High-speed cryptography, part 2: more elliptic-curve formulas; field arithmetic

Daniel J. Bernstein
University of Illinois at Chicago &
Technische Universiteit Eindhoven

Speed-oriented Jacobian standards

2000 IEEE "Std 1363"
uses Weierstrass curves
in Jacobian coordinates
to "provide the fastest
arithmetic on elliptic curves."
Also specifies a method of
choosing curves $y^2 = x^3 - 3x + b$.

2000 NIST "FIPS 186–2"
standardizes five such curves.

2005 NSA "Suite B" recommends
two of the NIST curves as
the only public-key cryptosystems
for U.S. government use.

Projective for Weierstrass

1986 Chudnovsky–Chudnovsky:
Speed up ADD by switching from
$(X/Z^2, Y/Z^3)$ to $(X/Z, Y/Z)$.
$7\mathbf{M} + 3\mathbf{S}$ for DBL if $a = -3$.
$12\mathbf{M} + 2\mathbf{S}$ for ADD.
$12\mathbf{M} + 2\mathbf{S}$ for reADD.

Option has been mostly ignored:
DBL dominates in ECDH etc.
But ADD dominates in
some applications: e.g.,
batch signature verification.

Montgomery curves

1987 Montgomery:

Use $by^2 = x^3 + ax^2 + x$.
Choose small $(a+2)/4$.

$2(x_2, y_2) = (x_4, y_4)$
$$\Rightarrow x_4 = \frac{(x_2^2 - 1)^2}{4x_2(x_2^2 + ax_2 + 1)}.$$

$(x_3, y_3) - (x_2, y_2) = (x_1, y_1)$,
$(x_3, y_3) + (x_2, y_2) = (x_5, y_5)$
$$\Rightarrow x_5 = \frac{(x_2 x_3 - 1)^2}{x_1(x_2 - x_3)^2}.$$

Represent (x, y)
as $(X{:}Z)$ satisfying $x = X/Z$.

$B = (X_2 + Z_2)^2$,
$C = (X_2 - Z_2)^2$,
$D = B - C$, $X_4 = B \cdot C$,
$Z_4 = D \cdot (C + D(a+2)/4) \Rightarrow$
$2(X_2{:}Z_2) = (X_4{:}Z_4)$.

$(X_3{:}Z_3) - (X_2{:}Z_2) = (X_1{:}Z_1)$,
$E = (X_3 - Z_3) \cdot (X_2 + Z_2)$,
$F = (X_3 + Z_3) \cdot (X_2 - Z_2)$,
$X_5 = Z_1 \cdot (E + F)^2$,
$Z_5 = X_1 \cdot (E - F)^2 \Rightarrow$
$(X_3{:}Z_3) + (X_2{:}Z_2) = (X_5{:}Z_5)$.

This representation
does not allow ADD but it allows
DADD, "differential addition":
$Q, R, Q - R \mapsto Q + R$.

e.g. $2P, P, P \mapsto 3P$.
e.g. $3P, 2P, P \mapsto 5P$.
e.g. $6P, 5P, P \mapsto 11P$.

$2\mathbf{M} + 2\mathbf{S} + 1\mathbf{D}$ for DBL.
$4\mathbf{M} + 2\mathbf{S}$ for DADD.
Save $1\mathbf{M}$ if $Z_1 = 1$.

Easily compute $n(X_1 : Z_1)$ using
$\approx \lg n$ DBL, $\approx \lg n$ DADD.
Almost as fast as Edwards nP.
Relatively slow for $mP + nQ$ etc.

Doubling-oriented curves

2006 Doche–Icart–Kohel:

Use $y^2 = x^3 + ax^2 + 16ax$.
Choose small a.

Use $(X : Y : Z : Z^2)$
to represent $(X/Z, Y/Z^2)$.

$3\mathbf{M} + 4\mathbf{S} + 2\mathbf{D}$ for DBL.
How? Factor DBL as $\hat{\varphi}(\varphi)$
where φ is a 2-isogeny.

2007 Bernstein–Lange:
$2\mathbf{M} + 5\mathbf{S} + 2\mathbf{D}$ for DBL
on the same curves.

$12\mathbf{M} + 5\mathbf{S} + 1\mathbf{D}$ for ADD.
Slower ADD than other systems, typically outweighing benefit of the very fast DBL.

But isogenies are useful.
Example, 2005 Gaudry: fast DBL+DADD on Jacobians of genus-2 hyperelliptic curves, using similar factorization.

Tricky but potentially helpful: tripling-oriented curves (see 2006 Doche–Icart–Kohel), double-base chains, . . .

Hessian curves

Credited to Sylvester
by 1986 Chudnovsky–Chudnovsky:

$(X : Y : Z)$ represent $(X/Z, Y/Z)$
on $x^3 + y^3 + 1 = 3dxy$.

12**M** for ADD:
$X_3 = Y_1X_2 \cdot Y_1Z_2 - Z_1Y_2 \cdot X_1Y_2$,
$Y_3 = X_1Z_2 \cdot X_1Y_2 - Y_1X_2 \cdot Z_1X_2$,
$Z_3 = Z_1Y_2 \cdot Z_1X_2 - X_1Z_2 \cdot Y_1Z_2$.

6**M** + 3**S** for DBL.

2001 Joye–Quisquater:

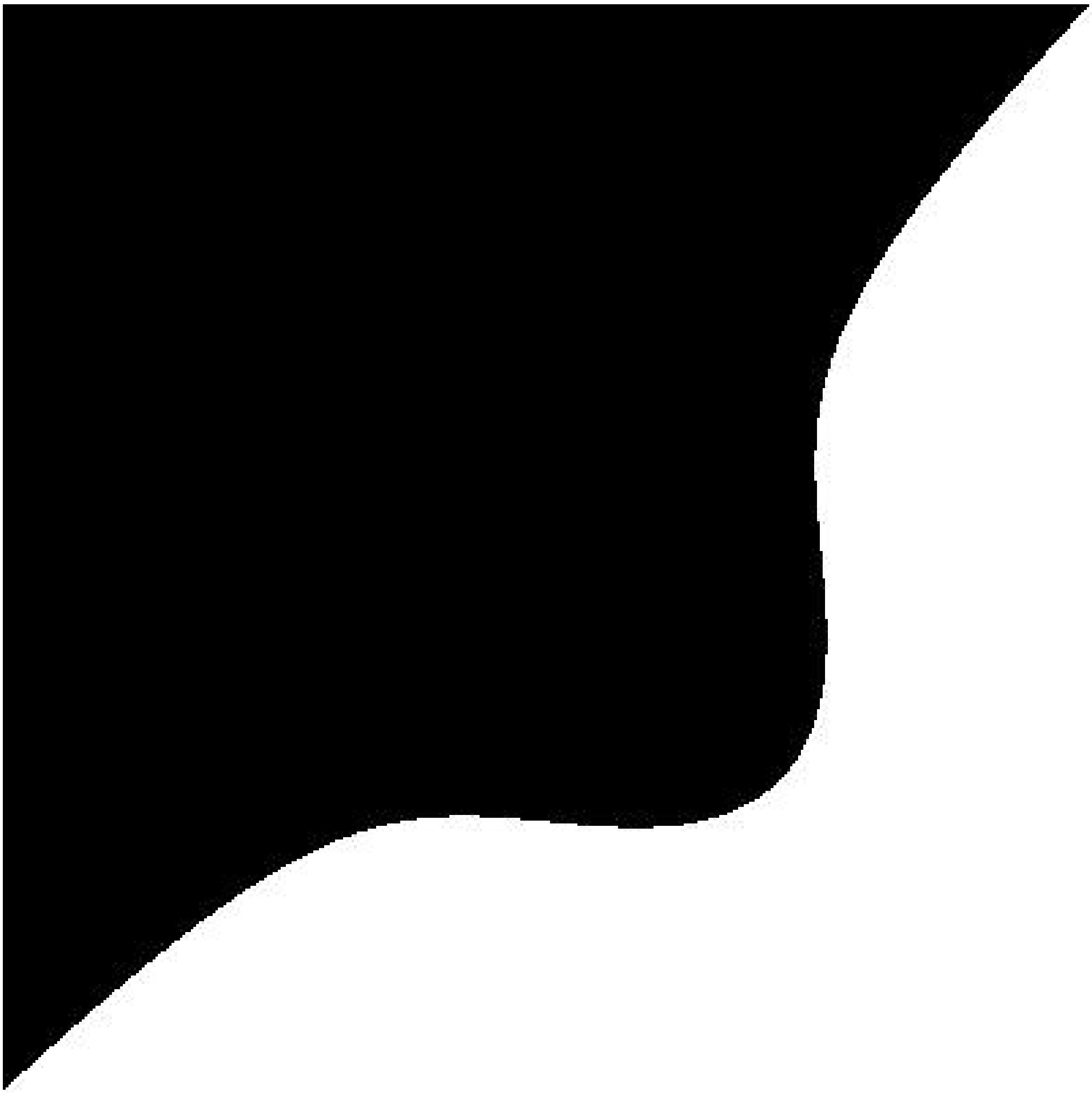
$2(X_1 : Y_1 : Z_1) =$
$(Z_1 : X_1 : Y_1) + (Y_1 : Z_1 : X_1)$
so can use ADD to double.

