APL '83: Proceedings of the international conference on APL
April 1983
Pages 243–249

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 ⎕*RL*, 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 ⎕*RL*, 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 ⎕*RL*.

Similarly, you might want to know what value should be assigned to ⎕*RL* 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 ⎕*RL* after ? has been executed k times.

(b) Given ⎕*RL*, 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 ∨ and ∧ to represent gcd and 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 [5].

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*PLUS, and SHARP APL, generates a new random link from the old one by

```
⎕RL← 2147483647|16807 ×⎕RL          (1)
```

See McDonnell [14]. Note that $2147483647 = 2^{31} - 1$, a prime, and $16807 = 7^5$. The default for ⎕*RL* 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 *APL*\360 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

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

Here M is called the **modulus** , a is called the **multiplier** , and c is called the **increment.** See Knuth [8]. The notation (mod 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.

Where used	a	c	X_0	M	Reference
APL/360 *et al.*	16807	0	16807	$2^{31}-1$	[11]
Waterloo microAPL	1001	0	345	32749	[24]
DG AOS/VS APL	16807	273905815	57794127	2^{32}	[16]
APL*MYRIADE	23813	0	1	32749	[22]
Burroughs APL/700	152587890725	116177073375	131131704506	2^{39}	[15 17]
DEC APLSF	30517578125	7261067085	0	2^{36}	[7]
D. H. Lehmer	14^{29}	0	--	$2^{31}-1$	[12]
A. Rotenberg	129	1	--	2^{35}	[23]
R. R. Coveyou	5^3	0	--	2^{13}	[10]

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

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

$$X_{n+k} = a^k X_n + c(1 + a + \cdots + a^{k-1}) \ (\text{mod } M). \qquad (3)$$

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

$$a^k X_0 \ (\text{mod } M) \qquad (4)$$

and

$$c(1 + a + a^2 + \cdots + a^{k-1}) \ (\text{mod } M) \qquad (5)$$

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 $f(31), f(30), \cdots, f(1)$. 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

$$\alpha\ G\ \omega{+}1 \longleftrightarrow \alpha\ INCRE\ \alpha\ G\ \omega$$

$$\alpha\ G\ 2\omega \longleftrightarrow DOUBLE\ \alpha\ G\ \omega$$

$$\alpha\ G\ 0 \longleftrightarrow IDENT\ \alpha$$

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

```
     ∇ Z←X BIN N
[1]    ⍝ GENERAL BINARY SCHEME
[2]     →((N=0),1=2|N)/L0,L1
[3]     Z←DOUBLE X BIN N÷2
[4]     →0
[5]    L0:Z←IDENT X
[6]     →0
[7]    L1:Z←X INCRE X BIN N-1
     ∇
```

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

```
IDENT  : (⍳1↑⍴⍵)∘.=⍳1↑⍴⍵
INCRE  : ⍺+.×⍵
DOUBLE : ⍵+.×⍵
```

then *M BIN N* computes the *N*-th power of the matrix M.

```
      ⎕←M←2 2⍴0 1 1 1
0 1
1 1
      M BIN 10
34 55
55 89
      M BIN 29
317811 514229
514229 832040
```

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.

```
    IDENT : M|1
    INCRE : M|α×ω
    DOUBLE: M|ω×ω
```

Here *X BIN N* computes $M|X{*}N$.

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

[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

$$f(0) = 1; \quad g(0) = 0$$

$$f(n+1) = a^{n+1} = a \cdot a^n = a \cdot f(n)$$

$$g(n+1) = 1 + a + a^2 + \cdots + a^n = 1 + a \cdot g(n)$$

$$f(2n) = a^{2n} = (a^n)^2 = f(n)^2$$

$$\begin{aligned} g(2n) &= 1 + a + a^2 + \cdots + a^{2n-1} \\ &= g(n) + f(n)g(n) \end{aligned}$$

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

```
    IDENT : M| 1 0
    INCRE : M| 0 1+α×ω
    DOUBLE: M| (ω[1]*2),ω[2]×1+ω[1]
```

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

```
    LINCON: M|(X0,C)+.×A BIN ω
```

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

```
      LINCON 0
73
      LINCON 100
49
      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 *APL*\360-derived pseudo-random number generator. The function *RLAI* returns, for its non-negative integer right argument K, what ⎕*RL* would be after K invocations of ?. For example,

