A Case Study of Number-Theoretic Computation: Searching for Primes in Arithmetic Progression

Paul Pritchard

TR 82-504
July 1982

Department of Computer Science
Cornell University
Ithaca, New York 14853

1. Introduction

The area of number-theoretic computation, by which we mean the design and use of programs that compute interesting properties of the integers, presents special challenges to the programmer. In a realm where computations typically involve hundreds of hours of machine time, and the only requirement is to produce "interesting" results, the relationship between correctness and efficiency is rather less dominated by the former than is usually (and wisely) expected to be the case. Yet there is a dearth of published methodological material that the interested novice might consult for guidance, or even for enjoyment, in this area. There are plenty of results to be found, for instance in Maths. of Comp., but thorough presentations of the methods whereby those results were obtained are lacking[1].

This paper addresses this gap in the literature. We report on our experiences in the design of a suite of programs that tackle a "typical" (if that is possible) problem in computational number theory. Mostly, only very elementary number theoretic facts are exploited in our algorithms.

2. Background

In one of a famous series of papers, Hardy and Littlewood [8] generalized the well-known conjecture that there are infinitely many "twin primes", i.e., pairs of prime numbers whose difference is 2. Their conjectures rest on "probabilistic" arguments, and supply asymptotic formulae for the number of n-tuples of primes (of any type) with no member larger than x. The Hardy-Littlewood conjectures imply that for any n>1 there are infinitely many

[1] Perhaps the reason is that this area has been more the province of mathematicians than of computer scientists.

sequences of primes in arithmetic progression (PAPs) of length n.

However, very little has actually been established about such progressions. Roughly speaking, the present state of knowledge is that n can be 3 1/2. More precisely, Chowla [2] showed that there are infinitely many PAPs of length 3, and recently Grosswald [6] has established the validity of the Hardy-Littlewood formula in this case, and Heath-Brown [10] has shown that there are infinitely many arithmetic progressions (APs) consisting of 3 primes and an "almost prime" (a number with at most 2 prime factors).

With the aid of computers, PAPs have been discovered that are substantially longer than those guaranteed to exist by the best available theorems. As of writing, a PAP of length n (but which cannot be extended to have length n+1) is known for $2 \le n \le 17$ (see [16], [7, p.11]). The single known PAP of length 17 was found by Weintraub [17].

It is now possible to state our chosen problem. Put rather crudely, it is to break or at least equal Weintraub's record. That is, we want to find PAPs of length 17 or greater. In this paper we design algorithms to tackle the problem; we will report elsewhere on our computational experience.

3. Mathematical Preliminaries

Before proceeding, we introduce our

[illegible]: Lower case variables, e.g. x,y,a,b, range over the integers, and are non-negative unless otherwise stated. In sums and products, p ranges over the primes.

$x|y$: x divides y (exactly).

x **mod** y : the least $m \ge 0$ such that $y|(x-m)$.

$x \equiv a \pmod{m}$: $m|x-a$ -- congruence notation (see [12]).

$x \not| y$: **not** $(x|y)$

(x,y) : the greatest common divisor of x and y ($x,y \geq 1$).

"x and y are coprime" : $(x,y) = 1$.

$\pi(n)$: the number of primes $\leq n$.

$\Pi(n)$: the product of the primes $\leq n$.

$R(n)$: $\{x \mid 1 \leq x \leq n$ **and** $(x,n) = 1\}$ -- a reduced residue class of n (see [12]).

$\{f(k)\}_{k=k_1}^{k_2}$: the sequence $f(k_1)$, $f(k_1+1),\ldots,f(k_2)$ (the upper index is sometimes unspecified).

$(\exists! x:\ldots)$: there is a unique x such that ...

iff : if and only if.

Algorithms are written in guarded command notation [4,5] extended by a "for all" iterator:

"**forall** x: predicate(x) **do** statement-list **od**".

We start by investigating the properties of PAPs. The following theorem turns out to be of crucial importance.

__Theorem 1__ [9, Theorem 57]

If $(j,m) = d$, then the congruence

$$j \cdot x \equiv c \pmod{m}$$

is soluble iff $d|c$, and it then has just d solutions (mod m). □

We are looking for a PAP $\{a+k\cdot b\}_{k=0}^{n-1}$, $n>2$, so we must have $a>1$, $b>1$ and $(a,b) = 1$. Now let p be a prime such that $p \not| b$. Then $(p,b) = 1$. So, by Theorem 1, the congruence

$$p \cdot x \equiv a \pmod{b}$$

has just one solution (mod b). This implies that

$$(\exists! k\colon 0 \le k < p \ \textbf{and}\ p \mid a+k\cdot b)$$

Pick the k with this property, and suppose $p \le n$. Then either $a+k\cdot b$ is nonprime or $a+k\cdot b = p$. In the latter case, either $a<n$ and $a+a\cdot b$ is nonprime, or $a=p=n$. So $\{a+k\cdot b\}_{k=0}^{n-1}$ contains a nonprime number unless $a=p=n$, in which case $a+n\cdot b$ is nonprime. We have proved

<u>Theorem 2</u> (Waring/Mathieu, ca. 1860; see [3, p.425])

In order to have a PAP of length $n > 2$, the difference, b, between the primes must be divisible by $\Pi(n-1)$. Furthermore, if n is prime, $n \mid b$ unless the first term in the series is n. □

Theorem 2 implies that a PAP of length $n > 2$ must either be a sub-AP (a subsequence that is also an AP) of an AP $\{e+k\cdot\Pi(n)\}_{k\ge 0}$ for some $e \in R(\Pi(n))$, or be a sub-AP with first term n of the AP $\{n+k\cdot\Pi(n-1)\}_{k\ge 0}$. In order to be systematic, and to take advantage of the greater density of the primes among the smaller numbers, let us search for all PAPs of length $\ge n$ having no term greater than a given bound. It is convenient to let the bound be $(N+1)\cdot\Pi(n)$. To avoid unnecessary recalculations, we consider each AP mentioned above, determine the primality of each of its members, and then test all possible sub-APs.

