Integer multiplication and the truncated product problem

David Harvey

Macquarie University Mathematics Colloquium, 5th October 2018

University of New South Wales

Topics for this talk

- Integer multiplication: history and state of the art

Topics for this talk

- Integer multiplication: history and state of the art
- Truncated products: a new algorithm and an open problem

Topics for this talk

- Integer multiplication: history and state of the art
- Truncated products: a new algorithm and an open problem
- My recent ski trip

Integer multiplication

$\mathsf{M}(n) :=$ complexity of multiplying n-bit integers.

Complexity model: any reasonable notion of counting "bit operations", e.g. multitape Turing machine or Boolean circuits.

The exponential-time algorithm

Complexity: $\mathrm{M}(n) = 2^{O(n)}$.

The exponential-time algorithm

Complexity: $\mathrm{M}(n) = 2^{O(n)}$.

$314 \times 271 =$

The exponential-time algorithm

Complexity: $\mathrm{M}(n) = 2^{O(n)}$.

$$314 \times 271 = 271 +$$

The exponential-time algorithm

Complexity: $\mathrm{M}(n) = 2^{O(n)}$.

$$314 \times 271 = 271 + 271 +$$

The exponential-time algorithm

Complexity: $\mathrm{M}(n) = 2^{O(n)}$.

$$314 \times 271 = 271 + 271 + 271 + \cdots$$

The exponential-time algorithm

Complexity: $\mathrm{M}(n) = 2^{O(n)}$.

$$314 \times 271 = 271 + 271 + 271 + \cdots$$
$$\cdots + 271 + 271 = 85094.$$

The exponential-time algorithm

Complexity: $\mathrm{M}(n) = 2^{O(n)}$.

$$314 \times 271 = 271 + 271 + 271 + \cdots$$
$$\cdots + 271 + 271 = 85094.$$

Jesse (age 6)

The exponential-time algorithm

Complexity: $\mathrm{M}(n) = 2^{O(n)}$.

$$314 \times 271 = 271 + 271 + 271 + \cdots$$
$$\cdots + 271 + 271 = 85094.$$

Jesse (age 6)

Conclusion: skiing is hard work if you use the wrong algorithm.

The classical algorithm

Complexity: $\mathrm{M}(n) = O(n^2)$.

Known to ancient Egyptians no later than 2000 BCE, probably much older.

The classical algorithm

Complexity: $\mathrm{M}(n) = O(n^2)$.

Known to ancient Egyptians no later than 2000 BCE, probably much older.

$$
\begin{array}{rl}
 & 314 \\
 & 271 \quad \times \\
\hline
 & 314 \\
2198 & \\
628 & \\
\hline
85094 &
\end{array}
$$

The classical algorithm

Complexity: $\mathrm{M}(n) = O(n^2)$.

Known to ancient Egyptians no later than 2000 BCE, probably much older.

$$\begin{array}{r} 314 \\ 271 \quad \times \\ \hline 314 \\ 2198 \\ 628 \\ \hline 85094 \end{array}$$

Zachary (age 8)

Kolmogorov's conjecture

Around 1956, Kolmogorov conjectured the lower bound:

$$\mathrm{M}(n) = \Omega(n^2).$$

Kolmogorov

Kolmogorov's conjecture

Around 1956, Kolmogorov conjectured the lower bound:

$$\mathrm{M}(n) = \Omega(n^2).$$

Kolmogorov

The appearance of this conjecture is probably

> *"based on the fact that throughout the history of mankind people have been using [the algorithm] whose complexity is $O(n^2)$, and if a more economical method existed, it would have already been found."*

— Karatsuba, 1995

Karatsuba's algorithm

In 1960, Kolmogorov organised a seminar on cybernetics at Moscow University, in which he stated his conjecture.

Karatsuba's algorithm

In 1960, Kolmogorov organised a seminar on cybernetics at Moscow University, in which he stated his conjecture.

Within a week, Karatsuba, a 23-year old student in the audience, discovered his famous subquadratic algorithm. He proved that

$$\mathsf{M}(n) = O(n^{\alpha}), \qquad \alpha = \frac{\log 3}{\log 2} \approx 1.58.$$

Karatsuba
(age > 23)

Karatsuba's algorithm

When Karatsuba told Kolmogorov of his discovery,

> *"Kolmogorov was very agitated because this contradicted his very plausible conjecture. At the next meeting of the seminar, Kolmogorov himself told the participants about my method,*
>
> — Karatsuba, 1995

Karatsuba's algorithm

When Karatsuba told Kolmogorov of his discovery,

> *"Kolmogorov was very agitated because this contradicted his very plausible conjecture. At the next meeting of the seminar, Kolmogorov himself told the participants about my method, and at this point the seminar was terminated."*
>
> — Karatsuba, 1995

Improvements to Karatsuba

Lots of action in the 1960's (Toom, Cook, Schönhage, Knuth), generalising and optimising Karatsuba's algorithm.

