

Efficient polynomial time algorithms computing industrial-strength primitive roots

Jacques Dubrois

Axalto, 50 Avenue Jean-Jaurès, B.P. 620-12 92542 Montrouge, France.

Jean-Guillaume Dumas

*Université Joseph Fourier, Laboratoire de Modélisation et Calcul, 50 av. des
Mathématiques. B.P. 53 X, 38041 Grenoble, France.*

Abstract

E. Bach, following an idea of T. Itoh, has shown how to build a small set of numbers modulo a prime p such that at least one element of this set is a generator of $\mathbb{Z}/p\mathbb{Z}$. E. Bach suggests also that at least half of his set should be generators. We show here that a slight variant of this set can indeed be made to contain a ratio of primitive roots as close to 1 as necessary. In particular we present an asymptotically $O\left(\sqrt{\frac{1}{\epsilon}}\log(p) + \log^2(p)\right)$ algorithm providing primitive roots of p with probability of correctness greater than $1 - \epsilon$ and several $O(\log^\alpha(p))$, $\alpha \leq 5.23$, algorithms computing "Industrial-strength" primitive roots.

Key words: computational complexity, cryptography, randomized algorithms

Email addresses: jdubrois@axalto.com (Jacques Dubrois),
Jean-Guillaume.Dumas@imag.fr (Jean-Guillaume Dumas).

1 Introduction

Primitive roots are generators of the multiplicative group of the invertibles of a finite field. We focus in this paper only on prime finite fields, but the proposed algorithms can work over extension fields or other multiplicative groups. Primitive roots are of intrinsic use e.g. for secret key exchange (Diffie-Hellman), pseudo random generators (Blum-Micali) or primality certification. The classical method of generation of such generators is by trial, test and error. Indeed within a prime field with p elements they are quite numerous ($\phi(\phi(p)) = \phi(p-1)$) among $p-1$ invertibles are generators. The problem resides in the test to decide whether a number g is a generator or not. The first idea is to test every g^i for $i = 1..p-1$ looking for matches. Unfortunately this is exponential in the size of p . An acceleration is then to factor $p-1$ and test whether one of the $g^{\frac{p-1}{q}}$ is 1 for q a divisor of $p-1$. If this is the case then g is obviously not a generator. On the contrary, one has proved that the only possible order of g is $p-1$. Unfortunately again, factorization is still not a polynomial time process: no polynomial time algorithm computing primitive roots is known. However, there exists polynomial time methods isolating a polynomial size set of numbers containing at least one primitive root. Elliot and Murata [3] also gave polynomial lower bounds on the least primitive root modulo p . One can also generate elements with exponentially large order even though not being primitive roots [6]. Our method is in between those two approaches.

As reported by Bach [1], Itoh's breakthrough was to use only a partial factorization of $p-1$ to produce primitive roots with high probability [4]. Bach then used this idea of partial factorization to give the actually smallest known set, deterministically containing one primitive root[1], if the Extended Riemann Hypothesis is true. Moreover, he suggested that his set contained at least half

primitive roots. In this paper, we propose to use a combination of Itoh’s and Bach’s algorithms producing a polynomial time algorithm generating primitive roots with a very small probability of failure (without the ERH). Such generated numbers will be denoted by “Industrial-strength” primitive roots. We also have a guaranteed lower bound on the order of the produced elements. In this paper, we analyze the actual ratio of primitive roots within a variant of Bach’s full set. As this ratio is close to 1, both in theory and even more in practice, selecting a random element within this set produces a fast and effective method computing primitive roots. We present in section 2 our algorithm and the main theorem counting this ratio. Then practical implementation details and effective ratios are discussed section 3.

