AN INTRODUCTION TO PRIME NUMBER SIEVES

by

Jonathan Sorenson

Computer Sciences Technical Report #909

January 1990

An Introduction to Prime Number Sieves

Jonathan Sorenson*

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

January 2, 1990

Abstract

We discuss prime number sieves, that is, sieve algorithms that find all the prime numbers up to a bound n. We start with the classical Sieve of Eratosthenes, which uses $O(n \log\log n)$ additions and $O(n)$ bits of storage space. We then show how to improve both the time and space used by this algorithm. The fastest known improvement runs in $O(n/\log\log n)$ additions, and is due to Pritchard. The most space-efficient algorithms use only $O(\sqrt{n})$ bits.

1 Introduction.

In this paper we discuss algorithms to solve the following problem: Given a positive integer n, find all the primes $p \leq n$. Perhaps the most famous algorithm to solve this problem is due to the ancient librarian Eratosthenes, who lived about 2000 years ago. The Sieve of Eratosthenes takes $O(n \log\log n)$ additions using $O(n)$ bits of storage to find all the primes up to n.

The purpose of this paper is to give an introduction to the many improvements and extensions of Eratosthenes's sieve. These algorithms fall into one of two broad categories:

1. Algorithms which discard each composite integer exactly once, thus using only $O(n)$ arithmetic operations, and

2. Algorithms which use only $O(\sqrt{n})$ space.

*Supported by NSF grant CCR-8552596

We cover one algorithm of each type, illustrating the basic ideas involved and sketching the running time analyses. We then show how to use *wheels* to improve the running times of each by a factor of $\log \log n$.

In 1977, Mairson [Mai77] published theoretically significant improvements to Eratosthenes's sieve. He developed an algorithm which uses only $O(n)$ arithmetic operations, thus falling into category (1). Using a wheel, though he did not call it that, he improved this to $O(n/\log \log n)$ operations. Unfortunately, he had to bring in multiplications to achieve these bounds, and his space requirement jumped from $O(n)$ bits for Eratosthenes's sieve to $O(n \log n/\log \log n)$ bits.

Since then, many researchers have designed category (1) algorithms, including Gries and Misra [GM78], Barstow [Bar79], Misra [Mis81], Pritchard [Pri81, Pri82], and Bengelloun [Ben86]. All of these can be improved to $O(n/\log \log n)$ arithmetic operations using wheels. Pritchard [Pri87] summarized these results.

Two of these are notable for additional reasons.

Pritchard [Pri81] showed how to replace multiplications with additions, thus improving on the Sieve of Eratosthenes at the bit-complexity level. He also showed how to reduce the storage requirements to $O(n/\log \log n)$ bits.

Bengelloun [Ben86] designed an algorithm which is *incremental*. This algorithm maintains a data structure that stores sufficient information so that, if the primes up to n are known, the primality of $n+1$ can be determined in $O(1)$ operations. This requires a total of $O(n \log n)$ bits of space. Such an algorithm is useful in applications like trial division where an upper bound on the primes needed is not known beforehand.

We will not cover trading multiplications for additions or incremental sieve ideas here, in the interest of length.

All of the category (1) algorithms listed above can find the primes up to one million in under a minute on a DEC VAXstation 3200. So in practice, the real limitation is space, not time.

Bays and Hudson [BH77] showed how to *segment* the Sieve of Eratosthenes to reduce the space requirement to only $O(\sqrt{n})$ bits while maintaining the same running time of $O(n \log \log n)$ additions, thus giving a category (2) algorithm. Brent [Bre73] also used a segmented sieve. Pritchard [Pri83] applied a wheel to give an algorithm using only $O(n)$ additions which allows segmentation. These segmented algorithms are the most useful in practice for larger values of n.

It is an open question whether it is possible to get a sublinear arithmetic complexity for this problem using only $O(\sqrt{n})$ space, or in other words, find an algorithm falling into both categories.