It was quickly realised that one could achieve

$$\mathsf{M}(n) = O(n^{1+\epsilon}), \qquad \text{any } \epsilon > 0.$$

Improvements to Karatsuba

Lots of action in the 1960's (Toom, Cook, Schönhage, Knuth), generalising and optimising Karatsuba's algorithm.

It was quickly realised that one could achieve

$$\mathrm{M}(n) = O(n^{1+\epsilon}), \qquad \text{any } \epsilon > 0.$$

Final result along these lines:

$$\mathrm{M}(n) = O(n\, 2^{\sqrt{2 \log n / \log 2}} \log n)$$

(given as an exercise in first edition of The Art of Computer Programming, vol. 2 "Seminumerical algorithms", Knuth 1969)

The Fast Fourier Transform

1965: introduction of FFT by Cooley–Tukey.

Problem: given polynomial $P(x) \in \mathbf{C}[x]$ of degree $< d$, want to compute values of $P(x)$ at complex d-th roots of unity.

The Fast Fourier Transform

1965: introduction of FFT by Cooley–Tukey.

Problem: given polynomial $P(x) \in \mathbf{C}[x]$ of degree $< d$, want to compute values of $P(x)$ at complex d-th roots of unity.

Naive algorithm requires $O(d^2)$ operations in $\mathbf{C}$.

(Operation = addition, subtraction, or multiplication in $\mathbf{C}$.)

FFT requires only $O(d \log d)$ operations.

The Fast Fourier Transform

1965: introduction of FFT by Cooley–Tukey.

Problem: given polynomial $P(x) \in \mathbf{C}[x]$ of degree $< d$, want to compute values of $P(x)$ at complex d-th roots of unity.

Naive algorithm requires $O(d^2)$ operations in $\mathbf{C}$.

(Operation = addition, subtraction, or multiplication in $\mathbf{C}$.)

FFT requires only $O(d \log d)$ operations.

(Gauss discovered the Cooley–Tukey algorithm around 1805, not published in his lifetime. He did not give a general complexity analysis.)

Schönhage–Strassen

The FFT was first applied to integer multiplication by Schönhage and Strassen in 1971.

Schönhage–Strassen

The FFT was first applied to integer multiplication by Schönhage and Strassen in 1971.

Actually they gave two algorithms:

- A fairly simple algorithm that I will explain some detail.
- A less obvious but more famous algorithm achieving

$$\mathsf{M}(n) = O(n \log n \log\log n),$$

  which was the champion for over 35 years.

Schönhage–Strassen

The FFT was first applied to integer multiplication by Schönhage and Strassen in 1971.

Actually they gave two algorithms:

- A fairly simple algorithm that I will explain some detail.
- A less obvious but more famous algorithm achieving

$$\mathsf{M}(n) = O(n \log n \log\log n),$$

  which was the champion for over 35 years.

They also suggested (but did not quite conjecture) that the right bound is $\mathsf{M}(n) = O(n \log n)$. This is still an open problem.

First Schönhage–Strassen algorithm

Input: positive n-bit integers u and v.

First Schönhage–Strassen algorithm

Input: positive n-bit integers u and v.

Choose base $B = 2^b$ where say $b \approx \log n$ (or perhaps $(\log n)^2$).

Cut up inputs into chunks of b bits, i.e., write u and v in base B.

Encode into polynomials $U(x), V(x) \in \mathbf{Z}[x]$, say degree $< d$, so that $U(B) = u$ and $V(B) = v$.

First Schönhage–Strassen algorithm

Input: positive n-bit integers u and v.

Choose base $B = 2^b$ where say $b \approx \log n$ (or perhaps $(\log n)^2$).

Cut up inputs into chunks of b bits, i.e., write u and v in base B.

Encode into polynomials $U(x), V(x) \in \mathbf{Z}[x]$, say degree $< d$, so that $U(B) = u$ and $V(B) = v$.

