Electronic Colloquium on Computational Complexity, Revision 1 of Report No. 136 (2011)

Probabilistic Search Algorithms with Unique Answers and Their Cryptographic Applications

Eran Gat and Shafi Goldwasser*

Weizmann Institute of Science, MIT*

Abstract

In this paper we introduce a new type of probabilistic search algorithm, which we call the *Bellagio* algorithm: a probabilistic algorithm which is guaranteed to run in expected polynomial time, and to produce a correct and *unique* solution with high probability. We argue the applicability of such algorithms for the problems of verifying delegated computation in a distributed setting, and for generating cryptographic public-parameters and keys in distributed settings. We exhibit several examples of Bellagio algorithms for problems for which no deterministic polynomial time algorithms are known. In particular,we show such algorithms for:

- finding a unique generator for $\mathbb{Z}_p^*$ when p is a prime of the form $kq+1$ for q is prime and $k = \text{polylog}(p)$. The algorithm runs in expected polynomial in $\log p$ time.
- finding a unique q'th non-residues of $\mathbb{Z}_p^*$ for any prime divisor q of $p-1$, extending Lenstra's [11] algorithm for finding unique quadratic non-residue of $\mathbb{Z}_p^*$. The algorithm runs in expected polynomial time in $\log p$ and q. The tool we use is a new variant of the Adleman-Manders-Miller probabilistic algorithm for taking q-th roots, which outputs a unique solution to the input equations and runs in expected polynomial time in $\log p$ and q.
- given a multi-variate polynomial $P \neq 0$, find a unique $\vec{a}$ such that $P(\vec{a}) \neq 0$. Alternatively you may think of this as producing a unique polynomial time verifiable certificate of inequality of polynomials.

More generally, we show a necessary and sufficient condition for the existence of a Bellagio Algorithm for relation R: R has a Bellagio algorithm if and only if it is deterministically reducible to some decision problem in BPP.

ISSN 1433-8092

1 Introduction

The study of whether in principle probabilistic algorithms for search problems are more powerful than deterministic algorithms has taken central stage in complexity theory since the early 70's. Indeed, the verdict is still out on this question. A reasonable question to ask from an algorithmic point of view is, assuming that useable randomness exists, are there any advantages offered by running deterministic algorithms rather than probabilistic algorithms ? Say, a probabilistic algorithm offers a significant speedup over the deterministic one, should one still prefer the latter?

Whereas in the case of Monte Carlo algorithm for which correctness holds only with high probability, one may argue the superiority of a deterministic algorithm, the answer seems much less obvious for Las Vegas probabilistic algorithms. Here, although only expected polynomial time runtime guarantees are given, upon termination the output correctness can be verified in polynomial time. Indeed, it is possible that the time to verify the correctness of the output of a Las Vegas algorithm can be much shorter than the time to run a deterministic algorithm for the problem. To exemplify this phenomena, consider primality testing viewed as a search procedure for a *witness* of primality which can be verified in deterministic polynomial time. To verify the witness output by the celebrated AKS algorithm (essentially the execution trace of its fastest variants) will take time $O(n^12)$ whereas to verify the correctness of the witness produced by the elliptic curve [15, 7] primality testing algorithm[1] will take time $O(n^3)$ for primes of length n

Yet, as highlighted in a talk of Lenstra on finding standard models for representing finite fields[12, 11], there is an important observable difference between deterministic and Las Vegas algorithms. For the same input, a deterministic algorithm will yield the same runtime and the *same output* in each execution. In contrast, the input does not fully determine the runtime or more importantly the output of a Las Vegas algorithm, which may vary depending on the particular randomness used in a particular execution. Generally, a probabilistic algorithm run twice on the same input with different randomness, is *not unique* and can (and usually does) return two different outputs. For example, on the same prime p, two executions of the widely used probabilistic procedure for finding a generator g for the cyclic group Z_p^* will generally yield two different generators; two executions of probabilistic primality certifying algorithms [1],[15]) will with high probability yield two different certificates of primality; two executions of the Schwartz-Zipple ([25, 20]) procedure on input polynomials f and g such that $f \neq g$, will with high probability output a different v such that $f(v) \neq g(v)$;

This raises a general goal – which is the *focus of our work* – to design expected polynomial time probabilistic algorithms which for every given input with high probability produce the same output, called the *canonical output*, regardless of the randomness used, which can be verified correct in deterministic polynomial time. We remark that obviously a deterministic polynomial time algorithm will answer this requirement. The challenge is to design probabilistic algorithms with unique outputs in those cases were either no deterministic polynomial time algorithms are known, or the probabilistic algorithm offer efficiency improvements.

[1]Las Vegas Assuming polynomial size gap between consecutive primes

We believe that the uniqueness of output per a given input is of marked importance in distributed settings where the same computation is performed by different parties on the same input but possibly with access to different sources of randomness. Such settings come up in the context of cryptography when a group of users may distrust other users randomness sources and yet they wish to generate common cryptographic system-wide keys (or public parameters in the case of IBE), or for the purpose of efficiently verifying the correctness of each others computation in delegated cloud computation, volunteer computing, or in simulating each others scientific experiments.

- **Cryptographic Keys:** Say a group (or pair) of users wishes to choose a common cryptographic key, a common generator g for $\mathbb{Z}_p^*$ for a given prime p, but nobody trusts each others "random choices" for g. An algorithm which finds a canonical g will come in handy.
- **Volunteer Computation or Delegating Computation to Several Servers:** Suppose that we wish to delegate computation, for example in a cloud computing or a volunteer computing infrastructure. In such infrastructures a central body uses the computational power of other willing users. The Berkeley Open Infrastructure for Network Computing (BOINC)[4, 5] is such a platform whose intent is to make it possible for researchers in fields as diverse as physics, biology and mathematics to tap into the enormous processing power of personal computers around the world. This is done by distributing the work, often in the form of delegation of smaller tasks. It could be ,however, that the server to which we delegated the work is corrupt and is returning a false answer to our query. We thus may wish to check the correctness of the returned value. One approach is to have the server prove that its output is correct. This is not always easy to do, and seems in general to require interaction to reduce the verification time, or making non-standard assumptions. A simpler approach, apparently used in practice in BOINC, is to delegate the work to many servers and accept their outputs only if they all return the same answer. Clearly, this approach works only if the algorithm the servers use produces a unique answer. A probabilistic algorithm which is guaranteed to produce a canonical output will enable a simple measure of checking correctness of probabilistic algorithm.

Before we describe our results, we remark that perhaps the most compelling challenge for finding a "unique output", which we urge the reader to keep in mind, is on input n to find a unique prime $p \in [n, 2n]$ via an expected polynomial time algorithm. Currently, density theorems on the distribution of primes imply that in expected n trials choosing at random $x \in [n, 2n]$, we will find an x which is a prime, which can be certified either probabilistically by [1, 15] or deterministically [3]. Deterministically, the best algorithm known is by Lagarias and Odlyzko in time $O(n^{\frac{1}{2}+o(1)})$. A recent work by Tao proposes a strategy to determine in time $O(n^{\frac{1}{2}-c})$ some $c > 0$ of how to tell whether a given interval in $[n, 2n]$ contains a prime as a way to improve Lagarias and Odlyzko's method, Nonetheless, this will still result in an exponential procedure.

1.1 The Contributions of This paper

1.1.1 A New Notion: Bellagio Probabilistic Algorithms

We suggest the study of a new type of probabilistic algorithm, the *Bellagio Algorithms*[2] A Bellagio algorithm for a search problem V is probabilistic algorithm which runs in expected polynomial time and which on the same input, produces a unique and correct output with high probability over the randomized choices of the algorithm.

More formally, let $V(\cdot,\cdot)$ be a polynomial time computable function (in its first argument). We say that $V(x,\cdot)$ is satisfiable if there exists y such that $V(x,y)=1$. V naturally defines a search problem: on input $x \in \{0,1\}^*$, find y such that $V(x,y)=1$ if one exists and else output *reject*; and a decision problem: on input $x \in \{0,1\}^*$ output 1 if and only if $V(x,\cdot)$ is satisfiable.
A *Bellagio algorithm* for search problem V is a probabilistic algorithm A such that (a) On every input x, A terminates in expected polynomial time; (b) If $V(x,\cdot)$ is satisfiable, then $Prob[V(x,A(x)) \neq 1] = negl(|x|)$ where the probability is taken over A's coin tosses, else $A(x)$ outputs *reject*; (c)Let $A(x,r)$ denote an execution of A on input x with coins r. Then, $Prob[A(x,r) \neq A(x,r')] = negl(|x|)$ where the probability is taken over the choices of r and r'. We call $\max_r\{y = A(x,r)\}$ the canonical output of A on input x.

