Integer factorization and discrete logarithm problems

Pierrick Gaudry

Integer factorization and discrete logarithm problems

Pierrick Gaudry

October 2014

Abstract

These are notes for a lecture given at CIRM in 2014, for the "Journées Nationales du Calcul Formel". We explain the basic algorithms based on combining congruences for solving the integer factorization and the discrete logarithm problems. We highlight two particular situations where the interaction with symbolic computation is visible: the use of Gröbner basis in Joux's algorithm for discrete logarithm in finite field of small characteristic, and the exact sparse linear algebra tools that occur in the Number Field Sieve algorithm for discrete logarithm in large characteristic.

Disclaimer: *These notes contain no new material. They also do not aim to be a survey, and exhaustivity is not a goal. As a consequence, many important works (old or recent) are not mentioned.*

1 The problems and their cryptographic significance

1.1 Cryptographic context

Cryptography, as a science that studies tools to secure communications in a wide sense, contains various sub-themes among which the so-called public-key cryptography covers algorithms such as digital signatures, and asymmetric encryption, i.e. encryption systems in which the key that is used to encrypt is publicly known, while the corresponding decryption key is known only by the recipient. This is useful in particular when the participants do not share a common secret prior to any interaction.

Most public key algorithms used today are based on difficult problems in number theory, and more precisely integer factorization and discrete logarithm problems, which constitute the central topic of this lecture. Let us give two examples.

EMV. The EMV is the standard adopted by Visa, Mastercard, the French Carte Bleue and many others, to secure the chip-and-pin smartcard transactions. The public key part of the standard is based on the RSA cryptosystem, whose security relies on the presumed difficulty of factoring integers. The typical key sizes found in today's cards are 1024 bits or slightly more. Anyone who would be able to factor numbers of this size could easily break into the system and create fake transactions.

SSL/TLS. The SSL/TLS standard is the one used to secure many protocols found on the internet. For instance, the `s` at the end of `https` acronym means that the connection between your web-browser and the server is secured using SSL/TLS. By clicking on the lock picture when browsing on such a web site, it is easy to know the algorithms that are used for this particular connection. Currently, depending on the browser, the client OS, and the server, one can see various public key algorithms, the most frequent being RSA, DSA and ECDSA. The security of DSA relies on the difficulty to compute discrete logarithms in a prime finite fields, while ECDSA's security is linked to discrete logarithms between points of an elliptic curve over a finite field.

In order to select the key sizes for these applications, it is necessary to study thoroughly the underlying problems: measuring what can an attacker do with the best known algorithm is crucial, especially when the technical constraints are such that taking a lot of margin to be on the safe side is not an option (like in embedding systems where each millijoule is expensive).

1.2 Integer factorization and related notions

Factoring integers is an old and well-known problem; we recall it here for completeness.

Definition 1. *The integer factorization problem is the following: given a positive integer N, compute its decomposition into prime numbers $N = \prod p_i^{e_i}$ (unique up to reordering).*

A related question is primality testing: how hard is it to decide whether a number is a prime or a composite? Both from the theoretical side [3, 2] and from the practical side [5] this is now considered as an easy problem. Indeed, there exists a polynomial-time deterministic algorithm, and there are practical algorithms (maybe probabilistic, maybe Monte-Carlo, like Miller-Rabin) that are efficient enough for most purposes.

As a consequence, it is easy to verify whether the result of integer factorisation is correct, so that if we have a probabilistic algorithm that might fail, detecting the failure is always easy, at least at the end.

Another important tool is the sieve of Eratosthenes: this is an efficient way to enumerate all primes up to a bound B in quasi-linear time in B. Although this is a very old algorithm, there have been some progress in the past decades [4, 12].

Using the sieve of Eratosthenes, the easiest algorithm we can think of for factoring an integer N is **trial division**: for all the primes p in increasing order, check if $p|N$. If yes, divide N by as many powers of p as possible. While the remaining part is not a prime, continue. The complexity of this algorithm is quasi-linear in the second largest prime dividing N.

The dependence of the complexity in the second largest prime dividing N is not specific to the trial division algorithm. It will occur for any factoring algorithm that extract prime divisor one at a time, with a sub-algorithm whose complexity depends on the size of the prime to be extracted and almost not on the size of N.

