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 $O(n^3)$)

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

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

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

Then perform the two exponentiations:

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

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 $O(n^3)$)

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

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

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

Then perform the two exponentiations:

$$\begin{aligned} s_p &= (m \bmod p)^{d_p} \bmod p \\ s_q &= (m \bmod q)^{d_q} \bmod q \end{aligned}$$

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 $O(n^3)$)

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

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

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

Then perform the two exponentiations:

$$\begin{aligned} s_p &= (m \bmod p)^{d_p} \bmod p \\ s_q &= (m \bmod q)^{d_q} \bmod q \end{aligned}$$

We then get s through CRT from s_p and s_q (in time $O(n^2)$).

Tool 2: Efficient algorithms

Example:
Karatsuba's divide-and-conquer algorithm for multiplication.

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

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

Tool 3: Algorithm selection from operand size (1)

Naive Karatsuba implementation:

```
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);
}
```

Tool 3: Algorithm selection from operand size (2)

Cleverer 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);
}
```

Tool 3: Algorithm selection from operand size (3)

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:

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

We write:

```
for (i = 0; i < n mod 4; i++)
  work-unit
for (i = 0; i < n; i += 4)
  work-unit
  work-unit
  work-unit
  work-unit
```

Tool 6: Software pipelining (1)

Goal: Handle latencies for operations.

Method: Rewrite a loop such as. . .

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

Tool 6: Software pipelining (2)

. . . into:

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

Tool 6: Software pipelining (3)

With feed-in and 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;
```

Tool 5 + 6: Combine unrolling and 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++;       |        }                    |
```

Tool 7: "Shallowing" of recurrencies (1)

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:

```
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;
    }
}
```

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;
    }
}
```

Tool 7: "Shallowing" of recurrencies (3)

Less 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 + vword;
      cy0 = sum0 < uword;
      sum1 = sum0 + cy;
      cy1 = sum1 < sum0;
      cy = cy0 + cy1;
      r[i] = sum1;
    }
}
```

Tool 7: "Shallowing" of recurrencies (3b)

Less 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 + vword;
      cy0 = sum0 < uword;
      sum1 = sum0 + cy;                0       3       6
      cy1 = sum1 < sum0;               1       4      ...
      cy = cy0 + cy1;                  2       5
      r[i] = sum1;
    }
}
```

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;
    }
}
```

Tool 7: "Shallowing" of recurrencies (4b)

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;                     0       2       4
      cy1 = cy & (sum0 == ~0);              0       2      ...
      cy = cy0 + cy1;                       1       3
      r[i] = sum1;
    }
}
```

Tool 8: Assembly

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
- ...

Tool 9: Run, don't jump

Conditional jumps come in two categories:

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

A non-predictable jump costs ≈30 plain instructions.

Tool 10: Measure it!

Intuition is good.

Measuring is better.

Tool 10: Measure it!

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

Integer multiplication

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)

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

Time complexity: $O(n^2)$

Our `mul_basecase` can become really simple:

```
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:

```
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
    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
```

Or a bit more complex (Alpha)...

```
        bge     r18,    $ent1
        ret     r31,    (r26),  1
$L1:    lda     r8,     0(r31)
        lda     r24,    0(r31)
        cmpeq   r20,    2,      r21
        bne     r21,    $2mod8
        cmpeq   r20,    3,      r21
        bne     r21,    $3mod8
        cmpeq   r20,    4,      r21
        bne     r21,    $4mod8
        cmpeq   r20,    5,      r21
        bne     r21,    $5mod8
        cmpeq   r20,    6,      r21
        bne     r21,    $6mod8
        cmpeq   r20,    7,      r21
        beq     r21,    $0mod8
$7mod8: ldq     r5,     0(r16)
        lda     r17,    8(r17)
        mulq    r19,    r3,     r7
        umulh   r19,    r3,     r24
        addq    r5,     r7,     r23
        cmpult  r23,    r7,     r20
        addq    r24,    r20,    r24
        stq     r23,    0(r16)
        lda     r16,    8(r16)
        ldq     r3,     0(r17)
$6mod8: ldq     r1,     8(r17)
        mulq    r19,    r3,     r25
        umulh   r19,    r3,     r3
        mulq    r19,    r1,     r28
        ldq     r0,     16(r17)
        ldq     r4,     0(r16)
        umulh   r19,    r1,     r8
        ldq     r1,     24(r17)
        lda     r17,    48(r17)
        mulq    r19,    r0,     r2
        ldq     r5,     8(r16)
        lda     r16,    -32(r16)
        umulh   r19,    r0,     r6
        addq    r4,     r25,    r4
        mulq    r19,    r1,     r7
        br      r31,    $ent6
$ent1:  lda     r17,    8(r17)
        lda     r16,    8(r16)
        lda     r8,     0(r0)
        ldq     r3,     0(r17)
$0mod8: ldq     r1,     8(r17)
```

