An introduction to high-speed arithmetic

D. J. Bernstein
University of Illinois at Chicago

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:

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

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

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

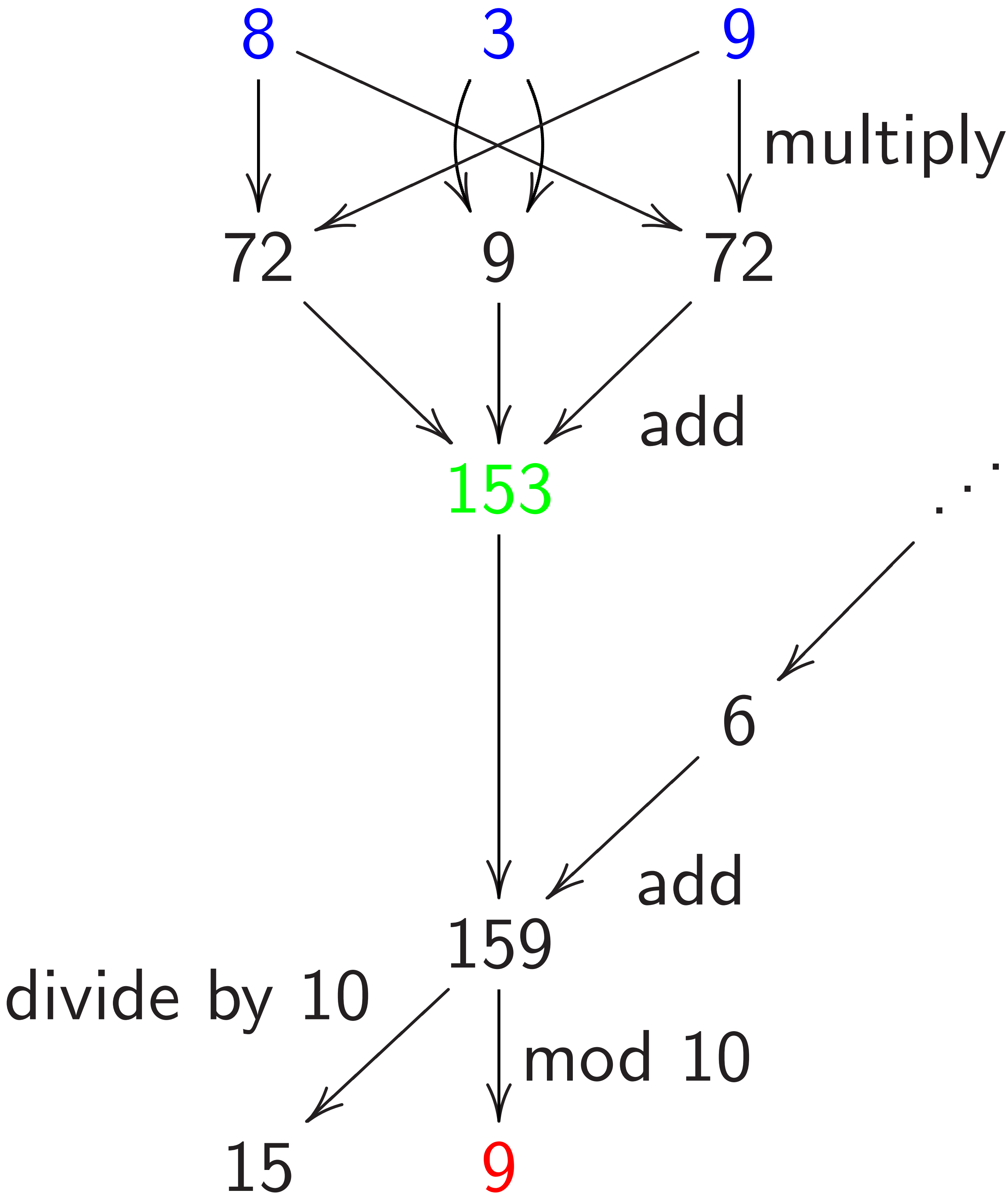
$64t^4 + 63t^3 + 9t^2 + 2t^1 + 1t^0$;

