STRONG PSEUDOPRIMES TO TWELVE PRIME BASES

JONATHAN SORENSON AND JONATHAN WEBSTER

Abstract. Let $\psi_m$ be the smallest strong pseudoprime to the first $m$ prime bases. This value is known for $1 \leq m \leq 11$. We extend this by finding $\psi_{12}$ and $\psi_{13}$. We also present an algorithm to find all integers $n \leq B$ that are strong pseudoprimes to the first $m$ prime bases; with reasonable heuristic assumptions we can show that it takes at most $B^{2/3+o(1)}$ time.

1. Introduction

Fermat’s Little Theorem states that if $n$ is prime and $\gcd(a,n) = 1$, then $a^{n-1} \equiv 1 \pmod{n}$. A composite number for which this congruence holds is called a pseudoprime to the base $a$. We can restate Fermat’s Little Theorem by algebraically factoring (repeatedly) the difference of squares that arises in $a^{n-1} - 1$. In which case, if $n$ is an odd prime, writing $n-1 = 2^s d$ with $d$ odd, and $\gcd(a,n) = 1$, then either $a^d \equiv 1 \pmod{n}$ or $a^{2^k d} \equiv -1 \pmod{n}$ for some integer $k$ with $0 \leq k < s$. If a composite $n$ satisfies the above, we call $n$ a strong pseudoprime to the base $a$. Unlike Carmichael numbers, which are pseudoprimes to all bases, there do not exist composite numbers that are strong pseudoprimes to all bases. This fact forms the basis of the Miller-Rabin probabilistic primality test [16]. Define $\psi_m$ to be the smallest integer that is a strong pseudoprime to the first $m$ prime bases. (See A014233 at oeis.org.)

The problem of finding strong pseudoprimes has a long history. Pomerance, Selfridge, and Wagstaff [15] computed $\psi_m$ for $m = 2, 3,$ and 4. Jaeschke [10] computed $\psi_m$ for $m = 5, 6, 7,$ and 8. By looking at a narrow class of numbers, Zhang [21] gave upper bounds on $\psi_m$ for $9 \leq m \leq 19$. Recently, Jian and Deng [11] verified some of Zhang’s conjectures and computed $\psi_m$ for $m = 9, 10,$ and 11. We continue in this effort with the following.

Theorem 1.1.

\[ \psi_{12} = 3186 65857 83403 11511 67461. \]
\[ \psi_{13} = 33170 44064 67988 73859 61981. \]
We also verified Jian and Deng’s work in finding $\psi_9 = \psi_{10} = \psi_{11}$. Note that the ERH implies $\psi_m \geq \exp(\sqrt{m/2})$ [2], and Alford, Granville, and Pomerance showed that $\psi_m$ exists for all positive $m$ [1].

The proof of our theorem relies primarily on running an improved algorithm for finding strong pseudoprimes:

**Theorem 1.2.** Given a bound $B > 0$ and an integer $m > 0$ that grows with $B$, there is an algorithm to find all integers $\leq B$ that are strong pseudoprimes to the first $m$ prime bases using at most $B^{2/3+o(1)}$ arithmetic operations. The running time bound, but not correctness, assumes Conjectures 3.6 and 3.8 are true, and that $m$ is proportional to $\log B / \log \log B$.

This result is an improvement over a heuristic $O(B^{9/11})$ from [3]. Our algorithm uses the work of Jian and Deng as a starting point, and our specific improvements are these:

- We applied the results in Bleichenbacher’s thesis [3] to speed the computation of GCDs. See §2.2.
- We applied the space-saving sieve [18,19] which enabled us to sieve using full prime signature information. See §2.3.
- We used a simple table of linked-lists indexed by a hash function based on prime signatures. We used this to store small primes by signature for very fast lookup. See §2.4.
- When it made sense to do so, we parallelized our code and made use of Big Dawg, Butler University’s cluster supercomputer.

Our changes imply that adding more prime bases for a pseudoprime search (that is, making $m$ larger) speeds up the algorithm.

The rest of this paper is organized as follows. Section 2 contains a description of the algorithm and its theoretical underpinnings. Section 3 contains the proof of the running time. Section 4 contains implementation details.

2. Algorithmic theory and overview

To find all $n \leq B$ that are strong pseudoprimes to the first $m$ prime bases, we use some well-known conditions to limit how many such integers $n$ we need to check. This is outlined in the first subsection below.

We construct all possibilities by generating $n$ in factored form, $n = p_1 p_2 \ldots p_{t-1} p_t$, where $t$ is the number of prime divisors of $n$, and $p_i < p_{i+1}$. It turns out that $t$ cannot be too big; we only had to test up to $t = 6$. So we wrote separate programs for each possible $t$, and the case $t = 2$ dominates the running time.

Let $k = p_1 p_2 \ldots p_{t-1}$. Our algorithms generate all candidate $k$ values and then for each $k$ search for possible primes $p_t$ to match with $k$ to form a strong pseudoprime candidate $n = kp_t$.

For the bound $B$, we choose a cutoff value $X$. For $k \leq X$, we use a GCD computation to search for possible primes $p_t$, if they exist. For $k > X$, we use sieving to find possible primes $p_t$. Again, we use separate programs for GCDs and for sieving.

Once each $n$ is formed, we perform a strong pseudoprime test for the first $m$ prime bases to see if we have, in fact, found what we are looking for.
2.1. Conditions on (strong) pseudoprimes. For the purposes of this paper, it suffices to check square-free odd integers. A computation by Dorais and Klyve [6] shows that a pseudoprime to the bases 2 and 3 must be square-free if it is less than $4.4 \cdot 10^{31}$. Since the values Zhang conjectured as candidates for $\psi_{12}$ and $\psi_{13}$ are less than $4.4 \cdot 10^{31}$, we restrict our search to square-free numbers.

In general, to rule out integers divisible by squares, it suffices to find all Wieferich primes $\leq B^{1/2}$, and for each such prime, perform appropriate tests. This can easily be done in well under $O(B^{2/3})$ time. See §3.6.1 in [3].

To each prime $p$ we associate a vector called its signature. Let $\nu = (a_1, a_2, \ldots, a_m)$ for $a_i$ distinct positive integers. Then the signature of $p$ is

$$\sigma_p^\nu = (v(\operatorname{ord}_p(a_1)), v(\operatorname{ord}_p(a_2)), \ldots, v(\operatorname{ord}_p(a_m)))$$

where $v$ denotes valuation with respect to 2 and $\operatorname{ord}_p(a)$ is the multiplicative order of $a$ modulo $p$.

Example 2.1. Let $p$ be the prime 151121. Then $\sigma_p^\nu = (3, 4, 0, 4, 2, 1, 2, 4)$ if $\nu = (2, 3, 5, 7, 11, 13, 17, 19)$.

