A CASE STUDY OF NUMBER-THEORETIC
COMPUTATION: SEARCHING FOR PRIMES IN
ARITHMETIC PROGRESSION

Paul A. PRITCHARD
Department of Computer Science, Cornell University, Ithaca, NY 14853, U.S.A.

Communicated by J. Nievergelt
Received July 1982
Revised April 1983

1. Introduction

The purpose of this paper is to present a thorough case study of the programming
process. The problem domain is number theory; we design programs to add to
what is already known about certain arrangements of natural numbers. This kind
of number-theoretic computation presents special challenges to the programmer.
Computations can involve thousands of hours of computer time, so efficiency is a
primary concern. Specifications are typically loose—they often amount to the vague
request "Produce interesting results"—and also are typically formulated by the
programmer. The relationship between correctness and efficiency is therefore 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 in this area that the
interested novice might consult. There are plenty of results to be found, for instance
in Mathematics of Computation, but thorough presentations of the methods whereby
those results were obtained are rare.

We address this gap in the literature, and take the opportunity to illustrate the
programmer's concerns from a wider perspective than is often taken. 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. The form of our report is
not a formal statement of the problem followed by a derivation of a beautiful
solution. Rather, the problem itself is only loosely defined at the outset, since the
form of our contribution will depend on what we can do well. Also, several potential
solutions (or, more accurately, avenues of approach) are derived, and we present
(in Section 9) a detailed discussion of how best to choose between these competing
approaches. We feel that the combination of practical complexity analysis and
parameter optimization that is involved in this choice is an important and under-
appreciated aspect of the programmer's task.
Mostly, only very elementary number theoretic facts are exploited in our algorithms (which are developed from first principles), although analyses of their computational complexity draw on 'deep' theorems. We hope to give a convincing and detailed account of all the issues involved in the design and performance of actual computer experiments, and to show that the very practical considerations involved do not preclude—but rather benefit from—a disciplined and systematic approach.

2. Background

In one of a famous series of papers, Hardy and Littlewood [9] generalized the well-known conjecture that there are infinitely many 'twin primes', i.e., pairs of prime numbers whose difference is two. Their conjectures are supported by heuristic arguments, and imply that for any \( n > 1 \) there are infinitely many sequences of primes in arithmetic progression (PAPS) of length \( n \). Moreover, Hardy and Littlewood give a conjectured asymptotic formula for the number of PAPS of length \( n \) with all terms \( \leq x \). (Grosswald [6] shows that with an additional assumption this formula can be cast in an easily computable form (see Section 9), and we shall make use of this later.)

However, very little has actually been established about such progressions. Roughly speaking, the present state of knowledge is that \( n \) can be \( 3 \frac{1}{2} \). More precisely, Chowla [2] showed that there are infinitely many PAPS of length three, and recently Grosswald [6] has established the validity of the Hardy-Littlewood asymptotic formula in this case, and Heath-Brown [11] has shown that there are infinitely many arithmetic progressions (APs) consisting of three primes and an 'almost prime' (a number with at most two 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. Before undertaking our computations, a PAP of length \( n \) (but which cannot be extended to have length \( n + 1 \)) was known only for \( 2 \leq n \leq 17 \) (see [7], [8, topic A5], [17]). The single PAP of length 17 was found by Weintraub [18].

It is now possible to state our chosen problem. It is to gather data on the distribution of 'long' PAPS, and hopefully in the process to break Weintraub's record. In this paper we design programs to tackle the problem and report briefly on our initial computational experience with them.

3. Mathematical preliminaries and a basic algorithm

Before proceeding, we introduce our notation:
- 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 \( m \geq 0 \) such that \( y \mid (x - m) \). Nota bene \( m \geq 0 \).

x div y: the integer part of \( x/y \).

\( x \equiv a \mod m \): \( m \mid x - a \) — congruence notation (see [13]).

\( x \not\equiv y \mod m \): not \( x \equiv y \).

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

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

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

\( R(n) \): \( \{x \mid 1 \leq x \leq n \text{ and } (x, n) = 1\} \) — a reduced residue system (mod \( n \)) (see [13]).

\( a..b \): \{ \( x \mid a \leq x \leq b \) \}.

\( \{f(k)\}_{k=1}^{n} \): the sequence \( f(k_1), f(k_1 + 1), \ldots, f(k_n) \) (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 (nondeterministic) ‘for all’ iterator:

\[
\text{forall } x: \text{predicate}(x) \text{ do statement-list od.}
\]

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

**Theorem 1** ([10, Theorem 57]). If \( (j, m) = d \), then the congruence

\[
j \cdot x \equiv c \mod m
\]

is soluble iff \( d \mid c \), 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 \( p \not\equiv \pm b \). Then \( (p, b) = 1 \). So by Theorem 1 the congruence

\[
b \cdot x \equiv -a \mod p
\]

has just one solution (mod \( p \)). This implies that

\[
(\exists!k: 0 \leq k < p \text{ and } p \mid a + k \cdot b).
\]

Pick the \( k \) with this property, and suppose \( p \leq n \). Then either \( a + k \cdot b \) is composite or \( a + k \cdot b = p \). In the latter case, either \( a < n \) and \( a + a \cdot b \) is composite, or \( a = p = n \).

So \( \{a + k \cdot b\}_{k=0}^{n-1} \) contains a composite number unless \( a = n \), which case \( a + n \cdot b \) is composite. We have proved

**Theorem 2** (Waring/Mathieu, ca. 1860; see [3, p. 425]). *In a PAP of length \( n > 2 \), the difference \( b \) between the primes is divisible by \( \Pi(n - 1) \). Furthermore, if \( n \) is prime then \( n \mid b \) unless the first term 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 $\{e + k \cdot \Pi(n)\}_{k=0}^{N}$ for some $e \in R(\Pi(n))$, or be a sub-AP with first term $n$ of the AP $\{n + k \cdot \Pi(n-1)\}_{k=0}^{N}$. 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 $\geq n$ 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.**

$$\{n > 2 \text{ and } N \geq n\}$$

for all $e: e \in R(\Pi(n))$ do

$$S := \{e + k \cdot \Pi(n)\}_{k=0}^{N};$$

mark all nonprime numbers in $S$:

sift$(S, e, n, N);$ search $S$ for a sub-AP of unmarked elements with length $\geq n$

od;

if $n$ is prime $\rightarrow S := \{n + k \cdot \Pi(n-1)\}_{k=0}^{N-(N+1)-n \div \Pi(n-1)}$;

sift$(S, n, n-1, n \cdot (N+1)-n \div \Pi(n-1))$;

search $S$ for a sub-AP of unmarked elements with first term $n$ and length $\geq n$

$\Box$ $n$ is composite $\rightarrow$ skip

{the set of accepted APs is that of all PAPs of length $\geq n$ with no term $> (N+1) \cdot \Pi(n)$, and no PAP is accepted twice}

The members of $R(\Pi(n))$ can be found by adapting the ‘wheel sieve’ of [14, 15], or just generated as needed by testing successive odd numbers.

To sift

$$S = \{e + k \cdot \Pi(n)\}_{k=0}^{N} \overset{df}{=} \{S_k\}_{k=0}^{N}$$

it suffices to consider each prime $p$, $n < p \leq \sqrt{e+N \cdot \Pi(n)}$, and mark each composite member of $S$ that has divisor $p$. (If $e = 1$, it too must be marked.) By Theorem 1, we need only find the unique solution of

$$k \cdot \Pi(n) = -e \pmod{p} \quad \text{and} \quad 0 \leq k < p \quad (1)$$

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

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

(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

$$a \cdot x + b \cdot y = (x, y) = 1,$$
Searching for primes in arithmetic progression

whence \( \text{inverse}(x, y) \) may be taken as \( a \) — see [12, p. 274]. So the solution of (1) is

\[
k = -e \cdot \text{inverse}(\Pi(n), p) \mod p.
\]

Procedure \textit{sift} is given below. The required primes can be calculated (just once) by the methods of [16].

\begin{verbatim}
procedure sift(S, e, n, N):
    if \( e = 1 \) \rightarrow \text{mark} S_0 \neq e \neq 1 -> \text{skip fi};
    forall \( p : p \) is prime and \( n < p \leq \sqrt{1 + N \cdot \Pi(n)} \) do
        set \( k \) such that (1):
        \[
        k := -e \cdot \text{inverse}(\Pi(n), p) \mod p;
        \]
        if \( S_k = p \rightarrow k := k + p \quad \text{if} \quad S_k \neq p \rightarrow \text{skip fi};
        \]
        \text{do} \( k \leq N \rightarrow \text{mark} S_k; k := k + p \) \text{od}
    \text{od}

