Factorization: state of the art

1. Batch NFS
2. Factoring into coprimes
3. ECM

D. J. Bernstein
University of Illinois at Chicago

Tanja Lange
Technische Universiteit Eindhoven

Merging congruences

Problem: Convert

$x \equiv a \pmod{299}$,

$x \equiv b \pmod{799}$

into a single congruence.

Solution:

$x \equiv 799 \cdot 180 \cdot a - 299 \cdot 481 \cdot b$
$\quad \pmod{299 \cdot 799}$.

Underlying computation,
by Euclid's algorithm:
$799 \cdot 180 - 299 \cdot 481 = 1$.

Problem: Convert

$x \equiv a \pmod{299}$,

$x \equiv b \pmod{793}$

into a single congruence.

Much more difficult.

Can't write 1 as $793u + 299v$;

793 and 299 aren't coprime.

Euclid's algorithm discovers

$\gcd\{299, 793\} = 13$: specifically,

$13 = 793 \cdot 20 - 299 \cdot 53$,

$299 = 13 \cdot 23$, $793 = 13 \cdot 61$.

$\gcd\{13, 23\} = 1$. Thus

$x \equiv a \pmod{299} \iff$

$x \equiv a \pmod{13}$,

$x \equiv a \pmod{23}$.

$\gcd\{13, 61\} = 1$. Thus

$x \equiv b \pmod{793} \iff$

$x \equiv b \pmod{13}$,

$x \equiv b \pmod{61}$.

Underlying computations:

$23 \cdot 4 - 13 \cdot 7 = 1$;

$61 \cdot 3 - 13 \cdot 14 = 1$.

Assuming $a \equiv b \pmod{13}$:

$$x \equiv a \pmod{299},$$
$$x \equiv b \pmod{793} \iff$$
$$x \equiv a \pmod{13},$$
$$x \equiv a \pmod{23},$$
$$x \equiv b \pmod{61} \iff$$
$$\begin{aligned} x \equiv \quad & -1 \cdot 23 \cdot 61 \cdot a \\ & + 13 \cdot 21 \cdot 61 \cdot a \\ & - 13 \cdot 23 \cdot 51 \cdot b \end{aligned}$$
$$\pmod{13 \cdot 23 \cdot 61}.$$

Problem: Convert

$x \equiv a \pmod{103816603}$,

$x \equiv b \pmod{22649627}$

into a single congruence.

$\gcd\{103816603, 22649627\} = 187$;

$103816603 = 187 \cdot 555169$;

$22649627 = 187 \cdot 121121$.

Now encounter another difficulty:
$187, 555169$ aren't coprime;
congruence mod 103816603
is *not* equivalent to
separate congruences
mod 187 and mod 555169.

Continue computing gcds and exact quotients:

$\gcd\{555169, 187\} = 17;$
$555169/17 = 32657;$
$187/17 = 11;$
$32657/17 = 1921;$
$1921/17 = 113;$
$121121/11 = 11011;$
$11011/11 = 1001;$
$1001/11 = 91.$

$11, 17, 91, 113$ are coprime;
$103816603 = 11 \cdot 17^4 \cdot 113;$
$22649627 = 11^4 \cdot 17 \cdot 91.$

$x \equiv \cdots \pmod{11^4 \cdot 17^4 \cdot 91 \cdot 113}.$

The natural coprime base

For any set $S \subseteq \{1, 2, 3, \ldots\}$:

There is a unique set "$\text{cb}\, S$" $\subseteq \{2, 3, \ldots\}$ such that

- $\text{cb}\, S$ can be obtained from $\{1\} \cup S$ via product, exact quotient, gcd;
- $\text{cb}\, S$ is coprime: $\gcd\{a, b\} = 1$ for all distinct $a, b \in \text{cb}\, S$; and
- S can be obtained from $\{1\} \cup \text{cb}\, S$ via product.

e.g. $\text{cb}\{103816603, 22649627\} = \{11, 17, 91, 113\}$.

Can use any coprime base
to merge congruences:
e.g., set of prime divisors.

Complete prime factorizations:
$103816603 = 11 \cdot 17^4 \cdot 113$;
$22649627 = 7 \cdot 11^4 \cdot 13 \cdot 17$.

$\{7, 11, 13, 17, 113\}$
is a coprime base for
$\{103816603, 22649627\}$.

But primality is overkill.
We only need coprimality.

Finding multiplicative relations

Define

$u_1 = 91$;

$u_2 = 119$;

$u_3 = 221$;

$u_4 = 1547$;

$u_5 = 6898073$.

Does $u_1^{1952681} u_2^{1513335} u_3^{634643}$ equal $u_4^{1708632} u_5^{439346}$?

Each side has logarithm ≈ 19466590.674872.

Which $(a, b, c, d, e) \in \mathbf{Z}^5$ have $u_1^a u_2^b u_3^c u_4^d u_5^e = 1$ in $\mathbf{Q}$?

Factor into primes:

$u_1 = p_1 p_2$ where $p_1 = 7$ and $p_2 = 13$;

$u_2 = p_1 p_3$ where $p_3 = 17$;

$u_3 = p_2 p_3$; $u_4 = p_1 p_2 p_3$;

$u_5 = p_1^4 p_2^2 p_3$.

Now $u_1^a u_2^b u_3^c u_4^d u_5^e = p_1^{a+b+d+4e} p_2^{a+c+d+2e} p_3^{b+c+d+e}$; and p_1, p_2, p_3 are distinct primes.

$u_1^a u_2^b u_3^c u_4^d u_5^e = 1 \Leftrightarrow$
$p_1^{a+b+d+4e} \cdots = 1 \Leftrightarrow$
$(a+b+d+4e, \ldots) = 0 \Leftrightarrow$
$(a, b, c, d, e) \in$
$(1, 1, 1, -2, 0)\mathbf{Z} + (3, 2, 0, -1, -1)\mathbf{Z}$.

Primality is again overkill.
All we needed was coprimality.

For any coprimes $p_1, p_2, \ldots$:

$p_1^{a_1} p_2^{a_2} \cdots = 1$
iff $(a_1, a_2, \ldots) = 0$.
Use in index calculus.

$p_1^{a_1} p_2^{a_2} \cdots = \square$
iff $(a_1, a_2, \ldots) \bmod 2 = 0$.
Use in index calculus mod 2:
e.g., in NFS.

Bad RSA randomness

2004 Bauer–Laurie:
checked 18000 PGP RSA keys;
found 2 keys sharing a factor.

2012.02.14 Lenstra–Hughes–Augier–Bos–Kleinjung–Wachter
“Ron was wrong, Whit is right”
(to appear, Crypto 2012):
checked $7 \cdot 10^6$ SSL/PGP RSA keys; found $6 \cdot 10^6$ distinct keys;
factored 12720 of those,
thanks to shared prime factors.

2012.02.17 Heninger–Durumeric–Wustrow–Halderman announcement (to appear, USENIX Security 2012): checked $>10^7$ SSL/SSH RSA keys; factored 24816 SSL keys, 2422 SSH host keys.

“Almost all of the vulnerable keys were generated by and are used to secure embedded hardware devices such as routers and firewalls, not to secure popular web sites such as your bank or email provider.”

These computations factored the RSA keys into coprimes.

e.g. Factoring RSA keys

$p_1q_1, p_2q_2, p_3q_3,$

p_4q_2, p_5q_5, p_3q_6

into coprimes produces

$p_1q_1, p_2 \cdot q_2, p_3 \cdot q_3,$

$p_4 \cdot q_2, p_5q_5, p_3 \cdot q_6,$

assuming $p_1, p_2, p_3, p_4, p_5,$

q_1, q_2, q_3, q_5, q_6 distinct.

Long history

More examples, applications of factoring into coprimes: see 1890 Stieltjes; 1974 Collins; 1985 Kaltofen; 1985 Della Dora DiCrescenzo Duval; 1986 Bach Miller Shallit; 1986 von zur Gathen; 1986 Lüneburg; 1989 Pohst Zassenhaus; 1990 Teitelbaum; 1990 Smedley; 1993 Bach Driscoll Shallit; 1994 Ge; 1994 Buchmann Lenstra; 1996 Bernstein; 1997 Silverman; 1998 Cohen Diaz y Diaz Olivier; 1998 Storjohann; . . .

Speed of factoring into coprimes

Obvious algorithm to compute cb S and factor S over cb S: $O(n^3)$ bit ops for n input bits. (frequently reinvented)

More careful algorithm, avoiding pointless gcd calculations: $O(n^2)$. (1990 Bach Driscoll Shallit)

Can do much better for large n: $n^{1+o(1)}$; in fact $n(\lg n)^{O(1)}$. (1995 Bernstein)

$n(\lg n)^{4+o(1)}$. (2004 Bernstein)

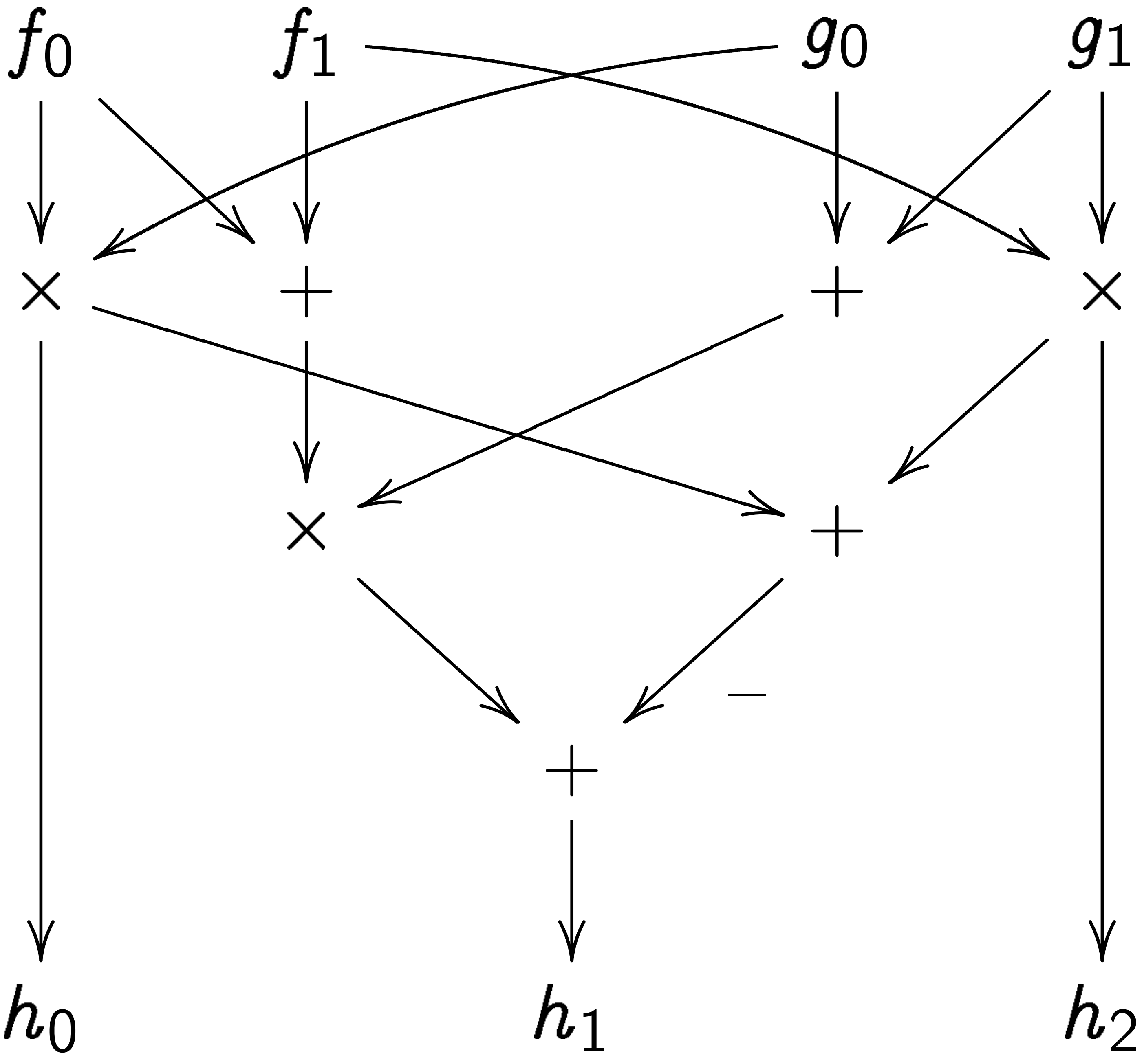
$n(\lg n)^{4+o(1)}$ is worst-case,
handling many obscure situations.

