Factorization myths

D. J. Bernstein

Thanks to:
University of Illinois at Chicago
NSF DMS–0140542
Alfred P. Sloan Foundation

Sieving c and $611 + c$ for small c:

1	
2	2
3	3
4	2 2
5	5
6	2 3
7	7
8	2 2 2
9	3 3
10	2 5
11	
12	2 2 3
13	
14	2 7
15	3 5
16	2 2 2 2
17	
18	2 3 3
19	
20	2 2 5

612	2 2 3 3
613	
614	2
615	3 5
616	2 2 2 7
617	
618	2 3
619	
620	2 2 5
621	3 3 3
622	2
623	7
624	2 2 2 2 3
625	5 5 5 5
626	2
627	3
628	2 2
629	
630	2 3 3 5 7
631	

etc.

Factoring 611 by the $\mathbf{Q}$ sieve:
Have complete factorization of $c(611 + c)$ for several c's.

$14 \cdot 625 = 2^1 3^0 5^4 7^1$.
$64 \cdot 675 = 2^6 3^3 5^2 7^0$.
$75 \cdot 686 = 2^1 3^1 5^2 7^3$.

$14 \cdot 64 \cdot 75 \cdot 625 \cdot 675 \cdot 686$
$= 2^8 3^4 5^8 7^4 = (2^4 3^2 5^4 7^2)^2$.

$\gcd\left\{14 \cdot 64 \cdot 75 - 2^4 3^2 5^4 7^2, 611\right\}$
$= 47$.

$611 = 47 \cdot 13$.

Myth #1: We want to
find all relations,
so we need to know exactly
which inputs are smooth.

"Inputs": $1, 2, 3, \ldots$; $612, 613, \ldots$.
"Smooth": no prime divisors > 10.
"Relation": smooth $c(611 + c)$.

e.g. 1994 Golliver Lenstra McCurley:
give up on annoying inputs? no—
"some relations get lost which
is something we try to avoid."

Reality: We want to
minimize price-performance ratio.

Inputs are potentially useful if
we can completely factor them.
Particularly useful if
largest prime factor is small.

Price is different: low if
many tiny prime factors and
second-largest prime factor is small.

Best to abort high-priced inputs,
including *most* of the useful inputs.

Myth #2: Sieving is the ultimate test for fully factored inputs.

Small-factor tests in CFRAC:
trial division, rho, ECM, et al.
All obsolete in context of
Q sieve, quadratic sieve, etc.

e.g. 2000 Lenstra:
sieving "much faster" than ECM.

Inputs are sieveable;
sieving is fast; so sieve.
Simple algorithm.
Sole parameter: largest prime.

Reality: Much more complicated.
Sieving is *not* the best algorithm;
random access to big memory is slow.
Other tests are *not* obsolete.
Can gain speed by
combining sieving with other tests.

Sieve up to (largest prime)$^\theta$;
abort if not too promising;
then use second small-factor test.
Parameters: largest prime;
θ; sieve length; second test.

e.g. 1994 Golliver Lenstra McCurley:
sieve using primes up to 2^{21};
abort unfactored parts above 2^{60};
then use SQUFOF and ECM
to find primes up to 2^{30}.
Here $\theta = 21/30 = 0.7$.

But they said no aborts! Huh?
Pointless change in perspective:
they view their relations as
superset of 2^{21}-smooth
rather than subset of 2^{30}-smooth.

Myth #3: The second small-factor test (rho, SQUFOF, ECM, etc.) is not a bottleneck.

e.g. 1996 Boender, te Riele: "sieving takes more than 85% of the total computing time."

Reality: If second test
isn't taking much time,
should abort fewer inputs.
Balance time for second test
with time for sieving.

Total time after balancing:
roughly $RS^{\theta}T^{1-\theta}$
where R is smoothness ratio,
S is sieve time per number,
T is second-test time per number.

Why $S^\theta T^{1-\theta}$?

1982 Pomerance, analyzing aborts for trial division and rho:

Aborting at (largest prime)$^\theta$
reduces #{inputs}
by a certain factor A
and reduces #{smooth inputs}
by $A^{1-\theta+o(1)}$,
in typical parameter ranges.

Balancing means $S \approx (1/A)T$
so $SA^{1-\theta} \approx S^\theta T^{1-\theta}$.

cr.yp.to/bib/entries.html#1982/pomerance

Better analysis and optimization:
use tight bounds on probability
of smoothness (2002 Bernstein);
use measurements of S
for various sieve lengths
in L1 cache, L2 cache, DRAM, disk;
account for NFS input sizes;
balance NFS input sizes across
multiple lattices (1995 Bernstein);
etc.