Search \( S \) involves a linear scan of the AP \( \{S_{h+kf}\}_{k=0}^\infty \), for each \( f, h \) such that \( 1 \leq f \leq N \div (n-1) \) and \( 0 < h \leq \min\{f-1, N-(n-1) \cdot f\} \), 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.

\begin{verbatim}
search S:
    forall \( f: 1 \leq f \leq N \div (n-1) \) do
        forall \( h: 0 < h \leq \min\{f-1, N-(n-1) \cdot f\} \) do
            \( i, count := h, 0; \)
            \text{do} \( i \leq N \rightarrow \text{if not marked}(S_i) \rightarrow count := count + 1 \)
            \( \quad \text{marked}(S_i) \rightarrow \text{check}(count, i, f); \)
            \( \quad count := 0 \).
            \text{fi;}
            \( i := i + f \)
        \text{od;}
        \text{check}(count, i, f)
    \text{od}
\text{od}

procedure check(count, i, f):
    \{global: \( S, n \} \)
    \{\[S_{i-(\text{count}-k)f}\]_{k=0}^{\text{count}-1} \text{ is a PAP}\}
    if \( \text{count} \geq n \rightarrow \text{accept}(i - \text{count} \cdot f, i - f, f)
    \quad \text{fi}
    \text{skip}\fi

procedure accept(i, j, f):
    \{global: \( e, n \} \)
    Note that there is a PAP with first term \( e + i \cdot \Pi(n) \), last term \( e + j \cdot \Pi(n) \),
    and common difference \( f \cdot \Pi(n) \)
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 in any case of much less importance than search \( S \), which for all but very small values of \( n \) or very large values of \( N \) is expected to discover all the PAPs found by the algorithm. In practice the special search may not be performed.

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 [17], which was (presumably) responsible for finding the PAP of length 17 reported in [18]. A great amount of computation is involved when looking for long PAPs. Weintraub [17] reports on a search with \( N = 16,680 \) and \( n = 16 \), so that \( \Pi(n) - 30,030 \). The number of values of \( e \) to try equals the cardinality of \( R(\Pi(n)) \) equals \( \phi(\Pi(n)) \) equals 5720. (\( \phi( \ ) \) denotes Euler's function (see [13]).) Weintraub 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 rough 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 for all loop, whose body has just three high-level statements. The cost of the assignment to \( S \) is \( \Theta(N) \) additions or possibly bit operations depending on the implementation of \( S \). 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. Given the analysis of Euclid's algorithm in [12], and the fact that \( \pi(n) \sim n/\log n \)—this is the celebrated prime number theorem [10, Theorem 6]—it follows that the cost of determining all the k-values =\( \Theta(\sqrt{N} \cdot \Pi(n)) \) multiplications, and can be less than this order only by a logarithmic factor. The cost of the marking is

\[
\sum_{n < p < \sqrt{e + N(\Pi(n))}} N/p \cdot \Theta(1) = \Theta(N \cdot \log(\log(N \cdot \Pi(n))/\log n))
\]

additions since

\[
\sum_{p \leq x} p^{-1} \sim \log \log x + B + o(1)
\]

where \( B \) is a constant—[10, Theorem 427].

The cost of search \( S \) is approximately that of \( N/(n-1) \) complete passes over \( S \), which is \( \Theta(N^2/n) \) additions. Note that all the constants implicit in our \( \Theta \)-bounds are small.
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 'long' PAPs, the relationship might be determined by seeking to maximize the (minute) number of PAPs found per second under the constraints of the available computational resources. We address such matters in Section 9. For the present, let us note that even if \( \Pi(n) \) is big as \( N^2 \), and this is far from the case with Weintraub's choices given above, search \( S \) costs \( \Omega(\sqrt{N}/\log N) \) times as many operations as does sift\( (S, e, n, N) \), because

\[
\log \Pi(n) = \sum_{p \leq n} \log p \sim n
\]  

(2)

—[10, Theorems 413 and 434].

In view of these facts, we decide to concentrate on speeding up the statement search \( S \). We expect that any gain in speed will 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 AP that are unmarked (and hence prime):

\[
S^{f, h} = \{S_{h-k \cdot f}\}_{k \geq 0}^{dt} = \{y_k\}_{k \geq 0}.
\]

We see that Theorem 1 provides pertinent information. For let \( p \) be a prime such that \( p > n \) and \( p \not\equiv f \). Then, by Theorem 1, the congruence

\[
(f \cdot \Pi(n)) \cdot x \equiv -e - h \cdot \Pi(n) \pmod{p}
\]

has just one solution \( \pmod{p} \). This implies

\[
(\exists! m : 0 \leq m < p \quad \text{and} \quad (\forall i : 0 \leq i : p \mid y_{m+i \cdot p})).
\]

(3)

Now let \( p_i(f), i > 0 \), be the \( i \)th smallest prime \( p \) such that \( p > n \) and \( p \not\equiv f \), and let \( m_i(f) \) be the \( m \) asserted by (3) to exist for \( p = p_i \). It is clear that the search for \( n \) or more successive unmarked terms of \( S^{f, h} \) need only take place in the intervals between successive terms marked 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 sub-interval \( I_c \) of size \( 2n - p_1 + 1 \) such that if \( I \) contains a progression of \( n \) primes then every term in \( I_c \) is unmarked. 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.

**Theorem 3.**

\[
(\forall f : 1 \leq f \leq N \div (n - 1) : 2n > p_1(f)) \iff N < (n - 1) \cdot \prod_{n < p < 2n} p.
\]
Proof. 

\[ N < (n-1) \cdot \prod_{n < p < 2n} p \iff N \text{ div } (n-1) < \prod_{n < p < 2n} p \]

iff \( \left( \forall f: 1 \leq f \leq N \text{ div } (n-1): f < \prod_{n < p < 2n} p \right) \)

iff \( \left( \forall f: 1 \leq f < N \text{ div } (n-1): 2n > p_1(f) \right) \). \[\Box\]

The condition on the right-hand side of Theorem 3 is hardly of any concern in a practical context; e.g., for \( n = 17 \) it is \( N < 16 \cdot 19 \cdot 23 \cdot 29 \cdot 31 = 6285808 \). Henceforth we require the condition to hold.

Before writing the search, we must address the possibility that the first and last intervals are exceptional. They can be exceptional in two ways. The first, already mentioned, is that they might be truncated 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 \( \{y_k\}_{k=0}^{n-1} \), because \( p_1 \mid y_m \), is consistent with \( p_1 = y_m \). Given our desire for extreme efficiency, this presents a problem. It is prudent to investigate more deeply, and we obtain

**Theorem 4.** If \( n > 2 \) and \( p_1 = y_m \), then \( m_1 = 0 \).

**Proof.** Suppose \( n > 2 \) and \( p_1 = y_m \), but that \( m_1 \neq 0 \). We have

\[ p_1 = e + (h + m_1 \cdot f) \cdot \Pi(n) \geq 1 + f \cdot \Pi(n) \]

Let \( p \) be the greatest prime \( < p_1 \), so that \( p \equiv n \) or \( p \mid f \). Now \( p_1 = \Pi(p) - 1 \) since \( n > 2 \) implies \( p_1 > 3 \) implies \( p > 2 \). Thence \( p > n \) and \( p \mid f \). This means that \( p_1 \geq 1 + p \cdot \Pi(n) > 2p \) since \( n > 2 \). But this contradicts Bertrand's Postulate [10, Theorem 418], that for each prime \( p \) there is a bigger prime \( < 2p \). So \( m_1 = 0 \). \[\Box\]

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 follows easily from 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

**Corollary:** If \( n > 4 \) and \( N < (n-1) \cdot \prod_{n < p < 2n} p \) and \( p_1 = y_m \), then \( p_1 = e \).

We now know that special consideration is only necessary when \( h = 0 \) (and \( p_1 = e \)), provided only that \( n > 4 \) and \( N < (n-1) \cdot \prod_{n < p < 2n} p \). Our theorems can easily be seen to guarantee the correctness of
Algorithm 2.
\[
\begin{align*}
\{ & n > 4 \quad \text{and} \quad n \leq N < (n-1) \cdot \prod_{p < 2n} p \\
\}\end{align*}
\]