Algorithm 1:

```
{n>2 and N>1}
forall e: e ∈ R(Π(n)) do
  S:= {e+k·Π(n)}_{k=0}^{N};
  mark all nonprime numbers in S:
     sift(S,e,n,N);
  search S {for a sub-AP of unmarked elements with length ≥ n}
od;
if prime(n) → S:= {n+k·Π(n-1)}_{k=0}^{⌊n·(N+1)-n/Π(n-1)⌋};
              sift(S,n,n-1,⌊n·(N+1)-n/Π(n-1)⌋);
              search S for a sub-AP of unmarked elements with first term n
              and length ≥ n
 ▯ not prime(n) → skip
fi
{the set of accepted APs is that of all PAPs of length ≥ n
 with no term > (N+1)·Π(n)}
```

The members of $R(\Pi(n))$ can be found by adapting the "wheel sieve" of [13,14].

To sift $S = \{e+k\cdot\Pi(n)\}_{k=0}^{N} = \{S_k\}_{k=0}^{N}$, it suffices to consider each prime p, $n<p \le \sqrt{e+N\cdot\Pi(n)}$, and mark each nonprime member of S with divisor p. (If $e = 1$, it too must be marked as nonprime.) By theorem 1, we need only find the unique solution of

$$k\cdot\Pi(n) \equiv -e \pmod{p} \textbf{ and } 0\le k<p \tag{3.1}$$

and then mark the elements $S_{k+i\cdot p}$, $i=0,1,2,\ldots$, with the exception of S_k if $S_k = p$. By theorem 1 again, there is a number $\text{inverse}(\Pi(n),p)$ such that

$$\Pi(n)\cdot\text{inverse}(\Pi(n),p) \equiv 1 \pmod{p}$$

(Given x,y such that $(x,y)=1$, $\text{inverse}(x,y)$ can be computed by adapting Euclid's greatest common divisor algorithm so that it computes a,b such that

$$a\cdot x + b\cdot y = \gcd(x,y) = 1,$$

whence inverse(x,y) may be taken as a -- see [11, p.274].) So the k of (3.1) is given by

$$k = -e \cdot \text{inverse}(\Pi(n),p) \textbf{ mod } p$$

Sift is given below. The required primes can be calculated (just once) by the methods of [15].

```
procedure sift(S,e,n,N):
   if e=1 → mark S_0  [] e≠1 → skip fi;
   forall p: prime(p) and n<p≤√((N+1)·Π(n)) do
      set k such that (3.1):
         k:= -e·inverse(Π(n), p) mod p;
      if S_k=p → k:= k+p  [] S_k≠p → skip fi;
      do k≤N → mark S_k;  k:= k+p od
   od
```

"search S" involves a linear scan of the AP $\{e+(h+k\cdot f)\cdot\Pi(n)\}_{k\geq 0}$, for each f,h such that $1 \leq f \leq \lfloor \frac{N}{n-1} \rfloor$ and $0 \leq h < f$, to see if n or more consecutive terms are unmarked (and hence prime). This amounts to checking S_h and every f'th element thereafter.

```
search S:
   forall f: 1≤f≤⌊N/(n-1)⌋ do
      forall h: 0≤h<f do
         i, count:= h, 0;
         do i≤N → if not marked(S_i) → count:= count+1
                  [] marked(S_i)     → if count≥n → accept(i-count·f,i-f,f)
                                        [] count<n → skip
                                       fi;
                                       count:= 0
                  fi;
                  i:= i+f
         od;
         if count≥n → accept(i-count·f, i-f, f)
          [] count<n → skip
         fi
      od
   od


procedure accept(i,j,f):
   {global: e,n}
   note that there is a PAP with first term e + i·Π(n),
      last term e + j·Π(n), and common difference f·Π(n)
```

The special search in the case that n is prime is similar to and simpler than the above, and is left to the reader. It is of much less importance than procedure search, which for all but very small values of n is expected to discover almost all the PAPs found by the algorithm.

Algorithm 1 is evidently correct in the sense that if there is a PAP of length n, then for some N>1 algorithm 1 will find that progression, and conversely. In its broad outline it is essentially the algorithm presented by Weintraub [16], which was (presumably) responsible for finding the PAP of length 17 reported in [17]. A great amount of computation is involved when looking for long PAPs. Weintraub [16] reports on a search with N = 16680 and

$n = 16$, so that $\Pi(n) = 30030$. The number of values of e to try -- the size of $R(\Pi(n))$ -- is 5760. He observes that

> The sieve itself proceeds quite rapidly on the computer while the search is more time-consuming.

4. A Rough Complexity Analysis

We might decide on the strength of Weintraub's observation to concentrate our efforts on speeding up the statement "search S". This section undertakes a simplified complexity analysis to get a more precise feel for the costs of the various components of algorithm 1.

The complexity of algorithm 1 is dominated by the **forall** loop, whose body has just 3 high-level statements. The cost of the assignment to S is $\theta(N)$ bit operations. Consider "sift(S,e,n,N)". For each prime p in the specified range, this involves a determination of k and then $\theta(N/p)$ markings. Since $\pi(n) \sim x/\log x$ -- the prime number theorem [9, Theorem 6] -- it can be shown that the cost of determining all the k-values $= O(\sqrt{N\cdot\Pi(n)})$ multiplications. The cost of the marking

$$= \sum_{n<p\leq\sqrt{e+N\cdot\Pi(n)}} N/p \cdot\theta(1) \quad \text{additions}$$

$$= \theta(N\cdot\log\log\sqrt{N\cdot\Pi(n)}) \quad \text{additions}$$

since $\sum_{p\leq x} 1/p \sim \log\log x$ -- [9, Theorem 427].