`cr.yp.to/papers.html#mlnfs`
`cr.yp.to/papers.html#psi`

Myth #4: ECM is the ultimate non-sieving small-factor test.

e.g. 2002 Leyland Lenstra Dodson used ECM to find primes $< 2^{30}$ in numbers $< 2^{90}$.

Reality: On these computers, for large factorizations, batch small-factor tests are faster.

`cr.yp.to/papers.html#sf`
`cr.yp.to/papers.html#smoothparts`

Given set P of primes
and sequence N of numbers,
can factor N over P
in time $y(\lg y)^{3+o(1)}$
where y is number of input bits.
(2000 Bernstein)

Variant (2004 Franke Kleinjung
Morain Wirth, in ECPP context):
Identify P-smooth elements of N
in time usually $y(\lg y)^{2+o(1)}$.

Slight variant (2004 Bernstein):
time always $y(\lg y)^{2+o(1)}$.

Myth #5: Must prespecify primes: e.g., all primes below 2^{30}.

Find many inputs that
fully factor over those primes;
weed out non-repeated primes.

Have to keep 2^{30} small
to speed up small-factor tests,
limit number of inputs found,
avoid processing huge number
of non-repeated primes.

Reality: Can quickly identify
inputs built from
primes that divide other inputs,
without prespecifying primes.
(2004 Bernstein)

Unlike the other algorithms,
doesn't allow split into
moderate-size independent batches;
communication costs comparable
to linear algebra.

Maybe benefit outweighs cost.

What's the algorithm?

Inputs $x_1, x_2, \ldots$.
Compute $y = x_1 x_2 \cdots$.
Compute $(y/x_1) \bmod x_1$,
$(y/x_2) \bmod x_2$, etc.
Output x_j if $(y/x_j)^{\text{big}} \bmod x_j = 0$.
(In practice can take big $= 1$;
anyway, not a bottleneck.)

Can iterate algorithm,
then factor into coprimes.

`cr.yp.to/papers.html#dcba`
`cr.yp.to/papers.html#smoothparts`
`cr.yp.to/papers.html#multapps`

Why is this so fast?

Can compute y quickly
with a product tree. (standard)

To compute $(y/x_j) \bmod x_j$:
compute $y \bmod x_1^2, y \bmod x_2^2, \ldots$
with a remainder tree;
divide $y \bmod x_j^2$ by x_j.
(1972 Moenck Borodin; alternative:
1997 Bürgisser Clausen Shokrollahi)

Many constant-factor speedups:
FFT doubling (2004 Kramer) et al.

Myth #6: NFS involves two sieves, "rational" and "algebraic."

Sieve c and sieve $611 + c$.

e.g. 1993 Lenstra Lenstra Manasse Pollard: second sieve is "much faster" than the alternative.

e.g. 1993 Buhler Lenstra Pomerance: Coppersmith's variant not "practical."

Reality: One sieve is enough.
Identify smooth values $611 + c$;
then check smoothness of c's.
Or vice versa.

Have time to check other
functions of c. (1993 Coppersmith)

Have time to check for
very large primes in c's.

All quite practical.
Obviously beneficial as soon as
smoothness chance $< S/T$.
Many parameters to optimize.

Myth #7: The direct square-root method—computing $14 \cdot 64 \cdot 75 \cdot 625 \cdot 675 \cdot 686$, then $\sqrt{14 \cdot 64 \cdot 75 \cdot 625 \cdot 675 \cdot 686}$—is a bottleneck.

Must use prime factorizations.
(generalization to number fields:
1993 Buhler Lenstra Pomerance,
1994 Montgomery, 1998 Nguyen)

e.g. 2001 Crandall Pomerance:
this is of "great consequence
for the overall running time."

Reality: The direct square-root method is not a bottleneck.

Standard square-root algorithms,
using fast multiplication,
take time only $y^{1+o(1)}$
where y is prime bound.
Smaller exponent than,
e.g., linear algebra.

No need to bother
using prime factorizations.

Timings on previous slides are
for a conventional computer:
a general-purpose processor
attached to a large memory.
(1945 von Neumann)

Myth #8: We want to minimize
time on a conventional computer.
This minimizes real time.

Okay, okay, parallel computers
aren't conventional computers,
but p processors achieve
at most a p-fold speedup.

Reality: We want to minimize price-performance ratio.

