MATHEMATICS OF COMPUTATION
Volume 83, Number 290, November 2014, Pages 3071–3091
S 0025-5718(2014)02800-7
Article electronically published on January 27, 2014

A SEARCH FOR WILSON PRIMES

EDGAR COSTA, ROBERT GERBICZ, AND DAVID HARVEY

Abstract. A Wilson prime is a prime p such that $(p-1)! = -1 \pmod{p^2}$. We report on a search for Wilson primes up to 2×10^{13}, and describe several new algorithms that were used in the search. In particular, we give the first known algorithm that computes $(p-1)! \pmod{p^2}$ in average polynomial time per prime.

1. Introduction

Wilson's theorem in elementary number theory states that

$$(p-1)! = -1 \pmod{p}$$

for any prime p. The corresponding *Wilson quotient* is

$$\frac{(p-1)!+1}{p} \in \mathbf{Z},$$

and we define w_p to be its residue modulo p in the interval $-p/2 \leq w_p < p/2$. A *Wilson prime* is a prime such that $w_p = 0$ or, equivalently,

$$(p-1)! = -1 \pmod{p^2}.$$

Only three Wilson primes are known: 5, 13 and 563.

All previously published searches for Wilson primes have used algorithms for computing w_p whose time complexity is essentially linear in p. (In this paper, unless otherwise specified, time complexity means number of steps on a multitape Turing machine; see [27].) Since the input size is proportional to $\log p$, these algorithms should be regarded as having exponential time complexity. For example, the simplest possible algorithm is to multiply successively by the integers $2, 3, \ldots, p-1$, reducing modulo p^2 after each multiplication. The best known algorithm for computing w_p has complexity $p^{1/2+o(1)}$ (see below), but this is still exponential in $\log p$.

The main theoretical contribution of this paper is an algorithm that computes w_p in *polynomial time on average*:

Theorem 1.1. *The Wilson quotients w_p for $2 \leq p \leq N$ may be computed in time $N \log^{3+o(1)} N$.*

Let $\pi(x)$ denote the number of primes $p \leq x$. By the prime number theorem, $\pi(x) \sim x/\log x$, so Theorem 1.1 implies that we can compute each w_p in time essentially $\log^4 p$ on average. While this result does not improve the complexity for

Received by the editor October 31, 2012 and, in revised form, December 5, 2012, January 31, 2013 and February 3, 2013.

2010 *Mathematics Subject Classification.* Primary 11A07; Secondary 11Y16.

The first author was partially supported by FCT doctoral grant SFRH/BD/ 69914/2010.

The third author was partially supported by the Australian Research Council, DECRA Grant DE120101293.

computing a single w_p, it is of course directly relevant to the problem of searching for Wilson primes.

The key idea of the new algorithm is to exploit redundancies among the products $(p-1)!$ for varying p. For example, the Wilson quotients for $N < p < 2N$ in some sense all incorporate the product $N!$. Instead of computing $N! \pmod{p^2}$ separately for each p, we will compute it modulo the product $\prod_{N<p<2N} p^2$. A remainder tree then yields $N! \pmod{p^2}$ for each p. Using FFT methods for integer arithmetic, this can all be achieved in average polynomial time per prime. Applying this idea recursively leads to an algorithm for computing the desired residues $(p-1)! \pmod{p^2}$. A detailed description is given in the proof of Theorem 1.1 in Section 2.

However, the space requirements of this algorithm render it impractical for large N, and we must implement a time-space tradeoff to obtain a practical algorithm:

Theorem 1.2. *Let $M < N$, and assume that $N - M > \sqrt{N} \log N \log\log N$. The Wilson quotients w_p for $M < p \leq N$ may be computed in time*

$$M \log^{2+o(1)} M + (N - M + \sqrt{N}) \log^{3+o(1)} N$$

and space $O(N - M)$.

The algorithm implementing Theorem 1.2 consists of two main phases that we call Stage 1 and Stage 2. Stage 1 involves computing $M!$ modulo $\prod_{M<p\leq N} p^2$, and contributes the $M \log^{2+o(1)} M$ term to the time bound. Stage 2, which contributes the second term, is a modification of the algorithm implementing Theorem 1.1.

The average time per prime in Stage 2 is essentially $\log^4 p$, the same as for Theorem 1.1. However, in Stage 1 the average time per prime behaves like

$$\frac{p}{N-M} \log^3 p.$$

This is no longer polynomial in $\log p$, and represents the price we pay for restricting the space consumption. If we now assume that the amount of RAM is fixed, then a reasonable strategy to compute w_p for all p up to some bound N_0 is to apply Theorem 1.2 to successive intervals $M < p \leq N$, where $N \leq N_0$, and where $N - M$ is chosen as large as possible given the available RAM.

This is in fact what we did for all $p < 2 \times 10^{13}$. We found no new Wilson primes up to this bound. Altogether this consumed over 1.1 million hours of CPU time. It is traditional, though meaningless, to give tables of "near misses". Table 1 shows the smallest $|w_p|$ that we found, and Table 2 shows the smallest residues when ordered by $|w_p/p|$.

Retaining all of the residues would have required archival storage in the terabyte range. Instead, we only recorded those residues for which $|w_p| \leq p/50000$, i.e., approximately 0.004% of the primes examined. There are 27 039 026 such primes; the residues may be downloaded from the third author's web page (247 MB).

The search for Wilson primes has an interesting history. The case $p = 5$ is trivial, and $p = 13$ was noticed at least as early as 1892 [26, p. 318]. In 1913, Beeger used the congruence

$$w_p = B_{p-1} - \frac{p-1}{p} \pmod{p},$$

where B_k is the k-th Bernoulli number, together with a published table of Bernoulli numbers, to check that there are no other Wilson primes less than 114 [2]. Several

TABLE 1. Primes $10^6 < p < 2 \times 10^{13}$ for which $|w_p| \leq 10$

p	w_p	p	w_p
56 151 923	−1	4 036 677 373	−5
11 774 118 061	−1	5 609 877 309 359	−6
14 296 621	+2	10 746 881	−7
87 467 099	−2	11 892 977	−7
16 556 218 163 369	+2	39 198 017	−7
8 315 831	+3	1 767 839 071	+8
93 559 087	−3	29 085 907	+9
51 802 061	+4	67 133 912 011	+9
258 818 504 023	+4	42 647 052 491	+10
1 239 053 554 603	−4	935 606 702 249	−10
1 987 272 877	+5		

TABLE 2. Primes $p < 2 \times 10^{13}$ for which $|w_p/p| \leq 1.5 \times 10^{-11}$

p	w_p	p	w_p
5	0	17 475 368 544 847	+154
13	0	13 561 740 531 809	+120
563	0	9 461 354 987 597	+94
16 556 218 163 369	+2	13 707 091 918 909	+143
5 609 877 309 359	−6	935 606 702 249	−10
14 875 476 519 749	−38	1 108 967 825 921	+12
15 395 725 531 427	+46	2 170 161 095 393	+25
1 239 053 554 603	−4	16 690 620 863 071	+203
4 663 421 363 459	+28	2 462 223 083 147	−35
7 746 014 299 613	+47	17 524 177 394 617	+256
11 273 815 078 217	+88	10 865 903 332 033	+159
7 338 481 259 891	−62	16 880 979 600 449	+253

years later he proved the congruence

$$(p-1)! = (-1)^{(p-1)/2} \left(\left(\frac{p-1}{2} \right)! \right)^2 (2^p - 1) \pmod{p^2}, \tag{1.1}$$

which reduces computation of w_p to that of $((p-1)/2)! \pmod{p^2}$. He used this identity, together with a direct computation of the relevant factorials, to produce a table of w_p for $p < 300$ [3]. We do not know when (1.1) was first discovered, but it appears (without proof) in [26].

Lehmer later used Beeger's original method together with a newly extended table of Bernoulli numbers to compute w_p for $p \leq 211$ [23]. In a companion article, she mentions that Beeger communicated that his earlier table contains four errors, namely for $p = 127$, 167, 173 and 241 [24]. Lehmer's table is correct, but there is an additional unnoticed error in Beeger's table, for $p = 239$. The errors are rather clustered together, and one speculates on the human factors (computational exhaustion?) that may have been responsible. For the modern reader, it is very easy to forget just how much effort is required to generate such a table by hand. We invite the reader to spend a few minutes verifying that $p = 13$ is indeed a Wilson prime!