Most applications are
much more constrained.
Take advantage of constraints:
$n(\lg n)^{3+o(1)}$, sometimes
$n(\lg n)^{2+o(1)}$.

Many more tweaks
save constant factors
or noticeable $1 + o(1)$ factors.

Slides coming up:
how these algorithms work,
with the most important tweaks.

Algebraic algorithms



× multiplies its two inputs.

$+$ adds its two inputs.

$+^{-}$ subtracts its two inputs.

This "**R**-algebraic algorithm" computes product $h_0 + h_1 x + h_2 x^2$ of $f_0 + f_1 x, g_0 + g_1 x \in \mathbf{R}[x]$.

More precisely: It computes the coeffs of the product (on standard basis $1, x, x^2$) given the coeffs of the factors (on standard bases $1, x$ and $1, x$).

3 mults, 4 adds.
Compare to obvious algorithm:
4 mults, 1 add.

(1963 Karatsuba)

Total **R**-algebraic complexity

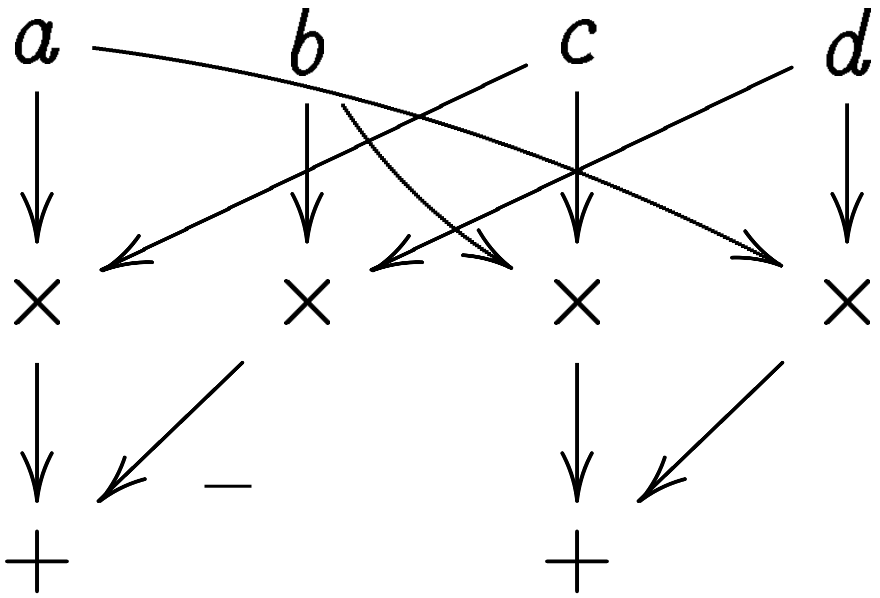
Are 3 mults, 4 adds
better than 4 mults, 1 add?

Depends on cost metric.

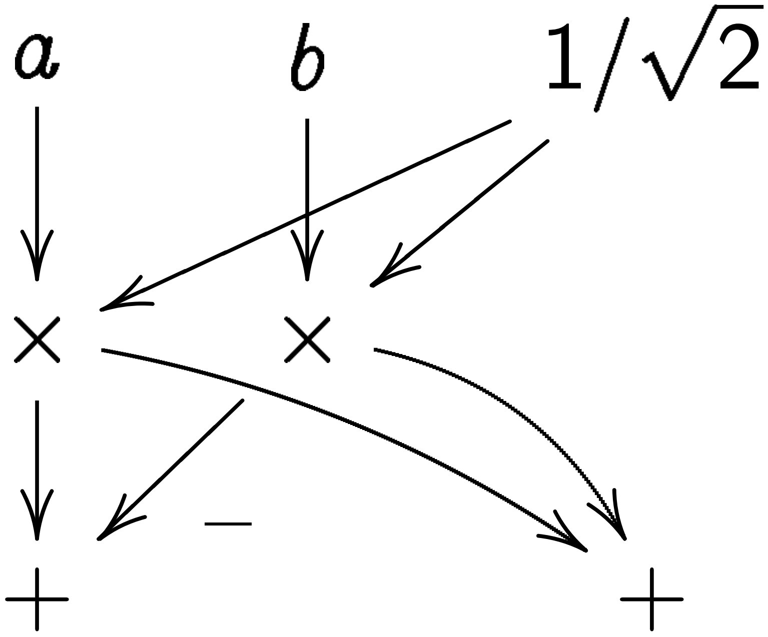
Cost metric for now: "**R**-ops".
$+$ ("add"): 1 **R**-op.
$+^-$ (also "add"): 1 **R**-op.
$\times$ ("mult"): 1 **R**-op.
Constant in **R**: 0 **R**-ops.

3 mults, 4 adds: 7 **R**-ops.
4 mults, 1 add: 5 **R**-ops.

6 **R**-ops to multiply in **C**
(on standard basis $1, i$):



4 **R**-ops to multiply by $\sqrt{i}$:



Many other cost measures.

Some measures emphasize adds.
e.g. floating point on one core
of Core 2 Quad: #cycles
$\approx \max\{\#\mathbf{R}\text{-adds}, \#\mathbf{R}\text{-mults}\}/2$.
Typically more adds than mults.

Some measures emphasize mults.
e.g. Dedicated hardware
for floating-point arithmetic:
mults more expensive than adds.

For simplicity we'll take
$\#\mathbf{R}\text{-adds} + \#\mathbf{R}\text{-mults}$.

Fast Fourier transforms

Define $\zeta_n \in \mathbf{C}$ as $\exp(2\pi i/n)$.
Define $T_n : \mathbf{C}[x]/(x^n - 1) \hookrightarrow\!\!\!\!\twoheadrightarrow \mathbf{C}^n$
as $f \mapsto f(1), f(\zeta_n), \ldots, f(\zeta_n^{n-1})$.

Can very quickly compute T_n.

First publication of fast algorithm:
1866 Gauss.

Easy to see that Gauss's FFT uses
$O(n \lg n)$ arithmetic operations
if $n \in \{1, 2, 4, 8, \ldots\}$.

Several subsequent reinventions,
ending with 1965 Cooley/Tukey.

Inverse map is also very fast.

Multiplication in $\mathbf{C}^n$ is very fast.

1966 Sande, 1966 Stockham:
Can very quickly multiply
in $\mathbf{C}[x]/(x^n - 1)$ or $\mathbf{C}[x]$ or $\mathbf{R}[x]$
by mapping $\mathbf{C}[x]/(x^n - 1)$ to $\mathbf{C}^n$.
"Fast convolution."

Given $f, g \in \mathbf{C}[x]/(x^n - 1)$:
compute fg as $T_n^{-1}(T_n(f)T_n(g))$.

Given $f, g \in \mathbf{C}[x]$, $\deg fg < n$:
compute fg from
its image in $\mathbf{C}[x]/(x^n - 1)$.

Cost $O(n \lg n)$.

A closer look at costs

More precise analysis of Gauss FFT (and Cooley-Tukey FFT):

$\mathbf{C}[x]/(x^n - 1) \hookrightarrow\!\!\!\!\rightarrow \mathbf{C}^n$ using
$n \lg n$ $\mathbf{C}$-adds (2 ops each),
$(n \lg n)/2$ $\mathbf{C}$-mults (6 each),
if $n \in \{1, 2, 4, 8, \ldots\}$.

Total cost $5n \lg n$.
After peephole optimizations:
cost $5n \lg n - 10n + 16$
if $n \in \{4, 8, 16, 32, \ldots\}$.

Either way, $5n \lg n + O(n)$.

What about cost of convolution?

$5n \lg n + O(n)$ to compute $T_n(f)$,
$5n \lg n + O(n)$ to compute $T_n(g)$,
$O(n)$ to multiply in $\mathbf{C}^n$,
similar $5n \lg n + O(n)$ for T_n^{-1}.

Total cost $15n \lg n + O(n)$
to compute $fg \in \mathbf{C}[x]/(x^n - 1)$
given $f, g \in \mathbf{C}[x]/(x^n - 1)$.

Total cost $(15/2)n \lg n + O(n)$
to compute $fg \in \mathbf{R}[x]/(x^n - 1)$
given $f, g \in \mathbf{R}[x]/(x^n - 1)$: map
$\mathbf{R}[x]/(x^n - 1) \hookrightarrow\!\!\!\!\rightarrow \mathbf{R}^2 \oplus \mathbf{C}^{n/2-1}$
(Gauss) to save half the time.

1968 R. Yavne: Can do better!

Cost $4n \lg n + O(n)$

to map $\mathbf{C}[x]/(x^n - 1) \hookrightarrow\!\!\!\!\rightarrow \mathbf{C}^n$,

if $n \in \{1, 2, 4, 8, 16, \ldots\}$.

1968 R. Yavne: Can do better!
Cost $4n \lg n + O(n)$
to map $\mathbf{C}[x]/(x^n - 1) \hookrightarrow\!\!\!\!\rightarrow \mathbf{C}^n$,
if $n \in \{1, 2, 4, 8, 16, \ldots\}$.

2004 James Van Buskirk:
Can do better!
Cost $(34/9) n \lg n + O(n)$.

Expositions of the algorithm:
Frigo, Johnson,
in *IEEE Trans. Signal Processing*;
Lundy, Van Buskirk,
in *Computing*;
Bernstein, *AAECC*.

Understanding the FFT

If $f \in \mathbf{C}[x]$ and
$f \bmod x^4 - 1 =$
$f_0 + f_1 x + f_2 x^2 + f_3 x^3$ then
$f \bmod x^2 - 1 =$
$(f_0 + f_2) + (f_1 + f_3)x$,
$f \bmod x^2 + 1 =$
$(f_0 - f_2) + (f_1 - f_3)x$.