Theorem 2.2 (Proposition 1 in [10]). Let $n = p_1 \cdots p_t$ with $t$ distinct primes, $\nu = (a_1, a_2, \ldots, a_m)$ with different integers such that $p_i \nmid a_j$ for all $1 \leq i \leq t$ and $1 \leq j \leq m$. Then $n$ is a strong pseudoprime to each base in $\nu$ if and only if $n$ is a pseudoprime to each base in $\nu$ and $\sigma_{p_i}^\nu = \cdots = \sigma_{p_t}^\nu$.

If $t > 2$, then this theorem greatly limits the number of candidate $k$ values we need to check. In the case $t = 2$ the initial sub-product $k$ is just a single prime, so this theorem does not, at first, appear to help. However, it does play a role in sieving for $p_t$. For the rest of this paper, we use $\nu$ as the vector containing the first $m = 11$ or 12 prime numbers.

2.2. Greatest common divisors. As above, let $k = p_1 \cdots p_t$. We explain in this subsection how to find all possible choices for $p_t$ to form $n = kp_t$ as a strong pseudoprime to the first $m$ prime bases using a simple GCD computation.

Let $b$ be one of the $a_i$ from $\nu$. The pseudprimality condition $b^{kp_t-1} \equiv 1 \pmod{kp_t}$ implies that

$$b^{kp_t-1} \equiv b^{k(p_t-1)+k-1} \equiv b^{k-1} \equiv 1 \pmod{p_t},$$

i.e., that $p_t$ divides $b^{k-1} - 1$. Considering multiple bases, we know that $p_t$ must divide

$$\gcd(a_1^{k-1} - 1, a_2^{k-1} - 1, \ldots, a_m^{k-1} - 1).$$

However, since we are concerned with strong pseudoprimes, we can consider only the relevant algebraic factor of $b^{k-1} - 1$. Following Bleichenbacher’s notation [3], let $k - 1 = u2^c$ for $u$ odd and

$$h(b, k) = \begin{cases} 
    b^u - 1 & \text{if } v(\operatorname{ord}_k(b)) = 0, \\
    b^{u2^{c-1}} + 1 & \text{if } v(\operatorname{ord}_k(b)) = c > 0.
\end{cases}$$

Theorem 2.3 (Theorem 3.23 of [3]). A necessary condition for an integer $n$ of the form $n = kp_t$, for a given $k$ with $p_t$ prime, to be a strong pseudprime to the bases $a_1, \ldots, a_m$ is that $p_t$ divide $\gcd(h(a_1, k), \ldots, h(a_m, k))$.\end{quote}
This theorem allows us to construct the product of all possible $p_t$ values for a given $k$ by computing a GCD. To carry this out, we figure out which of the $h(a_i, k)$ values would be the smallest, compute that, and then compute the next smallest $h$ value modulo the smallest, perform a GCD, and repeat until we get an answer $\leq k$, or use all the $a_i$ base values.

**Example 2.4.** Let $k = 151121$, which is prime. We have

- $v(\text{ord}_k(2)) = 3 \implies 2^{37780} + 1 \approx 8.189 \times 10^{11372}$,
- $v(\text{ord}_k(3)) = 4 \implies 3^{75560} + 1 \approx 1.914 \times 10^{36051}$,
- $v(\text{ord}_k(5)) = 0 \implies 5^{9445} - 1 \approx 5.911 \times 10^{6601}$.

In this case, we would begin the computations with $5^{9445} - 1$ since it is the smallest. We then compute $\gcd(2^{37780} + 1 \mod (5^{9445} - 1), 5^{9445} - 1) = 151121$. This tells us that any $n$ of the form $151121 \cdot p_t$ is not a strong pseudoprime to the vector $\nu = (2, 3, 5)$.

These computations start off easy as $k$ is small and progressively get harder (in an essentially linear fashion) as $k$ grows. As a practical matter, these computations only need to be done once. As an example, [11] found

$$84\ 98355\ 74122\ 37221 = 206135341 \cdot 412270681$$

to be a strong pseudoprime to the first eight prime bases. When considering $p_1 = 206135341$, they sieved (which we explain in the next section) for $p_2 = 412270681$. It is reasonable to ask whether there is a larger prime that can be paired with $p_1$ that would form a strong pseudoprime to eight bases. In our computation we check by computing a GCD to answer “no.”

**Algorithm 1:** Using GCDs to rule out $k$ values

**Input:** Integers $t$ and $k$, where $t$ is the number of prime divisors and $k$ is the product of $t - 1$ primes with matching signature, and the base vector $\nu$ containing the $m$ smallest primes.

1. Let $b \in \nu$ give the smallest estimated value for $h(b, k)$ as defined above.
2. If $b = a_1$, let $i = 2$ else let $i = 1$.
3. Compute $h(b, k)$.
4. Compute $x = h(a_i, k) \mod h(b, k)$ using modular exponentiation.
5. Set $x = \gcd(h(b, k), x)$.
6. **while** $x \geq k$ and $i < m$ **do**
7. \hspace{1em} Set $i = i + 1$; if $a_i = b$ set $i = i + 1$ again.
8. \hspace{1em} Compute $y = h(a_i, k) \mod x$.
9. \hspace{1em} Set $x = \gcd(x, y)$.
10. **end while**
11. **if** $x < k$ **then**
12. \hspace{1em} We can rule out $k$.
13. **else**
14. \hspace{1em} We factor $x$ and check each prime $p_t \mid x$ with $p_t > k$ to see if $kp_t$ is a strong pseudoprime.
15. **end if**

Computing $h(b, k)$ is the bottleneck of this algorithm. When using $m = 11$ or higher, we never found anything to factor in the last step.
2.3. **Sieving.** As above, we let \( k = p_1 \cdots p_t - 1 \) and try to construct \( n = kp_t \) by finding \( p_t \). In order to search up to a bound \( B \), we need to search primes \( p_t \) in the interval \([p_{t-1}, B/k]\). We combine two types of sieving to search this interval quickly. The first we call \( \lambda \)-sieving, and the second we call signature sieving. For each prime \( p \) define

\[
\lambda_p = \text{lcm}\{\text{ord}_p(a) : a \in \nu\}.
\]

By the r-rank Artin Conjecture [14], we expect with very high probability that \( \lambda_p = p - 1 \). Let \( \lambda = \text{lcm}\{\lambda_{p_1}, \ldots, \lambda_{p_{t-1}}\} \). Since \( a^{kp_t - 1} \equiv 1 \pmod{n} \) for any \( a \in \nu \) this means \( kp_t - 1 \) is a multiple of \( \lambda \). So, \( p_t \equiv k^{-1} \pmod{\lambda} \). This tells us that \( p_t \) lies in a specific residue class modulo \( \lambda \), and we call searching for \( p_t \) using this value \( \lambda \)-sieving.

