Uniform Random Number Generators and Primitive Trinomials*

Richard P. Brent
Computing Laboratory
University of Oxford
rpb@comlab.ox.ac.uk

1 May 2001

*To be presented at OUCL, 1 May 2001.
Copyright ©2001, R. P. Brent. oxford3t

Abstract

In this talk, which describes joint work with Samuli Larvala and Paul Zimmermann, we consider the problem of testing trinomials over GF(2) for irreducibility or primitivity. In particular, we consider trinomials whose degree is the exponent of a Mersenne prime. We describe a new algorithm for testing such trinomials. The algorithm is significantly faster than the standard algorithm, and has been used to find primitive trinomials of degree 3021377. Previously, the highest degree known was 859433.

One of the applications is to pseudo-random number generators. Using the new primitive trinomials, we can obtain uniform random number generators with extremely long period and good statistical properties in all dimensions less than 3021377.

Outline

- Definitions
- Connection with RNGs
- Sieving
- The standard algorithm
- The new algorithm
- Performance
- Some new primitive trinomials
- Application to RNGs

Introduction

Irreducible and primitive polynomials over finite fields have many applications in cryptography, coding theory, random number generation etc. See, for example, the books by Golomb, Knuth (Vol. 2), and Menezes *et al.*

In this talk, which describes joint work with Samuli Larvala (Helsinki University of Technology) and Paul Zimmermann (INRIA Lorraine), I will describe a new algorithm which has been used to find primitive polynomials of very high (in fact, "world record") degree over the field GF(2) of two elements.

Polynomials over GF(2)

GF(2) is just the set $\{0, 1\}$ with operations of addition and multiplication modulo 2.

Equivalently, GF(2) is the set of Boolean values $\{F, T\}$ with operations $\oplus$ (exclusive or) and & (and).

We consider polynomials over GF(2), that is, polynomials whose coefficients are in GF(2). *For the sake of brevity, we won't repeat this statement every time!*

Note that, for polynomials u, v over GF(2),

$$2u = 2v = 0 \ .$$

This implies that $u - v = u + v$ and

$$(u + v)^2 = u^2 + v^2 \ .$$

Some Definitions

We say that a polynomial $P(x)$ is *reducible* if it has nontrivial factors; otherwise it is *irreducible*.

If $P(x)$ is irreducible of degree $r > 1$, then

$$\mathrm{GF}(2^r) \approx Z_2[x]/(P(x)) \ ,$$

so we have a representation of the finite field GF(2^r) with 2^r elements. If x is generator for the multiplicative group of $Z_2[x]/(P(x))$, then we say that $P(x)$ is *primitive*.

Since the multiplicative group has order $2^r - 1$, we need to know the complete factorization of $2^r - 1$ in order to test if an irreducible polynomial is primitive. However, if r is a *Mersenne exponent*, i.e. $2^r - 1$ is prime, then irreducibility implies primitivity.

Random Number Generators

Pseudo-random number generators (RNGs) are widely used in simulation.

A program running on a fast computer or cluster of PCs might use 10^9 random numbers per second for many hours. Small correlations or other deficiencies could easily lead to spurious results.

In order to have confidence in the results of simulations, we need to have confidence in the statistical properties of the random numbers used.

There is no time to talk about random number generators in detail today, but I will mention some connections with primitive polynomials.

Generalized Fibonacci Generators

$$U_n = U_{n-r} \ \theta \ U_{n-s}$$

where r and s are fixed "lags" and θ is some binary operator. We always assume $0 < s < r$.

For example, the choice of $\theta = + \pmod{2^w}$ which we assume below is convenient on a binary machine. In this case the period is at most

$$2^{w-1}(2^r - 1) \ ,$$

and this is attained if $x^r + x^s + 1$ is a primitive polynomial over GF(2) and the initial values $U_0, \ldots, U_{r-1}$ are not all even.

The case of addition in GF(2), i.e. $w = 1$, $\theta = \oplus$, gives a *linear feedback shift register* (LFSR) generator which is easy to implement in hardware. These (and the case $w > 1$, $\theta = \oplus$) are also called Tausworthe generators.

