AN ANALYSIS OF TWO PRIME NUMBER SIEVES

by

Jonathan Sorenson

Computer Sciences Technical Report #1028

June 1991

An Analysis of Two Prime Number Sieves

Jonathan Sorenson*

Computer Sciences Department
University of Wisconsin-Madison
1210 West Dayton Street
Madison, WI 53706
sorenson@cs.wisc.edu

June 10, 1991

Abstract

A prime number sieve is an algorithm that finds all prime numbers up to a bound n. In this paper, we calculate the number of arithmetic operations used by two prime number sieves: the sieve of Eratosthenes and an efficient linear sieve due to Pritchard. Although Pritchard's sieve is asymptotically faster than the sieve of Eratosthenes by a factor of $\Theta(\log \log n)$, we show that the sieve of Eratosthenes uses fewer arithmetic operations than Pritchard's sieve for most feasible inputs. In addition, we analyze and compare versions of both algorithms that use wheels, and we give implementation timing results.

*Sponsored by NSF grant CCR-8552596

1 Introduction

In this paper, we analyze two algorithms that find all prime numbers up to a bound n. Such algorithms are often called prime number sieves.

Perhaps the most famous prime number sieve is attributed to Eratosthenes, the ancient librarian of Alexandria (see [DRK26]). His algorithm finds all the primes up to n using $\Theta(n \log\log n)$ arithmetic operations. Here an arithmetic operation is one addition, subtraction, multiplication, division, or comparison of $O(\log n)$-bit integers. We will refer to the sieve of Eratosthenes as Algorithm E.

In the last two decades, researchers have discovered many new prime number sieves that require only $\Theta(n)$ arithmetic operations. This includes algorithms by Mairson [Mai77], Gries and Misra [GM78], Barstow [Bar79], Misra [Mis81], Pritchard [Pri81, Pri82, Pri87], and Bengelloun [Ben86]. For a survey and categorization of these linear prime number sieves, see Pritchard [Pri87].

Despite the fact that these new algorithms use fewer arithmetic operations than the sieve of Eratosthenes by a factor of $\Theta(\log\log n)$, many researchers believe that the sieve of Eratosthenes is still the best algorithm to use in practice. Our goal in this paper is to give a rigorous basis for this belief. To be more specific, we compare the number of arithmetic operations used by Algorithm E to the number of operations used by a linear sieve do to Pritchard [Pri87, Algorithm 3.3, page 23]. This linear sieve is perhaps the asymptotically most efficient known linear prime number sieve. We will refer to this sieve as Algorithm P.

In section 3, we show that Algorithm E finds all prime numbers up to n using

$$2n \log\log n + cn + O(n/\log n) \tag{1}$$

arithmetic operations, where $c \approx 1.14$. (Here $\log x$ denotes the natural logarithm of x.) We show that the number of arithmetic operations used by Algorithm P is

$$8.5\,n + O(n/\log n). \tag{2}$$

Suppose $n \leq 5 \times 10^8$; then $\log\log n$ is at most 3, and so Algorithm E uses roughly $7.14\,n$ arithmetic operations as compared to $8.5\,n$ for Algorithm P. This suggests that Algorithm E is faster when $n \leq 5 \times 10^8$. In fact, we will prove that Algorithm E uses fewer arithmetic operations than Algorithm P when $25 \leq n \leq 7 \times 10^{14}$. This range covers most practical values of n; on a 32-bit computer, if n is to fit in one machine word then n can be at most $2^{32} < 4.3 \times 10^9$.

Both algorithms may be improved significantly using *wheels*. In section 4, we describe how this is done and analyze the resulting algorithms. We then put our theory to the test by giving timing results for implementations of several variants of both algorithms in section 5. We conclude with a some remarks in section 6.

We have chosen to gauge the performance of prime number sieves by counting how many arithmetic operations are performed, and there are two reasons for this. First, a "big-O" estimate of the running times of these algorithms is not sufficient for our purposes. Therefore, counting bit operations would force us to establish the precise algorithms used for the basic arithmetic operations. Though this is feasible, we prefer to avoid such detail. Second, we could have chosen to count other operations like indirect addressing and assignment. But for

prime number sieves, the number of arithmetic operations performed is a good predictor of actual performance. In addition, our results could be easily modified to incorporate counts of other operations; we leave such modifications to the reader.

Let us now review some results on the distribution of prime numbers.

2 Some Results on the Distribution of Primes

Let $\pi(x)$ denote the number of primes up to x. The *Prime Number Theorem* states that $\pi(x) \sim x/\log x$. Let $Li(x) = \int_2^x (1/\log t)dt$, which has the following asymptotic expansion:

$$Li(x) \quad = \quad \frac{x}{\log x} + \frac{1!\,x}{(\log x)^2} + \cdots + \frac{k!\,x}{(\log x)^{k+1}}(1 + o(1)).$$

Then a more precise version of the prime number theorem states that $\pi(x) \sim Li(x)$. Let us define $E(x)$ to be the error in this approximation:

$$E(x) \quad = \quad \max_{t \le x} |\pi(t) - Li(t)|.$$

Vinogradov showed there exists a constant $c > 0$ such that $E(x) = O(x \exp[-c(\log x)^{3/5}])$. In addition, if the Riemann Hypothesis is assumed, then $E(x) = O(\sqrt{x} \log x)$. It is also known that $E(x) > \sqrt{x}/\log x$ infinitely often (see Ingham [Ing32, p. 103]). The following explicit estimate was given by Rosser and Schoenfeld [RS62]:

$$\frac{x}{\log x} \quad < \quad \pi(x) \quad < \quad 1.26\frac{x}{\log x} \qquad \text{for } x \ge 17. \tag{3}$$

The following estimate for the sum of the logarithms of primes up to x is equivalent to the prime number theorem:

$$\sum_{p \le x} \log p \quad = \quad x + O(E(x) \log x). \tag{4}$$

Sums over the variable p will always denote sums over primes. For more on the prime number theorem, see Davenport [Dav80, Chapters 7,18] and Ingham [Ing32].

Other useful estimates include the following, the second of which is Mertens's Theorem:

$$\sum_{p \le x} \frac{1}{p} \quad = \quad \log\log x + B + O(E(x)/x); \tag{5}$$

$$\prod_{p \le x} \left(1 - \frac{1}{p}\right) \quad = \quad \frac{e^{-\gamma}}{\log x}(1 + O(E(x)/x)). \tag{6}$$

Here B is $0.2614972128\cdots$ and Euler's constant γ is $0.5772156649\cdots$. The following is an explicit version of equation (5):

$$\sum_{p \le x} \frac{1}{p} \quad \le \quad \log\log x + B + \frac{1}{\log^2 x} \qquad \text{for } x > 1. \tag{7}$$

See Rosser and Schoenfeld [RS62] and Schoenfeld [Sch76].

The following lemma gives an estimate for sums of powers of primes. The error term is an improvement over one given by Salat and Znam [SZ68].

Lemma 2.1 *If $\alpha \geq 0$ then $\sum_{p \leq x} p^\alpha = Li(x^{\alpha+1}) + O(x^\alpha E(x))$.*

Proof: For $\alpha = 0$, this is the prime number theorem.

For $\alpha > 0$, we use integration by parts twice and the prime number theorem to get

$$\begin{aligned}\sum_{p \leq x} p^\alpha &= \int_2^x t^\alpha d\pi(t) = x^\alpha \pi(x) - \int_2^x \pi(t) d(t^\alpha) \\ &= x^\alpha \pi(x) - \int_2^x Li(t) d(t^\alpha) + O\left(\int_2^x E(t) \alpha t^{\alpha-1} dt\right) \\ &= \int_2^x \frac{t^\alpha}{\log t} dt + O(x^\alpha E(x)).\end{aligned}$$

Substituting $w = t^{\alpha+1}$ then gives

$$\sum_{p \leq x} p^\alpha = \int_{2^{\alpha+1}}^{x^{\alpha+1}} \frac{dw}{\log w} + O(x^\alpha E(x)) = Li(x^{\alpha+1}) + O(x^\alpha E(x)).$$

The implied constant in the second term does not depend on α. □

By invoking the prime number theorem above, we have the following equation:

$$\sum_{p \leq x} p = \pi(x^2) + O(xE(x) + E(x^2)).$$

Interestingly, this says that $\sum_{p \leq x} p$ is a very good approximation to $\pi(x^2)$.