This paper is organized as follows. After some definitions and formulas in section 2, we review the Sieve of Eratosthenes in section 3. In section 4, we give an example of an algorithm using $O(n)$ arithmetic operations, that is, one in category (1). In section 5, we describe Bays and Hudson's segmented sieve algorithm, a category (2) algorithm. We then describe the concept of a wheel in section 6; we give some examples and describe a very simple basic data structure which makes them easy to use. We apply the wheel to our category (1) algorithm giving one which uses $O(n/\log \log n)$ arithmetic operations in section 7, and we also apply the wheel to our category (2) algorithm giving Pritchard's linear segmented wheel sieve in section 8. We close with remarks in section 9 and implementation results in section 10.

2 Some Definitions and Formulas.

In this section we place all the definitions in one place for easy reference, and we also quote a few formulas.

Definitions

p is always a prime. p_i is the ith prime, with $p_1 = 2$. If T is a set, then $\#T$ denotes its cardinality.

$$\begin{aligned}
\phi(n) &= \#\{x < n \ : \ \gcd(n,x) = 1\} \quad \text{(Euler's function)} \\
\pi(x) &= \#\{p \le x \ : \ p \text{ is prime }\} \\
M_k &= \prod_{i=1}^{k} p_i \\
W_k &= \{x < M_k \ : \ \gcd(M_k, x) = 1\} \quad \text{(the } k\text{th wheel)} \\
W_k(n) &= \{x \le n \ : \ \gcd(M_k, x) = 1\} \quad \text{(the } k\text{th wheel extended to } n\text{)}
\end{aligned}$$

wk is an array used in implementing a data structure for the kth extended wheel. $F(p)$ is the set of factors f used to generate multiples of the prime p. Δ is the length of a sieve segment.

S is the set of integers sifted by a sieve algorithm, and s is an array-based implementation of S. next(S,x) gives the smallest $y \in S$ with $y > x$, and prev(S,x) gives the largest $y \in S$ with $y < x$. For both functions we assume that $x \in S$.

Formulas

We make use of the following formulas whose proofs can be found in Hardy and Wright [HW79, chapter 22]. All sums and products are only over primes.

$$\sum_{p \le x} \log p = \Theta(x) \tag{1}$$

$$\sum_{p \le x} \frac{1}{p} = \log\log x + O(1) \tag{2}$$

$$\prod_{p \le x} \left(1 - \frac{1}{p}\right) \sim \frac{e^{-\gamma}}{\log x} = \Theta\left(\frac{1}{\log x}\right) \tag{3}$$

Note that (3) is Mertens's Theorem, where γ is Euler's constant and $e^{-\gamma} = 0.5614\cdots$.

3 The Sieve of Eratosthenes.

We will now review the Sieve of Eratosthenes. The algorithm works as follows.

We start with an array of bits of length n. We initialize each array location to 1, except for position 1, which we set to 0. By the end of the algorithm, in position x we will have a 0 if x is composite and a 1 if x is prime. Starting with $p = 2$, for each prime $p \leq \sqrt{n}$, we set position m to 0 for all multiples m of p, starting with p^2 and finishing with $p \cdot \lfloor n/p \rfloor$.

This gives the following algorithm.

```
*Initialize:*
    $s[1] := 0$;
    for $i := 2$ to $n$ do $s[i] := 1$;
*Main Loop:*
    $p := 2$;
    while $p^2 \leq n$ do
        *Remove multiples:*
            for $f := p$ to $\lfloor n/p \rfloor$ do: $s[pf] := 0$;
        *Find the next prime:*
            repeat $p := p + 1$ until $s[p] = 1$;
    end while;
```

Let us analyze the complexity of this algorithm. It uses $O(n)$ bits of space for the array s. Initialization takes $O(n)$ operations. The total time spent finding the next prime is equal to the number of times we add one to p, which is at most $O(\sqrt{n})$. Finally, the time spent removing multiples is at most

$$\sum_{p \leq \sqrt{n}} \frac{n}{p} \quad = \quad O(n \log \log n)$$

using (2).