Note that executions of a Bellagio algorithm can be easily made to be indistinguishable from executions of a deterministic algorithm with repect to any probabilistic polynomial time distinguisher algorithm which can observe the I/O behavior of the algorithm as well as its running time, by running it for a fixed polynomial number of steps per input (beyond the time of expected termination). Thus, we may view such algorithms as *pseudo deterministic*.

The first non-trivial example of a Bellagio search algorithm is contained in the work of Lenstra and Bart de Smit's [12, 11] on finding standard models for representing finite fields. They show a probabilistic algorithm which on input prime p, outputs a unique quadratic non-residues. We note that the obvious approach of picking a random $z \in Z_p^*$ and checking whether the Legendre symbol of z is -1 terminates in expected $O(1)$ trials and will clearly output a different z each time. It is not known how to find a quadratic non-residue in deterministic polynomial time unless we assume the ERH.

We emphasize that Lenstra et al. [12, 11] require their algorithm to always (with probability 1) produce the same quadratic non-residue, whereas we relaxed the requirement of uniqueness to hold only with high probability. This will be an important difference which plays out in our examples of a Bellagio algorithm to find unique non-zero's of polynomials and for our general theorem 4.

We proceed to exhibit three new simple examples of Bellagio algorithms with relevance to cryptographic applications. We then provide a general necessary and sufficient condition which allows the design of Bellagio algorithms: for any search problem for which there exists a deterministic polynomial time (cook) reduction from the search problem to a BPP decision problem, a Bellagio algorithm can be designed.

[2]Bellagio is a well known unique casino in Las Vegas,NV

1.1.2 Three Examples of Bellagio Algorithms

Example 1:

First, we consider the search problem of finding a generator of $\mathbb{Z}_p^*$for p for strong primes p where $p-1$ has a prime factor of size polylog(p). Such generators are of special interest to cryptography, since many protocols based on the Diffie-Hellman problem [14] make use of generators to establish keys. is especially common place in cryptographic applications. It is well known how to efficiently test if a given element of Z_p^* is a generator or not, given the factorization of $p-1$. Furthermore, we know that $\phi(p-1)$ elements are generators of $\mathbb{Z}_p^*$, where $\phi(x)$ is Euler's totient function, which implies that the density of generators is $O(\frac{1}{\log\log(p)})$ [19]. Thus, the following well known probabilistic procedure on input p outputs a generator: choose at random an element in $\mathbb{Z}_p^*$ and test if is a a generator. If so output it, otherwise repeat. This procedure will always produce correct outputs since the test ensures that the output is correct, and due to the large density of generators, will terminate in expected polynomial time. However, it most likely outputting a different generator in each execution. We remark that for general p, when the factorization of $p-1$ is unknown, no efficient algorithm for finding a generator is known.

Informal Theorem 1[3] Let p be a prime and q a prime such that q divides $p-1$. There exists a probabilistic (Bellagio) algorithm $\mathcal{G}$ for finding a generator of $\mathbb{Z}_p^*$ for any input prime p of the form $qk+1$ where k is of size polylog(p). We will call$\mathcal{G}(p):=\tilde{g}_p$ a *canonical generator* of Z_p^*. The algorithm runs in expected time polynomial in k and $\log(p)$ which This matches previous time bounds achieved by existing probabilistic algorithm (without canonical outputs).

The idea of the proof is to follow what we call a *Canonization* strategy: start with a probabilistic Las Vegas algorithm for finding generators and convert any of its (possible many) outputs whose value depend on the particular coin tosses made, to a single canonical generator. See details in section 3.

Example 2:

Second, we consider the problem of finding a unique qth non-residue in $\mathbb{Z}_p^*$ when p is prime. We call an element $a \in \mathbb{Z}_p^*$ a *q'th residue* for q divisor of $p-1$ if $a \equiv b^q \pmod p$ for some $b \in \mathbb{Z}_p^*$, and a *q'th non-residue* otherwise. Efficiently testing whether a number is a q'th non-residue or not can be done by checking whether $a^{(p-1/q)} \not\equiv 1 \pmod p$. The density of the qth non-residues constitue a $\frac{q-1}{q}$ of the elements of $\mathbb{Z}_p^*$Thus, to find a qth non-residue mod p, one can simply sample a random element in $\mathbb{Z}_p^*$ and tests if the sampled element is a qth non-residue till one is found. Clearly, when the algorithm terminates it produces a qth non-residue; but if the algorithm is run twice on the same p it will generally produce a different quadratic non-residue. The algorithm runs in expected polynomial time in $\log p$ and in the value of q. Our aim is to find an equally efficient probabilistic Bellagio algorithm $\mathcal{NR}$.

[3]We remark that the case of strong primes where $p-1$ is divisable by a large prime factor (as in the premise of the theorem)

As mentioned above, Lenstra's algorithm [11] for finding a canonical non-residue (which we explain in appendix A) is a Bellagio algorithm for the case of $q = 2$ – quadratic non-residues. We extend it as follows.

Informal Theorem 2: There exists a probabilistic (Bellagio) algorithm $\mathcal{NR}$ that on inputs p and q such that p is prime and q is a prime such that q divides $p-1$, finds a q'th non-residue in $\mathbb{Z}_p^*$. Call the output $\mathcal{NR}(p,q)$ a *canonical qth non-residue*. The algorithm runs in time polynomial in q and $\log(p)$. This matches previous time bound achieved by existing probabilistic algorithms (which do not have canonical outputs). See details in section 5.

The crucial idea of the proof will be how to reduce the question of finding a unique qth non-residue to repeated calls to a subroutine, which in itself should be a Bellagio algorithm , for taking q'th roots in $\mathbb{Z}_p^*$Ṅamely, we need an algorithm that on inputs p, q, a compute a canonical b in $\mathbb{Z}_p^*$ such that $b^q = a \mod p$ if one exists. One algorithm for computing qth roots, due to Adleman-Manders-Miller[2], runs in expected time polynomial in q and $\log(p)$, is in itself probabilistic and is guaranteed to produce one (but not necessarily the same one, in different executions) of the q possible qth roots of a. To ensure that ultimately our q-th non-residue finding algorithm produces a canonical output, we modify the AMM algorithm to produce a canonical b such that $b^q \equiv a \pmod p$. The idea is: find all q-th roots (which explains the runtime dependence on the value of q) and output the smallest one.[4]

Example 3:

In the above examples of finding a generator and finding a qth non-residue, under the Generalized Riemann Hypothesis, one may alternatively deterministic algorithms which simply enumerate the elements of $\mathbb{Z}_p^*$ testing for being a generator and testing for being qth non-residue respectively. The running time of such procedures is not known to be polynomial time, but Generalized Riemann Hypothesis, small generators and q-th non-residues exist . Our third example is of a different nature.

Consider the fundamental problem of Polynomial identity testing (PIT). Although deterministic algorithms remain elusive, many randomized algorithms are known for this problem. Formally, we are given an arithmetic circuit computing a multivariate polynomial over some field, and we have to determine whether that polynomial is identically zero or not. This problem embodies many special case algorithmic questions. For example, testing equivalence of read-once branching programs [8], and more. There are two well-studied models in which the PIT problem is considered. The first is the so-called black-box model in which the only access to the circuit is by asking for its value on inputs of our choice. The second setting is the non black-box model in which the circuit is given as input. The PIT problem is difficult in both settings, and its difficulty stems from that the polynomial is not given explicitly as a list of coefficients in either setting, bur rather in a form which allows evaluation without seeing the coefficients.

[4]We note that Lenstra [11] similarly used a square root taking algorithm to produce both square roots solutions to the quadratic equation $a = x^2 \mod p$. In this case there are only two b and $p - b$ solutions .

There are several randomized (black-box) algorithms designed for the problem [13, 25, 20] which are generally referred to as the Schwartz- Zippel algorithm. They are based on the idea that by substituting random values to the variables from a large enough domain, one gets, with high probability, a zero value only if the polynomial is zero. Conversely, if the input polynomial is not identically zero, the Schwartz-Zippel algorithm outputs an assignment to the variables on which the polynomial value is not zero, which in itself is a certificate (as polynomial time verifiable proof) that the input polynomial is not identically zero. This is not a unique certificate however ! By the very nature of the algorithm, to argue it is correct with high probability, the assignment must be chosen at random, and thus two different executions deeming the polynomial non-zero will find with high probability two different assignments proving this fact. For completion, we include the Schwartz-Zippel lemma.

Lemma ([Zip79, Sch80]). Let $f(x_1, ..., x_n)$ be a nonzero polynomial of degree at most dover field F and let $T \subset F$. If we choose $a = (a_1, ..., a_n) \in T_n$ uniformly at random, then $Pr[f(a) = 0] \leq d/|T|$.