The Generating Function

Suppose $x_n = x_{n-r} + x_{n-s}$ over any field F, e.g. $F = \mathrm{GF}(2)$. We define the *generating function*

$$G(t) = \sum_{n \geq 0} x_n t^n \ .$$

It is easy to see that

$$G(t) = \frac{G_0(t)}{P(t)} \ ,$$

where $G_0(t)$ is a polynomial of degree (at most) $r-1$ defined by the initial values $x_0, \ldots, x_{r-1}$, and

$$P(t) = 1 - t^s - t^r$$

is defined by the recurrence.

The generating function can be used to obtain various theoretical results (period, expected values, correlations, etc).

Some Well-Known Results

The following results can be found in texts such as Lidl, Menezes et al. Here μ is the Möbius function, and ϕ is Euler's phi function.

1. $x^{2^n} + x$ is the product of all irreducible polynomials of degree $d|n$. For example,

   $$x^8 + x = x(1+x)(1+x+x^3)(1+x^2+x^3) \ .$$

2. Let J_n be the number of irreducible polynomials of degree n. Then

   $$\sum_{d|n} dJ_d = 2^n \ \text{ and } \ J_n = \frac{1}{n} \sum_{d|n} 2^d \mu(n/d) \ .$$

   In particular, if n is prime then $J_n = (2^n - 2)/n$.

3. The number of primitive polynomials of degree n is $P_n = \phi(2^n - 1)/n \leq J_n$.

   In particular, if n is a Mersenne exponent, then $P_n = J_n = (2^n - 2)/n$.

The Reciprocal Polynomial

If $P(x) = \sum_{j=0}^{r} a_j x^j$ is a polynomial of degree r, with $a_0 \neq 0$, then

$$P_R(x) = x^r P(1/x) = \sum_{j=0}^{r} a_j x^{r-j}$$

is the *reciprocal polynomial*. Clearly $P(x)$ is irreducible (or primitive) iff $P_R(x)$ is irreducible (or primitive).

In particular, if

$$P(x) = 1 + x^s + x^r \ , \ \ 0 < s < r$$

is a trinomial, then the reciprocal trinomial is

$$P_R(x) = 1 + x^{r-s} + x^r \ .$$

If it is convenient, we can assume that $s \leq r/2$ (else consider the reciprocal trinomial).

Similarly, if r is odd, we can assume that s is odd (this will be useful later).

Searching for Irreducible Polynomials

The irreducible polynomials (over GF(2), as usual) of degree r are analogous to primes with r digits. When searching for large primes we can quickly eliminate most candidates by sieving out multiples of small primes. Similarly, when searching for irreducible polynomials, we can eliminate candidates by checking if they are divisible by irreducible polynomials of low degree.

Sieving

Given a candidate $P(x)$, we check if $\mathrm{GCD}(P(x), x^{2^n} + x) \neq 1$, for $n = 1, 2, 3, \ldots, N$ where $N < \deg(P)$ is some bound. Since all irreducible polynomials of degree n are divisors of $x^{2^n} + x$, $P(x)$ passes these checks iff it has no irreducible factors of degree $\leq N$.

If $2^n > r = \deg(P)$, we can save time and space by computing $x^{2^n} \bmod P(x)$ by repeated squaring and reduction, before computing the GCD.

Irreducible Trinomials

For applications such as random number generation, we want irreducible (or better, primitive) polynomials of high degree r and with a small number of nonzero terms. Hence, we restrict attention to *trinomials* of the form

$$P(x) = P_{r,s}(x) = 1 + x^s + x^r \ , \quad 0 < s < r \ .$$

For simplicity, assume r is an odd prime.

Swan's Theorem

Swan (1962) determines the parity of the number of irreducible factors by an argument involving the discriminant (actually, Swan's main result is due to Stickelberger (1897)).

If r is an odd prime, then Swan's theorem implies that $P_{r,s}(x)$ has an even number of irreducible factors (and hence is reducible) if $r = \pm 3 \bmod 8$ and $s \neq 2$ or $r-2$.

The condition on s can not be omitted, e.g. $x^{29} + x^2 + 1$ is irreducible.

Expectation of Success

The probability that a randomly chosen polynomial of degree r is irreducible is of order $1/r$. Empirically, it seems that the same holds for trinomials of prime degree $r = \pm 1 \bmod 8$ (this condition implies that Swan's theorem is not applicable).