Notice that the array s can be thought of as a set S; an integer $x \in S$ if $s[x] = 1$ and $x \notin S$ if $s[x] = 0$. Thus, the algorithm starts with $S = \{x \ : \ 2 \leq x \leq n\}$, and finishes with $S = \{p \leq n \ : \ p \text{ is prime }\}$.

Let next(S,x) be the smallest element in S larger than x. For all our implementations, we assume that $x \in S$.

For a fixed prime p, we can also think of the integers f used to remove the multiple pf as a set. Call this set $F(p)$. For Eratosthenes's sieve, we have $F(p) = \{f \ : \ p \leq f \leq \lfloor n/p \rfloor\}$. If $n = 100$, this gives us

$$\begin{array}{ll} F(2) = \{2, \ldots, 50\} & F(5) = \{5, \ldots, 20\} \\ F(3) = \{3, \ldots, 33\} & F(7) = \{7, \ldots, 14\}. \end{array}$$

Let us now rewrite the Sieve of Eratosthenes in the following abstract form.

```
Initialize:
    $S := \{2, \ldots, n\}$
Main loop:
    $p := 2$;
    while $p^2 \le n$ do
        Remove multiples:
            for each $f \in F(p)$ do
                $S := S - \{pf\}$
        Find the next prime:
            $p :=$ next($S$,$p$)
    end while;
```

Loop Invariant: If $p = p_i$, then for all $x \in S$ with $x \ge p_i$, $j < i$ implies $\gcd(x, p_j) = 1$.

In our descriptions of the various algorithms which follow, we will use this abstract algorithm as an outline. In other words, the variations on the Sieve of Eratosthenes differ mainly in their implementations of the sets S and $F(p)$.

4 A Linear Algorithm.

We now show how Eratosthenes's sieve can be improved to use only a linear number of arithmetic operations. The algorithm we present is largely due to Gries and Misra [GM78], though we use an idea or two from Pritchard [Pri81].

The problem with Eratosthenes's sieve is that each composite integer x may be "crossed off" the array s more than once, especially if x has many divisors. In the terms of our abstract algorithm, the set $F(p)$ is larger than necessary. Our goal is to reduce $F(p)$ to contain only elements which must be there, so that we remove each composite integer from S exactly once. If each removal uses $O(1)$ steps, that will give us a linear algorithm.

Choosing $F(p)$

Every composite integer x, $2 < x \le n$, can be written uniquely in the form $x = p \cdot f$ such that p is prime and f has no prime divisor smaller than p. This means that we will choose

$$F(p) \quad = \quad \{f \ : \ p \le f \le \lfloor n/p \rfloor, \ \forall q < p, \ \gcd(q, f) = 1\} \, .$$

For example, if $n = 100$, we want the following for $F(p)$:

$$\begin{aligned} F(2) &= \{2, \ldots, 50\} \\ F(3) &= \{3, 5, 7, 9, \ldots, 33\} \\ F(5) &= \{5, 7, 11, 13, 17, 19\} \\ F(7) &= \{7, 11, 13\} \end{aligned}$$

The next question, then, is how do we compute $F(p)$? Well, in some sense, we already have it. Consider S just before we remove multiples of p. By the loop invariant, S contains precisely the primes up to p and integers between p and n which are not divisible by any prime below p. So, we simply compute

$$F(p) \quad = \quad S \cap \{x \;:\; p \leq x \leq \lfloor n/p \rfloor\}\,.$$

If S were implemented as an array of bits, this computation is no faster than simply using all values of f in this range, which is what the Sieve of Eratosthenes does. To fix this, we implement S as a doubly linked list, and we use arrays for the implementation to allow us to randomly access nodes in the list.

