Electronic Colloquium on Computational Complexity, Report No. 207 (2015)

Finding Primitive Roots Pseudo-Deterministically

Ofer Grossman*

December 22, 2015

Abstract

Pseudo-deterministic algorithms are randomized search algorithms which output unique solutions (i.e., with high probability they output the same solution on each execution). We present a pseudo-deterministic algorithm that, given a prime p, finds a primitive root modulo p in time $\exp(O(\sqrt{\log p \log\log p}))$. This improves upon the previous best known provable deterministic (and pseudo-deterministic) algorithm which runs in exponential time $p^{\frac{1}{4}+o(1)}$. Our algorithm matches the problem's best known running time for Las Vegas algorithms which may output different primitive roots in different executions.

When the factorization of $p-1$ is known, as may be the case when generating primes with $p-1$ in factored form for use in certain applications, we present a pseudo-deterministic *polynomial* time algorithm for the case that each prime factor of $p-1$ is either of size at most $\log^c(p)$ or at least $p^{1/c}$ for some constant $c > 0$. This is a significant improvement over a result of Gat and Goldwasser [5], which described a polynomial time pseudo-deterministic algorithm when the factorization of $p-1$ was of the form kq for prime q and $k = \text{poly}(\log p)$.

We remark that the Generalized Riemann Hypothesis (GRH) implies that the smallest primitive root g satisfies $g \leq O(\log^6(p))$. Therefore, assuming GRH, given the factorization of $p-1$, the smallest primitive root can be found and verified deterministically by brute force in polynomial time.

1 Introduction

Pseudo-deterministic algorithms are randomized search algorithms which, with high probability, output the same solution on each execution. Formally, A is a pseudo-deterministic algorithm for a binary relation R if there exists some function s such that when executed on input x, the algorithm A outputs $s(x)$ with high probability, and $(x, s(x)) \in R$. In other words, when we execute A on input x, we get the same output $s(x)$ for almost all random seeds. Standard randomized search algorithms, on the other hand, may output a different y satisfying $(x, y) \in R$ on each execution with input x.

In [5], Gat and Goldwasser ask whether there exists a pseudo-deterministic algorithm that finds a primitive root mod p faster than the best known deterministic algorithm, which runs in time $p^{\frac{1}{4}+o(1)}$. We answer this question in the affirmative:

Theorem 1.1. *There exists a pseudo-deterministic algorithm for* Primitive-Root *that runs in expected time* $L_p(1/2) = \exp(O(\sqrt{\log p \log\log p}))$.

*ogrossma@mit.edu. Department of Mathematics, MIT.

ISSN 1433-8092

We note that this matches the time bound for the best known Las Vegas algorithms for Primitive-Root.

This problem may have cryptographic applications, as protocols based on the Diffie-Hellman problem [4] rely on primitive roots to establish keys. It may be desirable for two parties to independently generate the same key, or primitive root, for $\mathbb{F}_p$. In this situation, pseudo-deterministic algorithms are helpful while standard randomized algorithms will not suffice.

A closely related problem to Primitive-Root is Primitive-Root-Given-Factorization. This problem asks for a primitive root mod p, given both p and the factorization of $p-1$.

Primitive-Root-Given-Factorization may be relevant to applications since the factorization of $p-1$ is often known. For example, protocols may require efficient ways to verify that an element is a primitive root, in which case the factorization of $p-1$ will be known. For such applications, it is possible to efficiently generate primes p with $p-1$ in factored form [1].

Assuming the generalized Riemann Hypothesis (GRH), Shoup proved in [7] that the smallest non-residue mod p is of size $O(\log^6(p))$, which implies a brute force polynomial time algorithm for Primitive-Root-Given-Factorization. Without the GRH assumption, the best deterministic algorithm remains the $p^{\frac{1}{4}+o(1)}$ algorithm from [2].