Let m, a, b be positive integers. Let $a \bmod b = a - \lfloor a/b \rfloor \cdot b$, the remainder when a is divided by b. We write $b \mid a$ (b divides a) when $a \bmod b = 0$. We say that a and b are *relatively prime* when, for every integer $d > 0$, if $d \mid a$ and $d \mid b$ then $d = 1$. Euler's totient function $\phi(m)$ gives the number of integers up to m that are relatively prime to m. So if p is prime then $\phi(p) = p - 1$.

3 The Basic Algorithms

In this section we describe and analyze the basic versions of the sieve of Eratosthenes (Algorithm E) and Pritchard's linear sieve (Algorithm P). Then we prove that the sieve of Eratosthenes uses fewer arithmetic operations than Pritchard's sieve for practical inputs.

3.1 The Sieve of Eratosthenes

If a positive integer x is not prime, then it has at least two prime divisors, and one of these divisors can be no more than $\sqrt{x}$. Using this fact, the sieve of Eratosthenes finds all the primes up to n in two basic steps:

A. A set S is initialized to contain the integers from 2 to n.

B. For each prime p up to $\sqrt{n}$, all multiples of p larger than p itself are removed from S.

By the fact mentioned above, if an integer x is composite, it will be removed from the set S in step B. If x is prime, it is placed in the set S in step A but not removed. Thus S contains precisely the primes up to n.

We can improve step B by noticing that, for each prime p, the multiples of p between p and p^2 must also be multiples of smaller primes. This means the smallest multiple of p we need to remove from the set S is p^2.

If we represent the set S as an array $s[\,]$ of bits in the obvious way, this leads us to the following algorithm.

Algorithm E

Input: a positive integer n

Initialization

1. $s[1] := 0$;
2. For $x := 2$ to n do: $s[x] := 1$;

Main Loop

1. For $p := 2$ to $\lfloor\sqrt{n}\rfloor$ do:
2. If $s[p] = 1$ then { p is prime }
3. For $x := p^2$ to n step p do: $s[x] := 0$;

Theorem 3.1 *Let n be the input to Algorithm E. Then Algorithm E computes the set of primes up to n using*

$$2n \log\log n + (2B - 2\log 2 + 2)n - 2Li(n) + O(n^{1/2}E(n^{1/2}))$$

arithmetic operations. (*Here* $2B - 2\log 2 + 2 \approx 1.136700$.)

Proof: We leave the proof of correctness to the reader.

Initialization

Step 1 uses no arithmetic operations. Each iteration of a For-loop requires 2 arithmetic operations: incrementing the index variable, and comparing the index variable with the upper limit. Thus step 2 uses $2(n-1) = 2n + O(1)$ arithmetic operations.

Main Loop

The integer square root of n can be found with binary search in $O(\log n)$ iterations, each using $O(1)$ arithmetic operations. Thus steps 1 and 2 use a total of $O(\sqrt{n})$ arithmetic operations.

Step 3 executes once for each prime $p \le \sqrt{n}$, and for the prime p it iterates $\lfloor (n-p^2)/p \rfloor$ times. After counting the operation required to compute $x = p^2$, this gives the following total number of arithmetic operations for step 3:

$$2\sum_{p\le\sqrt{n}} \left\lfloor \frac{n-p^2}{p} \right\rfloor + \pi(\sqrt{n}) \;=\; 2n\sum_{p\le\sqrt{n}} \frac{1}{p} - 2\sum_{p\le\sqrt{n}} p + O(\pi(\sqrt{n})).$$

We apply (5) and Lemma 2.1 from section 2 to give

$$2n\left(\log\log\sqrt{n} + B + O(E(n^{1/2})/n^{1/2})\right) - 2\left(Li(n) + O(n^{1/2}E(n^{1/2}))\right) + O(\pi(\sqrt{n})).$$

Simplifying gives us a total of

$$2n\log\log n + (2B - 2\log 2)n - 2Li(n) + O(n^{1/2}E(n^{1/2}))$$

arithmetic operations for step 3.

We combine the operation counts for all steps to complete the proof. □

Notice that if we assume the Riemann Hypothesis, the error term in Theorem 3.1 is only $O(n^{3/4}\log n)$.

We implemented Algorithm E and counted the actual number of arithmetic operations used. The table below compares this actual count to the number of operations predicted by Theorem 3.1.

Algorithm E: Arithmetic Operations

n	*Actual*	*Predicted*	*Error*
10^3	4921	4649	5.53%
10^6	6247259	6229374	0.29%

In computing the predicted number of arithmetic operations, we approximated $Li(x)$ with Simpson's rule and we omitted the $O(n^{1/2}E(n^{1/2}))$ error term.

3.2 Pritchard's Linear Sieve

Our second prime number sieve is an algorithm due to Pritchard. It begins as the sieve of Eratosthenes does by initializing a set S to contain the integers from 2 to n. But then it removes each composite integer from S exactly once. This is done by noticing that a composite integer x can be written uniquely in the form $x = p \cdot f$, where p is the least prime factor of x and $f = x/p$. We will write $lpf(x)$ to denote the least prime factor of x.

The function lpf has several useful properties: if x is composite, then $lpf(x) \leq \sqrt{x}$, and if $x = p \cdot f$ where $p = lpf(x)$, then $p \leq lpf(f)$.

The main idea behind Pritchard's algorithm, then, is to construct each composite integer x in the form $p \cdot f$, where $p = lpf(x)$ and $f = x/p$, and then remove x from the set S. This is done by looping through all possible values of f that can occur, and for each f value, looping through all possible primes p such that $p \cdot f$ gives the unique representation we mentioned above for *some* x. This leads to the following algorithm outline.

A. A set S is initialized to contain the integers from 2 to n.

B. For each value of f from 2 to $n/2$, for each prime p such that $p \leq lpf(f)$ and $pf \leq n$, remove pf from the set S.

In step B we can avoid factoring f to determine $lpf(f)$ by noticing that $p = lpf(f)$ if and only if $p \mid f$ and for every prime $q < p$, $q \nmid f$. Also notice that the primes needed in step B are bounded by $\sqrt{n}$.

Implementing the set S as an array of bits in the obvious way leads us to the following algorithm.

Algorithm P

Input: a positive integer n

Initialization

1. Find primes up to $\lfloor\sqrt{n}\rfloor$;
2. Initialize the array $s[\,]$; (see Algorithm E)

Main Loop

1. For $f := 2$ to $\lfloor n/2 \rfloor$ do:
2. $\quad q := 2$; $x := qf$;
3. $\quad$ Repeat:
4. $\quad\quad p := q$; $s[x] := 0$;
5. $\quad\quad q :=$ nextprime(p); $x := qf$;
6. $\quad$ Until $x > n$ or $p \mid f$;

Here we are assuming that the primes used in the main loop are stored in an array, and the *nextprime* function merely returns the next prime in that array using one arithmetic operation.

This algorithm varies somewhat from the original algorithm given by Pritchard. For example, Pritchard pointed out the the primes needed in step B can be computed as part of the loop. However, finding them in the initialization step recursively is more efficient. For more details and Pritchard's version of this algorithm, see [Pri87].

Theorem 3.2 *Let n be the input to Algorithm P. Then Algorithm P computes the set of primes up to n using*

$$8.5n - 5Li(n) + O(\sqrt{n} + E(n))$$

arithmetic operations.

Proof: We leave the proof of correctness to the reader.