Implementing S

One way to implement S as a doubly linked list using an array is as follows (we use Pascal notation).

```
type
    $srec$ = record
        $in$ : boolean;
        $prev$, $next$ : integer;
    end;
var $s$ : array [1..n] of $srec$;
```

Then we implement the functions $next(S,x)$, which finds the next element in S larger than x for $x \in S$, $prev(S,x)$, which gives the next smaller element of S, and $remove(S,x)$, which removes x from S, as follows.

```
next($S$,$x$) returns $s[x].next$;
prev($S$,$x$) returns $s[x].prev$;

remove($S$,$x$): $s[x].in$ := false;
                 $s[s[x].next].prev := s[x].prev$;
                 $s[s[x].prev].next := s[x].next$;
```

So the next, prev, and remove functions all use $O(1)$ arithmetic operations.

We leave the problem of initializing the array s to the reader.

The total space used by s is $O(n \log n)$ bits, because s has n nodes, and each node stores two integer pointers which require $O(\log n)$ bits.

The Algorithm

We have one more problem to solve, and that is to generate the elements of $F(p)$. The simplest way is to start by setting $f := p$ and repeatedly executing $f := \text{next}(S, f)$ until $f \geq \lfloor n/p \rfloor$. For each value taken by f, we remove pf from S.

There is a problem with this straight-forward approach, however; we are changing S while using it to compute $F(p)$. The elements in $F(p)$ may be removed from S before we've had a chance to use them as values of f. For example, $p^2 \in F(p)$, but the first value of f, which is p, causes $pf = p^2$ to be removed from S. Thus f can not later take the value p^2, which means p^3 is not removed, though it should be, and so on. There are three ways to avoid this:

1. Make a pass through S to find all the elements of $F(p)$ and copy them into an auxiliary array. Then make a pass through this array to generate the values of f to remove pf from S. This solution is due to Mairson [Mai77].

2. Instead of just removing pf from S, remove $p^e f$ for $e = 1, 2, \ldots$ until $p^e f$ is larger than n. This works because those elements of $F(p)$ which are removed from S prematurely must be multiples of p. This solution is due to Gries and Misra [GM78].

3. Find the largest element in $F(p)$ and work down. This works since the elements removed are $pf > f$, so those elements in $F(p)$ which are removed are removed *after* they are used. This is due to Pritchard [Pri81], and is the method we use.

This leads to the algorithm below.

```
Initialize:
    initialize;
Main Loop:
    $p := 2$;
    while $p^2 \leq n$ do
        Remove multiples:
            Find the largest element of $F(p)$:
                $f := p$;
                while $p \cdot \text{next}(S,f) \leq n$ do $f := \text{next}(S,f)$;
            Loop down through the values of $f$:
                repeat
                    remove($S$,$pf$);
                    $f := \text{prev}(S,f)$;
                until $f < p$;
        Find the next prime:
            $p := \text{next}(S,p)$;
    end while;
```

As observed earlier, the space used by this algorithm is $O(n \log n)$ bits. The time, though, is $O(n)$ because each composite integer is crossed off exactly once, and the number of operations performed for each removal is $O(1)$.

5 Segmentation.

Perhaps the most practical improvement to the Sieve of Eratosthenes is the idea of *segmentation.* Bays and Hudson [BH77] describe the method in detail, and Brent [Bre73] also used this method.

The idea is to not sieve the entire interval from 2 to n at once. Instead, sieve intervals of length Δ one at a time, and after n/Δ intervals have been sieved, we have found all the primes. So, the basic steps are

```
1.  Find all the primes up to $\sqrt{n}$

2.  $l := \sqrt{n}$;
    While $l < n$ do:
        Sieve the interval [ $l+1 \ldots l+\Delta$ ];
        $l := l + \Delta$;
```

Step 1 can be done with trial division or one of the methods described in the previous sections. In Step 2, what I mean by "sieve the interval" is, for each prime $p \le \sqrt{n}$, cross off the multiples of p in the interval currently being sieved. The first number to cross off for the prime p in the interval $[l+1, l+\Delta]$ is $l + p - (l \bmod p)$. Successive multiples of p are found by just adding p.

