A LINEAR SIEVE ALGORITHM FOR FINDING PRIME NUMBERS

by

David Gries[+]
Jayadev Misra[++]

TR 77-313

+ Computer Science Department
Cornell University
Ithaca, N.Y. 14853

++ Computer Science Department
University of Texas at Austin
Austin, TX 78712

A Linear Sieve Algorithm for Finding Prime Numbers

by

David Gries
Computer Science Department
Cornell University

and

Jayadev Misra
Computer Science Department
University of Texas at Austin

This research was partially supported by NSF grants DCR75-09842 and MCS76-22360.

1. Introduction

An algorithm is presented for finding all primes between 2 and n, for $n \geq 4$, that executes in time proportional to n (assuming that multiplication of integers not larger than n can be performed in unit time). Like the Sieve of Eratosthenes, it works by removing nonprimes from the set $\{2,\ldots,n\}$. Unlike the Sieve of Eratosthenes, no attempt is ever made to remove a nonprime that was removed earlier; this allows us to develop a linear algorithm.

The algorithm deals with sets S satisfying $S \subset \{2,\ldots,n\}$. Two operations will be required on such sets:

remove(S,i) is defined for $i \in S$. This implements $S := S - \{i\}$.

next(S,i) is a function defined only for $i \in S$ such that there is an integer larger than i in S; it yields the next larger integer in S.

In order to achieve linearity, the total time spent executing these operations must be no worse than proportional to n. Thus, this algorithm provides an interesting context for a discussion of selection of data structures.

2. The algorithm

For $i \geq 2$, denote by $\ell p(i)$ the lowest prime which divides i evenly. The algorithm is based on the following theorem.

(2.1) Theorem. A nonprime x can be written uniquely as

$$x = p^k \cdot q$$

where (1) p is prime, $p = \ell p(x)$;
(2) $1 \leq k$;
(3) $p = q$ or $p < \ell p(q)$

Proof By the unique factorization theorem (see for example LeVeque [56]), x can be written uniquely as

$$x = p_1^{n_1} \cdot \ldots \cdot p_m^{n_m}$$

where $m \geq 1$, the p_i are primes, $p_i < p_{i+1}$ for $1 \leq i < m$, and $m = 1$ implies $n_1 > 1$. Hence the following yields the only choice for p, q and k of the theorem:

if $m = 1$, let $p = p_1$, $q = p_1$ and $k = n_1 - 1$,
if $m > 1$, let $p = p_1$, $q = p_2^{n_2} \cdot \ldots \cdot p_m^{n_m}$ and $k = n_1$. Q.E.D.

Subsequently, we write $x = X(p,q,k)$ to denote that x is nonprime and $x = p^k \cdot q$ where p, q and k have the properties described in Theorem (2.1).

A prime cannot be written as described in the theorem, so the algorithm to delete nonprimes from S need only produce all combinations of (p,q,k) and delete the corresponding nonprimes $x = X(p,q,k)$. The trick is to produce each combination exactly once, and in such an order that the next combination can be efficiently calculated from the current one. For this purpose, we use the total ordering α on nonprimes $x = X(p,q,k)$ induced by the lexicographic ordering of the corresponding triples (p,q,k):

(2.2) Definition. Let $x = X(p,q,k)$ and $\bar{x} = X(\bar{p},\bar{q},\bar{k})$. Then

$$
\begin{aligned}
x \,\alpha\, \bar{x} \Leftrightarrow\ & p < \bar{p} \ \underline{\text{or}} \\
& (p = \bar{p} \ \underline{\text{and}} \ q < \bar{q}) \ \underline{\text{or}} \\
& (p = \bar{p} \ \underline{\text{and}} \ q = \bar{q} \ \underline{\text{and}} \ k < \bar{k})
\end{aligned}
$$

Table (2.3) illustrates this ordering and at the same time depicts how the algorithm works. The rows give successive values for pairs (p,q), together with the contents of the set S before nonprimes with this p and q are deleted. In each row, the nonprimes $x = X(p,q,k)$ to be deleted for $k = 1,2,3$ have been circled.

