

# The Pseudosquares Prime Sieve

Jonathan P. Sorenson\*

Computer Science and Software Engineering,  
Butler University  
Indianapolis, IN 46208 USA  
sorenson@butler.edu  
<http://www.butler.edu/~sorenson>

**Abstract.** We present the pseudosquares prime sieve, which finds all primes up to  $n$ . Define  $p$  to be the smallest prime such that the pseudosquare  $L_p > n/(\pi(p)(\log n)^2)$ ; here  $\pi(x)$  is the prime counting function. Our algorithm requires only  $O(\pi(p)n)$  arithmetic operations and  $O(\pi(p)\log n)$  space. It uses the pseudosquares primality test of Lukes, Patterson, and Williams.

Under the assumption of the Extended Riemann Hypothesis, we have  $p \leq 2(\log n)^2$ , but it is conjectured that  $p \sim \frac{1}{\log 2} \log n \log \log n$ . Thus, the conjectured complexity of our prime sieve is  $O(n \log n)$  arithmetic operations in  $O((\log n)^2)$  space. The primes generated by our algorithm are proven prime unconditionally. The best current unconditional bound known is  $p \leq n^{1/(4\sqrt{\epsilon}-\epsilon)}$ , implying a running time of roughly  $n^{1.132}$  using roughly  $n^{0.132}$  space.

Existing prime sieves are generally faster but take much more space, greatly limiting their range ( $O(n/\log \log n)$  operations with  $n^{1/3+\epsilon}$  space, or  $O(n)$  operations with  $n^{1/4}$  conjectured space). Our algorithm found all 13284 primes in the interval  $[10^{33}, 10^{33} + 10^6]$  in about 4 minutes on a 1.3GHz Pentium IV.

We also present an algorithm to find all pseudosquares  $L_p$  up to  $n$  in sublinear time using very little space. Our innovation here is a new, space-efficient implementation of the wheel datastructure.

## 1 Introduction

A *prime number sieve* is an algorithm that finds all prime numbers up to a bound  $n$ . The fastest known sieves take  $O(n/\log \log n)$  arithmetic operations [2,10,18,23], which is quite fast, considering there are  $\pi(n) \sim n/\log n$  primes to find. However in practice, the utility of a prime number sieve is often limited by how much memory space it needs. For example, a sieve that uses  $O(\sqrt{n})$  space [2,19,23] cannot, on current hardware, generate primes larger than about  $10^{18}$ .

---

\* This work was supported by a grant from the Holcomb Research Institute. Special thanks to Hugh Williams for helpful discussions and for hosting me at the University of Calgary in March of 2005, and thanks to the referees, whose comments improved this paper.

Even with Galway's clever improvements [11] to the Atkin-Bernstein sieve [2], the space requirement is still  $n^{1/3+\epsilon}$ , giving an effective limit of roughly  $10^{27}$ . Galway also has a sieve with conjectured space use of  $O(n^{1/4})$  that runs linear time [12, ch. 6]. (Note that the space needed to write down the output, the primes up to  $n$ , is not included.)

If we applied trial division to each integer up to  $n$  separately, we would only need  $O(\log n)$  space, but the time of  $O(n\sqrt{n}/\log n)$  would be prohibitive. We could sieve by a few primes, then apply a quick base-2 pseudoprime test to remove most composites [17] and then use a prime test. If we used the AKS test [1] with Bernstein's complexity improvements [8], the result would be a sieve that takes  $n(\log n)^{2+o(1)}$  operations. (The modified AKS test is  $(\log n)^{4+o(1)}$  bit operations; we save a  $\log n$  factor with the 2-*psp* test, and another  $\log n$  factor because in this paper we count arithmetic operations instead of bit operations.) We can improve the time to  $O(n(\log n)^2)$  by using Miller's prime test [16], but then our output is correct only if the ERH is true. And of course if we are willing to accept probable primes, the Miller-Rabin [20] or Solovay-Strassen [22] tests could give us  $O(n \log n)$  operations. But in most applications for prime sieves, we need to be certain of our output.

In this paper, we present a new prime number sieve, the *pseudosquares prime sieve* (Algorithm PSSPS), that uses very little space and yet is fast enough to be practical. It uses an Eratosthenes-like sieve followed by the pseudosquares prime test of Lukes, Patterson, and Williams [15] (which effectively includes a base-2 pseudoprime test). Our sieve has a conjectured running time of  $O(n \log n)$  arithmetic operations and  $O((\log n)^2)$  bits of space. This is the complexity we observed in practice, and is as fast as using one of the probabilistic tests mentioned above. Assuming the ERH, we obtain  $O(n(\log n)^2/\log \log n)$  operations and  $O((\log n)^3/\log \log n)$  space. But in any case, the primes generated by our sieve are unconditionally proven prime.

Often, the user actually needs to find all primes in a short interval. On average, assuming reasonable conjectures, and after precomputation, our new algorithm will find all primes in an interval containing  $n$  of length at least  $(\log n)^2$  at a cost of  $O(\log n)$  operations per integer. In particular, we found all the primes in the interval  $[10^{33}, 10^{33} + 10^6]$  in just over 4 minutes on a 1.3 GHz Pentium IV running Linux.