Thus, if we consider all s in the range $0 < s < r/2$, we expect a small constant number c of irreducible trinomials of degree r. Empirical evidence suggests that $c \approx 3.2$.

For example, considering the 523 prime $r \in [1000, 10000]$ such that $r = \pm 1 \bmod 8$, we find exactly 1683 irreducible trinomials, giving an estimate $c = 3.22 \pm 0.08$.

Open Question. What is this constant c ?

Searching for Irreducible Trinomials

Suppose r is an odd prime, $r = \pm 1 \bmod 8$, and sieving has failed to show that $P(x) = P_{r,s}(x)$ is reducible. The *standard algorithm* computes

$$x^{2^r} \bmod P(x)$$

by r steps of squaring and reduction, then uses the result that $P(x)$ is irreducible iff

$$x^{2^r} = x \bmod P(x) \ .$$

All authors of papers which give tables of irreducible trinomials (Watson, Rodemich, Zierler, Kurita, Heringa, Kumada, ...) seem to have used essentially this algorithm, which is why we call it the *standard algorithm*.

Complexity of the Standard Algorithm

Since we are working over GF(2),

$$\left(\sum_j a_j x^j \right)^2 = \sum_j a_j x^{2j} \ .$$

Thus, each squaring step takes $O(r)$ operations.

Each reduction step also takes $O(r)$ operations, since $P(x)$ is a trinomial and we can apply

$$x^{j+r} = x^{j+s} + x^j \bmod P(x)$$

for $j = r-2, r-3, \ldots, 0$ to reduce the result of squaring to a polynomial of degree less than r.

Thus, the complete test for reducibility of $P_{r,s}$ takes $O(r^2)$ operations, and to test all s takes $O(r^3)$ operations (assuming that sieving leaves a constant fraction of trinomials to test).

If the sieve limit is N and $2^N \approx r^c$ for some $c < 1$, then $N \approx c \log_2 r$ and we expect $O(r/N)$ trinomials to survive sieving, so the overall complexity might be reduced from $O(r^3)$ to $O(r^3/\log r)$.

Improving the Standard Algorithm

The standard algorithm uses $2r$ bits of memory for squaring, and $2r + O(1)$ $\oplus$ operations for each reduction (we count bit-operations but in practice we perform 32 or 64 bit-operations in parallel using word-operations; this also applies to our new algorithm). Many of these operations are on bits which are necessarily zero. There is a better algorithm which avoids these redundant operations.

Both algorithms represent a polynomial $A(x) = \sum_{j=0}^{r-1} a_j x^j$ as a bit-vector $a_0 \dots a_{r-1}$.

Since r is odd and we can consider either $P_{r,s}$ or its reciprocal $P_{r,r-s}$, we can assume that s is odd.

The New Algorithm – Squaring

The first point is that there is no need to actually perform the squaring step! The standard algorithm would replace the bit vector

$$a_0 a_1 a_2 \dots a_{r-2} a_{r-1}$$

by its "square"

$$a_0 0 a_1 0 a_2 \dots 0 a_{r-2} 0 a_{r-1} .$$

However, we can simply keep the bit vector

$$a_0 a_1 a_2 \dots a_{r-2} a_{r-1}$$

and regard it as *implicitly* representing a square (in other words, we do not store the coefficients of odd-degree terms, since they are known to be zero).

The New Algorithm – Reduction

To see how the reduction can be performed after our "implicit squaring", consider the example $r = 7, s = 3$.
We initialise $A(x) \leftarrow x$, i.e. $a_0 \dots a_6 \leftarrow 0100000$.
The "squaring" operation is implicit: we keep the bit-vector 0100000 and regard this as representing

$$a_0 a_2 a_4 a_6 a_8 a_{10} a_{12}$$

We now reduce mod $P(x) = 1 + x^3 + x^7$.
Observe that $x^{12} = x^5 + x^8 \bmod P(x)$, so we should replace a_8 by $a_8 \oplus a_{12}$ and a_5 by $a_5 \oplus a_{12}$, but a_5 is currently zero, so we can simply regard the rightmost bit as representing a_5 rather than a_{12}. Thus, after the first step of the reduction we have a bit-vector representing

$$a_0 a_2 a_4 a_6 \underline{a_8} a_{10} a_5 .$$

The only bit(s) which could have changed, because they depend on the result of an $\oplus$ operation, are underlined.