In [5], polynomial time pseudo-deterministic algorithms are presented for Primitive-Root-Given-Factorization when the input prime satisfies $p-1=kq$, with q prime and $k=\mathrm{poly}(\log p)$. We improve upon this result by finding polynomial time pseudo-deterministic algorithms for primes satisfying $p-1=\prod_{i=1}^{k} q_i^{e_i}$, where for some constant c each of the q_i is either at most of size $\log^c(p)$ or at least of size $p^{1/c}$ (our dependence on c is exponential). It remains open to find a polynomial time pseudo-deterministic algorithm for Primitive-Root-Given-Factorization for general primes.

2 Preliminaries

In this section we establish some lemmas we will later use. All lemmas in this section assume p is a prime, $a,b \not\equiv 0 \bmod p$, and ord refers to the order in $\mathbb{F}_p^\times$ (the multiplicative group of $\mathbb{F}_p$).

Lemma 2.1. *Suppose $a,b \in \mathbb{F}_p^\times$. If $\mathrm{ord}(a)$ and $\mathrm{ord}(b)$ are relatively prime, then $\mathrm{ord}(ab)=\mathrm{ord}(a)\mathrm{ord}(b)$.*

Proof. First, we note that $(ab)^{\mathrm{ord}(a)\mathrm{ord}(b)}=1$. Therefore, $\mathrm{ord}(ab)|\mathrm{ord}(a)\mathrm{ord}(b)$.

Suppose $\mathrm{ord}(ab)<\mathrm{ord}(a)\mathrm{ord}(b)$. Let q be a prime dividing $\frac{\mathrm{ord}(a)\mathrm{ord}(b)}{\mathrm{ord}(ab)}$.

We know that $(ab)^{\mathrm{ord}(a)\mathrm{ord}(b)/q}=1$. However, q divides either $\mathrm{ord}(a)$ or $\mathrm{ord}(b)$. Suppose without loss of generality that $q|\mathrm{ord}(a)$. Then

$$1=a^{\mathrm{ord}(a)\mathrm{ord}(b)/q}\left(b^{\mathrm{ord}(b)}\right)^{(\mathrm{ord}(a)/q)}=a^{\mathrm{ord}(a)\mathrm{ord}(b)/q}.$$

Therefore, $\mathrm{ord}(a)|(\mathrm{ord}(a)/q)\mathrm{ord}(b)$. However, because $\mathrm{ord}(a)$ and $\mathrm{ord}(b)$ are relatively prime, this implies $\mathrm{ord}(a)|(\mathrm{ord}(a)/q)$, which is impossible. □

Definition 2.2 (qth residue)**.** Let $q|p-1$ be a prime. We call an element a which is a qth power (i.e., there exists some b such that $a=b^q$) a *qth residue*. Otherwise, we call a a *qth non-residue.*

Lemma 2.3. *Suppose q^e is the largest power of q dividing $p-1$. Then a qth non-residue has order divisible by q^e.*

Proof. Suppose g is a primitive root mod p. An element $a = g^k$ satisfies

$$\operatorname{ord}(a) = \frac{p-1}{\gcd(p-1,k)}.$$

If a is a qth non-residue, then we know k is not divisible by q. Therefore, $q \nmid \gcd(p-1,k)$. It follows that $\operatorname{ord}(a)$ is divisible by q^e, where q^e is the largest power of q dividing $p-1$. $\square$

The following lemma will show that to find a primitive root modulo p, it is enough if for each prime q_i dividing $p-1$ we find a q_ith non-residue.

Lemma 2.4. *Let $p - 1 = \prod_{i=1}^m q_i^{e_i}$. Suppose that for each i, the element a_i is a q_ith non-residue. Then the product*

$$\prod_{i=1}^m a_i^{(p-1)/q_i^{e_i}}$$

is a primitive root.

Proof. We can write $a_i = g^{k_i}$ for some primitive root g, and k_i not divisible by q_i. Then $a_i^{(p-1)/q_i^{e_i}} = g^{k_i(p-1)/q_i^{e_i}}$ must have order exactly $q_i^{e_i}$, since $q_i^{e_i}$ is the smallest number N such that $Nk_i(p-1)/q_i^{e_i}$ is divisible by $p-1$, which is the order of g.