As a consequence, the difficulty of the integer factorization problem is not uniform: factoring integers N for which the second largest prime is bounded by a polynomial in $\log N$ can be done in polynomial time. Conversely, it is generally assumed that the most difficult case is when N is the product of only two primes of the same size. This is the reason why they are used in the RSA algorithm.

An important notion that will be present in many algorithms is **smoothness**.

Definition 2. *An integer is B-smooth if all its prime factors are smaller than B.*

Being B-smooth becomes a less difficult condition when B grows. The two extreme cases are $B = 2$, where the only 2-smooth numbers are powers of 2, or B which is larger than the numbers we consider, so that all of them are smooth. The proportion of B-smooth numbers is an important topic in analytic number theory. For our purpose, the main result is by Canfield, Erdős and Pomerance.

Theorem 1. *Let $\psi(x, y)$ be the number of y-smooth positive integers smaller than x. Let $u = \log x / \log y$. Fix $\varepsilon > 0$ and assume that $3 \leq u \leq (1 - \varepsilon)\frac{\log x}{\log \log x}$. Then we have*

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

In order to get a better insight of this result, we introduce a function that, in the context of integer factorization is called the **subexponential function**.

Definition 3. *The subexponential function is*

$$L_N(\alpha, c) = \exp\left(c(\log N)^{\alpha}(\log \log N)^{1-\alpha}\right),$$

where α is in $[0, 1]$ and c is a positive number.

The main parameter is α: if $\alpha = 0$, then $L_N(\alpha, c) = (\log N)^c$, so that it describes a polynomial complexity, while if $\alpha = 1$, we get N^c, that is an exponential complexity. Any intermediate value of α yields a complexity that is called subexponential. Sometimes, when only a first order estimate is needed, the second parameter c is omitted.

The theorem of Canfield, Erdős and Pomerance can be restated in terms of the subexponential function as follows.

Corollary 2. *Let N be an integer and $B = L_N(\alpha, c)$. The proportion of numbers less than N that are B-smooth is in $L_N(1 - \alpha, (1 - \alpha)/c + o(1))^{-1}$.*

This estimate is at the heart of the analysis of many algorithms. Indeed, it says that by fixing a smoothness bound not too small and not too large, the proportion of smooth numbers will also be sufficiently high so that there is a chance to find one by trail and error. This leads to the question of actually detecting a smooth number.

Definition 4. *The smoothness test problem is as follows: given a number N and a bound B, decide whether N is B-smooth.*

The trial division algorithm can be stopped once the prime p that is considered gets larger than B. This gives a deterministic algorithm for the smoothness test problem with a complexity that is quasi-linear in B, i.e. exponential in the size of B.

There exists much better algorithms for testing the smoothness of integers. All known algorithms in this family also find the factorization of N in the case where N is smooth, just like trial division does. The most famous is the Elliptic Curve Method (ECM) invented by Lenstra [17]. Its behaviour is probabilistic, in the sense that when the algorithm does not return the factorization of N after a certain amount of time, all we can deduce is that the probability that N was B-smooth is small (and this probability can be made arbitrarily small). But the situation is even worse in theory: this property is not rigorously proven. There are very reasonable heuristics that would imply that the statement is correct, but with the current knowledge, it can be proven that it is correct only for most of the input numbers.

Let us give a sloppy statement; we refer to Lenstra's article for a precise theorem.

Theorem 3. *Under reasonable heuristics, there exists a probabilistic algorithm called ECM that, given an integer N and a bound B, returns either the factorization of N or fails. If N is B-smooth, then the probability that is succeeds is at least $1/2$. The running time is in $(\log N)^{O(1)} L_B(1/2, \sqrt{2} + o(1))$.*

By setting the smoothness bound to the maximal possible value of the second largest prime factor of N, namely $B = \sqrt{N}$, it is possible to turn ECM into a probabilistic factoring algorithm. Its complexity is therefore in $L_N(1/2, 1 + o(1))$. We will see in Section 2.1 another algorithm with the same complexity.