Clearly the cost of step 1 is $o(n)$. The space needed is at most $\sum_{p\le\sqrt{n}} \log p = O(\sqrt{n})$ using (1). The cost of step 2 is roughly

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

The space used is $O(\sqrt{n}+\Delta)$. This suggests choosing $\Delta = \sqrt{n}$, which gives us the same time complexity as the Sieve of Eratosthenes. The total space used is $O(\sqrt{n})$, as promised.

6 Wheels.

Before we can go on to describe the sublinear algorithm and the linear segmented algorithm, we require the concept of a *wheel.* The idea is simple and elegant, and it is useful in algorithms other than prime number sieves.

Let p_i be the ith prime, with $p_1 = 2$. Let us define

$$\begin{aligned} M_k &= \prod_{i=1}^{k} p_i \\ W_k &= \{x : 0 \le x < M_k,\ \gcd(x, M_k) = 1\} \\ W_k(n) &= \{x \le n : \gcd(x, M_k) = 1\} \end{aligned}$$

In other words, W_k denotes the integers between 0 and $M_k - 1$ which are relatively prime to the first k primes. We will call W_k the *kth wheel*, and $W_k(n)$ the kth wheel extended to n.

Wheels have several useful properties which we will need later.

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

(See (1), (3)).

We will now describe a data structure for wheels. In a constant number of operations, it will tell us whether or not an integer is in the extended wheel (is relatively prime to M_k) and what the next largest element in the extended wheel is.

```
var $wk$ : array [0..$M_k - 1$] of integer;
```

For each x, $0 \le x < M_k$, $wk[x] = 0$ if $\gcd(x, M_k) \ne 1$, and $wk[x] = g$ if $\gcd(x, M_k) = 1$, where $g = y - x$ and y is the next largest element in the kth extended wheel. Notice that $wk[M_k - 1] = 2$. Let us see some examples:

$k = 1$, $M_1 = 2$:

x	0	1
$w1[x]$	0	2

$k = 2$, $M_2 = 2 \cdot 3 = 6$:

x	0	1	2	3	4	5
$w2[x]$	0	4	0	0	0	2

$k = 3$, $M_3 = 2 \cdot 3 \cdot 5 = 30$ (non-zero entries only)

x	1	7	11	13	17	19	23	29
$w3[x]$	6	4	2	4	2	4	6	2

To compute this data structure, we execute the following steps.

1. Use trial division to find the first k primes. Compute M_k.
2. Sieve the array wk so that $wk[x] = 0$ or 1 depending on whether $\gcd(x, M_k)$ is 1. The 1 entries will be changed in the next step.
3. Set $wk[M_k - 1] = 2$, and then make a pass over wk starting from $M_k - 1$ going down. Save the previous x such that $wk[x]$ was nonzero. When the next smallest nonzero entry is found, store the difference. As a check, the value of $wk[1]$ should be $p_{k+1} - 1$.

It is not hard to see that the total cost for this is at most $O(M_k \log\log p_k) = O(M_k \log\log\log M_k)$ steps. Alternatively, a gcd algorithm could be used instead of a sieve for a complexity of $O(M_k \log^2 M_k) = O(M_k \cdot k^2)$ bit operations.

If we set $d = 1$ and then repeat the operation $d{:=}d + wk[d \bmod M_k]$, d will take on all the values in the extended kth wheel.

Also notice that if we want the previous element in the extended wheel, we can get this using the wheel's symmetry; the element previous to x is $x - wk[M_k - (x \bmod M_k)]$. We will not take advantage of this property here.

7 A Sublinear Algorithm.

We are now ready to convert our linear algorithm from section 4 to a sublinear algorithm using wheels. The technique we describe below will work for almost any linear algorithm, and certainly for most of those described in Pritchard's article [Pri87]. Our algorithm consists of the following steps.