Initialization

Step 1 may be done recursively (details omitted) using $O(\sqrt{n})$ arithmetic operations, and step 2 uses $2n + O(1)$ operations for a total of $2n + O(\sqrt{n})$ for initialization.

Main Loop

Step 1 uses $n + O(1)$ arithmetic operations, and step 2 is executed $\lfloor n/2 \rfloor$ times using one operation each time for a total of $(3/2)n + O(1)$ for steps 1 and 2.

The loop in steps 3–6 iterates once for each composite integer up to n. Each iteration uses 5 arithmetic operations: 1 is implied in the *nextprime* function call in step 5, 1 for computing $x := qf$, 1 for testing if $x > n$ in step 6, and 2 for testing if $p \mid f$ (a division and a comparison). Since there are $n - \pi(n)$ composite integers up to n, by the prime number theorem we have a total of $5(n - \pi(n)) = 5n - 5Li(n) + O(E(n))$ arithmetic operations for steps 3–6.

Combining the totals for all steps completes the proof. □

As we did with Algorithm E, in the table below we compare the actual number of operations used by an implementation of Algorithm P to the number of operations predicted by Theorem 3.2.

Algorithm P: Arithmetic Operations

n	*Actual*	*Predicted*	*Error*
10^3	7990	7617	4.67%
10^6	8118801	8102725	0.20%

For the predicted counts in this table we estimated $Li(x)$ with Simpson's rule, and we omitted the error terms in Theorem 3.2.

3.3 A Comparison

Now we will prove that the sieve of Eratosthenes uses fewer arithmetic operations than Pritchard's sieve for most practical values of n.

Lemma 3.3 *On input n, the number of arithmetic operations used by Algorithm E is at most*

$$2n \log\log n + 2(B - \log 2 + 1)n + \frac{8n}{\log^2 n} + 4\sqrt{n} + r(n)$$

and the number of arithmetic operations used by Algorithm P is at least

$$8.5n - 6.3\frac{n}{\log n} + 2\sqrt{n} - 4 + r(n).$$

Here $r(n)$ is the number of operations used to compute the square root of n.

Proof:

Algorithm E:

Initialization uses at most $2n$ operations. In the main loop, steps 1 and 2 use at most $3\sqrt{n}$ operations. Step 3 uses at most $2\sum_{p\le\sqrt{n}} n/p + \sqrt{n}$ operations. Applying (7) completes the proof for Algorithm E.

Algorithm P:

Initialization uses at least $2n + 2\sqrt{n}$ operations. In the main loop, steps 1 and 2 use at least $(3/2)n - 4$ operations. Step 3–6 use at least $5(n - \pi(n))$ operations. Applying (3) completes the proof for Algorithm P. □

Theorem 3.4 *Let n be the input for Algorithms E and P. If $25 \le n \le 7 \times 10^{14}$, then Algorithm P uses more arithmetic operations than Algorithm E.*

Proof: For the input n, let $E_0(n)$ be the upper bound on the number of arithmetic operations used by Algorithm E, and let $P_0(n)$ be the lower bound on the number of arithmetic operations used by Algorithm P, as given by Lemma 3.3. Let $f(n) = P_0(n) - E_0(n)$. Then we have

$$\frac{f(n)}{n} \quad \ge \quad 7.363 - 2\log\log n - t(n)$$

where

$$t(n) = \frac{6.3}{\log n} + \frac{8}{\log^2 n} + \frac{2}{\sqrt{n}} + \frac{4}{n}.$$

It suffices to show that $f(n)/n > 0$ for $25 \leq n \leq 7 \times 10^{14}$.

On the interval $25 \leq n \leq 1500$, we have $t(n) \leq t(25) \leq 3.3$ and $\log\log n \leq \log\log(1500) \leq 2$. This implies that $f(n)/n \geq 7.363 - 2 \times 2 - 3.3 = 0.063 > 0$.

On the interval $1500 \leq n \leq 10^{10}$, we have $t(n) \leq t(1500) \leq 1.07$, and $\log\log n \leq \log\log(10^{10}) \leq 3.14$. Thus $f(n)/n \geq 7.363 - 2 \times 3.14 - 1.07 = 0.013 > 0$.

And finally on the interval $10^{10} \leq n \leq 7 \times 10^{14}$, $t(n) \leq 0.29$ and $\log\log n \leq 3.535$ which implies that $f(n)/n \geq 0.003 > 0$.

That completes the proof. □

The range of values of n for which we proved Theorem 3.4 above is not optimal. This range can be increased both by improving the bounds given in Lemma 3.3 and by using more subintervals combined with tighter bounds on $t(n)$ and $\log\log n$ on each subinterval in the proof of the theorem.

4 Algorithms Using Wheels

Many prime number sieves can be improved through the use of a *wheel*. In this section we will describe exactly what a wheel is and give an algorithm for constructing the kth wheel. Then we will show how to use wheels to improve the sieve of Eratosthenes and Pritchard's algorithm. Finally, we compare these algorithms and show that, for most practical values of n, the sieve of Eratosthenes with a wheel is more efficient than Pritchard's sieve using a wheel.

4.1 What is a Wheel?

Let p_i denote the ith prime, and let $M = M(k) = \prod_{i=1}^{k} p_i$. The kth wheel W is an array of M entries (indexed by $0 \ldots M-1$) with the following two properties:

1. If x is divisible by one of the first k primes, then $W[x \bmod M] = 0$.
2. If x is relatively prime to M (and hence not divisible by any of the first k primes) then $x + W[x \bmod M]$ gives the smallest integer larger than x that is relatively prime to M.

M is called the *size* of the kth wheel. Below we give as examples the first, second, and third wheels:

$k = 1$, $M = 2$:

x	0	1
$W[x]$	0	2

$k = 2$, $M = 2 \cdot 3 = 6$:

x	0	1	2	3	4	5
$W[x]$	0	4	0	0	0	2

$k = 3$, $M = 2 \cdot 3 \cdot 5 = 30$ (non-zero entries only)

x	1	7	11	13	17	19	23	29
$W[x]$	6	4	2	4	2	4	6	2

The major purpose of the kth wheel is to generate a list of integers relatively prime to For example, we can list the integers from 1 up to n that are relatively prime to M as follows:

1. For $x := 1$ to n step $W[x \bmod M]$ do:
2. Output(x);

Since each loop iteration involves a division, addition, and comparison, the number of arithmetic operations used by this loop, for large n, is $3n(\phi(M)/M) + O(1)$. If we do not care about the order of the integers listed, we can improve on this bound as follows:

1. For $r := 1$ to M step $W[r \bmod M]$ do:
2. For $x := r$ to n step M do:
3. Output(x);

Here, the number of operations is only $2n(\phi(M)/M) + O(\phi(M))$. We will make use of similar tricks when incorporating wheels into both algorithms.

Constructing wheels is relatively easy; Algorithm W below describes one way. The basic idea is to think of the array $W[]$ as representing a set. It is initialized to contain the integers from 0 to $M - 1$. Then all multiples of the first k primes are removed in a way similar to that used by the sieve of Eratosthenes. Finally, a pass is made over the array to reset each non-zero entry so that it gives the distance to the next non-zero entry.

Algorithm W

Input: $k > 0$, the first k primes, and $M =$ the product of the first k primes

1. For $x := 0$ to $M - 1$ do: $W[x] := 1$;
2. For the first k primes p do:
3. For $x := 0$ to $M - 1$ step p do: $W[x] := 0$;
4. $W[M - 1] := 2$; $prev := M - 1$;
5. For $x := M - 2$ to 1 step -1 do:
6. If $W[x] \neq 0$ then $W[x] := prev - x$; $prev := x$;

Below, we show that Algorithm W uses $O(M \log\log\log M)$ arithmetic operations to construct the kth wheel.