Therefore, the element $a_i^{(p-1)/q_i^{e_i}}$ has order exactly $q_i^{e_i}$. It follows that the orders of each of the $a_i^{(p-1)/q_i^{e_i}}$ are relatively prime, and so by Lemma 2.1,

$$\operatorname{ord}\left(\prod_{i=1}^m a_i^{(p-1)/q_i^{e_i}}\right) = \prod_{i=1}^m \operatorname{ord}\left(a_i^{(p-1)/q_i^{e_i}}\right).$$

The order of $a_i^{(p-1)/q_i^{e_i}}$ is $q_i^{e_i}$, so the product of the orders is $\prod_{i=1}^m q_i^{e_i} = p-1$. Hence $\prod_{i=1}^m a_i^{(p-1)/q_i^{e_i}}$ is a primitive root. $\square$

Lemma 2.5. *Given p and $q|p-1$, there exists a pseudo-deterministic algorithm that finds a qth non-residue in time $q \cdot \operatorname{poly}(\log p)$.*

Proof. See Theorem 3 in [5]. $\square$

Lemma 2.6. *Given the factorization $p - 1 = \prod_{i=1}^m q_i^{e_i}$ and an element $a \in \mathbb{F}_p$, we can compute $\operatorname{ord}(a)$ in $\operatorname{poly}(\log p)$ time.*

Proof. See page 329 in [8]. $\square$

The following theorem from [3] gives a bound on smooth numbers (we say that n is m-smooth if all prime factors of n are at most m).

Theorem 2.7 (Canfield-Erdös-Pomerance)**.** *Let $\psi(x,y)$ denote the number of y-smooth positive integers bounded by x. Let $u = \frac{\log x}{\log y}$. Suppose that $u < (1-\delta)\frac{\log x}{\log\log x}$ for some $\delta > 0$. Then*

$$\frac{1}{x}\psi(x,y) = u^{-u+o(u)}$$

holds uniformly as u and x approach ∞.

3 Algorithm and Analysis

In this section, we present and analyze our algorithm.

The idea for the algorithm is as follows. First we factor $p-1$. Now, for each prime factor q of $p-1$, we find a qth non-residue. We then use Lemma 2.4, to construct a primitive root.

To find a qth non-residue, we first check if q is large or small (compared to $\exp(\sqrt{\log p \log\log p})$). If q is small, we run the algorithm from Lemma 2.5. If q is large, we check the elements $\{1, 2, \ldots, p-1\}$ (in order) until we find one which is a qth non-residue. Lemma 3.1 guarantees that for large q, we will encounter a qth non-residue within the first $\exp(\sqrt{\log p \log\log p})$ elements:

Lemma 3.1. *For all sufficiently large p, for all $q \geq \exp(\sqrt{\log p \log\log p})$ dividing $p-1$, there exists a positive $s \leq \exp(\sqrt{\log p \log\log p})$ which is a qth non-residue.*

Proof. Our strategy will be to assume Lemma 3.1 is false and then to write an inequality comparing the number of $\exp(\sqrt{\log p \log\log p})$-smooth numbers with the number of qth residues. We will then reduce this inequality to a contradiction.

We first calculate $\psi(p, \exp(\sqrt{\log p \log\log p}))$. We use the Canfield-Erdös-Pomerance theorem (Theorem 2.7), and see that $u = \frac{\log p}{\sqrt{\log p \log\log p}} = \frac{\sqrt{\log p}}{\sqrt{\log\log p}}$. Therefore,

$$\frac{1}{p}\psi(p, \exp(\sqrt{\log p \log\log p})) = \left(\frac{\sqrt{\log p}}{\sqrt{\log\log p}}\right)^{-\left(\frac{\sqrt{\log p}}{\sqrt{\log\log p}}\right)+o\left(\frac{\sqrt{\log p}}{\sqrt{\log\log p}}\right)}. \tag{1}$$

