

A Space-Efficient Fast Prime Number Sieve*

Brian Dunten Julie Jones Jonathan Sorenson
Department of Mathematics and Computer Science
Butler University
4600 Sunset Avenue
Indianapolis, IN 46208
USA
sorenson@butler.edu

Information Processing Letters 59 (1996) 79–84

Abstract

We present a new algorithm that finds all primes up to n using at most $O(n/\log \log n)$ arithmetic operations and $O(n/(\log n \log \log n))$ space. This algorithm is an improvement of a linear prime number sieve due to Pritchard. Our new algorithm matches the running time of the best previous prime number sieve, but uses less space by a factor of $\Theta(\log n)$.

In addition, we present the results of our implementations of most known prime number sieves.

Key Words: prime number sieve, sieve of Eratosthenes, number theoretic algorithms, analysis of algorithms, design of algorithms

1 Introduction

A *prime number sieve* is an algorithm that constructs a list of primes up to a given bound n . Eratosthenes is believed to be the inventor of the first prime number sieve over 2000 years ago. His sieve requires $\Theta(n \log \log n)$ arithmetic operations and $\Theta(n)$ space (see [4]). Currently, the fastest sieve is Pritchard's dynamic wheel sieve [8, 9]; it requires only $\Theta(n/\log \log n)$ operations and $\Theta(n/\log \log n)$ space. Pritchard's segmented wheel sieve is the most efficient in its use of space [10]; it requires $\Theta(n)$ operations and only $\Theta(\sqrt{n}/\log \log n)$ bits of space. Thus, when moving from a $\Theta(n)$ time sieve to a $\theta(n/\log \log n)$ time sieve, in improvement by a factor of $\theta(\log \log n)$, the price is a very large increase in space, namely a factor of $\theta(\sqrt{n})$.

*This research was sponsored by NSF grant CCR-9204414 and a grant from the Holcomb Research Institute. A preliminary version of this paper was presented by Julie Jones at the Butler Undergraduate Research Conference, April 15, 1994.

In this paper, we present a new space-efficient sublinear sieve. It matches the dynamic wheel sieve in speed, using only $\Theta(n/\log \log n)$ operations, while using only $\Theta(n/(\log n \log \log n))$ space, an improvement over the dynamic wheel sieve by a factor of $\Theta(\log n)$. Thus, this algorithm partially closes the complexity gap between fast sieves and space-efficient sieves. Our result is obtained by combining a wheel with segmenting in a new way. After discussing some preliminaries in Section 2, we present our algorithm in Section 3.

In order to determine whether our new algorithm is practical, we performed timing experiments comparing it to most known prime number sieves. We present the results from these experiments in Section 4.

2 Preliminaries

We begin with a discussion of our model of computation, followed by some definitions and a review of the *wheel* data structure.

Our model of computation for prime number sieves is a RAM with a potentially infinite, direct access memory. For simplicity, all basic arithmetic operations (including $+$, $-$, \times , $/$, comparisons, assignment, array indexing, branching) take constant time for $O(\log n)$ -bit integers, where n is the input. Memory may be addressed either at the bit level or in $O(\log n)$ -bit words. We measure the space used by an algorithm by counting the number of bits. Thus, under this model, an $O(n)$ time algorithm could touch every bit of an array of n words or $n \log n$ bits if using word access, but only $n/\log n$ words or n bits if using bit access.

We do not bother to measure the space used by the output of a prime number sieve, since it is the same for all algorithms.

The disparity in granularity between our measurement of time and space complexity requires some justification.

Observe that, if we were to measure time at the bit complexity level, an addition of two integers $\leq n$ would take $O(\log n)$ time and multiplication would take $O(\log^2 n)$ time (using classical methods [7]). However, by precomputing a table of all products of integers $\leq n^{1/4}$, say, one could effectively multiply in $O(\log n)$ bit operations through judicious use of table lookup. Division can be dealt with similarly. (See [13, Lemma 4.1].) For prime number sieves, such a construction is useful in theory but generally unnecessary in practice. We duplicate the theoretical effect of this construction without the associated messy details by assigning all operations unit cost. In any case, on many computers, multiply is implemented in hardware and so is effectively a constant time operation. (On the other hand, Pritchard has investigated practical and effective means for removing multiplications from sieves; see, for example, [8, 12].)

