Computing Bernoulli numbers

David Harvey (joint work with Edgar Costa)

University of New South Wales

27th September 2017
Jonathan Borwein Commemorative Conference
Noah's on the Beach, Newcastle, Australia

$$\zeta(2) = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \cdots = \frac{\pi^2}{6} = 1.644934\ldots$$

$$\zeta(2) = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \cdots = \frac{\pi^2}{6} = 1.644934\ldots$$

$$\zeta(4) = 1 + \frac{1}{2^4} + \frac{1}{3^4} + \cdots = \frac{\pi^4}{90} = 1.082323\ldots$$

$$\zeta(2) = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \cdots = \frac{\pi^2}{6} = 1.644934\ldots$$

$$\zeta(4) = 1 + \frac{1}{2^4} + \frac{1}{3^4} + \cdots = \frac{\pi^4}{90} = 1.082323\ldots$$

$$\zeta(6) = 1 + \frac{1}{2^6} + \frac{1}{3^6} + \cdots = \frac{\pi^6}{945} = 1.017343\ldots$$

$$\zeta(8) = 1 + \frac{1}{2^8} + \frac{1}{3^8} + \cdots = \frac{\pi^8}{9450} = 1.004077\ldots$$

$$\zeta(10) = 1 + \frac{1}{2^{10}} + \frac{1}{3^{10}} + \cdots = \frac{\pi^{10}}{93555} = 1.000994\ldots$$

$$\zeta(2) = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \cdots = \frac{\pi^2}{6} = 1.644934\ldots$$

$$\zeta(4) = 1 + \frac{1}{2^4} + \frac{1}{3^4} + \cdots = \frac{\pi^4}{90} = 1.082323\ldots$$

$$\zeta(6) = 1 + \frac{1}{2^6} + \frac{1}{3^6} + \cdots = \frac{\pi^6}{945} = 1.017343\ldots$$

$$\zeta(8) = 1 + \frac{1}{2^8} + \frac{1}{3^8} + \cdots = \frac{\pi^8}{9450} = 1.004077\ldots$$

$$\zeta(10) = 1 + \frac{1}{2^{10}} + \frac{1}{3^{10}} + \cdots = \frac{\pi^{10}}{93555} = 1.000994\ldots$$

$$\zeta(12) = 1 + \frac{1}{2^{12}} + \frac{1}{3^{12}} + \cdots = \frac{691\pi^{12}}{638512875} = 1.000246\ldots$$

Euler's formula for even n:

$$\zeta(n) = 1 + \frac{1}{2^n} + \frac{1}{3^n} + \cdots = (-1)^{n/2+1} \frac{B_n}{n!} (2\pi)^n,$$

where B_n are the *Bernoulli numbers* defined by

$$\frac{t}{e^t - 1} = \sum_{n=0}^{\infty} \frac{B_n}{n!} t^n.$$

The first few Bernoulli numbers:

$$B_0 = 1 \qquad B_1 = -1/2$$
$$B_2 = 1/6 \qquad B_3 = 0$$
$$B_4 = -1/30 \qquad B_5 = 0$$
$$B_6 = 1/42 \qquad B_7 = 0$$
$$B_8 = -1/30 \qquad B_9 = 0$$
$$B_{10} = 5/66 \qquad B_{11} = 0$$
$$B_{12} = -691/2730 \qquad B_{13} = 0$$
$$B_{14} = 7/6 \qquad B_{15} = 0$$

The first few Bernoulli numbers:

$$B_0 = 1 \qquad B_1 = -1/2$$
$$B_2 = 1/6 \qquad B_3 = 0$$
$$B_4 = -1/30 \qquad B_5 = 0$$
$$B_6 = 1/42 \qquad B_7 = 0$$
$$B_8 = -1/30 \qquad B_9 = 0$$
$$B_{10} = 5/66 \qquad B_{11} = 0$$
$$B_{12} = -691/2730 \qquad B_{13} = 0$$
$$B_{14} = 7/6 \qquad B_{15} = 0$$

A bigger Bernoulli number:

$$B_{100} = -94598037819122125295227433069493721872702841533066936133 / 385696204311395415197247711 / 33330$$

The denominator D_n of B_n is given by the von Staudt–Clausen formula:

$$D_n = \prod_{\substack{p \text{ prime} \\ p-1 \mid n}} p.$$

Example:

$$D_{100} = 2 \cdot 3 \cdot 5 \cdot 11 \cdot 101 = 33330.$$

The denominator D_n of B_n is given by the von Staudt–Clausen formula:

$$D_n = \prod_{\substack{p \text{ prime} \\ p-1 \mid n}} p.$$

Example:

$$D_{100} = 2 \cdot 3 \cdot 5 \cdot 11 \cdot 101 = 33330.$$

The numerator N_n of B_n is an integer with $n \log_2 n + O(n)$ bits.