We also present a new, space-efficient implementation of the wheel data structure that leads to an algorithm for finding all pseudosquares  $L_p \leq n$  in time  $O(n \cdot \exp[-c \log n / \log \log n])$  for a constant  $c > 0$ . This data structure may prove to be useful in other areas of computational number theory.

For more on finding pseudosquares, see Wooding and Williams [28] in this volume. For recent work on prime number sieves, see [2,11,12,23].

The rest of this paper is organized as follows. In §2 we discuss some preliminaries, including pseudosquares, followed by a description of our algorithm in §3. In §4 we present our new wheel data structure and give our algorithm for finding pseudosquares. We conclude in §5 with some timings.

## 2 Preliminaries

### 2.1 Model of Computation

Our model of computation is a RAM with a potentially infinite, direct access memory. If  $n$  is the input, then all arithmetic and memory access operations on integers of  $O(\log n)$  bits are assigned unit cost. Memory may be addressed either at the bit level or at the word level, where each machine word is composed of  $O(\log n)$  bits.

When we present code fragments, we use a C++ style that should be familiar to most readers [25]. We occasionally declare integer variables with an `INT` datatype instead of the `int` datatype. This indicates that these integers typically exceed 32 bits in practice and may require special implementation (we used the Gnu-MP `mpz_t` datatype and associated functions [13]). We still limit `INT`s to  $O(\log n)$  bits.

The space used by an algorithm under our model is counted in bits. The space used by the output of a prime number sieve (the list of primes up to  $n$ ) is not counted against the algorithm. For further discussion, see [10].

### 2.2 Some Definitions

$p$  always denotes a prime, with  $p_i$  denoting the  $i$ th prime, so that  $p_1 = 2$ . For integers  $a, b$  let  $\gcd(a, b)$  denote the greatest common divisor of  $a$  and  $b$ . We say  $a$  and  $b$  are *relatively prime* if  $\gcd(a, b) = 1$ . For a positive integer  $m$  let  $\phi(m)$  be the number of positive integers up to  $m$  that are relatively prime to  $m$ , with  $\phi(1) = 1$ . The number of primes up to  $x$  is given by  $\pi(x)$ . An integer  $x$  is a square, or *quadratic residue*, modulo  $p$  if the Legendre symbol  $(x/p) = 1$ .

We say  $f(n) = O(g(n))$  if there exists a constant  $c > 0$  such that  $f(n) \leq c \cdot g(n)$  for all sufficiently large  $n$ . We write  $f(n) = \Theta(g(n))$  if  $f(n) = O(g(n))$  and  $g(n) = O(f(n))$ . We say  $f(n) = o(g(n))$  if  $\lim_{n \rightarrow \infty} f(n)/g(n) = 0$ .

### 2.3 Pseudosquares

The *pseudosquare*  $L_p$  is the least non-square positive integer satisfying these two properties:

1.  $L_p \equiv 1 \pmod{8}$ , and
2.  $L_p$  is a quadratic residue modulo every odd prime  $q \leq p$ .

Thus  $L_3 = 73$  and  $L_5 = 241$ . See Williams [26, §16.2].

**Lemma 2.1 (Lukes, Patterson, and Williams[15]).** *Let  $n$  and  $s$  be positive integers. If*

1. *All prime divisors of  $n$  exceed  $s$ ,*
2.  *$n/s < L_p$  for some prime  $p$ ,*

- 3.  $p_i^{(n-1)/2} \equiv \pm 1 \pmod{n}$  for all primes  $p_i \leq p$ ,
- 4.  $2^{(n-1)/2} \equiv -1 \pmod{n}$  when  $n \equiv 5 \pmod{8}$ ,
- $p_i^{(n-1)/2} \equiv -1 \pmod{n}$  for some  $p_i \leq p$  when  $n \equiv 1 \pmod{8}$ ,

then  $n$  is a prime or a prime power.

Note that if  $n$  is prime, then the conditions of the lemma hold with  $s = 1$  and  $n < L_p$ .

### 2.4 Useful Estimates

Here  $x, x_1, x_2 > 0$ , and except for (5), all sums and products are only over primes.

$$\sum_{p \leq x} \frac{1}{p} = \log \log x + O(1); \tag{1}$$

$$\sum_{p \leq x} \log p = x(1 + o(1)); \tag{2}$$

$$\sum_{p \leq x} 1 = \pi(x) = \frac{x}{\log x}(1 + o(1)); \tag{3}$$

$$\prod_{p \leq x} \frac{p-1}{p} = O\left(\frac{1}{\log x}\right); \tag{4}$$

$$\sum_{\substack{x_1 < d \leq x_2 \\ \gcd(d, m) = 1}} \frac{1}{d} = \frac{\phi(m)}{m} \log(x_2/x_1)(1 + o(1)). \tag{5}$$

For proofs of (1)–(4), see Hardy and Wright [14]. For a proof of (5), see [23, Lemma 1].

