Merrily We Roll Along: Some Aspects of 

J. O. Shallit

Department of Mathematics
and
Department of Computing Services
University of California, Berkeley
Berkeley, CA 94720
(415) 642-5523

1. Abstract.

We present an efficient method for determining the number of invocations of \( ? \) given the value of \( EIRL \), and solving the inverse problem. A full implementation is given for the random-number generator in \( APL\360 \) and its descendants.

2. Introduction.

The \( APL \) function \( ? \) is a pseudo-random number generator. The numbers generated by \( ? \) depend on both the argument(s) to the function and the system variable \( DRL \), the random link.

Suppose you are running a program that uses random numbers (for example, a simulation of the length of queues at the post office, where customers enter at random times). You execute the program, but interrupt it before the simulation is complete. Now you'd like to know how far the simulation proceeded; since each time \( ? \) is used, a new random link is generated, it is possible to determine the number of invocations of \( ? \) by looking at \( DRL \).

Similarly, you might want to know what value should be assigned to \( DRL \) to get the effect of having executed \( ? \) a given number of times; for example, to run the third simulation in a sequence without having to rerun simulations one and two.

Recent number-theoretic results permit these questions to be answered in a reasonable length of time. We will solve two problems:

(a) Given \( k \), compute the value of \( DRL \) after \( ? \) has been executed \( k \) times.

(b) Given \( DRL \), compute the number of times that \( ? \) has been invoked. Since the random number generators discussed are periodic with period \( P \), we can answer this question only up to a multiple of \( P \).

In order to facilitate exposition, we will use both conventional mathematical notation and \( APL \) notation. Direct definition is used where the form of functions being discussed is appropriate. For a program to process direct definitions, see \[6\].

Following McDonnell \[13\], we will use the symbols \( \vee \) and \( \wedge \) to represent \( \text{gcd} \) and \( \text{lcm} \), respectively. Index origin 1 is assumed throughout.

No attempt will be made to rate the "quality" of the random number generators being discussed. It may be worthwhile to note, however, that the generator commonly in use may, in fact, be inadequate. See, for example, \[3\] or \[51\].

3. The Linear Conguential Method for Pseudo-Random Number Generation.

The algorithm for \( ? \) used in \( APL\360 \) and its descendants, including \( APLSV \), \( APL\CMS \), \( VS APL \), \( APL\replus \), and \( SHARP APL \), generates a new random link from the old one by

\[
DRL \rightarrow 2147483647 \times DRL + 1
\]  

(1)

See McDonnell \[14\]. Note that 2147483647 = \( 2^{31} - 1 \), a prime, and 16807 = \( 7^3 \). The default for \( DRL \) in a clear workspace is 16807. In the systems mentioned above, equation (1) is performed once for each use of \( ? \) on a
scalar; for arguments which require more than one random number to be generated, (1) is executed an appropriate number of times. In addition, (1) is executed twice if the right argument is larger than $2^{31}$. We call the execution of (1) an invocation of ?; hence ?1 2 3 4 counts as four invocations.

The APL360 method is a particular instance of a more general technique usually called the linear congruential method. In this technique, we start with an initial seed $X_0$, and generate new ones by

$$X_{n+1} = aX_n + c \pmod{M}. \quad (2)$$

Here $M$ is called the modulus, $a$ is called the multiplier, and $c$ is called the increment. See Knuth [8]. The notation $(\pmod{M})$ means that arithmetic is done modulo $M$; the reader whose elementary number theory is a little rusty should at this point read through Appendix I.

All of the APL systems that the author has seen use the linear congruential method to generate the values of ?RL. Table 1 gives a brief summary of the parameters for some commonly used systems.

We will now solve the first of our two problems for the general linear congruential scheme. Iteration of equation (2) gives

$$X_{n+k} = a^kX_n + c(1 + a + \cdots + a^{k-1}) \pmod{M}. \quad (3)$$

In order to answer the first of our two questions, we must be able to calculate the two quantities

$$a^kX_0 \pmod{M} \quad (4)$$

and

$$c(1 + a + a^2 + \cdots + a^{k-1}) \pmod{M} \quad (5)$$

Since the value of $k$ may, in general, be very large, we cannot use simple iteration; such a method would require time proportional to $k$. The quantity in equation (4) is amenable to the so-called "binary method". Since this method may not be familiar in the general form we will use later, we pause to sketch it here.