After these early attempts, the search entered the computer age with the work of Goldberg, who used the Bureau of Standards Eastern Automatic Computer (SEAC), one of the first stored-program electronic computers, to test all $p < 10\,000$ [16]. In this interval, not far beyond the previous search bound, was found the third Wilson prime $p = 563$. Fröberg pushed this further to 30 000 and then 50 000 [13, 14]. In [14] he also discusses a heuristic concerning the distribution of Wilson primes. Namely, if one assumes that w_p is uniformly distributed modulo p, then the probability that p is a Wilson prime is $1/p$, and the expected number of Wilson primes less than X is

$$\sum_{p<X} \frac{1}{p} = \log\log X + c + o(1),$$

where $c = 0.2615...$ is Mertens' constant. This suggests that there should be infinitely many Wilson primes, but that they should be very rare.

The search bound was successively increased to 200 183 by Pearson [28], 1 017 000 by Kloss [20], 3 000 000 by Keller (see [29, p. 350]), 4 000 000 by Dubner [12], 10 000 000 and then 18 876 041 by Gonter and Kundert [21]. (The computation was halted at 18 876 041 due to a power failure; see [29, p. 350]. Many authors have cited an unpublished manuscript "All prime numbers up to 18,876,041 have been tested without finding a new Wilson prime" by Gonter and Kundert, but we have been unable to locate a copy.)

None of these authors give many details on how they performed the computation. It seems likely that they were all aware of (1.1), and that they computed $((p-1)/2)! \pmod{p^2}$ by simply multiplying successively by $2, 3, \ldots, (p-1)/2$, reducing modulo p^2 at frequent intervals.

Significant algorithmic progress on the problem was made by Crandall, Dilcher and Pomerance, who searched up to 5×10^8 [10]. They introduced two new main ideas. The first is that for many p, there exist identities better than (1.1). For example, if $p = 1 \pmod 4$, write $p = a^2 + b^2$ with $a = 1 \pmod 4$. Then we have the remarkable identity (proved in [8])

$$\binom{\frac{1}{2}(p-1)}{\frac{1}{4}(p-1)} = \left(1 + \frac{2^{p-1}-1}{2}\right)\left(2a - \frac{p}{2a}\right) \pmod{p^2}.$$

Together with (1.1) this reduces the computation of w_p to that of $((p-1)/4)! \pmod{p^2}$. Similar identities are used in [10] to reduce to computation of $((p-1)/6)! \pmod{p^2}$ in the case that $p = 1 \pmod 6$.

We extend this technique considerably in Section 3, showing how to reduce to computation of $((p-1)/e)! \pmod{p^2}$ for essentially any "small" divisor e of $p-1$.

Second, [10] introduced a scheme that replaces most of the modular multiplications by modular additions. Indeed, they show how to compute $N! \pmod{p^2}$ using $N + O(N^{2/3})$ additions and only $O(N^{2/3})$ multiplications. This optimisation does not play a role in the present work.

Crandall, Dilcher, and Pomerance also mention an algorithm, essentially due to Strassen, that computes $(p-1)! \pmod{p^2}$ in time $p^{1/2+o(1)}$; however, they found it was not competitive with their quasi-linear time algorithm over the range of their search. This can be improved by a factor of $\log p$ [6], yielding the best known algorithm for computing a single w_p.

Following this work, Carlisle, Crandall, and Rodenkirch extended the search to 10^9 in 2006 (see [30, p. 241]) and then 6×10^9 in 2008 (personal communication). This work has not been published; we sketch their algorithm here. The basic idea is to explicitly compute the exponents appearing in the prime factorisation $N! = p_1^{e_1} \cdots p_r^{e_r}$, and then compute this product, term by term, modulo p^2. The complexity is $O(N/\log N)$ multiplications, which improves on the algorithms used in [10] by a factor of $\log N$.

2. Computing Wilson quotients in average polynomial time

In this section we give algorithms that prove Theorems 1.1 and 1.2. The algorithms depend on three fundamental operations: integer multiplication, integer division, and enumeration of primes. We discuss the complexity of these operations first. We will give only a high level description of all algorithms, allowing the industrious reader to supply their own details concerning data layout and access patterns by the Turing machine.

If X and Y are integers with at most N bits, their product can be computed in time $N \log^{1+o(1)} N$ and space $O(N)$ using FFT methods [15, 31]. For division with remainder, we want $Q = \lfloor X/Y \rfloor$ (assuming $Y > 0$) and $R = X \bmod Y$. These can also be computed in time $N \log^{1+o(1)} N$ and space $O(N)$ [5].

Consider the problem of enumerating the primes $M < p \le N$. In our implementation (see Section 4) we used a simple sieve of Eratosthenes, i.e., after precomputing a table of primes $q \le \sqrt{N}$, we initialise a bit-array of length $N - M$ and strike out multiples of each q to eliminate the composites. Assuming a RAM model with unit time access to arbitrary array elements, and in which integers of size $O(\log N)$ can be manipulated in unit time, the complexity is at most

$$\sum_{\substack{q \le \sqrt{N} \\ q \text{ prime}}} \left\lceil \frac{N-M}{q} \right\rceil \le \sum_{\substack{q \le \sqrt{N} \\ q \text{ prime}}} \left(\frac{N-M}{q} + 1 \right) = O((N-M)\log\log N + \sqrt{N})$$

by Mertens' theorem.

While this simple algorithm is perfectly adequate in practice, in the Turing model the analysis is incorrect, because of the unavailability of unit-time array access. For completeness, Proposition 2.2 below gives a bound for the Turing model, following the approach suggested in [32, p. 226]. This result is not optimal, but suffices for our purposes. The key tool is merge sort, which can be implemented efficiently on a Turing machine; see [11] for a discussion of this, and for further applications of this observation in computational number theory.

Proposition 2.1. *The primes $p \le N$ may be enumerated in time*

$$O(N \log^2 N \log\log N)$$

and space

$$O(N \log N \log\log N).$$

Proof. First enumerate the primes $q \le \sqrt{N}$ by trial division. There are $O(\sqrt{N})$ candidates, and each requires $O(N^{1/4})$ divisibility tests, so the time cost is $N^{3/4+o(1)}$.

Now for each $q \leq \sqrt{N}$, generate the multiples of q bounded by N. The number of such multiples is $d = \sum_{q \leq \sqrt{N}} \lfloor N/q \rfloor = O(N \log\log N)$. Each successive multiple is computed via a single addition of integers of size $O(\log N)$, so the time and space required to construct the list is $O(d \log N)$. Sort the list using merge sort; this costs time $O(d \log d \log N) = O(N \log^2 N \log\log N)$ and space $O(d \log N) = O(N \log N \log\log N)$. The complement of the resulting list in $1 \leq x \leq N$ is the desired set of primes, and can be computed in one more pass in time $O(d \log N)$. □

Proposition 2.2. *The primes $M < p \leq N$ may be enumerated in time*

$$O((N - M + \sqrt{N}) \log^2 N \log\log N)$$

and space

$$O((N - M + \sqrt{N}) \log N \log\log N).$$

Proof. First enumerate the primes $q \leq \sqrt{N}$ using Proposition 2.1. This requires time $O(\sqrt{N} \log^2 N \log\log N)$ and space $O(\sqrt{N} \log N \log\log N)$.

Now for each $q \leq \sqrt{N}$, generate the multiples of q in the interval $M < x \leq N$. Determining the first multiple of each q, namely $q\lceil (M+1)/q \rceil$, costs $O(\log^2 N)$ per prime (assuming naive arithmetic), so $O(\sqrt{N} \log^2 N)$ altogether. The number of such multiples is

$$\begin{aligned} d \leq \sum_{q \leq \sqrt{N}} \lceil (N-M)/q \rceil &\leq \sum_{q \leq \sqrt{N}} (N-M)/q + 1 \\ &= O((N-M) \log\log N + \sqrt{N}) = O((N - M + \sqrt{N}) \log\log N). \end{aligned}$$

The proof is concluded in the same way as Proposition 2.1. □

Having dealt with these preliminaries, we now turn to computing Wilson quotients. First we give a simple algorithm that proves Theorem 1.1, and which will serve as a template for the more involved algorithm needed for the proof of Theorem 1.2. The structure of the computation bears some similarity to the parallel prefix tree in [7].

Proof of Theorem 1.1. First use Proposition 2.1 to enumerate the primes $p \leq N$ in time $N \log^{2+o(1)} N$.

Let $d = \lceil \log_2 N \rceil$. For each $0 \leq i \leq d$ and $0 \leq j < 2^i$ let

$$U_{i,j} = \left\{ k \in \mathbf{Z} : j\frac{N}{2^i} < k \leq (j+1)\frac{N}{2^i} \right\}.$$

Thus $U_{i,0}, \ldots, U_{i,2^i-1}$ partition the interval $0 < k \leq N$ into 2^i sets of roughly equal size. For $0 \leq i < d$ we have the disjoint union $U_{i,j} = U_{i+1,2j} \cup U_{i+1,2j+1}$, and $|U_{d,j}| = 0$ or 1 for every j.