[As for Algorithm 1 but with search \( S \) replaced by search\(_2\) \( S \)]

The refinement of search\(_2\) \( S \) uses \( m_1 \), which is the size of the first (possibly empty) interval. From (3), we have
\[
0 \leq m_1 < p_1 \quad \text{and} \quad p_1 \mid e + (h + m \cdot f) \cdot \Pi(n).
\]

We use the notation \( S[i] \) to mean \( S_i \) is unmarked; this suggests the obvious implementation: a Boolean array.

search\(_2\) \( S \):

- add sentinels:
  - \( \text{forall } i: 1 \leq i \leq N \div (n-1) \text{ do mark } S_{-i}; \text{mark } S_{N+i}; \text{od; } \)
  - \( \text{forall } f: 1 \leq f \leq N \div (n-1) \text{ do } \)
    - set \( p_1 = \min\{p \mid p \text{ is prime and } p > n \text{ and } p \not\equiv f \}; \)
    - \( \text{forall } h: 0 \leq h \leq \min\{f-1, N-(n-1) \cdot f\} \text{ do } \)
      - set \( m_1 = \min\{m \mid m \geq 0 \text{ and } p_1 \mid e + (h + m \cdot f) \cdot \Pi(n)\}; \)
      - \( \text{if } m_1 \geq n \rightarrow \text{lastc} := h + (m_1 + n - p_1) \cdot f \)
      - \( \Box m_1 < n \rightarrow \text{lastc} := h + (m_1 + n) \cdot f \)
    - \( \)fi;
  - {\text{invariant: with the possible exception of a PAP of length } n \text{ starting at } y_0 \text{ when } h = 0, \text{ the PAPs of length } \geq n \text{ from } S_{f,h} \text{ have been accepted except for those containing a term } \geq S_{\text{lastc}} \text{ and } \{S_{\text{lastc}-(2n-p_1)\cdot f+k\cdot f}\}_{k=0}^{2n-p_1} \text{ is a critical region}}\)
  - \( \text{do lastc} \leq N \rightarrow \)
    - check interval:
      - firstc, \( i := \text{lastc}-(2n-p_1) \cdot f, \text{lastc} \);
      - \( S[i] \rightarrow i := i - f \text{ od; } \)
      - \( \text{if } i \geq \text{firstc} \rightarrow \text{skip } \{\text{nonprime in critical region}\} \)
      - \( \Box i < \text{firstc} \rightarrow \text{firsti}, i := i + f, \text{lastc} + f; \)
        - \( S[i] \rightarrow i := i + f \text{ od; } \)
        - \( \text{check } ((i - \text{firsti}) \div f, i, f)\{\text{see search } S\} \)
    - fi;
  - get next interval:
    - \( \text{lastc} := \text{lastc} + p_1 \cdot f \)
    - \( \text{od; } \)
  - \( \text{if } p_1 = e \rightarrow \text{check the AP } \{S_{k,f}\}_{k=0}^{n-1}: \)
    - \( i := f; \)
    - \( S[i] \rightarrow i := i + f \text{ od; } \)
    - \( \text{check } (i \div f, i, f) \)
    - \( \Box p_1 \neq e \rightarrow \text{skip } \)
  - fi
- \( \text{od } \)
The search of \( S^{f,h} \) 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. To quantify this observation, note that by the strong form of Dirichlet’s theorem (see [13]) the number \( \#p(S) \) of primes in \( S \) satisfies

\[
\#p(S) \sim \frac{1}{\phi(\Pi(n))} \frac{(N + 1) \cdot \Pi(n)}{\log ((N + 1) \cdot \Pi(n))}
\]

\[
\sim e^\gamma \cdot N \cdot \log n / (\log N + n)
\]

by (2) and the fact that

\[
\phi(\Pi(n)) = \Pi(n) \cdot \prod_{p \leq n} (1 - p^{-1}) \sim e^\gamma \cdot \Pi(n) / \log n
\]

—[10, Theorem 429]. Since the multiples of \( p_1 \) are avoided in the search, the probability \( p(f) \) that an examined member of \( S^{f,h} \) is prime is

\[
p \sim (1 - p_1^{-1})^{-1} \cdot e^\gamma \cdot \log n / (\log N + n).
\]

Hence for sufficiently large \( n \) and \( N \) the average number of looks at \( S \) per critical region of \( S^{f,h} \)

\[
= (1 - p) \cdot \sum_{j=1}^{\infty} j \cdot p^{-j} = (1 - p)^{-1},
\]

provided critical regions are sufficiently long, which will almost always be the case. So the cost of searching has been reduced to \( \Theta(N^2/n^2) \) additions. Note that the overhead involved in running over all values of \( f \) and \( h \) is also of this order.

Some of the searching can be avoided as follows. Let

\[ P_f = \{ p \mid p \text{ is prime and } n < p \text{ and } p \mid f \}. \]

Then if

\[ (\exists p : p \in P_f \text{ and } p \mid e + h \cdot \Pi(n)) \]

then all members of \( S^{f,h} \) are composite (except possibly the first), allowing that value of \( h \) to 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]. This observation can be made precise: the result of sifting \( S \) can be regarded as a string \( B \) of length \( N + 1 \) over the binary alphabet, with the \( k + 1 \)th symbol being 1 if and only if \( S_k \) was marked. Then the search for PAPs of length at least \( n \) reduces to finding all occurrences in \( B \) of the substring \((0 \Delta^{l-1})^{n-1}0\), where \( \Delta \) is a ‘don’t care’-symbol, for all \( f \in 1..N \text{ div } (n - 1) \). We leave the design of this general search, called \( \text{search}_{2B} \), to the reader. It stands in contrast to \( \text{search}_2 \) which exploits specific facts about the distribution of PAPs.
6. The approach pursued

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 composite multiple of $p_2$, and their intervals can therefore be ruled out of consideration; and of the remaining intervals, many will contain a composite multiple of $p_3$ in their critical region; 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 $\frac{18}{23} \cdot \frac{23}{29} \cdot \frac{29}{100} = 87.5\%$ of the values of $f$ have $(p_1, p_2, p_3) = (19, 23, 29)$. Further, the proportion $\sigma(f)$ of potentially good critical regions (i.e., those that do not contain a multiple of $p_2, \ldots, p_r$) is

$$\sigma = \prod_{i=2}^{r} \left(1 - \frac{2n - p_1 + 1}{p_i}\right) = \frac{7}{23} \cdot \frac{13}{29} = \frac{1}{7.3}. \quad (7)$$

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

$$(p_1, p_2, p_3) \in \{(19, 23, 31), (19, 29, 31), (23, 29, 31)\},$$

over 99\% of the values of $f$ are catered for. (The respective proportions for these 3 classes = $1/6.8$, $1/4.6$ and $1/2.8$.) 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 $(p_1, p_2, \ldots, p_r)$ that is considered, the var $\text{nextpgi}$:

$$\text{array } [0..-1 + \prod_{i=2}^{r} p_i] \text{ of } 1..-1 + \prod_{i=2}^{r} p_i$$

such that

$$(\forall j: 0 \leq j < \prod_{i=2}^{r} p_i): \text{the first interval between multiples of } p_1 \text{ that occurs after the } \text{jth interval} \ j \cdot p_1, (j + 1) \cdot p_1 \text{ and has no multiple of } p_2, \ldots, p_r \text{ in its critical region is the } (j + \text{nextpgi}[j])\text{th interval}. \quad (8)$$

Then if the current interval is the $\text{rth}$ 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:

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

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 \( S_{t, h} \). If \( m_1 \geq n \), 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 \( 2 \leq i \leq r \), where from (3) we have

\[
0 \leq m_i < p_i \text{ and } p_i \mid e + (h + m_i \cdot f) \cdot \Pi(n), \quad 1 \leq i \leq r,
\]

and the \( d_i \) are defined by

\[
d_i = (m_i - m_1) \mod p_i, \quad 2 \leq i \leq r.
\]

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 (9) and (10), we define \( d_i \) for the \( j \)th interval \( j \cdot p_1 \cdot (j + 1) \cdot p_1 \) 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

