Trading Time for Space in Prime Number Sieves

Jonathan P. Sorenson*

Department of Mathematics and Computer Science, Butler University, 4600 Sunset Avenue, Indianapolis, Indiana 46208, USA, sorenson@butler.edu, http://www.butler.edu/~sorenson/

Abstract. A prime number sieve is an algorithm that finds the primes up to a bound n. We present four new prime number sieves. Each of these sieves gives new space complexity bounds for certain ranges of running times. In particular, we give a linear time sieve that uses only $O(\sqrt{n}/(\log\log n)^2)$ bits of space, an $O_l(n/\log\log n)$ time sieve that uses $O(n/((\log n)^l \log\log n))$ bits of space, where $l > 1$ is constant, and two super-linear time sieves that use very little space.

1 Introduction

A prime number sieve is an algorithm that finds all prime numbers up to a bound n. In this paper we present four new prime number sieves, three of which accept a parameter to control their use of time versus space.

The fastest known prime number sieve is the dynamic wheel sieve of Pritchard [11], which uses $O(n/\log\log n)$ arithmetic operations and $O(n/\log\log n)$ bits of space. Dunten, Jones, and Sorenson [6] gave an algorithm with the same asymptotic running time, while using only $O(n/(\log n \log\log n))$ bits of space. Pritchard also invented a segmented wheel-based sieve that requires $O(n)$ operations and only $O(\sqrt{n}/\log\log n)$ bits of space [12]. This last sieve is more practical for larger values of n, because space becomes a serious concern when n exceeds, say, 10^7.

One could also apply a primality test to each integer up to n. If we were to use the Jacobi Sums test, this would take $n(\log n)^{O(\log\log\log n)}$ arithmetic operations and $(\log n)^{O(\log\log\log n)}$ space [1]. If the ERH is valid, we can improve this to $O(n(\log n)^3)$ operations and $O(\log n)$ bits of space [9, 2].

According to Bernstein [5], a method of Atkin uses $O(n/\log\log n)$ operations and $n^{1/2+o(1)}$ bits of space.

In this paper we present four new sieves. All of them give improved complexity bounds for particular combinations of time and space.

1. Let c be a constant with $0 < c \leq 1/2$, and let $\Delta := \Delta(n)$ with $\Delta = n^c$. Our first sieve is a modification of Pritchard's segmented wheel-based sieve combined with trial division. This sieve uses

$$O\left(\frac{n\sqrt{n}}{\Delta \log\log n} + \frac{n\log n}{(\log\log n)^2}\right)$$

* Supported by NSF Grant CCR-9626877

arithmetic operations and $O(\Delta)$ bits of space. If we choose $\Delta = \sqrt{n}/(\log n)^l$ with $1 \leq l$, this sieve gives new complexity bounds of $O(n(\log n)^l/\log\log n)$ time and $O(\sqrt{n}/(\log n)^l)$ bits of space.

2. Let c be a constant with $1/4 \leq c \leq 1/2$, and let $\Delta = n^c$ as above. Our second sieve is a modification of our first sieve. It uses $O(n\sqrt{n}/\Delta + n)$ arithmetic operations and $O(\Delta)$ bits of space. If we choose $\Delta = \sqrt{n}/(\log n)^l$ with $0 < l < 1$, this sieve gives new complexity bounds of $O(n(\log n)^l)$ time and $O(\sqrt{n}/(\log n)^l)$ bits of space.
3. Our third sieve is a modification of Pritchard's segmented wheel-based sieve. The sieve uses $O(n)$ arithmetic operations and

$$O\left(\frac{\sqrt{n}}{(\log\log n)^2}\right)$$

bits of space. This is an improvement in space use over the best previous linear time sieve by a factor proportional to $\log\log n$.

4. Let $B := B(n)$ with $\log n \leq B \leq \sqrt{n}$. Our fourth sieve is a combination of the sieve of Dunten, Jones, and Sorenson and Pritchard's segmented wheel-based sieve. This sieve uses

$$O\left(\frac{n\log(1+\log B/\log\log n)}{\log\log n}\right)$$

arithmetic operations and

$$O\left(\frac{n}{B\log\log n}\right)$$

bits of space. This gives an improved space bound of $O(n/((\log n)^l \log\log n))$ bits for $O(n/\log\log n)$-time sieves for any fixed $l \geq 1$ by choosing $B = (\log n)^l$.

The rest of this paper is organized as follows: In Sect. 2 we review some preliminaries, including the wheel data structure, the use of wheels and segmentation in prime number sieves, and we review Pritchard's segmented wheel-based sieve and the sieve of Dunten, Jones, and Sorenson. In Sects. 3, 4, and 5 we present our four new sieves. In Sect. 6 we present the results of our timing experiments.

2 Preliminaries

In this section we discuss our model of computation and review material on prime number sieves, including wheels and segmentation.

2.1 Model of Computation

Our model of computation is a RAM with a potentially infinite, direct access memory.