For each i, j let

$$A_{i,j} = \prod_{k \in U_{i,j}} k, \qquad S_{i,j} = \prod_{\substack{p \in U_{i,j} \\ p \text{ prime}}} p^2.$$

Note that $A_{i,j} = A_{i+1,2j} A_{i+1,2j+1}$, and that $A_{i,j}$ has $O(2^{-i} N \log N)$ bits. We have $A_{d,j} = 1$ or k according to whether $U_{d,j} = \emptyset$ or $\{k\}$. We may compute all the $A_{i,j}$ using a product tree [5], working from the bottom of the tree ($i = d$) to the top ($i = 0$). The cost at each level of the tree is $2^i (2^{-i} N \log N) \log^{1+o(1)} N = N \log^{2+o(1)} N$, so the total cost to compute all the $A_{i,j}$ is $N\log^{3+o(1)} N$. Similarly, we may compute all the $S_{i,j}$ using a product tree and the precomputed table of primes, in time $N\log^{3+o(1)} N$. (In fact, because of the estimate $\sum_{p\le N}\log p = O(N)$, this product tree takes time only $N\log^{2+o(1)} N$, but we will not use this here.)

Now let

$$W_{i,j} = \prod_{0\le r<j} A_{i,r} \pmod{S_{i,j}} = \left(\left\lfloor j\frac{N}{2^i}\right\rfloor\right)! \pmod{S_{i,j}}.$$

We may compute all the $W_{i,j}$ in time $N\log^{3+o(1)} N$ by working from the top of the tree to the bottom, starting with $W_{0,0}=1$ and then using the relations

$$W_{i+1,2j} = W_{i,j} \pmod{S_{i+1,2j}}, \tag{2.1}$$

$$W_{i+1,2j+1} = W_{i,j}A_{i+1,2j} \pmod{S_{i+1,2j+1}}. \tag{2.2}$$

Finally, we may read the Wilson quotients off the bottom layer of the $W_{i,j}$ tree: for each $p\le N$, let $j=\lceil 2^d p/N\rceil - 1$. Then $U_{d,j}=\{p\}$, so $S_{d,j}=p^2$ and $W_{d,j}=(p-1)! \pmod{p^2}$. □

Now we consider Theorem 1.2. The first step (Stage 1) is to evaluate $M! \pmod S$ where $S=\prod_{M<p\le N} p^2$. Using a full product tree for $M!$ would lead to time complexity $M\log^{3+o(1)} M$, since $\log M! \sim M\log M$. In the next proposition, we reduce this to $M\log^{2+o(1)} M$ by using a space-optimised variant of the factorial algorithm of [32]. In practice Stage 1 makes a significant contribution to the total running time, so the reduction in time by a factor of $\log M$ is significant.

Proposition 2.3. *Let $S>0$ be an integer with at most B bits. Then $N! \pmod S$ may be computed in time*

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

and space

$$O(B+\sqrt{N}\log N\log\log N).$$

Proof. Let $N! = p_1^{e_1}\cdots p_r^{e_r}$ be the prime factorisation of $N!$. For each j we have

$$e_j = \lfloor N/p_j\rfloor + \lfloor N/p_j^2\rfloor + \cdots + \lfloor N/p_j^{\lfloor \log N/\log p_j\rfloor}\rfloor \le \frac{N}{p_j-1}. \tag{2.3}$$

Let $d=\lceil \log_2(N+1)\rceil$, so that $N<2^d$, and for each $1\le j\le r$ let

$$e_j = f_{0,j} + 2f_{1,j} + \cdots + 2^{d-1}f_{d-1,j}$$

be the binary representation of e_j, i.e., with $f_{i,j}=0$ or 1. Then

$$N! = A_0(A_1)^2(A_2)^4\cdots(A_{d-1})^{2^{d-1}}, \tag{2.4}$$

where

$$A_i = p_1^{f_{i,1}}\cdots p_r^{f_{i,r}}.$$

Observe that, if $p_j-1>2^{-i}N$, then $e_j<2^i$ by (2.3), so $f_{i,j}=0$. Thus, actually

$$A_i = \prod_{p_j\le 2^{-i}N+1} p_j^{f_{i,j}},$$

and we have the following estimate for the size of A_i:

$$\log A_i \le \sum_{p\le 2^{-i}N+1} \log p = O(2^{-i}N).$$

We will first show how to compute $A_i \pmod{S}$ in time

$$(2^{-i}N + \sqrt{N}) \log^{2+o(1)} N$$

and space $O(B + \sqrt{N} \log N \log\log N)$.

Partition the interval $1 < k \leq 2^{-i}N + 1$ into subintervals, say $T_1, \ldots, T_m$, where each subinterval, except possibly the last, has length

$$L = \left\lfloor \max\left(\sqrt{N}, \frac{B}{\log N \log\log N}\right) \right\rfloor.$$

For each subinterval T_r, perform the following operations.

First use Proposition 2.2 to enumerate the primes in T_r. For each subinterval, this uses space $O((L + \sqrt{N}) \log N \log\log N) = O(B + \sqrt{N} \log N \log\log N)$. The time cost for each subinterval of length L is $(L + \sqrt{N}) \log^{2+o(1)} N = L \log^{2+o(1)} N$. There are at most $2^{-i}N/L$ such subintervals, so their total cost is $2^{-i}N \log^{2+o(1)} N$. The last interval has length at most $2^{-i}N$, so contributes $(2^{-i}N + \sqrt{N}) \log^{2+o(1)} N$. The time cost over all subintervals is therefore $(2^{-i}N + \sqrt{N}) \log^{2+o(1)} N$.

Now compute $f_{i,j}$ for each $p_j \in T_r$. Using (2.3), the time complexity for each prime is $(\log N/\log p) \log^{1+o(1)} N = \log^{2+o(1)} N$, which over all subintervals is $\pi(2^{-i}N) \log^{2+o(1)} N = 2^{-i}N \log^{2+o(1)} N$.

Append the primes for which $f_{i,j} = 1$ to a separate buffer. Whenever the total length of that buffer reaches B (i.e., when it contains $B/\log N$ primes), or when we finish processing the last interval, use a product tree to multiply together the primes in the buffer (using space $O(B)$), and then clear the buffer to receive more primes. Accumulate the result of the product tree into a running product for $A_i \pmod{S}$, using a single multiplication modulo S (again space usage is $O(B)$). The total time for the product trees over all intervals is $(\log A_i) \log^{2+o(1)} B = 2^{-i}N \log^{2+o(1)} N$, since we may certainly assume that $B = O(\log N!) = O(N \log N)$. The time for the modular multiplications is $\lfloor (\log A_i)/B \rfloor B \log^{1+o(1)} B = 2^{-i}N \log^{1+o(1)} N$. We conclude that $A_i \pmod{S}$ may be computed within the promised time and space bounds.

Now let

$$C_i = A_i (A_{i+1})^2 \cdots (A_{d-1})^{2^{d-1-i}}$$

for $0 \leq i \leq d-1$. We have $C_{d-1} = A_{d-1} \pmod{S}$ and $C_i = A_i (C_{i+1})^2 \pmod{S}$ for $0 \leq i \leq d-2$. Using these relations, we compute $A_{d-1}, C_{d-1}, A_{d-2}, C_{d-2}, \ldots, A_0, C_0 \pmod{S}$. By (2.4), at the end we have obtained $C_0 = N! \pmod{S}$. To estimate the time complexity, note that

$$\begin{aligned} \log C_i &= O(2^{-i}N + 2(2^{-i-1}N) + \cdots + 2^{d-1-i}(2^{-d+1}N)) \\ &= O(2^{-i}N \log N). \end{aligned}$$

Therefore computing $C_i = A_i (C_{i+1})^2 \pmod{S}$ from $A_i \pmod{S}$ and $C_{i+1} \pmod{S}$ has time complexity $2^{-i}N \log^{2+o(1)} N$. (Here we have used the fact that if X and Y are integers with at most M bits, then $XY \pmod{S}$ can be computed from $X \pmod{S}$ and $Y \pmod{S}$ in time $M \log^{1+o(1)} M$. Indeed, if $XY < S$, then no modular reduction is performed, whereas, if $XY \geq S$, we need to perform one modular reduction whose time cost is bounded by a constant multiple of the cost of the full multiplication.) The space complexity is $O(B)$, with the previous values of A_i and C_i discarded as we proceed. Summing over i, the total time cost is $N \log^{2+o(1)} N$. $\square$

Finally, we may prove Theorem 1.2.