Conventional computers do *not* minimize price-performance ratio. Can often split a conventional computer into two parallel computers *each of half the size*, with mild communication costs.

A mesh architecture achieves smaller cost *exponents* than a von Neumann architecture.

`cr.yp.to/papers.html#nfscircuit`
`cr.yp.to/nfscircuit.html`

VLSI literature makes this point
for a wide variety of computations.

Consider, e.g.,
multiplying two n-bit integers.

Time $\Theta(n \lg n \lg \lg n)$
on a conventional computer
with $\Theta(n)$ bits of memory.
(1971 Schönhage Strassen,
using FFT)

Knuth: “we leave the domain of conventional computer programming. . .”

Time $\Theta(n)$
on a 1-dimensional mesh
of size $\Theta(n)$.
(1965 Atrubin, elementary)

Time $n^{0.5+o(1)}$
on a 2-dimensional mesh
of size $\Theta(n)$.
(1983 Preparata, using FFT)

Similar speedups for factoring:

Want to factor n. Write
$L = \exp((\log n)^{1/3}(\log\log n)^{2/3})$.

NFS takes time $L^{1.901...+o(1)}$
on a conventional computer
of size $L^{0.950...+o(1)}$.
(1993 Coppersmith)

Can perform the same computation
in time $L^{1.426...+o(1)}$
on a 2-dimensional mesh
of size $L^{0.950...+o(1)}$.
(2001 Bernstein)

New parameters: Time $L^{2.012...+o(1)}$
on a conventional computer
of size $L^{0.748...+o(1)}$.
(2002 Pomerance)

Time $L^{1.185...+o(1)}$
on a 2-dimensional mesh
of size $L^{0.790...+o(1)}$.
(2001 Bernstein)

NFS cost (price-performance ratio)
has much lower exponent
on a 2-dimensional mesh
than on a conventional computer.

Myth #9: Mesh architectures simply make everything faster.

We can continue
designing algorithms
and writing programs
for conventional computers,
and then put them on
mesh computers to reduce cost.

e.g. Preparata multiplication mesh is straightforward implementation of traditional FFT-based algorithm.

Reality: Optimizing cost
on a 2-dimensional mesh
is very different from
optimizing time
on a conventional computer.

Example: ECM vs. my batch test.
Time on von Neumann architecture:
batch test is better.
Cost on mesh architecture:
ECM is better;
early-abort ECM is even better.

Current algorithm-analysis culture—
talk all about time;
maybe mention machine size,
but only as a secondary issue—
will eventually be considered
shortsighted, archaic, obsolete.

Yes, it's fun, but it's doomed!

Have to redesign algorithms
and rewrite programs
from the ground up,
focusing on cost rather than time.

A computational number theorist's adventures in mesh programming:

"Verilog": "circuit design" language, not hard to learn.

(Alternative to Verilog: VHDL.
Skip VHDL unless you like Ada.)

"Icarus Verilog": free software to compile and run Verilog programs ("simulate circuits")
on, e.g., Pentium running Linux.
Very slow, of course.

"FPGA": mesh device that can run moderately large Verilog programs at reasonable speed.

"Manual place and route": equivalent of assembly-language programming, when compiler isn't smart enough to use mesh sensibly.

"ASIC": chip that runs one Verilog program even more quickly. Expensive except in high volumes.

Several companies writing
higher-level programming tools:
SRC, StarBridge, OctigaBay, et al.
Willing to sacrifice
quite noticeable constant factor
for the sake of easy programming.

I'm doing this too.
Goal: Make it very easy
to build large meshes
for zero-communication operations
and for sorting, multiplication, etc.

Myth #10: MPQS beats ECM for finding huge factors; conjecturally $(\lg n)^{1+o(1)}$ faster.

ECM wants to find one y-smooth number near $\sqrt{n}$.
Time $y(\lg n)^{1+o(1)}$ per number.

MPQS wants to find $y(\lg y)^{-1+o(1)}$ y-smooth numbers below $y\sqrt{n}$; smoothness chance is lowered by $((\lg \sqrt{n})/\lg y)^{1+o(1)} = (\lg y)^{1+o(1)}$.
Time $(\lg n)^{o(1)}$ per number.

Reality: ECM beats MPQS
on mesh architectures
for all sufficiently large inputs.

Linear algebra is costly.
Reduce y to compensate.
Best MPQS cost still has
larger *exponent* than ECM cost.

My first public circuits
will be ECM circuits.
Note to chip designers:
Use Schönhage-Strassen!
Avoid carries; align roots.