If n is the input, then all arithmetic operations on integers of $O(\log n)$ bits are assigned unit cost. This includes $+$, $-$, $\times$, and division with remainder. In addition, comparisons, array indexing, assignment, branching, bit operations, and other basic operations are also assigned unit cost. Memory may be addressed either at the bit level or at the word level, where each machine word is composed of $O(\log n)$ bits.

The space used by an algorithm under our model is counted in bits. Thus, it is possible for an algorithm to touch n bits in only $O(n/\log n)$ time if memory is accessed at the word level. The space used by the output of a prime number sieve (the list of primes up to n) is not counted against the algorithm, as the output is the same for all algorithms discussed here. Note that, on occasion, we will use a sieve as a subroutine, and the primes it generates will be "consumed" as they are generated without being stored. So in this context it makes sense not to charge for the space used to write out this list of primes.

For a justification of this choice of computation model, see [6], where the same model is used.

2.2 Some Number Theory

An integer $p > 1$ is prime if its only divisors are itself and 1. We always have p denote a prime, with p_i denoting the ith prime, so that $p_1 = 2$. For integers a, b let $\gcd(a, b)$ denote the greatest commong divisor of a and b. We say a and b are *relatively prime* if $\gcd(a, b) = 1$. For a positive integer m let $\phi(m)$ be the number of positive integers up to m that are relatively prime to m, with $\phi(1) = 1$ (this is Euler's totient function). The number of primes up to x is given by $\pi(x)$.

We make use of the following estimates. Here $x > 0$, and all sums and products are only over primes.

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

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

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

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

For proofs of the estimates above, see Hardy and Wright [8].

2.3 Two Sieves

Next we review two sieves. Both of these algorithms make use of the fact that every composite integer x can be written uniquely in the form $x = p \cdot f$, where p is the least prime factor of x ($p = \mathrm{lpf}(x)$).

We present our algorithms as C++ code fragments [15]. When we choose to omit details, we do so by utilizing classes and objects, which we will describe only briefly, leaving the details of their implementation to the reader.

Algorithm 2.1: The Sieve of Eratosthenes. Our first sieve, the sieve of Eratosthenes, is well-known. It begins with a bit vector `S` of length n initialized to mostly ones, representing the set $\{2, \ldots, n\}$. Then, as each prime is found, its multiples are removed from the set by changing their corresponding bit positions to zero. Once this is done for all primes up to $\sqrt{n}$, only primes remain in the set represented by `S`.

The following C++ code fragment describes this algorithm. Here `BitVector` is a class that supports standard operations on a bit vector such as clearing and setting bits and testing a bit to see if it is one or zero. (A bit is *set* if it is one, and *clear* if it is zero.)

```
int p,q;
BitVector S(n);
S.setall(); S.clear(1);     // sets all bits except position 1
for(p=2; p<=sqrt(n); p=p+1) // loop over primes p up to sqrt(n)
  if(S[p]==1)               // make sure p is actually prime
    for(q=p*p; q<=n; q=q+p) // loop over multiples q of p
      S.clear(q);           // q is not prime
```

Note that we can rewrite the inner `for`-loop multiplicatively as follows:

```
    for(f=p; p*f<=n; f=f+1) // loop over multiples p*f of p
      S.clear(p*f);         // p*f is not prime
```

We would expect this second form to be less efficient in practice than the first, but some of our sieves are derived from this view of the algorithm. (Warning: in practice, if `n` approaches the word size, `p*f` may overflow giving a false true in the comparison `p*f<=n`. To avoid this problem, use `f<=n/p` instead.)

The running time of the sieve of Eratosthenes depends on the number of times the inner `for`-loop executes. But this is just

$$\sum_{p \le \sqrt{n}} \sum_{f=p}^{\lfloor n/p \rfloor} 1 \le \sum_{p \le \sqrt{n}} \frac{n}{p}$$

which is $O(n \log\log n)$ time by (1). The algorithm uses $O(n)$ space.

Algorithm 2.2: Pritchard's Linear Sieve. Our next sieve is a linear-time sieve devised by Pritchard [13, Algorithm 3.1].

Pritchard's idea was to switch the order of the `for`-loops in the sieve of Eratosthenes so that the outer loop iterates through f-values, while the inner loop iterates through primes p. Here each composite integer $x \le n$ is generated exactly once in the form $x = p \cdot f$ where $p = \mathrm{lpf}(x)$. To determine which primes p

to iterate through for a given value of f, Pritchard observed that when $p = \mathrm{lpf}(x)$, then $p \leq \mathrm{lpf}(f)$. Thus, p must run from 2 to $\mathrm{lpf}(f)$.

This leads to the algorithm given below. Here `PrimeList` is a class that finds and holds primes up to the specified bound, and could be implemented by using the sieve of Eratosthenes.

```
int p,i,f;
BitVector S(n);
S.setall(); S.clear(1); // sets all bits except position 1
PrimeList P(sqrt(n));   // finds the primes up to sqrt(n)
for(f=2; f<=n/2; f=f+1) // loop over f values
  for(i=1; i<=P.length(); i=i+1) // loop over primes<=sqrt(n)
  {
    p=P[i];             // P[i] is the ith prime, with P[1]==2
    if(p>n/f) break;    // if p is too large, get the next f
    S.clear(p*f);       // p*f is composite
    if(f%p==0) break;   // if p|f, p==lpf(f), so get the next f
  }
```

Because each composite integer x is constructed exactly once in the inner `for`-loop, this algorithm takes $O(n)$ time with $O(n)$ bits of space.