Lemma 4.1 *Let k, M and the first k primes $p_1 \ldots p_k$ be the inputs to Algorithm W. Then Algorithm W correctly computes the kth wheel $W[]$ using $O(M \log\log\log M)$ arithmetic operations.*

Proof: We leave the proof of correctness to the reader.

To prove the upper bound on the number of arithmetic operations, it is enough to show that each of the 6 steps uses $O(M \log\log\log M)$ operations.

Step 1 uses $2M + O(1) = O(M)$ operations, and step 2 uses $2k + O(1) = O(k) = O(M)$ operations.

Step 3 uses $O(M/p)$ operations for each prime p up to p_k. Summing over all k primes and using (5) from section 2 gives a total of $O(M \log\log p_k)$ operations. But

$$\log M \quad = \quad \log \prod_{i=1}^{k} p_i \quad = \quad \sum_{p \le p_k} \log p \quad \sim \quad p_k$$

by (4). Thus step 3 uses $O(M \log\log\log M)$ arithmetic operations.

Finally, steps 5 and 6 use $O(M)$ operations. □

Our description of wheels has been very concrete; hopefully this makes the concept easy to grasp. However, any data structure that uses the same basic idea is often called a wheel. Wheels have applications other than prime number sieves. For example, the divisors needed by a trial division algorithm can be generated efficiently using a wheel.

4.2 Using a Wheel with the Sieve of Eratosthenes

Recall that the sieve of Eratosthenes has two main steps: (A) the set S is initialized to be the integers from 2 to n, and (B) for each prime p up to $\sqrt{n}$, the multiples of p are removed from the set S. The basic idea for improving this algorithm is to use the kth wheel to "move" the execution of step B for the first k primes to step A. In other words, we have the following outline:

A. The set S is initialized to contain both the integers from 2 to n that are relatively prime to M and the first k primes.

B. For each prime $p > p_k$ up to $\sqrt{n}$, all multiples of p larger than p itself and relatively prime to M are removed from S.

One way to implement step A is as follows:

```
1.  Find the first k + 1 primes and store them in array pr[];
2.  Compute M = ∏_{i=1}^{k} pr[i] and the kth wheel W[];
3.  For x := 2 to n do:
4.      If W[x mod M] = 0 then s[x] := 0 else s[x] := 1;
5.  s[1] := 0;
6.  For i := 1 to k do: s[pr[i]] := 1;
```

For step B, we can generate the integers from p to n/p that are relatively prime to M using the kth wheel, and multiplying each by p gives the multiples of p we wish to remove.

```
1.  For p := pr[k + 1] to ⌊√n⌋ do:
2.      If s[p] = 1 then { p is prime }
3.          For f := p to ⌊n/p⌋ step W[f mod M] do: s[pf]:=0;
```

We can reduce the number of arithmetic operations for both steps using tricks similar to the one mentioned in the previous subsection. The details are in Algorithm E_k below.

Algorithm E_k

Input: positive integers n, k

Initialization

1. Find the first $k+1$ primes and store them in array $pr[\,]$;
2. Compute $M = \prod_{i=1}^{k} pr[i]$ and the kth wheel $W[\,]$;
3. For $r := 1$ to M do:
4. If $W[r \bmod M] = 0$ then $b := 0$ else $b := 1$;
5. For $x := r$ to n step M do: $s[x] := b$;
6. $s[1] := 0$;
7. For $i := 1$ to k do: $s[pr[i]] := 1$;

Main Loop

1. For $p := pr[k+1]$ to $\lfloor \sqrt{n} \rfloor$ do:
2. If $s[p] = 1$ then { p is prime }
3. For $f := p$ to $p + M - 1$ step $W[f \bmod M]$ do:
4. For $x := pf$ to n step Mp do: $s[x] := 0$;

Note that Algorithm E_0 is identical to Algorithm E. We have the following estimate for the number of arithmetic operations used by Algorithm E_k.

Theorem 4.2 *Let n, k be the inputs to Algorithm E_k, let M be the product of the first k primes, and let p_i be the ith prime. Then Algorithm E_k computes the set of primes up to n using*

$$2\frac{\phi(M)}{M} n \log\log n + (2 + C(k))n - 2\frac{\phi(M)}{M} Li(n) + 4\phi(M) Li(n^{1/2}) + R(n,k)$$

arithmetic operations, where

$$\begin{aligned} C(k) &= 2\frac{\phi(M)}{M}\left(B - \log 2 - \sum_{p \le p_k} \frac{1}{p}\right) \quad < \quad 0 \\ R(n,k) &= O\left(M \log\log\log M + (n^{1/2} + \phi(M)) E(n^{1/2})\right). \end{aligned}$$

Proof: We leave correctness to the reader.

Initialization

Using Lemma 4.1 we find that steps 1 and 2 require at most $O(M \log\log\log M)$ arithmetic operations. Steps 3–7 require $2n + O(M)$ arithmetic operations. Thus initialization uses $2n + O(M \log\log\log M)$ arithmetic operations total.

Main Loop

Steps 1 and 2 use $O(\sqrt{n})$ arithmetic operations.

Step 3 executes at most once for each prime below $\sqrt{n}$. The loop iterates $\phi(M)$ times using 3 arithmetic operations per iteration (the mod operation adds 1 to the standard For-loop operation count). This gives us a total of $3\phi(M)\pi(\sqrt{n}) + O(\sqrt{n})$ arithmetic operations.

Step 4 is executed $\phi(M)(\pi(\sqrt{n}) + O(1))$ times. Mp can be calculated during step 2 for a total cost of $O(\sqrt{n})$. Computing pf takes one operation for a total cost of $\phi(M)(\pi(\sqrt{n}) + O(1))$. For the prime p the loop iterates $\lfloor (n-p^2)/Mp \rfloor$ times at 2 operations per iteration. This gives us

$$2\phi(M) \sum_{p_k < p \le \sqrt{n}} \left\lfloor \frac{n-p^2}{Mp} \right\rfloor = 2\frac{\phi(M)}{M} \left[n \sum_{p \le \sqrt{n}} \frac{1}{p} - n \sum_{p \le p_k} \frac{1}{p} - \sum_{p \le \sqrt{n}} p + O(\sqrt{n}) \right]$$

arithmetic operations. We apply the results from section 2 and simplify to give a total of

$$2\frac{\phi(M)}{M} \left[n \log\log n + \left(B - \log 2 - \sum_{p \le p_k} \frac{1}{p} \right) n - Li(n) \right] + \phi(M) Li(n^{1/2}) + O\left((\sqrt{n} + \phi(M)) E(n^{1/2}) \right)$$

arithmetic operations for step 4.

Combining operation counts for all steps completes the proof. □

4.3 Using a Wheel with Pritchard's Linear Sieve

We modify Pritchard's sieve to use a wheel by altering the initialization step as we did for the sieve of Eratosthenes, and in the main loop we allow f to take on only those values that are relatively prime to M. This gives us the following algorithm.

Algorithm P_k

Input: positive integers n, k

Initialization

1. Find primes up to $\lfloor \sqrt{n} \rfloor$;
2. Compute M, $pr[\,]$, $W[\,]$, and initialize the array $s[\,]$;
   (see algorithm E_k)

Main Loop

1. For $g := pr[k+1]$ to $pr[k+1] + M - 1$ step $W[g \bmod M]$ do:
2. For $f := g$ to $\lfloor n/pr[k+1] \rfloor$ step M do:
3. $q := pr[k+1]$; $x := qf$;
4. Repeat:
5. $p := q$; $s[x] := 0$;
6. $q :=$ nextprime(p); $x := qf$;
7. Until $x > n$ or $p \mid f$;

We now give an estimate on the number of arithmetic operations used by Algorithm P_k.

Theorem 4.3 *Let n, k be the inputs to Algorithm P_k, let M be the product of the first k primes, and let p_i be the ith prime. Then Algorithm P_k computes the set of primes up to n using*

$$\left(2 + 3\frac{\phi(M)}{Mp_{k+1}} + 5\frac{\phi(M)}{M}\right) n - 5Li(n) + O\left(M \log\log\log M + \sqrt{n} + E(n)\right)$$

arithmetic operations.