Example of Reduction continued

Proceeding in a similar fashion, we observe that $x^{10} = x^3 + x^6 \bmod P(x)$, but $a_3 = 0$, so we replace a_6 by $a_6 \oplus a_{10}$ and implicitly regard the second bit from the right as representing a_3 rather than a_{10}. Thus, after the reduction we have a bit-vector representing

$$a_0 a_2 a_4 \underline{a_6} a_8 a_3 a_5 .$$

One more step of reduction gives a bit-vector representing

$$a_0 a_2 \underline{a_4} a_6 a_1 a_3 a_5 .$$

Observe that this bit-vector contains the coefficients of $A(x)^2 \bmod P(x)$, but they are in a shuffled order. We need to apply an *interleave* permutation to get back to the natural order

$$a_0 a_1 a_2 a_3 a_4 a_5 a_6 .$$

Interleaving

Interleaving is closely related to squaring. In fact, if we square $a_0a_2a_4a_6$:

$$a_0a_2a_4a_6 \to a_00a_20a_40a_60 ,$$

square and rightshift $a_1a_3a_50$:

$$a_1a_3a_50 \to 0a_10a_30a_500 ,$$

and apply a bitwise $\vee$ operation, we obtain

$$a_0a_1a_2a_3a_4a_5a_60 .$$

Thus, interleaving can be implemented by squaring and a few additional operations (shifting and $\vee$-ing). Although two squarings are necessary, the bit-vectors are only half as long as before, so the work involved is almost the same.

A Complete Example

Consider the example $r = 7$, $s = 3$. The k-th operation of (implicitly) squaring and reducing mod $P(x)$ is denoted by S_k, and the k-th operation of interleaving by I_k.

If we start with $A(x) = x$ and perform operations $S_1, I_1, S_2, I_2, \ldots, S_7, I_7$ we obtain the following:

$S_1 \to 0100000$, $\quad I_1 \to 0010000 \equiv x^2$
$S_2 \to 0010000$, $\quad I_2 \to 0000100 \equiv x^4$
$S_3 \to 0010100$, $\quad I_3 \to 0100100 \equiv x + x^4$
$S_4 \to 0110100$, $\quad I_4 \to 0110100 \equiv x + x^2 + x^4$
$S_5 \to 0100100$, $\quad I_5 \to 0110000 \equiv x + x^2$
$S_6 \to 0110000$, $\quad I_6 \to 0010100 \equiv x^2 + x^4$
$S_7 \to 0000100$, $\quad I_7 \to 0100000 \equiv x$

Since the final result is x, we deduce that $P(x) = 1 + x^3 + x^7$ is irreducible.

The New Algorithm

We now describe the new algorithm formally, in terms of bit-operations. As before, assume that r and s are odd.

To avoid confusion, we denote the working bit-array by $b_0b_1 \cdots b_{r-1}$. This bit-array is used to represent the coefficients $a_0a_1 \cdots a_{r-1}$ of the polynomial $A(x)$, but not necessarily in the natural order.

Let $\alpha = (r-1)/2$ and $\delta = (r-s)/2$. Since r and s are odd, α and δ are integers. Initially we set $b_1 \leftarrow 1$ and the other $b_j \leftarrow 0$ to represent $A(x) = x$.

Squaring and Reduction

Each step S_k is implemented by

for $j \leftarrow r-1$ downto $\alpha+1$ do
$\quad b_{j-\delta} \leftarrow b_{j-\delta} \oplus b_j$.

Squaring and reduction step S_k

Note that there are only $r/2 + O(1)$ "$\oplus$" bit-operations in the loop, which is a 75% reduction over the $2r + O(1)$ for the reduction step of the standard algorithm.

Interleaving

The obvious implementation of the interleaving step I_k requires a temporary bit-array (say $c_0 c_1 \cdots c_{r-1}$). For example:

$c_0 \leftarrow b_0$;
for $j \leftarrow 1$ to α do {forward interleave}
  begin
  $c_{2j-1} \leftarrow b_{j+\alpha}$;
  $c_{2j} \leftarrow b_j$;
  end;
for $j \leftarrow 0$ to $r-1$ do $b_j \leftarrow c_j$.