Next we look at reducing the running times of both sieves using a wheel.

2.4 Wheels

A *wheel*, as we will use it, is a data structure that encapsulates information about the integers relatively prime to the first k primes. Generally speaking, a wheel can often be used to reduce the running time of a prime number sieve by a factor of $\log\log n$. Pritchard was the first to show how to use a wheel in this way. In this section we will explain the wheel data structure and show how to apply it to the two prime number sieves we described above. We begin with the following definitions:

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

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

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

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

- `W[`x`].rp` is 1 if $x \in \mathcal{W}_k$, and 0 otherwise.
- `W[`x`].dist` is $d = y - x$, where $y > x$ is minimal with $\gcd(y, M_k) = 1$.
- `W[`x`].pos` is $\#\mathcal{W}_k(x)$.
- `W[`x`].inv` is in some sense the inverse of the `pos` field; it is -1 if $x = 0$, 0 if $x \geq \phi(M_k)$, and y such that `W[`y`].pos`$= x$ otherwise.

We say that `W` is the kth wheel, with size M_k. For our C++ notation, we will declare `W` to be of class type `Wheel(`k`)`, where k is an integer parameter.

For examples of the wheel data structure, see [12, 14].

Note that, given a value for k, the kth wheel can be constructed in time proportional to M_k. See [6].

Algorithm 2.3: The Sieve of Eratosthenes with a Wheel. Next we show how to use a wheel with the sieve of Eratosthenes. The idea is, for each prime $p \leq \sqrt{n}$, to generate only those multiples of p that are relatively prime to M_k.

```
int p,i,f,m,k,x;
BitVector S(n);
PrimeList P(sqrt(n));          // find the primes up to sqrt(n)
k = setk(n,P);                 // find k (see below)
Wheel W(k);                    // create the kth wheel W
m = W.size();                  // m=M_k
S.clearall();                  // clears all bits
for(i=1; i<=k; i=i+1)          // include the first k primes
  S.set(P[i]);
for(x=P[k+1]; x<=n; x=x+W[x%m].dist) // include the integers
  S.set(x);                    // relatively prime to M_k
//** Main Loop
for(p=P[k+1]; p<=sqrt(n); p=p+1) // loop over primes<=sqrt(n)
  if(S[p]==1)                  // make sure p is prime
    for(f=p; p*f<=n; f=f+W[f%m].dist) // generate p's multiples
      S.clear(p*f);            // p*f is not prime
```

Normally, one chooses a value for k such that M_k is between $n^{1/3}$ and $\sqrt{n}$, implying $p_k = \Theta(\log n)$. This way, the time and space used by the wheel is insignificant in comparison to the complexity of the sieve as a whole, yet significant time savings are obtained, as we shall see in a moment. Our `setk()` function above chooses such a value for k; we leave its details to the reader.

The running time, which is dominated by the inner `for`-loop, is at most

$$\sum_{p \leq \sqrt{n}} \sum_{\substack{f \leq n/p \\ \gcd(f, M_k) = 1}} O(1) = \sum_{p \leq \sqrt{n}} O(\#\mathcal{W}_k(n/p)) = \sum_{p \leq \sqrt{n}} O\left(\frac{n}{p \log\log M_k}\right)$$

$$= O\left(\frac{n \log\log n}{\log\log M_k}\right) = O(n).$$

Thus, the sieve of Eratosthenes with a wheel runs in linear time.

Algorithm 2.4: Pritchard's Sieve with a Wheel. Using a wheel with Pritchard's linear sieve uses a similar technique. To generate integers $x = pf$ that are relatively prime to M_k, we loop over only those f-values that are relatively prime to M_k, and only use primes larger than p_k. In fact, the setup is identical to what we did with the Sieve of Eratosthenes above, so we need only present the Main Loop here:

```
//** Main Loop
for(f=P[k+1]; f<=n/P[k+1]; f=f+W[f%m].dist)
  for(i=k+1; i<=P.length(); i=i+1)
  {
    p=P[i];             // P[i] is the ith prime, with P[1]==2
    if(p>n/f) break;    // if p is too large, get the next f
    S.clear(p*f);       // p*f is composite
    if(f%p==0) break;   // if p|f, p==lpf(f), so get the next f
  }
```

The running time to initialize the bit vector `S` is now significant in comparison to the main loop, so let us analyze the initialization time. We view `S` as a bit vector with $\Omega(\log n)$ bits packed in each word. Thus, by assigning a 0 to each word, the `S.clearall()` operation takes only $O(n/\log n)$ time. The loops that follow take at most $O(\#\mathcal{W}_k(n)) = O(n/\log\log n)$ time. From this, we see that initialization takes at most $O(n/\log\log n)$ time.