### 2.5 The Wheel

A *wheel*, as we will use it, is a data structure that encapsulates information about the integers relatively prime to the first  $k$  primes. Generally speaking, a wheel can often be used to reduce the running time of a prime number sieve by a factor proportional to  $\log p_k$ . Pritchard was the first to show how to use a wheel in this way [18,19]. We begin with the following definitions:

$$M_k := \prod_{i=1}^k p_i;$$

$$W_k(y) := \{x \leq y : \gcd(x, M_k) = 1\};$$

$$W_k := W_k(M_k).$$

Let  $\#S$  denote the cardinality of the set  $S$ . We have (see (2) and (4)):

$$\log M_k = p_k(1 + o(1));$$

$$\begin{aligned} \#W_k &= \phi(M_k) = M_k \prod_{i=1}^k \frac{p_i - 1}{p_i} = O\left(\frac{M_k}{\log \log M_k}\right); \\ \#W_k(n) &= O\left(\frac{n}{\log \log M_k}\right). \end{aligned}$$

Our data structure, then, is an array `W[]` of records or structs, indexed by  $0 \dots (M_k - 1)$ , defined as follows:

- `W[x].rp` is 1 if  $x \in W_k$ , and 0 otherwise.
- `W[x].dist` is  $d = y - x$ , where  $y > x$  is minimal with  $\gcd(y, M_k) = 1$ .

We say that `W` is the  $k$ th wheel, with size  $M_k$ . For our C++ notation, we will declare `W` to be of class type `Wheel(k)`, where  $k$  is an integer parameter. We can construct a wheel of size  $M_k$  in  $O(M_k)$  operations.

For examples of the wheel data structure, see [19,24].

### 3 Algorithm PSSPS

#### 3.1 Precomputations and Main Loop

We first construct a table of pseudosquares up to  $n/(\log n)^2$  using the algorithm we describe later in §4. In the code fragment below, this is stored in an array `pss[]` of structs or records:

- `pss[i].prime` is the smallest prime  $p$  (an `int`) such that
- `pss[i].pss` is  $L_p$  (an `INT`).

So `pss[1].prime=3` with `pss[1].pss=73`, and `pss[2].prime=5` with `pss[2].pss=241`. In practice, we can use the table from Wooding [27, pp. 92–93], which has 49 entries, with the largest being `pss[49].pss = 295363487400900310880401`, `pss[49].prime = 353`. Storing this table requires  $O(\pi(p) \log n)$  space.

Next we specify the parameters  $p$ , segment size  $\Delta$ , and sieve limit  $s$ :

- Let  $p$  be the smallest prime such that the pseudosquare  $L_p > n/(\pi(p)(\log n)^2)$ .
- $\Delta := \Theta(\pi(p) \log n)$ . Note  $\Delta \gg p$ .
- $s := \lfloor n/L_p \rfloor + 1 = \Theta(\Delta \log n)$ .

We conjecture  $p \sim (1/\log 2) \log n \log \log n$  (see below). Making  $\Delta$  larger improves overall performance; we choose here to give it roughly the same size as the pseudosquares table so it does not dominate overall space use. Our choice for  $s$  will balance the time spent in sieving versus the time applying the pseudosquares prime test.

In practice, we might choose  $\Delta$  first. One normally chooses  $\Delta$  to be as large as possible yet small enough to fit in cache memory, say around  $2^{20}$ . Then choose  $s = \Theta(\Delta \log n)$ , and pick the smallest prime  $p$  so that  $L_p > n/s$ . If this choice for  $p$  is larger than our largest pseudosquares table entry, we simply set  $p$  to the largest entry (353) and set  $s := \lfloor n/L_p \rfloor + 1$ . Once  $p$  and  $s$  are set, the pseudosquares table is no longer needed.

We wrap up precomputation by building a wheel of size  $\Theta(\log n)$ . In practice, a wheel of size  $30 = 2 \cdot 3 \cdot 5$  ( $k = 3$ ) works fine. We must have  $p_k \leq p$ . Our main loop iterates over segments of size  $\Delta$ .

```

int p,s,delta;
INT l,r,n;           // declare multiprec. ints
input(n);
pssentry pss[];     // Pseudosquare table
PssBuild(pss,n);    // Builds the pseudosq. tbl.
Initialize();       // Compute p,delta,s etc.
Wheel W(pi(log(n))); // Wheel of size O(log n)

/** Main Loop
Primelist PL(p);    // Find primes up to p
output(PL);         // Output primes up to p
for(l=p; l<n; l=l+delta) // Loop over segments
{
    r=min(l+delta,n);
    sieve(l,r,p,s,PL,W); // Sieve the interval [l+1,r]
}

```

Precomputation is dominated by the time to build the pseudosquares table; this is  $o(n)$  operations and  $O(\pi(p) \log n)$  space (we "cripple" the running time of our algorithm from §4 to meet the space bound). Constructing the wheel takes  $O(\log n)$  operations and space. The list of primes up to  $p$  takes at most  $O(p)$  operations and space. We will analyze the cost of the main loop at the end of this section.