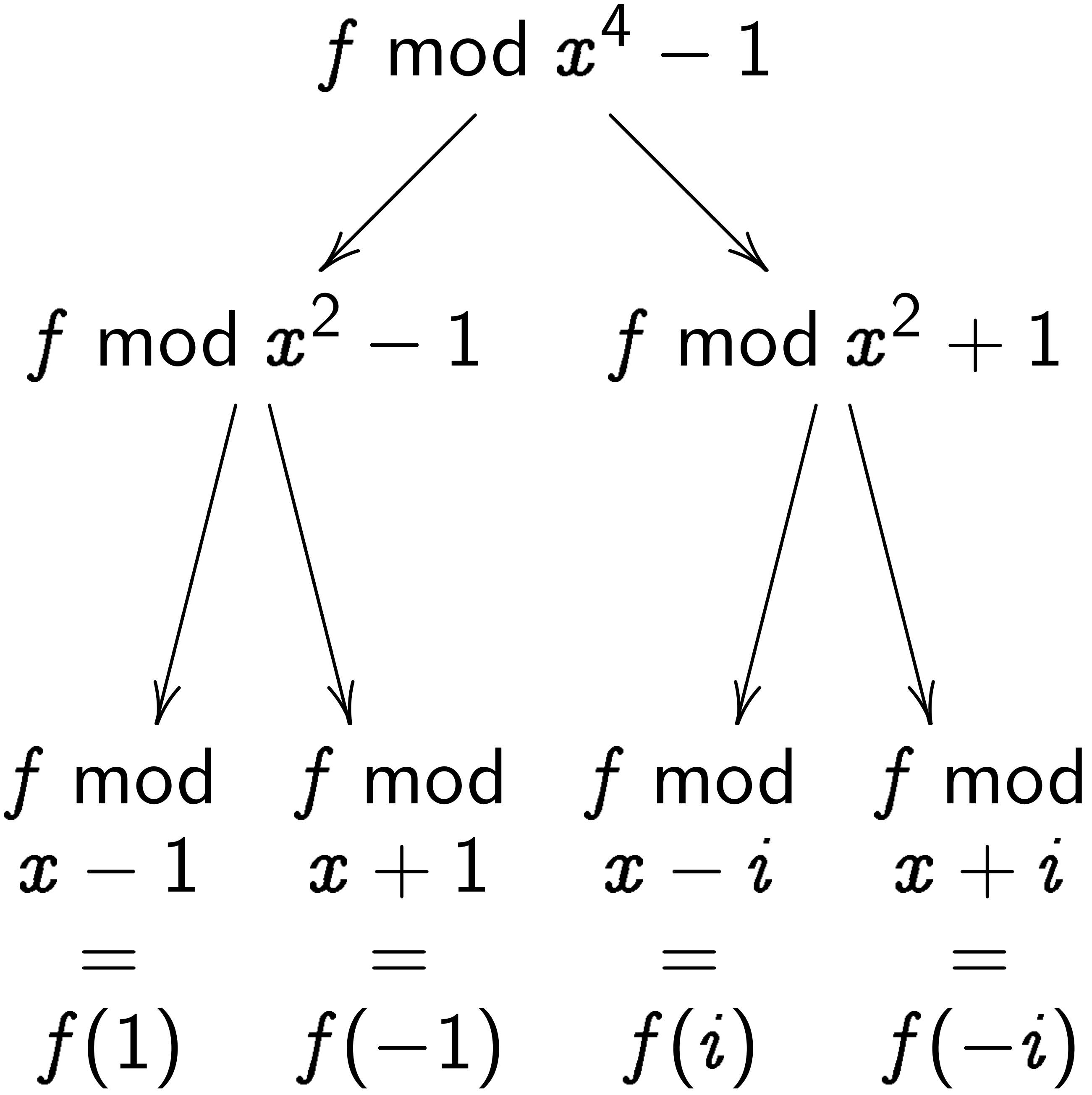
Given $f \bmod x^4 - 1$,
8 **R**-ops to compute
$f \bmod x^2 - 1$, $f \bmod x^2 + 1$.

"$\mathbf{C}[x]$-morphism $\mathbf{C}[x]/(x^4 - 1) \hookrightarrow\!\!\!\rightarrow$
$\mathbf{C}[x]/(x^2 - 1) \oplus \mathbf{C}[x]/(x^2 + 1)$."

If $f \in \mathbf{C}[x]$ and
$f \bmod x^{2n} - r^2 =$
$f_0 + f_1 x + \cdots + f_{2n-1} x^{2n-1}$ then
$f \bmod x^n - r =$
$$(f_0 + r f_n) + (f_1 + r f_{n+1})x \\ \quad + (f_2 + r f_{n+2})x^2 + \cdots,$$
$f \bmod x^n + r =$
$$(f_0 - r f_n) + (f_1 - r f_{n+1})x \\ \quad + (f_2 - r f_{n+2})x^2 + \cdots.$$

Given $f_0, f_1, \ldots, f_{2n-1} \in \mathbf{C}$,
$\le 10n$ **R**-ops to compute
$f_0 + r f_n, f_1 + r f_{n+1}, \ldots,$
$f_0 - r f_n, f_1 - r f_{n+1}, \ldots.$
Note: can compute in place.

The FFT: Do this recursively!



$f \bmod x^4 - 1$

$f \bmod x^2 - 1$

$f \bmod x^2 + 1$

$f \bmod x - 1 = f(1)$

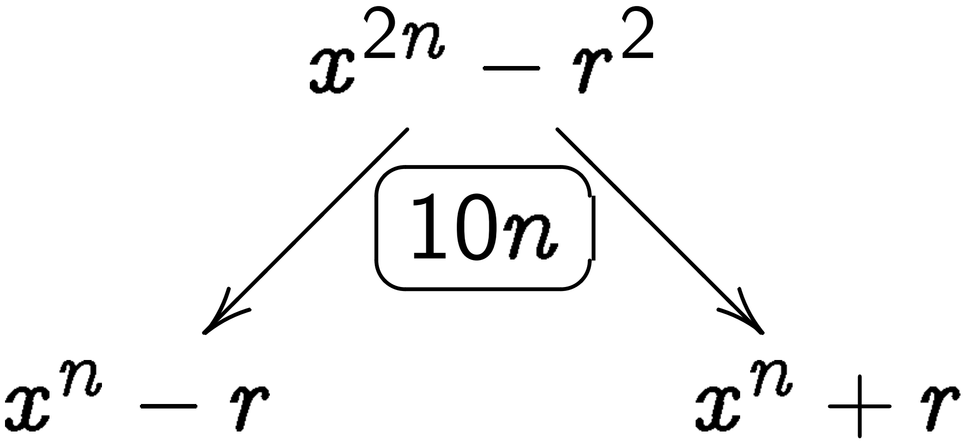
$f \bmod x + 1 = f(-1)$

$f \bmod x - i = f(i)$

$f \bmod x + i = f(-i)$

(expository idea: 1972 Fiduccia)

Modulus tree for one step:



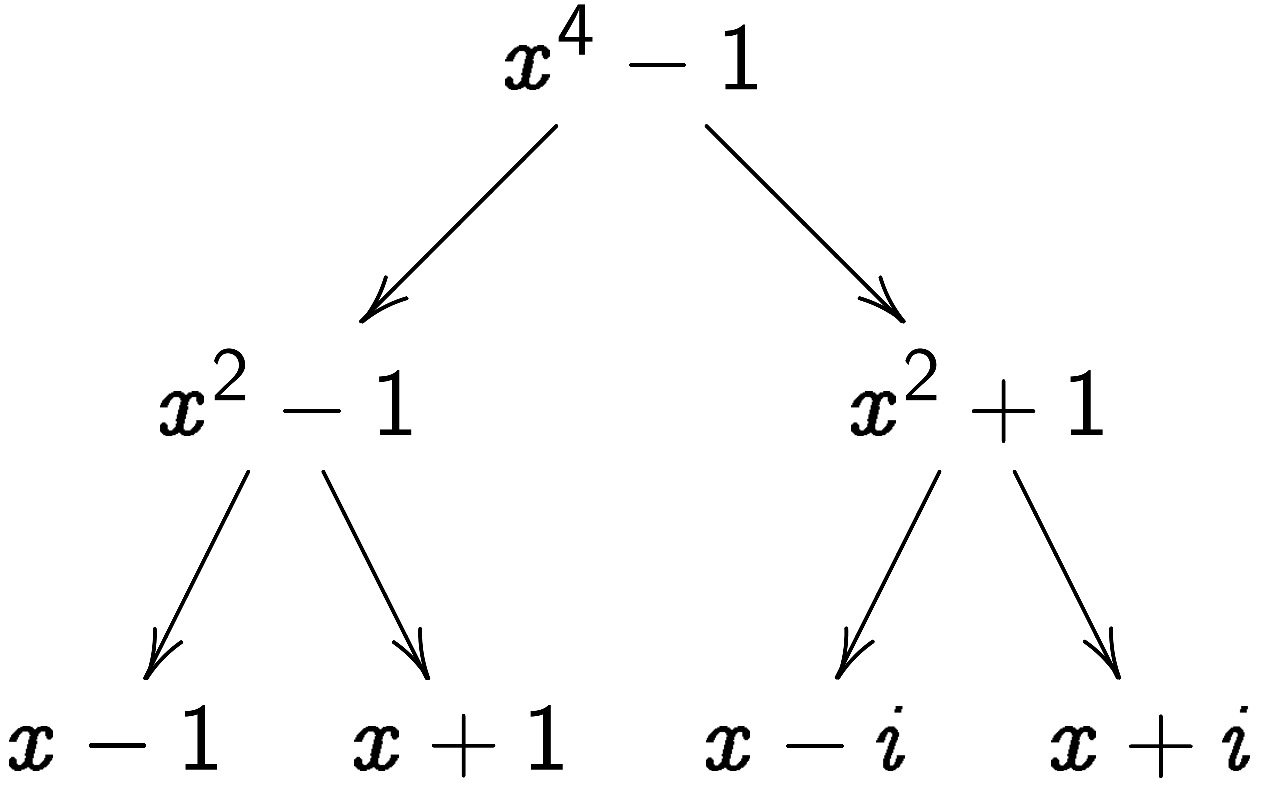
$x^{2n} - r^2$

$\boxed{10n}$

$x^n - r$

$x^n + r$

Modulus tree for full size-4 FFT:



$x^4 - 1$

$x^2 - 1$

$x^2 + 1$

$x - 1$

$x + 1$

$x - i$

$x + i$

Alternative: the twisted FFT

If $f \in \mathbf{C}[x]$ and
$f \bmod x^n + 1 =$
$g_0 + g_1 x + g_2 x^2 + \cdots$ then
$f(\zeta_{2n} x) \bmod x^n - 1 =$
$g_0 + \zeta_{2n} g_1 x + \zeta_{2n}^2 g_2 x^2 + \cdots.$

"$\mathbf{C}$-morphism $\mathbf{C}[x]/(x^n + 1) \hookrightarrow\!\!\!\!\rightarrow$ $\mathbf{C}[x]/(x^n - 1)$ by $x \mapsto \zeta_{2n} x$."