"Unified addition formulas,"
helpful against side channels.
But need to permute inputs.
2009 Bernstein–Kohel–Lange:
Easily avoid permutation!

2008 Hisil–Wong–Carter–Dawson:

$$(X : Y : Z : X^2 : Y^2 : Z^2 \\ : 2XY : 2XZ : 2YZ).$$

$6\mathbf{M} + 6\mathbf{S}$ for ADD.
$3\mathbf{M} + 6\mathbf{S}$ for DBL.



$$x^3 - y^3 + 1 = 0.3xy$$


The Hessian-ray: uniform
but
not strongly so

Jacobi intersections

1986 Chudnovsky–Chudnovsky:

$(S : C : D : Z)$ represent
$(S/Z, C/Z, D/Z)$ on
$s^2 + c^2 = 1$, $as^2 + d^2 = 1$.

$14\mathbf{M} + 2\mathbf{S} + 1\mathbf{D}$ for ADD.
"Tremendous advantage"
of being strongly unified.

$5\mathbf{M} + 3\mathbf{S}$ for DBL.
"Perhaps (?) . . . the most
efficient duplication formulas
which do not depend on the
coefficients of an elliptic curve."

2001 Liardet–Smart:

$13\mathbf{M} + 2\mathbf{S} + 1\mathbf{D}$ for ADD.

$4\mathbf{M} + 3\mathbf{S}$ for DBL.

2007 Bernstein–Lange:

$3\mathbf{M} + 4\mathbf{S}$ for DBL.

2008 Hisil–Wong–Carter–Dawson:

$13\mathbf{M} + 1\mathbf{S} + 2\mathbf{D}$ for ADD.

$2\mathbf{M} + 5\mathbf{S} + 1\mathbf{D}$ for DBL.

Also $(S : C : D : Z : SC : DZ)$:

$11\mathbf{M} + 1\mathbf{S} + 2\mathbf{D}$ for ADD.

$2\mathbf{M} + 5\mathbf{S} + 1\mathbf{D}$ for DBL.

Jacobi quartics

$(X{:}Y{:}Z)$ represent $(X/Z, Y/Z^2)$ on $y^2 = x^4 + 2ax^2 + 1$.

1986 Chudnovsky–Chudnovsky:
$3\mathbf{M} + 6\mathbf{S} + 2\mathbf{D}$ for DBL.
Slow ADD.

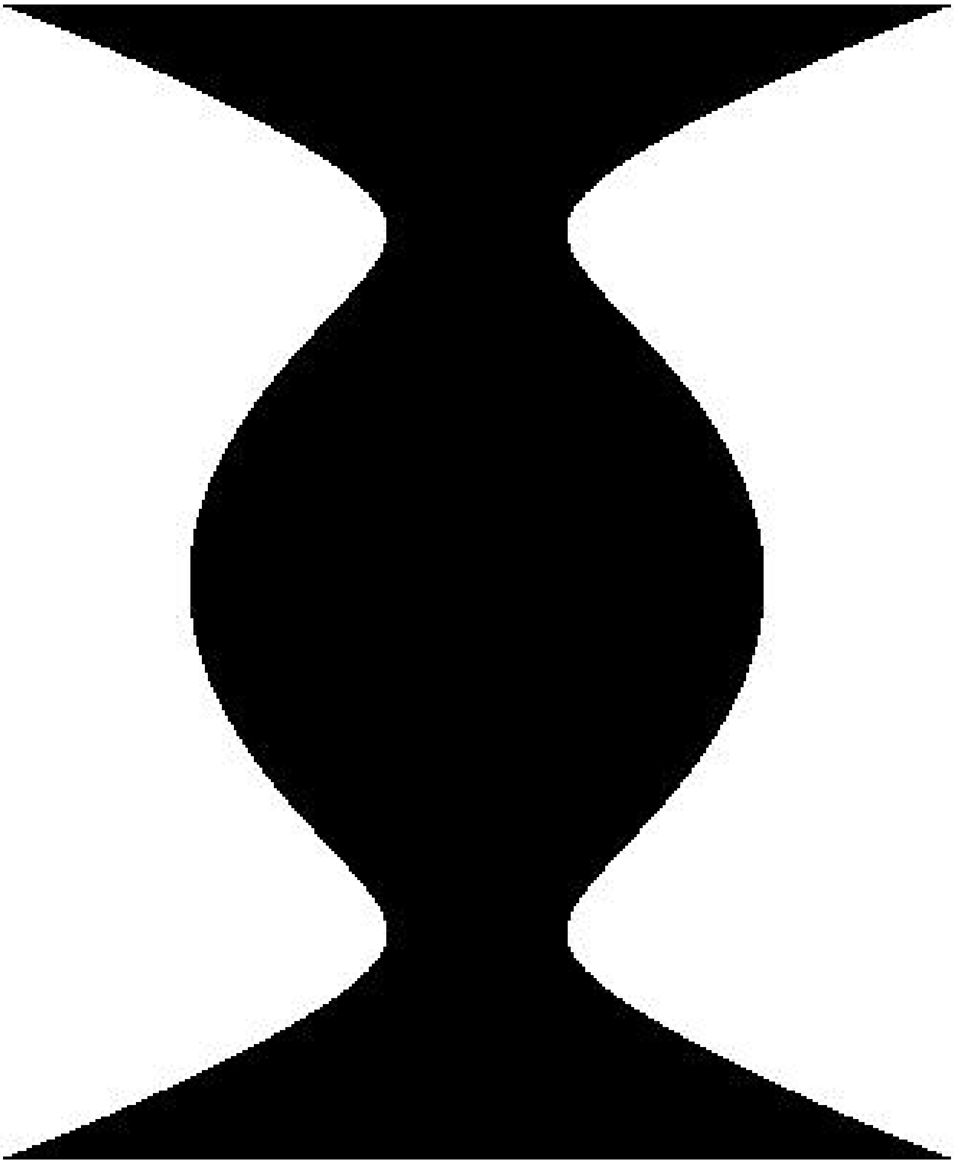
2002 Billet–Joye:
New choice of neutral element.
$10\mathbf{M} + 3\mathbf{S} + 1\mathbf{D}$ for ADD,
strongly unified.