Proof: We leave the proof of correctness to the reader.

Initialization

This requires $2n + O(M \log\log\log M + \sqrt{n})$ arithmetic operations.

Main Loop

Step 1 uses $O(M)$ arithmetic operations. Step 2 uses $2n/Mp_{k+1} + O(1)$ operations each time it is executed. Since it executes $\phi(M)$ times, this gives us a total of $2(\phi(M)/Mp_{k+1})n + O(1)$ arithmetic operations for step 2. Step 3 is executed $(\phi(M)/Mp_{k+1})n$ times. This gives us a total of $3(\phi(M)/Mp_{k+1})n + O(1)$ arithmetic operations for steps 1–3.

The loop in steps 4–7 iterates once for each composite integer up to n that is relatively prime to M. Each iteration uses 5 arithmetic operations (see the proof of Theorem 3.2). This gives us a total of $5((\phi(M)/M)n - \pi(n)) = 5(\phi(M)/M)n - 5Li(n) + O(E(n))$ arithmetic operations for steps 4–7.

Combining the totals for all steps completes the proof. □

4.4 A Comparison

We conclude our analysis of prime number sieves by showing that the sieve of Eratosthenes uses fewer arithmetic operations than Pritchard's sieve for most practical inputs, even when wheels are added to both algorithms.

In the table below we compare the number of operations used by Algorithms E_k and P_k for several different values of k. The entries in this table are accurate up to $O(n/\log n)$ terms, and the constants are rounded to the nearest 100th. Notice that increasing k (and hence M) improves both algorithms.

Comparing Arithmetic Operations

k	Algorithm E_k	Algorithm P_k
0	$2.00\,n \log\log n + 1.14\,n$	$8.50\,n$
1	$1.00\,n \log\log n + 1.07\,n$	$5.00\,n$
2	$0.67\,n \log\log n + 1.16\,n$	$3.87\,n$
3	$0.53\,n \log\log n + 1.22\,n$	$3.45\,n$
4	$0.46\,n \log\log n + 1.26\,n$	$3.21\,n$
5	$0.42\,n \log\log n + 1.29\,n$	$3.09\,n$

From this, it appears that Algorithm E_k will outperform Algorithm P_k for most practical values of n. We make this observation formal below.

Lemma 4.4 *Let M be the product of the first k primes, and let p_i denote the ith prime. On inputs n, k, the number of arithmetic operations used by Algorithm E_k is at most*

$$2\frac{\phi(M)}{M}n\log\log n + C(k)n + 2\frac{\phi(M)}{M}\frac{4n}{\log^2 n} + 3\sqrt{n} + (4\phi(M)+3)(1.26)\frac{2\sqrt{n}}{\log n} + I(n,k) + r(n),$$

and the number of arithmetic operations used by Algorithm P_k is at least

$$\frac{\phi(M)}{M}\left(5 + \frac{3}{p_{k+1}}\right)n - 5(1.26)\frac{n}{\log n} + 2\sqrt{n} - 3 + I(n,k) + r(n).$$

Here $r(n)$ is the number of operations to compute the square root of n, $I(n,k)$ is the cost of initialization for Algorithm E_k, and $C(k)$ is as in Theorem 4.2.

Proof:

Algorithm E_k:

The cost of initialization is covered by $I(n,k)$. In the main loop, steps 1 and 2 use at most $3\sqrt{n}$ operations. Step 3 uses at most $(3+4\phi(M))\pi(\sqrt{n})$ operations, including the cost of computing Mp and pf for step 4. Finally, step 4 uses at most $2\phi(M)\sum_{p_k<p\le\sqrt{n}} n/Mp$ operations. Applying the results from section 2 completes the proof.

Algorithm P_k:

Initialization uses at least $2\sqrt{n} + I(n,k)$ operations. In the main loop, steps 2 and 3 use at least $(3/p_{k+1})(\phi(M)/M)n - 3$ operations. Step 3–6 use at least $5(\phi(M)/M)n - 5\pi(n)$ operations. Applying (3) completes the proof. □

Theorem 4.5 *Let n, k be the inputs for Algorithms E_k and P_k, let M be the product of the first k primes, and let $g(M) = 10\phi(M)^2 + 100M - 50\phi(M) - 40$. If $1 \le n/g(M) \le 10^{12}$ and $0 \le k \le 9$, then Algorithm P_k uses more arithmetic operations than Algorithm E_k.*

Proof: On inputs n, k, let $E_k(n)$ be the upper bound on the number of arithmetic operations used by Algorithm E_k, and let $P_k(n)$ be the lower bound on the number of arithmetic operations used by Algorithm P_k, as given by Lemma 4.4. Let $f_k(n) = P_k(n) - E_k(n)$.

Then we have

$$\frac{f_k(n)}{n}\cdot\frac{M}{\phi(M)} \ge c(k) - 2\log\log n - \frac{M}{\phi(M)}t(n,k)$$

where

$$\begin{aligned} c(k) &= 5 + 3/p_{k+1} - C(k) = 5 + 3/p_{k+1} - 2B + 2\log 2 + 2\sum_{p\le p_k} 1/p \\ t(n,k) &= \frac{5(1.26)}{\log n} + \frac{8}{\log^2 n}\frac{\phi(M)}{M} + \frac{1}{\sqrt{n}} + \frac{3}{n} + \frac{(4\phi(M)+3)2(1.26)}{\sqrt{n}\log n}. \end{aligned}$$

It suffices to show that $(f_k(n)/n)(M/\phi(M)) > 0$ for the range indicated in the theorem.

The proof then proceeds as in Theorem 3.4, repeated for each k. Due to the large number of calculations needed, the author completed the proof of this theorem with the aid