The running time for the main loop is proportional to the number of composite integers in $\mathcal{W}_k(n)$. But this is at most $O(\#\mathcal{W}_k(n)) = O(n/\log\log n)$. Thus, the running time for Pritchard's linear sieve with a wheel is $O(n/\log\log n)$. It uses $O(n)$ space.

Reducing Space with a Wheel. With some further modification, both of these sieves can employ the wheel to reduce their space use to $O(n/\log\log n)$. The idea is to store only $\#\mathcal{W}_k(n)$ bits in `S`, one bit for each integer up to n relatively prime to M_k. The bit for the integer x would be at bit position $\texttt{W}[x \bmod M_k].\texttt{pos} + \phi(M_k) \cdot \lfloor x/M_k \rfloor$. Calculating the bit position for x takes $O(1)$ time if we use the fact that $\phi(M_k) = \texttt{W}[M_k - 1].\texttt{pos}$. Thus, using this method does not affect the asymptotic running time of either sieve in theory. In practice, this space-saving technique is not always worthwhile, as the increase in running time is quite noticeable.

2.5 Segmentation

Segmentation is perhaps the best method for reducing the space used by a prime number sieve. In this section, we will show how to apply segmentation to reduce the space used by the Sieve of Eratosthenes by a factor proportional to $\sqrt{n}$, and by Pritchard's sieve by a factor proportional to $\log n$.

The basic idea is to split the bit vector `S` into segments, each of size Δ. All the primes are found in a particular segment before moving to the next, permitting the reuse of the same space. The overall structure of the sieve is as follows:

```
Primelist P(sqrt(n)); // find primes up to sqrt(n)
Initialize();         // perform any necessary initialization
for(l=sqrt(n); l<n; l=l+delta)
{
  r=min(l+delta,n);
  sieve(l,r,P);        // sieve the interval [l+1,r]
}
```

Next, we show now to segment the Sieve of Eratosthenes and Pritchard's linear sieve.

Algorithm 2.5: Segmenting the Sieve of Eratosthenes. Bays and Hudson [4] were the first to publish a version of the sieve of Eratosthenes that was segmented. Pritchard [12] showed how to apply a wheel to this sieve to obtain $O(n)$ time using $O(\sqrt{n})$ bits of space. We present a slight modification of Pritchard's algorithm.

Below is the code fragment for the `sieve()` procedure. `S` is a bit vector of length `delta` $= \Delta$ which is reused for each segment.

```
//** Initialize the segment
S.clearall();
for(x=l+W[l%m].dist; x<=r; x=x+W[x%m].dist)
  S.set(x-l);
//** Main loop: Sieve by primes up to sqrt(n)
for(i=k+1; i<=P.length(); i=i+1)
{
  p=P[i];
  firstf=l/p+W[(l/p)%m].dist;
  for(f=firstf; p*f<=r; f=f+W[f%m].dist)
    S.clear(p*f-l);
}
```

After the main loop, `S` represents the set of primes between $l+1$ and r inclusive. We leave the details of the `Initialize()` procedure to the reader.

The running time of this algorithm is proportional to

$$\frac{n}{\Delta} \sum_{p \leq \sqrt{n}} \left(\frac{\Delta}{p \log \log n} + 1 \right)$$

which is $O(n)$ when $\Delta \gg \sqrt{n}$. As the list of primes up to $\sqrt{n}$ takes space proportional to $\sqrt{n}$, the total space used by this algorithm is minimized at $O(\sqrt{n})$ when $\Delta = O(\sqrt{n})$. By applying the space-saving technique using a wheel as mentioned earlier, and by playing some games in how the larger primes below $\sqrt{n}$ are stored, we can further reduce the space requirement to $O(\sqrt{n}/\log \log n)$.

For more details, see [12].

Algorithm 2.6: Segmenting Pritchard's Linear Sieve, with Wheel. Segmenting this algorithm is a bit more challenging. There are two main problems:

- For a given interval, which f-values should be used? Some f-values may have been "finished" on an earlier interval when a prime p is found with $p \mid f$. The solution is to maintain a bit vector `fok`, which keeps track of those f-values that have "finished." This bit vector takes $O(n/p_k) = O(n/\log n)$ bits. This reduces by a factor of $\log\log n$ with the wheel space-saving technique.
- For a given f-value, which primes p should be used? The solution is to keep a reverse-index of primes. This is simply an array `r[]` where `r[x]` gives the index i of the smallest prime p_i with $p_i \geq x$. We never need to use this for $x > \sqrt{n}$, so this is fast to construct and takes $O(\sqrt{n}\log n)$ bits of space.

With these ideas, the main loop of our algorithm looks like this:

```
for(f=P[k+1]; f<=n/P[k+1]; f=f+W[f%m].dist)
  if(fok[f]==1)
  {
    imin=max(k+1,r[l/f]); // find the first prime for this f
    for(i=imin; i<=P.length(); i=i+1)
    {
      p=P[i];
      if(p>r/f) break;
      S.clear(p*f-l);
      if(f%p==0) { fok.clear(f); break; }
    }
  }
```

The size of a segment is much larger than for Pritchard's segmented wheel sieve; we use $\Delta = n/p_k = O(n/\log n)$. Again, this reduces to $O(n/(\log n \log\log n))$ with the use of a wheel to save space. The running time remains at $O(n/\log\log n)$.