The lemma suggests a randomized algorithm for PIT: given a degree d polynomial f, pick at random $a \in T^n$ and check whether $f(a) = 0$. If $f \neq 0$, the probability of error is at most $d/|T|$, and otherwise, we are always correct.

One may define a natural *search variant* of the PIT decision problem which we call SPIT: on input a polynomial f, if f is not the zero polynomial, output an assignment $\vec{a}$ such that $f(\vec{a}) \neq 0$ and otherwise output *reject*. Our aim is to design a probabilistic Bellagio algorithm for SPIT which on an input polynomial f with high probability finds a unique $\vec{a}$ for which $f(\vec{a}) \neq 0$ when $f \neq 0$, and otherwise outputs *reject*.

Informal Theorem 3: Let $\epsilon > 0$. There exists a probabilistic (Bellagio) algorithm SPIT which on input a nonzero polynomial $f(x_1, ..., x_n)$ of degree at most d over field F with $T \subset F$ and $|T| > m$, outputs with probability $1 - \epsilon$ a unique assignment $a = (a_1, ..., a_n)$ to the variables of f such that $f(a_1, .., a_n) \neq 0$. The algorithm runs in expected polynomial time in $n, \log m, r, \epsilon^{-1}$.

We remark that the guarantee we give for the algorithm of theorem 3 will be different than the previous two examples in two respects. First it addresses an underlying decision problem, PIT, which is not even known to be in NP – when f is identically zero, we know of no short prof of this fact. Thus, the algorithm of theorem 3 may assert the polynomial identically zero even if it is not, with non-zero probability. Second, uniqueness is only guaranteed with high probability. There is a negligible but non zero probability, that on two different executions of the algorithm, the assignment which is output will be different even when in both executions the assertion is that the input f is non-zero.

The idea is to take the Schwartz-Zippel algorithm and modify it to look for the lexicographically first assignment which makes f non zero. To do this, we fix assignments to variable $x_1, ..., x_n$ one at a time, and at each time–say that at iteration i we established that $x_1, ...x_i = a_1, ..a_i$ – we run the Schwartz-Zippel algorithm on the polynomial in the $n - i$ unassigned variables $f' = f(a_1, ...a_i, x_{i+1}, ...x_n)$ to test with high probability if f' is identically zero. If it is deemed non-zero with high probability, we go on to fix the value of x_{i+1}; otherwise we try the lexicographically next value for x_i and re-iterate. What we get then, is an algorithm that with high probability will find

the first input $\vec{a} \in T^n$ such that $f(\vec{a}) \neq 0$. It is important to observe that this idea will fail if any of the calls to the Schwartz-Zippel algorithm will fail . The probability of this event can be made negligibly small. In such unlikely case, one of two bad events can occur. We will either find a different assignment a' such that $f(a') \neq 0$ or make an error and assert that f is identically zero. The definition of Bellagio algorithm made allowances for such event with negligible probability.

We remark that this is all the detail we shall give in this extended abstract for the proof of informal theorem 3. Full description will be provided in the final paper.

1.1.3 A General Theorem

Viewing the algorithm outlined in Example 3 above more generally, it is essentially a reduction of SPIT *search problem* to a sequence of calls to the Schwartz-Zipple algorithm on a sequence of PIT *decision problems*. This is very reminiscient of the general methodology of reducing search to decision for NP-complete problems.

In our context, it leads the way to a general paradigm for constructing Bellagio algorithms: Find a decision problem in BPP to which your search problem can be reduced and you can construct a Bellagio algorithm for the latter. In fact, this is a necessary condition. Namely, there will always exist a BPP set to which a Bellagio search problem can be deterministically reduced via a Cook reduction.

Informal Theorem 4: Any search problem S that is deterministically reducible in polynomial time (via a Cook reduction) to a decision problem in BPP, can be solved by a Bellagio algorithm. Conversely, if search problem S has a Bellagio algorithm then finding solutions for S is deterministically reducible to some BPP decision problem.

The idea is as follows: say the polynomial time reduction between search problem V and decision problem L on input x makes a polynomial number of calls on instances $z_1, ...z_k$of L to a *perfectly* correct and deterministic oracle for L. In such case, by definition, the reduction from search to decision, produces a deterministic and thus unique solution y to the search problem such that $V(x, y) = 1$ if $V(x,)$ is satisfiable. Now, replace each call to the perect oracle by a call to BPP algorithm, and make sure that the error probability in each call is small enough so that the union bound over the total in the polynomial number of calls, will give a negligible error in total. Then, with overwhelming probability each call to the BPP algorithm on $z_1, ..., z_k$ will return the same answers in differene executions of the BPP algorithms with different randomness, and thus the search algorithm will end up constructing a canonical y such that $V(x, y) = 1$ if $V(x,)$ is satisfiable. Recall that for a BPP decision problem L, or every $\epsilon >$, there exists a probabilistic Monte Carlo algorithm A that for every input x: if x in L then $Prob[A(x) = 1] > 1 - \epsilon$ and if x is not in L then $Prob[A(x) = 1] < \epsilon$, and the algorithm always terminates in polynomial time in $\epsilon^{-1}, |x|$ so can set the error bound as needed that.

We remark that in the case of finding non-zeros, the reduction underlying example 3 is indeed the standard search-to-decision reduction, but theorem 4 allows for more general reductions. For example, the best parallel algorithms for finding perfect matchings are based on randomized BPP

algorithm for testing whether a given determinant is formally zero or not, in a highly non-standard fashion and satisfy the conditios of theorem 4 [17, 16, 18, 10].

Finally, note that insisting on a deterministic reduction is unavoidable, since each PPT algorithm (for search) may be viewed as a PPT reduction to a trivial problem.

The converse is almost immediate. Suppose that A is a Bellagio algorithm for V, and consider the set L s.t. $(x, i, b) \in L$ iff Pr[ith bit of$A(x) = b] > 2/3$. Then, (i) L is a BPP language, and (ii) finding solutions wrt V is deterministically reducible to L.

1.2 Other Related Work

One approach to find a canonical element of some special type is to pick the smallest such element. For example, if one shows that the smallest quadratic non-residue is found within the first polylog(p) elements of $\mathbb{Z}_p^*$, we can efficiently find a canonical quadratic non-residue by going over all elements of $\mathbb{Z}_p^*$ one by one and checking whether they are non-residues until one is found.

Using this approach, two main results are known. The first holds under the assumption of the Generalized Reimann Hypothesis(GRH). Assuming GRH, Ankeny [6] showed that the smallest quadratic non-residue is of size $O(log^2(p))$. Without any assumptions, the best known tool for dealing with bounds on the smallest non-residue comes from the theory of character sums. Using character sums, the best known bound is $p^{\frac{1}{4}+o(1)}$ due to Burgess [9].

As for the problem of the least generator of $\mathbb{Z}_p^*$, Shoup [22] showed that, assuming GRH, the smallest generator is of size $O(\log^6(p))$. Hence if it is possible to verify whether an element is a generator in polynomial time, then going over the elements by brute force until some generator is found takes polynomial time, and gives a canonical generator. Without making any assumptions the best bound is $p^{\frac{1}{4}+o(1)}$ similar to the case of the smallest non-residue.

Organization: The rest of this paper is organized as follows. Since definitions of a Bellagio algorithm and general outline of the ideas of the proofs for theorem 3 and 4 appear in the introduction, we restrict our attention in this extended abstract to the constructions underlying theorem 1 and 2. In section 3 we define the problem of finding a canonical generator of $\mathbb{Z}_p^*$ and present an algorithm which solves this problem for primes of the form $p = kq + 1$ where q is prime and $k = \text{polylog}(p)$. We show an algorithm for canonically taking qth roots in $\mathbb{Z}_p^*$ in section 4. In section 5 we present an algorithm for finding a canonical q'th non-residue for any prime q dividing $p - 1$. Various number theoretic claims and propositions can be found in 2. For completion we describe and prove the algorithm given by Lenstra for finding a canonical non-residue in Appendix A. We present several open questions in 6.

2 Preliminaries

Throughout this section, let p be an odd prime unless stated otherwise. We begin by defining two special types of elements of $\mathbb{Z}_p^*$ - q'th non-residues and generators.