of a computer program that determined what subintervals to use and verified that $f_k(n)/n \cdot M/\phi(M)$ was positive on each subinterval. The output from that program (which is the rest of this proof) appears below.

```
K=0; M=1; phi(M)=1; phi(M)/M= 1.0000000; c(0)= 7.3632998; g(M)=20
[ 2.0000e+01,  4.5212e+01]:   t(n,k) <= 4.684703; 2*loglog n <= 2.675968; (f(n)/n)*(M/phi(M)) >=  0.002629;
[ 4.4986e+01,  4.2930e+03]:   t(n,k) <= 3.114052; 2*loglog n <= 4.248049; (f(n)/n)*(M/phi(M)) >=  0.001198;
[ 4.2715e+03,  8.0369e+10]:   t(n,k) <= 0.916376; 2*loglog n <= 6.446524; (f(n)/n)*(M/phi(M)) >=  0.000399;
[ 7.9967e+10,  1.3015e+15]:   t(n,k) <= 0.263646; 2*loglog n <= 7.099365; (f(n)/n)*(M/phi(M)) >=  0.000288;
[ 1.2950e+15,  5.0101e+15]:   t(n,k) <= 0.187656; 2*loglog n <= 7.175367; (f(n)/n)*(M/phi(M)) >=  0.000277;
[ 4.9850e+15,  5.7116e+15]:   t(n,k) <= 0.180420; 2*loglog n <= 7.182603; (f(n)/n)*(M/phi(M)) >=  0.000276;
** For k=0 Valid interval = [20,  5.7116e+15] **

K=1; M=2; phi(M)=1; phi(M)/M= 0.5000000; c(1)= 7.8632998; g(M)=120
[ 1.2000e+02,  1.4803e+03]:   t(n,k) <= 1.943092; 2*loglog n <= 3.975742; (f(n)/n)*(M/phi(M)) >=  0.001373;
[ 1.4729e+03,  8.0310e+07]:   t(n,k) <= 1.029875; 2*loglog n <= 5.802998; (f(n)/n)*(M/phi(M)) >=  0.000551;
[ 7.9909e+07,  2.9565e+15]:   t(n,k) <= 0.358523; 2*loglog n <= 7.145972; (f(n)/n)*(M/phi(M)) >=  0.000281;
[ 2.9418e+15,  3.1219e+18]:   t(n,k) <= 0.180031; 2*loglog n <= 7.503002; (f(n)/n)*(M/phi(M)) >=  0.000235;
[ 3.1063e+18,  1.1355e+19]:   t(n,k) <= 0.150163; 2*loglog n <= 7.562745; (f(n)/n)*(M/phi(M)) >=  0.000228;
[ 1.1299e+19,  1.3830e+19]:   t(n,k) <= 0.145680; 2*loglog n <= 7.571712; (f(n)/n)*(M/phi(M)) >=  0.000227;
** For k=1 Valid interval = [120,  1.3830e+19] **

K=2; M=6; phi(M)=2; phi(M)/M= 0.3333333; c(2)= 8.1299667; g(M)=500
[ 5.0000e+02,  2.6552e+03]:   t(n,k) <= 1.332986; 2*loglog n <= 4.129737; (f(n)/n)*(M/phi(M)) >=  0.001271;
[ 2.6419e+03,  1.7958e+06]:   t(n,k) <= 0.931560; 2*loglog n <= 5.334591; (f(n)/n)*(M/phi(M)) >=  0.000696;
[ 1.7868e+06,  6.7557e+12]:   t(n,k) <= 0.452680; 2*loglog n <= 6.771586; (f(n)/n)*(M/phi(M)) >=  0.000339;
[ 6.7219e+12,  1.9455e+18]:   t(n,k) <= 0.216354; 2*loglog n <= 7.480668; (f(n)/n)*(M/phi(M)) >=  0.000238;
[ 1.9358e+18,  1.4765e+20]:   t(n,k) <= 0.151123; 2*loglog n <= 7.676383; (f(n)/n)*(M/phi(M)) >=  0.000216;
[ 1.4692e+20,  4.0181e+20]:   t(n,k) <= 0.136906; 2*loglog n <= 7.719037; (f(n)/n)*(M/phi(M)) >=  0.000211;
[ 3.9980e+20,  4.9466e+20]:   t(n,k) <= 0.133991; 2*loglog n <= 7.727782; (f(n)/n)*(M/phi(M)) >=  0.000210;
** For k=2 Valid interval = [500,  4.9466e+20] **

K=3; M=30; phi(M)=8; phi(M)/M= 0.2666667; c(3)= 8.3585377; g(M)=3200
[ 3.2000e+03,  1.4056e+04]:   t(n,k) <= 1.025131; 2*loglog n <= 4.513247; (f(n)/n)*(M/phi(M)) >=  0.001049;
[ 1.3985e+04,  4.9122e+06]:   t(n,k) <= 0.770190; 2*loglog n <= 5.469673; (f(n)/n)*(M/phi(M)) >=  0.000651;
[ 4.8876e+06,  7.5609e+12]:   t(n,k) <= 0.421068; 2*loglog n <= 6.779195; (f(n)/n)*(M/phi(M)) >=  0.000338;
[ 7.5231e+12,  9.0510e+18]:   t(n,k) <= 0.214914; 2*loglog n <= 7.552379; (f(n)/n)*(M/phi(M)) >=  0.000230;
[ 9.0057e+18,  3.9200e+21]:   t(n,k) <= 0.145468; 2*loglog n <= 7.812830; (f(n)/n)*(M/phi(M)) >=  0.000202;
[ 3.9004e+21,  2.1365e+22]:   t(n,k) <= 0.127585; 2*loglog n <= 7.879901; (f(n)/n)*(M/phi(M)) >=  0.000195;
[ 2.1258e+22,  3.2194e+22]:   t(n,k) <= 0.123349; 2*loglog n <= 7.895786; (f(n)/n)*(M/phi(M)) >=  0.000193;
[ 3.2033e+22,  3.5421e+22]:   t(n,k) <= 0.122367; 2*loglog n <= 7.899470; (f(n)/n)*(M/phi(M)) >=  0.000193;
** For k=3 Valid interval = [3200,  3.5421e+22] **

K=4; M=210; phi(M)=48; phi(M)/M= 0.2285714; c(4)= 8.4884081; g(M)=41600
[ 4.1600e+04,  6.5663e+04]:   t(n,k) <= 0.840001; 2*loglog n <= 4.812501; (f(n)/n)*(M/phi(M)) >=  0.000904;
[ 6.5335e+04,  5.4129e+05]:   t(n,k) <= 0.760447; 2*loglog n <= 5.160692; (f(n)/n)*(M/phi(M)) >=  0.000759;
[ 5.3858e+05,  1.9394e+09]:   t(n,k) <= 0.540000; 2*loglog n <= 6.125441; (f(n)/n)*(M/phi(M)) >=  0.000469;
[ 1.9297e+09,  5.3611e+15]:   t(n,k) <= 0.299205; 2*loglog n <= 7.179109; (f(n)/n)*(M/phi(M)) >=  0.000277;
[ 5.3343e+15,  4.2070e+20]:   t(n,k) <= 0.175366; 2*loglog n <= 7.720973; (f(n)/n)*(M/phi(M)) >=  0.000211;
[ 4.1859e+20,  4.0057e+22]:   t(n,k) <= 0.133489; 2*loglog n <= 7.904202; (f(n)/n)*(M/phi(M)) >=  0.000193;
[ 3.9857e+22,  1.5535e+23]:   t(n,k) <= 0.121737; 2*loglog n <= 7.955621; (f(n)/n)*(M/phi(M)) >=  0.000188;
[ 1.5458e+23,  2.2361e+23]:   t(n,k) <= 0.118630; 2*loglog n <= 7.969215; (f(n)/n)*(M/phi(M)) >=  0.000186;
** For k=4 Valid interval = [41600,  2.2361e+23] **

K=5; M=2310; phi(M)=480; phi(M)/M= 0.2077922; c(5)= 8.6282682; g(M)= 2.51096e+06
[ 2.5110e+06,  7.9740e+06]:   t(n,k) <= 0.643334; 2*loglog n <= 5.531593; (f(n)/n)*(M/phi(M)) >=  0.000631;
[ 7.9341e+06,  2.9668e+09]:   t(n,k) <= 0.511793; 2*loglog n <= 6.164805; (f(n)/n)*(M/phi(M)) >=  0.000460;
[ 2.9520e+09,  7.9857e+15]:   t(n,k) <= 0.296520; 2*loglog n <= 7.200994; (f(n)/n)*(M/phi(M)) >=  0.000274;
[ 7.9458e+15,  2.4580e+21]:   t(n,k) <= 0.173319; 2*loglog n <= 7.793966; (f(n)/n)*(M/phi(M)) >=  0.000204;
[ 2.4457e+21,  6.6150e+23]:   t(n,k) <= 0.128608; 2*loglog n <= 8.009160; (f(n)/n)*(M/phi(M)) >=  0.000183;
[ 6.5819e+23,  3.8758e+24]:   t(n,k) <= 0.115424; 2*loglog n <= 8.072611; (f(n)/n)*(M/phi(M)) >=  0.000177;
[ 3.8564e+24,  6.3618e+24]:   t(n,k) <= 0.111803; 2*loglog n <= 8.090041; (f(n)/n)*(M/phi(M)) >=  0.000176;
[ 6.3300e+24,  7.2748e+24]:   t(n,k) <= 0.110828; 2*loglog n <= 8.094732; (f(n)/n)*(M/phi(M)) >=  0.000175;
** For k=5 Valid interval = [ 2.51096e+06,  7.2748e+24] **

K=6; M=30030; phi(M)=5760; phi(M)/M= 0.1918082; c(6)= 8.7278156; g(M)= 3.34491e+08
[ 3.3449e+08,  3.8942e+09]:   t(n,k) <= 0.486765; 2*loglog n <= 6.189593; (f(n)/n)*(M/phi(M)) >=  0.000454;
```

```
[ 3.8747e+09,  2.5324e+14]:   t(n,k) <= 0.330773; 2*loglog n <= 7.003013; (f(n)/n)*(M/phi(M)) >=  0.000302;
[ 2.5198e+14,  5.1234e+20]:   t(n,k) <= 0.191492; 2*loglog n <= 7.729255; (f(n)/n)*(M/phi(M)) >=  0.000210;
[ 5.0977e+20,  1.3594e+24]:   t(n,k) <= 0.132804; 2*loglog n <= 8.035253; (f(n)/n)*(M/phi(M)) >=  0.000180;
[ 1.3526e+24,  2.2587e+25]:   t(n,k) <= 0.113880; 2*loglog n <= 8.133927; (f(n)/n)*(M/phi(M)) >=  0.000172;
[ 2.2474e+25,  5.2527e+25]:   t(n,k) <= 0.108374; 2*loglog n <= 8.162632; (f(n)/n)*(M/phi(M)) >=  0.000169;
[ 5.2264e+25,  6.6770e+25]:   t(n,k) <= 0.106823; 2*loglog n <= 8.170718; (f(n)/n)*(M/phi(M)) >=  0.000169;
** For k=6 Valid interval = [ 3.34491e+08,  6.6770e+25] **

K=7; M=510510; phi(M)=92160; phi(M)/M= 0.1805254; c(7)= 8.8268871; g(M)= 8.49811e+10
[ 8.4981e+10,  3.4569e+12]:   t(n,k) <= 0.379255; 2*loglog n <= 6.725702; (f(n)/n)*(M/phi(M)) >=  0.000347;
[ 3.4396e+12,  3.7804e+18]:   t(n,k) <= 0.237333; 2*loglog n <= 7.511972; (f(n)/n)*(M/phi(M)) >=  0.000234;
[ 3.7615e+18,  6.1239e+23]:   t(n,k) <= 0.148095; 2*loglog n <= 8.006346; (f(n)/n)*(M/phi(M)) >=  0.000183;
[ 6.0933e+23,  1.0803e+26]:   t(n,k) <= 0.115515; 2*loglog n <= 8.186837; (f(n)/n)*(M/phi(M)) >=  0.000167;
[ 1.0749e+26,  5.8246e+26]:   t(n,k) <= 0.105508; 2*loglog n <= 8.242275; (f(n)/n)*(M/phi(M)) >=  0.000163;
[ 5.7954e+26,  9.5670e+26]:   t(n,k) <= 0.102613; 2*loglog n <= 8.258314; (f(n)/n)*(M/phi(M)) >=  0.000161;
[ 9.5191e+26,  1.1024e+27]:   t(n,k) <= 0.101790; 2*loglog n <= 8.262872; (f(n)/n)*(M/phi(M)) >=  0.000161;
** For k=7 Valid interval = [ 8.49811e+10,  1.1024e+27] **

K=8; M=9699690; phi(M)=1658880; phi(M)/M= 0.1710240; c(8)= 8.9046907; g(M)= 2.75197e+13
[ 2.7520e+13,  1.3965e+15]:   t(n,k) <= 0.308013; 2*loglog n <= 7.103411; (f(n)/n)*(M/phi(M)) >=  0.000287;
[ 1.3895e+15,  1.2465e+21]:   t(n,k) <= 0.194674; 2*loglog n <= 7.766203; (f(n)/n)*(M/phi(M)) >=  0.000206;
[ 1.2403e+21,  2.9141e+25]:   t(n,k) <= 0.130300; 2*loglog n <= 8.142636; (f(n)/n)*(M/phi(M)) >=  0.000171;
[ 2.8995e+25,  1.5554e+27]:   t(n,k) <= 0.107853; 2*loglog n <= 8.273899; (f(n)/n)*(M/phi(M)) >=  0.000160;
[ 1.5476e+27,  5.5463e+27]:   t(n,k) <= 0.100978; 2*loglog n <= 8.314105; (f(n)/n)*(M/phi(M)) >=  0.000157;
[ 5.5186e+27,  8.0923e+27]:   t(n,k) <= 0.098961; 2*loglog n <= 8.325897; (f(n)/n)*(M/phi(M)) >=  0.000156;
** For k=8 Valid interval = [ 2.75197e+13,  8.0923e+27] **

K=9; M=223092870; phi(M)=36495360; phi(M)/M= 0.1635882; c(9)= 8.9646606; g(M)= 1.33191e+16
[ 1.3319e+16,  3.4326e+17]:   t(n,k) <= 0.256486; 2*loglog n <= 7.396534; (f(n)/n)*(M/phi(M)) >=  0.000248;
[ 3.4154e+17,  4.7011e+22]:   t(n,k) <= 0.172442; 2*loglog n <= 7.910344; (f(n)/n)*(M/phi(M)) >=  0.000192;
[ 4.6776e+22,  3.2840e+26]:   t(n,k) <= 0.121203; 2*loglog n <= 8.223592; (f(n)/n)*(M/phi(M)) >=  0.000164;
[ 3.2676e+26,  9.7077e+27]:   t(n,k) <= 0.103543; 2*loglog n <= 8.331553; (f(n)/n)*(M/phi(M)) >=  0.000156;
[ 9.6591e+27,  2.8713e+28]:   t(n,k) <= 0.098084; 2*loglog n <= 8.364929; (f(n)/n)*(M/phi(M)) >=  0.000153;
[ 2.8569e+28,  3.9818e+28]:   t(n,k) <= 0.096456; 2*loglog n <= 8.374884; (f(n)/n)*(M/phi(M)) >=  0.000152;
** For k=9 Valid interval = [ 1.33191e+16,  3.9818e+28] **
```