p	q	s																										
2	2	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
2	3	1	2	3		5	6	7		9	10	11	12	13	14	15		17	18	19	20	21	22	23	24	25	26	27
2	5	1	2	3		5		7		9	10	11		13	14	15		17	18	19	20	21	22	23		25	26	27
2	7	1	2	3		5		7		9		11		13	14	15		17	18	19		21	22	23		25	26	27
2	9	1	2	3		5		7		9		11		13		15		17	18	19		21	22	23		25	26	27
2	11	1	2	3		5		7		9		11		13		15		17		19		21	22	23		25	26	27
2	13	1	2	3		5		7		9		11		13		15		17		19		21		23		25	26	27
3	3	1	2	3		5		7		9		11		13		15		17		19		21		23		25		27
3	5	1	2	3		5		7				11		13		15		17		19		21		23		25		
3	7	1	2	3		5		7				11		13				17		19		21		23		25		
5	5	1	2	3		5		7				11		13				17		19				23		25		

(2.3) <u>Table illustrating execution of the algorithm for n = 27</u>

The algorithm uses a variable S, which initially contains the set $\{2,\ldots,n\}$ and from which nonprimes are to be deleted, and integer variables p,q,k and x used to generate nonprimes in the order defined by α. The invariant relation P used in the loop of the algorithm is given in (2.4). It is not difficult to follow; lines (1)-(3) describe properties of p,q and k, line (4) describes the properties of the value x under consideration for deletion, and line (5) describes the current contents of set S.

(2.4) $P \equiv$ (1) p prime, $4 \leq p^2 \leq n$;

(2) $p = q$ or $p < \ell p(q)$; $p \cdot q \leq n$;

(3) $1 \leq k$;

(4) $x = X(p,q,k)$;

(5) $S = \{2,\ldots,n\} - \{y \mid y \text{ nonprime } \underline{\text{and}} \ y \ \alpha \ x\}$

The goal of the loop of the algorithm is to have S contain only the primes in $\{2,\ldots,n\}$. The object of each iteration of the loop is to get us closer to this goal, while keeping P invariantly true. We now investigate operations with this property.

If $x = X(p,q,k) \leq n$, then clearly x is to be deleted from S, and P can be restored by executing $k,x:=k+1,p \cdot x$. If $x > n$, we need to determine the next nonprime y (say), according to ordering α, to be deleted from S. Remarkably enough, Lemmas (2.5)-(2.6) indicate that under suitable conditions $y = p \cdot next(S,q)$, so that one can change p,q,k,x to denote the next nonprime to delete by executing $q:=next(S,q)$; $k,x:=1,p \cdot q$. Similarly, Lemmas (2.7)-(2.8) state the conditions under which $y = next(S,p)^2$. We shall prove these lemmas in Section 3.

(2.5) <u>Lemma</u>. Invariant P implies that $next(S,q)$ is defined, $next(S,q) < n$ and $p < \ell p(next(S,q))$. Writing $y = X(p,next(S,q),1)$, we have $x \; \alpha \; y$.

(2.6) <u>Lemma</u>. Suppose P and $x > n$. Write $y = X(p,next(S,q),1)$. Then no nonprime z in S satisfies $x \; \alpha \; z \; \alpha \; y$.

(2.7) <u>Lemma</u>. Invariant P implies that $next(S,p)$ is defined, $next(S,p) < n$ and $next(S,p)$ is prime. Writing $y = X(next(S,p),next(S,p),1)$, we have $x \; \alpha \; y$.

(2.8) <u>Lemma</u>. Suppose P <u>and</u> $x > n$ <u>and</u> $p \cdot next(S,q) > n$. Write $y = next(S,p)^2$. Then no nonprime z in S satisfies $x \; \alpha \; z \; \alpha \; y$.