The cost of "search S" amounts to that of $\theta(N/n)$ complete passes over S, which is $\Theta(N^2/n)$ additions.

The relative contribution of these high-level statements to the complexity of algorithm 1 depends on the relationship between N and $\Pi(n)$. In the practical context of seeking progressions of length at least 17, the relationship should be determined by seeking to maximize the (minute) number of PAPs found per second under the constraints of the available computational resources. We have something to say about such matters in §7. For the present, let us note that even if $\Pi(n) \simeq N^2$, and it is unlikely to be this large (consider Weintraub's figures given above), "search S" costs $\Omega(\sqrt{N/\log N})$ times as many operations as does "sift S", because $\log \Pi(n) = \sum_{p \le n} \log p \sim n$ -- [9, Theorems 413 and 434].

In view of these facts, we decide to concentrate exclusively on speeding up the statement "search S". In our practical context, we expect that any gain in speed would accrue to the entire algorithm, because of the dominant cost of this statement.

5. A Basis for Improvement

Meanwhile, back at "search S" ... consider the typical subsearch -- a search for n or more consecutive terms of the following progression that are in S (and hence prime):

$$S^{f,h} = \{e+h\cdot\Pi(n)+k\cdot f\cdot\Pi(n)\}_{k\ge 0} = \{y_k\}_{k\ge 0} \tag{5.1}$$

We see that theorem 1 provides pertinent information. For let p be a prime such that p>n and $p \nmid f$. Then, by theorem 1, the congruence

$$p\cdot x \equiv e+h\cdot\Pi(n) \pmod{f\cdot\Pi(n)}$$

has just 1 solution (mod $f\cdot\Pi(n)$). This implies

$$(\exists! m:\ 0\le m<p \ \textbf{and}\ (\forall i{:}0\le i{:}\ p|y_{m+i\cdot p})) \tag{5.2}$$

Now let p_i, $i>0$, be the i'th smallest prime p such that $p>n$ and $p\nmid f$, and let m_i be the m asserted by (5.2) to exist for $p = p_i$. It is clear that the search for n or more successive terms of $S^{f,h}$ that are in S need only take place in the intervals between successive terms removed from S by p_1. Furthermore, if $2n>p_1$, then in every such interval I (with the possible exception of the first and last, which may be truncated) there is a critical subinterval I_c of size $2n-p_1+1$ such that if I contains a progression of n primes then every term in I_c is in S. This means that a search in I can advantageously start in I_c. The following theorem gives a necessary and sufficient condition for these critical regions to always exist.

<u>Theorem 3</u>

$$(\forall f{:}1\le f\le\lfloor\tfrac{N}{n-1}\rfloor{:}\ 2n>p_1(f)) \quad \text{iff} \quad N < (n-1)\cdot \prod_{n<p<2n} p$$

<u>Proof</u>: $N < (n-1)\cdot \prod_{n<p<2n} p \quad \text{iff} \quad \lfloor\frac{N}{n-1}\rfloor < \prod_{n<p<2n} p$

$$\text{iff} \quad (\forall f{:}1\le f\le\lfloor\tfrac{N}{n-1}\rfloor{:}\ f < \prod_{n<p<2n} p) \quad \text{iff} \quad (\forall f{:}1\le f<\lfloor\tfrac{N}{n-1}\rfloor{:}\ 2n>p_1(f)).\ \Box$$

This condition (on the r.h.s. of theorem 3) is very likely to attain in a practical context -- for n=17 it amounts to $N < 16\cdot19\cdot23\cdot29\cdot31 = 6285808$. Henceforth we assume that the condition holds.

Before writing the search, we must address the possibility that the first and last intervals are exceptional. They can actually be exceptional in two ways. The first, already mentioned, is that they might be truncated normal intervals; this can be handled in a straightforward manner with the help of sentinels. The second possibility is more awkward. It is that the first interval is potentially $\{y_k\}_{k=0}^{m_1+p_1-1}$, because $p_1 | y_{m_1}$ is consistent with $p_1 = y_{m_1}$. Given our desire for extreme efficiency, this presents a problem. It is therefore prudent to investigate more deeply, and we obtain

<u>Theorem 4</u>: If $n > 2$ and $p_1 = y_{m_1}$ then $m_1 = 0$.

<u>Proof</u>: Suppose $n > 2$ and $p_1 = y_{m_1}$, but that $m_1 \neq 0$.

We have $p_1 = e+h\cdot\Pi(n)+m_1\cdot f\cdot\Pi(n) \geq 1+f\cdot\Pi(n)$. Let p be the greatest prime $< p_1$, so that $p \leq n$ or $p|f$. Now $p_1 \leq \Pi(p)-1$ since $n>2$ implies $p_1>3$ implies $p>2$. Therefore $p\leq n$ is a contradiction, so $p|f$. This means that $p_1 \geq 1+p\cdot\Pi(n)>2p$ since $n>2$. But this contradicts Bertrand's Postulate [9, Theorem 418], that for each prime p there is a bigger prime < 2p. So $m_1 = 0$. □

Theorem 4 shows that the only case requiring special consideration is that of the first interval when $m_1 = 0$ and $y_0 = p_1$. If the search within each interval starts with a backwards search through the critical region, it will discover a progression of more than n primes starting at y_0. Let us therefore decide on this. The special case now reduces to that of a PAP of length exactly n starting at y_0 (with the greatest term in the critical region being nonprime).