We measure space at the bit level, however, because the $\Theta(\log n)$ factor involved cannot be dismissed theoretically. In addition, this factor corresponds to the word size of a machine in practice, and so has a very real affect on the performance of prime number sieves, especially for larger values of n .

An integer $p > 1$ is prime if its only divisors are 1 and p . We always have p and q denote primes, and we write p_i for the i th prime (so $p_1 = 2$). For $x > 0$, let $\pi(x)$ denote the number of primes $\leq x$. For integers a, b let $\gcd(a, b)$ be the largest integer that divides both a and b .

We say a and b are *relatively prime* if $\gcd(a, b) = 1$. For a positive integer x let $\phi(x)$ denote the number of integers between 0 and x that are relatively prime to x (this is Euler’s totient function). For an introduction to number theory, see [6].

For a positive integer x , we define the *least prime factor* of x , $\text{lpf}(x)$, to be the smallest prime p such that $p \mid x$ (p divides x). Thus $\text{lpf}(p) = p$ for any prime p , and if x is composite, $\text{lpf}(x) \leq \sqrt{x}$.

Let $k > 0$ be an integer, and let $m = m(k) = p_1 p_2 \cdots p_k$. The k th wheel, \mathcal{W}_k , is the set of integers from 1 to m that are relatively prime to m (the reduced residue system modulo m). In prime number sieves, \mathcal{W}_k is used to “presift” the integers up to n so that all multiples of the first k primes can be quickly discarded. Pritchard pioneered the use of a wheel where m grows with n . We use a *static* wheel; once constructed, it is not changed, though its size will depend on n (see [10, 14]). In contrast, a *dynamic* wheel grows as the algorithm progresses (see [8, 9]).

In this paper we will take a concrete approach; we define the k th wheel, $W_k[\]$ to be an array of records of length m (indexed by $0 \dots m - 1$) where each record has four integer fields:

$W_k[x].rp$	is 1 if $\gcd(x, m) = 1$ and 0 otherwise,
$W_k[x].dist$	is $d = y - x$ where $y > x$ is smallest with $\gcd(y, m) = 1$,
$W_k[x].pos$	is the number of positive integers $y \leq x$ with $\gcd(y, m) = 1$, and
$W_k[x].inv$	is -1 if $x = 0$, 0 if $x \geq \phi(m)$, and the x th integer y with $\gcd(y, m) = 1$ otherwise.

We say that m is the *size* of the k th wheel. For examples, see [14, §2].

Lemma 2.1 $W_k[\]$ can be constructed in $O(m)$ time.

Proof: Almost any sublinear sieve can be modified to compute the rp field in $O(m)$ time. Then the rest can easily be done in $O(m)$ time. \square

A sieve is said to be *segmented* if it views the integers from 1 to n as $\lceil n/\Delta \rceil$ consecutive intervals or segments, each of length Δ (except perhaps the last). In a segmented sieve, all the primes in the current segment are found before moving on to the next segment. The idea is that the space occupied by the current segment can be reused for the next one. For more on segmented sieves, see [2, 10].

3 A New Space-Efficient Sublinear Sieve

In this section, we present our new sublinear, space efficient prime number sieve. Since this algorithm is based on a linear sieve due to Pritchard [11], we begin by reviewing this algorithm. We then proceed by showing how to employ a wheel with this sieve, and we conclude by segmenting the sieve.

Observe that every positive composite integer x has a unique representation in the form $x = pf$ where $p = \text{lpf}(x)$ and $f = x/p$. Also note that $\text{lpf}(f) \geq p$.

Pritchard’s linear sieve constructs each composite $x \leq n$ in the form pf exactly once by first looping over all possible values for f , and for each f , looping over primes p where pf is the unique representation for some x as mentioned above.