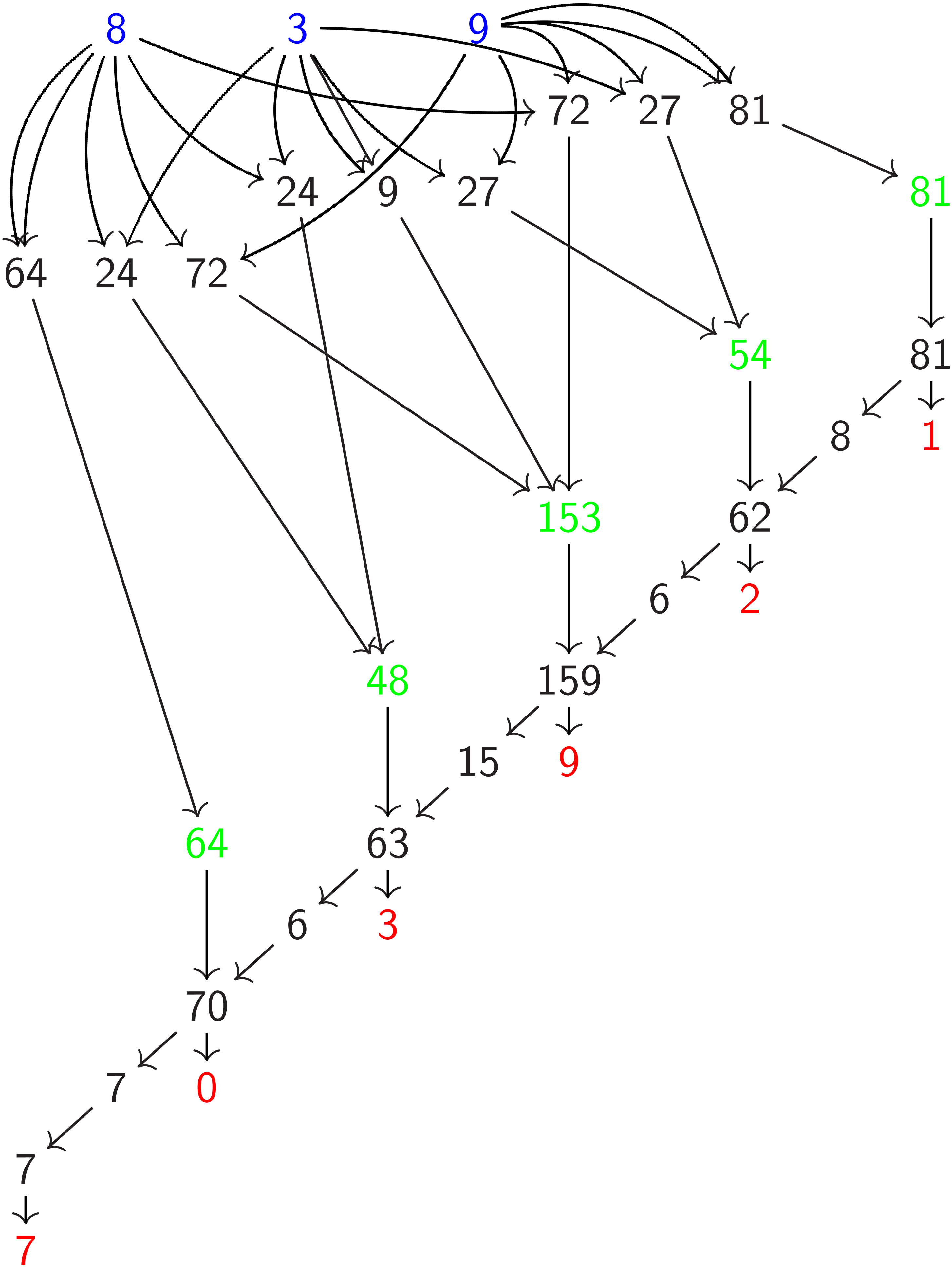
$70t^4 + 3t^3 + 9t^2 + 2t^1 + 1t^0$;