Definition 2.1. *Let p be an odd prime. An element a of $\mathbb{Z}_p^*$ is called a quadratic residue if a has square root modulo p, that is if $a \equiv b^2 \pmod{p}$ for some $b \in \mathbb{Z}_p^*$. If a doesn't have a square root modulo p, it is called a quadratic non-residue. Similarly, for any q, an element $a \in \mathbb{Z}_p^*$ is a q'th residue if $a \equiv b^q \pmod{p}$ for some $b \in \mathbb{Z}_p^*$, and a is a q'th non-residue if such element does not exist.*

Definition 2.2. *Let p be an odd prime. A generator of $\mathbb{Z}_p^*$ is an element $g \in \mathbb{Z}_p^*$ such that every element $a \in \mathbb{Z}_p^*$ can be uniquely expressed as $g^x \pmod{p}$ where $1 \leq x \leq p-1$.*

Definition 2.3. *The order of an element $a \in \mathbb{Z}_p^*$ is the smallest integer k such that $a^k \equiv 1 \pmod{p}$.*

We denote the order of a by $\mathrm{ord}(a)$. We state several useful number theoretic propositions regarding the order of an element:

Proposition 2.4. *If $\mathrm{ord}(a) = n$, $\mathrm{ord}(b) = m$, where n and m are co-prime, then $\mathrm{ord}(ab) = n \cdot m$.*

Proposition 2.5. *If $\mathrm{ord}(a) = k \cdot l$, then $\mathrm{ord}(a^k) = l$.*

Proposition 2.6. *If $\mathrm{ord}(a) = k$, then $\forall h \in \{1, \ldots, p-1\}$ we have $a^h \equiv 1 \pmod{p} \Leftrightarrow k|h$.*

Proposition 2.7. *An element $a \in \mathbb{Z}_p^*$ is a quadratic non-residue if and only if $a^{(p-1)/2} \equiv -1 \pmod{p}$.*

Proposition 2.8. *Let q be a prime such that $q|p-1$. An element $a \in \mathbb{Z}_p^*$ is a q'th residue if and only if $a^{(p-1)/q} \equiv 1 \pmod{p}$. Equivalently, $a \in \mathbb{Z}_p^*$ is a q'th non-residue if and only if $a^{(p-1)/q} \not\equiv 1 \pmod{p}$.*

Proof: Suppose a is a q'th residue, then there exists some $b \in \mathbb{Z}_p^*$ s.t. $a \equiv b^q \pmod{p}$. Hence, $a^{(p-1)/q} \equiv b^{p-1} \equiv 1 \pmod{p}$.

Conversely, Let a be such that $a^{(p-1)/q} \equiv 1 \pmod{p}$. Let g be a generator of $\mathbb{Z}_p^*$. Then a can be written as g^s for some integer s. Since g is a generator, its order is $p-1$, and hence by proposition 2.5 the order of $g^{(p-1)/q}$ is q. We have that $(g^{(p-1)/q})^s = (g^s)^{(p-1)/q} \equiv a^{(p-1)/q} \equiv 1 \pmod{p}$, hence by proposition 2.6 the order of $g^{(p-1)/q}$, which is q, divides s. Denote $s = kq$. Now $a \equiv g^{kq} = (g^k)^q \pmod{p}$. We get that a is a q'th residue, since $(g^k)^q \equiv a \pmod{p}$. ■

The proof of the above proposition gives an important relation between quadratic non-residues and generators of $\mathbb{Z}_p^*$. We see that if a is a q'th residue, then it is some power of g^q for any generator g. Formally:

Proposition 2.9. *Let q be a prime such that $q|p-1$, and let g be a generator of $\mathbb{Z}_p^*$. Let $a \in \mathbb{Z}_p^*$ and $a = g^s$ where $0 \leq s \leq p-1$. We have that a is a q'th residue if and only if $q|s$.*

Corollary 2.10. *Half of the elements of $\mathbb{Z}_p^*$ are quadratic non-residues. More generally, $\frac{q-1}{q}$ of the elements of $\mathbb{Z}_p^*$ are q'th non-residues for a prime q dividing $p-1$.*

Proposition 2.11. *The number of generators of $\mathbb{Z}_p^*$ is $\phi(p-1)$ where $\phi(x)$ is Euler's totient function. The density of generators of $\mathbb{Z}_p^*$ is $O(\frac{1}{\log\log(p)})$ [19].*

Claim 2.12. *Let g be a generator of $\mathbb{Z}_p^*$, and let k be such that $k|p-1$. The set $\{a \in \mathbb{Z}_p^* : \text{ord}(a)|k\}$ equals the set $\{g^{\frac{p-1}{k}\cdot i} : 1 \leq i \leq k\}$.*

Proof: We show mutual inclusion of the two sets:
$\supseteq$: Let $h = g^{\frac{p-1}{k}i}$ for some $1 \leq i \leq k$. Since $h^k = (g^{\frac{p-1}{k}i})^k = (g^{k\frac{p-1}{k}})^i \equiv 1^i \equiv 1 \pmod{p}$, proposition 2.6 gives us that $\text{ord}(h)|k$.
$\subseteq$: Let h be some element s.t. $\text{ord}(h)|k$. Since g is a generator of $\mathbb{Z}_p^*$, $h = g^m$ for some $m \in \mathbb{Z}_p^*$. By proposition 2.6, $\text{ord}(h)|k \Rightarrow h^k = g^{mk} \equiv 1 \pmod{p}$. Since g is a generator, $\text{ord}(g) = p-1$, thus by proposition 2.6, $p-1|mk$, which implies that $\frac{(p-1)}{k}|m$. Denote $m = \frac{p-1}{k} \cdot t$. If $t \leq k$ we are done, otherwise write $t = t_1 k + r$ where $r \leq k$. We now have $g^m = g^{\frac{p-1}{k}\cdot t} = g^{\frac{p-1}{k}\cdot(t_1 k + r)} = g^{(p-1)\cdot t_1} g^{\frac{p-1}{k}r} \equiv g^{\frac{p-1}{k}r} \pmod{p}$. Since $r \leq k$ we are done. ■

Claim 2.13. *An element g is a generator of $\mathbb{Z}_p^*$ if and only if $g^{(p-1)/q} \not\equiv 1 \pmod{p}$ for every prime q dividing $p-1$.*

Proof: Let g be a generator of $\mathbb{Z}_p^*$, then $\text{ord}(g) = p-1$ by definition. Hence for every $k < p-1$, $g^k \not\equiv 1 \pmod{p}$. Thus, for every $q|p-1$ we have that $g^{(p-1)/q} \not\equiv 1 \pmod{p}$.

Conversely, let g be such that $g^{(p-1)/q} \not\equiv 1 \pmod{p}$ for every prime $q|p-1$. Suppose that g is not a generator, hence $\text{ord}(g) = k$ for some $k < p-1$. Since the order of an element always divides the order of the group $\mathbb{Z}_p^*$, we have that $k|p-1$. Since $k < p-1$, we must have that for some prime divisor q of $p-1$, $k|\frac{p-1}{q}$. By proposition 2.6 we get that $g^{(p-1)/q} \equiv 1 \pmod{p}$, a contradiction. Hence $\text{ord}(g) = p-1$ and g is a generator of $\mathbb{Z}_p^*$. ■

3 A Canonical Generator of $\mathbb{Z}_p^*$ for $p = kq+1$, where q is prime and k is of size $\text{polylog}(p)$

We would like to find a probabilistic (Bellagio) polynomial time algorithm $\mathcal{G}$ satisfying the following requirement: For each prime input p there exists a generator $\tilde{g}_p$ of $\mathbb{Z}_p^*$ such that for any randomness ρ used by the algorithm, $\mathcal{G}(p,\rho) = \tilde{g}_p$, where by $\mathcal{G}(p,\rho)$ we mean $\mathcal{G}$ running on input p using randomness ρ.

We restrict our attention to a specific case in which the factorization of $p-1$ is of a special form. Specifically, we give a polynomial time algorithm for finding a canonical generator for primes of the form $p = kq+1$, where q is prime and k is of size $\text{polylog}(p)$. We note that in this case $p-1$ can be efficiently factored. We present an algorithm for finding a canonical generator in this case.

The algorithm is as follows: First, we find some element of large order $b' \in \mathbb{Z}_p^*$, such that the order of b' is greater than q. We then use b' to obtain an element b of order exactly q. Next, we find some canonical (in the sense that it will be unique per p) element c of order k [5]. Such elements are obtained by taking any randomly chosen generator to powers which are multiples of q. The product $b \cdot c \pmod{p}$ will give us a canonical generator.

Our algorithm for finding a canonical generator, which we denote $CGen$, is given below. Without loss of generality we assume that $p-1$ is given in factored form, that is both k and q are known:

Algorithm 1 Calculate $\tilde{g}_p$, a canonical generator of $\mathbb{Z}_p^*$, for $p = kq + 1$

Input: p prime, q prime, k such that $p = kq + 1$.