Baby example in base 10: $u = 314159265358$, $v = 271828182845$.

Take $B = 1000$, $d = 4$, so

$$U(x) = 314x^3 + 159x^2 + 265x + 358,$$
$$V(x) = 271x^3 + 828x^2 + 182x + 845.$$

First Schönhage–Strassen algorithm

It's enough to compute the polynomial product in $\mathbf{Z}[x]$:

$$UV(x) = 85094x^6 + 303081x^5 + 260615x^4 + 610706x^3 + 479009x^2 + 289081x + 302510.$$

Then evaluate at $x = B$ to get $uv = U(B)V(B) = UV(B)$:

$$\begin{array}{r} 85094B^6 \\ 303081B^5 \\ 260615B^4 \\ 610706B^3 \\ 479009B^2 \\ 289081B^1 \\ 302510B^0 \\ \hline uv = 8539734222618529838351 0 \end{array}$$

First Schönhage–Strassen algorithm

How to compute the polynomial product $U(x)V(x)$?

Standard evaluate-multiply-interpolate paradigm:

First Schönhage–Strassen algorithm

How to compute the polynomial product $U(x)V(x)$?

Standard evaluate-multiply-interpolate paradigm:

(1) Use FFT to (approximately) evaluate at $2d$-th roots of unity:

$$U(1.00000000 + 0.00000000i) = 1096.0000,$$
$$U(0.70710678 + 0.70710678i) = 323.35177 + 568.41483i,$$
$$\vdots$$

$$V(1.00000000 + 0.00000000i) = 2126.0000,$$
$$V(0.70710678 + 0.70710678i) = 782.06750 + 1148.3194i,$$
$$\vdots$$

First Schönhage–Strassen algorithm

(2) Multiply pointwise to get values of UV at $2d$-th roots of unity:

$$\begin{aligned} UV(1.00000000 + 0.00000000i) &= 1096.0000 \times 2126.0000 \\ &= 2330096.0, \end{aligned}$$

$$\begin{aligned} UV(0.70710678 + 0.70710678i) &= (323.35177 + 568.41483i) \times (782.06750 + 1148.3194i) \\ &= -399838.85 + 815849.86i, \end{aligned}$$

$$\vdots$$

First Schönhage–Strassen algorithm

(3) Since $\deg UV < 2d$, can use inverse FFT to recover approximate coefficients of UV:

$$UV(x) = 85093.9998x^6 + (303081.00 - 0.0002i)x^5 + \cdots$$

First Schönhage–Strassen algorithm

(3) Since $\deg UV < 2d$, can use inverse FFT to recover approximate coefficients of UV:

$$UV(x) = 85093.9998x^6 + (303081.00 - 0.0002i)x^5 + \cdots$$

Assuming we maintain sufficient precision during calculations ($O(\log n)$ bits is enough), we may round to nearest integer:

$$UV(x) = 85094x^6 + 303081x^5 + 260615x^4 + 610706x^3 + 479009x^2 + 289081x + 302510.$$

First Schönhage–Strassen algorithm

During the algorithm, we performed many multiplications in $\mathbf{C}$:

- during the FFTs (multiplications by roots of unity), and
- the pointwise multiplications.

These are handled by converting back to *integer* multiplication.

First Schönhage–Strassen algorithm

During the algorithm, we performed many multiplications in $\mathbf{C}$:

- during the FFTs (multiplications by roots of unity), and
- the pointwise multiplications.

These are handled by converting back to *integer* multiplication.

Example: to compute

$$(323.35177 + 568.41483i) \times (782.06750 + 1148.3194i),$$

we (recursively) compute the integer products

$$32335177 \times 78206750, \qquad 32335177 \times 11483194,$$
$$56841483 \times 78206750, \qquad 56841483 \times 11483194,$$

and then scale and add/subtract appropriately.

First Schönhage–Strassen algorithm

Complexity analysis: we reduced an integer product of size n to $O(d \log d) = O(n)$ integer products of size $O(\log n)$.

In other words

$$\mathrm{M}(n) < Cn\,\mathrm{M}(\log n) \qquad \text{for some constant } C > 0.$$

First Schönhage–Strassen algorithm

Complexity analysis: we reduced an integer product of size n to $O(d \log d) = O(n)$ integer products of size $O(\log n)$.