As can be guessed from its name, the ECM algorithm relies on the theory of elliptic curves. We will not describe it here, although this is an important tool in algorithmic number theory. In the following, we will take the ECM algorithm as a black-box, and will always assume that it works as expected from the heuristics. We finally mention that ECM is a very practical algorithm, for which efficient implementations exist, most notable GMP-ECM [28, 1].

1.3 Discrete logarithm problem

The discrete logarithm problem can be stated in various algebraic structures and with several assumptions about what is known. We give here the most standard definition in a cyclic group.

Definition 5. *Let G be a cyclic group of order N generated by g. Assuming that there exists a polynomial time algorithm for computing the group law in G, and that N is given, the discrete logarithm in G is as follows: given any element h in G, compute an integer x such that $h = g^x$.*

A naïve algorithm would try all possible values for x in turn, each new value requiring one application of the group law. The time complexity of this algorithm is bounded by N, i.e. exponential in the size of the input problem. In fact, to speak properly of the size of the input problem, and therefore of the complexity, it would be required to bound the sizes of the representations of elements of G. Since the main topic of this lecture is specific groups for which the size of the representations are easily controlled, we will not mention this issue any more and always take $\log N$ as input size.

The solution x of a discrete logarithm problem is an exponent for an element of order N; therefore only its value modulo N makes sense, and it is customary to view it, not as an element of $\mathbb{Z}$, but as an element of $\mathbb{Z}/N\mathbb{Z}$. Then, the Chinese Remainder Theorem readily comes to mind in order to split the discrete logarithm problem modulo N into simpler problems modulo the prime factors of N.

This simple strategy, combined with an Hensel-lift method to handle powers of prime is known as the Pohlig-Hellman algorithm that we describe now. We therefore assume that the factorization of $N = \prod p_i^{e_i}$ is known. For any j, we start by raising g and h to the power $N/p_j^{e_j} = \prod_{i \neq j} p_i^{e_i}$. We obtain g' of order $p_j^{e_j}$ and h' an element of the group generated by g'. It is easy to check that the discrete logarithm of h' with respect to g' is exactly $x \mod p_j^{e_j}$, where x was the discrete logarithm of h w.r.t. g. Since exponentiation can be done in polynomial time using a basic square-and-multiply technique, this gives a polynomial time reduction of the discrete logarithm problem modulo N to discrete logarithm problems modulo prime powers dividing N. The Hensel-like trick works similarly by projecting to appropriate subgroups of prime order, and is left as an exercise for the reader. For this topic and much more, we recommend to read Galbraith's book[11].

Theorem 4 (Pohlig-Hellman). *Let G be a cyclic group of order N of known factorization $N = \prod p_i^{e_i}$. The discrete logarithm problem in G can be reduced in polynomial time to, for each i, solving e_i discrete logarithm problems in subgroups of G of order p_i.*

The previous result is typical of what is called a generic algorithm, i.e. an algorithm that works in any group for which we have an efficient algorithm for the group law. In fact, it is possible to go further, while still staying in the realm of generic algorithms.

The **Baby-step giant-step** algorithm, works as follows. Let us write the discrete logarithm x as $x = x_0 + \lceil\sqrt{N}\rceil x_1$, where $0 \leq x_0, x_1 < \lceil\sqrt{N}\rceil$. In a first phase, we compute all the possible values for hg^{-x_0} and store them in a data structure that allows fast searching, together with the corresponding value of x_0. In the second phase, all values $g^{x_1\lceil\sqrt{N}\rceil}$ are computed, and each time we check whether the group element belongs to the list of elements computed in the first phase. When this is the case, we have $hg^{-x_0} = g^{x_1\lceil\sqrt{N}\rceil}$, so that $h = g^{x_0+x_1\lceil\sqrt{N}\rceil}$, and we deduce the solution from x_0 and x_1. The time and space complexity of this algorithm is in $O(\sqrt{N})$, where maybe a logarithmic factor should be added to take into account the overhead of the data structure queries.