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

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

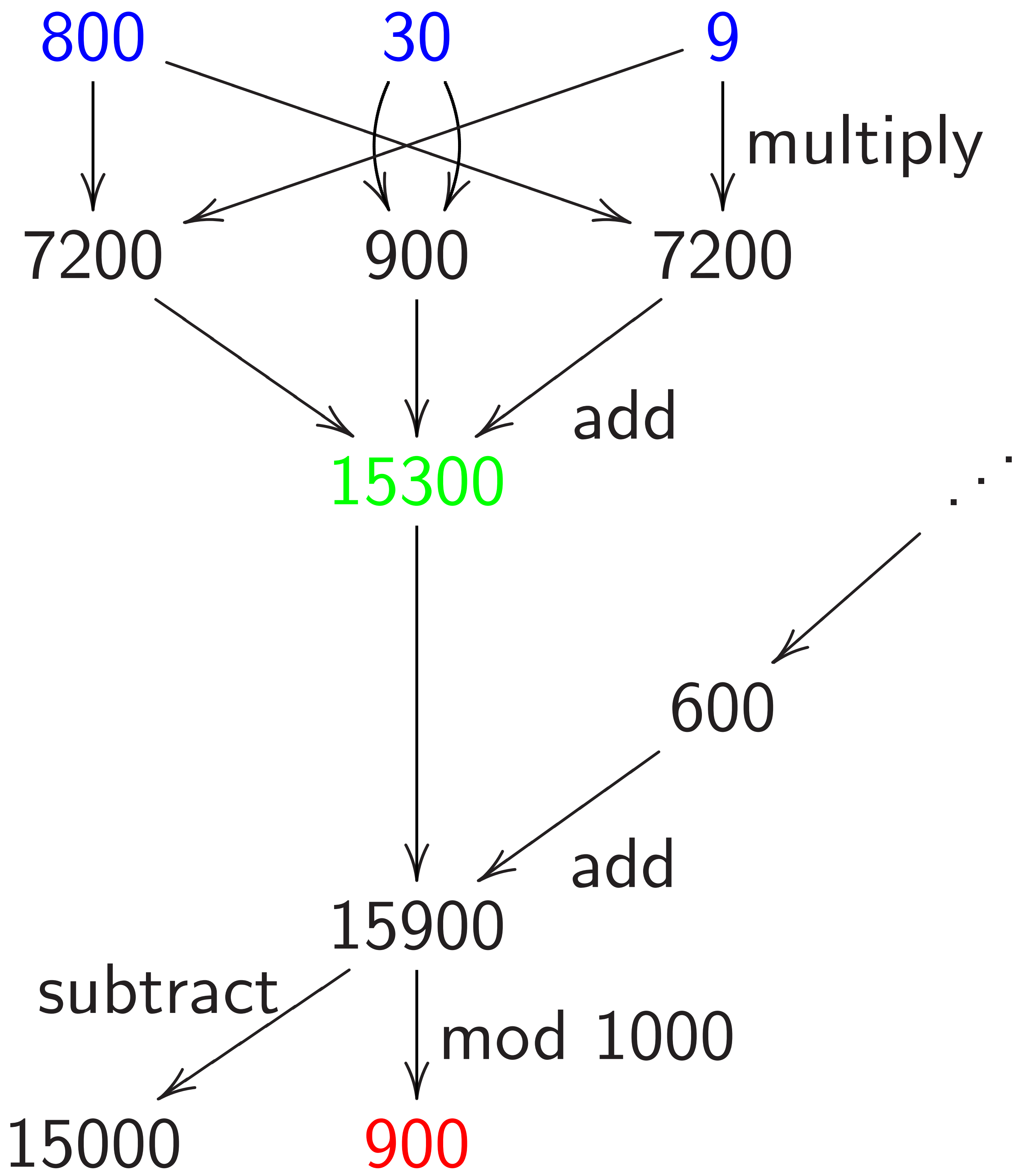
The scaled variation

$839 = 800 + 30 + 9 =$
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:
$640000t^4 + 48000t^3 + 15300t^2 +$
$540t^1 + 81t^0$;
$640000t^4 + 48000t^3 + 15300t^2 +$
$620t^1 + 1t^0$; $\ldots$
$700000t^5 + 0t^4 + 3000t^3 + 900t^2 +$
$20t^1 + 1t^0$.

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
$f_4 f_0 + f_3 f_1 + f_2 f_2 + f_1 f_3 + f_0 f_4$.

Compute more efficiently as
$2 f_4 f_0 + 2 f_3 f_1 + f_2 f_2$.
Or, slightly faster,
$2(f_4 f_0 + f_3 f_1) + f_2 f_2$.
Or, slightly faster,
$(2 f_4) f_0 + (2 f_3) f_1 + f_2 f_2$
after precomputing $2f_1, 2f_2, \ldots$.

Overall save $\approx 1/2$ of the work
if there are many coefficients.

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_0G_0, F_1G_1, $(F_0 + F_1)(G_0 + G_1)$.

243 adds for those products.

9 adds for $F_0G_0 - F_1G_1t^{10}$

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? Maybe; active research area.

Using CPU's integer instructions

Replace radix 10 with, e.g., 2^{24}.
Power of 2 simplifies carries.