2007 Bernstein–Lange:
$1\mathbf{M} + 9\mathbf{S} + 1\mathbf{D}$ for DBL.

2007 Hisil–Carter–Dawson:
$2\mathbf{M} + 6\mathbf{S} + 2\mathbf{D}$ for DBL.

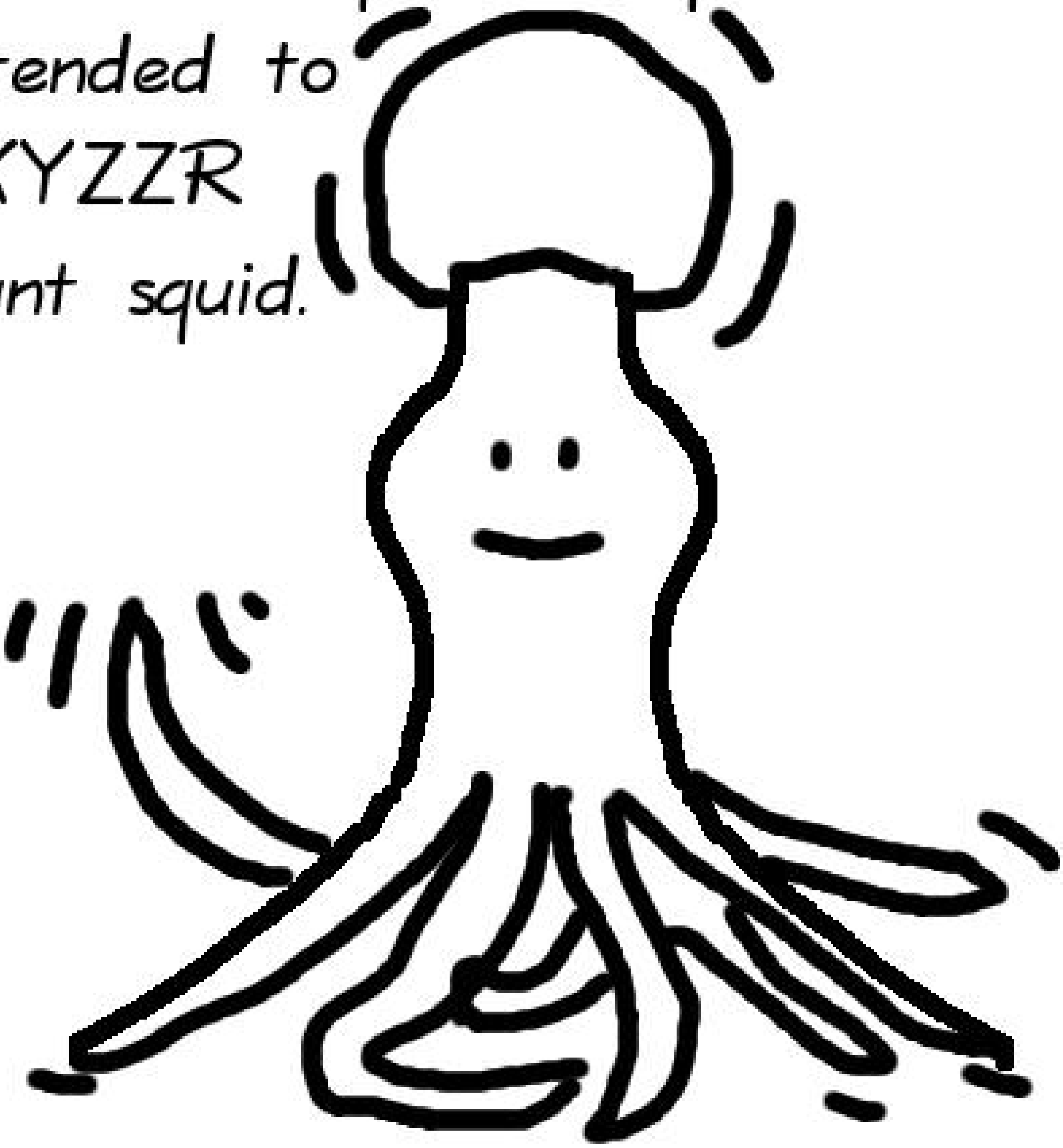
2007 Feng–Wu:
$2\mathbf{M} + 6\mathbf{S} + 1\mathbf{D}$ for DBL.
$1\mathbf{M} + 7\mathbf{S} + 3\mathbf{D}$ for DBL
on curves chosen with $a^2 + c^2 = 1$.

More speedups: 2007 Duquesne,
2007 Hisil–Carter–Dawson,
2008 Hisil–Wong–Carter–Dawson:
use $(X : Y : Z : X^2 : Z^2)$
or $(X : Y : Z : X^2 : Z^2 : 2XZ)$.
Can combine with Feng–Wu.
Competitive with Edwards!