### 3.2 Finding Primes in a Segment

Here we implement the `sieve()` function called in the main loop above. We begin by sieving, then we perform the pseudosquares prime test, and we finish by removing perfect powers.

**Sieving.** We sieve by the primes up to  $p$ , and then we sieve by integers from  $p$  to  $s$  using the wheel.

Here our `BitVector` class is created with left and right endpoints ( $\ell$  and  $r$ ), of length  $\Delta$ , that supports functions to set and clear bits. Also, the member function `first( $x$ )` will return the first integer larger than  $\ell$  divisible by  $x$ .

```

BitVector B(delta,l,r); // bit vector for the interval
B.setall();             // assume all are prime to start

/** Sieve by primes up to p
int i; INT x;
for(i=1; i<=PL.length(); i++)

```

```

// Loop through multiples of PL[i]:
for(x=B.first(PL[i]); x<=r; x=x+PL[i])
  B.clear(x);

/** Sieve by integers d up to s, gcd(d,m)=1
int d, m=W.size();          // m is the size of the wheel
for(d=W[p%m].next; d<=min(s,sqrt(l)); d=d+W[d%m].next)
  // Loop through multiples of d:
  for(x=B.first(d); x<=r; x=x+d)
    B.clear(x);

```

At this point,  $B$  represents only those integers from the interval  $[\ell + 1, r]$ , with no prime divisors smaller than  $\min\{s, \sqrt{\ell}\}$ . In practice, one can implement this so that all the work done in these inner loops requires only single-precision integers: simply work with  $x - \ell$  rather than  $x$ .

The time to sieve by primes is proportional to

$$\sum_{p_i \leq p} \left(1 + \frac{\Delta}{p_i}\right) = O(\pi(p) + \Delta \log \log p) = o(\Delta \log n).$$

Sieving by integers generated by the wheel between  $p$  and  $s$  takes time proportional to

$$\sum_{\substack{p < d \leq s \\ \gcd(d, m) = 1}} \left(1 + \frac{\Delta}{d}\right) = O\left(\frac{\phi(m)}{m}(s + \Delta \log(s/p))\right)$$

using (5). This simplifies to  $O((s + \Delta \log(s/p))/\log \log \log n) = o(\Delta \log n)$  using (4). In total, this phase requires  $o(\Delta \log n)$  operations and  $O(\Delta)$  space.

**The Pseudosquares Prime Test.** The next phase of our algorithm is based on Lemma 2.1, due to Lukes, Patterson, and Williams [15]. We code this prime test as function `psspt()`, which tests conditions (3) and (4) of the lemma. We make sure to perform the  $2^{(n-1)/2} \bmod n$  test first, for this has the effect of performing a base-2 pseudoprime test [17].

```

INT x;
for(x=l+1; x<=r; x++) // loop over the interval
  if(B[x]==1          // x meets conditions (1) & (2)
    if(!psspt(x,p))  // if x fails the test
      B.clear(x);    // x is not prime

```

Because of our earlier sieving, only  $O(\Delta/\log s) = O(\Delta/\log \log n)$  integers remain that pass conditions (1) and (2) for our prime test. (Recall that  $s = \Theta(\Delta \log n)$ .) Function `psspt()` will first effectively perform a base-2 pseudoprime test. This takes  $O(\log n)$  arithmetic operations per test, for a total time to this point of  $O(\Delta(\log n)/\log \log n) = o(\Delta \log n)$ . From [17] and elsewhere in the literature, we know that only  $O(n/\log n)$  integers up to  $n$  pass the base-2 pseudoprime

test, or an average of  $O(\Delta/\log n)$  per interval. (A particular interval could conceivably have more than this.) The `psspt()` function performs  $\pi(p) - 1$  more modular exponentiations, at a cost of  $O(\log n)$  arithmetic operations each, on each remaining integer for an overall average cost of  $O(\pi(p)\Delta)$  operations.

**Removing Perfect Powers.** At this point, the only remaining integers represented by `B` are either prime or the power of a prime. Note that if  $n \leq 6.4 \cdot 10^{37}$ , only primes remain and we are done [26, p. 417].

To remove the prime powers, in theory we use a perfect power testing algorithm [6,7,9] which, in our model of computation, requires sublinear time per integer on average, making the cost negligible ( $o(\Delta)$  operations on average, since we only perform the tests on the remaining  $O(\Delta/\log n)$  integers). In practice, one can very efficiently enumerate perfect powers using a priority queue data structure; we leave the details to the reader in the interest of space.

### 3.3 Complexity

Let us summarize what we have from above:

- Precomputation takes  $o(n)$  operations and  $O(\pi(p)\log n)$  space (dominated by building the pseudosquares table).
- Sieving a segment takes  $o(\Delta\log n)$  operations; a segment takes  $O(\Delta) = O(\pi(p)\log n)$  space.
- Performing base-2 pseudoprime tests and the pseudosquares prime test takes, on average,  $O(\pi(p)\Delta)$  operations per interval.
- Removing perfect powers takes  $o(\Delta)$  operations on average.