```
        lda     r16,    -16(r16)
        br      r31,    $ent0
$3mod8: ldq     r5,     0(r16)
        lda     r17,    8(r17)
        mulq    r19,    r3,     r7
        umulh   r19,    r3,     r8
        addq    r5,     r7,     r23
        cmpult  r23,    r7,     r20
        addq    r8,     r20,    r24
        stq     r23,    0(r16)
        lda     r16,    8(r16)
        ldq     r3,     0(r17)
$2mod8: ldq     r1,     8(r17)
        mulq    r19,    r3,     r25
        umulh   r19,    r3,     r3
        mulq    r19,    r1,     r28
        ble     r18,    $n23
        ldq     r0,     16(r17)
        ldq     r4,     0(r16)
        umulh   r19,    r1,     r8
        ldq     r1,     24(r17)
        lda     r17,    16(r17)
        mulq    r19,    r0,     r2
        ldq     r5,     8(r16)
        lda     r16,    0(r16)
        umulh   r19,    r0,     r6
        addq    r4,     r25,    r4
        mulq    r19,    r1,     r7
        br      r31,    $ent2
$5mod8: ldq     r5,     0(r16)
        lda     r17,    8(r17)
        mulq    r19,    r3,     r7
        umulh   r19,    r3,     r24
        addq    r5,     r7,     r23
        cmpult  r23,    r7,     r20
        addq    r24,    r20,    r8
        stq     r23,    0(r16)
        lda     r16,    8(r16)
        ldq     r3,     0(r17)
$4mod8: ldq     r1,     8(r17)
        mulq    r19,    r3,     r2
        umulh   r19,    r3,     r6
        mulq    r19,    r1,     r7
        ldq     r0,     16(r17)
        ldq     r4,     0(r16)
        umulh   r19,    r1,     r24
        ldq     r1,     24(r17)
```

```
        bis     r31,    r31,    r31
        cmpult  r22,    r8,     r21
        addq    r6,     r20,    r6
        ldq     r0,     0(r17)
        bis     r31,    r31,    r31
        addq    r5,     r7,     r23
        addq    r6,     r21,    r6
        ldq     r4,     0(r16)
        umulh   r19,    r1,     r8
        cmpult  r23,    r7,     r20
        addq    r23,    r6,     r23
        ldq     r1,     8(r17)
        mulq    r19,    r0,     r2
        cmpult  r23,    r6,     r21
        addq    r24,    r20,    r24
        ldq     r5,     8(r16)
        umulh   r19,    r0,     r6
        addq    r4,     r25,    r4
        stq     r22,    -16(r16)
        stq     r23,    -8(r16)
        bis     r31,    r31,    r31
        mulq    r19,    r1,     r7
        bis     r31,    r31,    r31
        addq    r24,    r21,    r24
$ent2:
        cmpult  r4,     r25,    r20
        bis     r31,    r31,    r31
        lda     r18,    -8(r18)
        addq    r4,     r24,    r22
        bis     r31,    r31,    r31
        cmpult  r22,    r24,    r21
        addq    r3,     r20,    r3
        ldq     r0,     16(r17)
        bis     r31,    r31,    r31
        addq    r5,     r28,    r23
        addq    r3,     r21,    r3
        ldq     r4,     16(r16)
        umulh   r19,    r1,     r24
        cmpult  r23,    r28,    r20
        addq    r23,    r3,     r23
        ldq     r1,     24(r17)
        mulq    r19,    r0,     r25
        cmpult  r23,    r3,     r21
        addq    r8,     r20,    r8
        ldq     r5,     24(r16)
        umulh   r19,    r0,     r3
        addq    r4,     r2,     r4
```