Sometimes a function $f(n)$ will be defined in terms of $f(n-1)$. To compute $f(32)$, for example, we must first compute $f(31), f(30), \ldots, f(1)$. If, however, it is possible to quickly compute $f(2n)$ in terms of $f(n)$, we can compute $f(32)$ in only 5 steps:

$$f(1) \rightarrow f(2) \rightarrow f(4) \rightarrow f(8) \rightarrow f(16) \rightarrow f(32)$$

We call this sort of idea a binary scheme. Suppose $G$ is a dyadic function such that

$$\alpha G \omega + 1 \rightarrow \alpha \text{ INCRE } \alpha G \omega$$

$$\alpha G 2\omega \rightarrow \text{ DOUBLE } \alpha G \omega$$

$$\alpha G 0 \rightarrow \text{ IDENT } \alpha$$

Then the function $\text{BIN}$ computes $\alpha G \omega$ in time proportional to $\log(\omega)$.

$$\forall Z+X \text{ BIN } N$$

$$[1] \text{ GENERAL BINARY SCHEME}$$

$$[2] +((N=0),1=2(N)/L0,L1)$$

$$[3] Z=\text{DOUBLE } X \text{ BIN } N+2$$

$$[4] +0$$

$$[5] L0:Z+\text{IDENT } X$$

$$[6] +0$$

$$[7] L1:Z+\text{INC } X \text{ BIN } N-1$$

For example, if the definitions of IDENT, INCRE, and DOUBLE are

$$\text{IDENT} : (11+P\omega)=11+P\omega$$

$$\text{INCRE} : a\cdot x\omega$$

$$\text{DOUBLE} : a+\cdot x\omega$$

then $\text{M BIN N}$ computes the $N$-th power of the matrix $M$.

$$M \text{ BIN } 10$$

$$\begin{bmatrix}
0 & 1 \\
1 & 1 \\
\end{bmatrix}
\rightarrow M \text{ BIN } 29$$

$$\begin{bmatrix}
34 & 55 \\
55 & 89 \\
\end{bmatrix}
\rightarrow M \text{ BIN } 514229$$

$$\begin{bmatrix}
317811 & 514229 \\
514229 & 832040 \\
\end{bmatrix}$$

Table 1: Parameters for Some Common Pseudo-Random Number Generators

<table>
<thead>
<tr>
<th>Where used</th>
<th>a</th>
<th>c</th>
<th>$X_0$</th>
<th>M</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>APL360 et al.</td>
<td>16007</td>
<td>0</td>
<td>16007</td>
<td>$2^{31}-1$</td>
<td>[13]</td>
</tr>
<tr>
<td>Waterloo microAPL</td>
<td>1001</td>
<td>0</td>
<td>245</td>
<td>33749</td>
<td>[14]</td>
</tr>
<tr>
<td>DG AOS/VS APL</td>
<td>16007</td>
<td>273925815</td>
<td>57794377</td>
<td>$2^{32}$</td>
<td>[18]</td>
</tr>
<tr>
<td>APL/MYRIADE</td>
<td>23813</td>
<td>0</td>
<td>1</td>
<td>33749</td>
<td>[15]</td>
</tr>
<tr>
<td>Burroughs APL/700</td>
<td>125247890725</td>
<td>11611703737</td>
<td>13113130466</td>
<td>$2^{39}$</td>
<td>[15, 17]</td>
</tr>
<tr>
<td>DEC APL/81</td>
<td>265477815</td>
<td>7261567085</td>
<td>0</td>
<td>$2^{60}$</td>
<td>[17]</td>
</tr>
<tr>
<td>D. H. Lehmer</td>
<td>129</td>
<td>1</td>
<td></td>
<td></td>
<td>[12]</td>
</tr>
<tr>
<td>A. Rencher</td>
<td>59</td>
<td>0</td>
<td></td>
<td></td>
<td>[31]</td>
</tr>
<tr>
<td>R. H. Coveney</td>
<td>61</td>
<td>0</td>
<td></td>
<td></td>
<td>[30]</td>
</tr>
</tbody>
</table>

Table 1: Parameters for Some Common Pseudo-Random Number Generators
It is now clear how to compute the quantity in equation (4) quickly. We could use the following definitions, where $M$ is a global variable.

\[
\text{IDENT} : M \mid 1
\]
\[
\text{INCRE} : M \mid a \times \omega
\]
\[
\text{DOUBLE} : M \mid \omega \times \omega
\]

Here $X \text{ BIN } N$ computes $M \mid X^N$.