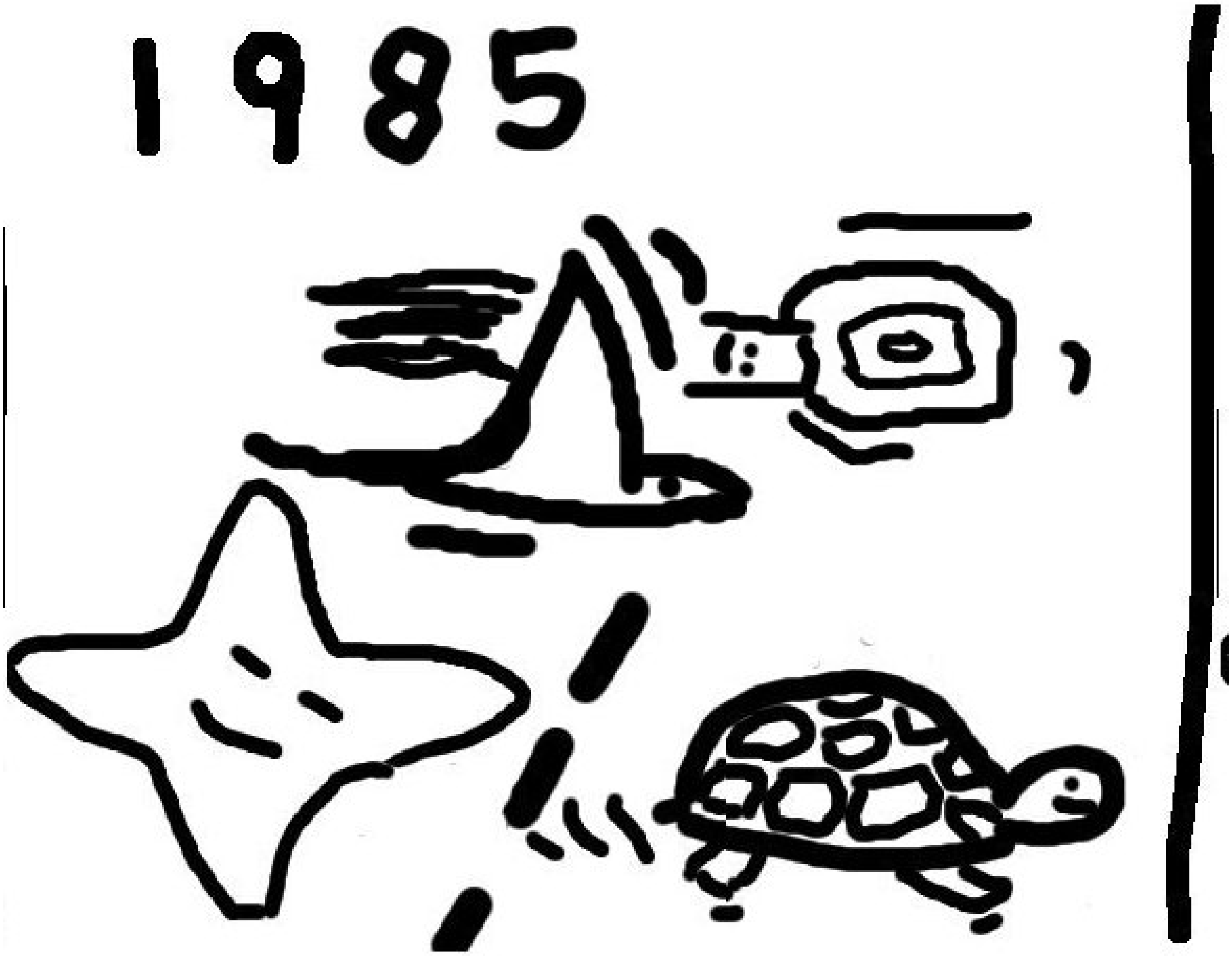
$$x^2 = y^4 - 1.9y^2 + 1$$

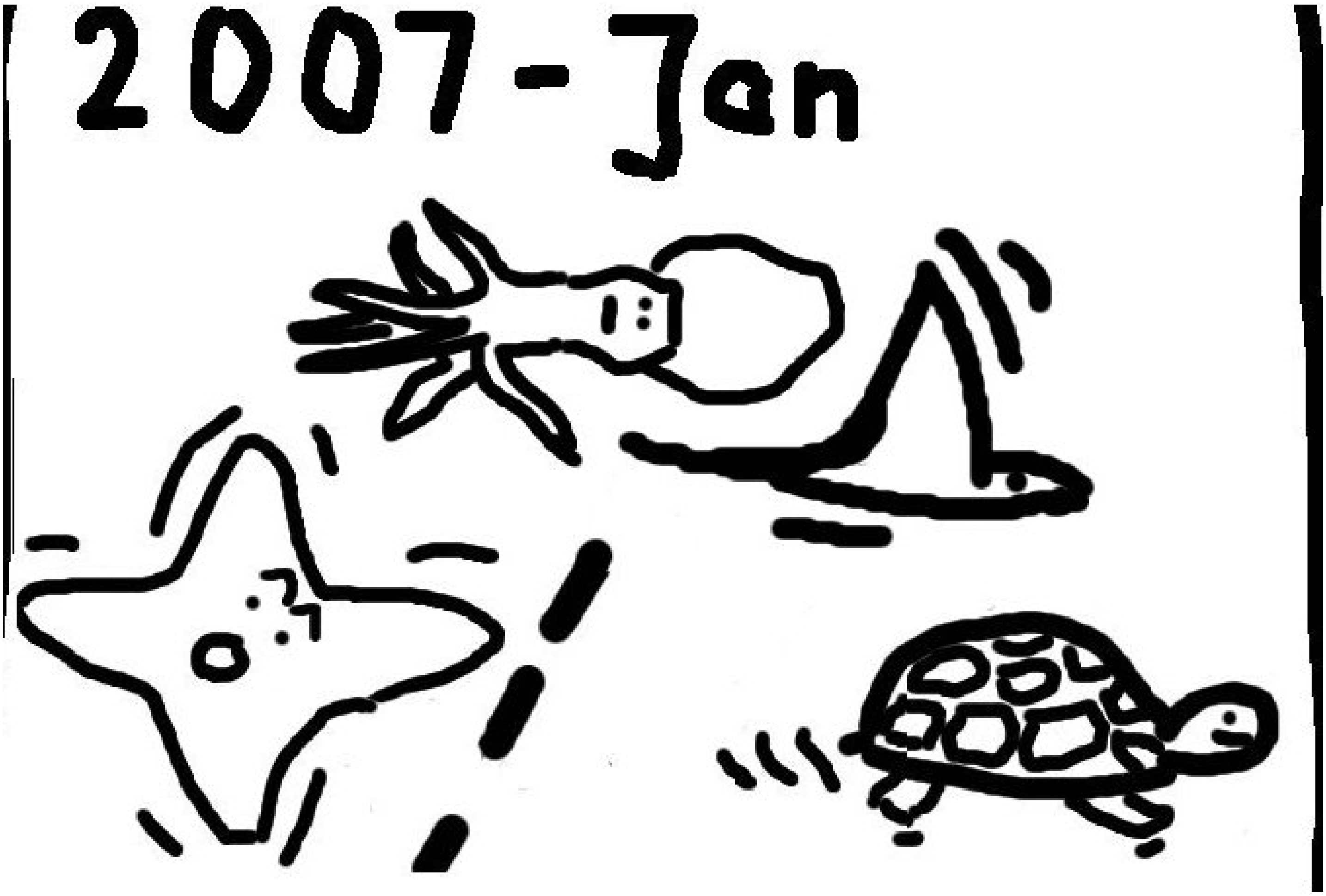
The Jacobi-quartic squid: can be extended to XXYZZR giant squid.