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

We now present a procedure that establishes properties (8) and (11) 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

\[
(\exists f: 1 \leq f \leq N \cdot \text{div} (n - 1): (\forall i: 1 \leq i \leq r: p[i] = p_1 (f))).
\]

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

\[
(\forall i: 2 \leq i \leq r: 0 \leq d[i] < p_i \text{ and } p_i | t \cdot p_1 + d[i]).
\]

procedure setpgi\( (N, n, r, p, dp, nextpgi)\):

\[
g := 0;
\]

establish (12):

\[
t := 1;
\]

\[
i := 2; \text{ do } i \neq r + 1 \rightarrow d[i], i := p[i] - p[1], i + 1 \text{ od};
\]

\{invariant: the nearest potentially good interval before the \( r \)th is the \( g \)th and (12) and (8) but with \( t \) as the upper bound for \( j \) and \( dp \) satisfies (11) when the range of \( dp \) is restricted to 0..t - 2\}

\[
do g \neq \prod_{i=2}^{r} p[i] \rightarrow
\]

\[
i := 2;
\]
{invariant: \(\exists j: 2 \leq j < i: d[j] < p[1] - n \text{ or } d[j] > n\)}

do\ i \neq r + 1 \text{ cand } (d[i] < p[1] - n \text{ or } d[i] > n) \rightarrow i = i + 1 \text{ od;}

if\ i = r + 1 \rightarrow \{\text{the rth interval is potentially good}\}

\[i := g;\]

do\ i \neq t \rightarrow \text{nextpgi}[i] := t - i; i := i + 1 \text{ od;}

\[g := t;\]

\[\Box i \neq r + 1 \rightarrow \text{skip}\]

fi;

\[dp[d[2], \ldots, d[r]] := t - 1;\]

\[t := t + 1;\]

re-establish (12):

\[i := 2;\]

do\ i \neq r + 1 \rightarrow d[i], i := (d[i] - p[1]) \mod p[i], i + 1 \text{ od od}\]

Before presenting the new algorithm, we must address the possibility that a PAP of length \(\geq n\) 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 \(i, 2 \leq i \leq r\). We would then have

\[p_i = e + (h + m_i \cdot f) \cdot \Pi(n).\]

Under the preconditions of Algorithm 2, the proof of the corollary of Theorem 4 shows that \(p_1 < \Pi(n)\). If we further require \(p_r < \Pi(n)\), this can occur only when \(e = p_1\) 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 effectively no restriction at all.

Algorithm 3.

\[
\left\{ n > 4 \text{ and } n \leq N < (n - 1) \cdot \prod_{n < p < 2n} p \right\}
\]

[As for Algorithm 2 but with search\(_2\) \(S\) replaced by search\(_3\) \(S\)]

search\(_3\) \(S\):

add sentinels; \{see search\(_2\)\}

forall\( f: 1 \leq f \leq N \div (n - 1) \) do

\[r := r(f);\]

forall\( i: 1 \leq i \leq r \) do

\[\text{set } p_i = \min\{p | p \text{ is prime and } p > n \text{ and } p \not\equiv f\} \text{ od;}
\]

\[P := \prod_{i = 2}^r p_i;\]

if\( p_r < \Pi(n) \rightarrow \text{setpgi}(N, n, r, p, dp, \text{nextpgi}) \) fi;
for all $h : 0 \leq h \leq \min \{f - 1, N - (n - 1) \cdot f \}$ do
  forall $i : 1 \leq i \leq r$ do
    set $m_i = \min \{m \mid m \geq 0 \text{ and } p_i \mid e + (h + m \cdot f) \cdot \Pi(n) \}$ od;
  forall $i : 2 \leq i \leq r$ do set $d_i = (m_i - m_1) \mod p_i$ od;
  $t := dp[d_2, \ldots, d_r]$;
  if $m_1 \geq n$ →
    if $t \neq 0$ → lastc, $t := h + (m_1 + n + (\text{nextpgi}[t - 1] - 2) \cdot p_1) \cdot f$, $(t - 1 + \text{nextpgi}[t - 1]) \mod P$
    $t = 0$ → lastc := $h + (m_1 + n - p_1) \cdot f$
  fi
  if $m_1 < n$ →
    lastc, $t := h + (m_1 + n + (\text{nextpgi}[t] - 1) \cdot p_1) \cdot f$, $(t + \text{nextpgi}[t]) \mod P$
  fi;
  do lastc ≤ $N$ →
    check interval; {see search2}
    get next interval:
      lastc, $t := \text{lastc + nextpgi}[t] \cdot p_1 \cdot f$, $(t + \text{nextpgi}[t]) \mod P$
  od
od;
if $e \in \{p_i \mid 1 \leq i \leq r\}$ → check the AP $\{S_k \cdot f\}_{k=1}^{n-1}$ {see search3}
if $e \notin \{p_i \mid 1 \leq i \leq r\}$ → skip
fi
od

Rather a lot has changed since the complexity analysis of Section 4. It is time
to take stock. Let us fix $n = 17$. The argument at the start of this section suggests
that it is reasonable to precompute $dp$ and $\text{nextpgi}$ 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 expect on
the basis of (6) and (7) that $\text{search}_3$ will be two orders of magnitude faster than
the search in Algorithm 1 (for values of $N \geq 10^4$ say), because the number of
lookups of $S[i]$ is reduced by that much. This appears to be more than enough to
outweigh the extra work involved in the statement "get next interval" of $\text{search}_3$.

But the above argument is flawed because it does not take into account the
overhead involved in iterating over $f$ and $h$. This was already $\Theta(N^2/n^2)$ operations
for $\text{search}_2$, and increases to $\Theta(r \cdot N^2/n^2)$ operations for $\text{search}_3$. This overwhelms
the time actually spent in examining $S$; $\text{search}_3$ therefore has the same order of
complexity as $\text{search}_2$! The trouble is that as $f$ approaches its maximum value the
number of intervals between multiples of $p_1$ approaches 1, and it becomes increas-
ingly 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 one feels that there should be a way to exploit the ideas in $\text{search}_3$, and
there is. 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 \( f_{\text{max}} \) of \( f \) fixed. The effect is that although more time is spent sifting, the overhead in searching is unchanged, allowing the speed-up in \( \text{search}_3 \) to take 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 choose \( f_{\text{max}} \) so that the minimum number of intervals between multiples of \( p_1 \), which \( \approx N/f_{\text{max}} \), is roughly equal to the inverse of the expected proportion of potentially good intervals.

In the sequel ‘Algorithm 3’ refers to Algorithm 3 as modified to include the extra parameter \( f_{\text{max}} \).

### 7. Efficient implementation

Algorithm 3 is sufficiently complex to present an interesting challenge to someone seeking a maximally efficient implementation. Let us try to meet this challenge.

Since \( N \) will likely be quite large, a space-efficient implementation of \( S \) is a necessity. Recalling that sentinels are needed at both ends of \( S \), we use

\[
\text{var } S : \text{array}[0..(N + 2 \cdot f_{\text{max}}) \div ss] \text{ of set of } 0..ss - 1,
\]

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

\[
(i + f_{\text{max}}) \mod ss \in S[(i + f_{\text{max}}) \div ss].
\]

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 \( \text{search}_3 \), we try to make the body of that loop as efficient as possible. Consider first the computation of the \( m_i \), for \( 1 \leq i \leq r \). From the defining relation (9) we have

\[
e + (h + m_i \cdot f) \cdot \Pi(n) \equiv 0 \pmod{p_i}.
\]

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

\[
m_i = -(h + e \cdot \text{inverse}(\Pi(n), p_i)) \cdot \text{inverse}(f, p_i) \mod p_i.
\]

(13)

Since the set of all \( p_i \) used in \( \text{search}_3 \) can be determined in advance, and is quite small, we precompute a table

\[
\text{inverse}_{p_i}[x] = \text{inverse}(x, p_i), \quad x \in 1..p_i - 1
\]

and a value

\[
\text{invPIn}_{p_i} = \text{inverse}(\Pi(n), p_i)
\]

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

\[
m_i := -(h + e \cdot \text{invPIn}[i]) \cdot \text{invfP}[i] \mod p_i
\]

(14)
where
\[
\text{invfp}[i] = \text{inverse}_{p_i}[(f \mod p_i)]
\]
and
\[
\text{invPIn}[i] = \text{invPIn}_{p_i}
\]
are set in the loop over the \( f \)-values. The \( d_i \) can then be directly computed from definition (10).