We can further narrow down the residue classes in which \( p_t \) may lie with respect to primes in \( \nu \). The following two propositions appeared originally in [10].

**Proposition 2.5.** For primes \( p \) and \( q \), if

\[
v(p - 1) = v(q - 1) \quad \text{and} \quad v(\text{ord}_p(a)) = v(\text{ord}_q(a)),
\]

then \( \left( \frac{a}{p} \right) = \left( \frac{a}{q} \right) \).

**Proposition 2.6.** For primes \( p \) and \( q \), if

\[
v(p - 1) < v(q - 1) \quad \text{and} \quad v(\text{ord}_p(a)) = v(\text{ord}_q(a)),
\]

then \( \left( \frac{a}{q} \right) = 1 \).

One may consider higher reciprocity laws, and Jaeschke did so [10]. We found quadratic reciprocity sufficient.

The authors of [11] consider searching for \( p_t \) modulo \( \text{lcm}\{\lambda, 9240\} \). They considered 30 residue classes modulo 9240 = \( 8 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \) that arose from the propositions above. With the inclusion of each additional prime from \( \nu \), we effectively rule out half of the cases to consider, thereby doubling the speed of the sieving process.

Ideally, we would like to include every prime in \( \nu \) for the best performance, but with normal sieving, the sieve modulus becomes quite large and requires we keep track of too many residue classes to be practical. Instead, we adapted the space-saving wheel datastructure [18,19], which had been used successfully to sieve for pseudosquares. Sieving for primes \( p_t \) with specified quadratic character modulo a list of small primes is the same algorithmic problem. This wheel sieve uses a datastructure that takes space proportional to \( \sum_{a \in \nu} a \) instead of space proportional to \( \prod_{a \in \nu} a \) as used in traditional sieving. It does, however, produce candidate primes \( p_t \) out of order. This is not an issue so long as we make sure the sieve modulus does not exceed \( B \). In practice, we dynamically include primes from \( \nu \) so that

\[
1000 \cdot k \cdot \text{lcm}\{\lambda, 8, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31\} < \psi_{13},
\]

and we favor the inclusion of smaller primes.
Algorithm 2: Sieving

**Input:** $k$ in factored form as $k = p_1 p_2 \cdots p_{t-1}$, $\lambda_{p_i}$ for $i < t$, search bound $B$, and base vector $\nu$ of length $m$.

1. Compute $\lambda_k = \text{lcm}\{\lambda_{p_i}\}$.
2. Set $w = 1$. Compute wheel modulus $w$ as follows:
3. for $i = 2, \ldots, m$ do
4. if $k \lambda_k a_i w < B/1000$ then
5. if $a_i$ does not divide $\lambda_k$ then
6. Set $w = w \cdot a_i$.
7. end if
8. end if
9. end for
10. if $\lambda_k$ divisible by 4 then
11. Set $\lambda = \lambda_k$.
12. else
13. Set $\lambda = \lambda_k/2$ and
14. Set $w = 8w$.
15. end if
16. Let $\sigma$ be the signature of $p_1$.
17. Build a wheel with modulus $w$ so that for each prime power $q \mid w$ we have primes $p$ generated by the wheel with $(p/q)$ consistent with $\sigma$ as in Propositions 2.5 and 2.6.
18. for each residue $r$ modulo $w$ generated by the wheel do
19. Use the Chinese Remainder Theorem to compute $x \mod w\lambda$ such that $x \equiv r \mod w$ and $x \equiv k^{-1} \mod \lambda$.
20. Sieve for primes in the interval $[k+1, B/k]$ that are $\equiv x \mod w\lambda$.
21. For each such probable prime $p_t$ found, perform a strong pseudoprime test on $kp_t$.
22. end for

Note that if $k$ consists entirely of primes that are $\equiv 3 \mod 4$, then all primes $p_t$ found by the sieve, when using all of $\nu$, will have the correct signature.

**Example 2.7.** Consider $p = 3 \cdot 10^8 + 317$. Now, $\lambda = p - 1 = 2^2 \cdot 7 \cdot 11 \cdot 23 \cdot 42349$ (factorization is obtained via sieving). We may use signature information for the primes 3, 5, 13, 17, 19, 29, 31, and 37. However, since $B = 4 \cdot 10^{24}$ we may only use the first 4 primes in the signature before the combined modulus is too large. The signature information for 3 implies that $q \equiv 5, 7 \pmod{12}$; however, the case $q \equiv 7 \pmod{12}$ need not be considered since we know (because of $\lambda$) that $q \equiv 1 \pmod{4}$. Given the signature information for the primes 5 and 13, we search in $\{2, 3 \pmod{5}\}$ and $\{2, 5, 6, 7, 8, 11 \pmod{13}\}$. At this point we sieve for primes $q \equiv 1 \pmod{\lambda}$. To this we add the following 12 new congruence conditions and sieve for each individual condition.

1. Let $q \equiv 2 \pmod{5}$.
   (a) Let $q \equiv 2 \pmod{13}$.
   (b) Let $q \equiv 5 \pmod{13}$.
   (c) Let $q \equiv 6 \pmod{13}$.
   (d) Let $q \equiv 7 \pmod{13}$. 
(e) Let $q \equiv 8 \pmod{13}$.
(f) Let $q \equiv 11 \pmod{13}$.

(2) Let $q \equiv 3 \pmod{5}$.
(a) Let $q \equiv 2 \pmod{13}$.
(b) Let $q \equiv 5 \pmod{13}$.
(c) Let $q \equiv 6 \pmod{13}$.
(d) Let $q \equiv 7 \pmod{13}$.
(e) Let $q \equiv 8 \pmod{13}$.
(f) Let $q \equiv 11 \pmod{13}$.

2.4. Hash table datastructure. In order to compute GCDs or sieve as described above, it is necessary to construct integers $k = p_1p_2\cdots p_{t-1}$ in an efficient way, where all the primes $p_i$ have matching signatures. We do this by storing all small primes in a hash table datastructure that supports the following operations:

- Insert the prime $p$ with its signature $\sigma_p$. We assume any prime being inserted is larger than all primes already stored in the datastructure. (That is, insertions are monotone increasing.) We also store $\lambda_p$ with $p$ and its signature for use later, thereby avoiding the need to factor $p - 1$.
- Fetch a list of all primes from the datastructure in the form of a sorted array $s[]$ whose signatures match $\sigma$. The fetch operation brings the $\lambda_p$ values along as well.