If $p_1 = y_0 = e+h\cdot\Pi(n)$, it would appear that h must be very small. Suppose $h\geq 1$. Then $p_1>\Pi(n)\geq 2n$ for $n>4$ -- this can be easily shown using

Bertrand's Postulate. But this is a contradiction if $N<(n-1)\cdot\prod_{n<p<2n} p$, by Theorem 3. Thus h = 0, and we have established the following

<u>Corollary</u>: If $n>4$ and $N<(n-1)\cdot\prod_{n<p<2n} p$ and $p_1 = y_{m_1}$, then $p_1 = e$. □

We now know that special consideration is only necessary when h = 0 (and p_1=e) -- provided that $n>4$ and $N<(n-1)\cdot\prod_{n<p<2n} p$. Our theorems can easily be seen to guarantee the correctness of

<u>Algorithm 2</u>:

$\{n>4$ **and** $1<N<(n-1)\cdot\prod_{n<p<2n} p\}$

[As for algorithm 1 but with "search S" replaced by "ssearch S"[2]]

The refinement of "ssearch S" uses m_1, which is the size of the first (possibly empty) interval. From (5.2), we have

$$0\le m_1<p_1 \ \textbf{and}\ p_1 \,|\, e+h\cdot\Pi(n)+m\cdot f\cdot\Pi(n) \qquad (5.3)$$

Also, we use the notation S[i] to mean S_i is unmarked; this suggests the obvious implementation: a Boolean array.

[2]For "Smarter Search".

```
ssearch S:
  add sentinels:
     forall i: 1≤i≤⌊N/(n-1)⌋ do mark S_{-i};  mark S_{N+i} od;
  forall f: 1≤f≤⌊N/(n-1)⌋ do
     set p_1 = min{p | p prime and p>n and p∤f};
     forall h: 0≤h≤min{f-1, N-(n-1)·f} do
        set m_1 = min{m | m≥0  and  p_1|e+h·Π(n)+m·f·Π(n)};
        if m_1 ≥ n  →  lastc:= h+m_1·f+n·f-p_1·f
         [] m_1 < n  →  lastc:= h+m_1·f+n·f
        fi;
        {invariant: with the possible exception of a PAP of length n
            starting at y_0 when h=0, the PAPs of length ≥ n from S^{f,h}
            have been accepted except for those containing a term ≥ S_lastc
             and  {S_k}_{k=lastc-(2n-p_1)·f}^{lastc} ∩ S^{f,h} is a critical region}
        do  lastc ≤ N  →
             check interval:
               firstc:= lastc - (2n-p_1)·f;
               i:= lastc;
               do S[i] → i:= i-f od;
               if i≥firstc → skip {nonprime in critical region}
                [] i<firstc → firsti, i:= i+f, lastc+f;
                              do S[i] → i:=i+f od;
                              if (i-firsti)≥ n·f  → accept(firsti, i-f, f)
                               [] (i-firsti)<n·f    → skip
                              fi
               fi;
             get next interval:
               lastc:= lastc+p_1·f
        od
     od;
     if  p_1 = e  →
       check the AP e,e+f,...,e+(n-1)·f:
         i:= f;
         do S[i] → i:=i+f od;
         if i=n·f → accept(0, (n-1)·f, f)  [] i≠n·f → skip  fi
      [] p_1 ≠ e → skip
     fi
  od
```

This search is faster than the original by a factor of almost p_1, because it is probable that the first "look" in a critical region is unsuccessful. Some of the searching can be avoided as follows. Let

$$P_f = \{p \mid p \text{ prime } \mathbf{and}\ n<p\ \mathbf{and}\ p|f\}$$

Then if

$$(\exists p\colon p\epsilon P_f \quad \mathbf{and} \quad p|e+h\cdot\Pi(n))$$

then <u>all</u> members of $S^{f,h}$ are nonprime, so that value of h can be skipped. However, very little computation is saved unless n is near 2. The ideas of searching in intervals and searching first in the critical region are reminiscent of the idea underlying the fast pattern-matching algorithm of Boyer and Moore [1].

6. A Further Improvement

Our search is now confined to the intervals of $S^{f,h}$ between successive multiples of p_1, and advantageously concentrates on the critical regions of these intervals. But it is apparent that many of these critical regions will contain a nonprime multiple of p_2, and their intervals can therefore be ruled out of consideration; and of the remaining intervals, many will contain a nonprime multiple of p_3 in their critical regions; et cetera. Now the pattern of (y-indices of) members of $S^{f,h}$ that are multiples of at least one of $p_1, p_2, \ldots, p_r$ repeats modulo $\prod_{i=1}^{r} p_i$. This pattern in turn determines the pattern of those intervals that (only) need be searched because their critical regions do not contain a multiple of $p_2, p_3, \ldots, p_r$. This latter pattern repeats modulo $\prod_{i=2}^{r} p_i$. So precomputation of this pattern will save work if the number of intervals to be searched substantially exceeds the period of repetition, provided this information can be efficiently exploited.