The above method can be speeded-up 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 (13)
\[
m_i = (m'_i - \text{inverse}(f, p_i)) \mod p_i. \tag{15}
\]
It follows that we need only use (14) to compute \( m_i \) for \( h = 0 \) (or \(-1\)), and then adapt (15) to update \( m_i \) after each increment of \( h \). But the values actually needed in \text{search3} are just \( m_1 \) and \( d_i, 2 \leq i \leq r \). By reapplying the above technique to the defining equations (10) and (15), we are finally led to the code given below.

\[
\begin{align*}
  &m_1 := m_1 - \text{invfp}[1]; \\
  &\text{if } m_1 \geq 0 \rightarrow \text{forall } i : 2 \leq i \leq r \text{ do} \\
  &\quad d_i := d_i - \text{dinc} \, 1[i]; \\
  &\quad \text{if } d_i < 0 \rightarrow d_i := d_i + p_i \land d_i \geq 0 \rightarrow \text{skip } \text{fi} \text{ od} \\
  &\text{if } m_1 < 0 \rightarrow \text{forall } i : 2 \leq i \leq r \text{ do} \\
  &\quad d_i := d_i - \text{dinc} \, 2[i]; \\
  &\quad \text{if } d_i < 0 \rightarrow d_i := d_i + p_i \land d_i \geq 0 \rightarrow \text{skip } \text{fi} \text{ od}; \\
  &m_1 := m_1 + p_1
\end{align*}
\]

where arrays \text{dinc} \, 1 and \text{dinc} \, 2 are set in the \( f \)-loop so that
\[
\begin{align*}
\text{dinc} \, 1[i] &= (\text{finfp}[i] - \text{finfp}[1]) \mod p_i, \quad i \in 2..r, \\
\text{dinc} \, 2[i] &= (\text{dinc} \, 1[i] + p_1) \mod p, \quad i \in 2..r.
\end{align*}
\]
The \( h \)-loop can be further streamlined by delaying the update of \( t \) until it is needed.

Since a significant amount of time may now be spent sifting, it is worthwhile to pay equally careful attention to procedure \text{sift}. We do not expect to find a substantial algorithmic improvement of \text{sift} (unlike the case for the 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 as required.

There is another matter to be addressed. It is that two of the calculations may lead to arithmetic overflow: the product \( e \cdot \text{inverse}(\Pi(n), p) \) and the term \( S_k \). The
Searching for primes in arithmetic progression

former is dealt with by the following refinement:

set $k$ such that (1):

$$k := \text{prodmod}(-e, \text{inverse}(\Pi(n), p), p)$$

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

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

$$= (p - e) \div \Pi(n) = k \text{ and } (p - e) \mod \Pi(n) = 0.$$  

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. A simpler alternative is available if $N \leq \Pi(n)$, because then $p \leq \Pi(n)$ 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

$$\text{if } e \text{ is prime } \rightarrow \text{unmark } S_0 \text{ else } e \text{ is not prime } \rightarrow \text{skip } \text{od}$$

Having thus optimized procedure $sift$, the only way to spend less time sifting is to actually do less sifting. But since PAPs of length at least 17 are extremely rare, so also must be quasi-PAPs of length at least 17, where a quasi-PAP is an AP of numbers which have only 'large' prime factors. So we decide to initially sift only up to a certain fraction $\lambda$, $0 < \lambda \leq 1$, of the maximum prime otherwise required, and to sift further if necessary (and only as much as necessary) when a quasi-PAP is 'accepted'. The value of $\lambda$ 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 exploiting standard optimization techniques. One obvious optimization is to set the sentinels of $S$ outside the loop over the $e$-values. Other optimizations include removing constant expressions from loops, 'rolling out' loops ($r$ can be made a constant function of $f$ for given $N$ by judicious choice of $r$-tuples $(p_1, \ldots, p_r)$, enabling the loops over $r$ in $\text{search}_3$ to be rolled out) and judicious manual optimization of compiled assembly code. It is only worthwhile to apply these low-level techniques to the innermost loops.

8. An alternative approach

The algorithms presented thus far were designed from the starting point of Weintraub's algorithm. Before committing ourselves to gigantic computations, it is prudent to step back, get a wider perspective, and see if there are other approaches that warrant consideration.

An obvious alternative is to have the search proceed by first determining the indices $k$ of the primes $S_k$ in $S$, and then simply examining all APs with first two
terms (known to be) prime. We use an array

\[
\text{var pos : array [1..N] of 0..N}
\]

to hold the indices of the primes in \(S\):

\[
(\forall i: 1 \leq i \leq \#p(S): S_{pos[i]} = \text{the ith prime in } S).
\]

(16)

**Algorithm 4.**

\{ \(n > 2 \text{ and } N \geq n\) \}

[As for algorithm 1 but with search \(S\) replaced by search\(_4\) \(S\)]

\(\text{search}_4 S:\)

establish (16);

add sentinels:

\[
\text{forall } i: 1 \leq i \leq N \text{ div } (n-1) \text{ do mark } S_{N+i} \od;
\]

\[
\text{forall } i: 1 \leq i \leq \#p(S) - n + 1 \text{ do}
\]

\[
j, f := i + 1, \text{pos}[i + 1] - \text{pos}[i];
\]

\[
\text{do } \text{pos}[i] + (n - 1) \cdot f \leq N \rightarrow
\]

check the \(AP\) \(\{S_{pos[i] + k \cdot f}\}_{k \geq 0};
\]

\[
k := pos[i] + 2 \cdot f;
\]

\[
\text{do } S[k] \rightarrow k := k + f \od;
\]

check \((k - \text{pos}[i]) \text{ div } f, k, f\); \{see search \(S\}\}

\[
j, f := j + 1, \text{pos}[j + 1] - \text{pos}[i]
\]

\od

For each \(i, 1 \leq i \leq \#p(S) - n + 1,\) and for each \(j > i\) such that \(\text{pos}[i] + (n - 1) \cdot (\text{pos}[j] - \text{pos}[i]) \leq N,\) the longest \(PAP\) with first two terms \(S_{pos[i]}, S_{pos[i]}\) is determined. So the cost is approximately

\[
\sum_{i=1}^{\#p(S) - n + 1} \frac{N - \text{pos}[i]}{n - 1} \cdot \frac{\#p(S)}{N} = \frac{\#p(S)^2}{2 \cdot (n - 1)}
\]

\[
= \Theta \left( \frac{N^2 \cdot (\log n^2)}{(n-1) \cdot (\log N + n)^2} \right)
\]

operations, using (4). We have assumed that the probability of the third term of an examined \(AP\) being prime is \(\approx \#p(S)/N,\) and that therefore the typical \(PAP\) examined has length two. If the average length of an examined \(PAP\) significantly exceeds two then search\(_4\) would be slowed by that factor. (Imagine if all the primes in \(S\) formed a single \(PAP\)! This assumption can be dispensed with by modifying
Searching for primes in arithmetic progression

search so that when checking the AP

$$S_{pos[i]+k.f}$$

it first 'looks backwards' at $$S_{pos[i]-f}$$ and only continues if this number is not prime. (Sentinels need to be added at the low end of $$S$$ for this modification.) Although this is a good policy for an unknown sequence $$S$$, it is a dubious one in our situation.

9. Comparing algorithms and choosing parameters

We now address the issue of choosing algorithms and values for their parameters so as to obtain optimal expected returns from our computational experiments. Our choices will of course depend on the experiment. The main experiments we have in mind are the following:

(i) Find the PAP(s) of length $$m$$ with minimum last term. Grosswald and Hagis [7] report that no such PAP is known for $$m > 10$$.

(ii) Find a PAP of record-breaking length (i.e., of length at least 18).

(iii) Test the asymptotic formula of Hardy and Littlewood for 'long' PAPS, by systematically finding large numbers of long PAPS.

In connection with problem (ii) above, we had noted that the first reported PAP of length 16 had a common difference of $$\Pi(23)$$, that all six PAPs of length 16 found by Weintraub [17] have a common difference divisible by $$\Pi(17)$$ and that Weintraub’s PAP of length 17 has a common difference divisible by $$\Pi(19)$$. These facts prompted us to generalize our algorithms so that although $$n(n)$$ is the common difference of $$S$$, the search is made for PAPS of length $$m < n$$. (It is a straightforward exercise to generalize our results accordingly.)