1: Go over elements $b' \in \mathbb{Z}_p^*$ one by one until one is found satisfying $(b')^k \not\equiv 1 \pmod{p}$. This element is the smallest element in $\mathbb{Z}_p^*$ satisfying $\mathrm{ord}(b') \geq q$.
2: Find the order of b' using Shoup's algorithm. Denote $\mathrm{ord}(b') := rq$, and let $b := (b')^r \pmod{p}$.
3: Find a generator g of $\mathbb{Z}_p^*$. This is done by picking a random element of $\mathbb{Z}_p^*$ and checking if its order is kq, using Shoup's algorithm, repeatedly until such element is found.
4: Compute the set $G = \left\{ g^{qi} \pmod{p} : 1 \leq i \leq k \right\}$. Sort its elements in increasing order.
5: Find the smallest $c \in G$ set s.t $\mathrm{ord}(c) = k$ by going over the sorted elements of G and checking their order until one is found.
6: Return $\tilde{g}_p = b \cdot c \pmod{p}$.

Theorem 1. *Let p be a prime of the form $kq+1$, where q is prime. Algorithm 1 finds a canonical generator of $\mathbb{Z}_p^*$ in expected polynomial time $O(\log\log(k) \cdot \log\log(p) \cdot \log^3(p) + k \cdot \log\log(k) \cdot \log^3(p))$. When k is of size polylog(p), the algorithm runs in expected polynomial time in $\log(p)$.*

Proof: We begin by showing that the output of the algorithm, $\tilde{g}_p = b \cdot c \pmod{p}$, is indeed a generator of $\mathbb{Z}_p^*$. It suffices to show that b is of order q and c is of order k. This is because since q is prime and $k < q$, we have that $\gcd(k, q)$=1. Hence proposition 2.4 gives us that $\mathrm{ord}(b \cdot c) = k \cdot q = p - 1$, hence $\tilde{g}_p$ is a generator of $\mathbb{Z}_p^*$.

We first show that b is of order q. Let b' be the element found in step 1. Since $(b')^k \not\equiv 1 \pmod{p}$, by proposition 2.6 $\mathrm{ord}(b') \nmid k$. Since $\mathrm{ord}(b') \mid p-1$ and $p-1$ is of the form $k \cdot q$ where q is prime, this gives us that $q | \mathrm{ord}(b)$. Denote $\mathrm{ord}(b') = rq$. By proposition 2.5, $\mathrm{ord}(b) = \mathrm{ord}((b')^r) = q$.

It remains to show that c is of order k. Let g be the generator found in step 3. By proposition 2.5, $\mathrm{ord}(g^q) = k$. Observe the set $G = \{g^{q \cdot i} : 1 \leq i \leq k\}$. Since $g^q \in G$, the set of elements in G whose order equals k is not empty, hence there is some smallest element $c \in G$ of order k which we find in step 5. Since c is defined to be of order k, this shows that our algorithm returns a generator of $\mathbb{Z}_p^*$.

We now show that the algorithm returns a canonical generator. To do so, we must show that for any randomness used, the algorithm always returns the same result. Observe that the only use of

[5] We use an algorithm shown by Shoup ([23], page 329) for the general case where the factorization of $p-1$ is known. Let $p - 1 = \prod_{i=1}^{r} q_i^{e_i}$. Shoup shows that one can find the order of $a \in \mathbb{Z}_p^*$ in time $O(\log(r)\log^3(p))$. Since r can be at most $\log(p)$, the algorithm works in time $O(\log\log(p)\log^3(p))$

randomness is to select the generator in step 3. This generator is only used to construct the set G in step 4. Thus, to show that the output of the algorithm is unique per prime p, we must show that for any two generators g_1, g_2 of $\mathbb{Z}_p^*$, $\left\{g_1^{qi} : 1 \leq i \leq k\right\} = \left\{g_2^{qi} : 1 \leq i \leq k\right\}$. This follows from claim 2.12 which states $\{g_1^{q \cdot i} : 1 \leq i \leq k\} = \{a \in \mathbb{Z}_p^* : \text{ord}(a) | k\} = \{g_2^{q \cdot i} : 1 \leq i \leq k\}$.

It remains to show that the algorithm runs in time polynomial in $\log(p)$. Denote $p-1 = q \cdot \prod_{i=1}^{r} k_i^{e_i}$. Note that r can be at most $\log(p)$.

Step 1 : By claim 2.12, we know that the number of elements whose order divides k, which is equal to the number of elements whose order is $\leq k$, is k. Hence one of the first $k+1$ elements has order at least q. We raise each of these elements to the power k and check whether they are equivalent to 1 modulo p. Each check takes time $O(\log^3(p))$ required for performing exponentiation. Hence finding b' takes time $k \cdot O(\log^3(p))$.

Step 2 : Denote $\text{ord}(b') = rq$. Finding the order of b' takes $O(\log(r)\log^3(p))$ time using Shoup's algorithm. Raising b' to the power of r takes $O(\log^3(p))$ time. Hence finding b takes time $O(\log(r)\log^3(p))$.

Step 3: We need to analyze the expected running time of this step. As stated in proposition 2.11, the density of generators of $\mathbb{Z}_p^*$ is $O(\frac{1}{\log\log(p)})$. Thus, we expect to find some generator after $O(\log\log(p))$ tries. For each element we select, we check whether it is a generator by verifying that its order is $p-1$ using Shoup's algorithm, hence step 3 has expected running time $O(\log\log(p) \cdot \log(r)\log^3(p))$.

Step 4 : Given a generator g, calculating g^q takes $O(\log^3(p))$. Once we have g^q, we generate the set $G := \{g^{q \cdot i} : 1 \leq i \leq k\}$ by performing k exponentiations, which takes time $k \cdot O(\log^3(p))$. Since sorting these elements takes time $k \cdot \log(k)$, in total this step takes time $k \cdot O(\log^3(p))$.

Step 5 : Checking the order of elements in G takes $O(\log(r)\log^3(p))$ time. Hence we find the element c in time at most $k \cdot O(\log(r)\log^3(p))$.

Step 6: Multiplying b and c to obtain $\tilde{g}_p$ takes $O(\log^2(p))$.

The above calculations gives us that the total running time of the algorithm is $O(\log\log(p) \cdot \log(r)\log^3(p)) + k \cdot O(\log(r)\log^3(p))\big)$. Since r is the number of prime divisors of k, we have that $r = O(\log(k))$. Taking into account that k is of size polylog(p), we can consider the running time as: $O(k \cdot \log\log(k) \cdot \log^3(p))$. Hence the algorithm runs in time polynomial in $\log(p)$ when k is of size polylog(p), as required. ■

We compare our algorithm to a different approach of calculating a canonical generator under the GRH assumption. Assuming GRH, one can show that the smallest generator $g \in \mathbb{Z}_p^*$ satisfies $g < c \cdot \log^6(p)$ for some constant $c > 0$. Hence, assuming GRH, the algorithm which checks elements of $\mathbb{Z}_p^*$ one by one to see if they are generators runs in time $O(\log^6(p)) \cdot O(\log\log(p)\log^3(p))$. This algorithm finds a canonical generator, the smallest one of $\mathbb{Z}_p^*$. We note that, even if the GRH assumption holds, for values of k of size less then $\log^6(p)$ our algorithm $CGen$ runs faster than the algorithm which finds the smallest generator.

Remark 1: The case where $p-1=2^k$ for some k is a simple case for finding a canonical generator. This is because a simple argument, presented in [24] chapter 6.3, shows that every quadratic non-residue in $\mathbb{Z}_p^*$ is a generator. Thus, in this case it suffices to find a canonical quadratic non-residue of $\mathbb{Z}_p^*$ which can be done using Lenstra's algorithm given in appendix A.

Remark 2: A variant of an algorithm for computing a canonical generator on an input prime, is to look for an algorithm $\mathcal{G}$ which generates pairs $(p, \tilde{g_p})$ where p is prime and $\tilde{g_p}$ is a generator of $\mathbb{Z}_p^*$. In this case the canonical property is that if $(p_1, \tilde{g_{p_1}})$,$(p_2, \tilde{g_{p_2}})$ are two outputs of $\mathcal{G}$ and $p_1 = p_2$, then $\tilde{g_{p_1}} = \tilde{g_{p_2}}$.

Such an algorithm can work by using known methods for generating large primes p with known factorization of $p-1$, and then use this factorization to compute a canonical generator. Methods for generating p with $p-1$ in factored form work by first generating a random factored number n and testing $n+1$ for primality. Details regarding this approach can be found in [23] pages 298-300.

4 Taking qth Roots Canonically in $\mathbb{Z}_p^*$

Our goal is to construct an probabilistic algorithm $\mathcal{A}$ which is given as input p, a prime divisor q of $p-1$ and a q'th residue $a \in \mathbb{Z}_p^*$ and satisfies the following property: for any $a \in \mathbb{Z}_p^*$ there exists $b \in \mathbb{Z}_p^*$ such that $b^q \equiv a \pmod{p}$ and for any randomness ρ used by $\mathcal{A}$ we have $\mathcal{A}(p, q, a, \rho) = b$.