For the sake of contradiction, assume that every element $s \leq \exp(\sqrt{\log p \log\log p})$ is a qth residue. Since the product of two elements which are qth residues is also a qth residue, every $\exp(\sqrt{\log p \log\log p})$-smooth number is a qth residue. We therefore know that $\psi(p, \exp(\sqrt{\log p \log\log p}))$ is bounded above by the number of qth residues, which is $p/q \leq p/\exp(\sqrt{\log p \log\log p})$. Combining this with (1) yields

$$\frac{1}{p}(p/\exp(\sqrt{\log p \log\log p})) \geq \left(\frac{\sqrt{\log p}}{\sqrt{\log\log p}}\right)^{-\left(\frac{\sqrt{\log p}}{\sqrt{\log\log p}}\right)+o\left(\frac{\sqrt{\log p}}{\sqrt{\log\log p}}\right)}.$$

Taking the log of both sides gives

$$-\sqrt{\log p \log\log p} \geq -\left(\left(\frac{\sqrt{\log p}}{\sqrt{\log\log p}}\right) + o\left(\frac{\sqrt{\log p}}{\sqrt{\log\log p}}\right)\right)\log\left(\frac{\sqrt{\log p}}{\sqrt{\log\log p}}\right).$$

Multiplying both sides by $\frac{-\sqrt{\log\log p}}{\sqrt{\log p}}$ results in

$$\log\log p \leq \left(1 + \left(\frac{\sqrt{\log\log p}}{\sqrt{\log p}}\right) o\left(\frac{\sqrt{\log p}}{\sqrt{\log\log p}}\right)\right)\log\left(\frac{\sqrt{\log p}}{\sqrt{\log\log p}}\right).$$

And this implies

$$\log\log p \leq (1 + o(1))\frac{1}{2}\log\log p.$$

The above inequality is clearly false, completing the proof. □

Now that we have proven Lemma 3.1, we are ready to analyze the algorithm (Figure 1).

Primitive-Root(p)
1 Factor $p - 1 = \prod_{i=1}^{m} q_i^{e_i}$.
2 **for** each q_i:
3   **if** $q_i > \exp(\sqrt{\log p \log\log p})$
4   Compute the order of $1, 2, \ldots,$ until an element a_i with $q_i^{e_i} | \mathrm{ord}(a_i)$ is found.
5   **if** $q_i \leq \exp(\sqrt{\log p \log\log p})$
6   Find a q_ith non-residue a_i using Lemma 2.5.
7 **return** $\prod_{i=1}^{m} a_i^{(p-1)/q_i^{e_i}}$.

Figure 1: A pseudo-deterministic algorithm finding a primitive root modulo a given prime p.

Correctness of the algorithm follows immediately from Lemma 2.4.

We will now analyze the time complexity of the algorithm:

Lemma 3.2. *The algorithm in Figure 1 runs in time* $L_p(1/2) = \exp(O(\sqrt{\log p \log\log p}))$.

Proof. By Lenstra and Pomerance's factoring algorithm [6], line 1 takes time $L_p(1/2)$.

For each $q_i > \exp(\sqrt{\log p \log\log p})$, by Lemma 3.1, in line 4 we have to find the order of at most $L_p(1/2)$ elements. By Lemma 2.6, finding the order each requires $\mathrm{poly}(\log p)$ time, so line 4 takes a total of $L_p(1/2)\,\mathrm{poly}(\log p) = L_p(1/2)$ time.

For $q_i \leq \exp(\sqrt{\log p \log\log p})$, line 6 takes at most $\exp(\sqrt{\log p \log\log p})\,\mathrm{poly}(\log p) = L_p(1/2)$ time by Lemma 2.5.

Since there are at most $\log p$ primes dividing $p-1$, the loop in line 2 takes a total of $L_p(1/2)\log p = L_p(1/2)$ time.

Calculating the product in line 7 takes $\mathrm{poly}(\log p)$ time. Therefore, the algorithm as a whole terminates in expected time $L_p(1/2)$. $\square$

We now show that the algorithm is pseudo-deterministic. Note that the only randomized steps of the algorithm are line 1 and line 6. In line 1, we use an algorithm that with high probability outputs the factorization of $p - 1$, which is always the same. In line 6, we use an algorithm which is pseudo-deterministic by Lemma 2.5.