Computing Bernoulli numbers is a very interesting problem.

Computing Bernoulli numbers is a very interesting problem.

> *"We have B_{96} and are well on the way towards B_{99}. I think that the average time required for each B will simmer down to about 20 hours. About* $1/3$ *of this time is used in typing results and* $1/10$ *of it is in checking. Of course, the final check (the exact division of a 250-digit number by a 50-digit number) would be sufficient, but coming as it does at the end of 20 hours it is necessary to check more frequently. We use as an additional check the casting out of 1000000001."*

(Letter from Dick Lehmer to Harry Vandiver, 1934, quoted in Corry 2008.)

What would Lehmer have thought of this?

```
sage: time x = bernoulli(10000)    # 28000 digits
Wall time: 9.68 ms
```

What would Lehmer have thought of this?

```
sage: time x = bernoulli(10000)    # 28000 digits
Wall time: 9.68 ms

sage: time x = bernoulli(20000)    # 61000 digits
Wall time: 60.9 ms

sage: time x = bernoulli(40000)    # 135000 digits
Wall time: 229 ms

sage: time x = bernoulli(80000)    # 294000 digits
Wall time: 982 ms

sage: time x = bernoulli(160000)   # 636000 digits
Wall time: 4.02 s
```

Implementation uses FLINT library (Bill Hart et al).

Notice the running time is roughly *quadratic* in n.

FLINT uses the *zeta function algorithm*:

- Approximate $\zeta(n)$ to precision of $O(n \log n)$ bits using the Euler product:

$$\zeta(n) = \prod_p \left(1 - \frac{1}{p^n}\right)^{-1}.$$

  To get enough bits, suffices to include primes up to about $n/(2\pi e)$.

FLINT uses the *zeta function algorithm*:

- Approximate $\zeta(n)$ to precision of $O(n \log n)$ bits using the Euler product:

$$\zeta(n) = \prod_p \left(1 - \frac{1}{p^n}\right)^{-1}.$$

  To get enough bits, suffices to include primes up to about $n/(2\pi e)$.
- Approximate π^n and $n!$ to precision of $O(n \log n)$ bits.

FLINT uses the *zeta function algorithm*:

- Approximate $\zeta(n)$ to precision of $O(n \log n)$ bits using the Euler product:

$$\zeta(n) = \prod_p \left(1 - \frac{1}{p^n}\right)^{-1}.$$

  To get enough bits, suffices to include primes up to about $n/(2\pi e)$.
- Approximate π^n and $n!$ to precision of $O(n \log n)$ bits.
- Compute denominator D_n using von Staudt–Clausen formula.

FLINT uses the *zeta function algorithm*:

- Approximate $\zeta(n)$ to precision of $O(n \log n)$ bits using the Euler product:

$$\zeta(n) = \prod_p \left(1 - \frac{1}{p^n}\right)^{-1}.$$

  To get enough bits, suffices to include primes up to about $n/(2\pi e)$.
- Approximate π^n and $n!$ to precision of $O(n \log n)$ bits.
- Compute denominator D_n using von Staudt–Clausen formula.
- Use Euler's formula to deduce real approximation to B_n, and hence recover N_n exactly.

FLINT uses the *zeta function algorithm*:

- Approximate $\zeta(n)$ to precision of $O(n \log n)$ bits using the Euler product:

$$\zeta(n) = \prod_p \left(1 - \frac{1}{p^n}\right)^{-1}.$$

  To get enough bits, suffices to include primes up to about $n/(2\pi e)$.
- Approximate π^n and $n!$ to precision of $O(n \log n)$ bits.
- Compute denominator D_n using von Staudt–Clausen formula.
- Use Euler's formula to deduce real approximation to B_n, and hence recover N_n exactly.

All steps run in $n^{1+o(1)}$ time, except possibly the first.