For more details, see [6].

3 Sieves That Use Very Little Space

Our first sieve is a very simple modification of Pritchard's segmented wheel sieve. Pritchard's sieve uses roughly $\sqrt{n}$ space, which includes the space to store the primes up to $\sqrt{n}$ and the space to hold the bit vector that represents the current segment.

Our new idea is to sieve by *all* the integers up to $\sqrt{n}$ instead of just the primes. We can then reduce the size of the segment, and we no longer need to store the primes up to $\sqrt{n}$. The tradeoff is an increased running time.

Algorithm 3.1

Choose c, with $0 < c \leq 1/2$. We use $\Delta = n^c$ for the segment size. We first find (and output) the primes up to $\log n$, for use in choosing k to construct the wheel.

We use $M_k \approx n^{c/2}$. This way the space used by the wheel is insignificant, yet it is sufficiently large to receive the $\Theta_c(\log\log n)$ speed benefit. Below is the code fragment for the `sieve()` function. As before, the current interval is $[l+1, r]$. The only thing new is the use of all integers relatively prime to M_k for sieving.

```
//** Initialize the segment
int x,d,f,m=W.size();
S.clearall();
for(x=l+W[l%m].dist; x<=r; x=x+W[l%m].dist)
  S.set(x-l);
//** Main loop: sieve by integers relatively prime to m
for(d=P[k+1]; d<=sqrt(r); d=d+W[d%m].dist)
{
  firstf=l/d+W[(l/d)%m].dist;
  for(f=firstf; d*f<=r; f=f+W[f%m].dist)
    S.clear(d*f-l);
}
```

At this point, `S` represents the primes in the interval $[l+1, r]$.

Complete details have been omitted in the interest of space.

Lemma 1. *Let n, m be positive integers with $m < n$. Then*

$$\sum_{\substack{x \le n \\ \gcd(x,m)=1}} \frac{1}{x} = \frac{\phi(m)}{m}(\log(n/m) + O(1)).$$

Proof. We have

$$\begin{aligned}\sum_{\substack{x \le n \\ \gcd(x,m)=1}} \frac{1}{x} &= \sum_{\substack{a \le m \\ \gcd(a,m)=1}} \sum_{mk+a \le n} \frac{1}{mk+a} \\ &= \sum_{\substack{a \le m \\ \gcd(a,m)=1}} \frac{1}{m} \sum_{k < n/m} \left[\frac{1}{k} + O\left(\frac{1}{k^2}\right)\right] \\ &= \frac{\phi(m)}{m}(\log(n/m) + O(1)).\end{aligned}$$

Here we used the fact that $\sum_{i=1}^{n} 1/i = \log n + \gamma + O(1/n)$[7]. □

Theorem 2. *Let $0 < c \le 1/2$, and set $\Delta = n^c$. Algorithm 3.1 correctly finds the primes up to n using $O(n\sqrt{n}/(\Delta(\log\log n)^2) + (n\log n)/(\log\log n)^2)$ operations and $O(\Delta)$ space.*

Proof. Correctness follows from our discussion above and from the previous section.

As discussed earlier, it takes $O(\Delta/\log n)$ operations to perform the `S.clearall()` function. Thus, the total number of arithmetic operations is at most proportional

to

$$\begin{aligned}
&\frac{n}{\Delta}\left(\frac{\Delta}{\log n}+\sum_{\substack{x\le\sqrt{n}\\ \gcd(x,M_k)=1}}\left(\frac{\Delta}{x\log\log n}+1\right)\right)\\
&\ll \frac{n}{\log n}+\frac{n}{\Delta}\frac{\phi(M_k)}{M_k}\left(\frac{\Delta\log n}{\log\log n}+\sqrt{n}\right)\\
&\ll \frac{n\sqrt{n}}{\Delta\log\log n}+\frac{n\log n}{(\log\log n)^2}.
\end{aligned}$$

The only significant space used is that of the wheel and the bit vector `S`, both of which are bounded by $O(\Delta)$ bits. □

Note that the second term in the running time, the $O((n\log n)/(\log\log n)^2)$ term, is only significant if we choose $\Delta \gg \sqrt{n}/\log n$.

One may also choose, say, $c = 1/3$ or $c = 1/4$. The first choice yields a sieve that uses roughly $n^{7/6}$ time with $n^{1/3}$ space, and the second yields a sieve that uses roughly $n^{5/4}$ time and $n^{1/4}$ space.

Algorithm 3.2

There is a slight modification we can make to Algorithm 3.1 that is of interest when Δ is close to $\sqrt{n}$. Currently, in the main loop of the `sieve()` function, we sieve by all integers up to $\sqrt{n}$ that are relatively prime to M_k. As an alternative, we will use Pritchard's segmented wheel sieve to find the primes up to $\sqrt{n}$ in $O(\sqrt{n})$ operations using $O(n^{1/4})$ bits of space. These primes do not need to be stored; they are used to sieve the interval as they are generated, and they are regenerated for each interval.