We write the algorithm in (2.9) using guarded commands (Dijkstra[75]). We will discuss the arguments necessary to ascertain correctness in Section 3. Note that variable k is used only in assignments to itself, so that all references to it may be deleted. It has been included only to clarify the relation between p,q and x.

```
(2.9)  {n ≥ 4}
     p,q,k,x,S:= 2,2,1,4,{2,...,n};
     do  x ≤ n                         → remove (S,x);
                                         k,x:=  k+1,p·x
     □   x > n  and p·next(S,q) ≤ n    → q:=  next(S,q);
                                         k,x:=  1,p·q
     □   x > n  and p·next(S,q) > n  and next(S,p)² ≤ n
                                       → p:=  next(S,p);
                                         q,k,x:=  p,1,p·p
     od
     {S = {y|2 ≤ y ≤ n and y prime}
```

The reader may feel more comfortable with algorithm (2.10) which uses conventional while loops instead of a guarded command loop. However, (2.9) is easier to prove correct; one loop with one invariant is easier to understand in this instance than three nested loops with three invariants.

```
(2.10)  p,S:= 2,{2,...,n};
        while  p·p ≤ n  do  begin
            q:= p;
            while p·q ≤ n  do  begin
                x:= p·q;
                while x ≤ n  do  begin
                    remove(S,x);  x:= p·x
                    end;
                q:= next(S,q)
                end;
            p:= next(S,p)
            end
```

The designers of ALPHARD -- Shaw, Wulf and London [76] -- might be happier with version (2.11). In (2.11), the generation of the next nonprime is performed in one place, making the algorithm more amenable to writing in terms of their "forms" and "iterators".

```
(2.11)  p,q,x,S:= 2,2,4, {2,...,n};
        do  x ≤ n  →  remove (S,x);
                      Generate next  x:
                           if p·x ≤ n → x:=  p·x
                           □ p·x > n  and p·next(S,q) ≤ n →
                                      q:=  next(S,q);
                                      x:=  p·q
                           □ p·x > n  and p·next(S,q) > n →
                                      p:=  next(S,p);
                                      q,x:=  p,p·p
                           fi
        od
```

3. Showing correctness and linearity

We prove Lemmas (2.5)-(2.8). Secondly, we briefly discuss the axiomatic proof method with respect to guarded commands (Dijkstra [75]) and then give some of the details of the proof.

In preparation for proving Lemmas (2.5) and (2.6), we first prove the following.

(3.1) Lemma. Consider any nonprime $z = X(\bar{p},\bar{q},\bar{k})$. We have

$$(P \text{ and } x \alpha z \text{ and } \bar{q} \le n) \text{ implies } \bar{q} \in S.$$

Proof. If $\bar{q}$ is prime it is in S. Suppose $\bar{q}$ is nonprime. From $x \alpha z$ and the decompositions of x and z we deduce $\ell p(x) = p \le \bar{p} < \ell p(\bar{q})$ so that $x \alpha \bar{q}$. From the definition of S and $x \alpha \bar{q}$ we deduce $\bar{q} \in S$. Q.E.D.

Our proofs of Lemmas (2.5) and (2.6) rest on the remarkable fact that for any positive integer $i > 1$, there is a prime v satisfying $i < v < 2i$[†].

Proof of Lemma (2.5). Let v be a prime satisfying $q < v < 2 \cdot q$. From P we conclude

$$p \le q < next(S,q) \le v < 2 \cdot q \le p \cdot q \le n$$

---

† As might be imagined, this is difficult to prove. J. Bertrand conjectured this in 1845, after showing empirically that it was true for $i < 10^6$. Chebyshev proved the conjecture in 1850. See LeVeque [56] for details. Our first draft of a proof and algorithm did not rely on this fact at all. It used weaker lemmas with more complicated proofs, and required the additional element $n+1$ to be in S so that $next(S,i)$ would be sure to be defined. For example, the original Lemma (2.5) read: Let $y = p \cdot next(S,q)$. Suppose P and $x > n$. Then either $next(S,q) = n+1$; or $y = X(p,next(S,q),1)$ and $x \alpha y$.

and hence $next(S,q) < n$. Secondly, no nonprime in S has a divisor less than p, so that $p \leq \ell p(next(S,q))$. To show that $p \neq \ell p(next(S,q))$, consider the fact that any nonprime $z = X(p,\bar{q},\bar{k})$ in S must have $q \leq \bar{q}$. Hence the smallest such nonprime z that may be in S is $p \cdot q$. Since $next(S,q) < p \cdot q$, $next(S,q)$ cannot be a nonprime with $p = \ell p(next(S,q))$.