Proof of Theorem 1.2. We must first enumerate the primes $M < p \leq N$. Using Proposition 2.2 directly for this would use too much space, but we may instead apply it to successive subintervals of length $K = \lfloor L / \log N \log\log N \rfloor$, where $L = N - M$. The space used is $O((K + \sqrt{N}) \log N \log\log N) = O(L + \sqrt{N} \log N \log\log N) = O(L)$, plus the space needed to store the primes, namely $O((\pi(N) - \pi(M)) \log N)$. To estimate the latter, note that according to [19, Thm. 6.6] we have $\pi(N) - \pi(M) = O(L/\log L)$. Our assumption $L > \sqrt{N} \log N \log\log N$ then implies that $(\pi(N) - \pi(M)) \log N = O(L)$. Thus the space usage is indeed $O(L)$. The time over all subintervals is $L \log^{2+o(1)} N + (L/K)\sqrt{N} \log^{2+o(1)} N = L \log^{2+o(1)} N + \sqrt{N} \log^{3+o(1)} N$.

Multiply the squares of the primes together using a product tree to obtain $S = S_{0,0} = \prod_{M<p\leq N} p^2$. The number of bits in S is $B = O(L)$, so this takes space $O(L)$ and time $L \log^{2+o(1)} N$.

Use Proposition 2.3 to compute $M! \pmod{S}$ in time $M \log^{2+o(1)} M$ and space $O(L + \sqrt{N} \log N \log\log N) = O(L)$. This is Stage 1.

For Stage 2, we use a similar strategy as in the proof of Theorem 1.1, but taking additional care to economise on space usage. Let $d = \lceil \log_2 L \rceil$. For each $0 \leq i \leq d$ and $0 \leq j < 2^i$ let

$$U_{i,j} = \left\{ k \in \mathbf{Z} : M + j\frac{L}{2^i} < k \leq M + (j+1)\frac{L}{2^i} \right\}.$$

For each i this yields a partition of the interval $M < k \leq N$ into 2^i sets. As in Theorem 1.1, let

$$A_{i,j} = \prod_{k \in U_{i,j}} k, \qquad S_{i,j} = \prod_{\substack{p \in U_{i,j} \\ p \text{ prime}}} p^2.$$

The definition of $W_{i,j}$ is slightly different; we take

$$W_{i,j} = M! \prod_{0 \leq r < j} A_{i,r} \pmod{S_{i,j}} = \left(\left\lfloor M + j\frac{L}{2^i} \right\rfloor \right)! \pmod{S_{i,j}}.$$

We do not have enough space to store all of the $A_{i,j}$ and $S_{i,j}$, so we must proceed differently to the proof of Theorem 1.1. We will use a strategy similar to the proof of [35, Lemma 2.1].

We begin at the top of the tree with $W_{0,0} = M! \pmod{S_{0,0}}$, which was computed above using Proposition 2.3. As in the proof of Theorem 1.1, we use relations (2.1) and (2.2) to work our way down the tree. Every new pair of values $W_{i+1,2j}$ and $W_{i+1,2j+1}$ overwrites the previous value of $W_{i,j}$. For fixed i, the total size of the $W_{i,j}$ at level i is $O(L)$, so the space for storing the $W_{i,j}$ never exceeds $O(L)$.

For the top $\ell = \lfloor 2 \log_2 \log N \rfloor$ levels of the tree, we recompute each required $A_{i,j}$ and $S_{i,j}$ as we encounter them, discarding intermediate values (i.e., $A_{i,j}$ and $S_{i,j}$ from lower levels of the product tree) as we proceed. Also, in the evaluation of (2.2), we do not compute $A_{i+1,2j+1}$ exactly, but rather only modulo $S_{i+1,2j+1}$, by reducing as appropriate during the product tree computation. The time complexity contributed by each level of the tree is thus $L \log^{3+o(1)} N$ (this is a factor of $\log N$

more than in Theorem 1.1, due to the recomputations), but over the first ℓ levels this amounts to only $L\log^{3+o(1)} N \log\log N = L\log^{3+o(1)} N$.

When we reach level ℓ, we switch back to the strategy of Theorem 1.1. For each j at level ℓ, we compute the entire trees beneath $A_{\ell,j}$ and $S_{\ell,j}$. This requires space $O(\log(A_{\ell,j})\log N) = O(2^{-\ell}L\log^2 N) = O(L)$. The time contribution from each level is $L\log^{2+o(1)} N$, so over all levels is $L\log^{3+o(1)} N$. The Wilson quotients are extracted from the $W_{d,j}$ just as in Theorem 1.1. □

3. Factorial identities modulo p^2

Let e be an even divisor of $p-1$, and let $f = (p-1)/e$. In this section we describe a method for reducing computation of $(p-1)! \pmod{p^2}$ to that of $f! \pmod{p^2}$.

As mentioned in the introduction, identity (1.1), corresponding to the case $e = 2$, has been applied to the computation of Wilson quotients for almost a century. The cases $e = 4$ and $e = 6$ were introduced by [10].

Our method can be applied in principle to any e. The simplest case, and the only case we will describe in this paper, is when the e-th cyclotomic field over $\mathbf{Q}$ has class number 1. It is known that this occurs for precisely the following values of e ([36, Ch. 11]):

$$2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,$$
$$32,34,36,38,40,42,44,48,50,54,60,66,70,84,90,$$

and these are the values of e that we used in our implementation.

It is straightforward to modify the algorithms given in the proof of Theorem 1.2 to compute $f! \pmod{p^2}$ instead of $(p-1)! \pmod{p^2}$. For example, given a set T of primes p lying in the interval $M < p \leq N$ and satisfying $p = 1 \pmod{e}$, the modified Stage 1 involves using Proposition 2.3 to compute $\lfloor M/e\rfloor! \pmod{\prod_{p\in T} p^2}$.

To apply this to the main Wilson prime search, each prime p is assigned to the "best" possible e for that prime, i.e., the largest divisor of $p-1$ appearing in the above list. Then for each e, our strategy is to use the (suitably modified) algorithm of Theorem 1.2 to compute w_p for all p assigned to e.

It is a difficult theoretical problem to analyse the savings that accrue from this strategy. If we assume that the amount of RAM is fixed, then Stage 1 will dominate for sufficiently large N. In Stage 1 we expect a speedup by roughly a linear factor of e, since we are only computing $\lfloor M/e\rfloor!$ rather than $M!$. Therefore, in the limit of large N, we expect a savings of a factor of e for the primes assigned to e.

In practice, however, these ideal conditions are not met. Stage 2 does make a significant contribution, especially for larger values of e. The effect of e on Stage 2 is complex. As e increases, a fixed interval $M < p \leq N$ will contain fewer and fewer primes of interest. The number of such primes depends in a complicated way on the complete list of admissible e. To make best use of available RAM, for larger e we will generally choose a larger interval, so that the number of primes in the interval is roughly constant, but the relationship is not linear.

In addition, we must take into account the cost of deducing $(p-1)! \pmod{p^2}$ from $((p-1)/e)! \pmod{p^2}$. We refer to this step of the computation as Stage 3. We have not attempted to give a theoretical bound for the cost of Stage 3. In general it becomes more expensive as e increases. In our computation it accounted for only a few percent of the total running time (see Table 4).

Let us estimate the overall savings, over many primes, under the assumption that the speedup is linear in e, and ignoring the cost of Stage 3. Let S be a set of permissible values of e, for example, the set $\{2, 4, \dots, 84, 90\}$ given above. We assume that for each $e \in S$, we apply the above strategy to those primes p for which e is the largest divisor of $p-1$ that appears in S. Let $Q_S = \mathrm{LCM}(S)$. For $k \in (\mathbf{Z}/Q_S\mathbf{Z})^*$, let $b_S(k) = \max\{e \in S : k = 1 \pmod{e}\}$. Then the expected savings is

$$R_S = \frac{1}{\phi(Q_S)} \sum_{k \in (\mathbf{Z}/Q_S\mathbf{Z})^*} \frac{1}{b_S(k)}.$$

For example, if we only use identity (1.1), then $S = \{2\}$, $Q_S = 2$, and $R_S = 1/2$, so we save a factor of 2 over the naive algorithm.

The identities used in [10] correspond to choosing $S = \{2, 4, 6\}$, in which case $Q_S = 12$ and $R_S = (1/6 + 1/4 + 1/6 + 1/2)/4 = 13/48$, saving a further factor of $24/13 \approx 1.85$.

Taking S to be the full set $S = \{2, 4, \dots, 84, 90\}$, we have

$$Q_S = 6983776800 = 2^5 \cdot 3^3 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19.$$

A brute force computation finds that

$$R_S = \frac{22695187978681}{201921527808000} \approx 0.112,$$

indicating a further savings of a factor of roughly 2.41 compared to [10].