The space used by this algorithm is $O(n^{1/4}+\Delta)$, and the number of arithmetic operations used is at most proportional to

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

This modification only makes sense if $c \ge 1/4$, as Pritchard's sieve requires $n^{1/4}$ space to find the primes up to $\sqrt{n}$ anyway.

This method is superior to Algorithm 3.1 when, say, $\Delta = \sqrt{n/\log n}$. In this case, Algorithm 3.1 takes $O((n\log n)/(\log\log n)^2)$ operations, whereas Algorithm 3.2 uses only $O(n\sqrt{\log n})$ operations. Both use the same amount of space at $O(\sqrt{n/\log n})$.

We have proved the following.

Theorem 3. *Let $1/4 \le c \le 1/2$, and set $\Delta = n^c$. Algorithm 3.2 correctly finds the primes up to n using $O(n\sqrt{n}/\Delta + n)$ operations and $O(\Delta)$ space.*

4 A Space-Efficient Linear Sieve

In this section, we present Algorithm 4.1 which takes $O(n)$ operations while using only $O(\sqrt{n}/(\log\log n)^2)$ bits of space. This is less space than Pritchard's segmented wheel sieve [12] by a factor of $\log\log n$.

Algorithm 4.1

The basic idea is as follows. On each interval, we first sieve by the primes up to a bound near to, but less than, $\sqrt{n}$, such as $\sqrt{n}/(\log n)^2$. We then sieve by all integers relatively prime to M_k between $\sqrt{n}/(\log n)^2$ and $\sqrt{n}$. We choose $\Delta = \sqrt{n}/\log\log n$.

```
int l,r,k,delta,plimit;
delta=sqrt(n)/log(log(n)); // compute our segment size
plimit=sqrt(n)/(log(n)*log(n));
Primelist P(plimit);  // find the primes up to plimit
k=setk(n,P);          // set a value for k (see below)
Wheel W(k);           // build the kth wheel
output(P);            // output the primes up to plimit
Bitvector B(delta);

for(l=plimit; l<n; l=l+delta)
{
  r=min(l+delta,n);
  sieve(l,r,W,P,S);   // sieve the interval [l+1,r]
}
```

We assume that the `setk()` function chooses a value for k that results in a wheel with size $M_k \approx n^{1/3}$.

Below is the code fragment for the `sieve()` function.

```
//** Initialize the segment
int x,d,f,m=W.size();
S.clearall();
for(x=l+W[l%m].dist; x<=r; x=x+W[l%m].dist)
  S.set(x-l);
//** Main loop 1: sieve by primes
for(i=k+1; i<P.length(); i=i+1)
{
  p=P[i];
  firstf=l/p+W[(l/p)%m].dist;
  for(f=firstf; p*f<=r; f=f+W[f%m].dist)
    S.clear(p*f-l);
}
//** Main loop 2: sieve by integers rel. prime to m
for(d=P[P.length()]; d<=sqrt(r); d=d+W[d%m].dist)
```

```
{
  firstf=l/d+W[(l/d)%m].dist;
  for(f=firstf; d*f<=r; f=f+W[f%m].dist)
    S.clear(d*f-l);
}
```

At this point, S represents the primes in the interval $[l+1, r]$. We assume the wheel space-saving technique is employed with S.

Theorem 4. *Algorithm 4.1 correctly finds the primes up to n using $O(n)$ operations and $O(\sqrt{n}/(\log\log n)^2)$ space.*

Proof. Correctness follows from our discussions above and from earlier sections.

The space used by the bit vector S is $O(\sqrt{n}/(\log\log n)^2)$, because $\Delta = \sqrt{n}/\log\log n$ and we are employing the space-saving technique using the wheel to save an additional factor of $\log\log n$. The list of primes up to $\sqrt{n}/(\log n)^2$ uses only $O(\sqrt{n}/(\log n)^2)$ bits. The wheel requires roughly $n^{1/3}$ bits. Thus, the total space used is $O(\sqrt{n}/(\log\log n)^2)$ bits.

The number of arithmetic operations is at most proportional to

$$\frac{n}{\Delta}\left(\frac{\Delta}{\log\log n} + \sum_{p\le\frac{\sqrt{n}}{(\log n)^2}}\left(\frac{\Delta}{p\log\log n}+1\right) + \sum_{\substack{\frac{\sqrt{n}}{(\log n)^2}\le d\le\sqrt{n}\\ \gcd(d,M_k)=1}}\left(\frac{\Delta}{d\log\log n}+1\right)\right).$$

The second term in the parentheses is easily seen to be $O(\Delta + \sqrt{n}/(\log n)^3) = O(\Delta)$ as $\Delta = \sqrt{n}/\log\log n$. This dominates the first term. For the third term, we apply Lemma 1 to obtain

$$\begin{aligned}
&\sum_{\substack{d\le\sqrt{n}\\ \gcd(d,M_k)=1}}\left(\frac{\Delta}{d\log\log n}+1\right) - \sum_{\substack{d\le\sqrt{n}/(\log n)^2\\ \gcd(d,M_k)=1}}\left(\frac{\Delta}{d\log\log n}+1\right)\\
&= \frac{\Delta}{\log\log n}\frac{\phi(M_k)}{M_k}\left[\log\left(\frac{\sqrt{n}}{M_k}\right) - \log\left(\frac{\sqrt{n}}{M_k(\log n)^2}\right) + O(1)\right] + O\left(\frac{\sqrt{n}}{\log\log n}\right)\\
&= O\left(\frac{\Delta}{\log\log n}\frac{\phi(M_k)}{M_k}\log\log n + \frac{\sqrt{n}}{\log\log n}\right)\\
&= O(\Delta).
\end{aligned}$$