But there is a much more compelling reason to do the precomputation. It is that many values of f will share the same values of $p_1, p_2, \ldots, p_r$, and hence the same pattern of potentially good intervals! For example, consider the case $n = 17$, and put $r = 3$. Then $\frac{18}{19} \cdot \frac{22}{23} \cdot \frac{28}{29} \cdot 100 = 87.5\%$ of the values of f have $\langle p_1, p_2, p_3 \rangle = \langle 19, 23, 29 \rangle$. Further, the proportion of potentially good critical regions (i.e., those that do not contain a multiple of $p_2, \ldots, p_r$) is

$$\prod_{i=2}^{r} \left(1 - \frac{2n - p_1 + 1}{p_i}\right) = \frac{5}{23} \cdot \frac{11}{29} \simeq \frac{1}{12 \cdot 1}$$

So fewer than one twelfth of the intervals need be examined, representing a substantial saving for these values of f. If in addition we precompute for the cases

$$\langle p_1, p_2, p_3 \rangle \in \{\langle 19,23,31 \rangle, \langle 19,29,31 \rangle, \langle 23,29,31 \rangle\},$$

over 99% of the values of f are catered for. (The respective proportions for these 3 classes $\simeq \frac{1}{11 \cdot 0}$, $\frac{1}{6 \cdot 3}$ and $\frac{1}{3 \cdot 5}$.) Note especially that these considerations are independent of e, so that the cost of the precomputing is negligible.

We would like to start our search of $S^{f,h}$ in a potentially good interval, and proceed directly (i.e. in O(1) operations) to the next potentially good interval, and so on. Therefore we introduce, for each r-tuple $\langle p_1, p_2, \ldots, p_r \rangle$ that is considered,

$$\textbf{var}\ \ \mathrm{nextpgi} : \textbf{array}\ [0..-1+\prod_{i=2}^{r} p_i]\ \textbf{of}\ \ 1..-1+\prod_{i=2}^{r} p_i$$

such that

$(\forall i{:}0\le i<\prod_{i=2}^{r} p_i$: the first interval between multiples of p_1 that occurs after the j'th interval $j\cdot p_1..(j+1)\cdot p_1$ and has no multiple of $p_2,\ldots,p_r$ in its critical region is the (j+nextpgi[j])'th interval) (6.1)

Then if the current interval is the t'th modulo $\prod_{i=2}^{r} p_i$, we can proceed directly to the next potentially good interval, and update t, by means of the following refinement:

get next interval:

$$t,lastc := (t+nextpgi[t]) \textbf{ mod } \prod_{i=2}^{r} p_i,\ lastc+nextpgi[t]\cdot p_1\cdot f$$

We now determine the interval number of the first potentially good interval in which to search, i.e., the initial value of t. Let t_0 be the interval number of the first (possibly incomplete) interval of $S^{f,h}$. If $m_1\ge n$, the first potentially good interval to be considered has the interval number

$$t_0-1+nextpgi[t_0-1].$$

Otherwise, it has the interval number

$$t_0+nextpgi[t_0].$$

Now t_0 is determined by the quantities d_i, for $2\le i\le r$, where from (5.2) we have

$$0\le m_i<p_i \quad \textbf{and} \quad p_i | e+h\cdot\Pi(n)+m_i\cdot f\cdot\Pi(n),\quad 1\le i\le r, \qquad (6.2)$$

and the d_i are defined by

$$d_i = (m_i-m_1) \textbf{ mod } p_i,\quad 2\le i\le r. \qquad (6.3)$$

In order to compute t_0 quickly, we employ an array dp that maps the (r-1)-tuple $<d_2, \ldots, d_r>$ to the associated interval number. In view of (6.2) and (6.3), we define the d_i for the j'th interval $j \cdot p_1 .. (j+1) \cdot p_1$ to be the difference between the least multiple of $p_i \geq (j+1) \cdot p_1$ and $(j+1) \cdot p_1$. We thus have

$$(\forall d_2 \ldots d_r : (\forall i: 2 \leq i \leq r: 0 \leq d_i < p_i):$$
$$(\forall i: 2 \leq i \leq r: p_i | (dp[d_2, \ldots, d_r]+1) \cdot p_1 + d_i)) \qquad (6.4)$$

We now present a procedure that establishes properties (6.1) and (6.4) of arrays nextpgi and dp. It operates on parameters N,n,r,p,dp and nextpgi. N and n are as for algorithm 2. Array p and integer r > 1 must satisfy

$$(\exists f: 1 \leq f \leq \lfloor \frac{N}{n-1} \rfloor : (\forall i: 1 \leq i \leq r: p[i] = p_i(f))) \qquad (6.5)$$

The algorithm employs an array d and interval number t such that

$$(\forall i: 2 \leq i \leq r: 0 \leq d[i] < p_i \textbf{ and } p_i | t \cdot p_1 + d[i]) \qquad (6.6)$$

```
procedure  setpgi(N,n,r,p,dp,nextpgi):
   g:= 0;
   establish (6.6):
      t:= 1;
      i:= 2; do i ≤ r → d[i], i:= p[i]-p[1], i+1 od;
   {invariant:  the nearest potentially good interval before the t'th is the g'th
          and  (6.6)  and  [(6.1) with t as the upper bound for i]  and
            dp satisfies (6.4) when the range of dp is restricted to 0..t-2 }
             r
   do g ≠  Π p[i] →
           i=2
      i:= 2;
      {invariant: (∃j:2≤j<i: d[j] < p[1]-n  or  d[j] > n)}
      do i ≠ r+1 cand (d[i]<p[1]-n or d[i]>n) → i=i+1 od;
      if i = r+1 → {the t'th interval is potentially good}
                   i:= g;
                   do i<t → nextpgi[i]:= t-i; i:= i+1 od;
                   g:= t
       [] i ≤ r → skip
      fi;
      dp[d[2],...,d[r]]:= t-1;
      t:= t+1;
      re-establish (6.6):
        i:= 2;
        do i ≤ r → d[i], i:= (d[i]-p[i]) mod p[i], i+1 od
   od
```

Note that this procedure works correctly even if r = 1 (taking $\prod_{i=2}^{r} p[i]$ to be 1 and dp[d_2, . . . ,d_r] to be dp[0]).