The main idea of our solution is to perform a canonization process on a probabilistic algorithm for taking qth roots. This algorithm, which we denote AMM, is due to Adelman, Manders and Miller [2]. The AMM algorithm takes a prime p, a prime divisor q of $p-1$ and a q'th residue a. It makes use of a q'th non-residue, γ, which it is either given as input or selects probabilistically. If it is given γ as input, we denote the algorithm by $AMM^{(\gamma)}$. The output of the algorithm is an element b such that $a \equiv b^q \pmod{p}$. The output of the AMM algorithm is effected by the choice of γ. The AMM algorithm was first shown in [2]. The running time of the algorithm is $O(q \cdot \log(p)^4)$.

The canonization process makes use of the fact that given some q'th root of $a \in \mathbb{Z}_p^*$, we can construct all q'th roots of a by obtaining all q'th roots of unity in $\mathbb{Z}_p^*$. Denote by E the set of all q'th roots of 1 in $\mathbb{Z}_p^*$. $|E| = q$ and it can be constructed in time polynomial in q given any q'th non residue γ. Let $b \in \mathbb{Z}_p^*$ such that $a \equiv b^q \pmod{p}$. All of the q'th roots of a are given by the elements $\{b \cdot e : e \in E\}$. Canonization process uses the AMM algorithm to obtain b, and use b to generate the set $\{b \cdot e : e \in E\}$. Picking the smallest element in this set provides a canonical way to take q'th roots. We denote this algorithm by $CAMM$, or $CAMM^{(\gamma,E)}$ if γ and E are given as input. See algorithm 2 for the formal definition of the algorithm.

Theorem 2. *Let p be a prime, q a prime divisor of $p-1$, $a \in \mathbb{Z}_p^*$ a q'th residue. Algorithm 2 returns a canonical q'th root of a in expected polynomial time $O(q \cdot \log^4(p))$.*

Proof: We begin by showing that algorithm 2 returns a q'th root of a. We first show that the set E is calculated correctly in step 2. By definition, $E = \{x \in \mathbb{Z}_p^* : \mathrm{ord}(x) = q\} \cup \{1\}$, hence contains

Algorithm 2 Taking q'th roots of $a \in \mathbb{Z}_p^*$ canonically for q prime, $q|p-1$.

Input: p prime, q a prime divisor of $p-1$, a a qth residue in $\mathbb{Z}_p^*$.
Optional Input: E - the set of all q'th roots of 1 in $\mathbb{Z}_p^*$
Optional Input: γ - q'th non residue.

1: If γ was not given as input, pick elements $\gamma \in \mathbb{Z}_p^*$ at random until one is found satisfying $\gamma^{(p-1)/q} \not\equiv 1 \pmod{p}$.
2: If E is not given as input, Let $E = \{\gamma^{\frac{p-1}{q}i} : 1 \leq i \leq q\}$.
3: Set $b \leftarrow AMM^{(\gamma)}(p, q, a)$.
4: Return $min\{b \cdot e \pmod{p} : e \in E\}$.

exactly all elements of $\mathbb{Z}_p^*$ whose order divides q. Thus, by claim 2.12 $|E| = q$. This means that once we find q distinct elements of $\mathbb{Z}_p^*$, each with order either q or 1, then we have found the set E. Let γ be some q'th non-residue. We show that the set $\{\gamma^{\frac{p-1}{q}i} : 1 \leq i \leq q\}$ contains q elements, each of order either q or 1, hence this set is E. Since γ is a q'th non-residue, $\gamma^{\frac{p-1}{q}} \not\equiv 1 \pmod{p}$. However, $(\gamma^{\frac{p-1}{q}})^q = \gamma^{p-1} \equiv 1 \pmod{p}$, hence $\text{ord}(\gamma^{\frac{p-1}{q}})|q$. Since q is prime, we must have that $\text{ord}(\gamma^{\frac{p-1}{q}}) = q$. Thus, the set $\{\gamma^{\frac{p-1}{q}i} : 1 \leq i \leq q\}$ has q distinct elements. Furthermore, For any i we have that $(\gamma^{\frac{p-1}{q}i})^q = (\gamma^{p-1})^i \equiv 1 \pmod{p}$. Summing up, we get that $E = \{\gamma^{\frac{p-1}{q}i} : 1 \leq i \leq q\}$, as required.

We now show that if a is a q'th residue, then all of its q'th roots are given by the set $\{b \cdot e : e \in E\}$, where b is the q'th root obtained by running the AMM algorithm in step 3. Suppose that $a \equiv b^q \equiv b_1^q \pmod{p}$ where $b_1 \in \mathbb{Z}_p^*$ and $b \not\equiv b_1 \pmod{p}$. We show that $b_1 = e \cdot b$ where $e \in E$. This holds since $(b_1 b^{-1})^q = b_1^q (b^q)^{-1} \equiv a \cdot a^{-1} \equiv 1 \pmod{p}$. Thus, the order of $b_1 b^{-1}$ divides q, and hence $b_1 b^{-1} \in E$, or equivalently $b_1 = e \cdot b$ for some $e \in E$. Moreover, for any $e \in E$ we have that $(b \cdot e)^q = b^q e^q \equiv a \cdot 1 \pmod{p}$. Hence $b \cdot e$ is a q'th root of a. We conclude that all of the q'th roots of a are given by the set $\{b \cdot e \pmod{p} : e \in E\}$. Since the returned value is $min\{b \cdot e \pmod{p} : e \in E\}$, we indeed return a q'th root of a.

Next, we show that the algorithm always returns a canonical root of the q'th residue a. To see this, observer that the randomness is only used to pick the q'th non-residue γ, which in turn is used to create the set E and to obtain b using the AMM algorithm. Let γ_1, γ_2 be two q'th non-residues in $\mathbb{Z}_p^*$. We have shown that for any q'th non residue γ, the set E which is defined as $\{x \in \mathbb{Z}_p^* : \text{ord}(x) = q\} \cup \{1\}$ satisfies $E = \{\gamma^{\frac{p-1}{q}i} : 1 \leq i \leq q\}$. Hence $\{\gamma_1^{\frac{p-1}{q}i} : 1 \leq i \leq q\} = E = \{\gamma_2^{\frac{p-1}{q}i} : 1 \leq i \leq q\}$. That is, the set created in step 2 is the same no matter what q'th non-residue is chosen using randomness ρ. Let b_1 be the q'th root obtained by running the AMM algorithm on input a using the q'th non-residue γ_1, and similarly for b_2. Denote by Q_a all q'th roots of a in $\mathbb{Z}_p^*$. We have shown that for any $b \in \mathbb{Z}_p^*$ satisfying $b^q \equiv a \pmod{p}$, the set $\{b \cdot e : e \in E\} = Q_a$. Hence, $\{b_1 \cdot e : e \in E\} = Q_a = \{b_2 \cdot e : e \in E\}$. Thus, the minimal element of this set, which is returned in step 4, is the same no matter which q'th non-residue was chosen, and the algorithm returns a canonical q'th root.

To finish our proof, we show that the algorithm runs in time polynomial in q and $\log(p)$. By corollary 2.10, $\frac{q-1}{q}$ of the elements of $\mathbb{Z}_p^*$ are q'th non-residues, hence we expect to find γ after two tries. Constructing the set $E = \{\gamma^{\frac{p-1}{q}i} : 1 \leq i \leq q\}$ takes q exponentiations, which require $O(q \cdot \log^3(p))$ time. The AMM algorithm requires time $O(q \cdot \log^4(p))$. After we obtain a q'th root b, we construct the set $\{b \cdot e : e \in E\}$. Since $|E| = q$, this step requires q multiplications which take $O(q \cdot \log^2(p))$. Hence the total running time of the algorithm is $O(q \cdot \log^4(p))$. ■

5 A Canonical q'th Non-residue of $\mathbb{Z}_p^*$

In a recent talk [11], Lenstra presented the idea of canonizing the output of probabilistic algorithms in order to find a canonical quadratic non-residue. His algorithm and a proof of correctness can be found in appendix A. Here, we want to find, given any prime p, a *canonical q'th non-residue* of $\mathbb{Z}_p^*$, which we denote $\tilde{q}_p$. Formally, we look for a probabilistic polynomial time algorithm $\mathcal{NR}$ such that for any prime p and prime $q|p-1$ there exists a q'th non residue $\tilde{q}_p \in \mathbb{Z}_p^*$ such that for any randomness ρ, $\mathcal{NR}(p,q,\rho) = \tilde{q}_p$, where by $\mathcal{NR}(p,q,\rho)$ we mean $\mathcal{NR}$ running on input p,q using randomness ρ.