2 The variant of Itoh/Bach’s algorithm

The salient features of our approach when compared to Bach’s are that:

- (1) We partially factor, but *with known lower bound on the remaining factors*.
- (2) *We do not require the primality* of the chosen elements.
- (3) *Random elements are drawn from the whole set of candidates instead of only from the first ones.*

Now, when compared to Itoh’s method, we use a deterministic process producing a number with a very high order and which has a high probability of being primitive. On the contrary, Itoh selects a random element but uses a polynomial process to prove that this number is a primitive root with high probability [4]. The difference here is that we use low order terms to build higher order elements whereas Itoh discards the randomly chosen candidates and restarts all over at each failure. Therefore we first compute the ratio of

Algorithm 1: Probabilistic Primitive Root

Input: A prime $p \geq 3$ and a failure probability $0 < \epsilon < 1$.

Output: A number, primitive root with probability greater than $1 - \epsilon$.

begin

 Compute B such that $(1 + \frac{2}{p-1})(1 - \frac{1}{B})^{\log_B \frac{p-1}{2}} = 1 - \epsilon$.

 Partially factor $p - 1 = 2^{e_1} p_2^{e_2} \dots p_h^{e_h} Q$ ($p_i < B$ and Q has no factor $< B$).

for each $1 \leq i \leq h$ **do**

 By trial and error, randomly choose α_i verifying: $\alpha_i^{\frac{p-1}{p_i}} \not\equiv 1 \pmod{p}$.

 Set $a \equiv \prod_{i=1}^h \alpha_i^{\frac{p-1}{p_i}} \pmod{p}$.

if *Factorization is complete* **then**

 Set Probability of correctness to 1 and **return** a .

else

 Refine Probability of correctness to $(1 + \frac{1}{Q-1})(1 - \frac{1}{B})^{\log_B Q}$.

 Randomly choose b verifying: $b^{\frac{p-1}{Q}} \not\equiv 1$ and **return** $g \equiv ab^{\frac{p-1}{Q}} \pmod{p}$.

end

primitive roots within the set. We have found afterwards that Itoh, independently and differently, proves quite the same within his [4, Theorem 1].

Theorem 1 *At least $\frac{\phi(Q)}{Q-1}$ of the returned values of Algorithm 1 are primitive roots.*

PROOF. We let $p - 1 = kQ$. In algorithm 1, the order of a is $(p - 1)/Q = k$ (see [1]). We partition Z/pZ^* by S and T where

$$S = \{b \in Z/pZ^* : b^k \not\equiv 1 \pmod{p}\} \quad \text{and} \quad T = \{b \in Z/pZ^* : b^k \equiv 1 \pmod{p}\}$$

and let $U = \{b \in Z/pZ^* : b^k \text{ has order } Q\}$. Note that for any $x \in Z/pZ^*$ of order n and any $y \in Z/pZ^*$ of order m , if $\gcd(n, m) = 1$ then the order of $z \equiv xy \pmod{p}$ is nm . Thus for any $b \in U$ it follows that $g \equiv ab^k \pmod{p}$ has

order $p - 1$. Since $U \subseteq S$, we have that $\frac{|U|}{|S|}$ of the returned values of algorithm 1 are primitive roots.

We thus now count the number of elements of U and S . On the one hand, we fix arbitrarily a primitive root $\tilde{g} \in Z/pZ^*$ and define $E = \{i : 0 \leq i \leq Q \text{ and } \gcd(i, Q) = 1\}$. $|E| = \varphi(Q)$ and it is not difficult to see that $U = \{\tilde{g}^{i+jQ} : i \in E \text{ and } 0 \leq j \leq k - 1\}$. This implies that $|U| = k\varphi(Q)$. On the other hand, we have $T = \{\tilde{g}^0, \tilde{g}^Q, \dots, \tilde{g}^{(k-1)Q}\}$. The partitioning thus gives $|S| = |Z/pZ^*| - |T| = p - 1 - k$. We thus conclude that $\frac{|U|}{|S|} = \frac{k\varphi(Q)}{p-1-k} = \frac{\varphi(Q)}{Q-1}$.