It is a little harder to compute the quantity in (5) efficiently.

[v] Knuth [8] replaces the polynomial $1 + a + a^2 + \cdots + a^{k-1}$ by the expression $\frac{a^k - 1}{a - 1}$, which, unfortunately, is expensive to compute when $k$ is large. And we cannot replace the numerator and denominator by their values (mod $M$) when $a - 1$ has a factor in common with $M$, since then $a - 1$ does not have a multiplicative inverse (mod $M$).

We now sketch an efficient way to calculate both

\[a^k \pmod{M}\]

and

\[1 + a + a^2 + \cdots + a^{k-1} \pmod{M}\]

simultaneously.

We will compute with pairs of numbers,

\[(f(n), g(n)) = (a^n, 1 + a + a^2 + \cdots + a^{n-1});\]

all values are considered (mod $M$). Then we find

\[
f(0) = 1; \quad g(0) = 0
\]
\[
f(n+1) = a^{n+1} - a \cdot a^n = a \cdot f(n)
\]
\[
g(n+1) = 1 + a + a^2 + \cdots + a^n = 1 + a \cdot g(n)
\]
\[
f(2n) = a^{2n} = (a^n)^2 = f(n)^2
\]
\[
g(2n) = 1 + a + a^2 + \cdots + a^{2n-1} = g(n) + f(n) \cdot g(n)
\]

These equations reduce the problem to a simple application of the binary scheme.

\[
\text{IDENT} : M \mid 1 0
\]
\[
\text{INCRE} : M \mid 0 1 + a \times \omega
\]
\[
\text{DOUBLE} : M \mid (\omega[1] \times 2), \omega[2] \times 1 + \omega[1]
\]

Now it is easy to compute $X_k$ given $X_0$, $a$, $c$, and $M$ as global variables:

\[
\text{LINCON} : M \mid (X_0, C) \ast \ast A \text{ BIN } \omega
\]

The function \text{LINCON} takes as its right argument the number of invocations of ?, and returns the proper value of \text{DRL}. For example, suppose $X_0 = 73$, $a = 371$, $c = 995$, and $M = 1024$. Then we find

\[
\text{LINCON} 0
\]
\[
73
\]
\[
\text{LINCON} 100
\]
\[
49
\]
\[
\text{LINCON} 1000
\]
\[
985
\]

Unfortunately, it is possible for this method to give incorrect answers in practice; this occurs when $M^2$ is so large that it is not exactly representable in the word size of the machine. We must then use an extended precision arithmetic package. In Appendix II, we give the functions to solve the first problem for the APL360-derived pseudo-random number generator. The function \text{RLAI} returns, for its non-negative integer right argument $K$, what \text{DRL} would be after $K$ invocations of ?. For example,

\[
\text{DRL}
\]
\[
16807
\]
\[
0 \times 200001
\]
\[
\text{DRL}
\]
\[
1625538587
\]
\[
\text{RLAI} 2000
\]
\[
1625538587
\]

4. The Second Problem.

Finding $k$ so that $k$ invocations of $?$ result in some chosen value of \text{DRL} is a much harder problem, as we will see below.

Suppose in equation (3) above we have been given $X_k$, $X_0$, $a$, $c$, and $M$; we wish to find $k$. Then from

\[X_k = a^k X_0 + c (1 + a + \cdots + a^{k-1}) \pmod{M}\]

we multiply both sides by $(a - 1)$ and add $c$ to get

\[(a - 1) X_k + c = a^k ((a - 1) X_0 + c) \pmod{M}\]

If we assume that $X_1 = X_0$ (mod $p$) for all primes $p$ that divide $M$, then

\[X_1 - X_0 \neq 0 \pmod{p}
\]
\[aX_0 + c - X_0 \neq 0 \pmod{p}
\]
\[(a - 1) X_0 + c \neq 0 \pmod{p},
\]

and so $(a - 1) X_0 + c$ is invertible (mod $M$), since it is invertible for all primes $p$ dividing $M$. Shallit

- 245 - Merrily We Roll Along: Some Aspects of ?
Now let \( d \) be the inverse of \((a - 1)X_0 + c \pmod{M}\). We see that
\[
a^k = ((a - 1)X_k + c) \cdot d \pmod{M}
\]
so we can reduce the linear congruential method to solving
\[
a^k = r \pmod{M}
\]
for \( k \). This problem is called \textit{index-finding}, or computing the discrete logarithm. We sometimes write \( k = \text{ind}_a r \), where the modulus \( M \) is understood. Index-finding is a problem that is well-known to be difficult in general if \( M \) is large; even if \( M \) is prime and the complete factorization of \( M - 1 \) is known, no really good methods exist. For example, see \cite{1} or \cite{21}.