Before presenting the new algorithm, we must address the possibility that a PAP of length ≥ n is missed. This can happen in two ways. The first was treated in algorithm 2 -- we must search for a PAP of length exactly n when $e = p_1$ and $h = 0$, for such a PAP would not otherwise be accepted. The second way is new. It is that a PAP is missed because p_i occurs in a critical region, for some i, $2 \le i \le r$. We would then have

$$p_i = e+h\cdot\Pi(n)+m_i\cdot f\cdot\Pi(n)$$

Under the preconditions of algorithm 2, the proof of the corollary to theorem 4 shows that $p_1<\Pi(n)$. If we further require $p_r<\Pi(n)$, this case can occur only when $e = p_i$ and $h = 0$. Also, e must be the first member of a critical region, so that the PAP must have length exactly n.

The two cases are considered conjointly in the new algorithm. Note that for n = 17, the requirement that $p_r<\Pi(17)$ is hardly a restriction at all.

<u>Algorithm 3</u>:

$\{n>4$ **and** $1<N<(n-1)\cdot \prod_{n<p<2n} p\}$

[As for algorithm 2 but with "ssearch S" replaced by "sssearch S"[3]]

[3]For "Still Smarter Search".

```
sssearch S:
  add sentinels:
     forall i: 1≤i≤⌊N/(n-1)⌋ do mark S_{-i};  mark S_{N+i} od;
  forall f: 1≤f≤⌊N/(n-1)⌋ do
     r:= r(f);
     forall i: 1≤i≤r do set p_i = min{p | p prime and p>n and p∤f} od;
     P:= Π_{i=2}^{r} p_i;
     if p_r < Π(n) → setpgi(N,n,r,p,dp,nextpgi) fi;
     forall h: 0≤h≤min{f-1, N-(n-1)·f} do
        forall i: 1≤i≤r do set m_i = min{m | m≥0 and p_i|e+h·Π(n)+m·f·Π(n)} od;
        forall i: 2≤i≤r do set d_i = (m_i-m_1) mod p_i od;
        t:= dp[d_2, . . . ,d_r];
        if m_1 ≥ n →
            if t≠0 → lastc, t:= h+m_1·f+n·f-p_1·f+(nextpgi[t-1]-1)·p_1·f,
                                (t-1+nextpgi[t-1]) mod P
             ▯ t=0 → lastc:= h+m_1·f+n·f-p_1·f
            fi
         ▯ m_1 < n →
            lastc, t:= h+m_1·f+n·f-p_1·f+nextpgi[t]·p_1·f, (t+nextpgi[t]) mod P
        fi;
        do lastc ≤ N →
           check interval;  {see ssearch}
           get next interval:
             lastc, t:= lastc+nextpgi[t]·p_1·f, (t+nextpgi[t]) mod P
        od
     od;
     if e ∈ {p_i|1≤i≤r} → check the AP e,e+f,...,e+(n-1)·f {see ssearch}
      ▯ e ∉ {p_i|1≤i≤r} → skip
     fi
  od
```

7. Practical Considerations

Rather a lot has happened since the complexity analysis of §4. It is time to take stock. Let us fix n = 17. The argument at the start of §6 suggests that it is reasonable to precompute dp and nextpgi arrays for the four

3-tuples $<p_1, p_2, p_3>$ given there. This means that r = 3 for over 99% of the f-values. (Choosing r=4 would lead to unacceptably large space requirements.) We might then expect that sssearch be 2 orders of magnitude faster than the search in algorithm 1. And experimentation indeed shows that the number of lookups of S[i] is reduced by a factor of 96. This would appear to be good enough to outweigh the extra work involved in the statement "get next interval" of sssearch.

However, a substantial overhead has been introduced: for each value of f in $1..\lfloor\frac{N}{n-1}\rfloor$, and for each h in 0..f-1, a non-trivial amount of computation -- say c>>1 operations -- is done (including the as yet unspecified calculation of the r m_i). This overhead amounts to

$$\sim c\cdot\frac{N^2}{(n-1)^2}$$

operations, and thus overwhelms the time actually spent in examining S! The trouble is that as f approaches its maximum value the number of intervals between multiples of p_1 approaches 1, and it becomes increasingly likely that none of these intervals are potentially good. It is literally a waste of time to do the c operations needed to discover this.

Yet all is not lost. For recall Weintraub's observation that searching took far more time than sifting. This suggests a way to escape from our dilemma -- we will substantially increase N while keeping the maximum value of f fixed. The effect is that although more time is spent sifting, the overhead is unchanged and the speedup in sssearch takes effect. We can experiment to find the point at which the number of looks at S per second is maximized. Provided the cost of sifting does not become too large, we might expect to set fmax, the maximum value of f, so that the number of intervals between multiples of p_1, which $\simeq$ N/fmax, is roughly equal to the inverse of the expected proportion of potentially good intervals. Our experiments confirm this expectation.

Now that N has been increased, a space-efficient implementation of S is a necessity. Recalling that sentinels are needed at both ends of S, we use

$$\textbf{var}\ S\text{:}\ \ \textbf{array}\ [0..\lfloor(N+2\cdot fmax)/ss\rfloor]\ \textbf{of set of}\ 0..ss\text{-}1$$

so that S[i] (i.e., the truth-value of "S_i is not marked") becomes

$$i+fmax\ \textbf{mod}\ ss\ \in\ S[\lfloor(i+fmax)/ss\rfloor]$$

On machines whose wordlength is equal to (or just exceeds) a power of 2, marking and testing S_i can be done very efficiently using arithmetic shifts and logical masking operations.