Modulus tree:

$$\begin{tikzcd}
x^n + 1 \arrow[d, "\boxed{6n}"'] \\
x^n - 1
\end{tikzcd}$$

Merge with the original FFT trick:

$$\begin{tikzcd}
 & x^{2n}-1 \arrow[dl] \arrow[dr] \arrow[d, phantom, "\boxed{4n}"] & \\
x^n-1 & & x^n+1 \arrow[d, "\boxed{6n}"'] \\
 & & x^n-1
\end{tikzcd}$$

“Twisted FFT” applies
this modulus tree recursively.

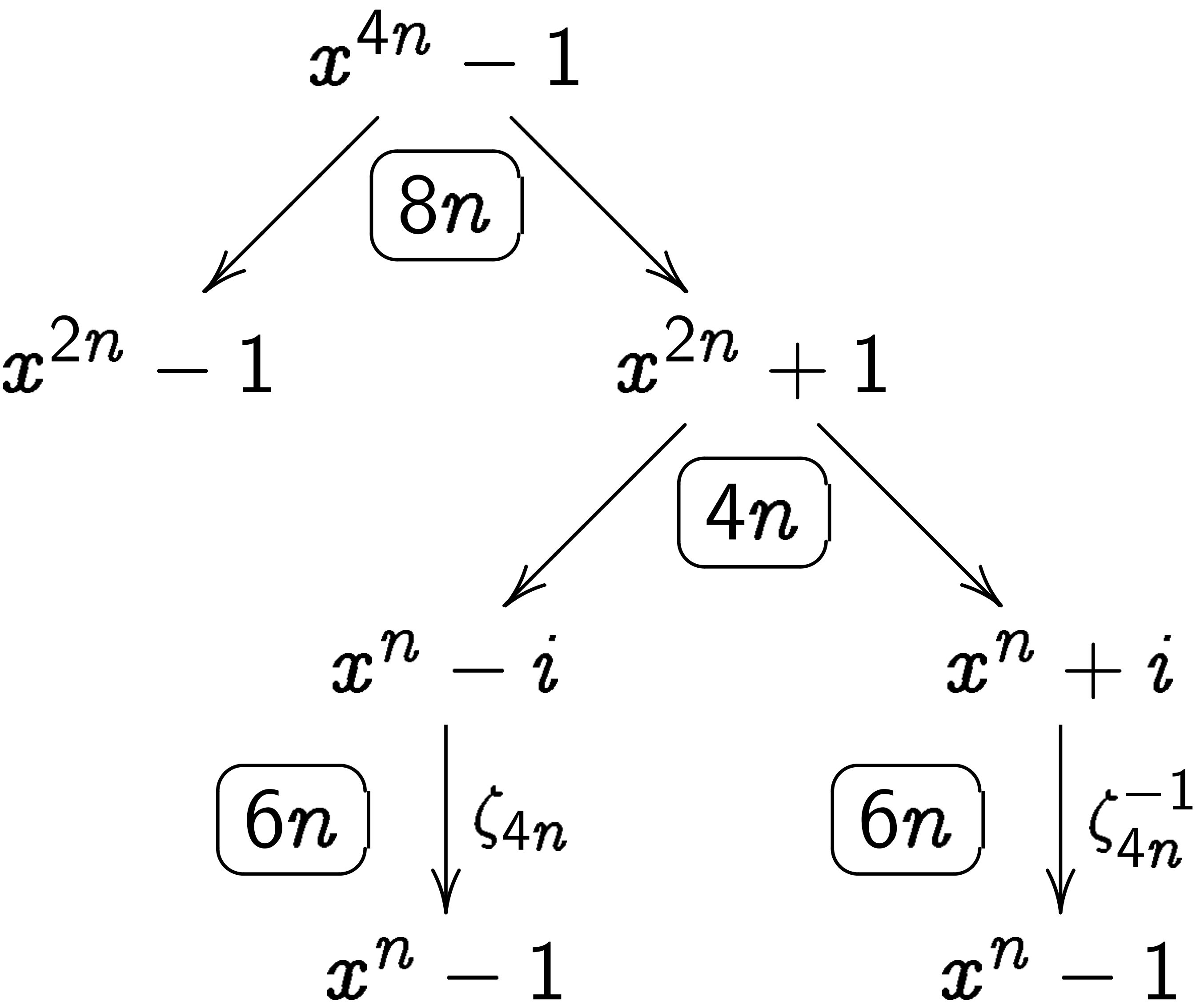
$5n \lg n + O(n)$ **R**-ops,
just like the original FFT.

The split-radix FFT

FFT and twisted FFT end up with
same number of mults by ζ_n,
same number of mults by $\zeta_{n/2}$,
same number of mults by $\zeta_{n/4}$,
etc.

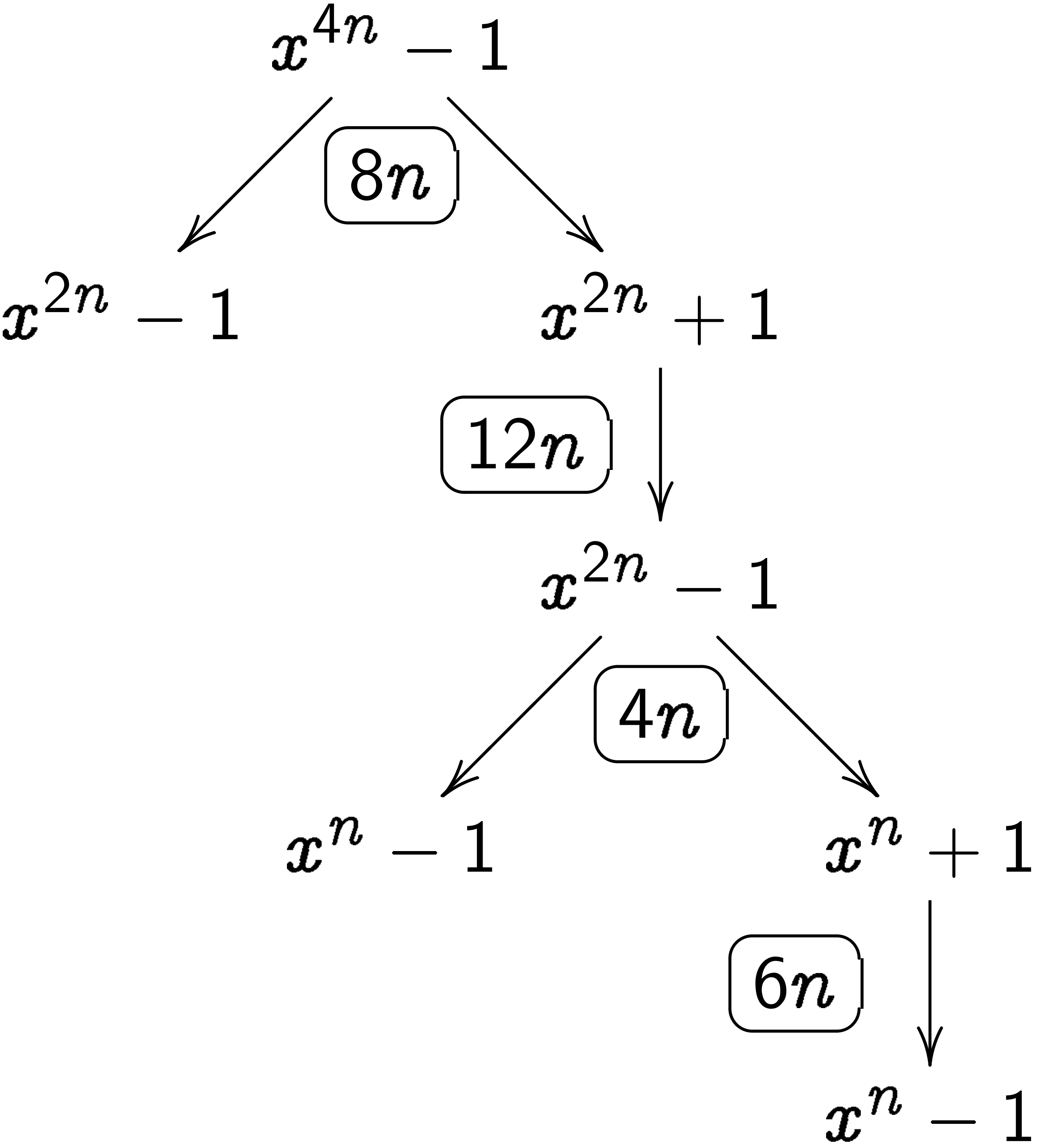
Is this necessary? No!
Split-radix FFT: more easy mults.
"Don't twist until you see
the whites of their i's."

(Same idea shows up in,
e.g., Fürer mult algorithm.)



Split-radix FFT applies
this modulus tree recursively.
$4n \lg n + O(n)$ **R**-ops.

Compare to how twisted FFT splits $4n$ into $2n, n, n$:



$$\begin{tikzcd}
 & x^{4n}-1 \arrow[dl] \arrow[dr] \arrow[d, phantom, "\boxed{8n}"] & & \\
x^{2n}-1 & {} & x^{2n}+1 \arrow[d, "\boxed{12n}"'] & \\
 & & x^{2n}-1 \arrow[dl] \arrow[dr] \arrow[d, phantom, "\boxed{4n}"] & \\
 & x^{n}-1 & {} & x^{n}+1 \arrow[d, "\boxed{6n}"'] \\
 & & & x^{n}-1
\end{tikzcd}$$

The tangent FFT

Several ways to achieve
6 **R**-ops for mult by $e^{i\theta}$.

One approach: Factor $e^{i\theta}$
as $(1 + i \tan\theta)\cos\theta$.
2 **R**-ops for mult by $\cos\theta$.
4 **R**-ops for mult by $1 + i \tan\theta$.

For stability and symmetry,
use $\max\{|\cos\theta|, |\sin\theta|\}$
instead of $\cos\theta$.