Let us now return for a moment to the simplifying assumption that \( X_1 = X_0 \pmod{p} \). This is not really much of a restriction, since if \( X_1 = X_0 \pmod{p} \), then in fact each \( X_i \) will be equal \( \pmod{p} \); this is not a very random sequence!

Even so, it is possible to solve for the case where \( X_0 = X_1 \pmod{p} \) by splitting \( M \) into the product of \( M_1 \) and \( M_2 \), where \( X_1 \neq X_0 \pmod{q} \) for all primes \( q \) dividing \( M_1 \) but \( X_1 = X_0 \pmod{p} \) for all \( p \) dividing \( M_2 \). Then we just solve
\[
a^k = ((a - 1)X_k + c) \cdot d \pmod{M_1}.
\]
If a solution exists, then the solution is valid \( \pmod{M} \). This is left to the reader.

In this paper we will only discuss index-finding where \( M \) is a prime, say \( p \). If \( M \) is composite, it is easy to break the problem up into index-finding for each prime power factor, and combine the results using the Chinese Remainder Theorem.

5. Index-finding \( \pmod{p} \).

We consider the problem of solving equation (9) for \( a \), where \( M = p \), an odd prime. First, we can assume that \( a \) is a primitive root. If it is not, we can find a primitive root \( g \) by the method of Appendix I. Then it is easy to find \( \text{ind}_a r \) by use of the formula
\[
\text{ind}_a r = \text{ind}_g r \cdot \text{ind}_a g \pmod{p-1},
\]
Hence equation (9) can be rewritten as
\[
g^k = r \pmod{p}
\]
where \( g \) is a generator and \( p \) is an odd prime.

If \( p \) is not too large, then we can solve the discrete logarithm problem easily. The function \( \text{POW} \) below computes \( B[1][2] \cdot G^s B[1][1] \) for a vector \( G \) and matrix \( B \) using a modification of the binary scheme.

\begin{verbatim}
V Z=IP P
[1] Z=P
V TAB=IP ((P-1)g) POW (-t+IP-1) MOD P
\end{verbatim}

returns a table of the powers of \( A \); hence
\[
-t+1+((P-1)g) \cdot \text{POW} (-t+1P-1) \pmod{P} \cdot r
\]
gives the discrete logarithm.

If we wish to solve the problem for a number of different \( r \), while holding \( G \) and \( P \) fixed, we can precompute a table and solve the problem by table lookup:

\begin{verbatim}
V Z=IP P
[1] Z=P
V TAB=IP ((P-1)g) POW (-t+1P-1) MOD P
\end{verbatim}

Here \( IP \) is a function that, given a permutation, computes the inverse permutation. Then \( TAB[A] \) solves the discrete logarithm problem for any particular \( A \).

If \( P \) has more than 4 or 5 digits, this method becomes impractical. A second method for index-finding was proposed by Shanks \cite{9}. It is best understood by considering a simple example. Suppose \( p = 23 \), \( g = 11 \), and \( r = 14 \). Assume
\[
g^k = r \pmod{p}
\]
Suppose \( k = 5c + d \), where \( 0 \leq c, d \leq 4 \). Then
\[
g^k = g^{5c+d} = r \pmod{p}
\]
Hence
\[
g^{5c} = r \cdot g^{-d} \pmod{p}
\]
We tabulate both sides of this equation for all values of \( c \) and \( d \):

\begin{tabular}{|c|c|c|}
\hline
\( c \) & \( g^c \pmod{p} \) & \( d \) & \( r \cdot g^{-d} \pmod{p} \) \\
\hline
0 & 1 & 0 & 14 \\
1 & 5 & 1 & 18 \\
2 & 2 & 2 & 10 \\
3 & 1 & 3 & 3 \\
4 & 4 & 4 & 17 \\
\hline
\end{tabular}

\textbf{Table 2: Example of Shanks' method of index-finding}
We see that the two sides of equation (11) coincide for
$c = 3, d = 2$. Hence $k = 5c + d = 17$.