Knowing now that it is the loop over all values of h that dominates the computation time of sssearch, we try to make the body of that loop as efficient as possible. Consider first the computation of the m_i, for $1 \le i \le r$. From the defining relation (6.2) we have

$$e+h\cdot\Pi(n)+m_i\cdot f\cdot\Pi(n)\ \equiv\ 0\ (\text{mod}\ p_i)$$

After appealing twice to theorem 1 to guarantee the required inverses, we have

$$m_i = -(h+e\cdot inverse(\Pi(n), p_i))\cdot inverse(f, p_i)\ \textbf{mod}\ p_i \tag{7.1}$$

Since the set of all p_i used in sssearch can be determined in advance, and is quite small, we precompute a table

$$inverse_{p_i}[x] = inverse(x, p_i),\ \ x \in 1..p_i-1$$

and a value

$$\mathrm{invPIn}_{p_i} = \mathrm{inverse}(\Pi(n), p_i)$$

for each possible p_i. Then the computation of m_i reduces to

$$m_i := (-(h+e\cdot \mathrm{invPIn}[i])\cdot \mathrm{invfp}[i]) \textbf{ mod } p_i \tag{7.2}$$

where

$$\mathrm{invfp}[i]=\mathrm{inverse}_{p_i}[f \textbf{ mod } p_i]$$

and

$$\mathrm{invPIn}[i] = \mathrm{invPIn}_{p_i}$$

are set in the loop over the f-values. The d_i can then be directly computed from definition (6.3).

A substantial speedup of the above method can be had by thoroughly exploiting recurrence relations. Suppose each new value of h is obtained by incrementing the previous value h'. Then writing m_i and m_i' for the corresponding values of m_i, we have from (7.1)

$$m_i = (m_i' - \mathrm{inverse}(f, p_i)) \textbf{ mod } p_i \tag{7.3}$$

It follows that we need only use (7.2) to compute m_i for h = 0 (or -1), and then adapt (7.3) to update m_i after each increment of h. But the values actually needed in sssearch are just m_1 and d_i, $2 \le i \le r$. By reapplying the above technique to the defining equations (6.3) and (7.3), we are finally lead to the code given below.

```
m1:= m1-invfp[1];
if m1≥0 → forall i: 2≤i≤r do
              di:= di-dinc1[i];
              if di<0 → di:= di+pi  [] di≥0 → skip  fi
           od
 [] m1<0 → forall i: 2≤i≤r do
              di:= di-dinc2[i];
              if di<0 → di:= di+pi  [] di≥0 → skip  fi;
              m1:= m1+p1
           od
fi
```

where arrays dinc1 and dinc2 are set in the f-loop so that

$$\text{dinc1}[i] = (\text{finvp}[i]-\text{finvp}[1]) \ \mathbf{mod}\ p_i, \quad 2\le i\le r,$$

$$\text{dinc2}[i] = (\text{dinc1}[i]+p_1) \ \mathbf{mod}\ p_i, \quad 2\le i\le r.$$

The h-loop can be improved further by delaying the update of t until it is needed.

Now that a significant amount of time is spent in sifting, it is worthwhile to pay equally careful attention to procedure sift. We do not expect that there is a substantial algorithmic improvement on sift (unlike the case for procedure search). So we concentrate for the present on an efficient implementation. First note that the same inverses are calculated over and over again for each value of e. We avoid this by precomputing these inverses and reading them in as required.

There is another matter to be addressed. It is that two of the calculations may lead to arithmetic overflow -- the product e•inverse(Π(n), p) and the term S_k. The former is dealt with by the following refinement:

```
set k such that (2.1):
   k:= prodmod(-e,inverse(Π(n),p),p)
```

Function prodmod(a,b,p) computes $a \cdot b$ **mod** p. It is presented in the appendix. To handle the latter, first note that

$$S_k = p \equiv e + k \cdot \Pi(n) = p$$

$$\equiv \lfloor (p-e)/\Pi(n) \rfloor = k \ \textbf{and} \ (p-e) \ \textbf{mod} \ \Pi(n) = 0$$

Both the quotient and remainder on division of (p-e) by $\Pi(n)$ can be efficiently maintained by taking the p-values in increasing order and using recurrence relations. However we do not use this method in our program for n=17, instead favouring a crude but effective test -- we have $N \le \Pi(n)$, so that $p \le \Pi(n)$ and the test $S_k = p$ becomes p = e. This enables the test to be removed altogether, provided only that the loop is followed by the statement

if e is prime → unmark S_0 [] e is not prime → skip **od**

Having thus optimized procedure sift, the only way to spend less time sifting is to actually do less sifting. Aha! Since PAPs of length at least 17 are extremely rare, so also must be psuedo-PAPs of length at least 17, where a psuedo-PAP is an AP of numbers which have only "large" prime factors. So we decide to initially sift only up to a certain fraction λ, $0<\lambda\le 1$, of the maximum prime otherwise required, and to sift further if necessary when a psuedo-PAP is "accepted". The value of λ is to be determined by experimentation -- it should be reduced until the time saved in sifting after a further reduction is offset by the extra time spent in searching.

The program can be given a final fine-tuning by removing constant expressions from inner loops, and by hand-coding the innermost loops (which are quite small) in assembly language. Our program , with n=17, N=300000 and fmax=1000, processes each value of e in a little over 2 minutes on a VAX 11/780 running under Berkeley Unix.