Example for $n = 100$:

$$2^{100} = 1267650600228229401496703205376$$

$$3^{100} = 515377520732011331036461129765621272702107522001$$

$$5^{100} = 78886090522101180541172856528278622967320643510902300477027893066406 25$$

$$7^{100} = 3234476509624757991344647769100216810857203198904625400933895331391691459636928060001$$

Example for $n = 100$:

$$2^{100} = 1267650600228229401496703205376$$

$$3^{100} = 515377520732011331036461129765621272702107522001$$

$$5^{100} = 78886090522101180541172856528278622967320643510902300477027893066406 25$$

$$7^{100} = 3234476509624757991344647769100216810857203198904625400933895331391691459636928060001$$

Corresponding factors in Euler product for $\zeta(100)$:

$$\left(1 - \frac{1}{2^{100}}\right)^{-1} = 1.000000000000000000000000000000078886090522101180541172856528340853120099254927973 74$$

$$\left(1 - \frac{1}{3^{100}}\right)^{-1} = 1.000000000000000000000000000000000000000000000000019403252174826328375885060288046504$$

$$\left(1 - \frac{1}{5^{100}}\right)^{-1} = 1.000000000000000000000000000000000000000000000000000000000000000000000000001267650600228$$

$$\left(1 - \frac{1}{7^{100}}\right)^{-1} = 1.000000000000000000000000000000000000000000000000000000000000000000000000000000000000000$$

What is the complexity?

- There are $O(n/\log n)$ primes up to $n/(2\pi e)$.
- For each prime, it costs $O(M(n \log n))$ bit operations to compute the Euler factor and accumulate it into the running product.

What is the complexity?

- There are $O(n/\log n)$ primes up to $n/(2\pi e)$.
- For each prime, it costs $O(M(n \log n))$ bit operations to compute the Euler factor and accumulate it into the running product.

Total complexity, assuming FFT multiplication:

$$n^2(\log n)^{1+o(1)}.$$

This explains the quadratic behaviour of FLINT.

In 2010 I published another quasi-quadratic algorithm for computing B_n, called the *multimodular algorithm*.

It has nothing to do with the zeta function or Euler's formula.

In 2010 I published another quasi-quadratic algorithm for computing B_n, called the *multimodular algorithm*.

It has nothing to do with the zeta function or Euler's formula.

Instead, it uses *Voronoi congruences*.

For example, if p is a prime and $2^n \neq 1 \pmod{p}$ then

$$B_n \equiv \frac{n}{2(2^n-1)}\left(1^{n-1} - 2^{n-1} + 3^{n-1} - \cdots - (p-1)^{n-1}\right) \pmod{p}.$$

Using congruences of this type, we can evaluate $B_n \pmod{p}$ using $O(p)$ operations in $\mathbf{F}_p$.

We do this for all $p < n \log_2 n + O(n)$, then reconstruct B_n using the Chinese remainder theorem.

For example, for $n = 10000$, we need all primes up to 64013.

We do this for all $p < n \log_2 n + O(n)$, then reconstruct B_n using the Chinese remainder theorem.

For example, for $n = 10000$, we need all primes up to 64013.

Theoretical complexity is

$$n^2 (\log n)^{2+o(1)}.$$

In theory this is a factor of $\log n$ worse than the zeta function algorithm.

We do this for all $p < n \log_2 n + O(n)$, then reconstruct B_n using the Chinese remainder theorem.

For example, for $n = 10000$, we need all primes up to 64013.

Theoretical complexity is

$$n^2 (\log n)^{2+o(1)}.$$

In theory this is a factor of $\log n$ worse than the zeta function algorithm.

In practice, it seems to perform better than zeta function algorithm for large n, say $n > 10^6$ or 10^7 or so.

The main reason for this is bettter locality. Each prime can be handled in almost zero space, whereas the zeta function algorithm is constantly manipulating enormous integers.

Edgar Costa and I have recently been thinking about how to *combine* information from these two algorithms.

For example, we could:

- Run the zeta function algorithm at half the target precision, to get real approximation to B_n.
- Run multimodular algorithm for half of the primes, to get modular information about B_n.

Combining these two pieces of information, we can reconstruct B_n.

What does this do to the running time?

Recall that to run the zeta function algorithm at full precision, we need to include Euler factors for $O(n/\log n)$ primes.

To get half the precision, we only need $O(\sqrt{n}/\log n)$ primes!

The small primes contribute much more information than the big primes.

Recall that to run the zeta function algorithm at full precision, we need to include Euler factors for $O(n/\log n)$ primes.

To get half the precision, we only need $O(\sqrt{n}/\log n)$ primes!

The small primes contribute much more information than the big primes.

Example: $n = 10000$.

- For full precision, need Euler factors for $p \leq 587$.
- For half precision, only need $p \leq 23$!!

This is basically because 23^{10000} has half as many bits as 587^{10000}.

Recall that to run the zeta function algorithm at full precision, we need to include Euler factors for $O(n/\log n)$ primes.

To get half the precision, we only need $O(\sqrt{n}/\log n)$ primes!

