2006

The Pseudosquares Prime Sieve

Jonathan P. Sorenson
Butler University, jsorenso@butler.edu

Recommended Citation

J. Sorenson, The Pseudosquares Prime Sieve, Proceedings of the 7th International Symposium on Algorithmic Number Theory (ANTS-VII), Florian Hess, Sebastian Pauli, and Michael Pohst eds., Berlin, Germany, pages 193-207, 2006. LNCS 4076, ISBN 3-540-36075-1.

The Pseudosquares Prime Sieve

Jonathan P. Sorenson*

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

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 $p \leq n^{1/(4\sqrt{e}-\epsilon)}$, implying a running time of roughly $n^{1.132}$ using roughly $n^{0.132}$ space.

Existing prime sieves are generally faster ($O(n/\log\log n)$ operations) but take at least $n^{1/3}$ space, greatly limiting their range. 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 $O(n\exp[-c\log n/\log\log n])$ operations and $O((\log n)^2)$ space (under the ERH), where $c > 0$ is constant. 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, 17, 22], 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, 18, 22] cannot, on current hardware, generate primes larger than about 10^{18}. 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}. (Note that the space needed to write down the output, the primes up to n, is not included.)

* This work was supported by a grant from the Holcomb Research Institute

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 [16] 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 [15], 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 [19] or Solovay-Strassen [21] 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 [14] (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. Our sieve 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 some fixed $c > 0$. This data structure may prove to be useful in other areas of computational number theory.

For recent advances on prime number sieves, see [2, 11, 22].

The rest of this paper is organized as follows. In §2 we discuss some preliminaries, 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 [24]. 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 [12]). 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 Number Theory

p always denotes a prime, with p_i denoting the ith 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 there exists an integer r such that $r^2 \equiv x \pmod{p}$. We normally require that $\gcd(x, p) = 1$.

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 [25, §16.2].

We make use of the following estimates. Here $x, x_1, x_2 > 0$, and except for (5), all sums and products are only over primes.

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

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

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

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

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

For proofs of (1)–(4), see Hardy and Wright [13]. For a proof of (5), see [22, Lemma 1].

2.3 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. We begin with the following definitions:

$$\begin{aligned} M_k &:= \prod_{i=1}^{k} p_i; \\ W_k(y) &:= \{x \le y : \gcd(x, M_k) = 1\}; \\ W_k &:= W_k(M_k). \end{aligned}$$

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

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

Our data structure, then, is an array `W[]` of records or structs, indexed by $0 \ldots (M_k - 1)$, 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 kth 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 [18, 23].

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

In practice, we can use the table from Wooding [26, 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 Δ, 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 $\Delta \gg p$.

– $s := \lfloor n/L_p \rfloor + 1 = O(\Delta \log n)$.

We conjecture $p \sim (1/\log 2) \log n \log\log n$ (see below). Making Δ 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 Δ first. One normally chooses Δ 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 Δ.

```
int p,s;
INT l,r,n;                  // declare multiprec. ints
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 inverval [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. 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 (ℓ and r), of length Δ, that supports functions to set and clear bits. Also, the member function `first(`x`)` will return the first integer larger than ℓ 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: work with $x - \ell$ rather than x.

The time to sieve by primes is proportional to

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

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

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

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 the following lemma, due to Lukes, Patterson, and Williams [14].

Lemma 3.1. *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 $n < L_p$.

We code this prime test as function `pssp t()`, 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 [16].

```
INT x;
for(x=l+1; x<=r; x++) // loop over the interval
  if(B[x]==1)         // x meets conditions (1) & (2)
    if(!pssp t(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. Function `pssp t()` 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 [16] 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 `pssp t()` 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 [25, 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/Δ, the number of segments, we prove the following.

Theorem 3.2. *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 3.3. $\log L_p \sim \log 2 \frac{p}{\log p}$, or equivalently, $p \sim \frac{1}{\log 2} \log L_p \log\log L_p$.

Lukes, Patterson, and Williams [14] studied the relationship between L_p and p for all known pseudosquares, and their data supports the conjecture.

Corollary 3.4. *If Conjecture 3.3 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 3.3 appears to hold.

Corollary 3.5. *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 [15] or AKS [1, 8] in a prime sieve.

Currently the best unconditional result is $p \leq {L_p}^{1/(4\sqrt{e}-\epsilon)} \approx {L_p}^{0.1516\ldots}$, due to Schinzel [20]. 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 3.6. *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 [22] 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 an 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 §2.3, 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 $O(\log M_k \sum_{i=1}^{k} p_i) = O((\log M_k)^3 / \log\log M_k)$ 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 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 primes up to 100. In fact the datastructure will 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 $0 \ldots p_i - 1$, each of which has 3 fields. Let m_i be the *input modulus*, which is $m_i := 2 \cdot 3 \cdot \cdots \cdot p_{i-1}$. For our example, $m_2 = 2, m_3 = 6$, and $m_4 = 30$. Here $0 \leq x < p_i$.