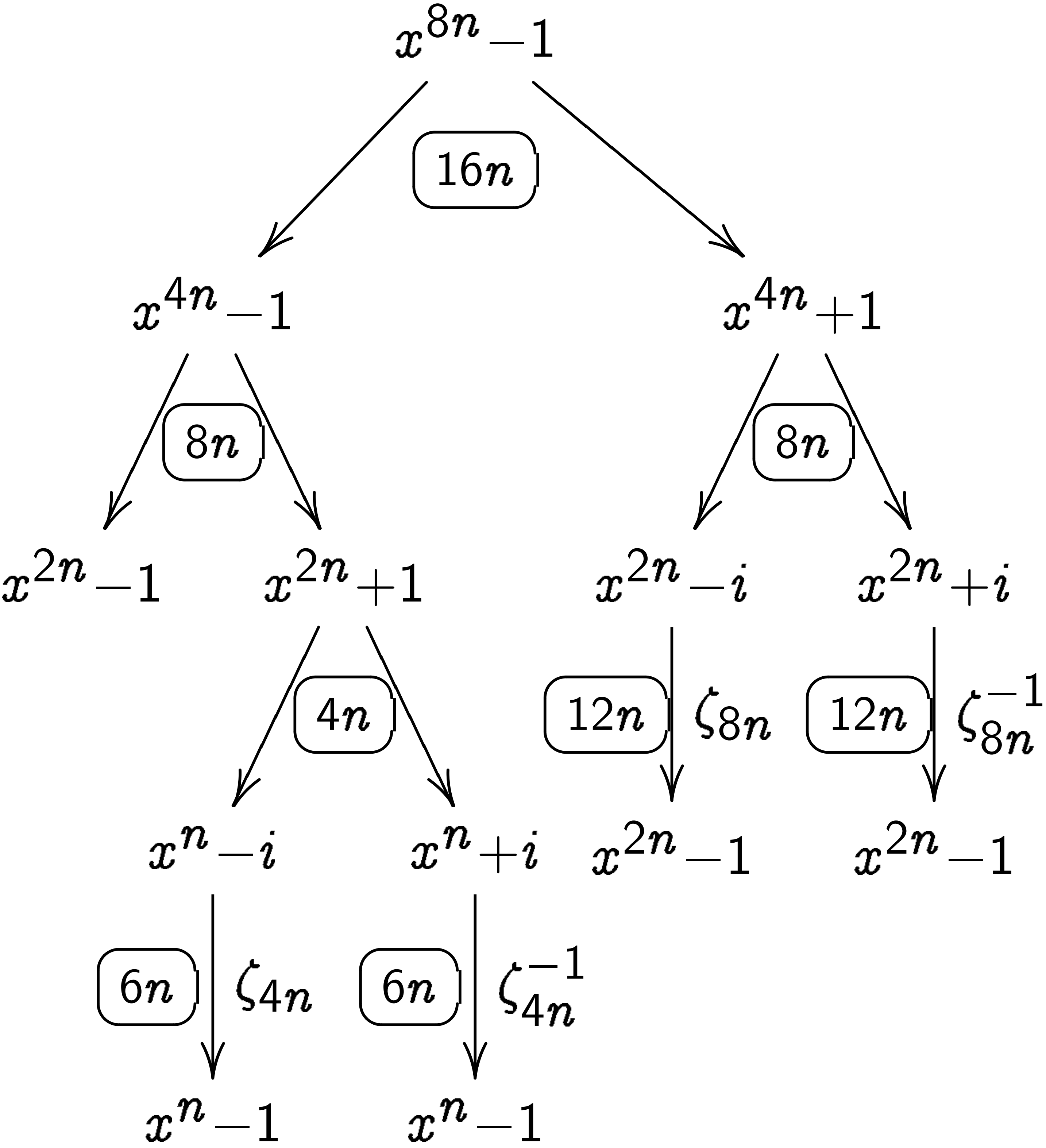
Surprise (Van Buskirk):
Can merge some cost-2 mults!

Rethink basis of $\mathbf{C}[x]/(x^n - 1)$.
Instead of $1, x, \ldots, x^{n-1}$ use
$1/s_{n,0}, x/s_{n,1}, \ldots, x^{n-1}/s_{n,n-1}$
where $s_{n,k} =$
$\max\left\{\left|\cos\frac{2\pi k}{n}\right|, \left|\sin\frac{2\pi k}{n}\right|\right\} \cdot$
$\max\left\{\left|\cos\frac{2\pi k}{n/4}\right|, \left|\sin\frac{2\pi k}{n/4}\right|\right\} \cdot$
$\max\left\{\left|\cos\frac{2\pi k}{n/16}\right|, \left|\sin\frac{2\pi k}{n/16}\right|\right\} \cdot$
$\cdots$.

Now $\left(g_0, g_1, \ldots, g_{n-1}\right)$ represents
$g_0/s_{n,0} + \cdots + g_{n-1}x^{n-1}/s_{n,n-1}$.

Note that $s_{n,k} = s_{n,k+n/4}$.
Note that $\zeta_n^k(s_{n/4,k}/s_{n,k})$ is
$\pm(1 + i\tan\cdots)$ or $\pm(\cot\cdots + i)$.

Look at how split-radix splits $8n$ into $2n, 2n, 2n, n, n$:



$x^{8n}-1$

$\boxed{16n}$

$x^{4n}-1$

$x^{4n}+1$

$\boxed{8n}$

$\boxed{8n}$

$x^{2n}-1$

$x^{2n}+1$

$x^{2n}-i$

$x^{2n}+i$

$\boxed{4n}$

$\boxed{12n}$ ζ_{8n}

$\boxed{12n}$ ζ_{8n}^{-1}

$x^{n}-i$

$x^{n}+i$

$x^{2n}-1$

$x^{2n}-1$

$\boxed{6n}$ ζ_{4n}

$\boxed{6n}$ ζ_{4n}^{-1}

$x^{n}-1$

$x^{n}-1$

New basis saves $12n$:
$4n$ in ζ_{8n} twist, $4n$ in ζ_{8n}^{-1} twist,
$2n$ in ζ_{4n} twist, $2n$ in ζ_{4n}^{-1} twist.

New basis costs $8n$:
$4n$ to change basis of $x^{2n}+1$,
$4n$ to change basis
of top-left $x^{2n}-1$.

Overall $68n$ instead of $72n$.
Recurse: $(34/9)n\lg n+O(n)$,
as in 2004 Van Buskirk.

Open: Can 34/9 be improved?

Integer multiplication via FFT

(1971 Pollard; independently
1971 Nicholson; independently
1971 Schönhage Strassen)

Write two n-bit integers
as polys of degree $O(n/\lg n)$
with $O(\lg n)$-bit coefficients.

Multiply in $\mathbf{R}[x]$, by FFT,
using floating-point arithmetic.
Round coefficients to integers.

$\Theta(n)$ $\mathbf{R}$-ops on coefficients,
each with precision $\Theta(\lg n)$.
$\Rightarrow n(\lg n)^{1+o(1)}$ bit ops.

More subtle FFT applications
violate this structure
for integer multiplication.

Still $n(\lg n)^{1+o(1)}$,
but save non-constant factors.

Surveys of techniques:

`http://cr.yp.to/papers.html#m3`

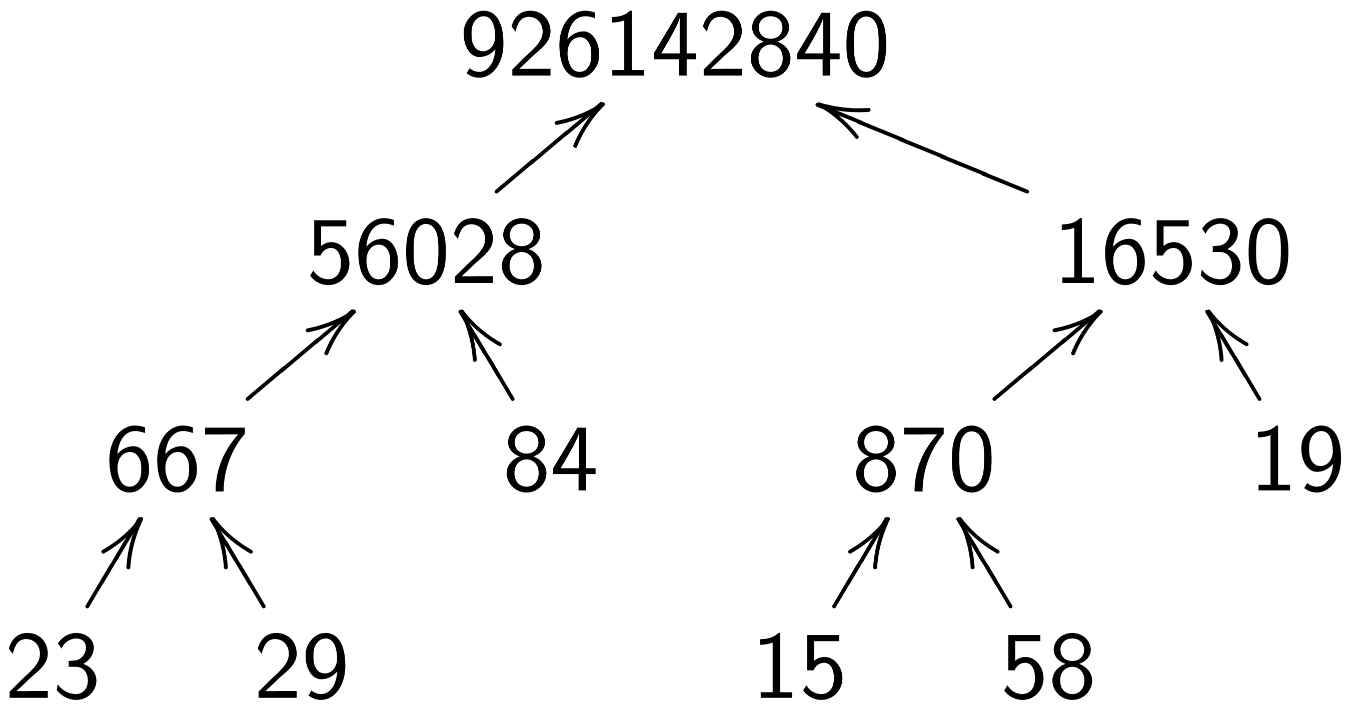
`http://cr.yp.to/papers.html#multapps`

Product trees

$n(\lg n)^{2+o(1)}$ bit ops
where n is number of input bits:
Given $x_1, x_2, \ldots, x_k \in \mathbf{Z}$,
compute $x_1 x_2 \cdots x_k$.

Actually compute
product tree of $x_1, x_2, \ldots, x_k$.
Root is $x_1 x_2 \cdots x_k$.
Has left subtree if $k \geq 2$:
product tree of $x_1, \ldots, x_{\lceil k/2 \rceil}$.
Also right subtree if $k \geq 2$:
product tree of $x_{\lceil k/2 \rceil + 1}, \ldots, x_k$.

e.g. tree for 23, 29, 84, 15, 58, 19:



Tree has $\le (\lg n)^{1+o(1)}$ levels.
Each level: $\le n(\lg n)^{0+o(1)}$ bits.

Obtain each level using
$n(\lg n)^{1+o(1)}$ bit ops
by multiplying lower-level pairs.

FFT doubling

(2004 Kramer)

Consider product tree for x_1, x_2, x_3, x_4, each $b/4$ bits.

Compute $x_1 x_2$ as $\mathrm{FFT}_{b/2}^{-1}(\mathrm{FFT}_{b/2}(x_1)\,\mathrm{FFT}_{b/2}(x_2))$.

Compute $x_1 x_2 x_3 x_4$ as $\mathrm{FFT}_{b}^{-1}(\mathrm{FFT}_{b}(x_1 x_2)\,\mathrm{FFT}_{b}(x_3 x_4))$. First half of $\mathrm{FFT}_b(x_1 x_2)$ is $\mathrm{FFT}_{b/2}(x_1 x_2)$, already known!

For large product trees, $1.5 + o(1)$ speedup.

Integer division

$n(\lg n)^{1+o(1)}$ bit ops
where n is number of input bits:
Given $a, b \in \mathbf{Z}$ with $b \neq 0$,
compute $\lfloor a/b \rfloor$ and $a \bmod b$.