In other words

$$\mathrm{M}(n) < Cn\,\mathrm{M}(\log n) \qquad \text{for some constant } C > 0.$$

Unrolling the recursion:

$$\begin{aligned} \mathrm{M}(n) &< C^2 n \log n\, \mathrm{M}(\log\log n) \\ &\cdots \\ &< C^{\log^* n} n \log n \log\log n \cdots \log^{(\log^* n)} n. \end{aligned}$$

First Schönhage–Strassen algorithm

Pollard's alternative: replace coefficient ring $\mathbf{C}$ by $\mathbf{F}_p$.

Choose $p = 1 \pmod{2^k}$ where 2^k is the desired transform length, so $\mathbf{F}_p$ contains appropriate roots of unity.

First Schönhage–Strassen algorithm

Pollard's alternative: replace coefficient ring $\mathbf{C}$ by $\mathbf{F}_p$.

Choose $p = 1 \pmod{2^k}$ where 2^k is the desired transform length, so $\mathbf{F}_p$ contains appropriate roots of unity.

Even better: use $\mathbf{F}_{p_1} \times \cdots \times \mathbf{F}_{p_r}$ plus Chinese remainder theorem.

First Schönhage–Strassen algorithm

Pollard's alternative: replace coefficient ring $\mathbf{C}$ by $\mathbf{F}_p$.

Choose $p = 1 \pmod{2^k}$ where 2^k is the desired transform length, so $\mathbf{F}_p$ contains appropriate roots of unity.

Even better: use $\mathbf{F}_{p_1} \times \cdots \times \mathbf{F}_{p_r}$ plus Chinese remainder theorem.

Examples in real life (using word-sized primes):

- Victor Shoup's NTL library
- My own integer multiplication code (used for "average polynomial time" zeta function computations)

Second Schönhage–Strassen algorithm

Replace $\mathbf{C}$ by the ring $\mathbf{Z}/F_k\mathbf{Z}$ where $F_k = 2^{2^k} + 1$ for $2^k \approx \sqrt{n}$.

The element 2 plays the role of a "fast" 2^{k+1}-th root of unity.

Second Schönhage–Strassen algorithm

Replace $\mathbf{C}$ by the ring $\mathbf{Z}/F_k\mathbf{Z}$ where $F_k = 2^{2^k} + 1$ for $2^k \approx \sqrt{n}$.

The element 2 plays the role of a "fast" 2^{k+1}-th root of unity.

My wife (age < 100)
skiing very fast

This algorithm achieves

$$\mathrm{M}(n) = O(n \log n \log\log n).$$

Second Schönhage–Strassen algorithm

This is essentially the algorithm implemented in GMP right now (with heavy optimisations).

You are using this code whenever you multiply large integers in Magma, Sage, Mathematica, Maple.

Second Schönhage–Strassen algorithm

This is essentially the algorithm implemented in GMP right now (with heavy optimisations).

You are using this code whenever you multiply large integers in Magma, Sage, Mathematica, Maple.

```
sage: u = ZZ.random_element(10^(10^9))
sage: v = ZZ.random_element(10^(10^9))
sage: time w = u*v
Wall time: 25.8 s
```

Fürer's breakthrough

Fürer (2007) suggested using the coefficient ring

$$\mathbf{C}[y]/(y^{2^m} + 1)$$

where $2^m \approx \log n$, with precision about $\log n$ bits.

This ring combines advantages of both Schönhage–Strassen algorithms:

- it contains "fast" roots of unity of order 2^{m+1}
- it also inherits high-order roots of unity from $\mathbf{C}$.

He uses the "fast" roots as often as possible, only uses "slow" roots when necessary.

Fürer's breakthrough

Fürer achieves the bound

$$\mathrm{M}(n) = O(n \log n \, K^{\log^* n})$$

for some unspecified constant $K > 1$.

The constant K measures the "expansion factor" at each level.

(An optimised version achieves $K = 16$.)

Fürer's breakthrough

Fürer achieves the bound

$$\mathrm{M}(n) = O(n \log n \, K^{\log^* n})$$

for some unspecified constant $K > 1$.

The constant K measures the "expansion factor" at each level.

(An optimised version achieves $K = 16$.)

The function $K^{\log^* n}$ grows *much* more slowly than $\log \log n$.

For example, if $n = 10^{10^{10^{10}}}$, then

$$16^{\log^* n} = 16^5, \qquad \log \log n = 10^{10^{10}}.$$

Fast roots are unnecessary

H.–van der Hoeven–Lecerf (2014) showed how to get the same bound without using "fast" roots of unity.