We then use this algorithm to generate all candidate $k$ values for a given $t$ and $\nu$:

**Algorithm 3: Generating $k$ values**

**Input:** $t > 2$, search bound $B$, cutoff $X < B$, base vector $\nu$ of length $m$.

1. Let $T$ denote our hash table datastructure.
2. Let $a_{m+1}$ denote the smallest prime not in $\nu$.
3. for each prime $p \leq \sqrt{B/a_{m+1}^{t-2}}$ (with $p - 1$ in factored form) do
4. Compute $\lambda_p$ from the factorization of $p - 1$.
5. $s[] := T.fetch(\sigma_p)$ (list of primes with matching signature).
6. for $0 \leq i_1 < \cdots < i_{t-2} \leq s.length$ do
7. Form $k = s[i_1] \cdots s[i_{t-2}] \cdot p$.
8. if $k \leq X$ then
9. Use $k$ in a GCD computation to find matching $p_t$ values.
10. else
11. Use $k$ for a sieving computation to find matching $p_t$ values.
12. end if
13. end for
14. if $p \leq (B/a_{m+1}^{t-3})^{1/3}$ then
15. $T.insert(p, \sigma_p, \lambda_p)$.
16. end if
17. end for

Note that the inner for-loop above is coded using $t-2$ nested for-loops in practice. This is partly why we wrote separate programs for each value of $t$. To make this work as a single program that handles multiple $t$ values, one could use recursion on $t$. 
Also note that we split off the GCD and sieving computations into separate programs, so that the inner for-loop was coded to make sure we always had either \( k \leq X \) or \( X < k < B/p \) as appropriate.

We implemented this datastructure as an array or table of \( 2^m \) linked lists and used a hash function on the prime signature to compute which linked list to use. The hash function \( h : \sigma \to 0, \ldots, (2^m - 1) \) computes its hash value from the signature as follows:

- If \( \sigma = (0, 0, 0, \ldots, 0) \), the all-zero vector, then \( h(\sigma) = 0 \), and we are done.
- Otherwise, compute a modified copy of the signature, \( \sigma' \). If \( \sigma \) contains only 0s and 1s, then \( \sigma' = \sigma \). If not, find the largest integer entry in \( \sigma \), and entries in \( \sigma' \) are 1 if the corresponding entry in \( \sigma \) matches the maximum, and 0 otherwise.
  
  For example, if \( \sigma = (3, 4, 0, 4, 2, 1, 2, 4) \) (our example from earlier), then \( \sigma' = (0, 1, 0, 1, 0, 0, 0, 1) \).
- \( h(\sigma) \) is the value of \( \sigma' \) viewed as an integer in binary.

So for our example, \( h((3, 4, 0, 4, 2, 1, 2, 4)) = 01010001_2 = 64 + 16 + 1 = 81 \).

Computing a hash value takes \( O(m) \) time.

Each linked list is maintained in sorted order; since the insertions are monotone, we can simply append insertions to the end of the list, and so insertion time is dominated by the time to compute the hash value, \( O(m) \) time. Note that the signature and \( \lambda_p \) are stored with the prime for use by the fetch operation.

The fetch operation computes the hash value and then scans the linked list, extracting primes (in order) with matching signature. If the signature to match is binary (which occurs roughly half the time), we expect about half of the primes in the list to match, so constructing the resulting array \( s[] \) takes time linear in \( m \) multiplied by the number of primes fetched. Note that \( \lambda \) values are brought along as well. Amortized over the full computation, fetch will average time that is linear in the size of the data returned. Also note that the average cost to compare signatures that don’t match is constant.

The total space used by the datastructure is an array of length \( 2^m \), plus one linked list node for each prime \( \leq B^{1/3} \). Each linked list node holds the prime, its \( \lambda \) value, and its signature, for a total of \( O(2^m \log B + mB^{1/3}) \) bits, since each prime is \( O(\log B) \) bits.

2.5. Algorithm outline. Now that all the pieces are in place, we can describe the algorithm as a whole.

Given an input bound \( B \), we compute the GCD/sieve cutoff \( X \) (this is discussed in § below).

We then compute GCDs as discussed above using candidate \( k \) values \( \leq X \) for \( t = 2, 3, \ldots \). For some value of \( t \), we’ll discover that there are no candidate \( k \) values with \( t - 1 \) prime factors, at which point we are done with GCD computations.

Next we sieve as discussed above, using candidate \( k \) values \( > X \) for \( t = 2, 3, \ldots \). Again, for some \( t \) value, we’ll discover there are no candidate \( k \) values with \( t - 1 \) prime factors, at which point we are done.

We wrote a total of eight programs: GCD programs for \( t = 2, 3, 4 \) and sieving programs for \( t = 2, 3, 4, 5, 6 \). Also, we did not bother to use the wheel and signature sieving, relying only on \( \lambda \) sieving, for \( t > 3 \).
3. Algorithm analysis

3.1. Model of computation. We assume a RAM model, with potentially infinite memory. Arithmetic operations on integers of $O(\log B)$ bits (single-precision) and basic array indexing and control operations all are unit cost. We assume fast, FFT-based algorithms for multiplication and division of large integers. For $n$-bit inputs, multiplication and division cost $M(n) = O(n \log n \log \log n)$ operations [17]. From this, an FFT-based GCD algorithm takes $O(M(n) \log n) = O(n (\log n)^2 \log \log n)$ operations; see, for example, [20].

3.2. Helpful number theory results. Let our modulus $q = \prod a_i \in \nu a_i$ so that $\phi(q) = \prod a_i \in \nu \phi(a_i)$; each base $a_i$ is an odd prime with $\phi(a_i) = a_i - 1$, except for $a_1 = 8$ with $\phi(a_1) = 4$. We use our two propositions from §2.3 together with quadratic reciprocity to obtain a list of $(a_i - 1)/2$ residue classes for each odd prime $a_i$ and two residue classes modulo 8. The total number of residue classes is $2^{-m} \phi(q)$. A prime we are searching for with a matching signature must lie in one of these residue classes.

Let us clarify that when we state in sums and elsewhere that a prime $p$ has signature equivalent to $\sigma$, we mean that the quadratic character $(p/a_i)$ is consistent with $\sigma$ as stated in Propositions 2.5 and 2.6. We write $\sigma \equiv \sigma_p$ to denote this consistency under quadratic character, which is weaker than $\sigma = \sigma_p$.

Lemma 3.1. Assume given a signature $\sigma$ of length $m$, and let $q = \prod a_i$ as above. Let $\epsilon > 0$. If $q < x^{1-\epsilon}$, then the number of primes $p \leq x$ with $\sigma_p \equiv \sigma$ is at most

$$O\left(\frac{x}{2^m \log x}\right).$$

Proof. This follows almost immediately from the Brun-Titchmarsh Theorem. See, for example, Iwaniec [9]. We simply sum over the relevant residue classes mod $q$ based on quadratic character using Propositions 2.5 and 2.6. \hfill \Box

We choose $m$ proportional to $\log B / \log \log B$ so that $q = B^c$ for some constant $0 < c < 1$. Our running time bound implies an upper bound on the number of strong pseudoprimes. Carl Pomerance pointed out that with $m$ this large, the usual heuristics, due to Erdős, about the density of Carmichael numbers likely do not apply. Those heuristics, for much smaller $m$, would imply at least $B^{1-\epsilon}$ strong pseudoprimes, which in turn require at a minimum the same running time. See [17].