- `W[`i`][`x`].rp` is 1 if $\gcd(x, p_i) = 1$, 0 otherwise (`int`),
- `W[`i`][`x`].dist` is the smallest integer $d > 0$ such that $\gcd(x + d, p_i) = 1$ (`int`), and
- `W[`i`][`x`].jump` is the smallest multiple $j > 0$ of the input modulus m_i such that $\gcd(x + j, p_i) = 1$ (`INT`).

This gives us the following three datastructures:

$p_2 = 3$:	0	1	2
rp	0	1	1
dist	1	1	2
jump	2	4	2

($m_2 = 2$, $\phi(3) = 2$)

$p_3 = 5$:	0	1	2	3	4
rp	0	1	1	1	1
dist	1	1	1	1	2
jump	6	6	6	6	12

($m_3 = 6$, $\phi(5) = 4$)

$p_4 = 7$:	0	1	2	3	4	5	6
rp	0	1	1	1	1	1	1
dist	1	1	1	1	1	1	2
jump	30	30	30	30	30	60	30

($m_4 = 30$, $\phi(7) = 6$)

We will explain how to compute the `jump` fields below.

To enumerate the primes (and 1) we use the following code fragment:

```
//** Loop for p[2]=3 (loops 2 times):
x2=1; // we want ==1 mod 2
if(!W[2][x2%p[2]].rp) // make sure gcd(x2,p[2])=1
  x2=x2+W[2][x2%p[2]].jump;
for(cnt2=0; cnt2<phi(p[2]); cnt2++)
{ //** Loop for p[3]=5 (loops 4 times):
  x3=x2;
  if(!W[3][x3%p[3]].rp) // make sure gcd(x3,p[3])=1
    x3=x3+W[3][x3%p[3]].jump;
  for(cnt3=0; cnt3<phi(p[3]); cnt3++)
  { //** Loop for p[4]=7 (loops ? times):
    x4=x3;
    if(!W[4][x4%p[4]].rp) // make sure gcd(x4,p[4])=1
      x4=x4+W[4][x4%p[4]].jump;
    for( ; x4<100; x4=x4+W[4][x4%p[4]].jump)  output(x4);
    x3=x3+W[3][x3%p[3]].jump;
  }
  x2=x2+W[2][x2%p[2]].jump;
}
```

Here `x2` will take the values 1 and 5.

When `x2` is 1, `x3` loops through 1, 7, 13, and 19. When `x2` is 5, `x3` loops through 11, 17, 23, and 29. (`x2` $= 5$ is not relatively prime to $p_3 = 5$, so the if-statement before the loop is triggered, adding 6 to get 11.)

The values `x4` loops through are listed in the table below, giving the primes from 11 to 100, plus 1.

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 `x2` $= 1$; they are $\equiv 1 \bmod 6$. The values to the right of the line arise from `x2` $= 5$; they are $\equiv 5 \bmod 6$.

To enumerate integers relatively prime to $210 = 2 \cdot 3 \cdot 5 \cdot 7$ up to n, simply replace the 100 bound for `x4` in the innermost loop with n.

Computing `jumps`. Computing the `rp` and `dist` fields 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 $x = 0 \ldots p_i - 1$, 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 $k \cdot p_k \log M_k = O(p_k^3 / \log p_k)$.

Using Recursion. The code fragment above to enumerate integers relatively prime to M_k requires $k-1$ nested loops in general. This is not practical to code, so we rely instead on recursion.

Here let k denote the number of prime moduli in the wheel; recall that 2 is not given a datastructure, so there will be $k-1$ levels to the recursion. To enumerate integers relatively prime to M_k up to n, we call `enumerate(2,1,`n`)`.

```
function enumerate(int i,INT x,INT n)
{
  //** make sure gcd(x,p[i])=1
  if(!W[i][x%p[i]].rp)  x=x+W[i][x%p[i]].jump;

  if(i==k) // base case for the recursion
    for( ; x<n; x=x+W[k][x%p[k]].jump)  output(x);
  else // recursive case for the recursion
  {
    for(int cnt=0; cnt<phi(p[i]); cnt++)
    {
      sieve(i+1,x,n); // recursive call
      x=x+W[i][x%p[i]].jump;
    }
  }
}
```

Note that we are assuming pass-by-value here, so that changes to x in recursive calls are not reflected in the calling function.

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

1. We choose k so that $M_k \le n$, but as large as possible. We assume that all pseudosquares L_{p_i} with $p_i \le p_k$ 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 $p \ge 3$. 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 $1 \ldots p_i - 1$ modulo p_i and mark the corresponding `qr` field with a 1.
4. Compute the `dist` and `jump` fields from the `qr` field as if it were the `rp` field.
5. Replace $\phi(p_i)$ 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 $p_k < p_i \le p$. 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 $L_p \le n$ in $O(n 2^{-k} / \log p_k)$ 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 $L_p \le n$ 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 3.3 implies only $O((\log n)^3 / \log\log n)$ space is needed; assuming the ERH instead does not increase this bound.

