BigNums

When 64 bits just isn't enough

YAPC::NA 2015

Dana Jacobsen, 10 Jun 2015

Who am I?

dana@acm.org

Software Engineer at AppNexus

Member of PDX.pm

Perl user since 4.011 (Nov 1991)

Author of ntheory module

Author of non-Pari replacements for Crypt::RSA & DSA

Contributor to RosettaCode and OEIS

Pascal’s Triangle

```
sub pascal {
  my $rows = shift;
  my @next = (1);
  for my $n (1 .. $rows) {
    print "@next\n";
    @next = (1,
             (map $next[$_]+$next[$_+1], 0 .. $n-2),
             1);
  }
}
```

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
....
... 2.19128700370045e+19 ... [line 69]

line 55 with 32-bit perl

Perl’s numbers

```
perl32 -E ‘say ~0’
4294967295

perl -E ‘say ~0’
18446744073709551615


perl -E 'say 84931153 * 72761567'
6.17972377939675e+15
```

More NVs, and use integer

```
perl32 -E ‘97829 * 125141'
1224241889                                    <= an NV (double)

perl -E ‘use integer; say 84931153 * 72761567'
6179723779396751                              <= looks good!

perl32 -E ‘use integer; say 84931153 * 72761567'
279860367                                     <= oh my

perl32 -E ‘use integer; say 2**31+1’
-2147483647                                   <= Argh
```

Math::BigInt

```
use Math::BigInt;
my $n = Math::BigInt->new(1);
my $m = Math::BigInt->new(10) ** 457 + 499;
```

No FP conversions on operations (integer semantics)

Arbitrary length: 10s of millions of digits if desired

Pascal’s Triangle

```
sub pascal {
  my $rows = shift;
  my @next = (1);
  for my $n (1 .. $rows) {
    print "@next\n";
    @next = (1,
             (map $next[$_]+$next[$_+1], 0 .. $n-2),
             1);
  }
}
```

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
....
...  2.1912870037045e+19 ...   [line 69]
```

Pascal’s Triangle

```
sub pascal {
  use bigint;
  my $rows = shift;
  my @next = (1);
  for my $n (1 .. $rows) {
    print "@next\n";
    @next = (1,
             (map $next[$_]+$next[$_+1], 0 .. $n-2),
             1);
  }
}
```

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
....
... **21912870037044995008** ... [line 69]

Binomial

```
sub binomial {
    my ($r, $n, $k) = (1, @_);
    for (1 .. $k) { $r *= $n--; $r /= $_ }
    $r;
}
```

FP results at (53,23) in 32-bit

FP results at (63,29) in 64-bit

Binomial

```
sub binomial {
    use bigint;
    my ($r, $n, $k) = (1, @_);
    for (1 .. $k) { $r *= $n--; $r /= $_ }
    $r;
}
```

Works correctly for large inputs

Math::BigInt extras

```
$x->bmodpow($y, $n)       # ($x ** $y) % $n
$n->bfac()                # n!
$n->bnok($k)              # binomial($n,$k)
$n->blog($base)           # log_$base of $n

my $n = Math::BigInt->from_bin(“1101010111”);
my $m = Math::BigInt->from_hex(“0xDEADC0DE”);
say $m->as_bin();
say $n->as_hex();
```

Math::BigInt gotchas

```
use bigint;
for my $x (3000000..3000001) {
  for my $y (3000000..3000001) {
    for my $z (3000000..3000001) {
      say $x * $y * $z;
    }
  }
}
2.7e+19
2.7000009e+19
...
```

Ranges are always signed ints

Math::BigInt gotchas

```
use bigint;
for my $x (3000000..3000001) {
  for my $y (3000000..3000001) {
    for my $z (3000000..3000001) {
      say 1 * $x * $y * $z;
    }
  }
}
27000000000000000000
27000009000000000000
...
```

Ranges are always signed ints

Coerce to bigint: 1* 0+

Math::BigInt gotchas

```
use bigint;
my @n = (qw/6038203321 5157712919 4485674059 6818955709/);
$n[0] *= $n[$_] for 1..$#n;
say $n[0];

9.52599789253049e+38
```

Strings aren’t bigints
Coerce or explicitly create object

User input, hash keys, etc.

Math::BigInt gotchas

```
use bigint;
say ~0;

-1
```

Complement of 0 isn't UV_MAX

Crossing a Camel with a Snail

RosettaCode AGM example. Calculate 10k digits of Pi.

```
2min 43s     bigint
```

Crossing a Camel with a Snail

RosettaCode AGM example. Calculate 10k digits of Pi.

```
2min 43s      bigint
      6.7s    bigint lib=>”Pari”;
      0.13s   bigint lib=>”GMP”;
```

Crossing a Camel with a Snail

RosettaCode AGM example. Calculate 10k digits of Pi.