The details are presented as Algorithm A below. For additional details, see Pritchard [11].

Algorithm A (Pritchard[11])

```

{Initialization}
  Find the primes  $p_j$  up to  $\lfloor \sqrt{n} \rfloor$ ;
  For  $i := 2$  to  $n$  do  $s[i] := 1$ ;
   $s[1] := 0$ ;
{Main Loop}
  For  $f := 2$  to  $\lfloor n/2 \rfloor$  do
    {For each prime  $p \leq \text{lpf}(f)$  with  $pf \leq n$  set  $s[pf] := 0$ : }
       $j := 1$ ;
      Repeat
         $s[p_j f] := 0$ ;  $q := p_j$ ;  $j := j + 1$ ;
      Until  $q \mid f$  or  $p_j f > n$ ;
{Output Primes}
  For  $i := 1$  to  $n$  do
    If  $s[i] = 1$  then output( $i$ );

```

This algorithm requires only $\Theta(n)$ operations and $\Theta(n)$ space.

Next, we demonstrate how to improve Algorithm A with a wheel. Essentially, the idea is to only generate composite x in the form pf for x relatively prime to the first k primes. This is done by only using primes $p > p_k$ and by using a wheel to generate all integers f with $p_{k+1} \leq f \leq n/p_{k+1}$.

The details are presented in Algorithm B below. (See also [14].)

Algorithm B

```

{Initialization}
  Find the primes  $p_j$  up to  $\lfloor \sqrt{n} \rfloor$ ;
  Find  $k$  such that  $(\log n)/4 \leq p_k \leq (\log n)/2$ ;
  Compute the  $k$ th wheel  $W_k[\ ]$ ;
  For  $i := 1$  to  $n$  step  $W_k[i \bmod m].dist$  do  $s[i] := 1$ ;
   $s[1] := 0$ ;
{Main Loop}
  For  $f := p_{k+1}$  to  $\lfloor n/p_{k+1} \rfloor$  step  $W_k[f \bmod m].dist$  do
    {For each prime  $p$  where  $p_k < p \leq \text{lpf}(f)$  and  $pf \leq n$  set  $s[pf] := 0$  }
       $j := k + 1$ ;
      Repeat
         $s[p_j f] := 0$ ;  $q := p_j$ ;  $j := j + 1$ ;
      Until  $q \mid f$  or  $p_j f > n$ ;
{Output Primes}
  For  $j := 1$  to  $k$  do Output( $p_j$ );
  For  $i := p_{k+1}$  to  $n$  step  $W_k[i \bmod m].dist$  do
    If  $s[i] = 1$  then output( $i$ );

```

Observe that this algorithm only requires $\Theta(n/\log \log n)$ operations. As written, it uses $\Theta(n)$ space, but using techniques from [14], this can be reduced to $\Theta(n/\log \log n)$. The idea here is to use the *pos* and *inv* fields of $W_k[]$ to implement a bijection between the sets $\{1 \leq x \leq n \mid \gcd(x, m) = 1\}$ and the integers in the interval $[1, n(\phi(m)/m) + O(1)]$ such that the bijection is computable in constant time. This allows for the use of an array of $n(\phi(m)/m) + O(1)$ bits to store the contents of $s[]$ with no significant effect on the asymptotic running time of the algorithm.

The main ideas behind segmenting Algorithm B are similar to those employed by Pritchard [10]. The integers up to n are divided into consecutive intervals of length Δ . Then the main loop from Algorithm B is applied to each of these intervals. Doing this without incurring additional complexity poses some problems.

First, on each interval we must figure out which values of f to use. There are two possible reasons that a particular value for f should not be used on the current interval:

- (1) The value for f may be out of range. For example, f may be larger than the right end point of the interval.
- (2) During the sieving of an earlier interval, a prime p_j was found such that $p_j \mid f$.