The small primes contribute much more information than the big primes.

Example: $n = 10000$.

- For full precision, need Euler factors for $p \leq 587$.
- For half precision, only need $p \leq 23$!!

This is basically because 23^{10000} has half as many bits as 587^{10000}.

Conclusion: we can compute half the bits in only $n^{3/2}(\log n)^{1+o(1)}$ time, which is asymptotically negligible!

On the other hand, for the multimodular algorithm, computing half of the bits only takes $1/4$ of the time needed to compute all the bits.

The small primes are cheaper than the big primes.

Conclusion: by combining the two algorithms, asymptotically we save a factor of 4!

On the other hand, for the multimodular algorithm, computing half of the bits only takes 1/4 of the time needed to compute all the bits.

The small primes are cheaper than the big primes.

Conclusion: by combining the two algorithms, asymptotically we save a factor of 4!

In fact, by adjusting the fraction of bits allocated to each algorithm, we can (asymptotically) save *any desired constant factor*!

Obvious question: what if we let the fraction vary with n? What is the best choice?

Suppose we compute fraction α of the bits with multimodular algorithm, and fraction $1 - \alpha$ with zeta function algorithm.

One can show that the zeta function algorithm contributes

$$n^{2-\alpha}(\log n)^{1+o(1)},$$

and multimodular algorithm contributes

$$\alpha^2 n^2 (\log n)^{2+o(1)}.$$

Suppose we compute fraction α of the bits with multimodular algorithm, and fraction $1 - \alpha$ with zeta function algorithm.

One can show that the zeta function algorithm contributes

$$n^{2-\alpha}(\log n)^{1+o(1)},$$

and multimodular algorithm contributes

$$\alpha^2 n^2 (\log n)^{2+o(1)}.$$

Optimal choice of α turns out to be around $\log\log n / \log n$. With this choice, contribution from both parts is

$$n^2 (\log n)^{o(1)}.$$

This is a factor of about $\log n$ faster than the zeta function algorithm!

Does it actually work?

Example: compute B_{10^6}.

- FLINT: 170s
- PARI: 190s (also uses zeta function algorithm)
- multimodular: 175s

Does it actually work?

Example: compute B_{10^6}.

- FLINT: 170s
- PARI: 190s (also uses zeta function algorithm)
- multimodular: 175s

I wrote a not-so-optimised implementation of the "combination" algorithm.

Tried several values of α, best seems to be $\alpha = 0.32$.

Runs in 36.5s (speedup 4.7x). More precisely:

- 5.6s computing π^n, $n!$, etc
- 8.7s evaluating Euler product
- 22.1s for multimodular computation

Actually... this is not the asymptotically fastest algorithm known.

In 2014 I published an algorithm that computes B_n in time

$$n^{4/3}(\log n)^{11/3+o(1)}.$$

Actually... this is not the asymptotically fastest algorithm known.

In 2014 I published an algorithm that computes B_n in time

$$n^{4/3}(\log n)^{11/3+o(1)}.$$

It is based on a "prime power" version of the Voronoi congruence.

It simultaneously computes $B_n \pmod{p^\lambda}$ for all primes up to $n^{1/3}(\log n)^{2/3}$, for λ around $n^{2/3}(\log n)^{1/3}$, using fast polynomial evaluation techniques.

Actually... this is not the asymptotically fastest algorithm known.

In 2014 I published an algorithm that computes B_n in time

$$n^{4/3}(\log n)^{11/3+o(1)}.$$

It is based on a "prime power" version of the Voronoi congruence.

It simultaneously computes $B_n \pmod{p^\lambda}$ for all primes up to $n^{1/3}(\log n)^{2/3}$, for λ around $n^{2/3}(\log n)^{1/3}$, using fast polynomial evaluation techniques.

To the best of my knowledge, a complete implementation does not yet exist.

Edgar and I have been working on it, on and off for a few years.

So now we have three algorithms:

- Zeta function algorithm
- Multimodular algorithm
- Prime power algorithm

They all compute independent information (more or less), so they could be combined.

So now we have three algorithms:

- Zeta function algorithm
- Multimodular algorithm
- Prime power algorithm

They all compute independent information (more or less), so they could be combined.

Unfortunately, in theory this yields at best a constant speedup over the prime power algorithm.

The reason is: as soon as we compute more than $1/3$ of the bits using the Euler product, the complexity is already worse than $n^{4/3}$. We would be better off using the prime power algorithm alone.

In practice, for feasible values of n, I expect that some combination of all three algorithms will be best.

We won't know until we try it!

Thank you!