From this, we see that the total number of arithmetic operations is $O((n/\Delta)\Delta) = O(n)$. □

5 A Space-Efficient Sublinear Sieve

In this section we present Algorithm 5.1, a sublinear sieve that uses less space than the sieve of Dunten, Jones, and Sorenson [6]. This sieve is a combination of Pritchard's segmented wheel sieve[12] and that of Dunten, Jones, and Sorenson.

Algorithm 5.1

In Algorithm 2.6, the `fok[]` array uses $O(n/(\log n \log\log n))$ bits of space. To reduce the space use of this algorithm, we must somehow reduce the number of f-values that we need in the main loop, thereby reducing the storage requirements for this array.

The idea is to pre-sift the current segment by all primes up to a bound B, where $\log n \le B \le \sqrt{n}$. Then, we only need to use f-values satisfying $B < f \le n/B$ and relatively prime to M_k. To avoid crossing off multiples of primes below B, we also pre-sieve the `fok[]` array by the primes up to B.

```
//** Sieve by primes up to B
for(i=k+1; P[i]<=B; i=i+1)
{
  p=P[i];
  firstf=l/p+W[(l/p)%m].dist;
  for(f=firstf; p*f<=r; f=f+W[f%m].dist)
    S.clear(p*f-l);
}
//** Main loop
for(f=B+W[B%m].dist; f<=n/B; f=f+W[f%m].dist)
  if(fok[f]==1)
  {
    imin=max(k+1,r[l/f]); // find the first prime for this f
    for(i=imin; i<=P.length(); i=i+1)
    {
      p=P[i];
      if(p>r/f) break;
      S.clear(p*f-l);
      if(f%p==0) { fok.clear(f); break; }
    }
  }
```

We use a segment size of $\Delta = n/B$. The only change to preprocessing is that we need to sieve the `fok[]` array. We assume the wheel is roughly $n^{1/3}$ in size.

Theorem 5. *Algorithm 5.1 correctly finds the primes up to n using*

$$O\left(\frac{n\log(1+\log B/\log\log n)}{\log\log n}\right)$$

arithmetic operations and $O(n/(B\log\log n)+\sqrt{n})$ bits of space.

Proof. Correctness follows from that of Algorithms 2.5 and 2.6.

The space used by the algorithm is dominated by that used for the segment and the `fok[]` array. Both can be implemented to use the wheel-based space saving technique so that they require only $O(\Delta/\log\log n) = O(n/(B\log\log n))$ bits. The $\sqrt{n}$ term arises from the space needed to store the primes up to $\sqrt{n}$.

To calculate the running time of the algorithm, we will first compute the preprocessing time, then the time spent on one segment, and finally combine these to compute the total time.

In preprocessing, we must compute the wheel, find the primes up to $\sqrt{n}$, compute the reverse index array `r[]`, and sieve the `fok[]` array. Except for the `fok[]` array, everything can be done in $O(\sqrt{n})$ operations. Sieving the `fok[]` array takes time proportional to

$$\sum_{p_k < p \le B} \frac{n/B}{p \log\log n} \ll \frac{n}{B \log\log n}(\log\log B - \log\log p_k + O(1))$$
$$\ll \frac{n \log(1 + \log B / \log\log n)}{B \log\log n},$$

as $p_k = \Theta(\log n)$.

In processing a segment, we first initialize the segment, then sieve by primes up to B, and finally execute our main loop.

As mentioned earlier, initializing the sieve takes $O(\Delta/\log n + \Delta/\log\log n)$ operations. This is the time needed to zero out the bit vector and the time to place ones in positions corresponding to integers relatively prime to M_k.

Sieving the segment by primes between p_k and B takes the same time as sieving the `fok[]` array: $O(n \log(1 + \log B/\log\log n)/(B \log\log n))$ operations.

Finally, the main loop crosses off each composite integer x from the current interval where x has no prime divisors less than B. This takes at most $O(\Delta/\log\log n)$ time.

Summing this time over the n/Δ segments, we obtain a total running time of $O(n \log(1 + \log B/\log\log n)/\log\log n)$ arithmetic operations. □

This theorem provides a nice space-time tradeoff between Algorithms 2.5 and 2.6. For example, one may choose $B = \sqrt{n}$, essentially obtaining Pritchard's segmented wheel sieve, or $B = \log n$, to get the sieve of Dunten, Jones, and Sorenson. Choosing $B = (\log n)^l$ for some fixed $l > 1$ gives new space bounds for $O(n/\log\log n)$-time sieves.

