Arithmetic on large numbers – algebra, algorithms, and assembly code

Torbjörn Granlund

Advanced algorithms 2014
PART 1: Optimiser tools (for arithmetic on large integers)
PART 2: Multiplication in GMP
Tool 1: Algebra. Example: RSA signing

We are to compute RSA-$n$ (in time $O(n^3)$)

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

where $p$ and $q$ are prime numbers, and $n = \log pq \approx \log m \approx \log d$.

Let $d_p = d \mod (p-1)$ and $d_q = d \mod (q-1)$.

Then perform the two exponentiations:

$$s_p = (m \mod p)^{d_p} \mod p$$
$$s_q = (m \mod q)^{d_q} \mod q$$

We then get $s$ through CRT from $s_p$ and $s_q$ (in time $O(n^2)$).
We are to compute RSA-n (in time $O(n^3)$)

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

where $p$ and $q$ are prime numbers, and $n = \log pq \approx \log m \approx \log d$.

Let $d_p = d \mod (p - 1)$ and $d_q = d \mod (q - 1)$.

Then perform the two exponentiations:

$$s_p = (m \mod p)^{d_p} \mod p$$
$$s_q = (m \mod q)^{d_q} \mod q$$

We then get $s$ through CRT from $s_p$ and $s_q$ (in time $O(n^2)$).
Tool 1: Algebra. Example: RSA signing

We are to compute RSA-$n$ (in time $O(n^3)$)

\[ s = m^d \mod pq \]

where $p$ and $q$ are prime numbers, and $n = \log pq \approx \log m \approx \log d$.

Let $d_p = d \mod (p-1)$ and $d_q = d \mod (q-1)$.

Then perform the two exponentiations:

\[ s_p = (m \mod p)^{d_p} \mod p \]
\[ s_q = (m \mod q)^{d_q} \mod q \]

We then get $s$ through CRT from $s_p$ and $s_q$ (in time $O(n^2)$).
Example:
Karatsuba’s divide-and-conquer algorithm for multiplication.

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

\[ UV = (2^{2n} + 2^n)U_1 V_1 - 2^n(U_1 - U_0)(V_1 - V_0) + (2^n + 1)U_0 V_0 \]
Naive Karatsuba implementation:

```c
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);
}
```
Cleverer Karatsuba implementation:

```c
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);
    }
```
Result:

The naive Karatsuba code is faster than base-case multiplication from 8000 bits (ca 2400 decimals).

The cleverer Karatsuba code is faster already at 830 bits (250 decimals).

(Tests done on Athlon64.)
Conclusion:

An unadvanced implementation of an advanced algorithm can be harmful.
Tool 4: Memory and cache locality

- Temporal locality
- Spatial locality
- Data layout, padding

Algorithm property: Divide-and-conquer algorithms have good locality.
Tool 5: Loop unrolling

Instead of:

```c
for (i = 0; i < n; i++)
    work-unit
```

We write:

```c
for (i = 0; i < n mod 4; i++)
    work-unit
```
```c
for (i = 0; i < n; i += 4)
    work-unit
    work-unit
    work-unit
    work-unit
    work-unit
```
Goal: Handle latencies for operations.

Method: Rewrite a loop such as...

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

```c
for (...) {
    *rp++ = r0;
    r0 = a0 * b0;
    a0 = *ap++;
    b0 = *bp++;
}
```
Tool 6: Software pipelining (3)

With feed-in and wind-down:

```c
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;
```
Tool 5 + 6: Combine unrolling and software pipelining

<table>
<thead>
<tr>
<th>feed-in</th>
<th>pipelined loop</th>
<th>wind-down</th>
</tr>
</thead>
<tbody>
<tr>
<td>a0 = *ap++;</td>
<td>for (...)</td>
<td></td>
</tr>
<tr>
<td>b0 = *bp++;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a1 = *ap++;</td>
<td></td>
<td>*rp++ = r0;</td>
</tr>
<tr>
<td>b1 = *bp++;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r0 = a0 * b0;</td>
<td></td>
<td>r0 = a0 * b0;</td>
</tr>
<tr>
<td>a0 = *ap++;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b0 = *bp++;</td>
<td></td>
<td>*rp++ = r1;</td>
</tr>
<tr>
<td>a1 = *ap++;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b1 = *bp++;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r1 = a1 * b1;</td>
<td></td>
<td>r1 = a1 * b1;</td>
</tr>
<tr>
<td>a1 = *ap++;</td>
<td></td>
<td>*rp++ = r0;</td>
</tr>
<tr>
<td>b1 = *bp++;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>
Definition: Recurrency = data dependency between consecutive iterations.

In arithmetic code: Different variations of propagation of "carry".