Corollary 2 *Algorithm 1 is correct and, when Pollard's rho algorithm is used, has an average running time of $O\left(\sqrt{\frac{1}{\varepsilon}} \log^2(p) + \log^3(p) \log(\log(p))\right)^*$.*

PROOF. There remains to show that $\frac{\phi(Q)}{Q-1} > 1 - \varepsilon$. Let $Q = \prod_{i=1}^{\omega(Q)} q_i^{f_i}$ where $\omega(Q)$ is the number of distinct prime factors of Q . Then $\phi(Q) = \prod_{i=1}^{\omega(Q)} \phi(q_i^{f_i}) = Q \prod_{i=1}^{\omega(Q)} \left(1 - \frac{1}{q_i}\right)$. Thus $\frac{\phi(Q)}{Q-1} = \left(1 + \frac{1}{Q-1}\right) \prod_{i=1}^{\omega(Q)} \left(1 - \frac{1}{q_i}\right)$. Now, since any factor of Q is bigger than B , we have: $\prod_{i=1}^{\omega(Q)} \left(1 - \frac{1}{q_i}\right) > \prod_{i=1}^{\omega(Q)} \left(1 - \frac{1}{B}\right) = \left(1 - \frac{1}{B}\right)^{\omega(Q)}$. To conclude, we minor $\omega(Q)$ by $\log_B(Q)$. This gives the probability refinement[†]. Since Q is not known at the beginning, one can minor it there by $\frac{p-1}{2}$ since $p - 1$ must be even whenever $p \geq 3$. Now for the complexity. For the computation of B , we use a Newton-Raphson's approximation. The second step depends on the factorization method. Both complexities here are given by the application of Pollard's rho algorithm. Indeed Pollard's rho would require at worst $L = 2[B]$ loops and $L = O(\sqrt{B})$ on the average thanks to the birthday paradox.

* Using fast integer arithmetic this can become :

$$O\left(\sqrt{\frac{1}{\varepsilon}} \log(p) \log^2(\log(p)) \log(\log(\log(p))) + \log^2(p) \log^2(\log(p)) \log(\log(\log(p)))\right) ;$$

but the worst case complexity is $O\left(\frac{1}{\varepsilon} \log^2(p) + \log^4(p) \log(\log(p))\right)$.

† Note that one can dynamically refine B as more factors of $p - 1$ are known.

Now each loop of Pollard's rho is a squaring and a gcd, both of complexity $O(\log^2 p)$. We conclude by the fact that $(1 + \frac{2}{p-1})(1 - \frac{1}{B})^{\log_B \frac{p-1}{2}} = 1 - \epsilon$ so that $B \leq \frac{1}{\epsilon}^\ddagger$. For the remaining steps, there is at worst $\log p$ distinct factors, thus $\log p$ distinct α_i , but only $\log \log p$ on the average. Each one requires a modular exponentiation which can be performed with $O(\log^3 p)$ operations using recursive squaring. Now, to get a correct α_i , at most $O(\log \log p)$ trials should be necessary. However, by an argument similar to that of theorem 1, less than $1 - \frac{1}{p_i}$ of the α_i are such that $\alpha_i^{\frac{p-1}{p_i}} \equiv 1$. This gives an average number of trials of $1 + \frac{1}{p_i}$, which is bounded by a constant. This gives $\log \times \log^3 \times \log \log$ in the worst case (distinct factors \times exponentiation \times number of trials) and only $\log \log \times \log^3 \times 2$ on the average.