Adapt radix to platform.

e.g. Every 2 cycles, Athlon 64 can compute a 128-bit product of two 64-bit integers.
(5-cycle latency; parallelize!)
Also low cost for 128-bit add.

Reasonable to use radix 2^{60}.
Sum of many products of digits fits comfortably below 2^{128}.
Be careful: analyze largest sum.

e.g. In 4 cycles, Intel 8051
can compute a 16-bit product
of two 8-bit integers.
Could use radix 2^6.
Could use radix 2^8,
with 24-bit sums.

e.g. Every 2 cycles, Pentium 4 F3
can compute a 64-bit product
of two 32-bit integers.
(11-cycle latency; yikes!)
Reasonable to use radix 2^{28}.

Warning: Multiply instructions
are very slow on some CPUs.
Pentium 4 F2: every 10 cycles!

Using floating-point instructions

Big CPUs have separate
floating-point instructions,
aimed at numerical simulation
but useful for cryptography.

In my experience,
floating-point instructions
support faster multiplication
(often much, much faster)
than integer instructions.
Other advantages: portability;
easily scaled coefficients.

Exceptions: some 64-bit CPUs.

e.g. Every 2 cycles, Pentium III can compute a 64-bit product of two floating-point numbers, and an independent 64-bit sum.

e.g. Every cycle, UltraSPARC III can compute a 53-bit product and an independent 53-bit sum. Reasonable to use radix 2^{24}.

e.g. Every 2 cycles, Pentium 4 can compute two 53-bit products and two independent 53-bit sums.

e.g. Every 2 cycles, Pentium M can compute two 53-bit products and two independent 53-bit sums.

e.g. Every cycle, Athlon can compute a 64-bit product and an independent 64-bit sum.

e.g. Every cycle, Core 2 Solo can compute two 53-bit products and two independent 53-bit sums. (Beware relatively high latency.)

How to do carries in
floating-point registers?
(No CPU carry instruction:
not useful for simulations.)

Exploit floating-point rounding:
add and subtract big constant.

e.g. Given α with $|\alpha| \leq 2^{75}$:
compute 53-bit floating-point sum
of α and constant $3 \cdot 2^{75}$,
obtaining a multiple of 2^{24};
subtract $3 \cdot 2^{75}$ from result,
obtaining multiple of 2^{24}
nearest α; subtract from α.

Modular arithmetic

$\lfloor a/p \rfloor$ is the quotient
when a is divided by p:
the largest integer $\leq a/p$.

$a \bmod p$ is the remainder:
$a \bmod p = a - p \lfloor a/p \rfloor$.

Examples:
$\lfloor 43/12 \rfloor = 3$; $43 \bmod 12 = 7$.
$\lfloor 17/12 \rfloor = 1$; $17 \bmod 12 = 5$.
$\lfloor 12/12 \rfloor = 1$; $12 \bmod 12 = 0$.
$\lfloor 7/12 \rfloor = 0$; $7 \bmod 12 = 7$.
$\lfloor -10/12 \rfloor = -1$;
$-10 \bmod 12 = 2$.

Often want to compute $a \bmod p$ where a is a gigantic integer produced by mults, adds, subs and p is relatively small.

e.g. $p = 314159$; $a = 7^{1024} = (((((((((7^2)^2)^2)^2)^2)^2)^2)^2)^2)^2$.

Useful fact: If we change the chain of mults, adds, subs by inserting "mod p" anywhere, the new chain output a' satisfies $a' \bmod p = a \bmod p$.
"$a' \equiv a$": a', a are equivalent.

More generally, inserting adds/subs of *any* multiples of p produces $a' \equiv a$.

e.g. $p = 17$,
$a = ((5^2) \cdot 5)^2 = 15625$:
$a \bmod p = 15625 \bmod 17 = 2$.

Can change a to, e.g., a'
$= (((5^2 \bmod 17) \cdot 5) \bmod 17)^2$
$= (((25 \bmod 17) \cdot 5) \bmod 17)^2$
$= ((8 \cdot 5) \bmod 17)^2$
$= (40 \bmod 17)^2 = 6^2 = 36$.
Then $a' \bmod p = 36 \bmod 17 = 2$.
No big numbers here!

Modular reduction

How to compute f mod 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."

We can do better: normally p is chosen with a special form (or dividing a special form; see "redundant representations") to make f mod p much faster.

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 crypto:
leak secrets through timing.
Can eliminate the branches,
but adjustment isn't free.

Speedup: Skip the adjustment
for intermediate results.
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.