An algorithm and computation to verify Legendre's Conjecture up to $7 \cdot 10^{13}$

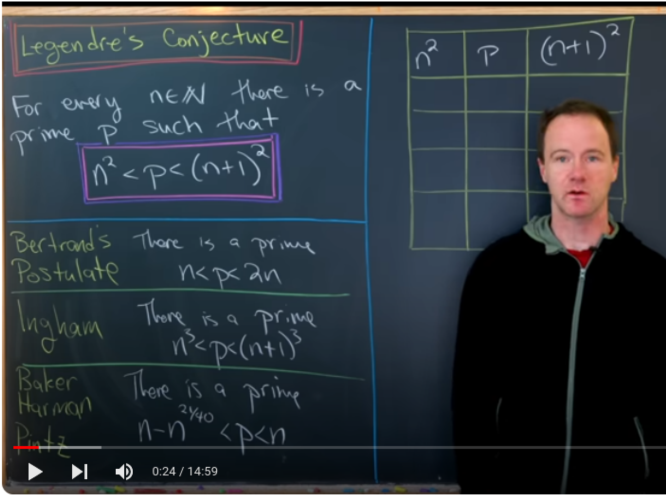
Jon Sorenson and Jonathan Webster

Computer Science & Software Engineering || Mathematical Sciences
Butler University, Indiana USA
jsorenso@butler.edu jewebste@butler.edu,

16th Algorithmic Number Theory Symposium, MIT, July 2024

Thanks to Frank Levinson for supporting Butler computing facilities and thanks to the Holcomb Awards Committee for financial support

Motivation: Michael Penn



Legendre's Conjecture is PROBABLY TRUE, and here's why

List of Conjectures

Legendre: $n^2 < p < (n+1)^2$.

Andrica: $p_{n+1} - p_n < 2\sqrt{p_n} + 1$ or $\sqrt{p_{n+1}} - \sqrt{p_n} < 1$.

Oppermann: $n^2 < p < n(n+1) < q < (n+1)^2$.

Cramér: $p_{n+1} - p_n \ll (\log p_n)^2$.

What Has Been Proven

Baker, Harman, Pintz (2000)	$x - x^{0.525} < p < x$
Cully-Hugill (2023)	$n^3 < p < (n+1)^3$ for $n > e^{e^{32.892}}$
Cully-Hugill, Johnston (2024)	$n^{90} < p < (n+1)^{90}$
Oliviera e Silva, Herzog, Pardi (2014)	$p_{n+1} - p_n \ll (\log p_n)^2$ for $p_{n+1} \leq 4 \times 10^{18}$

Going Bigger – Our Work

- An algorithm to verify Oppermann's conjecture:
  - $O(N \log N \log\log N)$ heuristic running time
  - $N^{O(1/\log\log N)}$ space

  Correctness is unconditional, including all prime tests
- Oppermann's conjecture holds for all $n \leq N = 7.05 \times 10^{13} \approx 2^{46}$.

  (We found 27-digit primes.)

Two Useful Lemmas

Cramér's Model: An integer $n \leq x$ is prime with probability $1/\log x$.

Lemma (2.1)

Assuming Cramér's model, the probability $\log n \log v$ *integers near* n *are all composite is* $O(1/v)$*, for large* n*.*

We obtain Cramér's conjecture by setting $v = n$.

Lemma (2.2)

Assuming Cramér's model, let M *be a multiple of* $\prod_{p \leq b} p$*, then the probability* $(\log n \log v)/\log b$ *integers near* n *relatively prime to* M *are all composite is* $O(1/v)$*, for large* n, b*.*

Two Useful Llammas

Ideas: Algorithm A

Basic Idea: Test $n^2+1, n^2+2, \cdots$ for primality. Etc.

- Filter with trial division and base-2 strong pseudoprime tests.
- Which prime test? Unconditional proof of primality required.
- This parallelizes nicely.
- We can get $O(N(\log N)^3)$ time with the pseudosquares prime test of Lukes, Patterson, Williams (1996).

Faster would be better.

Ideas: Algorithm B

Finding all primes up to N^2 using a seive would take $O(N^2/\log\log N)$ time. Extending the work of Oliviera e Silva, Herzog, Pardi (2014) is not feasible.

Basic Idea: Sieve an arithmetic progression!

- Set a modulus M so there are about $(2\log N)^2$ integers that are 1 mod M between n^2 and $n(n+1)$ for $n \le N$, so $N/M \approx 2N(2\log N)^2$.
- M should be divisible by $2 \cdot 3 \cdot 5 \cdots$.
- Sieving 1 mod M up to N^2 will take $O(N(\log N)^2/\log\log N)$ time using the Atkin-Bernstein (2004) sieve.
- Faster, but this needs $\sqrt{N^2}$ space.