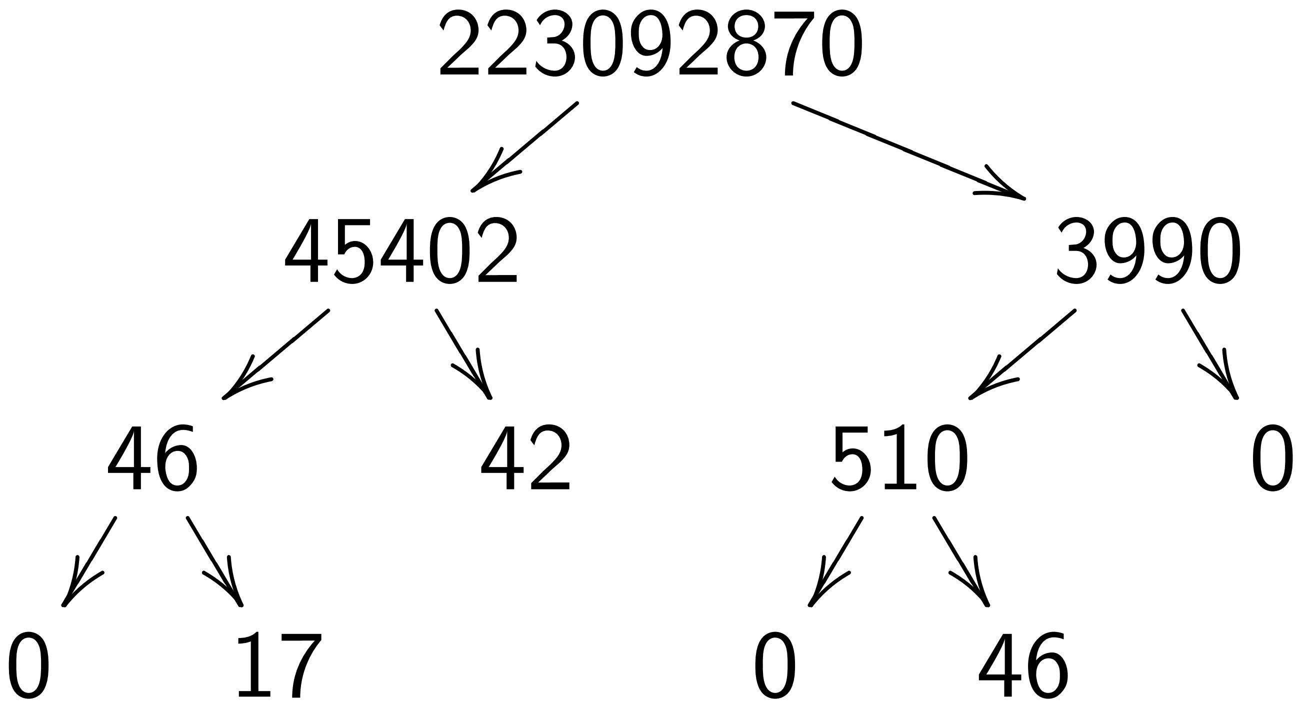
Idea: If r is close to $1/b$
then $(2 - rb)r$ is much closer.

(idea: 1740 Simpson;
$n^{1+o(1)}$ bit ops: 1966 Cook;
many subsequent speedups)

Remainder trees

Remainder tree
of $r, x_1, x_2, \ldots, x_k$ has
one node $r \bmod t$ for each node t
in product tree of $x_1, x_2, \ldots, x_k$.

e.g. remainder tree of
223092870, 23, 29, 84, 15, 58, 19:



223092870

45402 3990

46 42 510 0

0 17 0 46

$n(\lg n)^{2+o(1)}$ bit ops:
Given $r \in \mathbf{Z}$ and
nonzero $x_1, \ldots, x_k \in \mathbf{Z}$,
compute remainder tree
of $r, x_1, \ldots, x_k$.

In particular, compute
$r \bmod x_1, \ldots, r \bmod x_k$.

In particular, see which of
$x_1, \ldots, x_k$ divide r.

(1972 Moenck Borodin,
for "single precision" x_i's,
whatever exactly that means)

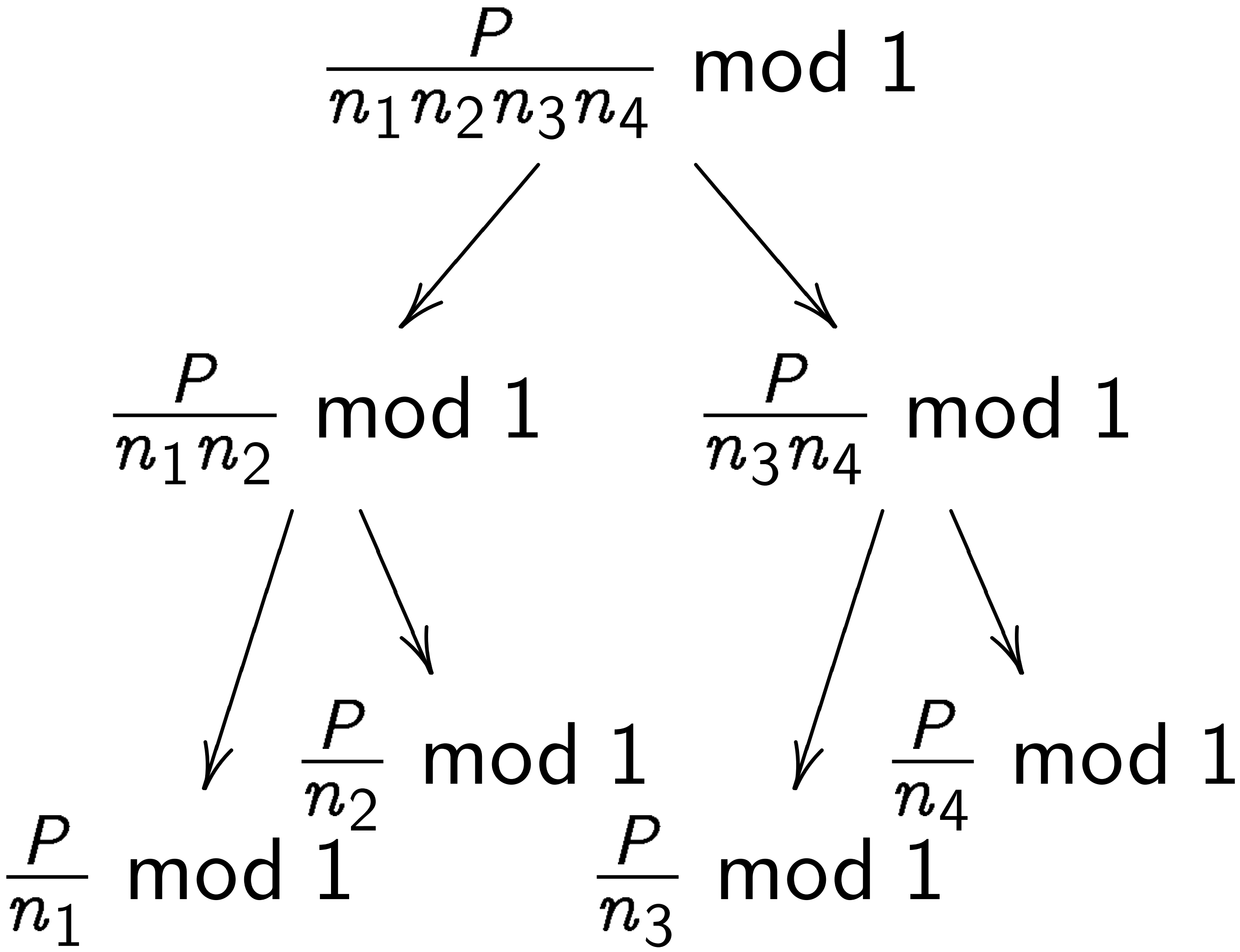
Scaled remainder trees

Replace almost all of the divisions with multiplications. Constant-factor speedup.

(speedup in function-field case, using polynomial reversal etc.: 2003 Bostan Lecerf Schost; structure: 2004 Bernstein)

With redundancies eliminated (1992 Montgomery, 2004 Kramer, etc.): new structure is $2.6 + o(1)$ times faster than remainder tree.

Scaled remainder tree:



$\frac{P}{n_1 n_2 n_3 n_4} \bmod 1$

$\frac{P}{n_1 n_2} \bmod 1$

$\frac{P}{n_3 n_4} \bmod 1$

$\frac{P}{n_1} \bmod 1$

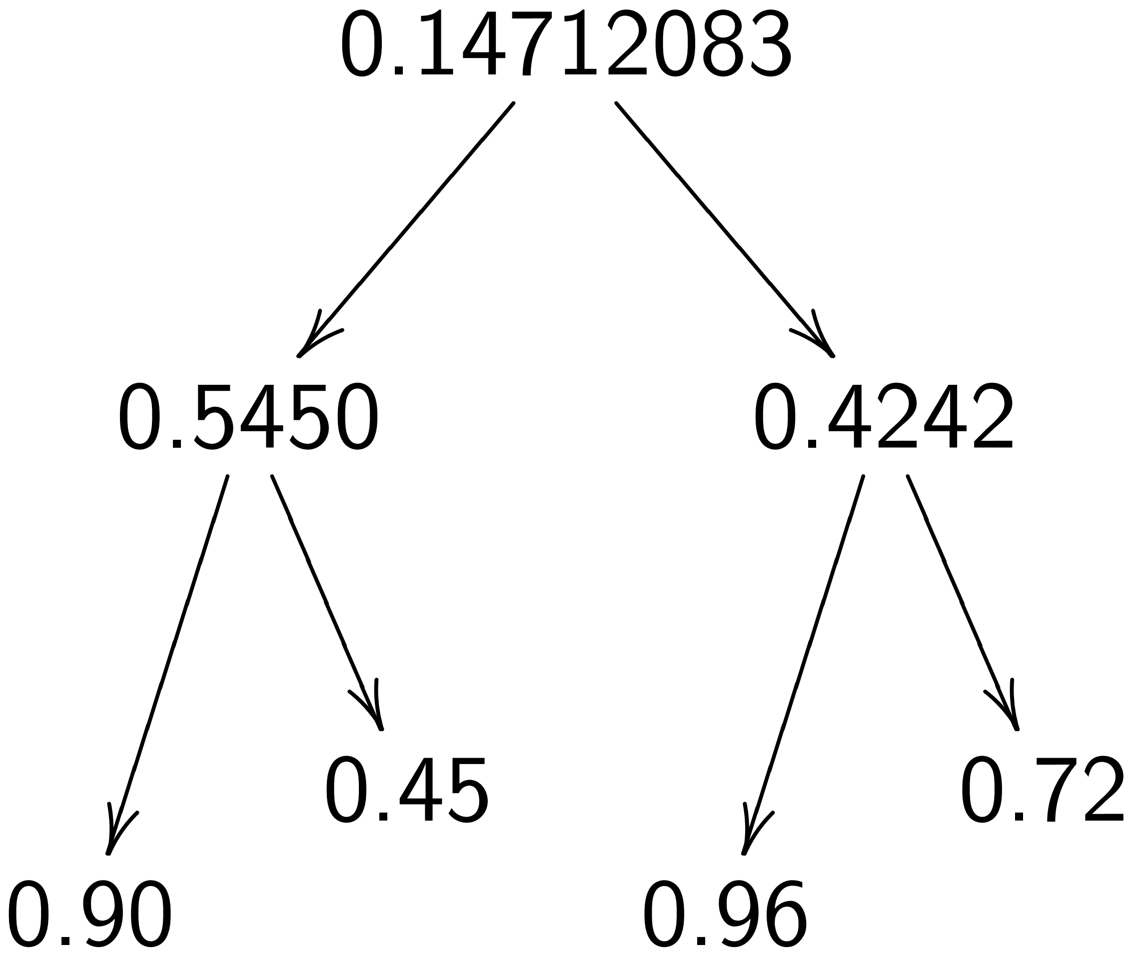
$\frac{P}{n_2} \bmod 1$

$\frac{P}{n_3} \bmod 1$

$\frac{P}{n_4} \bmod 1$

Represent each $P/\cdots \bmod 1$ as a nearby floating-point number.