Hence $p < \ell p(next(S,q))$. The relation $x \alpha y = X(p,next(S,q),1)$ follows immediately.

Proof of Lemma (2.6). From $x = X(p,q,k) > n$ and $y = X(p,next(S,q),1)$, we see that such a z would have a decomposition $z = X(p,\bar{q},\bar{k})$ with $q < \bar{q} < next(S,q)$. But $\bar{q}$ would not be in S, contradicting Lemma (3.1).

Proof of Lemma (2.7). Let v be a prime satisfying $p < v < 2 \cdot p$. From P we have

$$p < next(S,p) \leq v < 2 \cdot p \leq p^2 \leq n$$

and $next(S,p) \leq n$. Secondly, no nonprime in S has a divisor less than p, so that for all nonprimes $z \in S$ we have $p^2 \leq z$. Since $next(S,p) < p^2$, $next(S,p)$ must be prime. The fact $x \alpha y = X(next(S,p),next(S,p),1)$ follows immediately.

Proof of Lemma (2.8). This is similar to the proof of lemma (2.6) and is left to the reader.

We now discuss the proof method and give some details. The main part of algorithm (2.9) is a loop of the form

$$\underline{\text{do}}\ \ B1 \rightarrow SL1\ \square\ \ B2 \rightarrow SL2\ \square\ B3 \rightarrow SL3\ \ \underline{\text{od}}$$

Showing correctness involves exhibiting an invariant relation P (ours is given in (2.4) and an integer function t, and showing that the following hold:

1. P is true before execution of the loop;
2. P <u>and</u> <u>not</u> (B1 <u>or</u> B2 <u>or</u> B3) implies the desired result;
3. {P <u>and</u> Bi} SLi {P} for i=1,2,3;
4. Execution of the loop terminates:
   a) (P <u>and</u> (B1 <u>or</u> B2 <u>or</u> B3)) => $t \geq o$
   b) {P <u>and</u> Bi <u>and</u> $t \leq c+1$} SLi {$t \leq c$}
   for i=1,2,3, and any constant c.

Point 1 is obvious; point 2 we leave to the reader, since it can be shown quite easily with the help of Lemma (2.8).

Point 3 concerns the invariance of P under execution of each guarded command SLi. The only difficult point concerns the generation of new values for q,p and x to satisfy P. Lemmas (2.5)-(2.8) yield the facts necessary for this.

To see this a bit more formally in at least one case, consider determining the precondition Q in {Q} SL3 {P} where SL3 is the third guarded command list of the loop of (2.9) and P is in (2.4). SL3 is a sequence of assignments, so we apply the normal assignment and concatenation rules to arrive at

Q ≡ (1) next(S,p) prime, $4 \le \text{next}(S,p)^2 \le n$;

(2) next(S,p) = next(S,p) <u>or</u> (next(S,p) <lp(next(S,p)); $\text{next}(S,p)^2 \le n$;

(3) $1 \le 1$;

(4) $\text{next}(S,p)^2 = X(\text{next}(S,p), \text{next}(S,p), 1)$;

(5) S = {2,...,n}-{y|y nonprime <u>and</u> y α $\text{next}(S,p)^2$}

It is then a simple matter to prove that (P <u>and</u> B3) implies Q, with the help of Lemmas (2.7)-(2.8).

To show termination, we use the function t defined by

t = number of nonprimes z ε S satisfying (x=z <u>or</u> x α z)

+ number of nonprimes z ε S satisfying ¬x α z.

Note that $t \ge 0$. Execution of the first guarded command SL1 reduces the first term of t by at least one, since it removes x from S. Execution of the second and third guarded commands begin with x=xo (say) and x ∉ S and finish with xo α x and x ε S; hence they reduce the second term by one. Hence we conclude that the algorithm terminates.