Claim: If we use a chain of $k$ dependent operations between consuming input and generating output for a recurrency, then no CPU can perform an iteration in $< k$ cycles.
Tool 7: "Shallowing" of recurrencies (2)

Deep recurrency:

```c
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;
    }
}
```

Torbjörn Granlund (Google)
torbjorng@google.com
Nov 2014 16 / 51
Tool 7: ”Shallowing” of recurrencies (2b)

Deep recurrency:

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;
    }
}
Less deep recurrency:

```c
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;
    }
}
```
Tool 7: "Shallowing" of recurrencies (3b)

Less deep recurrence:

```c
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;
    }
}
```

Torbjörn Granlund (Google)
torbjorng@google.com
Nov 2014 19 / 51
Tool 7: "Shallowing" of recurrencies (4)

Shallow recurrency:

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;
    }
}
Shallow recurrency:

```
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;
    }
}
```
Implement in assembly!

- Find useful instructions
- Design micro-algorithms from available instructions
- Consider latency for instructions
- Which instructions can run in parallel?
- Alignment
- Trial-and-measure
- Trial-and-measure
- ...
Conditional jumps come in two categories:

1. Predictable
2. Random (or for other reasons unpredictable)

A non-predictable jump costs ≈30 plain instructions.
Intuition is good.

Measuring is better.
Intuition is good.

Measuring is better.
Optimiser tools (for arithmetic on large integers)

1. Algebra
2. Efficient algorithms
3. Algorithm selection from operand size
4. Memory and cache locality
5. Loop unrolling
6. Software pipelining
7. "Shallowing" of recurrencies
8. Assembly
9. Run, don’t jump
10. Measure it!
PART 2: Large integers in GMP
Problem: Compute \( W \leftarrow U \times V, \quad U, V \in \mathbb{Z} \)

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

Goal: Maximal real performance + lowest \{time, space\} complexity
Algorithm-1: Classic multiplication (1)

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

Time complexity: \( O(n^2) \)
Our `mul_basecase` can become really simple:

```c
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] = mul1add (w + i, u, un, v[i]);
}
```
What does `mul1add` look like? Perhaps like this:

```c
mul1add (word *w, word *u, size_t un, word vword) {
    cy_word = 0;
    for (j = 0; j < un; j++)
    {
        uword = u[i];
        lo = LO (uword * vword);
        hi = HI (uword * vword);
        wword = w[i];
        w[i] = LO (wword + lo + cy_word);
        cy_word = hi + HI (wword + lo + cy_word);
    }
    return cy_word;
}
```
Or like this (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          # 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
    addde r7, r7, r9 # add old cy_word [in]
    addze r9, r8     # new cy_word          [out]
    adddc r7, r7, r10 # add loaded (*w)
    stdu r7, 8(r3)   # +++w = result
    bdnz L1