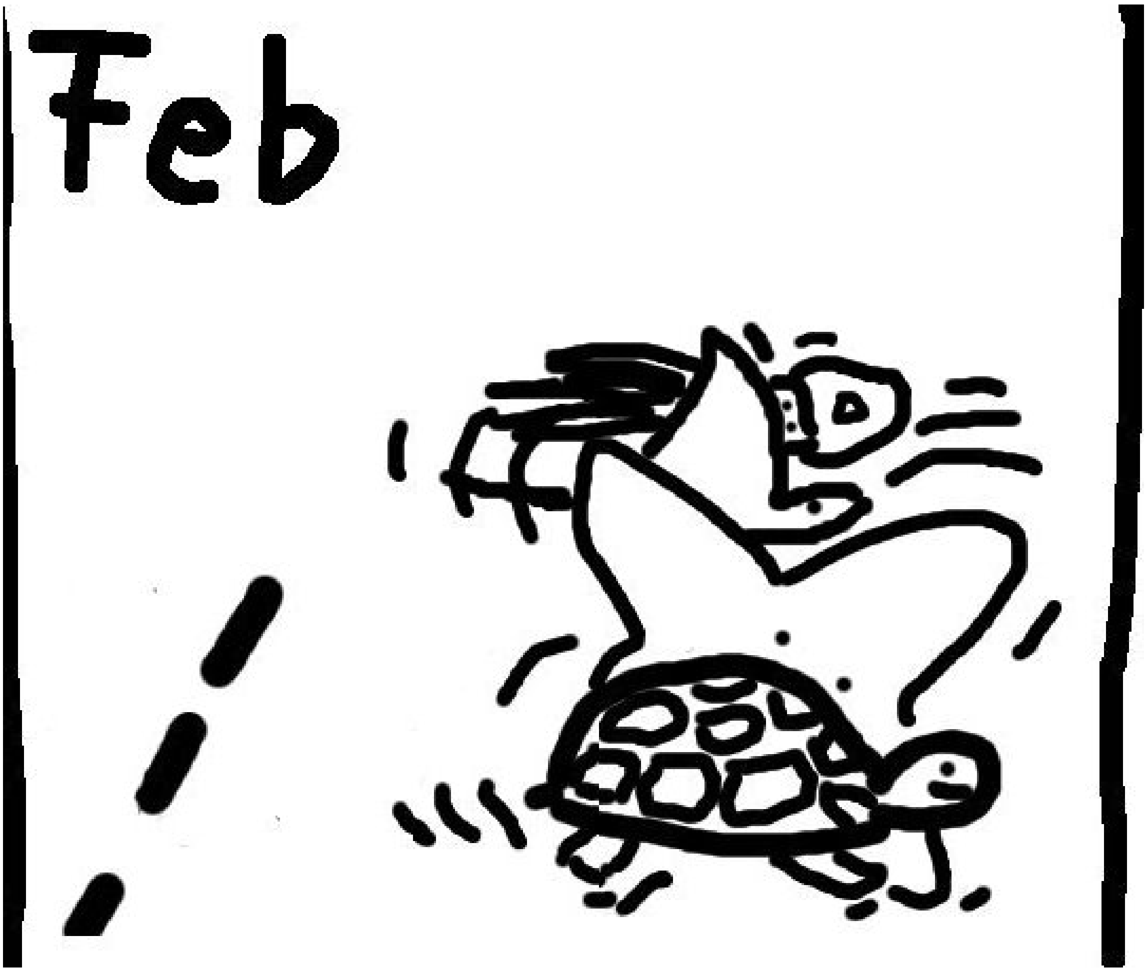



START

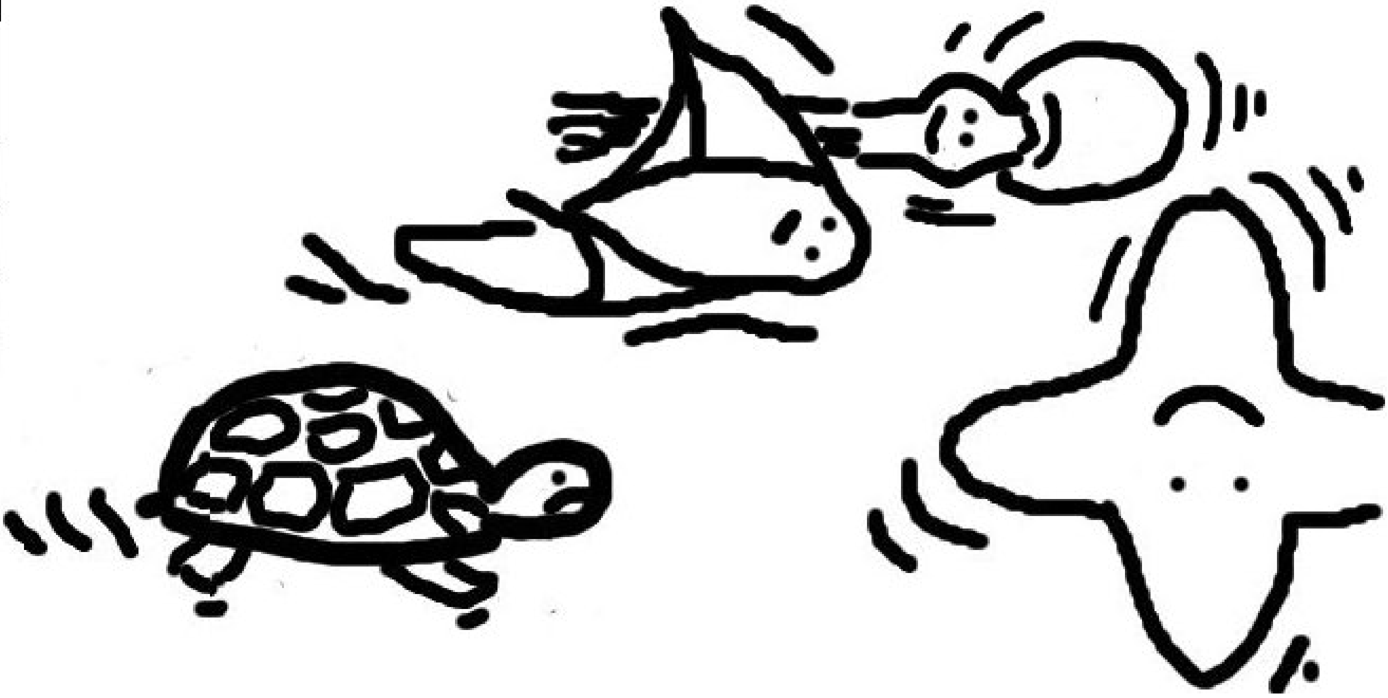
1985




2007-Jan


Feb

Mar



More addition formulas

Explicit-Formulas Database:
`hyperelliptic.org/EFD`

EFD has 583 computer-verified formulas and operation counts for ADD, DBL, etc. in 51 representations on 13 shapes of elliptic curves.

Not yet handled by computer: generality of curve shapes (e.g., Hessian order $\in 3\mathbf{Z}$); complete addition algorithms (e.g., checking for ∞).

How to multiply big integers

Standard idea: Use polynomial with coefficients in $\{0, 1, \ldots, 9\}$ to represent integer in radix 10.

Example of representation: $839 = 8 \cdot 10^2 + 3 \cdot 10^1 + 9 \cdot 10^0 =$ value (at $t = 10$) of polynomial $8t^2 + 3t^1 + 9t^0$.

Convenient to express polynomial inside computer as array $9, 3, 8$ (or $9, 3, 8, 0$ or $9, 3, 8, 0, 0$ or $\ldots$): "`p[0] = 9; p[1] = 3; p[2] = 8`"

Multiply two integers
by multiplying polynomials
that represent the integers.

Polynomial multiplication
involves *small* integer coefficients.
Have split one big multiplication
into many small operations.

Example, squaring 839:
$(8t^2 + 3t^1 + 9t^0)^2 =$
$64t^4 + 48t^3 + 153t^2 + 54t^1 + 81t^0$.

Oops, product polynomial usually has coefficients > 9.

So "carry" extra digits:

$ct^j \to \lfloor c/10 \rfloor t^{j+1} + (c \bmod 10)t^j$.