```
      ⎕RL
16807
      0ρ?2000ρ1

      ⎕RL
1625538587
      RLAI 2000
1625538587
```

4. The Second Problem.

Finding k so that k invocations of ? result in some chosen value of ⎕*RL* 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

$$X_k = a^k X_0 + c(1 + a + \cdots + a^{k-1}) \pmod M \qquad (6)$$

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

$$(a-1)X_k + c = a^k((a-1)X_0 + c) \pmod M \qquad (7)$$

If we assume that $X_1 \not\equiv X_0 \pmod p$ for all primes p that divide M, then

$$X_1 - X_0 \not\equiv 0 \pmod p$$

$$aX_0 + c - X_0 \not\equiv 0 \pmod p$$

$$(a - 1)X_0 + c \not\equiv 0 \pmod p,$$

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

Now let d be the inverse of $(a-1)X_0 + c \pmod{M}$. We see that

$$a^k = ((a-1)X_k + c)\,d \pmod{M} \tag{8}$$

so we can reduce the linear congruential method to solving

$$a^k = r \pmod{M} \tag{9}$$

for k. This problem is called **index-finding** , or computing the **discrete logarithm** . We sometimes write $k = \operatorname{ind}_a r$, 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 [1] or [21].

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

Even so, it is possible to solve for the case where $X_0 \equiv X_1 \pmod{p}$ by splitting M into the product of M_1 and M_2, where $X_1 \not\equiv X_0 \pmod{q}$ for all primes q dividing M_1 but $X_1 \equiv X_0 \pmod{p}$ for all p dividing M_2. Then we just solve

$$a^k \equiv ((a-1)X_k + c)\,d \pmod{M_1}.$$

If a solution exists, then the solution is valid (mod 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 (mod 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 $\operatorname{ind}_a r$ by use of the formula

$$\operatorname{ind}_a r = \frac{\operatorname{ind}_g r}{\operatorname{ind}_g a} \pmod{p-1}$$

Hence equation (9) can be rewritten as

$$g^k \equiv r \pmod{p} \tag{10}$$

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 *POW* below computes $B[;2]|G{*}B[;1]$ for a vector G and matrix B using a modification of the binary scheme.

```
     ∇ Z←A POW B;C;S
[1]    ⍝ <A> IS A VECTOR OF LENGTH K
[2]    ⍝ <B> IS A K×2 MATRIX
[3]    ⍝ RESULT IS B[;2]|A*B[;1]
[4]    ⍝
[5]     A←(¯1↑1,⍴A)⍴A
[6]     C←B[;2]
[7]     B←B[;1]
[8]     Z←(⍴A)⍴1
[9]    ⍝
[10]   L1:
[11]    S←((B≠0)∧0≠1|B←B÷2)/⍳⍴B
[12]    Z[S]←C[S]|A[S]×Z[S]
[13]    S←(0≠B←⌊B)/⍳⍴B
[14]    A[S]←C[S]|A[S]×A[S]
[15]    →(∨/B≠0)/L1
     ∇
```

Now

```
((P-1)⍴G) POW (¯1+⍳P-1) MOD P
```

returns a table of the powers of A; hence

```
¯1+(((P-1)⍴G) POW (¯1+⍳P-1) MOD P)⍳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:

```
    ∇ Z←IP P
[1]   Z←P
[2]   Z[P]←⍳⍴P
    ∇
      TAB←IP ((P-1)⍴G) POW (¯1+⍳P-1) MOD P
```

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 [9]. It is best understood by considering a simple example. Suppose $p = 23$, $g = 11$, and $r = 14$. Assume

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

Suppose $k = 5c + d$, where $0 \leqslant c, d \leqslant 4$. Then

$$g^k \equiv g^{5c+d} \equiv r \pmod{p}$$

Hence

$$g^{5c} \equiv r \cdot g^{-d} \pmod{p} \tag{11}$$

We tabulate both sides of this equation for all values of c and d:

c	$g^{5c} \pmod{p}$	d	$r \cdot g^{-d} \pmod{p}$
0	1	0	14
1	5	1	18
2	2	2	10
3	10	3	3
4	4	4	17

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

$$c\left\lceil \sqrt{p} \right\rceil + d,$$

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, Bernecky [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 proportional to $\sqrt{p}\log p$.

The function *INDEX* below takes a left argument that is a prime number, p. The right argument is a two element vector; the first element is a generator, g; the second is a number r such that $1 \leqslant r < p$. The result is $\text{ind}_g r$.

```
      ∇ Z←P INDEX GR;B;D;S;I;T;G;R
[1]     G←GR[1]
[2]     R←GR[2]
[3]     T←⌈P*0.5
[4]     I←¯1+⍳T
[5]     B←(T⍴G) POW(I×T) MOD P
[6]     D←P|R×(T⍴INV G,P) POW I MOD P
[7]     S←B FM D
[8]     Z←(P-1)|T⊥¯1+S,D⍳B[S]
      ∇
        23 INDEX 11 14
17
```

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 discussed 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 reconstructed via the Chinese Remainder Theorem. We precompute a table holding the values

$$1, g^{\frac{(p-1)}{q}}, g^{\frac{2(p-1)}{q}}, \cdots, g^{\frac{(q-1)(p-1)}{q}}; \qquad (12)$$

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

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

and so $r^{\frac{p-1}{q}}$ 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. Therefore, 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 (mod $2^{31} - 1$).