To use this method in general, we write the exponent $k$ in the form

$$c \sqrt{p} + d,$$

create two lists like those in Table 2, and search for an
element common to both lists. If your APL system has
implemented dyadic iota efficiently (see, for example, Ber-
necky [2]), this search can be done in time proportional to
$\sqrt{p} \log p$; unfortunately, many APL systems use a simple
search procedure that takes too much time when $p$ is
large. We present a solution that uses the upgrade and
downgrade primitives, and hence requires time propor-
tional to $\sqrt{p} \log p$.

The function $\text{INDEX}$ below takes a left argument that
is a prime number, $p$. The right argument is a two ele-
ment vector; the first element is a generator, $g$; the
second is a number $r$ such that $1 \leq r < p$. The result is
\begin{align*}
\text{INDEX} & \quad \text{GR; B; D; S; I; T; G; R} \\
1 & \quad \text{G} \div \text{GR}[1] \\
2 & \quad \text{R} \div \text{GR}[2] \\
3 & \quad \text{T} \div \text{P} \times 0.5 \\
4 & \quad \text{I} \div \text{T} + 1 \\
5 & \quad \text{B} \div (\text{T} \ast \text{G}) \ast \text{POW(1} \ast \text{T}) \mod \text{P} \\
6 & \quad \text{D} \div \text{P} \ast \text{R} \ast (\text{T} \ast \text{INV G P}) \ast \text{POW I MOD P} \\
7 & \quad \text{S} \div \text{B} \ast \text{FM D} \\
8 & \quad \text{Z} \div (\text{P} - 1) \ast \text{T} - 1 \ast \text{S} ; \text{D} ; \text{B} [\text{S}] \\
9 & \quad \text{23 INDEX 11 14} \\
17 & \quad \text{Shank's method works well when $p$ is smaller than $10^7$
or so; unfortunately, it is too slow for the case we are
most interested in: where $p = 2^{31} - 1$.

We turn to another method of index-finding first dis-
cussed by Pohlig and Hellman [20]. Their method
requires a table of length $q$ for every prime $q$ dividing
$p-1$; hence it is suitable only when $p-1$ has all small
prime factors. Luckily for us, the factorization of $2^{31} - 2$
is

$$2^{31} - 2 = 2 \cdot 3^2 \cdot 7 \cdot 11 \cdot 31 \cdot 151 \cdot 331$$

and the factors are small enough to make the computation feasible.

The Pohlig-Hellman method for solving equation (10)
is to compute the value of $k$ modulo each prime power factor dividing $p-1$; then the exponent $k$ is recon-
structed via the Chinese Remainder Theorem. We precompute a table holding the values

$$\begin{align*}
& \text{XI} \div \text{P} - 1 \\
& \text{1} , \text{g}^{q} , \text{g}^{q} \cdot \cdots \cdot \text{g}^{q} \quad ; \quad (12)
\end{align*}$$

these are the $q$-th roots of unity. If

$$g^k = r \pmod{p}$$

then, by raising both sides to the $\frac{p-1}{q}$ power, we find

$$\frac{k(p-1)}{q} = r \pmod{p}$$

and so $r^q$ is one of the numbers in (12). We can find
the corresponding value of $k$ by a simple table lookup.
This gives $k \pmod{q}$; we do this for each $q$ dividing
$p-1$. See [20].

There is another problem in the implementation of the
Pohlig-Hellman technique for $p = 2^{31} - 1$: $p^2$ cannot be
exactly represented in System/360 architecture. There-
fore, we must resort to extended-precision arithmetic.
The most costly operation is reducing an extended-
precision value $\pmod{p}$; luckily, however, for
$p = 2^{31} - 1$, there is an easy method: If we write the
number in base $2^{31}$, then $x \pmod{2^{31} - 1}$ is just the sum
of the digits of $x$. This is an easily-proved generalization
of the well-known technique called "casting out nines". If
we choose our base of representation to be a small power
of two, say $2^{20}$, then it is easy to convert the number to
base $2^{31}$ for computation of the remainder $\pmod{2^{31} - 1}$.