The algorithm works directly over $\mathbf{C}$, and achieves $K = 8$.

Fast roots are unnecessary

H.–van der Hoeven–Lecerf (2014) showed how to get the same bound without using "fast" roots of unity.

The algorithm works directly over $\mathbf{C}$, and achieves $K = 8$.

[Aside: one advantage of our approach is that it can be adapted to multiplication in $\mathbf{F}_p[x]$.

For fixed p, we can multiply polynomials of degree n using

$$O(n \log n \, 8^{\log^* n})$$

operations in $\mathbf{F}_p$.

It is not known how to achieve this using Fürer's method.]

Why $K = 8$?

Three factors of 2 from different sources:

(A) *FFT multiplication*. Need to recurse into both forward and inverse DFTs.

(B) *Coefficient growth.* If f and g have integer coefficients with k bits, then the coefficients of fg have roughly $2k$ bits.

(C) *Truncated product problem.* The algorithm works over $\mathbf{C}$. When multiplying complex numbers with k-bit mantissa, need to compute product with $2k$ bits and then truncate.

Seems very hard to do anything about (A) or (B).

The rest of the talk will focus on (C).

The truncated product problem

Here is the crux of the problem.

Suppose I want to compute $0.314159265358 \times 0.271828182845$.

The truncated product problem

Here is the crux of the problem.

Suppose I want to compute $0.314159265358 \times 0.271828182845$.

Converting to integer multiplication, I get the product

$$314159265358 \times 271828182845 = 85397342226185298383510.$$

The truncated product problem

Here is the crux of the problem.

Suppose I want to compute $0.314159265358 \times 0.271828182845$.

Converting to integer multiplication, I get the product

$$314159265358 \times 271828182845 = 85397342226185298383510.$$

But I really only want about 12 significant digits, so I would be happy with the answer

$$314159265358 \times 271828182845 \approx 85397342226000000000000,$$

which is equivalent to

$$0.314159265358 \times 0.271828182845 \approx 0.085397342226.$$

In other words, I only want the **top half** of the integer product.

The truncated product problem

Recall that we converted integer multiplication to polynomial multiplication:

$$\begin{aligned}(314x^3 + 159x^2 + 265x + 358) \times (271x^3 + 828x^2 + 182x + 845) \\ = 85094x^6 + 303081x^5 + 260615x^4 + 610706x^3 \\ + 479009x^2 + 289081x + 302510.\end{aligned}$$

We only want the "top half" of this polynomial.

The truncated product problem

Recall that we converted integer multiplication to polynomial multiplication:

$$\begin{aligned}(314x^3 + 159x^2 + 265x + 358) \times (271x^3 + 828x^2 + 182x + 845) \\ = 85094x^6 + 303081x^5 + 260615x^4 + 610706x^3 \\ + 479009x^2 + 289081x + 302510.\end{aligned}$$

We only want the "top half" of this polynomial.

But this is not what the FFT method computes!

The FFT actually computes the product modulo $x^8 - 1$.

We could compute a product modulo $x^4 - 1$ but this doesn't help.

The truncated product problem

Last year I proved that (under certain conditions) *one can compute a truncated product in* $3/4$ *of the time of the full product.*

This is the first known constant-factor savings for any type of truncated product problem.

The truncated product problem

Last year I proved that (under certain conditions) *one can compute a truncated product in* $3/4$ *of the time of the full product.*

This is the first known constant-factor savings for any type of truncated product problem.

Corollary: for integer multiplication, can improve $K = 8$ to $K = 6$.

The cancellation trick

Consider again the polynomial product

$$\begin{aligned}(314x^3 + 159x^2 + 265x + 358) \times (271x^3 + 828x^2 + 182x + 845) \\ = 85094x^6 + 303081x^5 + 260615x^4 + 610706x^3 \\ + 479009x^2 + 289081x + 302510.\end{aligned}$$

I said before that we only want the "top half" of this polynomial.

The cancellation trick

Consider again the polynomial product

$$\begin{aligned}(314x^3 + 159x^2 + 265x + 358) \times (271x^3 + 828x^2 + 182x + 845) \\ = 85094x^6 + 303081x^5 + 260615x^4 + 610706x^3 \\ + 479009x^2 + 289081x + 302510.\end{aligned}$$

I said before that we only want the "top half" of this polynomial.

Actually this is not quite true. It would be good enough to find a polynomial that **when evaluated at $x = B$** gives the same result as evaluating only the "top half" at $x = B$.