1. Choose k as large as possible such that $M_k \leq \sqrt{n}$. Find the first k primes and compute $wk[x]$ for all $x < M_k$.
2. Initialize the set S to the kth wheel extended to n. Recall S is implemented as a doubly linked list using arrays as described in section 4.

```
$d := 1$;
repeat
    $S := S \cup \{d\}$;
    $d := d + wk[d \bmod M_k]$;
until $d > n$;
```

3. Run the linear algorithm starting with $p :=$ next(S,1), which is just p_{k+1}.
4. Output $S - \{1\}$ and the first k primes.

Notice that S is initialized to contain only $O(n/\log\log n)$ elements, so the phase of the algorithm which uses the linear algorithm uses only $O(n/\log\log n)$ arithmetic operations. Clearly this dominates the running time, so we have a sublinear algorithm.

Pritchard [Pri81] describes how to reduce the operations to only additions and how to reduce the storage requirements. The main tricks for reducing the storage include:

- using differences instead of absolute addresses in the *prev* and *next* fields of s.
- using mapping functions to reduce the number of nodes in s to those only in the extended wheel (as we described it, those not in the extended wheel are just wasted space), and
- using a bit vector for the integers in the range n/p_k to n, since none of these integers are found in the sets $F(p)$, making the linked list structure on this interval unnecessary.

In practice, if space is a premium, then a segmented method should be used. So these tricks are, for the most part, only of theoretical interest.

8 A Linear Segmented Wheel Sieve.

Unfortunately, there is no apparent way to segment the linear and sublinear algorithms presented in earlier sections, because they use a linked list representation for S (and hence for $F(p)$). There is, however, a way to create a linear algorithm using segments and wheels. This algorithm, called the segmented wheel sieve, is due to Pritchard [Pri83].

The idea is to generate the set $F(p)$ using the kth wheel, where we choose k so that M_k is about $\sqrt{n}$ as before. Then, when sieving each interval, we only sieve on the primes p with $p_k < p \leq \sqrt{n}$.

Only those integers in the extended wheel which are not crossed off in this sieve process are prime. To determine where to start "crossing off" for each prime p, we simply store the last value of f used on the previous interval and use the wk array to find the next value for f from the extended wheel. This gives us the following modifications to our segmented algorithm from in section 5.

Preprocessing:

1. Find the first k primes such that $M_k \approx \sqrt{n}$ and compute the array $wk[x]$.
2. Find all the primes below $\sqrt{n}$.
3. For each prime $p_i \leq \sqrt{n}$, $i > k$, initialize $factor[i] := p_i$.

Sieving the Interval $[l+1,\ldots,l+\Delta]$:

1. Initialize the array of length Δ to all zeros. Mark 1's on the array for elements in $W_k(n)$.
2. For each prime p_i, $p_k < p_i \leq \sqrt{n}$ do

```
        $f := factor[i]$;
        while $pf \leq l + \Delta$ do
            set position $pf$ to 0
            $f := f + wk[f \bmod M_k]$;
        end while
        $factor[i] := f$;
```

The total number of operations for this algorithm will be roughly

$$\sum_{p_k<p\leq\sqrt{n}} \frac{1}{p}\cdot\frac{n}{\log\log n} \quad = \quad O(n).$$

The total space used is: $O(\sqrt{n})$ for the array wk, plus $O(\sqrt{n})$ for the primes below $\sqrt{n}$, plus $\Delta = O(\sqrt{n})$ for the interval. This gives a total of $O(\sqrt{n})$ bits.

Again, Pritchard shows how to remove the multiplications, though such techniques are of doubtful practical value.

9 Some Remarks.

I will make a few remarks.

- The Sieve of Eratosthenes, both the classical and segmented versions, can be easily modified to factor all the integers up to n. The only change is to store a list of prime divisors for each position in the array s instead of just one bit. For this to work, note that $F(p)$ must start with 1, and p is added to the list of every pth element in s.