The function $\text{PHIF}$ (Pohlig-Hellman index-finding)
uses the above techniques to compute the index of its
right argument. It requires the use of auxiliary tables and
constants which can be precomputed once and stored.
These variables are computed by the function $\text{SETUP}$.
The function $\text{NIQ}$ (number of invocations of ?) takes a
right argument which is a purported value of DRL, and
returns the least number of invocations of ? needed to get
that value. For example:

$$\begin{align*}
\text{NIQ} & \quad 16807 \\
\text{DRL} & \quad \text{OP?2000P} \\
\text{DRL} & \quad 0 \text{P72000P} \\
\text{DRL} & \quad 1625538587 \\
\text{NIQ} & \quad \text{DRL} \\
\text{DRL} & \quad 2000 \\
\text{NIQ} & \quad \text{DRL}
\end{align*}$$

The function $\text{NIQ}$ takes about 5 seconds of CPU time
to execute on an IBM 4341. The amount of time is
independent of its right argument. $\text{NIQ}$ does better than
simple search if the number of invocations of ? is greater
than about 6000.

See Appendix II for definitions.

In closing, it may be of some interest to note that
Plumstead [19] has shown how to deduce the values of $a,$
c, and $M$ in equation (2), given only the first few values of $X_r.$
6. Acknowledgements.

The author would like to thank Gene McDonnell for several suggestions, and the referee for corrections to the first draft.

Thanks also go to Doug Forkes, who suggested a way to speed up the implementation of Pohlig and Hellman's method.

References


Appendix I

Some Number-Theoretic Functions

1. Multiplicative inverses (mod N).

Given two relatively prime numbers A and N, a well-known theorem states that it is possible to find B such that 1 = N × A × B. This integer B is called the (multiplicative) inverse of A (mod N).

One quick way to find the inverse of A (mod N) is to use continued fractions. Continued fractions are a subject in themselves and we do not have space here to go into the theory in detail. The interested reader is referred to [4] and [18].

The function INV takes a two element vector X as its right argument; the result is the inverse of X[1] (mod X[2]). It is assumed that 1 = v/X.

INV: 1 (X[2]) (X[1]) (X[2]) (X[2])

CF: (X[1]/X[2]) (X[1]/X[2]) (X[1]/X[2])

2. The Chinese Remainder Theorem.

This theorem states that the system of equations

x = a1 (mod M1)

x = a2 (mod M2)

has a unique solution 0 ≤ x < M1M2 if M1 and M2 are relatively prime.

One way to find this solution is to compute b1 and b2 such that

b1 = 1 (mod M1), b2 = 0 (mod M2)

b1 = 0 (mod M2), b2 = 1 (mod M1)

Then x is given by

x = a1b1 + a2b2 (mod M1M2).

In fact we can take b1 = M2c2, b2 = M1c1, where c2 is
Appendix II

Function Listings

the inverse of $M_2 \pmod{M_1}$ and $c_1$ is the inverse of $M_1 \pmod{M_2}$.

The function $CRT$ is defined such that

$$M_1 A \Longleftrightarrow CRT M \Longleftrightarrow A$$

for two-element integer vectors $A$ and $M$ with $1 - \sqrt{M}$.

$CRT: (x/\omega) \rightarrow a + \ast (\phi \omega) \times (INV \phi \omega), INV \omega$

3. Primitive Roots

A primitive root, or generator, for an odd prime $p$ is a number $g$ such that the $p - 1$ numbers

$g, g^2, \ldots, g^{p-1}$

form a permutation of $1, 2, \ldots, p - 1$. Every prime has at least one primitive root [4].

Given the factorization of $p - 1$, it is easy to determine if a given $g$ is a primitive root: we just check to see that

$$g^{(p-1)/q} \neq 1 \pmod{p}$$

for all primes $q$ dividing $p - 1$. This is implemented in the function $IPR$. The left argument is a 2-column matrix $F$ containing the factorization of $P - 1$ in canonical form; the first column contains distinct primes and the second column contains exponents. The right argument $G$ is a purported generator. The result is 1 if $G$ is a primitive root; 0 otherwise.

We can find a generator quickly simply testing $2, 3, 4, \ldots$ in order. This is done with the function $FPR \omega$, which finds the first primitive root of the odd prime $\alpha$ greater than or equal to $\omega$.

$$FPR: w \rightarrow \alpha \rightarrow FPR \omega \rightarrow \alpha$$