Let *low* and *high* be integers such that the current interval is $[low + 1, high]$. For any integer x in this interval, we must have $\text{lpf}(x)f > low$. This, plus the fact that $\text{lpf}(x) \leq f$ gives $f > \lfloor \sqrt{low} \rfloor$. We can use the *dist* field of W_k to find the smallest f above this bound relatively prime to m . This gives the lower bound for f . Since we are using the k th wheel, the largest value for f on the current interval is $high/p_{k+1}$. This solves (1).

To solve (2), we use a bit string *fok[]* where we initialize $fok[f] := 1$ for all possible values of f , and then set $fok[f] := 0$ when a prime p_j is found such that $p_j \mid f$. *fok[]* is checked before a value of f is used on an interval.

Second, for each interval and for each value of f , we must determine the smallest prime p_j to use for the inner loop. But this is just the smallest prime larger than $\max\{p_k, low/f\}$. In order to find this prime, we need a reverse index $r[]$ where $r[x]$ is the index of the smallest prime $\geq x$ for all x up to \sqrt{n} . $r[]$ is easily computed as part of initialization.

Pulling all of this together, we obtain Algorithm C, which is presented below. Note that we choose $\Delta = \lfloor n/p_{k+1} \rfloor + 1$. This choice is justified later.

Algorithm C

```

{Initialization}
  Find the primes  $p_j$  up to  $\lfloor \sqrt{n} \rfloor$ ;
  Compute a reverse index  $r[]$  such that for all  $x \leq \lfloor \sqrt{n} \rfloor$ ,
     $r[x] = j$  if  $p_{j-1} < x \leq p_j$  for  $x > 2$ , and  $r[x] = 1$  for  $x \leq 2$ ;
  Find  $k$  such that  $(\log n)/4 \leq p_k \leq (\log n)/2$ ;
  Compute the  $k$ th wheel  $W_k[]$ ;
   $\Delta := \lfloor n/p_{k+1} \rfloor + 1$ ;
  For  $i := 1$  to  $\Delta$  step  $W_k[i \bmod m].dist$  do  $fok[i] := 1$ ;
{Output the Primes up to  $\lfloor \sqrt{n} \rfloor$ }
  For  $j := 1$  to  $\pi(\lfloor \sqrt{n} \rfloor)$  do output( $p_j$ );
{Loop over Segments}

```

For $low := \lfloor \sqrt{n} \rfloor$ to $n - 1$ step Δ do
 $high := \min\{low + \Delta, n\}$; $\{The\ interval\ is\ [low + 1, high]\}$
SieveSegment;

Procedure SieveSegment

$\{Initialize\ s[\]\}$
 $i_{\min} := low + W_k[low \bmod m].dist$;
For $i := i_{\min}$ to $high$ step $W_k[i \bmod m].dist$ do $s[i - low] := 1$;
 $\{Main\ Loop\}$
 $temp := \lfloor \sqrt{low} \rfloor$;
 $f_{\min} := temp + W_k[temp \bmod m].dist$;
For $f := f_{\min}$ to $high/p_{k+1}$ step $W_k[f \bmod m].dist$ do
If $fok[f] = 1$ then
 $j := \max\{k + 1, r[\lfloor low/f \rfloor]\}$;
While $p_j f \leq high$ and $fok[f] = 1$ and $p_j \leq f$ do
 $s[p_j f - low] := 0$;
If $p_j \mid f$ then $fok[f] := 0$;
 $j := j + 1$;
 $\{Output\ the\ Primes\ in\ the\ Interval\}$
For $i := i_{\min}$ to $high$ step $W_k[i \bmod m].dist$ do
If $s[i - low] = 1$ then output(i);

In the following theorem, we assume that Algorithm C is employing the space-saving techniques mentioned in [14].

Theorem 3.1 *Algorithm C computes the primes up to n using $O(n/\log \log n)$ arithmetic operations and $O(n/(\log n \log \log n))$ space.*

Proof: We analyze the running time by phase.

- *Initialization.*

Finding the primes up to \sqrt{n} , computing the reverse index, and computing the k th wheel takes $O(n^{1/2+o(1)})$ time. Initializing $fok[\]$ takes $O(n/(p_{k+1} \log \log n))$ time.