The function *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 *SETUP*. The function *NIQ* (number of invocations of ?) takes a right argument which is a purported value of ⎕*RL*, and returns the least number of invocations of ? needed to get that value. For example:

```
        ⎕RL
16807
        0⍴?2000⍴1

        ⎕RL
1625538587
        NIQ ⎕RL
2000
```

The function *NIQ* takes about 5 seconds of CPU time to execute on an IBM 4341. The amount of time is independent of its right argument. *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_i.

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

[1] Leonard Adleman, A Subexponential Algorithm for the Discrete Logarithm Problem, with Applications to Cryptography, *Proceedings of the 1980 Conference on Foundations of Computer Science (FOCS)*, IEEE, 55-60.

[2] Bob Bernecky, Speeding up Dyadic Iota and Dyadic Epsilon, APL 73 Congress Proceedings, North-Holland.

[3] George S. Fishman and Louis R. Moore, A Statistical Evaluation of Multiplicative Congruential Random Number Generators with Modulus $2^{31} - 1$, *Journ. Amer. Stat. Assoc.* 77 (1982) 129-136.

[4] G. H. Hardy and E. M. Wright, An Introduction to the Theory of Numbers, Oxford, Clarendon Press (1971).

[5] Thomas N. Herzog, Generating Uniform Pseudorandom Numbers, *APL Quote Quad* 12 (2) (1978) 22-23.

[6] K. E. Iverson, Notation as a Tool of Thought, *CACM* 23 (1980) 465.

[7] Doug Keenan, personal communication (I P Sharp Mailbox #1675771).

[8] D. E. Knuth, The Art of Computer Programming, V. 2 (Seminumerical Algorithms), Addison-Wesley, Reading, Mass. (1981), 9-14.

[9] D. E. Knuth, The Art of Computer Programming, V. 3 (Sorting and Searching), Addison-Wesley, Reading, Mass. (1973) 9, 575-576.

[10] J. B. Kruskal, Extremely Portable Random Number Generator, *CACM* 12 (1969) 93-94.

[11] R. H. Lathwell and J. E. Mezei, A Formal Description of APL, IBM Philadelphia Scientific Center Technical Report 320-3008, (November, 1971) 8,11.

[12] D. H. Lehmer, Random Number Generation on the BRL High-Speed Computing Machines, rev. by M. L. Juncosa, *Math. Rev.* **15** (1954) 559.

[13] E. E. McDonnell, A Notation for the GCD and LCM functions, APL 75 Congress Proceedings, ACM (1975) 240-243.

[14] E. E. McDonnell, How the Roll Function Works, *APL Quote Quad* 8 (3) (1978) 42-47.

[15] E. E. McDonnell, personal communication (I P Sharp Mailbox # 1675603).

[16] Randall Mercer, personal communication.

[17] Ronald C. Murray, personal communication (I P Sharp Mailbox # 1678290).

[18] C. D. Olds, Continued Fractions, Random House, 1963.

[19] Joan B. Plumstead, Inferring a Sequence Generated by a Linear Congruence, to appear.

[20] Stephen C. Pohlig and Martin E. Hellman, An Improved Algorithm for Computing Logarithms over GF(p) and its Cryptographic Significance, *IEEE Trans. Info. Theory* IT-24 (1978) 106-110.

[21] J. M. Pollard, Monte-Carlo Methods for Index Computation (mod p), *Math. Comp.* 32 (1978) 918-923.

[22] L. P. A. Robichaud, personal communication (I P Sharp Mailbox # 1675876).

[23] A. Rotenberg, A New Pseudo-Random Number Generator, *JACM* 7 (1960) 75-77.

[24] John Wilson, personal communication.

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 $1 = N \mid A \times B$. 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 $1 = \vee/X$.

```
INV: ω[2]|(¯1*⍴T)×(FC2 T←CF ω)[2;1]
FC2: (FC2 ¯1↓ω)+.×2 2⍴0 1 1,¯1↑ω : 0=⍴ω : 2 2⍴0 1 1 0
CF:  (⌊÷/ω), CF |\⌽ω : 0=¯1↑ω : ⍳0
```

2. The Chinese Remainder Theorem.

This theorem states that the system of equations

$$x = a_1 \pmod{M_1}$$
$$x = a_2 \pmod{M_2}$$

has a unique solution $0 \leqslant x < M_1 M_2$ if M_1 and M_2 are relatively prime.