As a final point, we note that of the 7 PAPs of length 16 reported in [16], all have a common difference divisible by $\Pi(17)$ (rather than just $\Pi(16)$), and that the sole known PAP of length 17 has a common difference divisible by $\Pi(19)$. We know of no explanation for this fascinating phenomenon (but hope to throw some light on it by experiments with the program outlined herein). However, it does suggest that our program be generalized so that while $\Pi(n)$ is used for the sieve, the search is made for a PAP of length m, $m \le n$. It is not difficult to generalize our argument so that it handles this situation. We have done this, and our program is parameterized with respect to m as well as n. However, there are several practical reasons for choosing $m=n=17$ rather than, say, $m=17$ and $n=19$, in addition to the fact that the latter search is not systematic. In the latter case much more time is needed to sift, and the factor of reduction in the number of looks at S, at 49, is rather less than the factor of 96 in the former case. Still ...

References

[1] Boyer, R.S. and J.S. Moore. A fast string searching algorithm. Comm. ACM 20, 10 (October 1977), 762-772.

[2] Chowla, S. There exist an infinity of 3-combinations of primes in A.P. Proc. Lahore Philos. Soc. 6, 2 (1944), 15,16.

[3] Dickson, L.E. History of the Theory of Numbers, vol.1. Chelsea Publishing Co., New York, 1971.

[4] Dijkstra, E.W. A Discipline of Programming. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[5] Gries, D. The Science of Programming. Springer-Verlag, Berlin Heidelberg and New York, 1981.

[6] Grosswald, E. Arithmetic progressions that consist only of primes. J. Number Theory 14, (1982), 9-31.

[7] Guy, R.K. Unsolved Problems in Number Theory. Springer-Verlag, New York Heidelberg and Berlin, 1981.

[8] Hardy, G.H. and J.E. Littlewood. Some problems of "partitio numerorum" III: on the expression of a number as a sum of primes. Acta Mathematica 44 (1923), 1-70.

[9] Hardy, G.H. and E.M. Wright. An Introduction to the Theory of Numbers. 5th Ed., Oxford University Press, Oxford, England, 1979.

[10] Heath-Brown, D.R. Three primes and an almost prime in arithmetic progression. J. London Math. Soc. (2nd series) 23, 3 (June 1981), 396-414.

[11] Knuth, D.E. The Art of Computer Programming, vol. 2: Seminumerical Algorithms. 2nd Ed., Addison-Wesley, Reading, Massachusetts, 1981.

[12] Le Veque, W.J. Elementary Theory of Numbers. Addison-Wesley, Reading, Massachusetts, 1965.

[13] Pritchard, P. A sublinear additive sieve for finding prime numbers. Comm. ACM 24, 1 (January 1981), 18-23.

[14] Pritchard, P. Explaining the wheel sieve. To appear in Acta Informatica.

[15] Pritchard, P. Fast compact prime-number sieves (among others). Tech. Report 81-473, Dept. of Computer Science, Cornell University, October 1981.

[16] Weintraub, S. Primes in arithmetic progression. B.I.T. 17 (1977), 239-243.

[17] Weintraub, S. Seventeen primes in arithmetic progression. *Maths. of Comp.* **31**, 140 (October 1977), 1030.

Appendix

The function

$$\text{prodmod}(a,b,p) = (a\cdot b) \textbf{ mod } p$$

can be derived from the following two facts:

$$\text{prodmod}(2a',b,p) \quad = \text{prodmod}(a',2b \textbf{ mod } p,p)$$

$$\text{prodmod}(2a'+1,b,p) = (\text{prodmod}(2a',b,p)+b) \textbf{ mod } p$$

A recursive formulation is immediate, but the facts also readily suggest an invariant assertion for an iterative version.

```
function prodmod(a,b,p):
   {in: a=a0 and b=b0 and p>0}
   {returns a0·b0 mod p}
   a,b,c:= a mod p,b mod p,0;
   {invariant: (a·b+c) mod p = a0·b0 mod p and 0≤a,b,c<p}
   do a≠0 and even(a) → a,b:= a/2, 2*b mod p
    [] odd(a)          → a,c:= a-1, (b+c) mod p
   od;
   return(c)
```

Note that if $p \le \lfloor \text{maxint}/2 \rfloor+1$, where maxint is the greatest representable integer, then overflow cannot occur in prodmod. It is possible, but messy, to remove this requirement by weakening it to $p \le$ maxint, by exploiting the fact that

$$x \textbf{ mod } p = -(p-x \textbf{ mod } p) \textbf{ mod } p$$

In our application, we did not need and therefore did not choose to do this; we instead exploited the same fact to minimize a. Two further modifications were made to increase efficiency. First, the multiplicative **mod** operations in the loop were replaced by additive ones; the restriction on a,b and c in the invariant makes this possible. Second, the loop is terminated as soon as it is possible to complete the calculation without the possibility of overflow occurring. The final form of the algorithm is given below.

```
function prodmod(a,b,p):
   {in: a=a0 and b=b0 and p>0}
   {returns a0·b0 mod p}
   a,b,c:= a mod p,b mod p,0;
   if p-a<a → a, neg:= p-a, True  [] p-a≥a → neg:= False fi;
   if p-b<b → b, neg:= p-b, not neg  [] p-b≥b → skip  fi;
   if b<a → a, b:= b, a  [] b≥a → skip  fi;
   {invariant: (a·b+c) mod p = a0·b0 mod p and 0≤a,b,c<p}
   do a≥⌊maxint/(p-1)⌋ and even(a) → a,b:= a/2, 2*b;
                                       if b>p → b:= b-p  [] b≤p → skip  fi
    [] odd(a)                        → a,c:= a-1, b+c;
                                       if c>p → c:= c-p  [] c≤p → skip  fi
   od;
   c:= (a·b+c) mod p;
   if neg and c≠0 → return(p-c)  [] (not neg) or c=0 → return(c) fi
```