Now we explain the reduction. Fix a primitive e-th root of unity $\omega \in \mathbf{Z}_p$. Let $\Gamma_p : \mathbf{Z}_p \to \mathbf{Z}_p^*$ denote the p-adic gamma function. The next proposition, whose proof is adapted from [4, Thm. 9.3.1], gives a congruence between $(p-1)!/f!^e$ and a special value of the p-adic gamma function.

Proposition 3.1. *Let*

$$C = \frac{1}{p} \sum_{j=1}^{e-1} \big((1-\omega^j)^p - (1-\omega^j)\big) \in \mathbf{Z}_p,$$

where e and ω are defined as above. Then

$$\frac{(p-1)!}{f!^e} = -\Gamma_p(1/e)^e(1+pC) \pmod{p^2}.$$

Proof. Let $M = p^2 - (p^2-1)/e = p^2 - f(p+1)$. Then $M = 1/e \pmod{p^2}$ and $1 \le M < p^2$. By the definition and elementary properties of $\Gamma_p(x)$ (see for example [22, Ch. 14]) we have

$$\begin{aligned}\Gamma_p(1/e) &= \Gamma_p(M) \pmod{p^2}\\ &= -\prod_{\substack{1 \le j < M\\ p \nmid j}} j \pmod{p^2}.\end{aligned}$$

Splitting the product into blocks of length p we obtain

$$\Gamma_p(1/e) = -\left(\prod_{k=0}^{\lceil M/p\rceil - 1} \prod_{r=1}^{p-1} (kp + r)\right)\left(\prod_{j=M}^{\lceil M/p\rceil p - 1} j\right)^{-1} \pmod{p^2}.$$

Since $\lceil M/p \rceil = p - f + \lfloor f/p \rfloor = p - f$,

$$\Gamma_p(1/e) = -\left(\prod_{k=0}^{p-f-1}\prod_{r=1}^{p-1}(kp+r)\right)\left(\prod_{j=p^2-fp-f}^{p^2-fp-1} j\right)^{-1} \pmod{p^2}.$$

For the first term, observe that for any $k \in \mathbf{Z}$ we have

$$\frac{\prod_{r=1}^{p-1}(kp+r)}{(p-1)!} = \prod_{r=1}^{p-1}(1+kp/r) = 1 + kp\sum_{r=1}^{p-1} 1/r = 1 \pmod{p^2}.$$

Therefore,

$$\prod_{k=0}^{p-f-1}\prod_{r=1}^{p-1}(kp+r) = (p-1)!^{p-f} \pmod{p^2}.$$

For the second term, we have

$$\prod_{j=p^2-fp-f}^{p^2-fp-1} j = \prod_{j=-fp-f}^{-fp-1} j = (-1)^f \prod_{r=1}^{f}(r+fp) \pmod{p^2}.$$

To evaluate this last product, note that

$$\frac{\prod_{r=1}^{f}(r+fp)}{f!} = \prod_{r=1}^{f}(1+fp/r) = 1 + fp\sum_{r=1}^{f} 1/r \pmod{p^2}.$$

Moreover, for any $1 \le j \le e-1$,

$$\begin{aligned}\frac{(1-\omega^j)^p - (1-\omega^j)}{p} &= \frac{1}{p}\sum_{k=1}^{p-1}\binom{p}{k}(-\omega^j)^k \\ &= \sum_{k=1}^{p-1}\frac{(p-1)(p-2)\cdots(p-k+1)}{k(k-1)\cdots 1}(-\omega^j)^k \\ &= -\sum_{k=1}^{p-1}\frac{(k-1)!}{k!}\omega^{jk} = -\sum_{k=1}^{p-1}\omega^{jk}/k \pmod{p}.\end{aligned}$$

Thus,

$$C = -\sum_{j=1}^{e-1}\sum_{k=1}^{p-1}\omega^{jk}/k = -\sum_{k=1}^{p-1}\frac{1}{k}\sum_{j=1}^{e-1}\omega^{jk} \pmod{p}.$$

Since

$$\sum_{j=1}^{e-1}(\omega^k)^j = -1 + \begin{cases} e & \text{if } e \mid k, \\ 0 & \text{otherwise,}\end{cases}$$

we get

$$C = -\sum_{k=1}^{p-1}\frac{1}{k} - e\sum_{r=1}^{f}\frac{1}{er} = -\sum_{r=1}^{f} 1/r \pmod{p}.$$

Putting everything together, we have

$$\Gamma_p(1/e) = \frac{-(p-1)!^{p-f}(-1)^f}{f!(1-fpC)} \pmod{p^2}.$$

From Wilson's theorem we have $(p-1)!^p = -1 \pmod{p^2}$, and so

$$\Gamma_p(1/e)^e = \frac{(p-1)!^{-ef}}{f!^e(1-fpC)^e} = \frac{(p-1)!^{-p+1}}{f!^e(1-efpC)} = \frac{-(p-1)!}{f!^e(1+pC)} \pmod{p^2}.$$

Rearranging, we obtain the desired formula. □

Next we will use the Gross–Koblitz formula to relate $\Gamma_p(1/e)$ to a certain Gauss sum. Let $K = \mathbf{Q}(\zeta_e)$, where ζ_e is a primitive e-th root of unity. The ring of integers of K is $O_K = \mathbf{Z}[\zeta_e]$. Let $\omega_0 \in \mathbf{Z}$ be an integer congruent to $\omega \pmod p$, and let $P = (p, \zeta - \omega_0)$. Then P is a prime ideal of O_K of degree 1 lying above p, i.e., $O_K/P \cong \mathbf{F}_p$. Let $\chi : \mathbf{F}_p^* \to K^*$ be the $(-f)$-th power of the Teichmüller character; that is, $\chi(u) = u^{-f} \pmod P$ for any $u \in \mathbf{F}_p^*$. Define the Gauss sum

$$S(\chi) = \sum_{j=1}^{p-1} \chi(j)\zeta_p^j \in K(\zeta_p),$$

where ζ_p is a primitive p-th root of unity.

Proposition 3.2. *We have $S(\chi)^e \in K$. Regarding K as embedded in $\mathbf{Q}_p$ via the map that sends ζ_e to ω, we have*

$$-\Gamma_p(1/e)^e = \frac{(-S(\chi))^e}{p}.$$

Proof. The first statement follows from [22, Ch. 1, Thm. 1.3(i)]. The second statement is a consequence of the Gross–Koblitz formula, for example, [22, Ch. 15, Thm. 4.3]. In the notation of [22], take $r = 1$, $q = p$, $a = p - 1 - f$. The above formula falls out after taking e-th powers. □

The final ingredient is the Stickelberger factorisation of the ideal of K generated by $S(\chi)^e$. For $c \in (\mathbf{Z}/e\mathbf{Z})^*$, let σ_c denote the automorphism of $K/\mathbf{Q}$ that sends ζ_e to ζ_e^c.

Proposition 3.3. *The ideal generated by $S(\chi)^e$ decomposes as*

$$(S(\chi)^e) = \prod_{\substack{c=1 \\ (c,e)=1}}^{e-1} \sigma_{c^{-1}}(P)^c.$$

Proof. Raise both sides of [22, Ch. 1, Thm. 2.2] to the power of e. □

Proposition 3.4. *Suppose that P is principal, and let θ be a generator. Let*

$$\beta = \prod_{\substack{c=1 \\ (c,e)=1}}^{e-1} \sigma_{c^{-1}}(\theta)^c \in O_K.$$

Then

$$S(\chi)^e = \zeta_e^i \beta$$

for some $0 \le i < e$.

Proof. By Proposition 3.3, $S(\chi)^e$ and β differ by a unit of O_K. Moreover,

$$\sigma_{-1}(\beta) = \prod_c \sigma_{-c^{-1}}(\theta)^c = \prod_c \sigma_{c^{-1}}(\theta)^{e-c}$$

so

$$\beta\sigma_{-1}(\beta) = \prod_c \sigma_{c^{-1}}(\theta)^e = N_{K/\mathbf{Q}}(\theta)^e = N(P)^e = p^e.$$

Thus the image of β under every complex embedding $K \to \mathbf{C}$ has absolute value $p^{e/2}$. But $S(\chi)^e$ has the same property [22, p. 4]. Therefore $S(\chi)^e/\beta$ has absolute value 1 in every complex embedding, and so is a root of unity in K [36, Lemma 1.6]. Since e is even, every root of unity is a power of ζ_e, and the conclusion follows. □

Theorem 3.5. *Let $p = 1 \pmod{e}$, where e is even. Assume that $K = \mathbf{Q}(\zeta_e)$ has class number 1. Assume we are given as input:*

- *a primitive e-th root of unity in $\mathbf{F}_p^*$, represented as an integer $1 \le \omega_0 < p$,*
- *a generator θ of the ideal $P = (p, \zeta_e - \omega_0)$, represented as $\theta = g(\zeta_e)$ for some polynomial $g \in \mathbf{Z}[x]$ of degree less than $\phi(e)$, and*
- *$f! \pmod{p^2}$.*

Then we may compute $(p-1)! \pmod{p^2}$ using $O(e^2 + e\log p)$ arithmetic operations on integers with $O(\log p)$ bits.