The cancellation trick

I claim that it suffices to compute

$$Bx^{-d+1}U(x)V(x) \pmod{x^d - Bx^{d-1} + B}.$$

The cancellation trick

I claim that it suffices to compute

$$Bx^{-d+1}U(x)V(x) \pmod{x^d - Bx^{d-1} + B}.$$

In our running example:

$$\begin{aligned} &1000x^{-3}U(x)V(x) \pmod{x^4 - 1000x^3 + 1000} \\ &\quad = 84614991x^3 + 781800919x^2 + 549393490x + 913216000. \end{aligned}$$

The cancellation trick

I claim that it suffices to compute

$$Bx^{-d+1}U(x)V(x) \pmod{x^d - Bx^{d-1} + B}.$$

In our running example:

$$\begin{aligned} &1000x^{-3}U(x)V(x) \pmod{x^4 - 1000x^3 + 1000} \\ &\quad = 84614991x^3 + 781800919x^2 + 549393490x + 913216000. \end{aligned}$$

Evaluating at $x = B$ yields

$$85397342225706000.$$

Compare with full product:

$$853973422261852983835 10.$$

The cancellation trick

Why does it work?

$$
\begin{aligned}
&85094000B^3\\
&\quad 479009B^3\\
&-479009000B^2\\
&\quad 303081000B^2\\
&\quad\quad -289081B^2\\
&\quad\quad\quad 289081000B^1\\
&\quad\quad\quad 260615000B^1\\
&\quad\quad\quad\quad -302510B^1\\
&\quad\quad\quad\quad\quad 302510000B^0\\
&\quad\quad\quad\quad\quad 610706000B^0\\
\hline
&8539734225706000
\end{aligned}
$$

Interesting higher-order coefficients in BLUE.

Unwanted lower-order coefficients in RED.

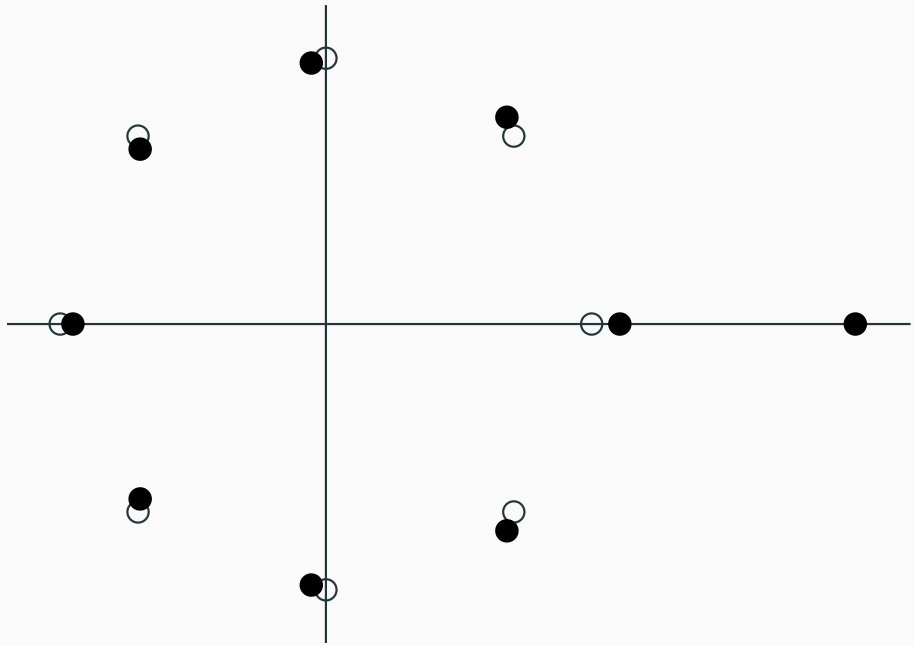
Roots of the special modulus

How do we multiply modulo $P(x) = x^d - Bx^{d-1} + B$?

This polynomial has roots very close to those of $x^{d-1} - 1$ (plus one extra root near B itself).

Example for $d = 9$ and $B = 2$:

- ◦ roots of $x^8 - 1$
- • roots of $x^9 - 2x^8 + 2$



A fast truncated multiplication algorithm

Sketch of fast algorithm for evaluating at roots of $P(x)$:

1. Compose $U(x)$ with a power series that maps roots of $x^{d-1} - 1$ to roots of $P(x)$.
2. Use ordinary FFT to evaluate at roots of $x^{d-1} - 1$.