We obtain such an algorithm by showing a reduction from the problem of finding a canonical q'th non-residue to that of taking q'th roots. If the latter has canonical output then we can find q'th non-residues canonically. We take qth roots canonically using the $CAMM$ algorithm described in the previous section. The reduction is obtained by showing that when taking q'th roots of -1 repetitively until we are eventually unable to do so, in which case we obtain a q'th non-residue. A full overview follows.

The algorithm takes as input a prime p. We check whether -1 is a q'th non residue by checking if $-1^{(p-1)/q} \equiv -1 \pmod{p}$. If it indeed a non-residue, return -1 as the canonical q'th non-residue. If -1 is a quadratic residue, we probabilistically select a q'th non residue γ. Using γ, we construct the set $E = \{x \in \mathbb{Z}_p^* : \text{ord}(x) = q\} \cup \{1\}$ as shown in the previous section. We run the $CAMM$ algorithm with input p,q and using -1 as the q'th residue. We use γ and E which the algorithm requires instead of recalculating them. The $CAMM$ algorithm returns a canonical q'th root of -1, which we denote by b. If b is a q'th non-residue, we are done, otherwise we continue by canonically taking a q'th root of b. Continue in this manner until some q'th non-residue is found. We return that element as the canonical quadratic non-residue. Formally, see algorithm 3.

Theorem 3. *Let p be a prime. Algorithm 3 finds a canonical q'th non-residue of $\mathbb{Z}_p^*$ in expected polynomial time $O(q \cdot \log^5(p))$.*

Proof: We begin by showing that the algorithm must halt and return a q'th non-residue. If -1 is a q'th non-residue, we return it as our output. Otherwise, if it is a q'th residue, by proposition 2.9 we know that for some generator g we have $g^{q^k Q} \equiv -1 \pmod{p}$ where $k > 0$. Let b be the q'th root obtained by performing the first iteration of step 5. Clearly, $g^{q^{k-1}Q}$ is also q'th root of -1. Denote by E the set $\{x \in \mathbb{Z}_p^* : \text{ord}(x) = q\} \cup \{1\}$ as above. In the proof of theorem 2 we have shown

Algorithm 3 Calculate a canonical q'th non-residue of $\mathbb{Z}_p^*$ for a prime $q|p-1$

Input: p prime, q prime dividing $p-1$

1: If $-1^{(p-1)/q} \not\equiv 1 \pmod{p}$, return -1.
2: Pick elements $\gamma \in \mathbb{Z}_p^*$ at random until one is found satisfying $\gamma^{(p-1)/q} \not\equiv 1 \pmod{p}$.
3: Let $E = \{\gamma^{\frac{p-1}{q}i} : 1 \leq i \leq q\}$.
4: Set $a \leftarrow -1$.
5: Repeat until $a^{(p-1)/q} \not\equiv 1 \pmod{p}$: Set $b \leftarrow CAMM_q^{(\gamma,E)}(p,q,a)$, and set $a \leftarrow b$.
6: Return a

that all q'th roots of -1 are given by the set $\{g^{q^{k-1}Q} \cdot e : e \in E\}$. Thus, b is of the form $g^{q^{k-1}Q} \cdot e$ for some $e \in E$. Furthermore, by claim 2.12 the set E can also be written as $\{g^{\frac{p-1}{q}i} : 1 \leq i \leq q\}$. Denote $p-1 = q^m \cdot Q'$ where $m \geq k$. We now have that b is of the form $g^{q^m Q' i} \cdot g^{q^{k-1}Q}$. Continuing inductively in this manner, we get that after t q'th root extractions we are left with an element that can be expressed as g to a power of the form $(\sum_{j=k-t+1}^{m} q^j a_j) + q^{k-t}Q$, where a_j are just coefficients of the powers of q. The smallest power of q in this sum is decreased by one after each iteration. After k iterations, the smallest power of q in this sum becomes 0, hence the square root is of the form $g^{(\sum_j q^j a_j)+Q}$. Since $q \nmid Q$, we have that $q \nmid (\sum_j q^j a_j) + Q$, and hence by proposition 2.9 we have found a q'th non-residue, which the algorithm outputs.

The proof that the algorithm outputs a canonical q'th non residue is similar to that of Lenstra's algorithm for finding quadratic non-residues given the previous explanations, since step 5 is performed in a canonical manner.

It remains to prove that the algorithm has expected running time polynomial in q and $\log(p)$. By corollary 2.10, $\frac{q-1}{q}$ of the elements of $\mathbb{Z}_p^*$ are q'th non-residues, hence we expect to find one after two tries, and each attempt takes time $O(q \cdot \log^2(p))$. Constructing the set $E = \{\gamma^{\frac{p-1}{q}i} : 1 \leq i \leq q\}$ takes q exponentiations, which require $O(q \cdot \log^3(p))$ time. For the analysis of the other parts of the algorithm, suppose that -1 is a q'th residue. Then, as previously explained, -1 is of the form $g^{q^k \cdot Q}$ where $q \nmid Q$ and $k > 0$. Since the exponent is smaller than $p-1$, we get that k can be at most $\log(p)$. We run the $CAMM^{(\gamma,E)}$ algorithm for taking q'th roots at most k times. Each time that the $CAMM^{(\gamma,E)}$ algorithm is run requires a running time of $O(q \cdot \log^4(p))$. Since k is at most $\log(p)$, we get that the total expected running time of the algorithm is $O(q \cdot \log^4(p))$, as required. ■

6 Open Questions

We find most intriguing question is the one stated in the introduction: find a Bellagio algorithm for finding a prime in the interval $n, 2n]$ on input n.

With respect to the question of finding a canonical generator of $\mathbb{Z}_p^*$—, we left open the question of how to find a canonical generator of $\mathbb{Z}_p^*$ for a general prime p. A good starting point may be to

study the case where p is of the form $2qr+1$ where q, r are of size $O(\sqrt{p})$. When dealing with p of this sort there are only a few possibilities for the order of an element. It can be either $2, q, r, 2q, 2r$ or $2qr$. We note that it suffices to find a canonical element of a power which is a multiple of q and a canonical element of power which is a multiple of r in order to find a canonical generator. Currently, there is no known method of finding a canonical generator for the general case in time less then $p^{\frac{1}{4}+o(1)}$, where this running time is obtained using the character sums analysis as mentioned in section 1.2.

As discussed in the introduction, when the factorization of $p-1$ is unknown, we do not know how to obtain a certified generator, even without requiring the output to be canonical. It will be of interest even to find a canonical generator in time as expensive as the best known factorization algorithm, which is sub-exponential in $\log(p)$. In other words, can we find a suitable reduction between factoring and finding a canonical generator?

Finding a reduction between having a black-box access to an algorithm which is guaranteed to output a generator of $\mathbb{Z}_p^*$ and finding a canonical generator would be quite interesting as well. Note that the canonization process described within is not black box.

6.1 Acknowledgements

We are grateful to Amir Shpilka for encouraging us to include his observation of how to canonize the Schwartz Zipple algorithm. We are also thankful to Vinod Vaikuntanathan, Oded Goldreich and Dana Ron for discussions and MUCH encouragement on this paper.

References

[1] L. Adleman and M.-D. Huang, *Primality testing and abelian varieties over finite fields*, in Lecture Notes in Mathematics, vol. 1512, Springer, 1992.

[2] L. Adleman, K. Manders, and G. Miller, *On taking roots in finite fields*, in Proceedings of the 18th Annnual Symposium on Foundations of Computing, 1977, pp. 175–178.

[3] M. Agrawal, N. Kayal, and N. Saxena, *PRIMES is in P*, Annals of Mathematics, 160 (2004), pp. 781–793.

[4] D. Anderson, *Public computing: Reconnecting people to science*, in Conference on Shared Knowledge and the Web, Citeseer, 2003, pp. 17–19.

[5] D. P. Anderson, *BOINC: A system for public-resource computing and storage*, in Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, IEEE Computer Society, 2004, pp. 4–10.

[6] N. Ankeny, *The least quadratic non residue*, Annals of Mathematics, (1952), pp. 65–72.

[7] A. Atkin and F. Morain, *Elliptic curves and primality proving*, Math. Comput., 61 (1993), pp. 29–68.

[8] M. Blum, A. K. Chandra, and M. N. Wegman, *Equivalence of free boolean graphs can be tested in polynomial time*, Information Processing Letters, 10 (1980), pp. 80–82.

[9] D. Burgess, *On character sums and primitive roots*, Proceedings of the London Mathematical Society, 12 (1962), pp. 179–192.

[10] S. Chari, P. Rohatgi, and A. Srinivasan, *Randomness-optimal unique element isolation with applications to perfect matching and related problems*, SIAM Journal on Computing, 24 (1995), pp. 1036–1050.

[11] B. de Smit and H. W. Lenstra, *Standard models for finite fields.* www.damtp.cam.ac.uk/user/na/FoCM/FoCM08/Talks/Lenstra.pdf.