The big-O estimates given in the above theorem are strictly speaking meaningless, since they only apply to finitely many e. We give the estimates anyway as an indication of how the running time might reasonably be expected to behave in practice.

We may compute a suitable ω_0 using a simple probabilistic algorithm as follows. Select a random $1 \le x \le p-1$. Then $\omega_0 = x^f \pmod{p}$ has order exactly e with probability $\phi(e)/e \ge 1/e$. We can compute the exact order using at most e arithmetic operations in $\mathbf{Z}/p\mathbf{Z}$. This is repeated until we find a suitable ω_0.

The computational bottleneck is actually in finding θ, which we consider after the proof of the theorem.

Proof. We will take "arithmetic operation" to mean an addition or multiplication modulo p, p^2 or p^3.

We first lift the root of unity, putting $\omega_1 = \omega_0^{p^2} \pmod{p^3}$, so that $\omega_1 = \omega \pmod{p^3}$. This requires $O(\log p)$ arithmetic operations. We next compute the powers ω_1^i for $0 \le i < e$, using $O(e)$ arithmetic operations. Computing C from Proposition 3.1 requires $O(e\log p)$ arithmetic operations.

Let γ be the image in $\mathbf{Z}_p$ of β/p, where β is as in Proposition 3.4, i.e.,

$$\gamma = \frac{1}{p} \prod_{\substack{c=1 \\ (c,e)=1}}^{e-1} g(\omega^{c^{-1}})^c.$$

With this formula, we may compute $\gamma \pmod{p^2}$ using $O(e^2)$ arithmetic operations. Combining Propositions 3.1, 3.2, 3.3 and 3.4, we have

$$\omega^{-i}(p-1)! = (-f!)^e \gamma(1 + pC) \pmod{p^2}$$

for some $0 \le i < e$, so we can compute $\omega^{-i}(p-1)! \pmod{p^2}$ using a further $O(\log e)$ operations. However, we know that $(p-1)! = -1 \pmod{p}$, so we can determine i by comparing with the tabulated powers of ω. □

Before discussing the computation of θ, we illustrate Theorem 3.5 with a numerical example. Take $p = 3333331$, $e = 18$, $f = 185185$, and the 18th root

$\omega_0 = 1819843$. The Teichmüller lift is

$$\omega = 1819843 + 1422487p + 90367p^2 \pmod{p^3},$$

and

$$C = \frac{(1-\omega)^p - (1-\omega) + \cdots + (1-\omega^{17})^p - (1-\omega^{17})}{p} = 418399 \pmod{p}.$$

Using the cyclotomic GCD algorithm discussed below, we find a generator $\theta = g(\zeta_e)$ of $P = (p, \zeta_e - \omega_0)$ given by

$$g(x) = -5x^5 - 10x^4 + 7x^3 + 3x^2 + 10x - 4.$$

Then

$$\begin{aligned}\gamma &= \frac{1}{p} g(\omega) g(\omega^{11})^5 g(\omega^{13})^7 g(\omega^5)^{11} g(\omega^7)^{13} g(\omega^{17})^{17} \\ &= 1628187 + 503367p \pmod{p^2}.\end{aligned}$$

Now assuming that we have computed

$$f! = 461190 + 275007p \pmod{p^2},$$

we find that

$$\omega^{-i}(p-1)! = (-f!)^e \gamma(1+pC) = 1780730 + 2171988p \pmod{p^2}.$$

Comparing with the powers of ω, we find that $\omega_0^3 = -1780730 \pmod{p}$, so $i = 3$ and

$$(p-1)! = 3333330 + 27003p \pmod{p^2}.$$

We conclude that $w_p = 27004$.

Now we consider the problem of computing θ. The standard approach to the ideal generator problem is based on lattice reduction (see, for example, [9]), and indeed there exist highly optimised implementations in software packages such as Pari/GP [34].

After some experimentation we settled on a different approach, which we found to be considerably faster than Pari in practice. Our algorithm is closer in spirit to the elementary Euclidean GCD algorithm. We emphasise that this is not a general-purpose algorithm for finding ideal generators: it assumes that K has class number 1, and also uses the fact that we know in advance that the generator is an irreducible element whose norm is not too small. In addition, we are unable to prove that the "algorithm" terminates. In practice we find that it does terminate quite quickly. Pseudocode is shown in Algorithm 1 below. The algorithm is applied to the inputs $X = p$ and $Y = \zeta_e - \omega_0$, and their GCD is precisely the desired θ.

Several aspects of the algorithm deserve further discussion.

All elements of O_K appearing in the algorithm are represented exactly, as $\mathbf{Z}$-linear combinations of the basis elements $\{1, \zeta_e, \ldots, \zeta_e^{d-1}\}$, where $d = \phi(e) = [K : \mathbf{Q}]$, i.e., as polynomials in ζ_e. We first attempt to run the algorithm with all coefficients represented by signed 64-bit integers, and ignoring all overflows. If the algorithm terminates, we can check the output by verifying that the proposed θ divides both p and $\zeta_e - \omega_0$. This usually succeeds. If it is incorrect, or if the algorithm runs for too long without terminating, we restart it. If this fails several times, we switch to an implementation that uses an arbitrary precision representation for the coefficients. This eliminates the possibility of overflow, so that if the

Algorithm 1: Heuristic cyclotomic GCD

```
Input: $X, Y \in O_K$
       $S$ = precomputed list of elements of $O_K$ of small norm
Output: A generator of $(X, Y)$
1  while $X \neq 0$ and $Y \neq 0$ do
2      if $N(X) < N(Y)$ then swap $X$ and $Y$
3      $Q \leftarrow$ an element of $O_K$ near $X/Y$
4      $Z \leftarrow X - QY$
5      if $N(Z) < N(Y)$ then $X \leftarrow Z$
6      else
7          $U \leftarrow$ randomly selected element of $S$
8          if $U \mid Y$ then $Y \leftarrow Y/U$ else $X \leftarrow XU$
9      end
10 end
11 if $X = 0$ then return $Y$ else return $X$
```

algorithm terminates, the output is guaranteed to be correct. Again, if it runs for too long, we restart it. In practice this always eventually succeeds.

Exact multiplication of elements of O_K (lines 4 and 8) is achieved by naive polynomial multiplication followed by reduction modulo the cyclotomic polynomial $\phi_e(x)$. Exact division (line 8) is achieved by the formula $X/Y = X\prod_{\sigma\neq 1}\sigma(Y)/N(Y)$, where the denominator $N(Y) = \prod_\sigma \sigma(Y)$ is a rational integer. Here σ denotes an automorphism of K, which is evaluated by cyclic permutation of coordinates followed by reduction modulo $\phi_e(x)$.

Let $\tau_1, \overline{\tau_1}, \ldots, \tau_{d/2}, \overline{\tau_{d/2}}$ be the complex embeddings $K \hookrightarrow \mathbf{C}$, and let $\tau = (\tau_1, \ldots, \tau_{d/2}) : K \to \mathbf{C}^{d/2}$ be the corresponding vector of embeddings. For each variable V in Algorithm 1, we also maintain a second representation, namely a double-precision floating point approximation to $\tau(V)$.

In lines 2 and 5, the norms are approximated by multiplying together the coordinates of $\tau(V)$, rather than by computing an exact norm in $\mathbf{Z}$.

In line 3, we first approximate $\tau(X/Y)$ by computing $\tau_i(X)/\tau_i(Y)$ (as a floating-point complex number) for each i. Applying the inverse of τ yields an approximation to X/Y in $K \otimes_{\mathbf{Q}} \mathbf{R}$. We select Q by simply rounding each coordinate to the nearest integer. In the ideal situation we will have $N(X/Y - Q) < 1$. If this holds, then line 5 will succeed in updating X, and then we have made some progress in reducing the norm. However, there is no guarantee that $N(X/Y - Q) < 1$ will occur. One possibility is that there exists some $Q' \in O_K$ such that $N(X/Y - Q') < 1$, but that our simple-minded method for selecting Q did not locate it. To mitigate against this, we make a few attempts to adjust the coordinates of Q to locate a suitable Q'. This may still fail, and moreover it may turn out that there does not exist *any* Q' with the right property. This may occur if K is not Euclidean with respect to the norm; for example, it is known that $\mathbf{Q}(\zeta_{32})$ has this property [25]. In this case, we will fall through to lines 7–8.