By multiplying the average cost per segment by  $n/\Delta$ , the number of segments, we prove the following.

**Theorem 3.1.** *Let  $p$  be defined as above. Algorithm PSSPS finds all primes up to  $n$  using  $O(\pi(p)n) + o(n\log n)$  arithmetic operations and  $O(\pi(p)\log n)$  space.*

The work of Bach and Huelsbergen [4] implies the following conjecture.

*Conjecture 3.2.*  $\log L_p \sim \log 2 \frac{p}{\log p}$ , or equivalently,  $p \sim \frac{1}{\log 2} \log L_p \log \log L_p$ .

Lukes, Patterson, and Williams [15] studied the relationship between  $L_p$  and  $p$  for all known pseudosquares, and their data supports the conjecture. See also [28].

**Corollary 3.3.** *If Conjecture 3.2 is true, then Algorithm PSSPS finds all primes up to  $n$  in  $O(n\log n)$  arithmetic operations and  $O((\log n)^2)$  space.*

Fortunately in practice, Conjecture 3.2 appears to hold.

**Corollary 3.4.** *If the ERH is true, then Algorithm PSSPS finds all primes up to  $n$  in  $O(n(\log n)^2/\log \log n)$  arithmetic operations and  $O((\log n)^3/\log \log n)$  space.*

This follows from Bach's Theorem [3], which implies  $p < 2(\log n)^2$ , or asymptotically  $p < (1 + o(1))(\log n)^2$ . Note that this weaker result still outperforms the use of Miller's prime test [16] or AKS [1,8] in a prime sieve.

Currently the best unconditional result is  $p \leq L_p^{1/(4\sqrt{e}-\epsilon)} \approx L_p^{0.1516\dots}$ , due to Schinzel [21]. Since we use  $L_p \approx n/p$ , we obtain that  $p \approx n^{1/(4\sqrt{e}+1-\epsilon)} \approx n^{0.132}$ . This implies the following much weaker result:

**Corollary 3.5.** *Let  $\epsilon > 0$ . Algorithm PSSPS finds all primes up to  $n$  in  $O(n^{1+1/(4\sqrt{e}+1-\epsilon)}) \approx n^{1.132}$  arithmetic operations and  $O(n^{1/(4\sqrt{e}+1-\epsilon)}) \approx n^{0.132}$  space.*

Algorithm 3.1 from [23] would require a running time of roughly  $n^{1.368}$  to stay within the same space bound. Of course, an AKS-based sieve would give the best unconditional result.

## 4 Finding Pseudosquares

In this section we present a sublinear-time algorithm to find all pseudosquares  $L_p \leq n$ . It makes use of a new way to implement a wheel-like datastructure that uses significantly less space.

We begin by presenting our new wheel datastructure, after which we show how to adapt it to find pseudosquares.

### 4.1 A New Wheel

As mentioned in §2.5, the wheel datastructure is used primarily to enumerate integers relatively prime to  $M_k$ , like this:

```
for(x=1; x<n; x=x+W[x%m].dist)
    output(x);
```

Here we present a new implementation of the wheel, which has the following differences:

- The space used by the wheel is proportional to the sum of the moduli instead of their product:  $O(\log M_k \sum_{i=1}^k p_i) = O((\log M_k)^3 / \log \log M_k)$  bits instead of  $O(\log p_k \prod_{i=1}^k p_i) = O(M_k \log \log M_k)$  bits. This is a huge savings.
- The integers relatively prime to  $M_k$  are not enumerated in ascending order.

**An Example - Enumerating Primes up to 100.** Sometimes it is best to introduce a new datastructure with an example. We construct our new wheel with moduli 2, 3, 5, 7 to enumerate 1, plus the primes  $p_i$  with  $7 < p_i \leq 100$ .

For each prime modulus  $p_i$  except for 2, we create an array of structs or records, indexed from  $0 \dots p_i - 1$ , each of which has 2 fields. Let  $m_i$  be the *input modulus*, which is  $m_i := 2 \cdot 3 \cdots p_{i-1}$ . For our example,  $m_2 = 2$ ,  $m_3 = 6$ , and  $m_4 = 30$ . Here  $0 \leq x < p_i$ .



1	31	61	91	11	41	71
37	67	97		17	47	
13	43	73		23	53	83
19	79			29	59	89

The values to the left of the line arise from  $x_2 = 1$ ; they are  $\equiv 1 \pmod 6$ . The values to the right of the line arise from  $x_2 = 5$ ; they are  $\equiv 5 \pmod 6$ .

**Computing jumps.** Computing the `rp` field for each prime is the same as for the basic wheel, and takes time linear in  $p_i$ . Computing the `jump` fields is a bit trickier, and takes  $O(p_i^2 \log p_i)$  operations.