This implies our main theorem:

Theorem 3.3. *There exists a pseudo-deterministic algorithm for* Primitive-Root *that runs in expected time* $L_p(1/2)$.

4 Finding a Primitive Root Given Factorization

A related problem to Primitive-Root is Primitive-Root-Given-Factorization:

Definition 4.1. The Primitive-Root-Given-Factorization problem is the problem of finding a primitive root mod p when both p and the factorization of $p - 1$ are given as input.

For Primitive-Root-Given-Factorization, the best known Las-Vegas algorithm runs in polynomial time. The best previously known pseudo-deterministic algorithm runs in time $p^{\frac{1}{4}+o(1)}$. The algorithm from section 3 improves this to $L_p(1/2)$.

In [5], Gat and Goldwasser pose as a problem to find a polynomial time pseudo-deterministic algorithm for Primitive-Root-Given-Factorization. The authors present a polynomial time algorithm for the case $p-1 = kq$, where q is prime and k is of size $\operatorname{poly}(\log p)$. We improve upon this result with a polynomial time algorithm for all p where each prime factor of $p-1$ is of size either at most $\log^c(p)$ or at least $p^{1/c}$, for some constant $c > 1$. Our algorithm runs in time $\log^c(p)\operatorname{poly}(\log p)$. We describe our algorithm in Figure 2.

Primitive-Root-Given-Factorization$(p, p-1 = \prod_{i=1}^{m} q_i^{e_i})$

1 **for** each q_i:
2   **if** $q_i > \exp(\sqrt{\log p \log\log p})$
3   Compute the order of $1, 2, \ldots,$ until an element a_i with $q_i^{e_i} | \operatorname{ord}(a_i)$ is found.
4   **if** $q_i \leq \exp(\sqrt{\log p \log\log p})$
5   Find a q_ith non-residue a_i using Lemma 2.5.
6 **return** $\prod_{i=1}^{m} a_i^{(p-1)/q_i^{e_i}}$.

Figure 2: A pseudo-deterministic algorithm finding a primitive root modulo a prime p, given both p and the factorization of $p-1$.

Correctness of the algorithm follows immediately from Lemma 2.4.

We now prove that if there is some constant c such that all q_i satisfy either $q_i < \log^c p$ or $q_i > p^{1/c}$, then the algorithm terminates in time at most $\log^c(p)\operatorname{poly}(\log p)$.

First, note that for large enough p, if $q_i < \log^c p$ then $q_i < \exp(\sqrt{\log p \log\log p})$. Also, if $q_i > p^{1/c}$ then $q_i > \exp(\sqrt{\log p \log\log p})$.

To prove that line 3 takes polynomial time, we argue that for all fixed $\varepsilon > 0$, for large enough p, if $q_i > p^{1/c}$ then there exists an $a < \log^{c+\varepsilon}(p)$ that is a q_ith non-residue. We do this with a similar strategy to our proof of Lemma 3.1. We know that there are at most $\frac{p-1}{q_i}$ elements which are q_ith residues. Suppose for the sake of contradiction that all $a < \log^{c+\varepsilon}(p)$ are q_ith residues. This implies that there are at least $\psi(p, \log^{c+\varepsilon}(p))$ elements which are q_ith residues. Therefore, we have the inequality

$$\frac{p}{q_i} \geq \psi(p, \log^{c+\varepsilon}(p)).$$

By the Canfield-Erdös-Pomerance theorem (Theorem 2.7), $\psi(p, \log^{c+\varepsilon}(p)) = pu^{-u+o(u)}$, where $u = \frac{\log p}{\log\log^{c+\varepsilon} p}$. Plugging this in and taking the log of both sides yields

$$\log\left(\frac{1}{q_i}\right) \geq -\left(\frac{\log p}{\log\log^{c+\varepsilon}(p)} + o\left(\frac{\log p}{\log\log^{c+\varepsilon}(p)}\right)\right)\log\left(\frac{\log p}{\log\log^{c+\varepsilon}(p)}\right).$$