- *Output the Primes up to $\lfloor \sqrt{n} \rfloor$.*

This takes $O(\sqrt{n})$ time.

- *Loop over Segments.*

Let $S(n)$ denote the time taken in the SieveSegment procedure. Then this phase takes time $O(S(n)(n/\Delta))$.

We now bound $S(n)$.

- *Initialize $s[\]$ and Output the Primes in the Interval.*

This takes $O(\Delta/\log \log n)$ time for both phases.

– *Main Loop.*

The outer for-loop executes at most $O(n/(p_{k+1} \log \log n))$ times, once for each value of f relatively prime to m . The inner while-loop executes, in total, no more than once for each composite integer in the current interval that is relatively prime to m , which is no more than $O(\Delta/\log \log n)$ times.

Thus $S(n) = O([n/p_{k+1} + \Delta]/\log \log n)$.

Combining this information, we obtain that the running time of Algorithm C is bounded by

$$O(n^{1/2+o(1)}) + O(n/(p_{k+1} \log \log n)) + O([n/\Delta][n/p_{k+1} + \Delta]/\log \log n).$$

Recall that, by our choice for k , $p_{k+1} = \Theta(\log n)$. This simplifies our running time to

$$O([1 + n/(p_{k+1} \Delta)][n/\log \log n]).$$

From this, we see that choosing $\Delta = \Theta(n/p_{k+1})$ completes the proof of the running time.

The total space used is

$$O(\Delta/\log \log n + n/(p_{k+1} \log \log n)) = O(n/(\log n \log \log n)).$$

The correctness of the algorithm follows from that of Algorithm B and the discussion above. \square

4 Implementation Results

In this section, we present the results of our timing experiments (see Table 1).

Any timing experiments depend heavily on the machines and compilers used and the programmers involved. We urge the reader to keep this fact in mind when drawing conclusions based on our data.

Below we list the sieves that we chose to implement. We gave each sieve a label; the labels are used to identify the algorithms in Table 1. Algorithms whose primary data structure is a bit string also appear with one or more of the following three label suffixes: B, W, or S. The “plain” version of these algorithms uses a character array to represent the bit string. When a B suffix appears, 8 bits are stored in each character instead of only one. This *bit-packing* incurs a noticeable performance cost. A W denotes the use of a static wheel with maximal $m \leq \sqrt{n}$. An S indicates that the sieve is segmented.

Beng	Bengelloun's incremental sieve [3]
DWS	Pritchard's dynamic wheel sieve [8, 9]
Erat	The sieve of Eratosthenes; Erat-S is Bays and Hudson's segmented version of this sieve [2], and Erat-SW is Pritchard's segmented wheel sieve [10].
GM	The sieve of Gries and Misra [5] (see also [11]).
GP	The linear sieve originally due to Gale and Pratt; it uses a bit string as its primary data structure. We used the array representation for F as described by Pritchard [11]. See also Barstow [1].
New-SW	Algorithm C from Section 3.
Prit 3.1	Algorithm 3.1 from Pritchard [11].
Prit 3.2	Algorithm 3.2 from Pritchard [11].
Prit 3.3	Algorithm 3.3 from Pritchard [11]; this is Algorithm A, and Prit 3.3-W is Algorithm B.

All of these algorithms were implemented in ANSI C.

We implemented the various sieves on three different platforms: a CompuAdd 433E PC (486/33 chip), an HP 9000 series 715/75 workstation, and a VAX 6000-610 mainframe. The results for the three platforms were not significantly different, so we present here only the HP workstation results. We used the Gnu C compiler (`gcc 2.6.3`) with default level optimization.

In Table 1, the algorithms are listed in the left column, with times given in CPU seconds for each value of the input n . Dashes indicate that the algorithm required more memory than was available.

We were not surprised to discover that our new algorithm does not favor well in comparison to the existing sieves, although the algorithm is in fact usable. The best sieves to use in practice appear to be either DWS or Erat-SW, depending on the value of n .