For each column  $x = 0 \dots p_i - 1$ , we do the following:

1. Compute a list of distances to *all* other residue classes that are relatively prime to  $p_i$ .  
For  $p_i = 7$  and  $x = 5$ , we get the list 1, 3, 4, 5, 6.
2. For each distance  $d$  in the list, use the extended Euclidean algorithm [5, §4.3] to find a multiple of the modulus,  $a \cdot p_i$ , such that  $d + ap_i$  is divisible by  $m_i$ , the input modulus.

Continuing our example, for  $d = 1$  we must use  $a = 17$  to get  $1 + 17 \cdot 7 = 120$ . Repeating this for the entire list gives  $120 = 17 \cdot 7 + 1$ ,  $150 = 21 \cdot 7 + 3$ ,  $60 = 8 \cdot 7 + 4$ ,  $180 = 25 \cdot 7 + 5$ ,  $90 = 12 \cdot 7 + 6$ .

3. Write down the smallest number from the list computed in the last step.  
In our example, it is 60.

The value of `jump` entries will not exceed  $p_i m_i$ .

The total time to build a datastructure for the first  $k$  primes is proportional to  $k \cdot p_k^2 \log p_k = O(p_k^3)$  operations. The space needed is proportional to  $k \cdot p_k \log M_k = O(p_k^3 / \log p_k)$ .

If  $n > M_k$ , then analyzing the running time reduces to counting the number of times `output(x)` is called, which gives us  $O((\phi(M_k)/M_k)n)$  operations.

**Theorem 4.1.** *Let  $n > M_k$ . Using our new implementation of the wheel datastructure, we can enumerate integers up to  $n$  relatively prime to  $M_k$  in  $O((\phi(M_k)/M_k)n)$  operations. Precomputing the datastructure requires  $O(p_k^3)$  operations and  $O(p_k^3 / \log p_k)$  space.*

## 4.2 Enumerating Pseudosquares

To search for pseudosquares  $L_p \leq n$ , we simply make a few minor changes to our new wheel datastructure and `enumerate()` function from above:

1. We choose  $k$  so that  $M_k \leq n$ , but as large as possible. We assume that all pseudosquares  $L_{p_i}$  with  $p_i \leq p_k$  are already known. (If not, find them recursively with a smaller  $n$ .)
2. Our first prime is  $p_3 = 5$ , with input modulus  $m_3 = 24$ ; we know  $L_p \equiv 1 \pmod{24}$  for  $p \geq 3$ . Each successive input modulus satisfies  $m_i := p_{i-1} m_{i-1}$ .

3. We change the **rp** field to a **qr** field, set to 1 if  $x$  is a quadratic residue modulo  $p_i$ , and 0 otherwise. This can be computed in linear time by setting all the **qr** bits to 0, then square each integer  $1 \dots p_i - 1$  modulo  $p_i$  and mark the corresponding **qr** field with a 1.
4. Compute the **jump** field from the **qr** field as if it were the **rp** field.
5. Replace  $p_i - 1$  with  $(p_i - 1)/2$  in the loop control for the recursive case of the **enumerate()** function.
6. Integers  $x$  that are output by the **enumerate()** function are checked to see if they are quadratic residues modulo the primes  $p_i$  with  $p_k < p_i \leq p$ . If they pass, then we check to see if they are squares. The average cost for this is  $O(1)$  operations per  $x$  value if we precompute a table of quadratic residues modulo several primes  $p_i > p_k$ .
7. An integer  $x$  that passes all these tests is  $L_p$ ; output it, and find the next prime to serve as  $p$  to begin the search for the next pseudosquare.

The algorithm described above can find all pseudosquares  $L_p \leq n$  in  $O(n2^{-k}/\log p_k)$  operations, as the **output()** function will be called roughly

$$\frac{1}{4} \cdot \prod_{i=1}^k \frac{(p_i - 1)/2}{p_i} \cdot n$$

times. By our choice for  $k$ , we have  $k = \Theta(\log n / \log \log n)$ . We have proven the following.

**Theorem 4.2.** *Our algorithm will find all pseudosquares  $L_p \leq n$  in*

$$O(n \exp[-c \log n / \log \log n])$$

*operations, for  $c > 0$  fixed, using  $O(p + (\log n)^3 / \log \log n)$  space.*

Conjecture 3.2 implies only  $O((\log n)^3 / \log \log n)$  space is needed; assuming the ERH instead does not increase this bound.

We use this algorithm in our prime sieve with  $p_k \approx (\log n)^{2/3}$  to keep our space usage under control, yet maintain a  $o(n)$  running time.

Our crude implementation of this algorithm found  $L_{223} \approx 1.16 \times 10^{16}$  in about 17 hours on a single 1.3GHz Pentium IV processor.

Robert Threlfal observed that this wheel can be used to factor integers of the form  $n = p^2q$ ,  $p, q$  prime, by using  $(-1/n)$ ,  $(2/n)$ ,  $(3/n)$ ,  $(5/n)$ , etc. to initialize the datastructures to search for  $q$ .

## 5 Timing Results

In our first set of results (Table 1), we compared our new sieve to the sieve of Eratosthenes and the Atkin-Bernstein sieve to find the primes up to  $10^9$ .