One way to find this solution is to compute b_1 and b_2 such that

$$b_1 = 1 \pmod{M_1}, \quad b_2 = 0 \pmod{M_1}$$
$$b_1 = 0 \pmod{M_2}, \quad b_2 = 1 \pmod{M_2}$$

Then x is given by

$$x = a_1 b_1 + a_2 b_2 \pmod{M_1 M_2}.$$

In fact we can take $b_1 = M_2 c_2$, $b_2 = M_1 c_1$, where c_2 is the inverse of M_2 (mod M_1) and c_1 is the inverse of M_1 (mod M_2).

The function *CRT* is defined such that

$$M|A\ CRT\ M \longleftrightarrow A$$

for two-element integer vectors A and M with $1 = \vee/M$.

```
CRT: (×/ω) | α +.× (⌽ω) × (INV ⌽ω), INV ω
```

3. Primitive Roots

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

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

form a permutation of $1, 2, \cdots, p-1$. 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

$$g^{\frac{p-1}{q}} \not\equiv 1 \pmod p$$

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.

```
IPR: ∧/1≠((1↑ρα)ρω) POW (T÷α[;1]) MOD 1+T←α[;1]×.*α[;2]
```

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

```
FPR: ω : ~α IPR ω : α FPR ω+1
```

Appendix II

Function Listings

```
NIQ: (P-1)|(PHIF ω)-1

RLAI: B⊥(0,G) TOTHE ω+1

TI: MOD231 CAR +/0 ¯1⊖(3↑α)∘.×ω

MOD: (,α), [1.5] ω

    ∇ SETUP;K
     ⍝ EXECUTE ONCE TO SET UP CONSTANTS AND TABLES
      B←1048576 ⍝ BASE OF COMPUTATION = 2*20
      P←2147483647 ⍝ (2*31)-1
      G←16807 ⍝ THE GENERATOR
      F← 2 7 9 11 31 151 331 ⍝ PRIME POWER FACTORS OF P-1
      C←'1 0 CRT' APPLY F,[1.5] F×.*(⍳7)∘.≠⍳7
      K←1
     L1:
      ⍎'V',(⍕F[K]),'←MTAB F[K]'
      →((ρF)≥K←K+1)/L1
    ∇

    ∇ Z←Y TOTHE N
      Z← 0 1
      →(N=0)/0
     L1:→(0=1|N←N÷2)/L0
      Z←Y TI Z
      →(0=N←⌊N)/0
     L0:Y←Y TI Y
      →L1
    ∇

    ∇ Z←MOD231 V;T;S
      T←,⍉(20ρ2)⊤V
      S←⌈(ρT)÷31
      Z←(2ρB)⊤P|+/2⊥⍉(S,31)ρ(-S×31)↑T
    ∇

    ∇ Z←CAR V
      Z←(0,B|V)+(⌊V÷B),0
     L0:
      →(∧/B>Z)/0
      Z←(B|Z)+⌊(1↓Z,0)÷B
      →L0
    ∇

    ∇ Z←PHIF A;M
     ⍝ POHLIG-HELLMAN INDEX-FINDING FOR <A>
      A←(2ρB)⊤A
      M←'A ILOC' APPLY F
      Z←(P-1)|M+.×C
    ∇

    ∇ Z←F APPLY X
      Z←⍳0
      K←1
     L0:
      →(K>1↑ρX)/0
      Z←Z,⍎F,' X[K',((¯1+ρρX)ρ';'),']'
      K←K+1
      →L0
    ∇

    ∇ Z←MTAB Q;R;T
      R←T←(0,G) TOTHE(P-1)÷Q
      Z←,1
     L0:
      →(Q=ρZ)/0
      Z←Z,B⊥R
      R←R TI T
      →L0
    ∇

    ∇ Z←A FM B;E;F;G
     ⍝ FIND MATCH
     ⍝ <A> AND <B> ARE VECTORS WITH AT LEAST
     ⍝ ONE ELEMENT IN COMMON; RESULT IS THE INDEX
     ⍝ INTO <A> OF A COMMON ELEMENT
      E←⍋A,B
      F←⌽⍒A,B
      G←(E≠F)/⍳ρE
      Z←⌊/E[G]
    ∇

    ∇ Z←X ILOC Y;T
      T←B⊥X TOTHE(P-1)÷Y
      Z←(⍎'V',(⍕Y),'⍳T')-1
    ∇
```