We apply this theorem in our prime sieve using $p_k \approx (\log n)^{2/3}$ 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 $n = p^2 q$, 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. 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

Table 1. Sieve Algorithm Comparison

Algorithm	*Time in Seconds*	Δ	p	s
Atkin-Bernstein	7.2	—	—	—
Eratosthenes	5.9	—	—	—
PSSPS	58.1	25000	0	31622
PSSPS	103.5	10000	0	31622
PSSPS	183.0	5000	0	31622
PSSPS	367.2	2500	0	31622

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.

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 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 Δ, the size of our interval, p, the largest entry from the pseudosquares table used for prime tests (a value of 0 indicates no such tests were performed), and the sieve limit s.

Next, we show how our sieve performs when finding all primes in an interval of length 10^6 for much larger values for n (Table 2). 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), the time sieving took in seconds, and the value of s used. The last four columns present the results from the pseudosquares prime tests, with the number of primes found first, followed by the time in seconds, the value of p, and an estimate of L_p used by the prime test. 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

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

n	*Rem.*	*Time*	s	*Primes*	*Time*	p	L_p
10^{10}	43427	0.11	100004	43427	0	0	0
10^{11}	39434	0.13	316229	39434	0	0	0
10^{12}	36249	0.17	1000000	36249	0	0	0
10^{13}	33456	0.25	3162277	33456	0	0	0
10^{14}	30892	0.49	10000000	30892	0	0	0
10^{15}	28845	1.25	31622776	28845	0	0	0
10^{16}	28774	2.17	57063204	27168	17.51	67	$1.75{\cdot}10^{8}$
10^{17}	28286	4.19	111269821	25463	19.49	79	$8.98{\cdot}10^{8}$
10^{18}	31717	2.09	42343580	24280	24.55	103	$2.36{\cdot}10^{10}$
10^{19}	31628	2.48	50951495	23069	27.36	113	$1.96{\cdot}10^{11}$
10^{20}	27342	23.29	509514950	21632	39.73	113	$1.96{\cdot}10^{11}$
10^{21}	28668	15.95	348208470	20832	44.59	131	$2.87{\cdot}10^{12}$
10^{22}	31814	2.7	55885834	19757	55.63	173	$1.78{\cdot}10^{14}$
10^{23}	30253	6.64	143644910	18939	55.84	181	$6.96{\cdot}10^{14}$
10^{24}	30879	4.06	85900327	18149	63.90	211	$1.16{\cdot}10^{16}$
10^{25}	27748	39.06	859003269	17549	63.43	211	$1.16{\cdot}10^{16}$
10^{26}	29965	6.54	140326390	16587	67.00	233	$7.12{\cdot}10^{17}$
10^{27}	30512	5.18	98057476	16139	74.55	263	$1.01{\cdot}10^{19}$
10^{28}	29863	8.14	143167432	15606	80.97	277	$6.98{\cdot}10^{19}$
10^{29}	30368	6.11	106761861	15002	107.82	293	$9.36{\cdot}10^{20}$
10^{30}	30944	4.24	73264612	14496	117.25	331	$1.36{\cdot}10^{22}$
10^{31}	30616	5.74	100509639	13955	116.26	347	$9.94{\cdot}10^{22}$
10^{32}	28542	18.95	338566228	13653	122.46	353	$2.95{\cdot}10^{23}$
10^{33}	26244	121.22	3385662272	13284	124.56	353	$2.95{\cdot}10^{23}$

$n + 10^6$. 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/primality_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. T. Granlund. The Gnu multiple precision arithmetic library, edition 4.1.3. http://www.swox.se/gmp/manual, 2004.
13. G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers*. Oxford University Press, 5th edition, 1979.
14. R. F. Lukes, C. D. Patterson, and H. C. Williams. Some results on pseudosquares. *Math. Comp.*, 65(213):361–372, S25–S27, 1996.
15. G. Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences*, 13:300–317, 1976.
16. Carl Pomerance, J. L. Selfridge, and Samuel S. Wagstaff, Jr. The pseudoprimes to $25 \cdot 10^9$. *Math. Comp.*, 35(151):1003–1026, 1980.
17. P. Pritchard. A sublinear additive sieve for finding prime numbers. *Communications of the ACM*, 24(1):18–23,772, 1981.
18. P. Pritchard. Fast compact prime number sieves (among others). *Journal of Algorithms*, 4:332–344, 1983.
19. M. O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12:128–138, 1980.
20. A. Schinzel. On pseudosquares. *New Trends in Prob. and Stat.*, 4:213–220, 1997.
21. 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.
22. 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.
23. Jonathan P. Sorenson and Ian Parberry. Two fast parallel prime number sieves. *Information and Computation*, 144(1):115–130, 1994.
24. B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2nd edition, 1991.
25. 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.
26. Kjell Wooding. Development of a high-speed hybrid sieve architecture. Master's thesis, The University of Calgary, Calgary, Alberta, November 2003.