Lemma 3.2. We have

$$\sum_{p \leq x, \sigma_p \equiv \sigma} \frac{1}{p} = O(2^{-m} \log \log x).$$

This follows easily from Lemma 3.1 by partial summation. We could make this tighter by making use of Theorem 4 from [12], and observe that the constants $C(q,a)$ (in their notation) cancel one another in a nice fashion when summing over different residue classes modulo $q$. See also section 6 of that paper.

Let $\pi_{t,\sigma}(x)$ denote the number of integers $\leq x$ with exactly $t$ distinct prime divisors, each of which has signature equivalent to $\sigma$ of length $m$. 

Lemma 3.3.
\[ \pi_{t,\sigma}(x) \ll x(\log \log x)^{t-1} \frac{1}{2^m \log x}. \]

Proof. Proof is by induction on \( t \). \( t = 1 \) follows from Lemma 3.1.
For the general case, we sum over the largest prime divisor \( p \) of \( n \),
\[ \pi_{t,\sigma}(x) = \sum_{p \leq x, \sigma_p = \sigma} \pi_{t-1,\sigma}(x/p) \]
\[ \ll \sum_{p \leq x, \sigma_p = \sigma} x(\log \log(x/p))^{t-2} \frac{1}{p^{2(t-1)m} \log(x/p)} \]
\[ \ll x(\log \log x)^{t-1} \frac{1}{2^m \log x}. \]

We used Lemma 3.2 for the last step. (See also the proof of Theorem 437 in [8, §22.18].) \( \Box \)

Let \( \lambda_{p,m} \) denote the value of \( \lambda_p \) for the prime \( p \) using a vector \( \nu \) with the first \( m \) prime bases. We need the following lemma from [13, Cor. 2.4].

Lemma 3.4. There exists a constant \( \tau > 0 \) such that
\[ \sum_{p \leq x} 1_{\lambda_{p,m}} \ll \frac{x^1/(m+1)}{(\log x)^{1+\tau/(m+1)}}. \]

Using this, we obtain the following.

Theorem 3.5. Let \( 0 < \theta < 1 \). Then
\[ \sum_{x^\theta < p \leq x} \frac{1}{p\lambda_{p,m}} \ll \frac{1}{x^{\theta-1/(m+1)} \log x}. \]

Proof. We have
\[ \sum_{x^\theta < p \leq x} \frac{1}{p\lambda_{p,m}} \leq \frac{1}{x^\theta} \sum_{x^\theta < p \leq x} \frac{1}{\lambda_{p,m}} \]
\[ \ll \frac{1}{x^\theta} \frac{x^{1/(m+1)}}{(\log x)^{1+\tau/(m+1)}}. \]

Note that \( \tau > 0 \). \( \Box \)

3.3. A conjecture. In order to prove our \( B^{2/3+o(1)} \) running time bound for sieving for \( t > 2 \), we make use of the conjecture below. Let \( \mathcal{K} = \mathcal{K}(t, \sigma) \) be the set of integers with exactly \( t \) distinct prime divisors each of which has signature matching \( \sigma \). In other words, \( \pi_{t,\sigma}(x) \) counts integers in \( \mathcal{K} \) bounded by \( x \).

Conjecture 3.6. Let \( B, \sigma, t, m, \nu \) be as above. Then
\[ \sum_{k \leq x, k \in \mathcal{K}(t,\sigma)} \frac{1}{\lambda_{k,m}} \ll x^{o(1)}. \]

Theorem 3.7. Assuming Conjecture 3.6 is true, we have
\[ \sum_{x^\theta \leq k \leq x, k \in \mathcal{K}(t,\sigma)} \frac{1}{k\lambda_{k,m}} \ll x^{-\theta+o(1)}. \]
Proof. The proof is essentially the same as that of Theorem 3.5. Simply substitute Conjecture 3.6 for Lemma 3.4. □

3.4. Factoring. In practice, the cost of factoring in Step 14 of Algorithm 1 is negligible. Trial division is perfectly fine in practice. Proving that factoring is not the runtime bottleneck is, however, a bit challenging. We conjecture that factoring is rare:

Conjecture 3.8. Given $x > 0$ sufficiently large and an integer $m$ that grows with $x$, the number of integers $k \leq x$, such that there exists a prime $p > k$ where $kp$ is a strong pseudoprime to the first $m$ prime bases, is at most $x^{o(1)}$.

In theory, the result of a GCD computation can be quite large. We have, for example, the upper bound

$$\text{gcd}(2^{k-1} - 1, 3^{k-1} - 1) \ll \exp(\epsilon k)$$

for any fixed $\epsilon > 0$ from [4], which is tight. This means the GCD result can have up to $\epsilon k$ bits, making simple arithmetic operations very costly. However, we don’t need to completely factor these numbers; we are only concerned with prime divisors up to $B/k$. With trial division, this is $B/k \cdot \epsilon k = O(B)$ time for each preproduct $k$; this is way too slow. Andrew Shallue pointed out to us that we can use the Pollard-Strassen factoring algorithm (see, for example, [5 §5.5]), which can be implemented to only find prime divisors up to $B/k$ in roughly $\sqrt{B/k} \cdot \epsilon k = O(\sqrt{Bk})$ time for each $k$. This gives us the following lemma.

Lemma 3.9. Given a bound $B$ and cutoff $X$, let $m$ grow with $B$ and assume Conjecture 3.8 is true. The total cost of factoring GCD results in Step 14 of Algorithm 1 is $(BX)^{1/2+o(1)}$.

Proof. We apply the conjecture with $x = X$ and obtain an upper bound by replacing $k$ with $X$ to get $\sqrt{BX} \cdot X^{o(1)}$. □

In the following section we see that choosing $X = B^{1/3}$ optimizes the running time and, with this choice for $X$, factoring also takes $B^{2/3+o(1)}$ time, assuming our conjecture is true.

Andrew Shallue pointed out that if we admit to making factoring the algorithm bottleneck (which is not close to true in practice) we can remove Conjecture 3.8 re-optimize $X$, and obtain a $B^{5/7+o(1)}$ running time. We leave the details to the reader.

As a final comment, we note that we can replace our conjecture with the following, and obtain a $B^{2/3+o(1)}$ runtime bound a slightly different way:

Conjecture 3.10. Let $\nu = \{a_i\}$ be the first $m$ primes. Then

$$\text{gcd}(2^{k-1} - 1, 3^{k-1} - 1, \ldots, a_{m-1}^{k-1} - 1) \ll \exp(kf(m)),$$

where $f(m)$ is a function that goes to zero with $m$.