```
        cmpult  r22,    r8,     r21
        addq    r6,     r20,    r6
        ldq     r0,     -32(r17)
        bis     r31,    r31,    r31
        addq    r5,     r7,     r23
        addq    r6,     r21,    r6
        ldq     r4,     32(r16)
        umulh   r19,    r1,     r8
        cmpult  r23,    r7,     r20
        addq    r23,    r6,     r23
        ldq     r1,     -24(r17)
        mulq    r19,    r0,     r2
        cmpult  r23,    r6,     r21
        addq    r24,    r20,    r24
        ldq     r5,     40(r16)
        umulh   r19,    r0,     r6
        addq    r4,     r25,    r4
        stq     r22,    16(r16)
        stq     r23,    24(r16)
        bis     r31,    r31,    r31
        mulq    r19,    r1,     r7
        bis     r31,    r31,    r31
        addq    r24,    r21,    r24
$ent6:
        cmpult  r4,     r25,    r20
        bis     r31,    r31,    r31
        lda     r16,    64(r16)
        addq    r4,     r24,    r22
        bis     r31,    r31,    r31
        cmpult  r22,    r24,    r21
        addq    r3,     r20,    r3
        ldq     r0,     -16(r17)
        bis     r31,    r31,    r31
        addq    r5,     r28,    r23
        addq    r3,     r21,    r3
        ldq     r4,     -16(r16)
        umulh   r19,    r1,     r24
        cmpult  r23,    r28,    r20
        addq    r23,    r3,     r23
        ldq     r1,     -8(r17)
        mulq    r19,    r0,     r25
        cmpult  r23,    r3,     r21
        addq    r8,     r20,    r8
        ldq     r5,     -8(r16)
        umulh   r19,    r0,     r3
        addq    r4,     r2,     r4
        stq     r22,    -32(r16)
```