A fast truncated multiplication algorithm

Sketch of fast algorithm for evaluating at roots of $P(x)$:

1. Compose $U(x)$ with a power series that maps roots of $x^{d-1} - 1$ to roots of $P(x)$.
2. Use ordinary FFT to evaluate at roots of $x^{d-1} - 1$.

Why do we get a factor of $3/4$ speedup?

- Save factor of 2 in transform length (d vs $2d$).
- Lose factor of $3/2$ due to larger coefficients.

Back to the real world

Does it work in practice?

Can I actually speed up truncated multiplication in GMP?

Back to the real world

Does it work in practice?

Can I actually speed up truncated multiplication in GMP?

No. GMP does not use FFTs over $\mathbf{C}$. It works over $\mathbf{Z}/F_k\mathbf{Z}$.

Back to the real world

Does it work in practice?

Can I actually speed up truncated multiplication in GMP?

No. GMP does not use FFTs over $\mathbf{C}$. It works over $\mathbf{Z}/F_k\mathbf{Z}$.

Can I speed up truncated multiplication in my own integer arithmetic library?

Back to the real world

Does it work in practice?

Can I actually speed up truncated multiplication in GMP?

No. GMP does not use FFTs over $\mathbf{C}$. It works over $\mathbf{Z}/F_k\mathbf{Z}$.

Can I speed up truncated multiplication in my own integer arithmetic library?

No. My library does FFTs over $\mathbf{F}_p$.

The archimedean property of $\mathbf{C}$ is absolutely crucial.

My dream

The million dollar question

Can the truncated multiplication algorithm be adapted to work over $\mathbf{F}_p$?

The cancellation trick still works.

What is missing is a way of evaluating quickly at the roots of a polynomial like

$$P(x) = x^d - Bx^{d-1} + B.$$

For example, is it possible to choose d and/or B and/or p so that the roots of $P(x)$ modulo p have some special structure?

Has anyone seen these sorts of polynomials before?

Primes with cyclic structure

Instead of trying to solve the truncated product problem, we could just avoid it altogether.

Idea: switch coefficient ring from $\mathbf{C}$ to $\mathbf{F}_p$, where p has some sort of "cyclic" structure.

Then multiplication modulo p might map more efficiently onto the FFT, and will hopefully lead to $K = 4$.

Four algorithms along these lines have been proposed.

Primes with cyclic structure, attempt #1

H.–van der Hoeven–Lecerf (2014): use a Mersenne prime

$$p = 2^q - 1.$$

Multiplication in $\mathbf{F}_p$ can be converted (using Crandall–Fagin trick, 1994) to multiplication modulo $x^d - 1$.

Primes with cyclic structure, attempt #1

H.–van der Hoeven–Lecerf (2014): use a Mersenne prime

$$p = 2^q - 1.$$

Multiplication in $\mathbf{F}_p$ can be converted (using Crandall–Fagin trick, 1994) to multiplication modulo $x^d - 1$.

We do not know if there are infinitely many such primes.

Proof of $K = 4$ depends on (a slight weakening of) the Lenstra–Pomerance–Wagstaff conjecture:

$$\#\{\text{Mersenne primes } p < x\} \sim \frac{e^\gamma}{\log 2} \log\log x.$$

Primes with cyclic structure, attempt #1

H.–van der Hoeven–Lecerf (2014): use a Mersenne prime

$$p = 2^q - 1.$$

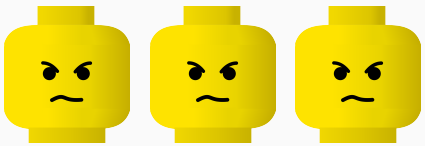
Multiplication in $\mathbf{F}_p$ can be converted (using Crandall–Fagin trick, 1994) to multiplication modulo $x^d - 1$.

We do not know if there are infinitely many such primes.

Proof of $K = 4$ depends on (a slight weakening of) the Lenstra–Pomerance–Wagstaff conjecture:

$$\#\{\text{Mersenne primes } p < x\} \sim \frac{e^\gamma}{\log 2} \log\log x.$$

This seems very, very, very hard.



Primes with cyclic structure, attempt #2

Covanov–Thomé (2015): use a generalised Fermat prime

$$p = r^{2^\lambda} + 1.$$

Multiplication in $\mathbf{F}_p$ is converted to multiplication modulo $x^{2^\lambda} + 1$.