When comparing alternative algorithms for a problem it is highly desirable to have good estimates for their running times. So we proceed to rework our earlier 'rough' estimates, and first consider Algorithm 2. After easy modifications to account for the added parameters $$\lambda$$ and $$m$$, and writing $$t$$ for $$N \cdot \Pi(n)$$, the cost $$T_2$$ of processing each of the $$\phi(\Pi(n))$$ values of $$e$$ is seen to be

$$T_2 = \frac{c_1 \cdot \lambda \sqrt{t}}{\log(\lambda \sqrt{t})} + c_2 \cdot N \cdot \log \left( \frac{\log(\lambda \sqrt{t})}{\log n} \right) + \frac{c_3 \cdot N^2}{(1 - \#p(S)/N) \cdot (m - 1) \cdot \rho_1} + \frac{c_4 \cdot N^2}{2 \cdot (m - 1)^2}$$

(machine language) operations. Here we have made explicit the constants that were implicit in our earlier estimates. The constant $$c_i$$ is the number of operations needed for each iteration of the (part of the) loop involved in the corresponding task; e.g., $$c_1$$ is the number of operations required in each iteration of the forall-loop of procedure sift, excluding those for the inner loop for marking elements of $$S$$ as these are covered by $$c_2$$. We have assumed that $$\lambda$$ is sufficiently large that the extra
time needed to check quasi-PAPs—by further sifting or by primality tests—is negligible. The denominator in the first summand is $c_1 \cdot \pi(\lambda \sqrt{t})$ because the values $\pi(I(n, p))$ are precomputed. $1/\bar{p}_1$ is the average (over all possible values of $f$) of $1/p_1$; it approaches $1/n$ from below as $n \to \infty$. The correction factor $(1 - \#p(S)/N)^{-1}$ comes from (6), but here $\#p(S)$ denotes the number of unmarked elements of $S$; for $\lambda < 1$ not all of these are necessarily prime. The last summand is for the overhead in iterating over $f$ and $h$.

Similar analyses can be done for Algorithm 2B—i.e., Algorithm 2 with the Boyer/Moore variation, which in view of (17) may as well be thrown into the pot—and Algorithms 3 and 4. (It is already apparent that Algorithm 1 is uniformly inferior to Algorithm 2.) The respective costs are as follows:

$$T_2 = \frac{c_1 \cdot \lambda \sqrt{t}}{\log(\lambda \sqrt{t})} + c_2 \cdot N \cdot \log \left( \frac{\log(\lambda \sqrt{t})}{\log n} \right)$$

$$+ \frac{c_5 \cdot N^2}{(1 - \#p(S)/N) \cdot m - \#p(S)/N \cdot (m - 1)} + \frac{c_6 \cdot N^2}{2 \cdot (m - 1)^2},$$

(18)

$$T_3 = \frac{c_1 \cdot \lambda \sqrt{t}}{\log(\lambda \sqrt{t})} + c_2 \cdot N \cdot \log \left( \frac{\log(\lambda \sqrt{t})}{\log n} \right)$$

$$+ \frac{c_7 \cdot N^2 \cdot \bar{r}}{(1 - \#p(S)/N) \cdot (m - 1) \cdot d \cdot \bar{p}_1} + \frac{c_8 \cdot \bar{r} \cdot N^2}{2 \cdot (m - 1)^2 \cdot d^2},$$

(19)

where $\bar{r}$ and $\bar{\sigma}$ are the averages (over all values of $f$) of $r$ and the speed-up given by (7) respectively, and $f_{\max} = d \cdot N/(m - 1), d \geq 1$;

$$T_4 = \frac{c_1 \cdot \lambda \sqrt{t}}{\log(\lambda \sqrt{t})} + c_2 \cdot N \cdot \log \left( \frac{\log(\lambda \sqrt{t})}{\log n} \right)$$

$$+ \frac{c_9 \cdot \#p(S)^2}{(1 - \#p(S)/N) \cdot 2 \cdot (m - 1)}.$$  

(20)

Algorithms 2, 2B, 3 with $d = 1$ and 4 find exactly the same PAPs with the same settings for $m$, $n$ and $N$, so we can extract some information at this point. Firstly, the choice between Algorithms 2 and 2B depends solely on the constants $c_3$, $c_4$, $c_5$ and $c_6$. Comparing the code for the two searches reveals that $c_3 = c_5$, so that the search sans overhead of Algorithm 2 is faster (and becomes more so as $n/m$ increases), and that $c_4 > c_6$, so that the search of algorithm 2B has a smaller overhead. Secondly, Algorithm 3 with $d = 1$ will be slower than Algorithm 2 because of the extra overhead of the last summand of (19). Thirdly, consider Algorithm 4. When $\lambda = 1$, the asymptotic formula (4) for $\#p(S)$ reveals that $search_4$ is much faster than the other searches for sufficiently large $m$ and/or $n$. However as $n \to \infty$ while $N$ is held fixed (as it must be because of storage limitations) the first summand of $T_4$ becomes dominant, and will eventually exceed $T_3$ (with $\lambda$ small), the crossover point being roughly $I\Pi(n) = N^3$. Also the amount of external storage for the sifting primes grows alarmingly, and ever longer-precision arithmetic becomes necessary in procedure $sift$, since inverses modulo larger primes are used. If $\lambda \sqrt{t}$ is kept
bounded by adjusting $\lambda$, the cost of the search eventually dominates $T_4$, but it only varies inversely with $m$, so that $T_4$ will eventually exceed $T_2$.

To analyse Algorithm 3 for $d > 1$, and to decide on settings for the parameters of a chosen algorithm for a particular task, further information is needed about the distribution of PAPs. Fortunately, a heuristic asymptotic formula for the number $N_{m,n}(x)$ of PAPs of length $m$ with all terms $\leq x$ and common difference a multiple of $\Pi(n)$, $n \geq m$, can be obtained from the formula for the case $n = m$ in [6]. We get

$$N_{m,n}(x) \sim \prod_{m \leq p < n} \frac{1}{p + 1 - m} \cdot \frac{c_m \cdot x^2}{\log^m x}$$

(21)

where

$$c_m = \frac{1}{2 \cdot (m-1)} \cdot \prod_{p \geq m} \left( \frac{1}{p} \cdot \frac{(p-1)^{m-1}}{p^{m-1} - (p-1)^{m-1}} \right).$$

Furthermore, when common differences are bounded by $\Pi(n) \cdot d^{-1} \cdot N/(m-1)$, $d \geq 1$, as in Algorithm 3, the predicted number of PAPs of length $m$ is $d^{-1} \cdot N_{m,n}(x)$. Although not explicitly given in [6], these estimates follow easily from the arguments therein.

Now let us consider each of the three problem classes, and determine the optimal algorithm. We start with (i), for which two decisions are immediate: we must put $n = m$, and we rule out Algorithm 3. The smallest known PAPs of length $m$, $11 \leq m \leq 17$, are given in [7]. The expected number of PAPs of length $m$ up to the limit $(N + 1) \cdot \Pi(m)$ can be found from (21). For given $m$, we determine $N$ so that three PAPs are expected. (Better safe than sorry, and three is a reasonable choice given the data in [7].) If the smallest known PAP allows us to reduce $N$, we do so. For $m = 11, 12, 13$ (cases when $N$ was reduced), $N$ is so small that the sifting time is most important, and any of the three algorithms will do. For $14 < m \leq 19$, $N$ is large enough for the searching time to dominate, so the algorithms can be roughly compared by comparing the searching terms in (18) and (20). The searching term in (20) is smallest when $A = 1$, in which case the ratio $\#P(S)/N$ can be accurately estimated from the first part of (4). But since procedure sift has to be coded anyway, we can get a ‘perfect’ estimate by sifting $S$ for a typical value of $e$, say $e = \Pi(n)/2 - 2$. The results do not imply a decisive vote either for or against Algorithm 4 ($\#P(S)/N$ steadily decreases from 0.336 for $m = 14$ down to 0.226 for $m = 19$). So Algorithms 2B (which is simpler than Algorithm 2) and 4 were coded, near optimal settings for $\lambda$ were determined by experimentation, and the resulting run times on our typical $e$ compared. In all cases Algorithm 2B was faster. With $\lambda = 1$, search2B was roughly twice as fast as search4 when $m = 14$. As $m$ increased, the latter steadily gained ground, until at $m = 19$ it was slightly faster.