- Luo's recent algorithm [Luo89], is simply the Sieve of Eratosthenes with the 2nd wheel. Its asymptotic time and space complexity are the same as that of Eratosthenes's sieve, though with a smaller constant of proportionality.

- Bengelloun's incremental sieve [Ben86] uses a clever loop structure to generate the values of f and the primes p. For an excellent description of this algorithm, see also Pritchard's article [Pri87].

- In those algorithms where M_k is chosen near $\sqrt{n}$, notice that what we really want is $\log\log M_k = \Theta(\log\log n)$. So, in fact, we could make $M_k = \exp[O(\log^d n)]$ for $0 < d < 1$. For example we could choose k so that p_k is about $\sqrt{\log n}$ giving $d = 1/2$. In practice, choosing p_k equal to 7 or 11 is close to optimal. Anything too much larger may make the array wk too large to be practical, since it is referenced frequently and should fit in main memory.

- The kth extended wheel useful in trial division. First reduce n, the number to be factored, modulo M_k to see if it is relatively prime to the first k primes. Then generate wheel elements for use as trial divisors. This is a very nice compromise between trial division by all numbers and only primes.

- For a parallel implementation, we can use the segmented sieve and assign each processor intervals of length Δ. This is probably optimal for up to $O(\sqrt{n})$ processors.

10 Implementation Results.

The bottom line in designing efficient algorithms is to make them fast in practice. With this in mind, I implemented one example of each type of algorithm discussed in this paper: The Sieve of Eratosthenes, Gries and Misra's linear algorithm, Pritchard's sublinear algorithm (1), Bays and Hudson's segmented sieve, and Pritchard's linear segmented wheel sieve (2). The results appear below.

I had each algorithm find all the primes up to n for $n = 10^d$, $d = 2, 3, \ldots, 9$. $\pi(n)$ gives the number of primes up to n. Under each algorithm name is a list of times, in CPU seconds. These columns give the time needed for that algorithm to find all the primes for that value of n.

The first three algorithms were only run on values of n up to 10^6, for lack of space. The limit of 10^9 for the others was chosen because the largest integer representable in one machine word is around 2×10^9; I wished to avoid extended precision arithmetic.

I wrote the routines in Pascal, and the machine I used was a DEC VAXstation 3200 running Ultrix.

The results below suggest that to find all the primes below n for n up to about one million, one should use the sieve of Eratosthenes; the faster methods are more difficult to implement and the added complexity is not really justified. If n is much larger, either of the segmented methods is good; Pritchard's wheel sieve is a substantial improvement over Bays and Hudson's algorithm, but even for $n = 10^9$ the difference between the two is only about 1.25 hours.

Notice that Gries and Misra's algorithm is not as fast as Eratosthenes's, in spite of the fact that Gries and Misra's algorithm has a faster asymptotic running time (in arithmetic operations). This discrepency can be attributed to the constant of proportionality in the running times; Gries and Misra's algorithm uses a linked list as the primary data structure, and each step requires several pointer manipulations, whereas the sieve of Eratosthenes works with a simple array.

n	$\pi(n)$	Eratosthenes	Gries-Misra	Pritchard-1	Bays-Hudson	Pritchard-2
100	25	0.001	0.001	0.001	0.001	0.001
10^3	168	0.008	0.013	0.005	0.012	0.010
10^4	1229	0.091	0.142	0.044	0.116	0.095
10^5	9592	0.980	1.498	0.450	1.142	0.816
10^6	78498	10.180	15.730	4.680	11.900	8.380
10^7	664579	—	—	—	115.160	77.810
10^8	5761455	—	—	—	1149.520	795.730
10^9	50847534	—	—	—	12074.410	7666.480

Acknowledgements

I would like to thank my advisor Eric Bach and Jeffrey Shallit, who was my instructor for CS 812 at the University of Wisconsin-Madison in the fall of 1989, where this paper was a term project.