The initial value for t is a bound on the number of times the loop will iterate. This is bounded by 2·(number of nonprimes in S) < 2·n. Hence the algorithm is linear if we can satisfactorily implement S and the operations on it. This is the subject of Section 4.

4. Implementing the set S

We discuss three approaches to implement $S \subset \{2,\ldots,n\}$, all dealing with forms of linked lists. We will actually implement sets $S \cup \{n+1\}$. The purpose is to provide an "anchor" for one end of the linked list. The integer 2 serves the same purpose at the other end of the list since it is never deleted.

Approach 1. If we implement S as a doubly-linked list, then remove (S,i) and next(S,i) can each be performed in constant time. Thus we use

```
var s: array (2:n+1) of record (pred,succ:integer)
```

where the various parts of s are used as follows:

$s(i).\text{succ} = \text{next}(S \cup \{n+1\}, i)$ for $i \in S$;

$s(i).\text{pred} \equiv$ unique integer j such that next$(j) = i$,
for $i \in S \cup \{n+1\}$, $i \neq 2$.

At any time, the elements of S can be found by following the successor chain beginning at $s(2)$ and ending just before $s(n+1)$. This approach requires roughly $2n$ locations (each of $\log_2 n$ bits). The three operations on S are:

```
S:= {2,...,n} ::
        i:= 1;
        do i < n → i:= i+1; s(i).succ:= i+1; s(i+1).pred:= i  c
remove(S,i) :: s(s(i).pred).succ:= s(i).succ;
               s(s(i).succ).pred:= s(i).pred;
next(S,i) :: s(i).succ
```

Approach 2. It would be nice to reduce the amount of space needed to implement the set. For example, if we can use a singly-linked list rather than a doubly-linked list, we would save half the space. We can do this, if we use one extra bit for each integer i to tell whether or not i is in S. We will not be able to show that next(S,i) can be performed in constant time, but we will be able to show that the total time spent evaluating next(S,i) in the algorithm contributes no more than time n.

The description below (and in approach 3) uses next(S,i) to denote the first integer larger than i that is in $S \cup \{n+1\}$, for *all* i in {2,...,n}, (The algorithm, remember, evaluates next(S,i) only for $i \in S$.) Let us use

var s: **array** (2..n+1) **of** integer;

var in: **array** (2..n+1) **of** boolean

where

(1) $in(i) \Longleftrightarrow i \in S \cup \{n+1\}$ for $2 \le i \le n+1$

(2) $i < s(i) \le next(S \cup \{n+1\}, i)$ for $2 \le i \le n$

(3) $s(n+1) = n+2$

We write $s^1(i) = s(i)$, $s^2(i) = s(s(i)), \ldots, s^t(i) = s(s^{t-1}(i))$.

Then for $i \in S$, next(S,i) is the value $s^t(i)$ satisfying

(4.1) **not** $in(s^1(i))$ **and** ... **and not** $in(s^{t-1}(i))$ **and** $in(s^t(i))$.

Let us look at an example. Suppose 31 and 37 are in S but 32 through 36 are not. Then three possible representations of this part of S are (using *t* and *f* for *true* and *false*):

(4.2)

	i: 31	32	33	34	35	36	37
(1)	s(i),in(i): 32,t	33,f	34,f	35,f	36,f	37,f	38,t
(2)	s(i),in(i): 37,t	33,f	34,f	35,f	36,f	37,f	38,t
(3)	s(i),in(i): 34,f	34,f	34,f	37,f	37,f	37,f	38,t

Thus we see that many different values of s and in may represent the same set S . Operations S:= {2,...,n} and remove(S,i) are:

```
S:= {2,...,n}   :: i:= 1;
                   do i < n+1 → i:= i+1; s(i),in(i):= i+1, true od;
remove(S,i)     :: in(i):= false
```

The implementation of next(S,i) must sequence through the values in(s(i)), $in(s^2(i))$,... until a true value is found. At the same time, it will change s(i) to the value next(S,i) so that a later evaluation of next(S,i) need not perform the same search!

```
next(S,i) :: do not in(s(i)) → s(i):= s(s(i)) od;
             s(i)
```

Note that evaluation of next(S,i) causes a side effect, but a "beneficial" side effect which is hidden from algorithm (2.4). Evaluation of next(S,31) with S defined as the first representation in (4.2) would change S to the second, but equivalent, representation in (4.2) and return the value 37.