addze r3, r9
blr
```
<table>
<thead>
<tr>
<th>( n )</th>
<th>( \text{Base} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 10^0 )</td>
<td>3 ns</td>
</tr>
<tr>
<td>( 10^1 )</td>
<td>84 ns</td>
</tr>
<tr>
<td>( 10^2 )</td>
<td>7.5 ( \mu s )</td>
</tr>
<tr>
<td>( 10^3 )</td>
<td>740 ( \mu s )</td>
</tr>
<tr>
<td>( 10^4 )</td>
<td>75 ms</td>
</tr>
<tr>
<td>( 10^5 )</td>
<td>7.5 s</td>
</tr>
<tr>
<td>( 10^6 )</td>
<td>997 s</td>
</tr>
<tr>
<td>( 10^7 )</td>
<td>1.2 days</td>
</tr>
<tr>
<td>( 10^8 )</td>
<td>220 days</td>
</tr>
<tr>
<td>( 10^9 )</td>
<td>60 years</td>
</tr>
</tbody>
</table>

(Measured on a 2.9 GHz Haswell PC, GMP 6.0.)
Base-\(\beta\) integers vs polynomials (1)

Integer:

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

Corresponding polynomial:

\[
u(x) = \sum_{i=0}^{n-1} u_i x^i
\]
Base-$\beta$ integers vs polynomials (1)

Integer:

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

 Corresponding polynomial:

$$u(x) = \sum_{i=0}^{n-1} u_i x^i$$
Consider a number in base 10

\[ 18 \ 5054 \ 0856 \ 8445 \ 1320 \ 8201 \]

and let \( \beta = 10^4 \), then we can form the polynomial

\[ 18x^5 + 5054x^4 + 0856x^3 + 8445x^2 + 1320x + 8201 \]

with the same "coefficients".
Algorithm-2: Karatsuba’s “magic formula” (1)

Give integers \(U, V < 2^{2n}\). Let \(\beta = 2^n\).

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

\[
UV = (\beta^2 + \beta) U_1 \times V_1 + \\
- \beta(U_1 - U_0) \times (V_1 - V_0) + \\
+ (\beta + 1) U_0 \times V_0
\]

Time complexity:

\[
T(n) = 3T(n/2) + O(n)
\]

\[
T(n) \in O(n^{\log 3/\log 2}) \subset O(n^{1.59})
\]
### Performance now

<table>
<thead>
<tr>
<th>$n$</th>
<th>Base</th>
<th>Kara</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^0$</td>
<td>3 ns</td>
<td>n/a</td>
</tr>
<tr>
<td>$10^1$</td>
<td>84 ns</td>
<td>115 ns (bad!)</td>
</tr>
<tr>
<td>$10^2$</td>
<td>7.5 µs</td>
<td>4.7 µs</td>
</tr>
<tr>
<td>$10^3$</td>
<td>740 µs</td>
<td>193 µs</td>
</tr>
<tr>
<td>$10^4$</td>
<td>75 ms</td>
<td>7.7 ms</td>
</tr>
<tr>
<td>$10^5$</td>
<td>7.5 s</td>
<td>0.3 s</td>
</tr>
<tr>
<td>$10^6$</td>
<td>997 s</td>
<td>11 s</td>
</tr>
<tr>
<td>$10^7$</td>
<td>1.2 days</td>
<td>7.4 min</td>
</tr>
<tr>
<td>$10^8$</td>
<td>220 days</td>
<td>4 h</td>
</tr>
<tr>
<td>$10^9$</td>
<td>60 years</td>
<td>8 days</td>
</tr>
</tbody>
</table>
Let the integer $U$ be represented by the polynomial $u(x)$, i.e., $u(\beta) = U$ for some $\beta^n$ we choose suitably. Analogously for $V, v(x)$.

Toom’s observation: We can evaluate $u(x)$ and $v(x)$ in some points $x_0, x_1 \ldots x_k$, then multiply $u(x_0)$ with $v(x_0)$, $u(x_1)$ with $v(x_1)$ etc. The product $w(x)$ is given with interpolation.

If $u(x)$ and $v(x)$ have degree $k$, then $w(x)$ will have degree $2k$, and we need $2k + 1$ eval points in order to uniquely determine the coefficients of $w(x)$.

In Toom language, Karatsuba’s algorithm has $k = 1$ and evaluates in the points $0, -1$ och $\infty$. 
Let the integer $U$ be represented by the polynomial $u(x)$, i.e., $u(\beta) = U$ for some $\beta^n$ we choose suitably. Analogously for $V$, $v(x)$.

Toom’s observation: We can evaluate $u(x)$ and $v(x)$ in some points $x_0, x_1 \ldots x_k$, then multiply $u(x_0)$ with $v(x_0)$, $u(x_1)$ with $v(x_1)$ etc. The product $w(x)$ is given with interpolation.

If $u(x)$ and $v(x)$ have degree $k$, then $w(x)$ will have degree $2k$, and we need $2k + 1$ eval points in order to uniquely determine the coefficients of $w(x)$.

In Toom language, Karatsuba’s algorithm has $k = 1$ and evaluates in the points $0, -1 \text{ och } \infty$. 
Algorithm-3: Toom’s Karatsuba generalisation (2)

Example:

Cut the operands $U$ and $V$ in 3 pieces each in order to form two degree-2 polynomials $u(x)$ and $v(x)$. We need to evaluate in $k + 1 = 5$ points, e.g., $-1, 0, +1, +2, \infty$.

Time complexity:

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

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

Cut the operands in 4 pieces and evaluate in $k + 1 = 7$ points, e.g., $-1, -1/2, 0, +1/2, +1, +2, \infty$.

Time complexity:

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

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

<table>
<thead>
<tr>
<th>$n$</th>
<th>Base</th>
<th>Kara</th>
<th>Toom 3,4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^0$</td>
<td>3 ns</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>$10^1$</td>
<td>84 ns</td>