Simplifying gives

$$\log q_i \leq \left(\frac{log p}{\log\log^{c+\varepsilon}(p)} + o\left(\frac{\log p}{\log\log^{c+\varepsilon}(p)}\right)\right)\log\left(\frac{\log p}{\log\log^{c+\varepsilon}(p)}\right).$$

But we know that $q_i \geq p^{1/c}$. Therefore, $\log q_i \geq \frac{1}{c}\log p$. Plugging this in and simplifying yields

$$\frac{1}{c} \leq \left(\frac{1}{\log\log^{c+\varepsilon}(p)} + \frac{1}{\log p} o\left(\frac{\log p}{\log\log^{c+\varepsilon}(p)}\right)\right) \log\left(\frac{\log p}{\log\log^{c+\varepsilon}(p)}\right).$$

Further simplifying now gives

$$\frac{1}{c} \leq \left(\frac{1}{(c+\varepsilon)\log\log p} + \frac{1}{\log p} o\left(\frac{\log p}{\log\log^{c+\varepsilon}(p)}\right)\right) \left(\log\log p - \log\log\log^{c+\varepsilon}(p)\right).$$

However, the right side approaches $\frac{1}{c+\varepsilon}$, whereas the left side is $\frac{1}{c}$. Therefore, we have reached a contradiction, and so within the first $\log^{c+\varepsilon}(p)$ elements that we test in line 3, we will encounter a q_ith non-residue.

Therefore, line 3 of the algorithm requires calculating the order of up to $\log^{c+\varepsilon}(p)$ elements, each of which takes $\operatorname{poly}(\log p)$ time by Lemma 2.6. Line 5 takes up to $\log^c(p)\operatorname{poly}(\log p)$ time by Lemma 2.5. Since there are at most $\log p$ primes dividing p, the loop in line 1 is of length up to $\log p$. It follows that our algorithm terminates and outputs a primitive root in expected time $\log^c(p)\operatorname{poly}(\log p)$.

Note that on every execution of the algorithm, we output the same primitive root, since the only randomized step of the algorithm is line 5 which is pseudo-deterministic by Lemma 2.5.

This completes the proof of the following theorem:

Theorem 4.2. *For any constant $c > 1$, there exists a pseudo-deterministic algorithm for* Primitive-Root-Given-Factorization *that runs in polynomial time for all p where each prime factor q of $p-1$ satisfies either $q < \log^c(p)$ or $q > p^{1/c}$.*

5 Discussion

It would be interesting to find a polynomial time pseudo-deterministic algorithm for Primitive-Root-Given-Factorization for general primes.

The slowest step in Las Vegas algorithms for Primitive-Root is factoring $p-1$. It would be interesting to find an algorithm which can verify an element is a primitive root without using the factorization of $p-1$.

Acknowledgments

I would like to thank Shafi Goldwasser for introducing me to the primitive root problem, for helpful discussions, and for advice and encouragement on the paper. I would also like to thank Andrew Sutherland for helpful discussions.

References

[1] Eric Bach. How to generate factored random numbers. *SIAM Journal on Computing*, 17(2):179–193, 1988.

[2] DA Burgess. On character sums and primitive roots. *Proceedings of the London Mathematical Society*, 3(1):179–192, 1962.

[3] E Rodney Canfield, Paul Erdös, and Carl Pomerance. On a problem of oppenheim concerning "factorisatio numerorum". *Journal of Number Theory*, 17(1):1–28, 1983.

[4] Whitfield Diffie and Martin E Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, 1976.

[5] Eran Gat and Shafi Goldwasser. Probabilistic search algorithms with unique answers and their cryptographic applications. In *Electronic Colloquium on Computational Complexity (ECCC)*, volume 18, page 136, 2011.

[6] Hendrik W Lenstra and Carl Pomerance. A rigorous time bound for factoring integers. *Journal of the American Mathematical Society*, 5(3):483–516, 1992.

[7] Victor Shoup. Searching for primitive roots in finite fields. *Mathematics of Computation*, 58(197):369–380, 1992.

[8] Victor Shoup. *A computational introduction to number theory and algebra*. Cambridge university press, 2009.

ECCC ISSN 1433-8092

http://eccc.hpi-web.de