3 Industrial-strength primitive roots

Of course, the only problem with this algorithm is that it is not polynomial. Indeed the partial factorization up to factors of any given size is still exponential. This gives the non polynomial factor $\sqrt{\frac{1}{\epsilon}}$. Other factoring algorithms with better complexity could also be used, provided they can guarantee a bound on the unfound factors. For that reason, we propose another algorithm with an attainable number of loops for the partial factorization. Therefore, the algorithm is efficient and we provide experimental data showing that it also has a very good behavior with respect to the probabilities: *Heuristic 2: Apply Algorithm*

[‡] We let $1 - \epsilon'_p = (1 - \epsilon)/(1 + \frac{2}{p-1})$ and $y = \omega(Q)$. With this we can solve for B and find $B\epsilon'_p = \epsilon'_p/(1 - (1 - \epsilon'_p)^{1/y})$. The latter is easily shown increasing for $0 < \epsilon'_p < 1$ as soon as $y \geq 1$. It is thus bounded by its value at $\epsilon'_p = 1$, which is 1. Therefore, $B \leq \frac{1}{\epsilon'_p} < \frac{1}{\epsilon}$ independently of p and Q .

1 with $B \leq \log^2(p) \log^2(\log(p))$. With Pollard's rho factoring, the algorithm has now an average bit polynomial complexity of : $O(\log^3(p) \log(\log(p)))$ (just replace B by $\log^2(p) \log^2(\log(p))$ and use $L = \sqrt{B}$). In practice, L could be chosen not higher than a million: in figures 1 we choose Q with known factorization and compute $\frac{\phi(Q)}{Q-1}$; the experimental data then shows that in practice

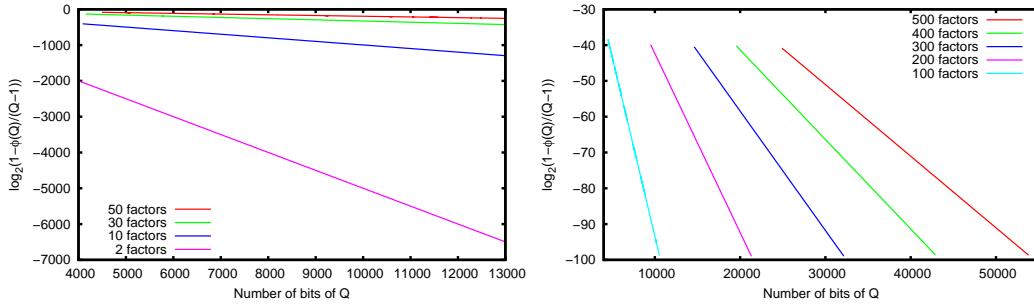


Fig. 1. Actual probability of failure (powers of 2) with $L = 2^{20}$

no probability less than $1 - 2^{-40}$ is possible even with L as small as 2^{20} .

Provided that one is ready to accept a fixed probability, further improvements on the asymptotic complexity can be made. Indeed, D. Knuth said *"For the probability less than $(\frac{1}{4})^{25}$ that such a 25-times-in-row procedures gives the wrong information about n . It's much more likely that our computer has dropped a bit in its calculations, due to hardware malfunctions or cosmic radiations, than that algorithm P has repeatedly guessed wrong."*[§] We thus provide a version of our algorithm guaranteeing that the probability of incorrect answer is lower than 2^{-40} : *Algorithm 3: If p is small ($p < 44905100$), factor $p-1$ completely, otherwise apply Algorithm 1 with $B = \log^{5.231534} p$.* With Pollard's rho factoring, the average asymptotic bit complexity is then $O(\log^{4.615767} p)$: Factoring numbers lower than 44905100, takes constant time. Now for larger primes and $B = \log^\alpha(p)$, we just remark that $(1 + \frac{2}{p-1})(1 - \frac{1}{B})^{\log_B \frac{p-1}{2}}$ is in-

[§] More precisely, cosmic rays only can be responsible for 10^5 software errors in 10^9 chip-hours at sea level[5] . At 1GHz, this makes 1 error every 2^{55} computations.

creasing in p , so that it is bounded by its first value. Numerical approximation of α so that the latter is $1 - 2^{-40}$ gives 5.231534. The complexity exponent follows as it is $2 + \frac{\alpha}{2}$. One can also apply the same arguments e.g. for a probability $1 - 2^{-55}$ and factoring all primes $p < 2^{512}$ (since 513-bit numbers are nowadays factorizable), then slightly degrading the complexity to $O(\log^{5.229768} p)$. We have thus proved that a probability of at least $1 - 2^{-40}$ can always be guaranteed. In other words, our algorithm is able to efficiently produce “industrial-strength” primitive roots.