Combining the Baby-step giant-step and the Pohlig-Hellman algorithms, the discrete logarithm problem in a group of order N can be solved in time and space $\tilde{O}(\sqrt{p})$ where p is the largest prime factor of N. There exists also another algorithm, with the same time complexity, but with essentially no memory. This is the Pollard Rho algorithm for which their exist numerous variants. They also have the advantage to be parallelizable with almost no communications between nodes, and almost no overhead [27]. Their drawback is that they can not be rigorously analyzed without assumptions on the existence of hash functions with nice properties. Cryptographers would use for this the "random oracle model", where the runtime analysis is made on average on all the hash functions. In practice the variants of Pollard Rho algorithm are very efficient and the most basic choices of hash functions are enough.

For our purpose, the last thing that is worth mentioning about generic algorithms is that they are essentially optimal. Indeed, Nechaev and then Shoup [19, 25] have proven that any probabilistic algorithm that can solve the discrete logarithm with a non-negligible success probability in a group of prime order p requires to perform at least $\Omega(\sqrt{p})$ group-operations.

Of course, generic groups do not exist in practice: in practice, the group law is performed with an algorithm that relies on a structured representation of the elements. A discrete logarithm algorithm is therefore "free" to make operations on the elements that are not group law operations, and the lower bound does not apply.

The most classical groups in cryptography, namely multiplicative groups of finite fields, and group of points of an elliptic curve in a finite field, behave very differently. For elliptic curves over prime fields, apart from a few particular cases that are easy to avoid, we do not know a better discrete logarithm algorithm than the generic algorithms. They are as close as generic groups as they can be. For finite fields, this is another story, as we will see in the next section.

2 Combining congruences

2.1 Basic subexponential factoring algorithms

Many factorization algorithms start from the following observation: if X and Y are such that $X^2 \equiv Y^2 \mod N$ in a non-trivial manner, i.e. with $X \not\equiv \pm Y \mod N$, then $\mathrm{GCD}(X - Y, N)$ gives a proper factor of N. Let us assume that N has only two prime factors $N = pq$, and let us

furthermore assume that X being fixed, Y is chosen uniformly among all possible values that satisfy $X^2 \equiv Y^2 \mod N$. Reducing the equation modulo p, we get that $X \equiv \pm Y \mod p$, and the same holds modulo q. Using the Chinese Remainder Theorem (CRT), it is then clear that there are 4 choices of Y, and among them, 2 will give a proper factorization, and 2 will give nothing. This 50% chance of success can only improve if N has more than 2 factors.

We remark readily that, although there are efficient algorithms to compute square root in a finite field, there are no such algorithm to extract a square root in the ring $\mathbb{Z}/N\mathbb{Z}$, with a composite N: the best known strategy is to factor N and use the CRT.

Let us now describe one of the most basic strategy to generate two numbers with congruent squares modulo N. Let x be a random number modulo N. Let us compute $x^2 \mod N$, viewed as an integer between 0 and $N-1$. The chances that this is a perfect square are low. So, instead, we are going to hope for this number to be B-smooth, for a smoothness bound B that will be tuned later. With the ECM algorithm, the test costs $B^{o(1)}$. We perform this procedure many times, until we get a collection of **relations**: numbers x_i's for which

$$x_i^2 \equiv \prod_{p<B} p^{e_{p,i}} \mod N,$$

where the product goes through all the primes smaller than $p < B$, and the exponents $e_{p,i}$ are non-negative integers, most of them being 0. Let us form the matrix M whose columns are labelled by the primes p, and for which the i-th row contains the exponents $e_{p,i}$ of the relation corresponding to x_i. The goal will be to create a square, that is an integer for which all the exponent in its prime decomposition are even. Hence the matrix that we form needs only to encode the parity of the exponents, and not the exponent themselves. In other words, the matrix M is viewed as a matrix over the field $\mathbb{F}_2$ with two elements.

If there are more rows than columns in M, there exists a non-zero left-kernel vector v_i. Then, by construction, multiplying together all the relations for which $v_i = 1$, we get an equality between two squares modulo N: on the left-hand-sides, this is a product of the x_i^2, and on the right-hand-side we have a number with only even exponents in the prime decomposition. From this, we deduce two numbers X and Y such that $X^2 \equiv Y^2 \mod N$, and we hope that they yield a proper factorization.