Now, we cannot show that evaluation of next(S,i) can be performed in constant time -- the time is proportional to the value t used in (4.1). However, the beneficial side effect will allow us to show that the time necessary for <u>all</u> evaluations of next(S,i) during execution of the algorithm (2.4) is proportional to n.

That this is so follows easily from the fact that for any given integer j which is <u>not</u> in S (and therefore for which <u>not</u> in(s(j)) holds), at most <u>once</u> during execution of the algorithm will the body s(i):= s(s(i)) of the loop in next(S,i) be executed with s(i) = j . Consider, for example, the first partial representation of S in (4.2). Evaluation of next(S,31) causes s(32),...,s(37) to be referenced during the loop and changes the representation to the second one in (4.2). Thereafter, s(32:36) will no longer be referenced.

The most obvious way to implement S with a singly-linked list would be to require S(i)=next(S,i) instead of S(i) < next(S,i). This, however, would require the operation remove(S,i) to do too much work -- it would have to find the predecessor j(say) of i and change s(j). By waiting to change s(j) until we really need next(S,j), we save ourselves work. This trick of postponing such an operation turns out to be useful in many algorithms. It is used, for example, in the sorting algorithm given in the answer to exercise 12 of Section 5.2.1 (page 596) of Knuth [73].

Approach 3. Note that in approach 2, the value s(i) satisfies

$$i < s(i) \le next(S,i)$$

Instead of storing in s(i) the index of another entry s(j), let us store the increment needed to get from s(i) to s(j). Thus we will always have $i < i + s(i) \le next(S,i)$. The reason for this is that we may be able to reduce the number of bits needed for each s(i), since the increment will in general be much smaller than n. Asymptotically speaking, there are n/ln(n) or more primes in {2,...,n} , and if they were evenly distributed the increment to get from one to another would not be greater than $\ell n(n)$. Hence we would need only $ceil(\log_2 ln(n))$ bits for each s(i) instead of $ceil(\log_2(n))$.

Unfortunately, the primes are not evenly distributed, so we cannot assume s(i) will be so small. Knuth [1973,page 402] gives a table of "record breaking" gaps between prime numbers. For primes less than or equal to 20831533 we see that the largest gap is 210, so that 8 bits will suffice for $n \le 20831533$.

Let us implement S assuming that $s(i) \le b$ where b is to be chosen by the user, and revise the next(S,i) routine so that it never attempts to change s(i) to something larger than b.

For example, consider the first representation in (4.3) of S with 31,37 in S but 32-36 not in S, and suppose next(S,31) is to be evaluated. If $b \geq 4$ the representation would be changed to line (2) of (4.3) and the value 37 would result. If $b = 3$, however, line (3) would be the final representation.

(4.3)

	i:31	32	33	34	35	36	37
(1) s(i),in(i):	1,<u>t</u>	1,<u>f</u>	1,<u>f</u>	1,<u>f</u>	1,<u>f</u>	1,<u>f</u>	1,<u>t</u>
(2) s(i),in(i):	6,<u>t</u>	1,<u>f</u>	1,<u>f</u>	1,<u>f</u>	1,<u>f</u>	1,<u>f</u>	1,<u>t</u>
(3) s(i),in(i):	3,<u>t</u>	1,<u>f</u>	1,<u>f</u>	3,<u>f</u>	1,<u>f</u>	1,<u>f</u>	1,<u>t</u>

If b is too small, of course, then we may no longer have a linear algorithm because too much time is spent evaluating next(S,i). An unsolved problem is to determine how small b may be (as a function of n) and still have a linear algorithm.