e.g. Scaled remainder tree for $P = 8675309$, $n_1 = 10$, $n_2 = 20$, $n_3 = 30$, $n_4 = 40$:



0.14712083

0.5450

0.4242

0.90

0.45

0.96

0.72

Integer gcd

$n(\lg n)^{2+o(1)}$ bit ops:
Given $a, b \in \mathbf{Z}$, compute $\gcd\{a, b\}$.
(1971 Schönhage;
core idea: 1938 Lehmer;
$n(\lg n)^{5+o(1)}$: 1971 Knuth)

Better bound when a
is much larger than b:
$\le n(\lg n)^{1+o(1)} + m(\lg m)^{2+o(1)}$
where m is number of bits in b.
Idea: $\gcd\{b, a \bmod b\}$.

Modular squaring ad nauseam

$n(\lg n)^{2+o(1)}$ bit ops:
Given $a, b \in \mathbf{Z}$ with $a \neq 0$,
compute $\gcd\{a, b^\infty\}$.

Algorithm:
Compute $b \bmod a$,
$b^2 \bmod a = (b \bmod a)^2 \bmod a$,
$b^4 \bmod a = (b^2 \bmod a)^2 \bmod a$,
$b^8 \bmod a = (b^4 \bmod a)^2 \bmod a$,
etc., until b^{2^k} with $2^k \geq n$.
Then compute $\gcd\{a, b^\infty\}$
as $\gcd\left\{a, b^{2^k} \bmod a\right\}$.

Factoring a, b into coprimes

Given $a, b \in \mathbf{Z}$, $a \geq b \geq 2$:
Compute $a_0 = a$; $g_0 = \gcd\{a_0, b\}$;
$a_1 = a_0/g_0$; $g_1 = \gcd\{a_1, g_0^2\}$;
$a_2 = a_1/g_1$; $g_2 = \gcd\{a_2, g_1^2\}$;
etc., stopping when $g_k = 1$.

How long does this take?

e.g. $a = 2^{100}3^{100}$, $b = 2^{137}3^{13}$:
$a_0 = 2^{100}3^{100}$, $g_0 = 2^{100}3^{13}$,
$a_1 = 3^{87}$, $g_1 = 3^{26}$,
$a_2 = 3^{61}$, $g_2 = 3^{52}$,
$a_3 = 3^{9}$, $g_3 = 3^{9}$,
$a_4 = 1$, $g_4 = 1$.

Consider a prime p.

Define $e = \operatorname{ord}_p a$: i.e.,

p^e divides a but p^{e+1} doesn't.

Define $f = \operatorname{ord}_p b$.

$e >$		f	$3f$	$7f$
$e \le$	f	$3f$	$7f$	$15f$
$\operatorname{ord}_p a_0$	e	e	e	e
$\operatorname{ord}_p g_0$	e	f	f	f
$\operatorname{ord}_p a_1$	0	$e - f$	$e - f$	$e - f$
$\operatorname{ord}_p g_1$	0	$e - f$	$2f$	$2f$
$\operatorname{ord}_p a_2$	0	0	$e - 3f$	$e - 3f$
$\operatorname{ord}_p g_2$	0	0	$e - 3f$	$4f$
$\operatorname{ord}_p a_3$	0	0	0	$e - 7f$
$\operatorname{ord}_p g_3$	0	0	0	$e - 7f$

$2^e \le p^e \le a < 2^n$ so $e < n$.
Thus $g_k = 1$ for $k = \lceil \lg n \rceil$.

Ops to divide a_i by g_i,
square g_i, and compute
$\gcd\{a_{i+1}, g_i^2\}$:
$\le n(\lg n)^{1+o(1)} + m_i(\lg m_i)^{2+o(1)}$
where m_i is number of bits in g_i.

$a = a_k \prod g_i$ so $\sum m_i \le O(n)$.

Total ops for all a_i, g_i:
$\le n(\lg n)^{2+o(1)}$.

Next step: Compute
$x_0 = g_0/\mathrm{gcd}\{g_0, g_1^\infty\}$,
$x_1 = g_1/\mathrm{gcd}\{g_1, g_2^\infty\}$,
etc.

Write $m_i' = m_i + m_{i+1}$.
Ops $\le \sum m_i' (\lg m_i')^{2+o(1)}$
$\le n(\lg n)^{2+o(1)}$.

e.g. $a = 2^{100}3^{100}$, $b = 2^{137}3^{13}$:
$g_0 = 2^{100}3^{13}$, $g_1 = 3^{26}$,
$g_2 = 3^{52}$, $g_3 = 3^9$, $g_4 = 1$;
$x_0 = 2^{100}$, $x_1 = 1$, $x_2 = 1$,
$x_3 = 3^9$.

Compute

$b \bmod g_1, b \bmod g_2, \ldots$

using a remainder tree; and

$y_0{=}\gcd\left\{b, x_0^\infty\right\}$,

$y_1{=}\gcd\left\{g_0, x_1^\infty\right\}$,

$y_2{=}\gcd\left\{\gcd\{b \bmod g_1, g_1\}, x_2^\infty\right\}$,

$y_3{=}\gcd\left\{\gcd\{b \bmod g_2, g_2\}, x_3^\infty\right\}$,

$y_4{=}\gcd\left\{\gcd\{b \bmod g_3, g_3\}, x_4^\infty\right\}$,

etc.

$n(\lg n)^{2+o(1)}$ bit ops.

e.g. $a = 2^{100}3^{100}$, $b = 2^{137}3^{13}$:

$x_0 = 2^{100}$, $x_1{=}1$, $x_2{=}1$, $x_3 = 3^9$;

$y_0 = 2^{137}$, $y_1{=}1$, $y_2{=}1$, $y_3 = 3^{13}$.

Now $\text{cb}\{a, b\}$ is disjoint union of $\text{cb}\{x_0, y_0/x_0\}$, $\text{cb}\{x_1, y_1\}, \text{cb}\{x_2, y_2\}, \ldots,$ $\{a_k\} - \{1\}$, $\{b/\text{gcd}\{b, a^\infty\}\} - \{1\}$.

e.g. $\text{cb}\left\{2^{100}3^{100}, 2^{137}3^{13}\right\} = \text{cb}\left\{2^{100}, 2^{37}\right\} \cup \text{cb}\left\{3^9, 3^{13}\right\}$.

Recursion multiplies total ops by a constant factor, since product $x_0(y_0/x_0)x_1y_1x_2y_2\cdots$ is at most $ab/a^{1/3} \le (ab)^{5/6}$.

$n(\lg n)^{2+o(1)}$ bit ops to compute $\text{cb}\{a, b\}$.

What about $\operatorname{cb} S$ for $\#S \ge 3$?

$n(\lg n)^{2+o(1)}$ if $\lg \#P \in (\lg n)^{o(1)}$:
multiset S, coprime set P
$\mapsto \gcd\{s, p^\infty\}$
for each $s \in S$, each $p \in P$.

$n(\lg n)^{2+o(1)}$:
a, coprime set $Q \mapsto \operatorname{cb}(\{a\} \cup Q)$.

More complicated than the case
$Q = \{b\}$ but same basic ideas.

$n(\lg n)^{3+o(1)}$:
coprime set P, coprime set Q
$\mapsto \operatorname{cb}(P \cup Q)$.

Idea of $\text{cb}(P \cup Q)$ algorithm:
Replace Q with $\text{cb}(\{a\} \cup Q)$
for each $a \in P$ successively.
But that's too slow if $\#P$ is large,
so first replace P with P' having
$\#P' \in O(\lg n)$ and $\text{cb}\, P' = \text{cb}\, P$.
e.g. $p_0 p_1 p_4 p_5 p_8 p_9 \cdots \in P'$.

$n(\lg n)^{4+o(1)}$: $S \mapsto \text{cb}\, S$.

Idea of $\text{cb}(P \cup Q)$ algorithm:
Replace Q with $\text{cb}(\{a\} \cup Q)$
for each $a \in P$ successively.
But that's too slow if $\#P$ is large,
so first replace P with P' having
$\#P' \in O(\lg n)$ and $\text{cb}\, P' = \text{cb}\, P$.
e.g. $p_0 p_1 p_4 p_5 p_8 p_9 \cdots \in P'$.

$n(\lg n)^{4+o(1)}$: $S \mapsto \text{cb}\, S$.

Having computed $Q = \text{cb}\, S$,
how to factor S over Q?

Idea of cb$(P \cup Q)$ algorithm:
Replace Q with cb$(\{a\} \cup Q)$
for each $a \in P$ successively.
But that's too slow if $\#P$ is large,
so first replace P with P' having
$\#P' \in O(\lg n)$ and cb $P' =$ cb P.
e.g. $p_0 p_1 p_4 p_5 p_8 p_9 \cdots \in P'$.

$n(\lg n)^{4+o(1)}$: $S \mapsto \operatorname{cb} S$.

Having computed $Q = \operatorname{cb} S$,
how to factor S over Q?

More generally,
how to factor S over Q
when Q is any coprime set?

Coprime factors, union

$n(\lg n)^{2+o(1)}$ bit ops:
Given $x_1, x_2, \ldots, x_k \in \mathbf{Z}$ and finite set $Q \subseteq \mathbf{Z} - \{0\}$, compute $\{p \in Q : x_1 x_2 \cdots x_k \bmod p = 0\}$.

Special case that p is prime or that $Q = \text{cb}\{x_1, \ldots, x_k\}$: see whether p divides any of $x_1, x_2, \ldots, x_k$.