Example, squaring 839:

$64t^4 + 48t^3 + 153t^2 + 54t^1 + 81t^0$;

$64t^4 + 48t^3 + 153t^2 + 62t^1 + 1t^0$;

$64t^4 + 48t^3 + 159t^2 + 2t^1 + 1t^0$;

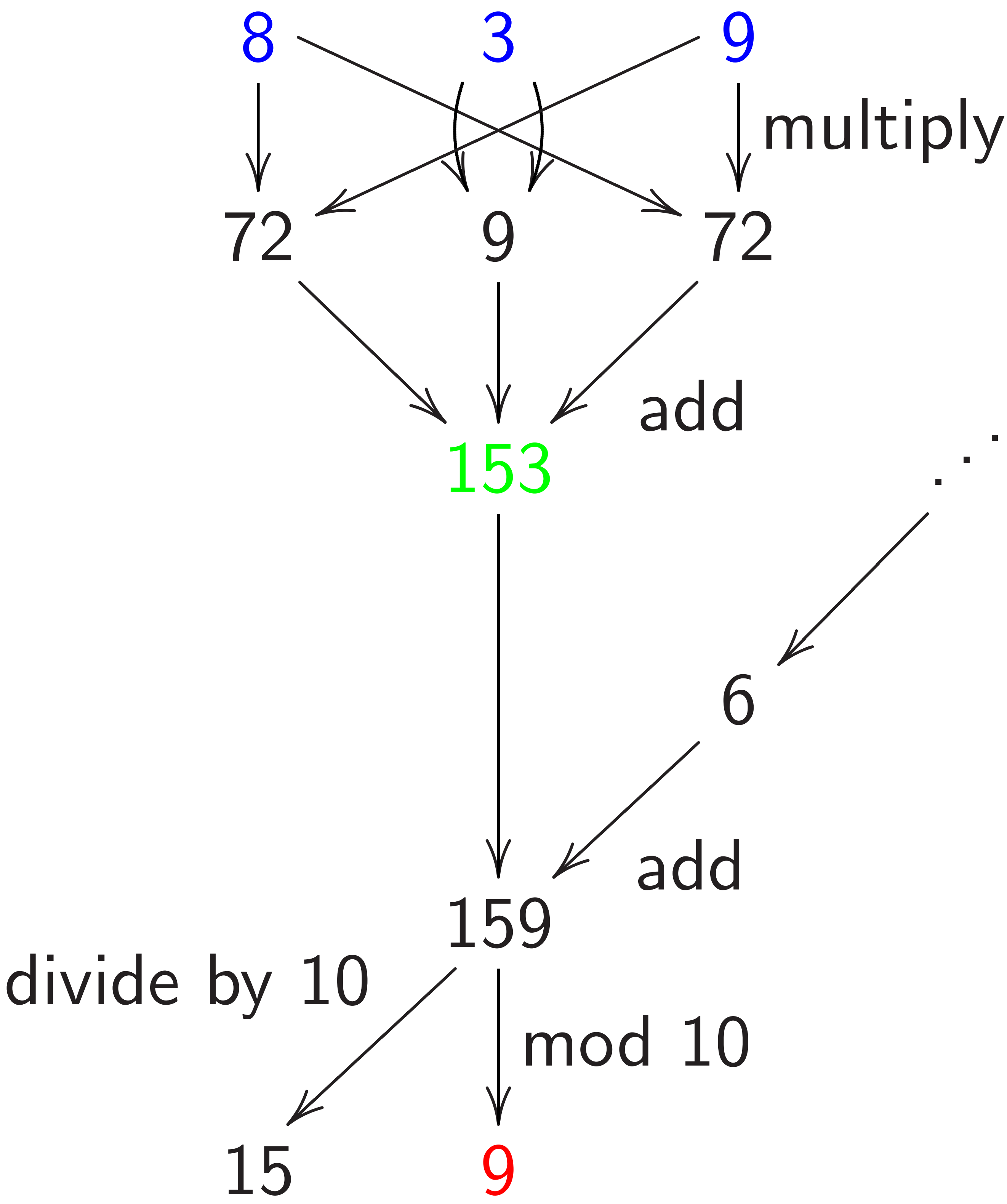
$64t^4 + 63t^3 + 9t^2 + 2t^1 + 1t^0$;

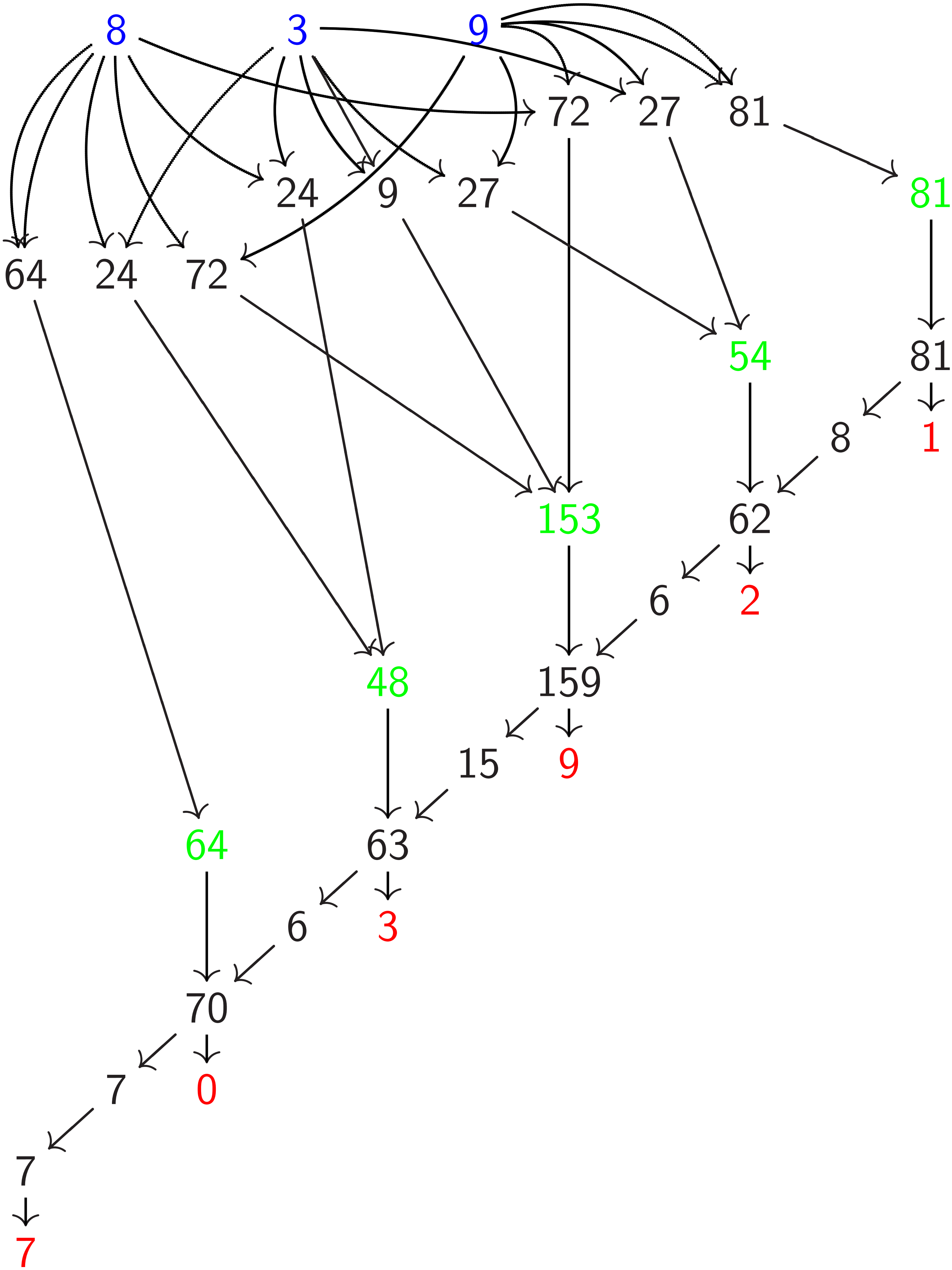
$70t^4 + 3t^3 + 9t^2 + 2t^1 + 1t^0$;

$7t^5 + 0t^4 + 3t^3 + 9t^2 + 2t^1 + 1t^0$.

In other words, $839^2 = 703921$.

What operations were used here?