The goal of lines 7–8 is to make some random perturbation, in the hope that we will be lucky in finding a good Q on the next iteration. In our implementation, we take S to be the set of elements of O_K of norm q, where q is the smallest prime $q = 1 \pmod e$ (i.e., take all the conjugates of a generator of any prime ideal dividing qO_K). If we are lucky enough that U divides Y, then we know U cannot divide X, since we have assumed that the GCD has norm p, which is much larger than q. Thus dividing Y by U does not change the GCD. Otherwise, we simply multiply X by U and continue. This cannot change the GCD for the same reason.

The rationale for this perturbation strategy is as follows. If X/Y is sufficiently close to an integer, then our method for selecting Q should find it. Otherwise, UX/Y is likely to be "randomly distributed" modulo the integer lattice, and there is a reasonable chance that it will be close to an integer. We have not attempted to formulate this argument precisely or prove anything about it.

Finally, we discuss the issue of units. Whenever we compute a new element of O_K, say X, we examine the size of its coefficients, and compare this to $N(X)$. If the coefficients are too large, we apply a balancing procedure, replacing X by $u^{-1}X$ for a suitable unit $u \in O_K^*$. This of course does not alter the GCD. An extreme example of an "unbalanced" element is a high power of a nontrivial unit $u \in O_K^*$, which has large coefficients but norm 1. Without this balancing step, we soon encounter coefficient explosion (and overflow).

The condition we used to test for unbalancedness in our implementation is as follows: if $X = c_0 + c_1\zeta_e + \cdots + c_{d-1}\zeta_e^{d-1}$, we declare that X is unbalanced if $\frac{1}{d}\sum_{i=0}^{d-1} |c_i| > 10|N(X)|^{1/d}$. There is no particular theoretical justification for this particular measure of size, nor of the choice of constant 10. We used it because it is fast to evaluate and seems to give good results in practice.

To balance an element X we proceed as follows. (This strategy is inspired by the definition of "unbalanced" in [37].) Consider the logarithmic embedding $L : O_K \setminus \{0\} \to \mathbf{R}^{d/2}$ defined by $a \mapsto (\log|\tau_i(a)|)_i$. By Dirichlet's unit theorem, the image of the unit group O_K^* under this map is a lattice of full rank in the hyperplane $t_0 + \cdots + t_{d/2-1} = 0$. The vector $(\log|\tau_i(X)| - \frac{1}{d}\log|N(X)|)_i$ lies in this hyperplane. Armed with a precomputed list of generators of O_K^* (obtained for example via Pari), we may therefore use simple linear algebra over $\mathbf{R}$ to select a unit u so that $\log|\tau_i(u)|$ is close to $\log|\tau_i(X)| - \frac{1}{d}\log|N(X)|$ for all i. Then we replace X by $u^{-1}X$ and continue. The rationale is that our choice of u ensures that $|\tau_i(u^{-1}X)|$ is close to $|N(X)|^{1/d}$ for all i, so that the coefficients of $u^{-1}X$ will be reasonably small (although they might not actually satisfy the test for balancedness mentioned in the previous paragraph).

4. Implementation and hardware

Our implementation is written in C, using OpenMP for parallelisation at the level of the individual compute node. We used the GMP library [17] for multiple-precision integer arithmetic, with the following important exception.

For very large integer multiplications — for operands exceeding around 10^7 bits, depending on the hardware — we switch to our own implementation based on number-theoretic transforms (NTTs). This proceeds by splitting the input into small chunks of perhaps several words each, converting the problem to that of multiplying polynomials in $\mathbf{Z}[x]$. This is then achieved by reducing modulo several suitable 62-bit primes q, multiplying the polynomials using FFTs over $\mathbf{Z}/q\mathbf{Z}$, and reconstructing the product in $\mathbf{Z}[x]$ via the Chinese Remainder Theorem. The FFT arithmetic is optimised using techniques described in [18]. To ensure the running time behaves smoothly as a function of the input size, we allow the number of primes to vary between 3 and 6, and we select a transform length of the form $2^k 3^\ell$ where $0 \le \ell \le 6$; that is, we use mainly radix-2 transforms, but allow a few layers of radix-3 transforms. We use a strategy similar to Bailey's trick [1] to improve memory locality.

The main reason that we did not use GMP's large integer multiplication code is that GMP does not take advantage of multiple cores in a shared memory environment. In contrast, our implementation is parallelised using OpenMP. This is crucial, because in Stage 1, the average complexity per prime is inversely proportional to the amout of RAM available. To make effective use of n cores, it is not good enough to process n intervals separately using one core each, since each core will have only $1/n$ of the available RAM, and will run in effect at $1/n$ of the speed. We must actually parallelise within the integer arithmetic, to get all cores working cooperatively on a single interval.

Furthermore, our integer multiplication code is optimised heavily in favour of conserving memory. Its performance varies across platforms, but in all cases is competitive with GMP. For example, on a node of Katana (see below), multiplying two 1-gigabyte integers took 178s using GMP, with peak memory usage 9.1GB. Our code performs the same multiplication in 121s using only 5.3GB; running on 8 cores it takes 20s (a 6-fold speedup), using the same memory.

A natural extension of this idea, which we did not pursue, is to increase the effective RAM available by making use of the fast networks on modern HPC systems to treat several nodes as a single computational unit. Whether this yields any speedup in searching for Wilson primes is an interesting question for future research.

We ran our implementation over a period of about four months on several clusters at New York University ("Cardiac", "Bowery", and "Union Square"), the University of New South Wales ("Katana" and "Tensor"), and the National Computational Infrastructure facility at the Australian National University ("Vayu"). Table 3 summarises the characteristics of the nodes on each cluster, and the total CPU time expended on each cluster. Table 4 gives a breakdown of the total CPU time into Stage 1, Stage 2 and Stage 3.

In the previous section it was pointed out that Stage 1 should dominate the computation for sufficiently large p. The data in Table 4 shows that we have not yet reached this region. A more detailed accounting shows this behaviour beginning to occur in some parts of the computation; for example, for $e = 2$, on the machines with 32GB RAM, we found that Stage 1 starts to dominate for p around 5×10^{12}. The threshold increases with e and with the amount of RAM per node.

We used a client-server strategy to distribute work among the clusters. A master script ran on a server at NYU. When a compute node is ready to begin work, it sends a request via HTTP to the server. The server is responsible for choosing a value of e (as in Section 3) and a range of primes $M < p < N$ to assign to that node. This basic outline is complicated by the fact that the time needed to complete a single block was generally much longer than the running time permitted for a single job by each cluster's job scheduler. It was therefore necessary to serialise intermediate computations to disk at appropriate intervals, and reload them by another job later on. Load balancing was also complicated by varying cluster availability over the duration of the project.

Any computation of this size is bound to run into hardware failures and other problems. We took several measures to validate our results.

TABLE 3. Cluster data

Cluster	Architecture	RAM (GB)	Core-hours
Cardiac	AMD Barcelona 16 cores, 2.3GHz	32	465 000
Bowery	Intel Nehalem 12 cores, 2.67–3.07GHz	48/96/256	263 000
Union Square	Intel Xeon 8 cores, 2.33GHz	16/32	154 000
Tensor	Intel Xeon 8 cores, 3.0GHz	16/24	145 000
Katana	Intel Xeon 12 cores, 2.8–3.06GHz	24/96/144	123 000
Vayu	Intel Nehalem 8 cores, 2.93GHz	24	13 000

TABLE 4. Breakdown of CPU time

	Core-hours
Stage 1	464 000
Stage 2	655 000
Stage 3	44 000

First, for each p we check that our proposed valued for $(p-1)! \pmod{p^2}$ satisfies $(p-1)! = -1 \pmod{p}$. Second, in the notation of the proof of Theorem 3.5, we check that $(-f!)^e\gamma$ is an eth root of unity modulo p. This simultaneously provides a strong verification of the cyclotomic GCD computation and of the computation of $f!$, at least modulo p.

Finally, we wrote a completely independent program to compute w_p using the $p^{1/2+o(1)}$ algorithm of [6], together with identity (1.1) (but none of the results of Section 3). The underlying polynomial arithmetic is handled by the NTL library [33]. We ran this implementation on the 27 039 026 saved residues and found complete agreement. This computation was run on Katana and Tensor, together with a Condor cluster, utilising idle time on machines in the School of Mathematics and Statistics at UNSW; it took 440 000 CPU hours altogether.