<td>115 ns</td>
<td>n/a</td>
</tr>
<tr>
<td>$10^2$</td>
<td>7.5 µs</td>
<td>4.7 µs</td>
<td>4.6 µs</td>
</tr>
<tr>
<td>$10^3$</td>
<td>740 µs</td>
<td>193 µs</td>
<td>147 µs</td>
</tr>
<tr>
<td>$10^4$</td>
<td>75 ms</td>
<td>7.7 ms</td>
<td>4.1 ms</td>
</tr>
<tr>
<td>$10^5$</td>
<td>7.5 s</td>
<td>0.3 s</td>
<td>107 ms</td>
</tr>
<tr>
<td>$10^6$</td>
<td>997 s</td>
<td>11 s</td>
<td>2.8 s</td>
</tr>
<tr>
<td>$10^7$</td>
<td>1.2 days</td>
<td>7.4 min</td>
<td>1.17 min</td>
</tr>
<tr>
<td>$10^8$</td>
<td>220 days</td>
<td>4 h</td>
<td>30 min</td>
</tr>
<tr>
<td>$10^9$</td>
<td>60 years</td>
<td>8 days</td>
<td>12 h</td>
</tr>
</tbody>
</table>
Algorithm-4: the FFT family

FFT is an algorithm that computes certain DFTs efficiently. Input data is a degree-$2^k$ polynomial, output data is another degree-$2^k$ polynomial.

FFT needs coefficients in a ring $R$ with principal roots of unity of order $2^k$.

FFT-based integer multiplication has this structure:

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. $p'(x) \leftarrow u'(x) \cdot v'(x)$ point multiplication
4. $p(x) \leftarrow \text{FFT}^{-1}(p'(x))$
5. $P = p(\beta)$
### Performance now

<table>
<thead>
<tr>
<th>$n$</th>
<th>Base</th>
<th>Kara</th>
<th>Toom 3,4</th>
<th>SS FFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^0$</td>
<td>3 ns</td>
<td>n/a</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>$10^1$</td>
<td>84 ns</td>
<td>115 ns</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>$10^{1.5}$</td>
<td>750 ns</td>
<td><strong>661 ns</strong></td>
<td>970 ns</td>
<td></td>
</tr>
<tr>
<td>$10^2$</td>
<td>7.5 µs</td>
<td>4.7 µs</td>
<td><strong>4.6 µs</strong></td>
<td>8.3 µs</td>
</tr>
<tr>
<td>$10^3$</td>
<td>740 µs</td>
<td>193 µs</td>
<td><strong>147 µs</strong></td>
<td>187 µs</td>
</tr>
<tr>
<td>$10^4$</td>
<td>75 ms</td>
<td>7.7 ms</td>
<td>4.1 ms</td>
<td><strong>2.8 ms</strong></td>
</tr>
<tr>
<td>$10^5$</td>
<td>7.5 s</td>
<td>0.3 s</td>
<td>107 ms</td>
<td>44 ms</td>
</tr>
<tr>
<td>$10^6$</td>
<td>997 s</td>
<td>11 s</td>
<td>2.8 s</td>
<td><strong>0.58 s</strong></td>
</tr>
<tr>
<td>$10^7$</td>
<td>1.2 days</td>
<td>7.4 min</td>
<td>1.17 min</td>
<td>7.1 s</td>
</tr>
<tr>
<td>$10^8$</td>
<td>220 days</td>
<td>4 h</td>
<td>30 min</td>
<td>100 s</td>
</tr>
<tr>
<td>$10^9$</td>
<td>60 years</td>
<td>8 days</td>
<td>12 h</td>
<td><strong>20 min</strong></td>
</tr>
</tbody>
</table>
We have assumed \( U \) and \( V \) are of the same size, and then mapped these to polynomials of the same degree \( k \).

What if operands are of different size? Pad with zeros?
The problem:
Split in squares recursively = same size operands:
Adaption of algorithms to different size operands

- Plain $O(n^2)$ algorithm trivially works
- Karatsuba-Toom not as obvious
- Karatsuba-Toom-Bodrato-Zanoni found solution (2006)
- FFT ”simple” (but at some cost)
In 2006 M. Bodrato and A. Zanoni generalised Toom’s algorithm, suggesting the use of polynomials $u(x)$, $v(x)$ of different degrees $k_u$ and $k_v$.

This is useful for multiplication of different-size operands.

Example: If the size of $U$ and $V$ relates as 3:2, we may map $U$ to a degree-2 polynomial $u(x)$, and $V$ to a degree-1 polynomial $v(x)$. We need to evaluate in 4 points. (Why 4?)
Toom-Bodrato-Zanoni primitives in GMP

<table>
<thead>
<tr>
<th>deg($u$)</th>
<th>deg($v$)</th>
<th>k</th>
<th>points</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>$-1, 0, \infty$</td>
<td>toom22_mul</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>4</td>
<td>$-1, 0, +1, \infty$</td>
<td>toom32_mul</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>5</td>
<td>$-1, 0, +1, +2, \infty$</td>
<td>toom33_mul</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>5</td>
<td>$-1, 0, +1, +2, \infty$</td>
<td>toom42_mul</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>6</td>
<td>$-2, -1, 0, +1, +2, \infty$</td>
<td>toom43_mul</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>6</td>
<td>$-2, -1, 0, +1, +2, \infty$</td>
<td>toom52_mul</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>7</td>
<td>$-2, -1, 0, +1/2, +1, +2, \infty$</td>
<td>toom44_mul</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>7</td>
<td>$-2, -1, 0, +1/2, +1, +2, \infty$</td>
<td>toom53_mul</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>7</td>
<td>$-2, -1, 0, +1/2, +1, +2, \infty$</td>
<td>toom62_mul</td>
</tr>
</tbody>
</table>
Questions?