The implementation of S is then

<u>var</u> s: <u>array</u>(2..n+1) <u>of</u> 1..b;

<u>var</u> in: <u>array</u>(2..n+1) <u>of</u> boolean

where the invariant I for this implementation of S is

$I \equiv 1 < b \leq n,$

$\quad i < i + s(i) \leq next(S \cup \{n+1\}, i),$ for $2 \leq i \leq n,$

$\quad s(n+1) = 1,$

$\quad in(i) \equiv i \in S \cup \{n+1\}$ for $2 \leq i \leq n+1$

S:= {2,...,n}:: i:= 1;

<u>do</u> i < n+1 → i:= i+1, s(i):= 1; in(i):= <u>true</u> <u>od</u>;

remove(S,i):: in(i):= <u>false</u>;

```
next(S,i):: var k,t,incr: 1..n;
            {result of loop: t = next (S,i)}
            {invariant of loop: I and i ≤ k < next (S,i) and
                                t = k + s(k) ≤ next (S,i)}
            k:= i; t:= k+s(k);
            do not in(t) → incr:= s(k) + s(t);
                           if incr ≤ b → s(k):= incr
                            [] incr > b → k:= t
                           fi;
                           t:= k+s(k)
            od;
            t
```

5. Discussion

Algorithm (2.9) also works for $n = 2$ and $n = 3$; it was just easier to present a proof for $n \geq 4$ only.

Dexter Kozen has discovered how to simply extend the algorithm to find the complete factorization of an integer n in no worse than linear time. This is not surprising, since Shank's [69] algorithm finds factors of n in time $O(N^{(1/4)+\varepsilon})$ for $\varepsilon > 0$. However Kozen's technique actually can be used to build a table in time n which yields the complete factorization of all integers between 2 and n! It is quite simple. Assume a singly-linked list is used to implement set S, say using approach 2, and use three new arrays $xp, xk, xq(2,\ldots,n)$. When a nonprime $x = p^k q$ is about to be deleted in algorithm (2.9) the values p,k and q are available, so just record them in $xp(x), xk(x)$ and $xq(x)$. Upon termination, for each i, $2 \leq i \leq n$, $in(i)$ indicates whether or not it is prime; if not prime, i's lowest prime factor is in $xp(i)$ and its multiplicity in $xk(i)$, while the other factors can be determined from $xq(i)$ in a similar fashion.

The development of this algorithm emphasizes several points. First, it could not have been developed without recognition of an important property of nonprimes -- their unique decomposition given in Theorem (2.1). Efficient algorithms come less from clever tricks than from a good understanding of properties of the values being manipulated. Secondly, the correctness of the algorithm rests on some nontrivial mathematical theorems (Lemmas 2.5-2.8). Once these theorems are understood, the algorithm itself seems quite simple. We see here a distinction between the complexity of an algorithm and the complexity of its mathematical underpinnings, two quite different things. Finally, the algorithm provides a good basis for a discussion of control structures and programming style -- we showed three ways of writing the algorithm -- and for a discussion of the selection of data structures. One sees here the usual time-space tradeoff and the use of "beneficial" side effects.

Acknowledgements

We note that Gale and Pratt [77] have discovered a different linear sieve algorithm. Thanks go to Jim Donahue, Don Knuth and Garry Levin for carefully reading an earlier draft of this paper and for providing many constructive criticisms.

References

Dijkstra, E.W. Guarded commands, nondeterminacy and formal derivation of programs. CACM 18 (August 1975), 453-457.

Gale, R., and V. Pratt. CGOL - an algebraic notation for MACLISP users. Working paper, MIT AI Lab., January 1977.

Knuth, D. The Art of Computer Programming - Volume 3/Sorting and Searching. Addison-Wesley, Menlo Park, California 1973.

Leveque, W.J. Topics in Number Theory, Volume I. Addison-Wesley, Reading, Massachussetts, 1956.

Miller. G.L. Riemann's hypothesis and tests for primality, Proc. Seventh Annual ACM Symposium on Theory of Computing (1975), 234-239.

Shank, D. Class number, a theory of factorization and Genera. Proc. Sym. in Pure Mathematics 20 (1969), American Math. Society (1971) 415-440.

Shaw, M., W.A. Wulf, and R.L. London. Abstraction and verification in ALPHARD: iterators and generators. Computer Science Dept., Carnegie-Mellon, August 1976.