This would lead to a running time of

$$\sum_{k \leq X} \sqrt{B/k} \cdot k^{o(1)} = (BX)^{1/2+o(1)},$$

the same bound obtained with our lemma.
3.5. Proof of Theorem 1.2. We now have the pieces we need to prove our running time bound.

As above, each pseudoprime candidate we construct will have the form \( n = k p_t \), where \( k \) is the product of \( t - 1 \) distinct primes with matching signature. Again, \( \nu = (2, 3, 5, \ldots) \) is our base vector of small primes of length \( m \).

3.5.1. \( t = 2 \). In this case \( k \) is prime.

**GCD computation.** For each prime \( k = p \leq X \) we perform a GCD computation as described in §2.2. The bottleneck of this computation is computing GCDs of \( O(p) \)-bit integers. This gives a running time proportional to

\[
\sum_{p \leq X} M(p) \log p \ll \pi(X) X (\log X)^2 \log \log X \ll X^2 \log X \log \log X.
\]

**Sieving computation.** For each prime \( k = p \) with \( X < p < B / p \), we sieve the interval \([p + 1, B / p]\) for primes that are \( \equiv 1 \mod \lambda_p \). We also employ signature sieving, but for simplicity of analysis, we omit that for now. Using the methods from [18], we can sieve an arithmetic progression of length \( \ell \) in \( O(\ell \log B) \) operations. We do not need proof of primality here, so a fast probabilistic test works fine. This gives a running time proportional to

\[
\sum_{X < p \leq B / p} \frac{B \log B}{p \lambda_p} \ll B \log B \sum_{X < p \leq B / X} \frac{1}{p \lambda_p}.
\]

At this point we know \( X^2 \leq B \) and, to keep the GCD and sieving computations balanced, \( X \geq B^{1/4} \), say. This means Theorem 3.5 applies; we set \( x^\theta = X \) and \( x = B / X \) to obtain

\[
B \log B \sum_{X < p \leq B / X} \frac{1}{p \lambda_p} \ll B \log B \frac{(B / X)^{1/(m+1)}}{X \log(B / X)}
\]

\[
\ll \frac{B}{X} (B / X)^{1/(m+1)}
\]

\[
= (B / X)^{1 + o(1)},
\]

assuming that \( m \to \infty \) with \( B \).

Minimizing \( X^{2+o(1)} + (B / X)^{1+o(1)} \) implies \( X = B^{1/3} \) and gives the desired running time of \( B^{2/3+o(1)} \). This completes our proof. \( \Box \).

3.5.2. \( t > 2 \). In this case \( k \) is composite.

**GCD computation.** For \( t > 2 \) we construct integers \( k = rp \) for computing GCDs, with \( r \) consisting of exactly \( t - 2 \) prime divisors less than \( p \), with signatures matching \( \sigma_p \). For each prime \( p \), we perform a hash table lookup and fetch the list of such primes; this is Step 5 of Algorithm 3. This overhead cost is \( O(m) + O(\pi_{\sigma_p}(p)) = O(m + 2^{-m} p / \log p) \). Summing this over all primes \( p \leq X \) gives \( O(2^{-m} X^2 / \log X) \).
Next, for each prime \( p \leq X \) we construct at most \( \pi_{t-2,\sigma_p}(X/p) \) values for \( r \) (Step 7 of Algorithm 3), and using Lemma 3.3 and multiplying by the cost of computing the GCD, we obtain

\[
\sum_{p \leq X} \pi_{t-2,\sigma_p}(X/p) \cdot M(X) \log X \ll X^2 \left( \frac{\log \log X}{2m} \right)^{t-2} \log X
\]

for the total running time.

**Sieving computation.** Again, the main loop enumerates all choices for the second-largest prime \( p = p_{t-1} \). First, we construct all possible \( k \)-values with \( k > X \), \( k < B/p \), and \( p \mid k \), so \( k/p = r = p_1 \cdots p_{t-2} \) with all the \( p_i < p_{i+1} \) and \( p_{t-2} < p \). We also have \( p_{t-2} < B^{1/3} \). This implies \( p > X^{1/(t-1)} \).

For a given \( p \), fetching the list of primes below \( u := \min \{ p, B/p^2 \} \) with matching signatures takes \( O(m + u/(2^m \log u)) \) time (Algorithm 3, Step 5). Summing over all \( p \leq \sqrt{B} \), splitting the sum at \( p = B^{1/3} \), takes \( 2^{-m} B^{2/3+o(1)} \) time.

As above, we write \( r = k/p \). We claim that the total number of \( k \) values is \( 2^{-m(t-2)} B^{2/3+o(1)} \). Let \( u \) be as above. There are at most \( \pi_{t-2,\sigma_p}(u) \) values of \( r \) for each \( p \). Again, splitting this at \( B^{1/3} \), we have

\[
\sum_{X^{1/(t-1)} < p \leq B^{1/3}} \pi_{t-2,\sigma_p}(p) \ll \sum_{X^{1/(t-1)} < p \leq B^{1/3}} \frac{p \log \log p)^{t-3}}{2((t-2)m \log p)} \ll 2^{-m(t-2)} B^{2/3+o(1)}
\]

and

\[
\sum_{B^{1/3} < p \leq \sqrt{B}} \pi_{t-2,\sigma_p}(B/p^2) \ll \sum_{B^{1/3} < p \leq \sqrt{B}} \frac{B \log \log B)^{t-3}}{p 2((t-2)m \log B)} \ll \frac{1}{B^{1/3} \log B} \frac{B \log \log B)^{t-3}}{2((t-2)m \log B} \ll 2^{-m(t-2)} B^{2/3+o(1)}.
\]

This covers work done at Step 7 of Algorithm 3.

Finally, the cost of sieving is at most

\[
\sum_{X^{1/(t-1)} < p \leq \sqrt{B}} \sum_r \frac{B \log B}{rp \lambda_{rp}}.
\]

If we use Theorem 3.7 based on our conjecture, and use the lower bound \( X \leq k = rp \), we obtain the bound

\[
\left( \frac{\log \log B}{2m} \right)^{t-2} \frac{B^{1+1/(m+1)+o(1)}}{X}.
\]

Without the conjecture, we use Lemma 3.2 together with the argument outlined in Hardy and Wright [8] in deriving (22.18.8) to obtain that

\[
\sum_r \frac{1}{r} \ll \left( \frac{\log \log B}{2m} \right)^{t-2}.
\]

We then use

\[
\frac{1}{\lambda_{rp}} \leq \frac{1}{\lambda_p}.
\]
As above, this leads to the bound

\[
\left( \frac{\log \log B}{2^m} \right)^{t-2} B^{1+1/(m+1)} X^{1/(t-1)}.
\]

To balance the cost of GCDs with sieving, we want to balance (3.2) with (3.3) or (3.4), depending on whether we wish to assume Conjecture 3.6 or not. Simplifying a bit, this means balancing \(X^{2+o(1)}\) with either \((B/X)^{1+o(1)}\) or \(B^{1+o(1)}/X^{1/(t-1)}\). The optimal cutoff point is then either \(X = B^{1/3}\) under the assumption of Conjecture 3.6 or \(X = B^{(t-1)/(2t-1)}\) unconditionally, for a total time of

\[
\left( \frac{\log \log B}{2^m} \right)^{t-2} B^{2/3+o(1)}
\]

with our conjecture, or

\[
\left( \frac{\log \log B}{2^m} \right)^{t-2} B^{1-\frac{1}{2t} + o(1)}
\]

without. We have proven the following.