Forward interleave I_k with copy

We call this a "forward interleave" because the first loop index j increases.

We can avoid the final loop (copying the c array to b) by alternately using the array b and the array c (or by interchanging pointers appropriately). However, the space required is still $2r + O(1)$ bits, the same as for the standard algorithm.

A Refinement: Overlapping Arrays

We can interleave in the backward direction (replace "for $j \leftarrow 1$ to α" by "for $j \leftarrow \alpha$ downto 1" above). Suppose we also interchange the roles of b and c to avoid the final copy.

The point of interleaving alternately in the forward and backward directions is that we can save space by using a single working array of size $3r/2 + O(1)$ bits. The b and c arrays can partially overlap – in fact b_j can occupy the same memory as $c_{j+\alpha}$ ($j = 0, 1, \ldots$), as shown:

$$\boxed{b_0\ b_1\ \cdots\ b_\alpha\ \cdots\ b_{r-1}}$$

$$\boxed{c_0\ c_1\ \cdots\ c_\alpha\ \cdots\ c_{r-1}}$$

Note that the "forward interleave" transmits data from b to c (i.e. to the left) and the "backward interleave" transmits data from c to b (i.e. to the right)!

Effect of the Refinement

Partially overlapping the arrays b and c can improve performance dramatically on machines with memory hierarchies and cache sizes of less than $2r$ bits, because the working set is reduced in size by 25%. It has little effect on machines with much larger caches.

Generalization of the New Algorithm

If we replace $1 + x^s + x^r$ by

$$1 + x^{s_1} + \cdots + x^{s_k} + x^r \ ,$$

then an obvious generalisation of our new algorithm is applicable *provided* that r is odd and the s_i all have the same parity (all odd or all even).

Comparison of the Algorithms

The new algorithm has 75% fewer $\oplus$ operations than the standard algorithm.

Perhaps more significant than the number of operations is the number of memory references, which is reduced by 56%, from $8r/w + O(1)$ loads/stores to $\frac{7r}{2w} + O(1)$ loads/stores, on a machine with wordlength w bits.

Also significant on some machines is that the working set size is reduced by 25%, so memory references are more likely to be in the cache.

In practice the improvement provided by the new algorithm depends on many factors: the values of r and (to a lesser extent) s, the cache size, the compiler and compiler options used, whether inner loops are written in assembler, etc, but it is generally at least a factor of two.

Performance of the New Algorithm

Table 1 gives normalised times for the standard and new algorithms on various processors, for $r = 3021377$. The third column is the "normalised time" $c = \text{time(nsec)}/r^2$.

processor	algorithm	c
300 Mhz P-II	standard	6.31
"	new (no overlap)	2.60
"	new (overlap)	1.64
500 Mhz P-III	"	0.77
833 Mhz P-III	"	1.66
300 Mhz SGI R12000	"	1.16
667 Mhz DEC Alpha	"	0.60

Table 1: Normalised time to test reducibility

Note that $3r/2$ bits is 553KB. The L2 cache size was 512KB on the P-II and P-III machines *except* only 256KB for the 833 Mhz P-III. The program was written in C, except that on PCs the inner loops were written in assembler to use the 64-bit MMX registers.

Times for Various Degrees

In Table 2 we show the time for a full reducibility test with our new algorithm and various degrees r on a machine (300 Mhz Pentium P-II) with 512KB L2 cache.

r	time T (sec)	$c = 10^9 T/r^2$
19937	0.42	1.06
44497	2.10	1.06
110503	14.4	1.18
132049	21.7	1.24
756839	812	1.42
859433	1027	1.39
3021377	15010	1.64
6972593	198000	4.10

Table 2: Time to test reducibility on a P-II

New Primitive Trinomials

In Table 3 we give a table of primitive trinomials $x^r + x^s + 1$ where r is a Mersenne exponent (i.e. $2^r - 1$ is prime). We assume that $0 < 2s \le r$ (so $x^r + x^{r-s} + 1$ is not listed).

Results for $r < 756839$ are given by Heringa *et al.* [8]. We have confirmed these results.

The entries for $r < 3021377$ have been checked by running at least two different programs on different machines.