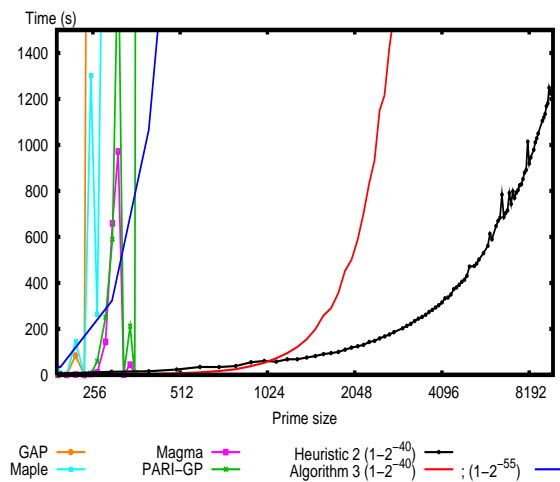


Fig. 2. Generations of primitive roots

always factoring $p-1$ completely. Still we can see the huge progress in primitive root generation that our algorithm has enabled.

4 Conclusion

We provide here a new very fast and efficient algorithm generating primitive roots. On the one hand, the algorithm has a polynomial time bit complexity when all existing algorithms were exponential. This is for instance illustrated when comparing it to existing software on figure 2. On the other hand, our

¶ swox.com/gmp, maplesoft.com, pari.math.u-bordeaux.fr, gap-system.org,

This is for instance illustrated when comparing our algorithm, implemented in C++ with GMP, to existing software (Maple 9.5, Pari-GP, GAP 4r4 and Magma 2.11) ¶ on an Intel PIV 2.4GHz. This comparison is shown on figure 2. Of course, the comparison

is not fair as other softwares are

algorithm is probabilistic in the sense that the answer might not be a primitive root. We have seen in this paper however, that the chances that an incorrect answer is given are less important than say “hardware malfunctions”. For this reason, we call our answers “Industrial-strength” primitive roots.

An application of our algorithm is a new primality test: *attempt to construct a primitive root for a number n with algorithm 1 ; if n is prime, the algorithm will succeed, otherwise the $\alpha_i^{\frac{p-1}{p_i}}$ will be 1 too often [2].* When a given probability of success is required, this algorithm can be competitive with repeated applications of Miller-Rabin’s test and allows to quantify the information gained by finding elements of large order.

References

- [1] E. Bach, Comments on search procedures for primitive roots, *Mathematics of Computation* 66 (220) (1997) 1719–1727.
- [2] J. Dubrois and J-G. Dumas. Polynomial time algorithms computing industrial-strength primitive roots. Tech. Rep. arXiv cs.SC/0409029 (2004).
- [3] P. D. T. A. Elliott and L. Murata, On the average of the least primitive root modulo p , *Journal of The london Mathematical Society* 56 (2) (1997) 435–454.
- [4] T. Itoh and S. Tsujii, How to generate a primitive root modulo a prime, *IPSIJ SIG Technical Report* 1989-AL-9-2 (1989).
- [5] T. O’Gorman, J. Ross, A. Taber, J. Ziegler, H. Muhlfeld, C. Montrose, H. Curtis, and J. Walsh. Field testing for cosmic ray soft errors in semiconductor memories. *IBM Journal of Research and Development*, 40(1):41–50, January 1996.
- [6] J. v. Gathen and I. Shparlinski, Orders of Gauß periods in finite fields, *Applicable Algebra in Engineering, Communication and Computing* 9 (1998) 15–24.