The runtime of this algorithm is clearly linked to the choice of the smoothness bound B. If this is large, then smooth elements are frequent, but we will need more relations to fill-in the matrix. If this is small, we need less relations, but they are hard to find. By choosing B of the form $L_N(1/2, \beta)$, we can get a perfect balance and optimize the overall cost. By the prime number theorem, the number of primes below B is around $B/\log B$, which we bound crudely by B, since polynomial factors are not visible with the L notations. The probability that a number smaller than N is B-smooth is $L_N(1/2, 1/2\beta + o(1))^{-1}$. Therefore collecting the relations has an expected cost of $L_N(1/2, \beta + 1/2\beta + o(1))$. This is optimized by taking $\beta = \sqrt{2}/2$, and gives a runtime of $L_N(1/2, \sqrt{2} + o(1))$. Since the matrix is sparse, the complexity of finding a kernel element is quasi-quadratic, that is $L_N(1/2, 2\beta)$. This is again the same value, so that the overall runtime of the algorithm is $L_N(1/2, \sqrt{2} + o(1))$.

In the algorithm that we described above, the complexity is driven by the size of the elements we test for smoothness (which, in turn, determines the smoothness bound). There is an easy way to reduce this size from roughly N to $\sqrt{N}$. Indeed, instead of taking random values modulo N for the x_i's, it is possible to take $x_i = \lceil \sqrt{N} \rceil + \varepsilon_i$, for small random values of ε_i. In that case,

$x_i^2 - N \approx \sqrt{N}$, and therefore the smoothness test is done on an integer that has a size half times smaller than before. Re-tuning β according to this new setting, we get an overall complexity of $L_N(1/2, 1 + o(1))$.

There was a fair amount of hand-waiving in the description and in the analysis of these sub-exponential factorization algorithms. In fact, turning them into rigorously proven versions is not at all a trivial task. A variant with an $L_N(1/2, \sqrt{2} + o(1))$ complexity was proven by Pomerance in 1987 [22], and a proven complexity of $L_N(1/2, 1 + o(1))$ was obtained in 1992 by Lenstra and Pomerance [18], but this required to use rather more sophisticated algebraic structures, namely class groups of number fields.

This is in fact a general feature of most of the efficient algorithms for factorization and discrete logarithm: getting a rigorous proof of the expected running time is often pretty hard or even far beyond the current knowledge in number theory. As a consequence, in many cases we have to content ourselves with an analysis based on heuristics.

Following the general strategy of combining congruences to construct a modular equality between two squares, the algorithm with the currently best known complexity is the Number Field Sieve (NFS). It was invented in the early 90's, starting with an idea by Pollard for a small class of numbers, and generalized by many authors to make it applicable to any number [16]. The heuristic complexity of the NFS algorithm is

$$L_N\left(1/3, \sqrt[3]{64/9} + o(1)\right),$$

the important improvement compared to previous algorithm being the replacement of the $1/2$ by a $1/3$ in the formula.

We will not explain the NFS algorithm here, but in Section 4 in the context of discrete logarithm computations in prime fields.

2.2 Combining congruences for discrete logarithm

The strategy of combining congruences using smoothness properties can be used for discrete logarithm computations. We describe it first for prime fields, i.e. finite fields of the form $\mathbb{F}_p$, where p is a prime. More precisely, the discrete logarithm problem is defined in the multiplicative group of $\mathbb{F}_p$, and we assume that we work in a prime order subgroup of order $\ell | p - 1$. Let g be a generator of this subgroup, and let h be an element in $\langle g \rangle$. In general ℓ is assumed to be large enough so that any event that occurs with probability $1/\ell$ is considered unlikely. The algorithm proceeds in 3 main phases:

1. Collect relations between small elements.
2. With sparse linear algebra, compute the logarithms of all the small elements.
3. Find a relation between the target h and the small elements.