During this checking process, the entry with

$$r = 859433, \quad s = 170340$$

was found. This was surprising, because Kumada *et al.* [10] claimed to have searched the whole range for $r = 859433$. It turns out that Kumada *et al.* missed this entry because of a bug in the sieving routine!

New Primitive Trinomials cont.

The three entries for $r = 756839$ are new (Kumada *et al.* did not search for this r), as are the two entries for $r = 3021377$.

r	s	Notes
756839	215747	BLZ, 14 June 2000
	267428	BLZ, 11 June 2000
	279695	BLZ, 9 June 2000
859433	170340	BLZ, 26 June 2000
	288477	Kumada *et al.* [10]
3021377	361604	BLZ, 8 August 2000
	1010202	BLZ, 17 Dec 2000

Table 3: Primitive trinomials

The new entries are from Brent, Larvala and Zimmermann (BLZ) [4].

New Irreducible Trinomials

There is a large gap between some of the Mersenne exponents r for which primitive trinomials exist. For example, there are none in the interval $132049 < r < 756839$. In Table 4 we give some irreducible trinomials to fill this gap. As usual, we only list $s \leq r/2$. The exponents r were chosen to be close to the arithmetic progression $10^5, 2 \times 10^5, 3 \times 10^5, \ldots$ with the constraints that:

1. r is prime.
2. $r = \pm 1 \bmod 8$.
3. $2^r - 1$ is composite, but no prime factors of $2^r - 1$ are known. These factors are certainly larger than 2^{32} (see GIMPS [6]).

New Irreducible Trinomials cont.

Because of the constraints, we can be sure that the trinomials listed are irreducible. They are extremely likely to be primitive, but we can not prove this without knowing the complete factorisation of $2^r - 1$.

The search is incomplete for $r \geq 600071$.

r	s
100151	4764, 15503
200033	10175, 55224, 95397, 96236, 97575, 98763
300151	49950, 87430
400033	17865, 103623
500231	4862, 10101, 203207, 205310
600071	111503
700057	24829, 121384
800057	92487
900217	82555

Table 4: Some irreducible trinomials

Another Way

There is another way to generate irreducible polynomials of high degree. It is known [Golomb, §6.9] that if

$$P(x) = \sum_{j=0}^{r} a_j x^j$$

is primitive (of degree r) then

$$Q(x) = \sum_{j=0}^{r} a_j x^{2^j - 1}$$

is irreducible (of degree $2^r - 1$).
For example, $P(x) = 1 + x^3 + x^{20}$ is primitive, which implies that

$$Q(x) = 1 + x^7 + x^{1048575}$$

is irreducible. To determine if $Q(x)$ is primitive we have to completely factorise $2^{1048575} - 1$, which is probably difficult.

A Very Large Irreducible Trinomial

In the same way, $P(x) = 1 + x^{361604} + x^{3021377}$ is primitive, so

$$Q(x) = 1 + x^{2^{361604}-1} + x^{2^{3021377}-1}$$

is irreducible, and extremely likely to be primitive, but to prove this we have to factor

$$2^{2^{3021377}-1} - 1$$

which seems infeasible, since the *smallest* prime factor must be larger than the Mersenne prime exponent $2^{3021377} - 1$.

Some Random Number Generators

Although it is easy to suggest bad random number generators, there is no "best" generator. Here are a few possibilities for generalized Fibonacci random number generators based on primitive trinomials $1 + x^s + x^r$. The memory requirements are stated on the assumption that each full-word random number takes eight bytes.

1. Combine a "small" generator using (say) $(r, s) = (3217, 576)$, and a "large" generator with $(r, s) = (3021377, 361604)$. Assume the small generator gives full words, and the large generator gives single bits (though they can be generated a word at a time). If the generators are combined by Coppersmith's shrinking method, the period is

$$2^{3021376}(2^{3217} - 1)$$

and the memory requirement is 394KB.

Some RNGs continued

2. Combine $(r, s) = (1279, 418)$, $(2281, 1029)$ by addition.

Period $> 10^{1000}$. Fast, uses only 28KB memory. The numbers satisfy a 9-term recurrence so we expect good statistical properties.

3. Combine $(r, s) = (127, 30)$, $(521, 158)$, and $(607, 273)$ by addition (and perhaps shuffling).

Period $> 10^{377}$. Uses only 10KB memory, so should run in cache. 27-term recurrence (if combined by addition), so expect excellent statistical properties.