8 3 9
72 27 81
24 9 27 81
64 24 72
54 81
8 1
153 62
6 2
48 159
15 9
64 63
6 3
70
7 0
7
7

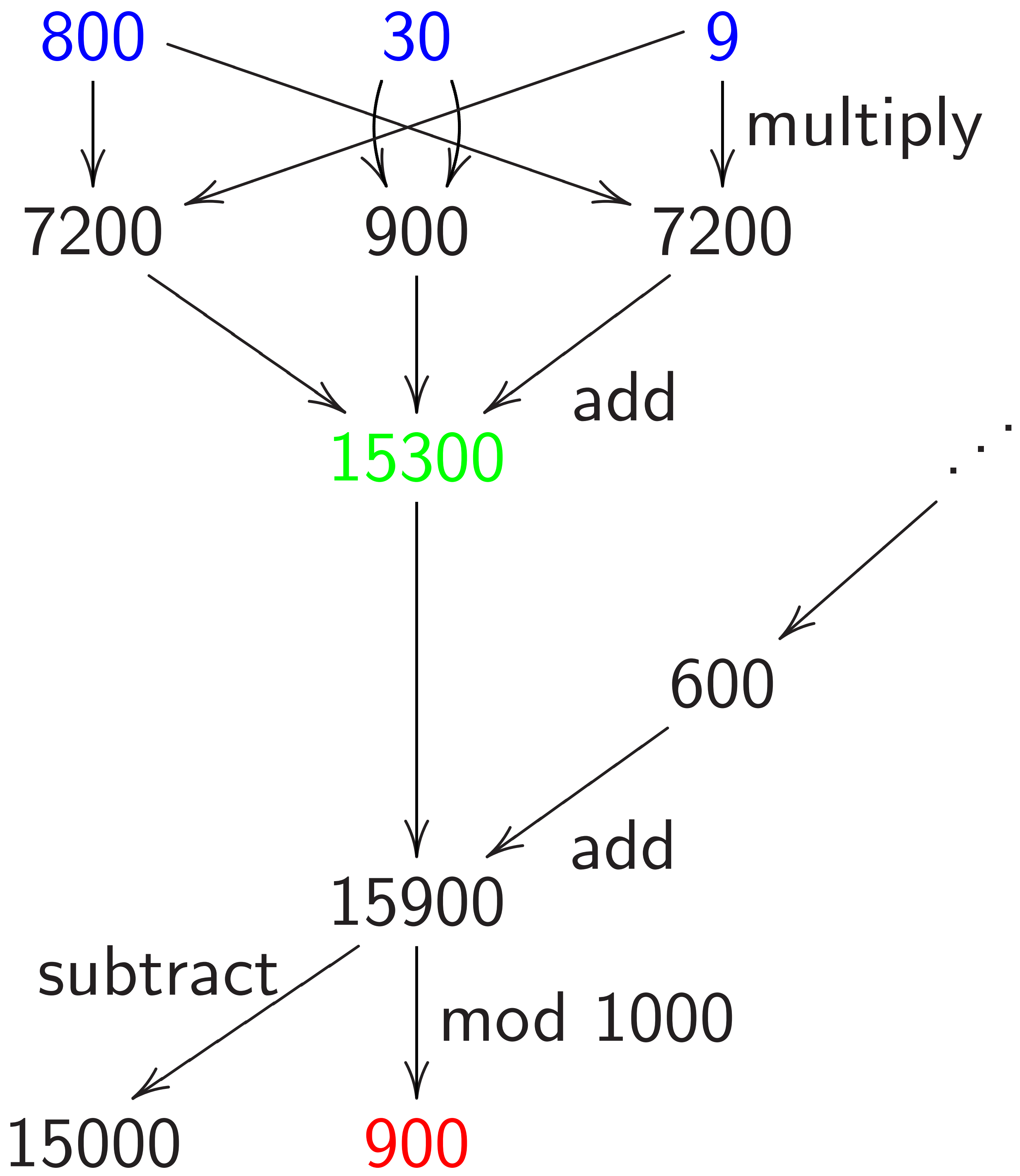
The scaled variation

$839 = 800 + 30 + 9 =$
value (at $t = 1$) of polynomial
$800t^2 + 30t^1 + 9t^0$.

Squaring: $(800t^2 + 30t^1 + 9t^0)^2 =$
$640000t^4 + 48000t^3 + 15300t^2 +$
$540t^1 + 81t^0$.

Carrying:
$640000t^4 + 48000t^3 + 15300t^2 +$
$540t^1 + 81t^0$;
$640000t^4 + 48000t^3 + 15300t^2 +$
$620t^1 + 1t^0$; ...
$700000t^5 + 0t^4 + 3000t^3 + 900t^2 +$
$20t^1 + 1t^0$.

What operations were used here?



Speedup: double inside squaring

$(\cdots + f_2 t^2 + f_1 t^1 + f_0 t^0)^2$
has coefficients such as

$f_4 f_0 + f_3 f_1 + f_2 f_2 + f_1 f_3 + f_0 f_4.$
5 mults, 4 adds.

Speedup: double inside squaring

$(\cdots + f_2 t^2 + f_1 t^1 + f_0 t^0)^2$
has coefficients such as
$f_4 f_0 + f_3 f_1 + f_2 f_2 + f_1 f_3 + f_0 f_4$.
5 mults, 4 adds.

Compute more efficiently as
$2 f_4 f_0 + 2 f_3 f_1 + f_2 f_2$.
3 mults, 2 adds, 2 doublings.

Save $\approx 1/2$ of the mults
if there are many coefficients.

Faster alternative:

$2(f_4 f_0 + f_3 f_1) + f_2 f_2$.

3 mults, 2 adds, 1 doubling.

Save $\approx 1/2$ of the adds
if there are many coefficients.

Faster alternative:

$2(f_4 f_0 + f_3 f_1) + f_2 f_2$.

3 mults, 2 adds, 1 doubling.

Save $\approx 1/2$ of the adds if there are many coefficients.

Even faster alternative:

$(2f_0)f_4 + (2f_1)f_3 + f_2 f_2$,

after precomputing $2f_0, 2f_1, \ldots$.

3 mults, 2 adds, 0 doublings.

Precomputation ≈ 0.5 doublings.

Speedup: allow negative coeffs

Recall $159 \mapsto 15, 9$.

Scaled: $15900 \mapsto 15000, 900$.

Alternative: $159 \mapsto 16, -1$.

Scaled: $15900 \mapsto 16000, -100$.

Use digits $\{-5, -4, \ldots, 4, 5\}$
instead of $\{0, 1, \ldots, 9\}$.
Small disadvantage: need $-$.
Several small advantages:
easily handle negative integers;
easily handle subtraction;
reduce products a bit.

Speedup: delay carries

Computing (e.g.) big $ab + c^2$: multiply a, b polynomials, carry, square c poly, carry, add, carry.

e.g. $a = 314$, $b = 271$, $c = 839$:

$(3t^2 + 1t^1 + 4t^0)(2t^2 + 7t^1 + 1t^0) =$

$6t^4 + 23t^3 + 18t^2 + 29t^1 + 4t^0$;

carry: $8t^4 + 5t^3 + 0t^2 + 9t^1 + 4t^0$.

As before $(8t^2 + 3t^1 + 9t^0)^2 =$

$64t^4 + 48t^3 + 153t^2 + 54t^1 + 81t^0$;

$7t^5 + 0t^4 + 3t^3 + 9t^2 + 2t^1 + 1t^0$.