Algorithm:

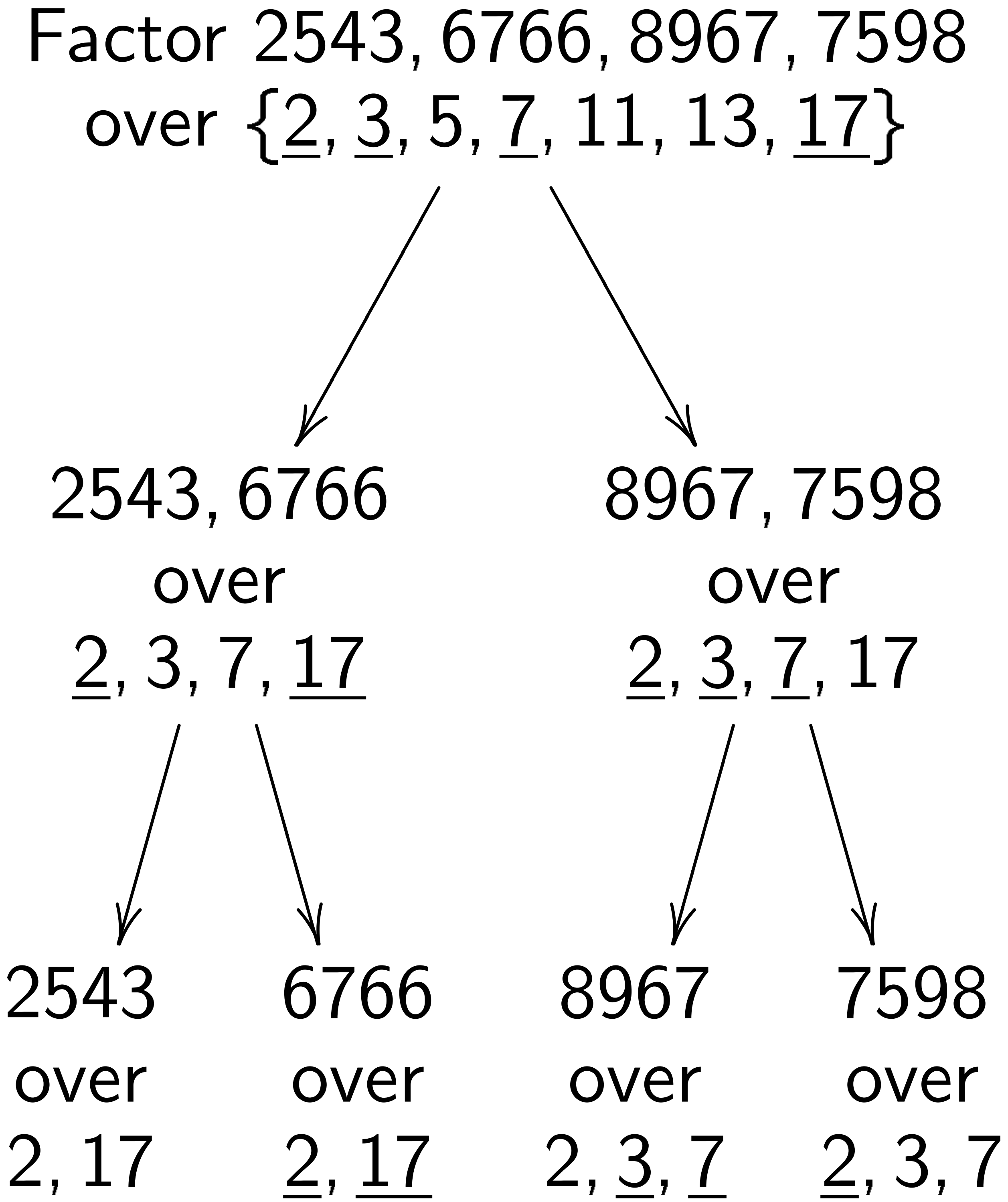
1. Use a product tree to compute $r = x_1 x_2 \cdots x_k$.
2. Use a remainder tree to see which $p \in Q$ divide r.

Coprime factors, separately

$n(\lg n)^{3+o(1)}$ bit ops:
Given $x_1, x_2, \ldots, x_k \in \mathbf{Z}$ and
finite coprime set Q,
compute $\{p \in Q : x_1 \bmod p = 0\}$,
$\ldots$, $\{p \in Q : x_k \bmod p = 0\}$.
(2000 Bernstein)

Algorithm for $k \geq 1$:

```
1. Replace $Q$ with
   $\{p \in Q : x_1 \cdots x_k \bmod p = 0\}$.
2. If $k = 1$, print $Q$ and stop.
3. Recurse on $x_1, \ldots, x_{\lceil k/2 \rceil}, Q$.
4. Recurse on $x_{\lceil k/2 \rceil + 1}, \ldots, x_k, Q$.
```



Factor 2543, 6766, 8967, 7598
over $\{\underline{2}, \underline{3}, 5, \underline{7}, 11, 13, \underline{17}\}$

2543, 6766
over
$\underline{2}, 3, 7, \underline{17}$

8967, 7598
over
$\underline{2}, \underline{3}, \underline{7}, 17$

2543
over
$2, 17$

6766
over
$\underline{2}, \underline{17}$

8967
over
$2, \underline{3}, \underline{7}$

7598
over
$\underline{2}, 3, 7$

Each level: $\leq n(\lg n)^{0+o(1)}$ bits.

Exponents of a coprime

$n(\lg n)^{2+o(1)}$ bit ops:
Given nonzero $p, x \in \mathbf{Z}$,
find $e, p^e, x/p^e$ with maximal e.

Algorithm:

```
1. If $x \bmod p \ne 0$:
   Print $0, 1, x$ and stop.
2. Find $f, (p^2)^f, r = (x/p)/(p^2)^f$
   with maximal $f$.
3. If $r \bmod p = 0$: Print
   $2f + 2, (p^2)^f p^2, r/p$ and stop.
4. Print $2f + 1, (p^2)^f p, r$.
```

Exponents of coprimes

$n(\lg n)^{3+o(1)}$ bit ops:
Given finite coprime set Q and nonzero $x \in \mathbf{Z}$, find maximal $e, \prod_{p \in Q} p^{e(p)}, x/\prod_{p \in Q} p^{e(p)}$.

Algorithm:

1. Replace Q with $\{p \in Q : x \bmod p = 0\}$.
2. Find maximal f, s, r with $s=\prod (p^2)^{f(p^2)}$, $r=(x/\prod p)/s$.
3. Find $T=\{p \in Q : r \bmod p=0\}$.
4. Print $e, s\prod_{p \in T} p, r/\prod_{p \in T} p$ where $e(p) = 2f(p^2) + [p \in T]$.

Smooth parts, old approach

$n(\lg n)^{3+o(1)}$ bit ops:
Given nonzero $x_1, x_2, \ldots, x_k \in \mathbf{Z}$
and finite coprime set Q,
compute Q-smooth part of x_1,
Q-smooth part of x_2, ...,
Q-smooth part of x_k.

Q-smooth means product of
powers of elements of Q.

Q-smooth part means
largest Q-smooth divisor.
In particular, see which of
$x_1, x_2, \ldots, x_k$ are smooth.

Algorithm:

1. Find $Q_1 = \{p : x_1 \bmod p = 0\}$,
   $\ldots, Q_k = \{p : x_k \bmod p = 0\}$.
2. For each i separately:
   Find maximal e, s, r with
   $s = \prod_{p \in Q_i} p^{e(p)}$, $r = x_i/s$.
   Print s.

e.g. factoring

$2543, 6766, 8967, 7598$

over $\{2, 3, 5, 7, 11, 13, 17\}$:

2543 over $\{\}$, smooth part 1;

6766 $\{2, 17\}$, smooth part 34;

8967 $\{3, 7\}$, smooth part 147;

7598 $\{2\}$, smooth part 2.

Smooth parts, better approach

Given nonzero $x_1, x_2, \ldots, x_k \in \mathbf{Z}$ and finite coprime set Q: Typically $n(\lg n)^{2+o(1)}$ bit ops to obtain smooth parts of x's. (2004 Franke Kleinjung Morain Wirth, in ECPP context)

Algorithm:
Compute $r = \prod_{p \in Q} p$ and then $r \bmod x_1, \ldots, r \bmod x_k$.
For each i separately:
Replace x_i by $x_i/\gcd\{x_i, r \bmod x_i\}$ repeatedly until gcd is 1.

Slight variant (2004 Bernstein): Always $n(\lg n)^{2+o(1)}$ bit ops.

Compute smooth part of x_i as $\gcd\{x_i, (r \bmod x_i)^{2^c} \bmod x_i\}$ where $c = \lceil \lg \lg x_i \rceil$.

Or, to see if x_i is smooth, see if $(r \bmod x_i)^{2^k} \bmod x_i = 0$.

Slight variant (2004 Bernstein):
Always $n(\lg n)^{2+o(1)}$ bit ops.

Compute smooth part of x_i as
$\gcd\{x_i, (r \bmod x_i)^{2^c} \bmod x_i\}$
where $c = \lceil \lg \lg x_i \rceil$.

Or, to see if x_i is smooth,
see if $(r \bmod x_i)^{2^k} \bmod x_i = 0$.

Minor problem: These algorithms don't factor the smooth numbers.

Slight variant (2004 Bernstein):
Always $n(\lg n)^{2+o(1)}$ bit ops.

Compute smooth part of x_i as
$\gcd\{x_i, (r \bmod x_i)^{2^c} \bmod x_i\}$
where $c = \lceil \lg \lg x_i \rceil$.

Or, to see if x_i is smooth,
see if $(r \bmod x_i)^{2^k} \bmod x_i = 0$.

Minor problem: These algorithms don't factor the smooth numbers.

Solution: Feed smooth numbers to the old algorithm.
Normally very few smooth numbers, so this is very fast.

Smoothness without hermits

Typical application: NFS.
Want to find nontrivial subset of $x_1, x_2, \ldots$ with square product.
Q is set of small primes.

Don't want *all* smooth numbers.
Want smooth numbers only if they are built from primes that divide the *other* numbers.

Directly find those numbers, without ever looking at Q.

Compute $r = x_1 x_2 \cdots x_k$.
Compute $(r/x_1) \bmod x_1$, . . . ,
$(r/x_k) \bmod x_k$.
For each i separately: see if
$((r/x_i) \bmod x_i)^{2^c} \bmod x_i = 0$
where $c = \lceil \lg \lg x_i \rceil$.

Finds x_i iff all primes in x_i
are divisors of other x's.
$n(\lg n)^{2+o(1)}$ bit ops.
(2004 Bernstein)

Compute $(r/x_1) \bmod x_1$, . . . ,
$(r/x_k) \bmod x_k$ by computing
$r \bmod x_1^2, \ldots, r \bmod x_k^2$.
(1972 Moenck Borodin)

Variant:

Compute $r = x_1 x_2 \cdots x_k$.
Compute $(r/x_1) \bmod x_1$, . . . ,
$(r/x_k) \bmod x_k$.
For each i separately: see if
$\gcd\{(r/x_i) \bmod x_i, x_i\} > 1$.

Variant:

Compute $r = x_1 x_2 \cdots x_k$.
Compute $(r/x_1) \bmod x_1, \ldots,$ $(r/x_k) \bmod x_k$.
For each i separately: see if $\gcd\{(r/x_i) \bmod x_i, x_i\} > 1$.

Finds x_i iff *at least* one prime in x_i is a divisor of other x's.

Variant:

Compute $r = x_1 x_2 \cdots x_k$.
Compute $(r/x_1) \bmod x_1$, . . . ,
$(r/x_k) \bmod x_k$.
For each i separately: see if
$\gcd\{(r/x_i) \bmod x_i, x_i\} > 1$.

Finds x_i iff *at least* one prime in
x_i is a divisor of other x's.

This is a good algorithm
for checking RSA keys
to find shared primes.