +1, +2, \infty$	toom33_mul
3	1	5	$-1, 0, +1, +2, \infty$	toom42_mul
3	2	6	$-2, -1, 0, +1, +2, \infty$	toom43_mul
4	1	6	$-2, -1, 0, +1, +2, \infty$	toom52_mul
3	3	7	$-2, -1, 0, +1/2, +1, +2, \infty$	toom44_mul
4	2	7	$-2, -1, 0, +1/2, +1, +2, \infty$	toom53_mul
5	1	7	$-2, -1, 0, +1/2, +1, +2, \infty$	toom62_mul

$0 < n < 6001$



schoolbook

toom22

toom32

toom33

toom42

toom43

toom52

toom44

toom53

toom62

toom54

toom63

toom72

fft ng

fft mod 2^k+1

toom32 partial

toom42 partial

toom52 partial

toom62 partial

$15 < n < 6001$