1 The r.h.s. of (21) is actually the dominant term of an infinite series for $N_{m,n}(x)$. Grosswald [6] gives a computable expression for the next term, which contributes significantly for the numbers under consideration; the remaining terms contribute little more, so we use the first two terms in our calculations.
However, the sift time grew appreciably, and the far greater optimal reduction in $\lambda$ for Algorithm 2B outweighed the advantage of search. Because $n = m$ we might expect Algorithm 2 to be slower than Algorithm 2B; this is indeed the case, but the difference is only a few percent. Note that the three programs were not optimized, but we figured that optimization would speed up each search by roughly the same factor. Of course it is preferable—in fact, definitive (for a given computer)—to carry out the comparisons with optimized programs.

Now consider problem (ii). We want to find a PAP of length $m$, for $m = 18, 19$. We postpone consideration of Algorithm 3. As with problem (i), we can determine from (21) the value of $N$ needed to yield three expected PAPs of length $m$ in a complete search. But since $n$ is now free to vary, we compute $N$ not only for $n = m$ but also for $n$ ranging over the (first few) primes $>m$. As a relative measure of cost for Algorithm 2B, we compute $N^2 \cdot \phi(\Pi(n))/\phi(\Pi(m))$. For both values of $m$, $n = 23$ gives lowest cost measure: the improvement over the case $n = m$ is by a factor of about 10 for $m = 18$ and 2 for $m = 19$.

These are important gains. The reason is as follows. Let $N_0$ denote the value of $N$ needed to give three (say) expected PAPs for $n = m$. Let $q_k$ denote the $k$th prime $>m$, and $N_k$ denote the value of $N$ needed for $n = q_k$. By setting

$$N_{m,q_k}(N_k \cdot \Pi(q_k)) = N_{m,q_{k-1}}(N_{k-1} \cdot \Pi(q_{k-1}))$$

we find (using an argument like that for the derivation of (24) below) that

$$N_k = N_{k-1} \cdot q_k^{-1} \cdot \frac{-q_k + 1 - m \cdot (1 + \Delta)^{m/2}}{2 \log(N_{k-1} \cdot \Pi(q_{k-1})) - m}$$

(22)

where

$$\Delta = \frac{\log(q_k + 1 - m)}{2 \log(N_{k-1} \cdot \Pi(q_{k-1})) - m}.$$ 

Since the number of values of $e$ to examine increases by a factor of $q_k - 1$, the speed-up in the search for $n = q_k$ over that for $n = q_{k-1}$ is by a factor of approximately

$$\frac{q_k^2}{(q_k - 1) \cdot (q_k + 1 - m) \cdot (1 + \Delta)^m}.$$ 

(23)

Since $\log \Pi(m) \sim m$, and $(1 + x/m)^m = e^x$ for large $m$ and small $x$, we see that $(1 + \Delta)^m$ significantly increases the denominator of (23). In fact, provided $N_0$ is not too large, the speed-up for $k = 1$ is roughly equal to $q_1/(q_1 + 1 - m)^2$; but as $k$ increases, (23) soon falls below 1. This phenomenon nicely accounts for our earlier observation concerning the larger than necessary common differences of record-breaking PAPs, especially since much smaller values of $N$ are needed as $n$ increases.

Since $n$ now exceeds $m$, we might expect Algorithm 2 to outperform Algorithm 2B, and experimentation shows that the former is now a few percent faster. Since Algorithm 4 is ruled out by the results concerning problem (i), two questions remain with problem (ii): Should $N$ be increased? Is Algorithm 3 useful? The point of the first question is that it may be more effective to initiate a search that would produce
larger numbers of PAPs, and to stop the search as soon as a PAP is found. The answer is "No", because with \( m \) and \( n \) fixed, \( N_{m,n}((N + 1) \cdot \Pi(n)) \) grows slower than \( N^2 \), whereas \( T_2 \) grows as \( N^2 \). The answer to the second question follows from our discussion of problem (iii) below. It is also negative.

Lastly, we address problem (iii). In [7] it is noted that the asymptotic formula for \( N_{m,n}(x) \) can be quite inaccurate when only small values are predicted. Since the formula has only conjectural status, it is interesting to test its predictions against observed counts. But this is most meaningful when the counts are high. Hence our desire to find large numbers of long PAPs since the formula has thus far been tested for only very small values of \( m \).

As far as Algorithms 2, 2B and 4 are concerned, the best strategy for problem (iii) is clear from the preceding discussion: choose \( m \) so that the following primes \( q_k \) are close, then choose the number of PAPs desired, then use (21) to find the optimal values of \( n \) and \( N \). A good choice is to seek 100 PAPs of length \( m = 16 \) with Algorithm 2, whence the best choices of \( n \) and \( N \) turn out to be 23 and 1735 respectively. The speed-up with \( n = 23 \) as compared to \( n = 16 \) is by a factor exceeding 18!

Let us finally turn to Algorithm 3. The basic hope with this approach is that a small value of \( \bar{\sigma} \) will permit a choice of \( N \) and \( d \) that makes \( T_3 \) smaller than \( T_2 \) in a situation where Algorithms 2 and 3 are expected (because of (21)) to find equal numbers of PAPs. From (21) we see that the value of \( N \) for general \( d \) must be at least \( \sqrt{d} \) times that for \( d = 1 \). If the ratio were exactly \( \sqrt{d} \), then with \( d \) large \( T_3 \) would be dominated by the third summand of (19), permitting a speed-up by a factor of \( 1/\bar{\sigma} \). That is the most we can hope for.

Unfortunately, the payoff is much less, because of the log power in (21). To see this, suppose

\[
d^{-1} \cdot N_{m,n}(N' \cdot \Pi(n)) = N_{m,n}(N \cdot \Pi(n)).
\]

Then from (21) we have

\[
N^{-2} = d \cdot N^2 \cdot \log^m(N' \cdot \Pi(n))/\log^m(N \cdot \Pi(n))
= d \cdot N^2 \cdot (1 + \Delta)^m
\]

where

\[
\Delta = \log(N'/N)/\log(N \cdot \Pi(n))
= \frac{1}{2} \log(d \cdot (1 + \Delta)^m)/\log(N \cdot \Pi(n))
= \frac{1}{2} \log d + m \cdot \Delta)/\log(N \cdot \Pi(n))
\]

provided \( \Delta \ll 1 \). So

\[
\Delta \approx \frac{\log d}{2 \log(N \cdot \Pi(n)) - m}
\]
and
\[ N' = N \cdot \sqrt{d} \cdot (1 + \Delta)^{m/2}. \tag{24} \]

Now it is clear that for \( \bar{r} > 1 \) the speed-up of Algorithm 3 (from the case \( \bar{r} = 1 \)) will be by a factor rather less than \( 1/\bar{\sigma} \). For the latter is biggest when \( n = m \), but then \( \Delta \) will be almost \( \log d/m \) for \( N \) sufficiently small, whence \( N' \) will be almost \( d \cdot n \) and (19) shows that \( T_3 \) will be smallest with \( d \) almost 1!

The above argument is pessimistic in the sense that \( N \) is likely to be large when \( n = m \), but it nevertheless greatly dampens our enthusiasm for Algorithm 3. Let us consider applying Algorithm 3 to problem (iii). There are two ways we might do this. The first is to try to speed up the optimal version of Algorithm 2. But with \( m = 16 \) and \( n = 23 \), putting \( \bar{r} = 3 \) gives \( \bar{\sigma} = 1/1.29 \), which is not worth further consideration. The other approach is to try to gain a large speed-up with \( n = m \). This is best regarded as the limiting case of the strategy of reducing \( n \) from its optimal setting for Algorithm 2, with the hope of obtaining an overcompensating decrease in \( \bar{\sigma} \). With \( n = m = 16 \) and \( \bar{r} = 3 \), \( \bar{\sigma} = 1/20.8 \). But we know to expect a much smaller speed-up than \( 1/\bar{\sigma} \), and algorithm 2 with \( n = 23 \) is 18 times faster than with \( n = 16 \). Also Algorithm 2 needs \( N = 591 \, 000 \) to give 100 expected PAPs with \( m = n = 16 \), so the much larger \( N \) needed for Algorithm 3 will be impractical. Our best chance is thus with \( n = 19 \), when Algorithm 2 needs \( N = 8740 \). With \( \bar{r} = 3 \) we find \( \bar{\sigma} = 1/2.25 \). Since this must compensate for an increase in the cost measure for Algorithm 2 by a factor of \( \approx 1.15 \) (this is the reduction in the cost-measure in moving from \( n = 19 \) to \( n = 23 \)), and \( N \) is small, and \( n \) is still close to \( m \), it is doubtful that Algorithm 3 is worthwhile here.