Primes with cyclic structure, attempt #2

Covanov–Thomé (2015): use a generalised Fermat prime

$$p = r^{2^\lambda} + 1.$$

Multiplication in $\mathbf{F}_p$ is converted to multiplication modulo $x^{2^\lambda} + 1$.

They are apparently much more common than Mersenne primes.

Proof of $K = 4$ depends on a strong form of the Bateman–Horn conjecture.

But we can't even prove there are infinitely many primes of the form $r^2 + 1$!

Primes with cyclic structure, attempt #2

Covanov–Thomé (2015): use a generalised Fermat prime

$$p = r^{2^\lambda} + 1.$$

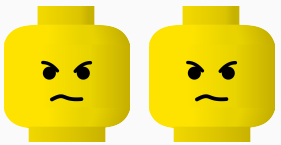
Multiplication in $\mathbf{F}_p$ is converted to multiplication modulo $x^{2^\lambda} + 1$.

They are apparently much more common than Mersenne primes.

Proof of $K = 4$ depends on a strong form of the Bateman–Horn conjecture.

But we can't even prove there are infinitely many primes of the form $r^2 + 1$!

This seems very, very hard.



Primes with cyclic structure, attempt #3

H.–van der Hoeven (2016): use a "plain vanilla FFT prime"

$$p = a \cdot 2^k + 1, \qquad 1 \leq a < k^2.$$

Multiplication in $\mathbf{F}_p$ converted to multiplication modulo $x^m + a$.

Primes with cyclic structure, attempt #3

H.–van der Hoeven (2016): use a "plain vanilla FFT prime"

$$p = a \cdot 2^k + 1, \qquad 1 \leq a < k^2.$$

Multiplication in $\mathbf{F}_p$ converted to multiplication modulo $x^m + a$.

Proof of $K = 4$ depends on a conjectural bound of Heath–Brown for the least prime in an arithmetic progression.

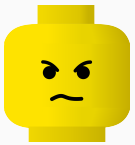
Primes with cyclic structure, attempt #3

H.–van der Hoeven (2016): use a "plain vanilla FFT prime"

$$p = a \cdot 2^k + 1, \qquad 1 \leq a < k^2.$$

Multiplication in $\mathbf{F}_p$ converted to multiplication modulo $x^m + a$.

Proof of $K = 4$ depends on a conjectural bound of Heath–Brown for the least prime in an arithmetic progression.

This seems quite tricky. 

Primes with cyclic structure, attempt #4

Finally: H.–van der Hoeven (ANTS 2018) show that for an almost arbitrary prime p, one can represent elements of $\mathbf{F}_p$ as expressions

$$a_0 + a_1\theta + \cdots + a_{m-1}\theta^{m-1},$$

where θ is a fixed $2m$-th root of unity modulo p, and the a_i are integers with around $(\log p)/m$ bits.

Primes with cyclic structure, attempt #4

Finally: H.–van der Hoeven (ANTS 2018) show that for an almost arbitrary prime p, one can represent elements of $\mathbf{F}_p$ as expressions

$$a_0 + a_1\theta + \cdots + a_{m-1}\theta^{m-1},$$

where θ is a fixed $2m$-th root of unity modulo p, and the a_i are integers with around $(\log p)/m$ bits.

We give fast algorithms for arithmetic in this representation, and conversions to and from the standard representation.

The key ingredient is Minkowski's theorem concerning lattice vectors in symmetric convex sets (geometry of numbers!).

Primes with cyclic structure, attempt #4

Finally: H.–van der Hoeven (ANTS 2018) show that for an almost arbitrary prime p, one can represent elements of $\mathbf{F}_p$ as expressions

$$a_0 + a_1\theta + \cdots + a_{m-1}\theta^{m-1},$$

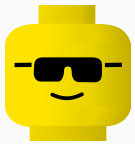
where θ is a fixed $2m$-th root of unity modulo p, and the a_i are integers with around $(\log p)/m$ bits.

We give fast algorithms for arithmetic in this representation, and conversions to and from the standard representation.

The key ingredient is Minkowski's theorem concerning lattice vectors in symmetric convex sets (geometry of numbers!).

This is enough to prove unconditionally

$$\mathrm{M}(n) = O(n \log n \, 4^{\log^* n}).$$



D.H. (age < 40),
demonstrating the currently fastest known skiing algorithm

Thank you!