**Theorem 3.11.** Assuming Conjectures 3.6 and 3.8, our algorithm takes \(B^{2/3+o(1)}\) time to find all integers \(n \leq B\) that are strong pseudoprimes to the first \(m\) prime bases, if \(m \to \infty\) with \(B\).

If we choose \(m\) so that \(q\) is a fractional power of \(B\), then Lemma 3.3 implies that there is a constant \(c > 0\) such that if \(t > c \log \log B\), then there is no work to do. This explains why our computation did not need to go past \(t = 6\). It also explains why, in practice, the \(t = 2\) case is the bottleneck of the computation and is consistent with the implications of our conjecture that the work for each value of \(t\) decreases as \(t\) increases.

### 4. Implementation Details

In this final section, we conclude with some details from our algorithm implementation.

#### 4.1. Strong pseudoprimes found

In addition to our main results for \(\psi_{12}\) and \(\psi_{13}\), we discovered the following strong pseudoprimes. This first table contains pseudoprimes found while searching for \(\psi_{12}\).

| \(n\) | Factorization |
|-------|---------------|
| 3825123056546413051 | 747451 · 149491 · 34233211 |
| 230245660726188031 | 787711 · 214831 · 1360591 |
| 360681321802296925566181 | 424665351661 · 849330703321 |
| 16428021864367263986221 | 286600958341 · 573201916681 |
| 318665857834031151167461 | 399165290221 · 798330580441 |
| 7395010240794120709381 | 60807114061 · 121614228121 |
| 16428021864367263986221 | 286600958341 · 573201916681 |
This second table contains pseudoprimes found while verifying $\psi_{13}$.

| $n$ | Factorization                                                                                                                                 |
|-----|---------------------------------------------------------------------------------------------------------------------------------------------|
| 318665857834031151167461 | 399165290221·798330580441                                                                                                                   |
| 2995741773170734841812261 | 12238753555821·2447750711641                                                                                                                |
| 667636712015520329618581 | 577770158461·1155540316921                                                                                                                  |
| 3317044064679887385961981 | 1287836182261·2575672364521                                                                                                                 |
| 311026909730073345712981 | 1247050393261·2494100678521                                                                                                                 |
| 55272788069763694556181 | 525703281661·105140653321                                                                                                                   |
| 36068132180296925566181 | 424665351661·849330703321                                                                                                                   |
| 7395010240794120709381 | 608071140661·121614228121                                                                                                                   |
| 3404730287403079539471001 | 1304747157001·2609494314001                                                                                                                 |
| 164280218643672633986221 | 286600958341·573201916681                                                                                                                   |

4.2. **Hash table datastructure.** It is natural to wonder how much time is needed to manage the hash table datastructure from §2.4. Specifically, we measured the time to create the table when inserting all primes up to a set bound. For this measurement, we used a vector $\nu$ with the first 8 prime bases. We also measured how long it took, for each prime up to the bound listed, to also perform the fetch operation, which returns a list of primes with matching signature with $\lambda$ values.

| Largest Prime | Table Entries | Creation Time | Fetching Time |
|---------------|---------------|---------------|---------------|
| $10^6$        | 78490         | 1.13          | 2.58          |
| $10^7$        | 664571        | 11.49         | 244.49        |
| $10^8$        | 5761447       | 1:56.22       | 3:30:50.2     |

The times are given in seconds, or minutes:seconds, or hours:minutes:seconds as appropriate.

We felt the datastructure was a success in that it was fast and small enough that it was not noticed in the larger computation.

4.3. **Almost-linear GCDs.** In Figure 1, we present the times to perform Algorithm 1 on selected prime values for $k$ (so this is for the $t = 2$ case). The data points were obtained by averaging the time needed to carry out Algorithm 1 for the
first ten primes exceeding \( n \cdot 10^7 \) for \( 1 \leq n \leq 35 \). Averaging was required because the size of the \( h \) values is not uniformly increasing in \( k \). This explains the variance in the data; for example, the first ten primes after \( 35 \cdot 10^7 \) happened to have smaller \( h \) values on average than the first ten primes after \( 34 \cdot 10^7 \).

It should be clear from the data that our GMP algorithm for computing GCDs was using an essentially linear-time algorithm. Note that if we chose to extend the computation to large enough \( k \) values, memory would become a concern and we would expect to see paging/thrashing degrade performance.

4.4. **GCD/sieving crossover.** In Figure 2, we present a comparison of the timings for computing GCDs (Algorithm 1, diamonds on the graph) with signature sieving (Algorithm 2, squares on the graph). For this graph, we are looking at choosing the crossover point \( X \), and again, we are focusing on the \( t = 2 \) case.

- One would expect that the signature sieving curve should behave as an inverse quadratic; the time to sieve for one prime \( k = p \) is proportional to

\[
\frac{B \log B}{p \lambda_p},
\]

and we expect \( \lambda_p \approx p - 1 \) most of the time. However, utilizing signatures obscures this to some degree, since the algorithm cannot make use of all prime bases if they divide \( \lambda \), hence the minor variations in the curve. Let us elaborate on this here.

The data points for signature sieving in Figure 2 represent the average sieve time for the first 50 primes past \( n \cdot 10^7 \) for \( 10 \leq n \leq 60 \). There is a lot of variance in these timings because of the inclusion and exclusion of primes from the signature depending on whether they are relatively prime to \( \lambda \).

![Figure 2. Signature sieving and GCD timing](image-url)
For example, \( p = 174594421 \) has \( \lambda = 2^2 \cdot 3^3 \cdot 5 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \) and therefore has few primes from the base set to use in signature sieving. The inability to add extra primes shows up in the timing data; a careful inspection of the data shows a strange jump when primes transition from \( 28 \cdot 10^7 \) to \( 29 \cdot 10^7 \). The data points to a steady decrease, then an almost twofold increase in time followed by a steady decrease again. We believe this is because, on average, one less base prime may be used in signature sieving. Using one less base prime results in approximately twice as much work.