```
        addq    r6,     r21,    r6
        ldq     r4,     0(r16)
        umulh   r19,    r1,     r8
        cmpult  r23,    r7,     r20
        addq    r23,    r6,     r23
        cmpult  r23,    r6,     r21
        addq    r24,    r20,    r24
        ldq     r5,     8(r16)
        addq    r4,     r25,    r4
        stq     r22,    -16(r16)
        stq     r23,    -8(r16)
        addq    r24,    r21,    r24
        cmpult  r4,     r25,    r20
        addq    r4,     r24,    r22
        cmpult  r22,    r24,    r21
        addq    r3,     r20,    r3
        addq    r5,     r28,    r23
        addq    r3,     r21,    r3
        cmpult  r23,    r28,    r20
        addq    r23,    r3,     r23
        cmpult  r23,    r3,     r21
        addq    r8,     r20,    r8
        stq     r22,    0(r16)
        stq     r23,    8(r16)
        addq    r8,     r21,    r0
        ret     r31,    (r26),  1
$n23:   ldq     r4,     0(r16)
        ldq     r5,     8(r16)
        umulh   r19,    r1,     r8
        addq    r4,     r25,    r4
        cmpult  r4,     r25,    r20
        addq    r4,     r24,    r22
        cmpult  r22,    r24,    r21
        addq    r3,     r20,    r3
        addq    r5,     r28,    r23
        addq    r3,     r21,    r3
        cmpult  r23,    r28,    r20
        addq    r23,    r3,     r23
        cmpult  r23,    r3,     r21
        addq    r8,     r20,    r8
        stq     r22,    0(r16)
        stq     r23,    8(r16)
        addq    r8,     r21,    r0
        ret     r31,    (r26),  1
```

Performance now

n	Base
10^0	3 ns
10^1	84 ns
10^2	7.5 μs
10^3	740 μs
10^4	75 ms
10^5	7.5 s
10^6	997 s
10^7	1.2 days
10^8	220 days
10^9	60 years

(Measured on a 2.9 GHz Haswell PC, GMP 6.0.)

Base-β 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-β 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-β integers vs polynomials (2)

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 $\beta = 2^n$.

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

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

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

n	Base	Kara
10^0	3 ns	n/a
10^1	84 ns	115 ns (bad!)
10^2	7.5 μs	4.7 μs
10^3	740 μs	193 μs
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 days	7.4 min
10^8	220 days	4 h
10^9	60 years	8 days

Algorithm-3: Toom's Karatsuba generalisation (1)

Let the integer U be represented by the polynomial $u(x)$, i.e., $u(\beta) = U$ for some β^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 ∞.

Algorithm-3: Toom's Karatsuba generalisation (1)

Let the integer U be represented by the polynomial $u(x)$, i.e., $u(\beta) = U$ for some β^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 ∞.

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:

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

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

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:

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

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

Performance now

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 μs	4.7 μs	4.6 μs
10^3	740 μs	193 μs	147 μs
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 days	7.4 min	1.17 min
10^8	220 days	4 h	30 min
10^9	60 years	8 days	12 h

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

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 μs	4.7 μs	**4.6 μs**	8.3 μs
10^3	740 μs	193 μs	**147 μs**	187 μs
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 days	7.4 min	1.17 min	**7.1 s**
10^8	220 days	4 h	30 min	**100 s**
10^9	60 years	8 days	12 h	**20 min**

Same size vs different size operands (1)

We have assumed U and V are of the same size, and then mapped these to polynomials of the same degree k.

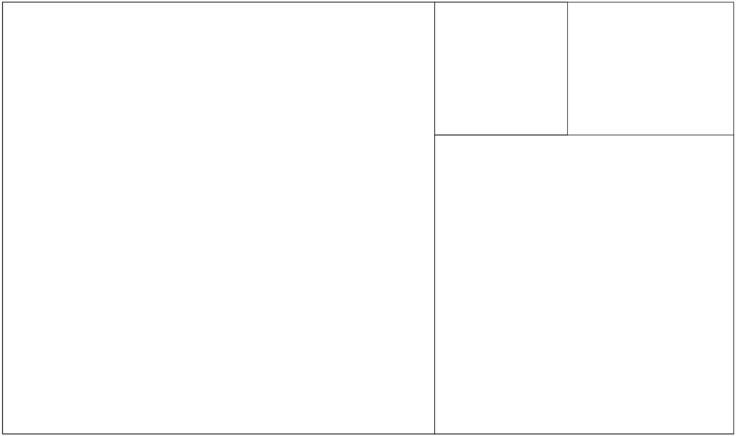
What if operands are of different size? Pad with zeros?

Same size vs different size operands (2)

The problem:

Same size vs different size operands (3)

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)

The Bodrato-Zanoni Toom generalisation

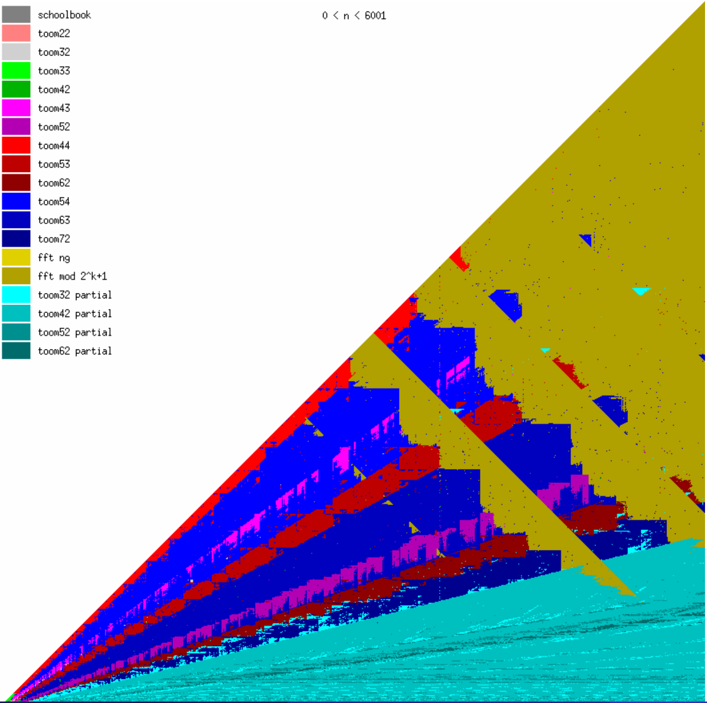
In 2006 M.Bodrato and A.Zanoni generalised Toom's algorithm, suggesting the use of polynomials $u(x)$, $v(x)$ of **different** degrees k_u and k_v.

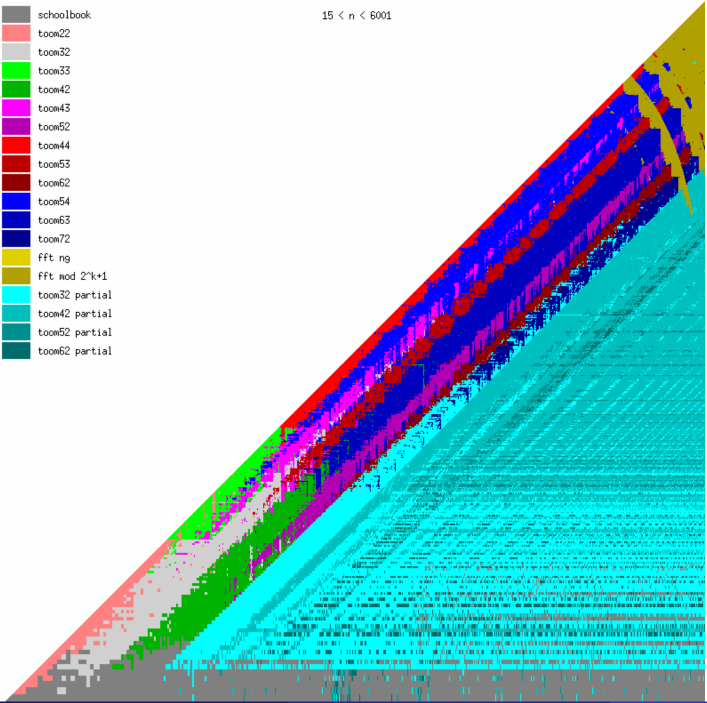
This is useful for multiplication of *different-size* operands.

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

deg(u)	deg(v)	k	points	name
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


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
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

The End

Questions?