Acknowledgments

Special thanks to Paul Pritchard for his many helpful comments on an earlier draft of this paper, and to David Cole for his comments on Algorithm C. We also wish to thank the anonymous referees for their many suggestions.

References

- [1] D. R. Barstow. An experiment in knowledge-based automatic programming. *Artificial Intelligence*, 12:73–119, 1979.
- [2] C. Bays and R. Hudson. The segmented sieve of Eratosthenes and primes in arithmetic progressions to 10^{12} . *BIT*, 17:121–127, 1977.
- [3] S. Bengelloun. An incremental primal sieve. *Acta Informatica*, 23(2):119–125, 1986.
- [4] 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.

Table 1: Running Times in CPU Seconds

<i>Algorithm</i>	<i>Input</i>						
	10^3	10^4	10^5	10^6	10^7	10^8	10^9
Beng	0.0015	0.0156	0.172	1.95	–	–	–
DWS	0.0004	0.0036	0.042	0.63	–	–	–
GM	0.0009	0.0092	0.116	1.49	–	–	–
Prit 3.1	0.0010	0.0106	0.140	1.67	–	–	–
Erat	0.0011	0.0112	0.118	1.68	–	–	–
GP	0.0023	0.0226	0.242	3.04	–	–	–
Prit 3.2	0.0016	0.0164	0.172	1.94	–	–	–
Prit 3.3	0.0023	0.0238	0.246	2.73	–	–	–
Erat-W	0.0011	0.0114	0.100	1.20	–	–	–
GP-W	0.0012	0.0114	0.096	1.11	–	–	–
Prit 3.2-W	0.0012	0.0122	0.104	1.18	–	–	–
Prit 3.3-W	0.0012	0.0116	0.098	1.10	–	–	–
Erat-B	0.0018	0.0192	0.206	2.16	24.87	–	–
GP-B	0.0030	0.0292	0.298	3.02	36.54	–	–
Prit 3.2-B	0.0021	0.0216	0.222	2.37	24.78	–	–
Prit 3.3-B	0.0029	0.0298	0.308	3.16	32.98	–	–
Erat-WB	0.0012	0.0128	0.122	1.24	12.26	–	–
GP-WB	0.0013	0.0128	0.106	1.11	11.00	–	–
Prit 3.2-WB	0.0013	0.0132	0.114	1.19	11.46	–	–
Prit 3.3-WB	0.0013	0.0132	0.110	1.15	11.00	–	–
New-SW	0.0017	0.0170	0.146	1.48	14.76	153.31	–
New-SWB	0.0019	0.0190	0.162	1.64	15.07	158.61	1555.39
Erat-S	0.0020	0.0168	0.160	1.52	14.71	142.10	1440.66
Erat-SW	0.0014	0.0136	0.122	1.28	12.23	125.15	1291.15

- [5] D. Gries and J. Misra. A linear sieve algorithm for finding prime numbers. *Communications of the ACM*, 21(12):999–1003, 1978.
- [6] G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers*. Oxford University Press, 5th edition, 1979.
- [7] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley, Reading, Mass., 2nd edition, 1981.
- [8] P. Pritchard. A sublinear additive sieve for finding prime numbers. *Communications of the ACM*, 24(1):18–23,772, 1981.
- [9] P. Pritchard. Explaining the wheel sieve. *Acta Informatica*, 17:477–485, 1982.

- [10] P. Pritchard. Fast compact prime number sieves (among others). *Journal of Algorithms*, 4:332–344, 1983.
- [11] P. Pritchard. Linear prime-number sieves: A family tree. *Science of Computer Programming*, 9:17–35, 1987.
- [12] P. Pritchard. Improved incremental prime number sieves. In L. M. Adleman and M.-D. Huang, editors, *First International Algorithmic Number Theory Symposium (ANTS-1)*, pages 280–288, May 1994. LNCS 877.
- [13] J. Sorenson. Two fast GCD algorithms. *Journal of Algorithms*, 16:110–144, 1994.
- [14] J. Sorenson and I. Parberry. Two fast parallel prime number sieves. *Information and Computation*, 144(1):115–130, 1994.