```
2min 43s    bigint
      6.7s  bigint lib=>”Pari”;
      0.13s bigint lib=>”GMP”;
```

To install: `cpan Math::BigInt::GMP`

```
try  => ”GMP,Pari”
only => “GMP”
```

The perils of CPAN

Neither back end will install since 5.21.0 (including 5.22.0). Doh!

GMP backend turns large ints into negative inputs.
Workaround: stringify everything: `$n=Math::BigInt->new("$m");`
RT filed 3.5 years ago. Patch submitted 2 years ago. Still waiting.

Pari backend doesn't work on Win64. At all.

threads plus loading Pari = boom.

A racing snail without its shell is just sluggish

Binomial example. Binomials $n = 1..300, $k = 1..$n/2

```
34.7s    bigint
34.3s    bigint lib=>”Pari”
30.7s    bigint lib=>”GMP”;
```

A lot of pure Perl code per op =
a lot of overhead per operation.

… but all hope is not lost

Binomial example. Binomials $n = 1..300, $k = 1..$n/2

```
34.7s    bigint
34.3s    bigint lib=>"Pari"
30.7s    bigint lib=>"GMP"

 1.4s    Math::Pari qw/:int/
 3.8s    Math::GMP qw/:constant/
 0.5s    Math::GMPz
```

A lot of pure Perl code per op = a lot of overhead per operation.

Math::Int64

```
use Math::Int64 "int64";
my ($n,$m) = ( int64(12), int64("28237489237498237 4") );
say $n * $m;
```

With a non-ancient C compiler, gives you fast 64-bit objects.

You could just install 64-bit Perl.

It has some nice extra features, like use integer but more.

Math::Int128

```
use Math::Int128 "int128";
my ($n,$m) = ( int128(12), int128("28237489237498237498") );
say $n * $m;
```

Must have 128-bit support in your C compiler (e.g. gcc) and architecture (x86_64 or Power).

Very similar to Math::Int64

Fast. Very similar to Math::GMPz

Big Integer Math Libraries

Roll your own

LibTomMath

Pari

GMP

gwnum

Math::Pari

```
use Math::Pari  qw/:int/;
```

Number one issue: no active maintainer.

Based on Pari 2.1.x line from 2000. Current Pari is 2.7.3.

Math::Pari

```
use Math::Pari  qw/:int/;
```

Number one issue: no active maintainer.

Based on Pari 2.1.x line from 2000. Current Pari is 2.7.3.

```
perl -MMath::Pari=isprime -E 'while (1) { die "22027*44053 is prime?  Cool story, bro!" if isprime(22027*44053); }'
```

```
22027*44053 is prime?  Cool story, bro!
```

Will tell you 9 is prime too. Fixed in Pari 2.3.0.

Math::Pari

Downloads code from 3rd party (Pari) ftp server

threads + Pari => immediate segfault

Doesn’t work on Win64

XS internals: “There is way way way too much crack-fuelled cheating.”

`(Nick Clark, p5p list, 2008)`

Used by classic Perl crypto modules. Replacements available.

Math::GMP

```
use Math::GMP  qw/:constant/;
```

Overall a good choice.

There are a few issues:

- more overhead than needed per call, but nothing compared to Math::BigInt.
- intify process can do wonky things with XS modules. Patch submitted.

Math::GMPz

Exposes GMP mpz functions to Perl.

No handholding, no auto-bigint option, no safety net if you call its functions with bad arguments.

Very low overhead. Fastest of all options.

I've found a couple very small bugs. Fixed and new release sent out within a day of submitting.

Lots of undocumented & unsupported functions for the curious.

Performance hacks for Math::BigInt

Make a Math::BigInt object only what has to be.

```
my $n = 50;                          # n is a native int
my $mult = Math::BigInt->bone;       # mult is bigint 1
```

Use two code paths for native vs. bigint, or by initial value

```
my $mult = ($n>20) ? Math::BigInt->bone : 1;
```

Performance hacks for Math::BigInt

Call functions directly:

```
my $q = $g->copy->bsub($r)->bdiv($w);   # ($g-$r)/$w
```

Always use binc and bdec, or ++ / - -

```
$n->binc();    # 4x faster than $n += 1
```

Predefine constants as bigints outside of loops.

Floating Point

Floating point is not easy to get right.

1991 ACM Computing Surveys article:

“What every computer scientist should know about floating point”

48 pages.

Floating Point Modules

Math::LongDouble		worth mentioning
Math::BigFloat	(and bignum)	in core. Slow.
Math::GMPf		Low level API
Math::MPFR		Low level API

As much Pi as you want

```
15min 51s    Math::BigFloat->bpi(10_000)
    26.1s    Math::BigFloat->bpi(10_000)     Pari backend
     0.4s    Math::BigFloat->bpi(10_000)     GMP backend
```

As much Pi as you want

15min 51s	Math::BigFloat->bpi(10_000)	
26.1s	Math::BigFloat->bpi(10_000)	Pari backend
0.4s	Math::BigFloat->bpi(10_000)	GMP backend
1min 37s	Math::BigFloat->bpi(100_000)	GMP backend
4hr 34min	Math::BigFloat->bpi(1_000_000)	GMP backend

As much Pi as you want

```
 15min 51s     Math::BigFloat->bpi(10_000)
       26.1s   Math::BigFloat->bpi(10_000)     Pari backend
        0.4s   Math::BigFloat->bpi(10_000)     GMP backend
  1min 37s     Math::BigFloat->bpi(100_000)    GMP backend
4hr 34min      Math::BigFloat->bpi(1_000_000)  GMP backend


        1.8s   GMP:  use ntheory; say length(Pi(1_000_000));
        1.3s   MPFR: use Math::MPFR;
                     my $pi = Math::MPFR::Rmpfr_init2(1_000_000*3.321923);
                     Math::MPFR::Rmpfr_const_pi($pi,MPFR_RNDZ);
                     say length($pi);
```

Summary

Integers:

- Math::BigInt or bigint [Simple, in core, slow]
- Math::GMP or Math::GMPz or Math::GMP qw/:constant/
  Recommended if you have or can install GMP

Floats:

- Math::BigFloat or bignum [Simple, in core, slow]
- Math::MPFR

Want:

- transparent Math::GMPz and Math::MPFR

Thank you, and Questions