6 Timing Results

In this section we present the timing results from our implementation of Algorithms 3.1 (with $\Delta = n^{0.4}$), Algorithm 4.1, and Algorithm 5.1 (with l ranging from 1.5 to 3). We also implemented several of the algorithms mentioned in Sect. 2 for purposes of comparison. As we are primarily interested in space-efficient algorithms, we focused on segmented sieves.

All algorithms were implemented in C++. Timing results are given in CPU seconds. For smaller values of n, the time given is an average over several runs. Dashed entries indicate that the algorithm was not run for that input due to excessive space requirements. The results are presented in Table 1.

Table 1. Average running times in CPU seconds

Algorithm	$n = 10^4$	$n = 10^5$	$n = 10^6$	$n = 10^7$	$n = 10^8$	$n = 10^9$
Alg. 2.1 Sieve of Eratosthenes	0.00086	0.0104	0.136	2.80	–	–
Bays and Hudson	0.00282	0.0274	0.264	2.68	27.44	286
Alg. 2.5 Segmented with wheel	0.00210	0.0212	0.216	2.27	23.61	242
Alg. 2.2 Pritchard's Linear Sieve	0.00248	0.0242	0.258	2.68	–	–
Alg. 2.4 With wheel	0.00136	0.0130	0.146	1.74	–	–
Alg. 3.1	0.00904	0.0828	1.002	12.03	122.28	1478
Alg. 4.1	0.00612	0.0674	0.750	8.16	78.11	816
Alg. 5.1 $l = 1.50$	0.00278	0.0252	0.266	2.42	25.10	261
Alg. 5.1 $l = 1.75$	0.00284	0.0248	0.262	2.42	25.06	258
Alg. 5.1 $l = 1.79$	0.00280	0.0234	0.234	2.37	24.36	254
Alg. 5.1 $l = 2.00$	0.00276	0.0244	0.256	2.40	24.83	255
Alg. 5.1 $l = 2.25$	0.00290	0.0248	0.260	2.48	25.40	259
Alg. 5.1 $l = 2.50$	0.00318	0.0256	0.266	2.51	25.85	262
Alg. 5.1 $l = 2.75$	0.00402	0.0290	0.270	2.54	26.56	269
Alg. 5.1 $l = 3.00$	0.00508	0.0410	0.268	2.55	26.71	274

The computing platform was a 200MHz Pentium Pro with 96MB RAM and a 512kB cache running Linux 2.0. We used the gnu `g++` compiler version 2.7.2.1 with `-O` optimization.

We remind the reader that all timing results depend not only on the particular platform, operating system, and compiler, but also on the language and the programmer. Thus, any conclusions drawn from such data must be met with a certain degree of scepticism.

With that said, it seems clear that Algorithm 5.1 is very practical. If l is chosen carefully, and the code is tuned more than ours was, it may be capable of surpassing Pritchard's segmented wheel sieve. We found $l = 1.79 \pm 0.005$ to be nearly optimal.

Acknowledgements

Special thanks to Justin Hockemeyer, who, during the summer of 1997, implemented a version of Algorithm 3.1. He was supported by the Butler Summer Institute.

References

1. Adleman, L. M., Pomerance, C., Rumely, R.: On distinguishing prime numbers from composite numbers. Annals of Mathematics **117** (1983) 173–206
2. Bach, E.: Analytic Methods in the Analysis and Design of Number-Theoretic Algorithms. MIT Press, Cambridge (1985)
3. Bach, E., Shallit, J.: Algorithmic Number Theory, Vol. 1. MIT Press, Cambridge (1996)

4. Bays, C., Hudson, R.: The segmented sieve of Eratosthenes and primes in arithmetic progressions to 10^{12}. BIT **17** (1977) 121–127
5. Bernstein, D. J.: Personal communication. (1998)
6. Dunten, B., Jones, J., Sorenson, J. P.: A space-efficient fast prime number sieve. Information Processing Letters **59** (1996) 79–84
7. Greene, D. H., Knuth, D. E.: Mathematics for the Analysis of Algorithms. 3rd edn. Birkhäuser, Boston (1990)
8. Hardy, G. H., Wright, E. M.: An Introduction to the Theory of Numbers. 5th edn. Oxford University Press (1979)
9. Miller, G.: Riemann's hypothesis and tests for primality. Journal of Computer and System Sciences **13** (1976) 300–317
10. Pomerance, C. (ed.): Cryptology and Computational Number Theory. Proceedings of Symposia in Applied Mathematics, Vol. 42. American Mathematical Society, Providence (1990)
11. Pritchard, P.: A sublinear additive sieve for finding prime numbers. Communications of the ACM **24**(1) (1981) 18–23,772
12. Pritchard, P.: Fast compact prime number sieves (among others). Journal of Algorithms **4** (1983) 332–344
13. Pritchard, P.: Linear prime-number sieves: A family tree. Science of Computer Programming **9** (1987) 17–35
14. Sorenson, J. P., Parberry, I.: Two fast parallel prime number sieves. Information and Computation **144**(1) (1994) 115–130
15. Stroustrup, B.: The C++ Programming Language. 2nd edn. Addison-Wesley (1991)