[12] ——, *Modeling finite fields*, 2008. Talk given at the Haifa University by Prof. Lenstra.

[13] R. A. DeMillo and R. J. Lipton, *A probabilistic remark on algebraic program testing*, Information Processing Letters, 7 (1978), pp. 193–195.

[14] W. Diffie and M. E. Hellman, *New directions in cryptography*, IEEE Transactions on Information Theory, IT-22 (1976), pp. 644–654.

[15] S. Goldwasser and J. Kilian, *Primality testing using elliptic curves*, Journal of the ACM, 46 (1999), pp. 451–472.

[16] R. Karp, E. Upfal, and A. Wigderson, *Constructing a perfect matching is in random NC*, Combinatorica, 6 (1986), pp. 35–48.

[17] L. Lovasz, *On determinants, matchings, and random algorithms*, in Fundamentals of Computing Theory, L. Budach, ed., Akademia-Verlag, 1979.

[18] K. Mulmuley, U. Vazirani, and V. Vazirani, *Matching is as easy as matrix inversion*, Combinatorica, 7 (1987), pp. 105–113.

[19] J. Rosser and L. Schoenfield, *Approximate formulas for some functions of prime numbers*, Illinois J. Math, 6 (1962), pp. 64–94.

[20] J. T. Schwartz, *Fast probabilistic algorithms for verication of polynomial identities*, Journal of the ACM, 27 (1980), pp. 701–717.

[21] D. Shanks, *Five number-theoretic algorithms*, in Proceedings of the 2nd Manitoba Conference on Numerical Mathematics, 1972, pp. 51–70.

[22] V. Shoup, *Searching for primitive roots in finite fields*, in Proceedings of the 22nd Annual ACM Symposium on Theory of Computing,, 1990, pp. 546–554.

[23] ——, *A Computational Introduction to Number Theory and Algebra*, Cambridge University Press, New York, NY, USA, 2009.

[24] J. Tattersall, *Elementary number theory in nine chapters*, Cambridge University Press, 2005.

[25] R. Zippel, *Probabilistic algorithms for sparse polynomials*, Symbolic and Algebraic Computation, (1979), pp. 216–226.

A Lenstra's Algorithm for Finding Canonical Quadratic Non-Residues

We want to find, given any prime p , a *canonical quadratic non-residue* of $\mathbb{Z}_p^*$, which we denote $\tilde{q}_p$. Formally, we look for a probabilistic polynomial time algorithm $\mathcal{Q}$ such that for any prime p there exists a quadratic non-residue $\tilde{q}_p \in \mathbb{Z}_p^*$ such that for any randomness ρ, $\mathcal{Q}(p, \rho) = \tilde{q}_p$, where by $\mathcal{Q}(p, \rho)$ we mean $\mathcal{Q}$ running on input p using randomness ρ.

Lenstra's algorithm for finding $\tilde{q}_p$ makes use of an algorithm for calculating square roots modulo a prime p, which we refer to as the *Shanks-Tonelli algorithm (ST)* [21]. This algorithm takes as input a prime p and a quadratic residue a. The ST algorithm also makes use of some quadratic non-residue which it is either given as input or selects probabilistically. If a quadratic non-residue γ is given as input, we denote the algorithm as $ST^{(\gamma)}$. Depending on the non-residue that it uses, the algorithm returns an element $b \in \mathbb{Z}_p^*$ satisfying $a \equiv b^2 \pmod{p}$. The ST algorithm has expected running time polynomial in $\log(p)$. We describe a general form of this algorithm for taking q'th roots for any prime q dividing $p - 1$ in appendix **??**.

The main idea of Lenstra's algorithm for finding a canonical quadratic non-residue is to continuously take square roots of some element of $\mathbb{Z}_p^*$, in this case -1, using the ST algorithm. Since the ST algorithm can return any one of the two square roots, depending on the quadratic non-residue which it uses, we will make sure that it works in a canonical way. At some point, we will not be able to take square roots any more, which means that we have found a quadratic non-residue.

We explain the algorithm in more detail. The algorithm takes as input a prime p. We check whether -1 is a quadratic non-residue by checking if $-1^{(p-1)/2} \equiv -1 \pmod{p}$. If it indeed a non-residue, return -1 as the canonical non-residue. If not, then there is some $x \in \mathbb{Z}_p^*$ such that $x^2 \equiv -1 \pmod{p}$. The algorithm probabilistically selects some quadratic non-residue γ by selecting random elements until a quadratic non-residue is found, as explained in the beginning of this section. It runs the $ST^{(\gamma)}$ square root algorithm on input p, using -1 as the quadratic residue, and using γ as the quadratic non-residue which the algorithm requires. Note that both x and $-x$ are square roots of -1, and the algorithm may return any one of them, depending on γ. Clearly, given one we can obtain the other. We compare the two square roots, taking the smaller square root, i.e. the one smaller or equal to $\frac{p-1}{2}$. Denote that root by b. If b is a non-residue, we are done, otherwise we continue by taking a square root of b using the $ST^{(\gamma)}$ algorithm. Continue in this manner until some quadratic non-residue is found. We return that element as the canonical quadratic non-residue. Formally, the algorithm is described as follows:

Algorithm 4 Calculate a canonical quadratic non-residue of $\mathbb{Z}_p^*$

1: Find the largest power of 2 dividing $p-1$, denoted by k, and let $p-1 = 2^k \cdot Q$.
2: If $-1^{(p-1)/2} \equiv -1 \pmod{p}$, return -1. Otherwise, set $a \leftarrow -1$.
3: Pick elements $\gamma \in \mathbb{Z}_p^*$ at random until one is found satisfying $\gamma^{(p-1)/2} \not\equiv 1 \pmod{p}$.
4: Repeat $k-1$ times: Set $b \leftarrow ST^{(\gamma)}(p,a)$, and set $a \leftarrow min\{b \pmod{p}, -b \pmod{p}\}$.
5: Return a

Theorem 4 (Lenstra [11])**.** *Let p be a prime. Algorithm 4 finds a canonical quadratic non-residue of $\mathbb{Z}_p^*$ in time polynomial in log(p).*

Proof: We start by showing that the algorithm must terminate and return some quadratic non-residue. If -1 is a quadratic non-residue, we return it. We deal with the case where -1 is a quadratic residue. Write $p-1 = 2^k \cdot Q$ where Q is odd. Let g be any generator of $\mathbb{Z}_p^*$. Henceforth, we consider -1 as $g^{(p-1)/2} \pmod{p}$. We show that we can perform step 4 $k-1$ times: Initially, the square roots are $g^{2^{k-2}Q}$ and $-g^{2^{k-2}Q}$. For $g^{2^{k-2}Q}$ we can clearly take another square root. As for $-g^{2^{k-2}Q}$, we can write it as $g^{2^{k-1}Q} \cdot g^{2^{k-2}Q}$, and now it is apparent that a square root can be taken. After t steps, a is of the form $g^{(\sum_{i=k-t}^{k-1} 2^i a_i Q) + 2^{k-t-1}Q}$ where $a_i \in \{0,1\}$. Note that the smallest power in the sum is reduced by one after each step. Hence, at the end of step 4 the square root is of the form $g^{(\sum_{i=1}^{k-1} 2^i a_i Q)+Q}$. Hence, since Q is odd, our output a is g to some odd power, which by proposition 2.9 means that a is a quadratic non-residue.

We now show that algorithm 4 returns a canonical result. The only step which is effected by the choice of randomness is step 3. Let ρ be the randomness used by the algorithm, and let γ_ρ be the resulting non-residue selected by the algorithm. To see that the canonical property holds it suffices to show that step 4 of the algorithm is performed in a canonical way. To see this, observe that if the Shanks-Tonelli algorithm is run on $a \in \mathbb{Z}_p^*$ such that $a \equiv x^2 \pmod{p}$, it can return either x or $-x$ as its output, depending on γ_ρ. Since in step 4 we always takes the smaller square root out of the two, the resulting root b will be the same no matter what non-residue γ_ρ was used by the ST algorithm. Thus, the algorithm satisfies the canonical property.

It remains to show that the algorithm is efficient. Given $p-1 := 2^k \cdot Q$, we can find k in time $\log(p)$. Since half of the elements of $\mathbb{Z}_p^*$ are quadratic non-residues, we expect to find a one in two tries. Each attempt requires performing exponentiation, which takes $\log^3(p)$ time. We then perform $k-1$ applications of the Shanks-Tonelli algorithm, which runs in time polynomial in $\log(p)$. Hence the expected running time of the algorithm is polynomial in $\log(p)$, which completes the proof of the claim. ■

ECCC ISSN 1433-8092

http://eccc.hpi-web.de