That completes the proof. □

5 Implementation Results

In this section, we give the timing results from implementations of several versions of both the sieve of Eratosthenes and Pritchard's linear sieve. We used a DECstation 3100, and we used the "C" programming language [KR78]. The DECstation has a RISC-style MIPS R2000 CPU which is rated at 12 MIPS.

In previous sections we analyzed the number of arithmetic operations used by prime number sieves. Although we believe that such an analysis gives a reasonable prediction of how prime number sieves behave in practice, there are several other factors such an analysis does not take into account that can affect program behavior. Below we describe three of these factors.

- *Multiplication and Division.*
  Our analysis treated all arithmetic operations equally. In practice, multiplication and division are much slower operations than addition and subtraction. As a result, since Algorithm E uses none of the more expensive multiplies or divides in its innermost loop and Algorithm P does, we expect the difference between their actual running times to be greater than what our theory predicts.

- *Compiler Optimization.*
  The compiler optimized all the algorithms that were implemented. Often it is difficult to say exactly what effect optimization has on the running time and whether the results of optimization improve both algorithms equally.

- *Cache.*
  The MIPS R2000 CPU has a cache used to decrease the average cost of fetching instructions and data from main memory. The presence of a cache will cause algorithms with some locality of reference to execute faster. However, prime number sieves using larger wheels tend to generate seemingly random memory references, thus degrading their performance.

  To help alleviate this problem, we used the following alternate initialization routine for all algorithms that used wheels.

  1. For $i := 0$ to $M - 1$ do:
  2. If $W[i] = 0$ then $Y[i] := 0$ else $Y[i] := 1$;
  3. For $j := 0$ to n step M do:
  4. For $i := 0$ to $M - 1$ do: $s[j + i] := Y[i]$;

  Notice that this algorithm proceeds sequentially through the array $s[\,]$. Although more arithmetic operations are used, the increase in memory reference locality greatly improved performance. For example, for $n = 5000000$, Algorithm P_5 used more than 9 CPU seconds with the old initialization routine as compared to less than 5 with the new one.

  This improvement does not help the main loop of either algorithm.

In the table below we give timing results for several variants of the sieve of Eratosthenes. The left column gives the input n, and under each algorithm is the average number of CPU seconds used to find the primes up to n.