4. Combine the last two generators in 3 (restricted to single bits) by shrinking to obtain a cryptographically strong generator with memory less than 160 bytes (excluding the program).

Last Words

Large primitive trinomials are analogous to large Mersenne primes – there is an interest in finding them just to prove that it can be done.

We can not *prove* that arbitrarily large Mersenne primes exist, though this is conjectured to be true. Similarly, we can not *prove* that primitive trinomials of arbitrarily high degree exist, though empirical evidence suggests that the set of degrees for which they exist has positive density.

We did not really *need* to find primitive trinomials of degree 3021377 in order to obtain good random number generators, but the application to random number generators is a good justification for the time spent on the search.

References

[1] S. L. Anderson, Random number generators on vector supercomputers and other advanced architectures, *SIAM Rev.* **32** (1990), 221–251.

[2] R. P. Brent, On the periods of generalized Fibonacci recurrences, *Math. Comp.* **63** (1994), 389–401. `http://www.comlab.ox.ac.uk/oucl/work/richard.brent/pub/pub133.html`

[3] R. P. Brent, Random number generation and simulation on vector and parallel computers, *Proc. Fourth Euro-Par Conference, LNCS* **1470**, Springer-Verlag, Berlin, 1998, 1–20. `···/pub185.html`

[4] R. P. Brent, S. Larvala and P. Zimmermann, *A fast algorithm for testing irreducibility of trinomials mod 2* (preliminary report), Oxford University Computing Laboratory, Report PRG-TR-13-00, December 2000. `···/pub199.html`

[5] D. Coppersmith, H. Krawczyk and Y. Mansour, The shrinking generator, *Proc. CRYPTO'93, LNCS* **773** (1994), 22–39.

[6] GIMPS, The Great Internet Prime Search, `http://www.mersenne.org/`

[7] S. W. Golomb, *Shift register sequences*, Aegean Park Press, revised edition, 1982.

[8] J. R. Heringa, H. W. J. Blöte and A. Compagner. New primitive trinomials of Mersenne-exponent degrees for random-number generation, *International J. of Modern Physics C* **3** (1992), 561–564.

[9] D. E. Knuth, *The art of computer programming, Volume 2: Seminumerical algorithms* (third ed.), Addison-Wesley, Menlo Park, CA, 1998.

[10] T. Kumada, H. Leeb, Y. Kurita and M. Matsumoto, New primitive t-nomials ($t = 3, 5$) over GF(2) whose degree is a Mersenne exponent, *Math. Comp.* **69** (2000), 811–814.

[11] Y. Kurita and M. Matsumoto, Primitive t-nomials ($t = 3, 5$) over GF(2) whose degree is a Mersenne exponent ≤ 44497, *Math. Comp.* **56** (1991), 817–821.

[12] R. Lidl and H. Niederreiter, *Introduction to Finite Fields and their Applications*, Cambridge Univ. Press, Cambridge, second edition, 1994.

[13] G. Marsaglia, A current view of random number generators, in *Computer Science and Statistics: The Interface* (edited by L. Billard), Elsevier Science Publishers B. V. (North-Holland), 1985, 3–10.

[14] M. Matsumoto and T. Nishimura, Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator, *ACM Trans. on Modeling and Computer Simulations*, 1998.

[15] A. J. Menezes, P. C. van Oorschot and S. A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, New York, 1997. `http://cacr.math.uwaterloo.ca/hac/`

[16] R. G. Swan, Factorization of polynomials over finite fields, *Pacific J. Math.* **12** (1962), 1099–1106.

[17] S. Tezuka, Efficient and portable combined Tausworthe random number generators, *ACM Trans. on Modeling and Computer Simulation* **1** (1991), 99–112.

[18] I. Vattulainen, T. Ala-Nissila and K. Kankaala, Physical tests for random numbers in simulations, *Phys. Rev. Lett.* **73** (1994), 2513–2516.

[19] N. Zierler and J. Brillhart, On primitive trinomials (mod 2), *Inform. and Control* **13** (1968), 541–554 and **14** (1969), 566–569.

[20] N. Zierler, Primitive trinomials whose degree is a Mersenne exponent, *Inform. and Control* **15** (1969), 67–69.