- In the computation for \( \psi_{12} \), our actual crossover point \( X \) was \( 30 \cdot 10^7 \), even though the timing data would suggest the optimal crossover point is around \( 12 \cdot 10^7 \). From a cost-efficiency point of view, we chose poorly. However, the choice was made with two considerations. One is that the program to compute GCDs (Algorithm 1) is simple to write, so that program was up and running quickly, and we let it run while we wrote the sieving code (Algorithm 2). Second is that, initially, we did not know which \( \psi_m \) value we would ultimately be able to compute. Since the results from GCD computations apply to all larger values of \( m \), we opted to err in favor of computing more GCDs.

4.5. Signature sieving. Figure 3 shows the impact that signature sieving makes. Here the squares in the graph give \( \lambda \)-sieving times with no signature information used, and the diamonds show the same \( \lambda \)-sieving work done while taking advantage of signature information using the space-saving wheel. Since there is relatively little variance involved in \( \lambda \)-sieving, each point represents the first prime after \( n \cdot 10^7 \) for \( 10 \leq n \leq 60 \). On the same interval, signature sieving practically looks as though it takes constant time compared to \( \lambda \)-sieving. If we had opted to not incorporate signature sieving, then the expected crossover point given these timings would occur when the primes are around \( 42 \cdot 10^7 \).

![Figure 3. Signature sieving and \( \lambda \)-sieving](image-url)
ACKNOWLEDGEMENTS

The authors thank Frank Levinson for his generous support of Butler University’s cluster supercomputer, Big Dawg. They would also like to thank Andrew Shallue and Carl Pomerance for their feedback on earlier drafts of this paper, and Paul Pollack for his help with some details regarding an earlier proof approach to the running time bound for $\lambda$-sieving.

REFERENCES

[1] W. R. Alford, Andrew Granville, and Carl Pomerance, On the difficulty of finding reliable witnesses, Algorithmic number theory (Ithaca, NY, 1994), Lecture Notes in Comput. Sci., vol. 877, Springer, Berlin, 1994, pp. 1–16, DOI 10.1007/bf02635431. MR1322705 (96d:11136)

[2] Eric Bach, Explicit bounds for primality testing and related problems, Math. Comp. 55 (1990), no. 191, 355–380, DOI 10.2307/2008811. MR1023756 (91m:11096)

[3] Daniel Bleichenbacher, Efficiency and security of cryptosystems based on number theory, Ph.D. thesis, Swiss Federal Institute of Technology Zurich, 1996.

[4] Yann Bugeaud, Pietro Corvaja, and Umberto Zannier, An upper bound for the G.C.D. of $a^n - 1$ and $b^n - 1$, Math. Z. 243 (2003), no. 1, 79–84, DOI 10.1007/s00209-002-0449-z. MR1953049

[5] Richard Crandall and Carl Pomerance, Prime numbers, Springer-Verlag, New York, 2001. A computational perspective. MR1821158

[6] François G. Dorais and Dominic Klyve, A Wieferich prime search up to $6.7 \times 10^{15}$, J. Integer Seq. 14 (2011), no. 9, Article 11.9.2, 14. MR2859986

[7] P. Erdős, On pseudoprimes and Carmichael numbers, Publ. Math. Debrecen 4 (1956), 201–206. MR0080031

[8] G. H. Hardy and E. M. Wright, An introduction to the theory of numbers, 5th ed., The Clarendon Press, Oxford University Press, New York, 1979. MR568909 (81i:10002)

[9] Henryk Iwaniec, On the Brun-Titchmarsh theorem, J. Math. Soc. Japan 34 (1982), no. 1, 95–123, DOI 10.2969/jmsj/03410095. MR639808 (83a:11042)

[10] Gerhard Jaeschke, On strong pseudoprimes to several bases, Math. Comp. 61 (1993), no. 204, 915–926, DOI 10.2307/2153262. MR1192971 (94d:11004)

[11] Yupeng Jiang and Yingpu Deng, Strong pseudoprimes to the first eight prime bases, Math. Comp. 83 (2014), no. 290, 2915–2924, DOI 10.1090/S0025-5718-2014-02830-5. MR3246815

[12] A. Languasco and A. Zaccagnini, A note on Mertens’ formula for arithmetic progressions, J. Number Theory 127 (2007), no. 1, 37–46, DOI 10.1016/j.jnt.2006.12.015. MR2351662 (2009g:11136)

[13] Francesco Pappalardi, On the order of finitely generated subgroups of $\mathbb{Q}^*$ (mod $p$) and divisors of $p - 1$, J. Number Theory 57 (1996), no. 2, 207–222, DOI 10.1006/jnth.1996.0044. MR1382747 (97d:11141)

[14] Francesco Pappalardi, On the $r$-rank Artin conjecture, Math. Comp. 66 (1997), no. 218, 853–868, DOI 10.1090/S0025-5718-97-00805-3. MR1377664 (97f:11082)

[15] Carl Pomerance, J. L. Selfridge, and Samuel S. Wagstaff Jr., The pseudoprimes to $25 \cdot 10^9$, Math. Comp. 35 (1980), no. 150, 1003–1026, DOI 10.2307/2006210. MR575272 (82g:10030)

[16] Michael O. Rabin, Probabilistic algorithm for testing primality, J. Number Theory 12 (1980), no. 1, 128–138, DOI 10.1016/0021-9991(80)90084-0. MR566880 (81f:10003)

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

[18] Jonathan P. Sorenson, The pseudosquares prime sieve, Algorithmic number theory, Lecture Notes in Comput. Sci., vol. 4076, Springer, Berlin, 2006, pp. 193–207, DOI 10.1007/11792086-15. MR2282925 (2007m:11168)

[19] Jonathan P. Sorenson, Sieving for pseudosquares and pseudocubes in parallel using doubly-focused enumeration and wheel datastructures, Algorithmic number theory, Lecture Notes in Comput. Sci., vol. 6197, Springer, Berlin, 2010, pp. 331–339, DOI 10.1007/978-3-642-14518-6_26. MR2721430 (2011m:11245)
[20] Damien Stehlé and Paul Zimmermann, *A binary recursive gcd algorithm*, Algorithmic number theory, Lecture Notes in Comput. Sci., vol. 3076, Springer, Berlin, 2004, pp. 411–425, DOI 10.1007/978-3-540-24847-7_31. MR2138011 (2006e:11194)

[21] Zhenxiang Zhang, *Two kinds of strong pseudoprimes up to $10^{36}$*, Math. Comp. 76 (2007), no. 260, 2095–2107 (electronic), DOI 10.1090/S0025-5718-07-01977-1. MR2336285 (2008h:11114)

Department of Computer Science and Software Engineering, Butler University, Indianapolis, Indiana 46208
E-mail address: sorenson@butler.edu

Department of Mathematics and Actuarial Science, Butler University, Indianapolis, Indiana 46208
E-mail address: jewebste@butler.edu