Sieve of Eratosthenes: Timing Results

n	E_0	E_1	E_2	E_3	E_4	E_5
500	0.00043	0.00029	0.00027	0.00048	0.00140	0.01035
5000	0.00473	0.00305	0.00254	0.00293	0.00551	0.03207
50000	0.05156	0.03281	0.02617	0.02578	0.03164	0.10663
500000	0.55817	0.35037	0.28240	0.28358	0.28397	0.47028
5000000	6.06719	3.82905	3.20722	3.25214	3.28104	3.38455

Each algorithm was run several times on each input, and the time reported in the table is the average of these times. The results of all algorithms were checked using a trial division routine.

Algorithm E_2 gives the best performance in practice. This contradicts our previous analysis; we would expect that using larger wheels would improve performance when n is sufficiently large. This discrepancy is most likely due to the CPU's cache; for larger wheels, the array $s[\,]$ appears to be accessed randomly causing the cache to be missed almost every time.

We also timed several variants of Pritchard's sieve, and those results appear in the table below. The methods used were identical to those used for the sieve of Eratosthenes.

Pritchard's Sieve: Timing Results

n	P_0	P_1	P_2	P_3	P_4	P_5
500	0.00147	0.00074	0.00050	0.00051	0.00095	0.00649
5000	0.01519	0.00777	0.00527	0.00453	0.00430	0.00973
50000	0.15507	0.08203	0.05664	0.04843	0.04180	0.04413
500000	1.60732	0.86245	0.61597	0.53825	0.47146	0.43787
5000000	16.47706	9.00177	6.57146	5.83518	5.27427	4.88251

Here we see that as larger wheels are used, the running time decreases. This is what our theory predicts.

Our main result, Theorem 4.5, essentially says that the sieve of Eratosthenes uses fewer arithmetic operations than Pritchard's sieve. So we expect the sieve of Eratosthenes to perform better in practice. This is in fact the case, as can be seen by comparing the time used by variants of both algorithms on the input $n = 5000000$.

Comparing Algorithm E_k to Algorithm P_k for $n = 5000000$

$k =$	0	1	2	3	4	5
E_k	6.06719	3.82905	3.20722	3.25214	3.28104	3.38455
P_k	16.47706	9.00177	6.57146	5.83518	5.27427	4.88251

In the table above we have simply copied the last line of each of the two previous tables.

6 Remarks

We conclude with a few remarks:

Luo [Luo89] conjectured that, for Algorithm E_k, choosing $k > 2$ would result in a less efficient algorithm due to the cost of increased overhead. In theory, Luo was not correct. Using larger wheels is beneficial if n is sufficiently large. However, in practice Luo was correct, but we believe it is not because of an increased cost in overhead. We think that the reason for degraded performance is due to a decrease in the locality of memory references.

In this paper, we did not concern ourselves with the space used by prime number sieves. In fact, by segmenting the sieve of Eratosthenes we get an algorithm using only $O(\sqrt{n})$ space; see the paper by Bays and Hudson [BH77].

In addition, the kth wheel can be used to reduce the space requirement of Pritchard's sieve to $O((\phi(M)/M)n)$. The technique is to only store a bit for each integer relatively prime to M. With some precomputation, it is possible to construct an invertible mapping to this space from the integers $1 \ldots n$ such that the mapping is computable in $O(1)$ arithmetic steps.

It is possible to choose a wheel based on the size of the input. If k is chosen maximal so that $M \leq \sqrt{n}$, by (5) from section 2 we have

$$\frac{\phi(M)}{M} = \prod_{p \leq p_k} \left(1 - \frac{1}{p}\right) \sim \frac{1}{e^\gamma \log p_k} \sim \frac{1}{e^\gamma \log\log M} = O\left(\frac{1}{\log\log n}\right).$$

By using a wheel of this size and incorporating the space reduction method mentioned above for Pritchard's algorithm, the result is a prime number sieve using only $O(n/\log\log n)$ arithmetic operations. Such sublinear prime number sieves are not new; the first is due to Mairson [Mai77], and Pritchard's dynamic wheel sieve is the first sublinear sieve that uses only additions and subtractions [Pri81, Pri82].

Pritchard's fixed wheel sieve [Pri83], which uses $O(n)$ arithmetic operations and $O(\sqrt{n})$ space, is essentially a segmented version of the sieve of Eratosthenes using a wheel of size $M \approx \sqrt{n}$ as mentioned above. This is probably the most practical sieve algorithm known. Parberry[Par81] gave a practical parallel version of this algorithm running in $O(\sqrt{n})$ time using $O(\sqrt{n})$ processors.

Acknowledgements

My thanks go to Jim Larus, Dan Lieuwen, and Scott Vandenberg for explaining to me some of the peculiarities of the C compiler, to Karen Miller for providing information on the MIPS R2000 CPU, to Joao Meidanis both for pointing out that Lemma 2.1 holds for all positive values of α and for his comments on a earlier draft of this paper, and finally to my PhD advisor Eric Bach for encouraging me to write this paper.

References

[Bar79] D. R. Barstow. An experiment in knowledge-based automatic programming. *Artificial Intelligence*, 12:73–119, 1979.

[Ben86] S. Bengelloun. An incremental primal sieve. *Acta Informatica*, 23(2):119–125, 1986.

[BH77] C. Bays and R. Hudson. The segmented sieve of Eratosthenes and primes in arithmetic progressions to 10^{12}. *BIT*, 17:121–127, 1977.

[Dav80] H. Davenport. *Multiplicative Number Theory.* Springer-Verlag, New York, 1980.

[DRK26] M. L. D'ooge, F. E. Robbins, and L. C. Karpinski. *Nicomachus of Gerasa: Introduction to Arithmetic.* MacMillan, New York, 1926. University of Michigan Studies Humanistic Series Volume 16.

[GM78] D. Gries and J. Misra. A linear sieve algorithm for finding prime numbers. *Communications of the ACM*, 21(12):999–1003, 1978.

[Ing32] A. E. Ingham. *The Distribution of Prime Numbers.* Cambridge Tract No. 30. Cambridge University Press, 1932.

[KR78] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[Luo89] X. Luo. A practical sieve algorithm for finding prime numbers. *Communications of the ACM*, 32(3):344–346, 1989. Correction in 32(11):1367.

[Mai77] H. G. Mairson. Some new upper bounds on the generation of prime numbers. *Communications of the ACM*, 20(9):664–669, 1977.

[Mis81] J. Misra. An exercise in program explanation. *ACM Transactions on Programming Languages and Systems*, 3(1):104–109, 1981.

[Par81] I. Parberry. Parallel speedup of sequential prime number sieves. Technical Report TR30, University of Queensland, 1981.

[Pri81] P. Pritchard. A sublinear additive sieve for finding prime numbers. *Communications of the ACM*, 24(1):18–23,772, 1981.

[Pri82] P. Pritchard. Explaining the wheel sieve. *Acta Informatica*, 17:477–485, 1982.

[Pri83] P. Pritchard. Fast compact prime number sieves (among others). *Journal of Algorithms*, 4:332–344, 1983.

[Pri84] P. Pritchard. Some negative results concerning prime number generators. *Communications of the ACM*, 27(1):53–57, 1984.

[Pri87] P. Pritchard. Linear prime-number sieves: A family tree. *Science of Computer Programming*, 9:17–35, 1987.

[RS62] J. B. Rosser and L. Schoenfeld. Approximate formulas for some functions of prime numbers. *Illinois Journal of Mathematics*, 6:64–94, 1962.

[Sch76] L. Schoenfeld. Sharper bounds for the Chebyshev functions $\theta(x)$ and $\psi(x)$. II. *Mathematics of Computation*, 30(134):337–360, 1976.

[SZ68] T. Salat and S. Znam. On the sums of prime powers. *Acta Facultatis Rerum Naturalium Universitatis Comenianea: Mathematica*, 21:21–25, 1968.