$+$: $7t^5 + 8t^4 + 8t^3 + 9t^2 + 11t^1 + 5t^0$;

$7t^5 + 8t^4 + 9t^3 + 0t^2 + 1t^1 + 5t^0$.

Faster: multiply a, b polynomials, square c polynomial, add, carry.

$(6t^4 + 23t^3 + 18t^2 + 29t^1 + 4t^0) +$
$(64t^4 + 48t^3 + 153t^2 + 54t^1 + 81t^0)$
$= 70t^4 + 71t^3 + 171t^2 + 83t^1 + 85t^0$;
$7t^5 + 8t^4 + 9t^3 + 0t^2 + 1t^1 + 5t^0$.

Eliminate intermediate carries.
Outweighs cost of handling slightly larger coefficients.

Important to carry between multiplications (and squarings) to reduce coefficient size;
but carries are usually a bad idea before additions, subtractions, etc.

Speedup: polynomial Karatsuba

How much work to multiply polys
$f = f_0 + f_1 t + \cdots + f_{19} t^{19}$,
$g = g_0 + g_1 t + \cdots + g_{19} t^{19}$?

Using the obvious method:
400 coeff mults, 361 coeff adds.

Faster: Write f as $F_0 + F_1 t^{10}$;
$F_0 = f_0 + f_1 t + \cdots + f_9 t^9$;
$F_1 = f_{10} + f_{11} t + \cdots + f_{19} t^9$.
Similarly write g as $G_0 + G_1 t^{10}$.

Then $fg = (F_0 + F_1)(G_0 + G_1) t^{10}$
$+ (F_0 G_0 - F_1 G_1 t^{10})(1 - t^{10})$.

20 adds for $F_0 + F_1$, $G_0 + G_1$.
300 mults for three products
$F_0 G_0$, $F_1 G_1$, $(F_0 + F_1)(G_0 + G_1)$.
243 adds for those products.
9 adds for $F_0 G_0 - F_1 G_1 t^{10}$
with subs counted as adds
and with delayed negations.
19 adds for $\cdots (1 - t^{10})$.
19 adds to finish.

Total 300 mults, 310 adds.
Larger coefficients, slight expense;
still saves time.

Can apply idea recursively
as poly degree grows.

Many other algebraic speedups in polynomial multiplication: "Toom," "FFT," etc.

Increasingly important as polynomial degree grows. $O(n \lg n \lg \lg n)$ coeff operations to compute n-coeff product.

Useful for sizes of n that occur in cryptography? In some cases, yes! But Karatsuba is the limit for prime-field ECC/ECDLP on most current CPUs.

Modular reduction

How to compute $f \bmod p$?

Can use definition:
$f \bmod p = f - p \lfloor f/p \rfloor$.
Can multiply f by a precomputed $1/p$ approximation; easily adjust to obtain $\lfloor f/p \rfloor$.

Slight speedup: "2-adic inverse"; "Montgomery reduction."

e.g. 314159265358 mod 271828:

Precompute

$\lfloor 1000000000000/271828 \rfloor$

$= 3678796.$

Compute

$314159 \cdot 3678796$

$= 1155726872564.$

Compute

$314159265358 - 1155726 \cdot 271828$

$= 578230.$

Oops, too big:

$578230 - 271828 = 306402.$

$306402 - 271828 = 34574.$

We can do better: normally
p is chosen with a special form
to make f mod p much faster.

Special primes hurt security
for $\mathbf{F}_p^*$, Clock($\mathbf{F}_p$), etc.,
but not for elliptic curves!

gls1271: $p = 2^{127} - 1$,
with degree-2 extension.

Curve25519: $p = 2^{255} - 19$.

NIST P-224: $p = 2^{224} - 2^{96} + 1$.

secp112r1: $p = (2^{128} - 3)/76439$.
Divides special form.

Small example: $p = 1000003$.
Then $1000000a + b \equiv b - 3a$.

e.g. $314159265358 =$
$314159 \cdot 1000000 + 265358 \equiv$
$314159(-3) + 265358 =$
$-942477 + 265358 =$
-677119.

Easily adjust $b - 3a$
to the range $\{0, 1, \ldots, p - 1\}$
by adding/subtracting a few p's:
e.g. $-677119 \equiv 322884$.

Hmmm, is adjustment so easy?

Conditional branches are slow.
(Also dangerous for defenders: branch timing leaks secrets.)
Can eliminate the branches, but adjustment isn't free.

Speedup: Skip the adjustment for intermediate results.
"Lazy reduction."
Adjust only for output.

$b - 3a$ is small enough to continue computations.

Can delay carries until after multiplication by 3.

e.g. To square 314159 in $\mathbf{Z}/1000003$: Square poly $3t^5 + 1t^4 + 4t^3 + 1t^2 + 5t^1 + 9t^0$, obtaining $9t^{10} + 6t^9 + 25t^8 + 14t^7 + 48t^6 + 72t^5 + 59t^4 + 82t^3 + 43t^2 + 90t^1 + 81t^0$.

Reduce: replace $(c_i)t^{6+i}$ by $(-3c_i)t^i$, obtaining $72t^5 + 32t^4 + 64t^3 - 32t^2 + 48t^1 - 63t^0$.

Carry: $8t^6 - 4t^5 - 2t^4 + 1t^3 + 2t^2 + 2t^1 - 3t^0$.

To minimize poly degree, mix reduction and carrying, carrying the top sooner.

e.g. Start from square $9t^{10} + 6t^9 + 25t^8 + 14t^7 + 48t^6 + 72t^5 + 59t^4 + 82t^3 + 43t^2 + 90t^1 + 81t^0$.

Reduce $t^{10} \to t^4$ and carry $t^4 \to t^5 \to t^6$: $6t^9+25t^8+14t^7+56t^6-5t^5+2t^4+82t^3+43t^2+90t^1+81t^0$.

Finish reduction: $-5t^5 + 2t^4 + 64t^3 - 32t^2 + 48t^1 - 87t^0$. Carry $t^0 \to t^1 \to t^2 \to t^3 \to t^4 \to t^5$: $-4t^5 - 2t^4 + 1t^3 + 2t^2 - 1t^1 + 3t^0$.

Speedup: non-integer radix

$p = 2^{61} - 1$.

Five coeffs in radix 2^{13}?
$f_4 t^4 + f_3 t^3 + f_2 t^2 + f_1 t^1 + f_0 t^0$.
Most coeffs could be 2^{12}.

Square $\cdots + 2(f_4 f_1 + f_3 f_2) t^5 + \cdots$.
Coeff of t^5 could be $> 2^{25}$.

Reduce: $2^{65} = 2^4$ in $\mathbf{Z}/(2^{61} - 1)$;
$\cdots + (2^5(f_4 f_1 + f_3 f_2) + f_0^2) t^0$.
Coeff could be $> 2^{29}$.
Very little room for
additions, delayed carries, etc.
on 32-bit platforms.

Scaled: Evaluate at $t = 1$.

f_4 is multiple of 2^{52};

f_3 is multiple of 2^{39};

f_2 is multiple of 2^{26};

f_1 is multiple of 2^{13};

f_0 is multiple of 2^0. Reduce:

$\cdots + (2^{-60}(f_4 f_1 + f_3 f_2) + f_0^2)t^0$.

Better: Non-integer radix $2^{12.2}$.

f_4 is multiple of 2^{49};

f_3 is multiple of 2^{37};

f_2 is multiple of 2^{25};

f_1 is multiple of 2^{13};

f_0 is multiple of 2^0.

Saves a few bits in coeffs.