We used a 1.3 GHz Pentium IV running Linux, with the Gnu g++ compiler. The code for the Atkin-Bernstein and Eratosthenes sieves came, unmodified, from Dan Bernstein's website (<http://cr.yp.to>). Our code for Algorithm

**Table 1.** Sieve Algorithm Comparison

<i>Algorithm</i>	<i>Time in Seconds</i>	$\Delta$
Atkin-Bernstein	7.2	—
Eratosthenes	5.9	—
PSSPS	58.1	25000
PSSPS	103.5	10000
PSSPS	183.0	5000
PSSPS	367.2	2500

**Table 2.** Finding all primes between  $n$  and  $n + 10^6$

$n$	<i>Rem.</i>	$Time_s$	$s$	<i>Primes</i>	$Time_p$	$p$	$L_p$	$Time_{tot}$
$10^{15}$	28845	1.25	31622776	28845	0	0	0	1.25
$10^{16}$	28774	2.17	57063204	27168	17.51	67	$1.75 \cdot 10^8$	19.68
$10^{17}$	28286	4.19	111269821	25463	19.49	79	$8.98 \cdot 10^8$	23.68
$10^{18}$	31717	2.09	42343580	24280	24.55	103	$2.36 \cdot 10^{10}$	26.64
$10^{19}$	31628	2.48	50951495	23069	27.36	113	$1.96 \cdot 10^{11}$	29.84
$10^{20}$	27342	23.29	509514950	21632	39.73	113	$1.96 \cdot 10^{11}$	63.02
$10^{21}$	28668	15.95	348208470	20832	44.59	131	$2.87 \cdot 10^{12}$	60.54
$10^{22}$	31814	2.7	55885834	19757	55.63	173	$1.78 \cdot 10^{14}$	58.33
$10^{23}$	30253	6.64	143644910	18939	55.84	181	$6.96 \cdot 10^{14}$	62.48
$10^{24}$	30879	4.06	85900327	18149	63.90	211	$1.16 \cdot 10^{16}$	67.96
$10^{25}$	27748	39.06	859003269	17549	63.43	211	$1.16 \cdot 10^{16}$	102.49
$10^{26}$	29965	6.54	140326390	16587	67.00	233	$7.12 \cdot 10^{17}$	73.54
$10^{27}$	30512	5.18	98057476	16139	74.55	263	$1.01 \cdot 10^{19}$	79.73
$10^{28}$	29863	8.14	143167432	15606	80.97	277	$6.98 \cdot 10^{19}$	89.11
$10^{29}$	30368	6.11	106761861	15002	107.82	293	$9.36 \cdot 10^{20}$	113.93
$10^{30}$	30944	4.24	73264612	14496	117.25	331	$1.36 \cdot 10^{22}$	121.49
$10^{31}$	30616	5.74	100509639	13955	116.26	347	$9.94 \cdot 10^{22}$	122.00
$10^{32}$	28542	18.95	338566228	13653	122.46	353	$2.95 \cdot 10^{23}$	141.41
$10^{33}$	26244	121.22	3385662272	13284	124.56	353	$2.95 \cdot 10^{23}$	245.78

PSSPS was not optimized for single-precision use; it used functions from the GnuMP package for arithmetic, and in particular, to perform modular exponentiations for the pseudosquares prime tests.

In Table 1, for each sieve we give the time to find the primes to  $10^9$  in seconds. For our new algorithm, we also show different times for various choices for  $\Delta$ , the size of our interval. In every case, Algorithm PSSPS sieved up to  $s = 31622$  and used  $p = 0$ , the largest entry from the pseudosquares table used for prime tests (a value of 0 indicates no such tests were performed).

Our goal here was to verify our results and to see how bad the  $\log n \log \log n$  factor in the running time affects Algorithm PSSPS. When we used  $\Delta = 500$ , we were able to force the algorithm to use  $p = 17$ , but the running time became quite

large. Simply put, our algorithm is not appropriate for inputs this small; it ends up performing what is, essentially, the sieve of Eratosthenes in a non-efficient way.

Next, in Table 2 we show how our sieve performs when finding all primes in an interval of length  $10^6$  for much larger values for  $n$ . The first column gives  $n$ , the starting point of the interval searched for primes. The next three columns report the performance of the sieving stage, giving the number of integers that are free of factors below  $s$  (the remainder),  $Time_s$ , the time sieving took in seconds, and the value of  $s$  used. The next four columns present the results from the pseudosquares prime tests, with the number of primes found first, followed by the time in seconds ( $Time_p$ ), the value of  $p$ , and the approximate value of  $L_p$  used by the prime test. The last column gives the total time,  $Time_{tot}$  (sieving plus prime tests). The number of tests performed (beginning with a base-2 psp test) matches the number in column 2 (the remainder), with the *Primes* column giving the number of integers that pass the test. Note that  $s \cdot L_p$  should match or exceed  $n + 10^6$ . When  $p = 0$ , sieving only was used, in which case  $s \geq \lfloor \sqrt{n + 10^6} \rfloor$  must hold.