To get a quantitative feel for the comparative performance of Algorithm 3 (without coding it), we estimate the crucial constants in (17) and (19). The following settings, although rough, are not unrealistic: \( c_3 = c_4 = c_7 = e_8 \). Now to choose \( N \) and \( d \) nearly optimally, under the constraints of problem (iii), we use (21) to minimize the cost-measure
\[ C_3 = N^2 \cdot \bar{\sigma} \cdot d^{-1} + \bar{r} \cdot N^2 \cdot d^{-2}/2. \]

The results for the two searches contemplated above are as follows. For the search for 100 PAPs with \( m = 16 \) and \( n = 19 \), with \( \bar{r} = 3 \) and \( \bar{\sigma} = 1/2.25 \), \( C_3 \) attains its minimum \( 1.20 \cdot 10^8 \) with \( N = 25 \, 000 \). The corresponding cost-measure for Algorithm 2, viz. \( 3N^2/2 \), is equal to \( 1.15 \cdot 10^8 \) as \( N = 8740 \). Thus if our estimates are accurate, Algorithm 3 will be slightly slower than Algorithm 2 in this situation. For the search for 100 PAPs with \( m = n = 16 \), with \( \bar{r} = 3 \) and \( \bar{\sigma} = 1/20.8 \), \( C_3 \) attains its minimum of \( 18 \cdot 10^{10} \) with \( N = 8500 \, 000 \) and \( d = 36 \). The cost-measure for Algorithm 2 (with \( N = 591 \, 000 \)) is \( 52 \cdot 10^{10} \), about 3 times larger. It is now apparent that Algorithm 3 will only be useful when many PAPs of length \( m \) with small common differences (i.e., with \( n = m \)) are desired, and that the improvement over Algorithm 2 is not as spectacular as we first hoped.
As a final point, (19) suggests that using the parameter $d$ may pay dividends in Algorithms 2 and 2B. Indeed, since $N$ will increase by a factor less than $d$, this variation would reduce the overhead of the searches (represented by the last summands in (17) and (18)). Unfortunately, however, (24) shows that $N$ increases by a factor significantly exceeding $\sqrt{d}$, so that the search times (represented by the third summands of (17) and (18)) will increase. Since Algorithm 2 has a significant overhead, we might nevertheless expect a small gain. Experimentation with the optimal parameter settings for problem (ii) reveals a maximal speed-up of ten to twenty percent with $d$ near 1.25.

10. Some computational results

Our first experiment was performed with programs that were created without the benefit of the analysis of Section 9, as we were unaware of (21) for general $n$ and $d$. We implemented Algorithm 3 in the programming language Pascal, with hand optimization of the assembly code for the inner loops. We chose $m = n = 17$, $N = 300,000$, $f_{\text{max}} = 1000$ (so that $d = 18.75$) and $\lambda = 0.1$. Each value of $e$ was processed in a little over two minutes on a VAX-11/780 running under Berkeley Unix. After processing about 60% of the $e$-values, the program had found 6 PAPs of length 17, including Weintraub's [18]. No longer PAPs were found. The predicted numbers of PAPs for the complete search are 5.8 of length 17 and 0.6 of length 18. Our hope to find a PAP of length 18 was clearly unrealistic, and our unrealised expectations led us to the analyses of Section 9.

Our next experiment was simultaneously one of type (i), (ii) and (iii). We put $m = 17$, $n = 19$ and $N = 30,000$. The Hardy–Littlewood formula predicted 1.9 PAPs of length 19, 13 of length 18 and 86 of length 17 in this very long search. Thus we gambled on finding the minimum PAP of length 19 (for (i)), and expected to find a PAP of length 18 (for (ii)) and sufficiently many PAPs of length 17 to test their number against the prediction of (21) (for (iii)). We used Algorithm 2B (there is probably very little difference in using Algorithm 2). Our program was again written in Pascal, with the inner loops manually optimized, and again $\lambda = 0.1$ proved optimal. Each value of $e$ was processed in about half a minute. After completing 38% of the search, the program had found 28 PAPs of length 17 and 2 PAPs of length 18 (making 32 of length 17 as against the predicted 32.6). The PAP of length 18 with smaller last term is

$$ (7,922,693 + 10,153 \cdot 9,699,690) + k \cdot (533 \cdot 9,699,690), \quad k = 0, 1, \ldots, 17. $$

11. Final remarks

We hope to have accomplished two things with this paper. The first is to have given a realistic and honest view of the programming process, one that is much
broader than usually taken in methodological studies. A programmer must often
do more than derive a hopefully beautiful solution to a specified problem (though
that is hard enough); in general she will design several solutions, may even modify
the specifications to permit more effective solutions, and must choose between the
solutions. This last task is best based on complexity analyses, which we believe
play an important and underappreciated role in programming. Since all program-
mers (as distinct from coders) must frequently choose between alternative
algorithms, and perform some kind of complexity analysis in so doing, it is apparent
that mathematical ability and knowledge (and not necessarily limited to discrete
mathematics, even for discrete problem domains) is an invaluable asset, even if the
programmer works exclusively in application areas that have thus far resisted
mathematical formalization.

Our other hope is to have presented a useful compendium of methods for the
computational number theorist. We learnt two main lessons in the (sometimes
painful) process reported herein. The first is that complex problems for which
maximally efficient solutions are desired can be successfully tackled with a straight-
forward and systematic approach. We believe that a disciplined programming
methodology based on invariants and correctness proofs is at least as important in
this domain as elsewhere. The other is that when choosing between algorithms,
and determining good settings for parameters, intuition is no substitute for precise
complexity analyses. Our original decision to write programs for finding PAPs was
based on the promise of great improvements in efficiency by using the methods of
Algorithm 3. We eventually obtained speed-ups (over Algorithm 1) of two orders
of magnitude (for problems (ii) and (iii)). Yet the simple idea behind Algorithm 2
contributed one order of magnitude, and the other was due not to Algorithm 3,
but rather to optimal choices of parameters.

Appendix

The function
\[ \text{prodmod}(a, b, p) = (a \cdot b) \mod p \]
can be derived from the following two facts:
\[ \text{prodmod}(2a', b, p) = \text{prodmod}(a', 2b \mod p, p), \]
\[ \text{prodmod}(2a' + 1, b, p) = (\text{prodmod}(2a', b, p) + b) \mod p. \]
A recursive formulation is immediate, but these facts also readily suggest an
invariant assertion for an iterative version.

function \text{prodmod}(A, B, p):
{\text{in: } p > 0}
{\text{returns } A \cdot B \mod p}
Searching for primes in arithmetic progression

\[ a, b, c := A \mod p, B \mod p, 0; \]
\[
\{\text{invariant: } (a \cdot b + c) = A \cdot B \mod p \text{ and } 0 \leq a, b, c < p \}
\]

\[ \text{do } a \neq 0 \text{ and even(a) } \rightarrow a, b := a \div 2, 2b \mod p \]
\[ \text{odd(a) } \rightarrow a, c := a - 1, (b + c) \mod p \]
\[ \text{od; } \]
\[ \text{return(c) } \]

Note that if \( p \leq \text{maxint} \div 2 + 1 \), where \text{maxint} is the greatest representable integer, then overflow cannot occur in \text{prodmod}. Two modifications can be made to increase efficiency. First, the multiplicative \text{mod} operations in the loop can be replaced by additive ones; the restriction on \( a, b \) and \( c \) in the invariant makes this possible. Second, the loop can be terminated as soon as it is possible to complete the calculation without the possibility of overflow occurring. To do this, replace the conjunct \( a \neq 0 \) of the first guard in the loop with \( a \geq \text{maxint} \div (p - 1) \), and replace the statement \text{return(c)} with \text{return}((a \cdot b + c) \mod p).

References