Heuristic lower bound: $(\log n)/\log\log n$ per prime, or $\Omega(N(\log N)/\log\log N)$ total time.

Ideas: Algorithm C

Basic Idea: Use Algorithm A on an arithmetic progression.

- Choose a prime R with with $R > (N^2)^{1/3} = N^{2/3}$.
- Set M, the sieve modulus, to $M := R \cdot 2 \cdot 3 \cdot 5 \cdots$.
- Use the Brillhart-Lehmer-Selfridge (BLS) prime test.

This matches the running time of Algorithm B, but with very little space.

BLS Code

```
// Theorem 4.1.5 from Crandall and Pomerance
// Brillhart-Lehmer-Selfridge
// assumes R is prime and n^1/3 < R < n^1/2, R | n-1
bool primetest(int128 n, int64_t R)
{
  countprimetests++;
  int128 q=(n-1)/R;

  //cout << "In primetest: n="<<n<<endl;

  bigint2mpz(Q,q);
  bigint2mpz(N,n);

  mpz_mul_ui(N,Q,R);
  mpz_add_ui(N,N,1);

  //cout << "R="<<R<<" Q="<<Q<<" N="<<N<<endl;

  //check gcd(a^(n-1)/R -1, n)=1
  mpz_powm(save,two,Q,N);
  mpz_sub_ui(answer,save,1);
  mpz_gcd(answer,answer,N);
  int64_t ans=mpz_get_si(answer);
  //cout << "GCD check, ans="<<ans<<endl;
  if(ans!=1) return false;

  //check a^(n-1) mod n ==1
  mpz_powm_ui(answer,save,R,N);
  ans=mpz_get_si(answer);
  //cout << "Fermat check, ans="<<ans<<endl;
  if(ans!=1) return false;

  //write n = c2*R^2 + c1*R + a0
  //check c1^2-4*c2 not a square

  int64_t c1= q % R;
  int64_t c2= (q-c1)/R;
  //cout << "c1="<<c1<<" c2="<<c2<<endl;
  bool result=!issquare(((int128)c1)*c1 - 4*c2);
  //cout << "Square check, ans="<<result<<endl;
  return result;
}
```

Our Algorithm

Assorted Parameters

$B := N^{c/\log\log N}$, $c > 0$, the prime bound for sieving;
$b := 0.2\log N$, the small prime bound for the modulus M;
$s := \log n$ spots in an interval;
$t := B/(2s)$, the batch or segment size

Setup

For each i, $0 \le i < t$:

- There are at least s integers between $(n+i)^2$ and $(n+i)(n+i+1)$ that are 1 mod M
- There are at least s integers between $(n+i)(n+i+1)$ and $(n+i+1)^2$ that are 1 mod M
- The probability all s integers are composite is $O(1/\log n)$ by Lemma 2.2

Our Algorithm

Basic Outline

Repeat N/t times (sequentially or in parallel):

1. Set a prime $R \geq (n+t)^{2/3}$ and $M = R \cdot 2 \cdot 3 \cdots$.
2. Sieve the segment $[n^2, (n+t)^2]$ on the arithmetic progression 1 mod M by primes up to B. The segment size is
$$\frac{(n+t)^2 - n^2}{M} \geq 2ts = 2t \log n \approx B.$$
3. On each interval, test numbers in the arithmetic progression that passed the sieve for primality using BLS until a prime is found. Probability of failure is $O(1/\log n)$.
4. Apply Algorithm C using up to 4 different Rs if previous step failed. If all that fails, use Algorithm A.

Running time: $O(N(\log N) \log \log N)$.

The Computation

We verified Oppermann's conjecture up to $N = 7.05 \times 10^{13} > 2^{46}$.

The Computation

We used the following parameter choices in practice:

- $s = 128$ (note $\log(10^{27}) \approx 62$)
- $B = 2^{17} = 13172$, so segments easily fit in cache
- t varied with M but was around 450
- A list of about 50 primes R of different sizes was precomputed
- M was always divisible by $30 = 2 \cdot 3 \cdot 5$

Details

Our code ran on two platforms (both Linux):

1. A server with 4 Intel Phi Co-processors with 64 cores each
2. A cluster of 192 cores (Butler's Big Dawg)

The code is in C++, using MPI and GMP in places.

Some stats:

- BLS prime tests returned **true** about 33% of the time.
- The failure rate was only 0.004%.
- The first alternate prime R with BLS was always successful at finding a prime.

The End

Thank you!

Questions?