Since  $L_{353}$  is currently the largest pseudosquare known, any increase in size beyond 33 decimal digits would have to be absorbed entirely by using a larger value for  $s$ , which will greatly degrade performance unless a correspondingly longer interval is used.

Notice that the pseudosquares prime test was not even used until the input was 16 digits in length.

## References

1. Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. [http://www.cse.iitk.ac.in/news/primalty\\_v3.pdf](http://www.cse.iitk.ac.in/news/primalty_v3.pdf), 2003.
2. A. O. L. Atkin and D. J. Bernstein. Prime sieves using binary quadratic forms. *Mathematics of Computation*, 73:1023–1030, 2004.
3. E. Bach. *Analytic Methods in the Analysis and Design of Number-Theoretic Algorithms*. MIT Press, Cambridge, 1985.
4. Eric Bach and Lorenz Huelsbergen. Statistical evidence for small generating sets. *Math. Comp.*, 61(203):69–82, 1993.
5. Eric Bach and Jeffrey O. Shallit. *Algorithmic Number Theory*, volume 1. MIT Press, 1996.
6. Eric Bach and Jonathan P. Sorenson. Sieve algorithms for perfect power testing. *Algorithmica*, 9(4):313–328, 1993.
7. Daniel J. Bernstein. Detecting perfect powers in essentially linear time. *Math. Comp.*, 67(223):1253–1283, 1998.
8. Daniel J. Bernstein. Proving primality in essentially quartic time. To appear in *Mathematics of Computation*; <http://cr.yp.to/papers.html#quartic>, 2006.
9. Daniel J. Bernstein, Hendrik W. Lenstra Jr., and Jonathan Pila. Detecting perfect powers by factoring into coprimes. To appear in *Mathematics of Computation*, 2006.
10. Brian Dunten, Julie Jones, and Jonathan P. Sorenson. A space-efficient fast prime number sieve. *Information Processing Letters*, 59:79–84, 1996.

11. William F. Galway. Dissecting a sieve to cut its need for space. In *Algorithmic number theory (Leiden, 2000)*, volume 1838 of *Lecture Notes in Comput. Sci.*, pages 297–312. Springer, Berlin, 2000.
12. William F. Galway. *Analytic Computation of the Prime-Counting Function*. PhD thesis, University of Illinois at Urbana-Champaign, 2004. Available at [http://www.math.uiuc.edu/~galway/PhD\\_Thesis/](http://www.math.uiuc.edu/~galway/PhD_Thesis/).
13. T. Granlund. The Gnu multiple precision arithmetic library, edition 4.1.3. <http://www.swox.se/gmp/manual>, 2004.
14. G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers*. Oxford University Press, 5th edition, 1979.
15. R. F. Lukes, C. D. Patterson, and H. C. Williams. Some results on pseudosquares. *Math. Comp.*, 65(213):361–372, S25–S27, 1996.
16. G. Miller. Riemann’s hypothesis and tests for primality. *Journal of Computer and System Sciences*, 13:300–317, 1976.
17. Carl Pomerance, J. L. Selfridge, and Samuel S. Wagstaff, Jr. The pseudoprimes to  $25 \cdot 10^9$ . *Math. Comp.*, 35(151):1003–1026, 1980.
18. P. Pritchard. A sublinear additive sieve for finding prime numbers. *Communications of the ACM*, 24(1):18–23, 772, 1981.
19. P. Pritchard. Fast compact prime number sieves (among others). *Journal of Algorithms*, 4:332–344, 1983.
20. M. O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12:128–138, 1980.
21. A. Schinzel. On pseudosquares. *New Trends in Prob. and Stat.*, 4:213–220, 1997.
22. R. Solovay and V. Strassen. A fast Monte Carlo test for primality. *SIAM Journal on Computing*, 6:84–85, 1977. Erratum in vol. 7, p. 118, 1978.
23. Jonathan P. Sorenson. Trading time for space in prime number sieves. In Joe Buhler, editor, *Proceedings of the Third International Algorithmic Number Theory Symposium (ANTS III)*, pages 179–195, Portland, Oregon, 1998. LNCS 1423.
24. Jonathan P. Sorenson and Ian Parberry. Two fast parallel prime number sieves. *Information and Computation*, 144(1):115–130, 1994.
25. B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2nd edition, 1991.
26. Hugh C. Williams. *Édouard Lucas and primality testing*. Canadian Mathematical Society Series of Monographs and Advanced Texts, 22. John Wiley & Sons Inc., New York, 1998. A Wiley-Interscience Publication.
27. Kjell Wooding. Development of a high-speed hybrid sieve architecture. Master’s thesis, The University of Calgary, Calgary, Alberta, November 2003.
28. Kjell Wooding and H. C. Williams. Doubly-focused enumeration of pseudosquares and pseudocubes. In *Proceedings of the 7th International Algorithmic Number Theory Symposium (ANTS VII)*, Berlin, Germany, 2006.