ACKNOWLEDGMENTS

We thank the HPC facilities and staff at NYU, UNSW and NCI for providing computational resources and support. Thanks to Richard Brent, Claus Diem, Felix Fröhlich and Mark Rodenkirch for helpful discussions, and the referees for their comments which improved and simplified the presentation. Several members of mersenneforum.org used their personal computers to search up to 4×10^{11} with an early implementation written by the second author.

References

[1] David H. Bailey, *FFTs in external or hierarchical memory*, Journal of Supercomputing **4** (1990), 23–35.

[2] N. G. W. H. Beeger, *Quelques remarques sur les congruences* $r^{p-1} \equiv 1 \pmod{p^2}$ *et* $(p-1)! \equiv -1 \pmod{p^2}$, Messenger of Mathematics **43** (1913), 72–84.

[3] ______, *On the congruence* $(p-1)! \equiv -1 \pmod{p^2}$, Messenger of Mathematics **49** (1920), 177–178.

[4] Bruce C. Berndt, Ronald J. Evans, and Kenneth S. Williams, *Gauss and Jacobi sums*, Canadian Mathematical Society Series of Monographs and Advanced Texts, John Wiley & Sons Inc., New York, 1998. A Wiley-Interscience Publication. MR1625181 (99d:11092)

[5] Daniel J. Bernstein, *Fast multiplication and its applications*, Algorithmic number theory: lattices, number fields, curves and cryptography, Math. Sci. Res. Inst. Publ., vol. 44, Cambridge Univ. Press, Cambridge, 2008, pp. 325–384. MR2467550 (2010a:68186)

[6] Alin Bostan, Pierrick Gaudry, and Éric Schost, *Linear recurrences with polynomial coefficients and application to integer factorization and Cartier-Manin operator*, SIAM J. Comput. **36** (2007), no. 6, 1777–1806, DOI 10.1137/S0097539704443793. MR2299425 (2008a:11156)

[7] Richard P. Brent and H. T. Kung, *A regular layout for parallel adders*, IEEE Trans. Comput. **31** (1982), no. 3, 260–264, DOI 10.1109/TC.1982.1675982. MR648375 (83b:68002)

[8] S. Chowla, B. Dwork, and Ronald Evans, *On the mod* p^2 *determination of* $\binom{(p-1)/2}{(p-1)/4}$, J. Number Theory **24** (1986), no. 2, 188–196, DOI 10.1016/0022-314X(86)90102-2. MR863654 (88a:11130)

[9] Henri Cohen, *A course in computational algebraic number theory*, Graduate Texts in Mathematics, vol. 138, Springer-Verlag, Berlin, 1993. MR1228206 (94i:11105)

[10] Richard Crandall, Karl Dilcher, and Carl Pomerance, *A search for Wieferich and Wilson primes*, Math. Comp. **66** (1997), no. 217, 433–449, DOI 10.1090/S0025-5718-97-00791-6. MR1372002 (97c:11004)

[11] Claus Diem, *On the complexity of some computational problems in the Turing model*, Preprint, `http://www.math.uni-leipzig.de/~diem/preprints/turing.pdf`, 2011.

[12] Harvey Dubner, *Searching for Wilson primes*, J. Recreational Math. **21** (1989), no. 1, 19–20.

[13] Carl-Erik Fröberg, *Diagonalization of Hermitian matrices*, Math. Tables Aids Comput. **12** (1958), 219–220. MR0100351 (20 #6784)

[14] Carl-Erik Fröberg, *Investigation of the Wilson remainders in the interval* $3 \leq p < 50{,}000$, Ark. Mat. **4** (1963), 479–499 (1963). MR0174510 (30 #4711)

[15] Martin Fürer, *Faster integer multiplication*, SIAM J. Comput. **39** (2009), no. 3, 979–1005, DOI 10.1137/070711761. MR2538847 (2011b:68296)

[16] Karl Goldberg, *A table of Wilson quotients and the third Wilson prime*, J. London Math. Soc. **28** (1953), 252–256. MR0055358 (14,1062d)

[17] Törbjorn Granlund, *The gnu Multiple Precision Arithmetic Library (Version 5.0.5)*, `http://gmplib.org/`.

[18] David Harvey, *Faster arithmetic for number-theoretic transforms*, preprint `http://arxiv.org/abs/1205.2926`, 2012.

[19] Henryk Iwaniec and Emmanuel Kowalski, *Analytic number theory*, American Mathematical Society Colloquium Publications, vol. 53, American Mathematical Society, Providence, RI, 2004. MR2061214 (2005h:11005)

[20] K. E. Kloss, *Some number-theoretic calculations*, J. Res. Nat. Bur. Standards Sect. B **69B** (1965), 335–336. MR0190057 (32 #7473)

[21] Esayas George Kundert, *A von Staudt-Clausen theorem for certain Bernoullianlike numbers and regular primes of the first and second kind*, Fibonacci Quart. **28** (1990), no. 1, 16–21. MR1035125 (91e:11022)

[22] Serge Lang, *Cyclotomic fields I and II*, 2nd ed., Graduate Texts in Mathematics, vol. 121, Springer-Verlag, New York, 1990. With an appendix by Karl Rubin. MR1029028 (91c:11001)

[23] Emma Lehmer, *Questions, discussions, and notes: A note on Wilson's quotient*, Amer. Math. Monthly **44** (1937), no. 4, 237–238, DOI 10.2307/2300697. MR1523917

[24] Emma Lehmer, *Questions, discussions, and notes: On the congruence* $(p-1)! \equiv -1 (\operatorname{mod} p^2)$, Amer. Math. Monthly **44** (1937), no. 7, 462. MR1524028

[25] Hendrik W. Lenstra Jr., *Euclidean number fields. I*, Math. Intelligencer **2** (1979/80), no. 1, 6–15, DOI 10.1007/BF03024378. MR558668 (81b:12002)

[26] G. B. Mathews, *Theory of numbers*, 2nd ed, Chelsea Publishing Co., New York, 1961. MR0126402 (23 #A3698)

[27] Christos H. Papadimitriou, *Computational complexity*, Addison-Wesley Publishing Company, Reading, MA, 1994. MR1251285 (95f:68082)

[28] Erna H. Pearson, *On the congruences* $(p-1)! \equiv -1$ *and* $2^{p-1} \equiv 1 \pmod{p^2}$, Math. Comp. **17** (1964), 194–195. MR0159780 (28 #2996)

[29] Paulo Ribenboim, *The book of prime number records*, Springer-Verlag, New York, 1988. MR931080 (89e:11052)

[30] Paulo Ribenboim and Wilfrid Keller, *Die welt der primzahlen: Geheimnisse und rekorde*, Springer-Verlag, New York, 1996.

[31] A. Schönhage and V. Strassen, *Schnelle Multiplikation grosser Zahlen* (German, with English summary), Computing (Arch. Elektron. Rechnen) **7** (1971), 281–292. MR0292344 (45 #1431)

[32] Arnold Schönhage, Andreas F. W. Grotefeld, and Ekkehart Vetter, *Fast algorithms*, Bibliographisches Institut, Mannheim, 1994. A multitape Turing machine implementation. MR1290996 (96c:68043)

[33] Victor Shoup, *NTL: a library for doing number theory (Version 5.5.2)*, `http://www.shoup.net/ntl/`.

[34] The PARI Group, Bordeaux, *PARI/GP, version* 2.3.5, 2010, available from `http://pari.math.u-bordeaux.fr/`.

[35] Joachim von zur Gathen and Victor Shoup, *Computing Frobenius maps and factoring polynomials*, Comput. Complexity **2** (1992), no. 3, 187–224, DOI 10.1007/BF01272074. MR1220071 (94d:12011)

[36] Lawrence C. Washington, *Introduction to cyclotomic fields*, 2nd ed., Graduate Texts in Mathematics, vol. 83, Springer-Verlag, New York, 1997. MR1421575 (97h:11130)

[37] Douglas Wikström, *On the l-ary GCD-algorithm in rings of integers*, Automata, languages and programming, Lecture Notes in Comput. Sci., vol. 3580, Springer, Berlin, 2005, pp. 1189–1201, DOI 10.1007/11523468_96. MR2184711 (2006j:11169)

Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, New York 10012-1185

E-mail address: `edgarcosta@nyu.edu`

Eötvös Loránd University, H-1117 Budapest, Pázmány Péter sétány 1/C, Hungary

E-mail address: `robert.gerbicz@gmail.com`

School of Mathematics and Statistics, University of New South Wales, Sydney NSW 2052, Australia

E-mail address: `d.harvey@unsw.edu.au`