The first two phases do not depend on h and therefore they can be seen as a precomputation if many discrete logarithms have to be computed in the same field. This works as follows. Let a be a random integer, and compute g^a in $\mathbb{F}_p^*$. The finite field $\mathbb{F}_p$ can be represented as $\mathbb{Z}/p\mathbb{Z}$, so that its elements can be viewed as integers between 0 and $p - 1$. Hence, it makes sense to check whether g^a is B-smooth as an integer less than p. Again the bound B will be fixed during the complexity

analysis. If it is smooth, we get a relation. And we collect many of them: we assume that we have found a collection of integers a_i's such that we can write

$$g^{a_i} \equiv \prod_{q<B} q^{e_{q,i}} \mod p,$$

where the product goes through all the primes smaller than B, and the exponents $e_{q,i}$ are non-negative integers, most of them being 0. By taking logarithms, each of this equation becomes a linear equation between discrete logarithms modulo ℓ:

$$a_i \equiv \sum_{q<B} e_{q,i} \log q \mod \ell.$$

The a_i's and the exponents $e_{q,i}$ are known, so that the only unknowns are the $\log q$, for $q < B$. If we get enough relations, we can hope to get a linear system of maximal rank, so that all the $\log q$ are uniquely determined and can be computed with (sparse) linear algebra.

Let us insist a bit on a subtle point: a small element $q < B$ whose discrete logarithm occurs in a relation has no reason at all to belong to the subgroup generated by g. This is an element of the full multiplicative group $\mathbb{F}_p^*$, and if the order ℓ of g is much smaller than $p-1$, then the chances are quickly negligible that we are actually writing an equation between elements of $\langle g \rangle$. Still, the equations that we wrote above are correct. Indeed, the first one is an equation between field elements, and the second one can be deduced by projecting a similar equation between discrete logarithms in the full group $\mathbb{F}_p^*$, i.e. a linear equation modulo $p-1$. But ℓ divides $p-1$, so this equation also makes sense modulo ℓ, even if the elements involved are not in the subgroup.

Once the logarithms of the small elements have been computed by linear algebra, we look for another relation, but this time involving the target element h: several values of a are tried until h^a, seen as an integer between 0 and $p-1$, is B-smooth. When this is the case, then $a \log h$ can be expressed as a linear combination of known logarithms, and provided that a is not divisible by ℓ, we deduce the value of $\log h$.

The complexity analysis is very similar with the one we sketched in the context of integer factorization. By setting the smoothness bound B to $L_p(1/2, \sqrt{2}/2)$, the probability that an element is smooth is in $L_p(1/2, \sqrt{2}/2 + o(1))^{-1}$, so that the total cost for getting a square matrix is in $L_p(1/2, \sqrt{2} + o(1))$. Again, since the matrix is sparse, we can assume a quadratic running time, and this fits in the same complexity.

Let us remark that this algorithm can be made rigorous, as shown by Pomerance [22], after modifying a bit the generation of relations to guarantee that we get a matrix of maximal rank with high probability.

We also insist again on the fact that the algorithm has to manipulate elements of the full multiplicative group and can not stay inside the subgroup of order ℓ. This is the main reason why the complexity analysis depends mostly on p and not on ℓ.

The Number Field Sieve algorithm, originally designed for integer factorization, has been adapted in the 90's to the computation of discrete logarithms in prime fields [14, 23]. The complexity ended up being of the same form: $L_N\left(1/3, \sqrt[3]{64/9} + o(1)\right)$, thus strengthening the links between discrete logarithm and factorization (although there is no known reduction from one problem to the other). Much more information on this algorithm will be given in Section 4.

For finite fields of small characteristic, combining congruences can also be used to compute discrete logarithms. This time, however, the smoothness of integers must be replaced by the smoothness of polynomials. For simplicity, we consider the case of fields of characteristic 2, which can be represented in the form $\mathbb{F}_{2^n} = \mathbb{F}_2[t]/\varphi(t)$, where $\varphi(t)$ is an irreducible polynomial of degree n in $\mathbb{F}_2[t]$. Hence, any element can be uniquely represented by a polynomial of degree less than n. The multiplicative group $\mathbb{F}_{2^n}^*$ is cyclic of order $2^n - 1$, and we consider a discrete logarithm problem in a subgroup of prime order $\ell | 2^n - 1$, generated by an element g.

The algorithm follows the same three steps as in the prime field case. First, we collect relations by raising g to a random power a, until g^a is b-smooth. In that case, this means that, viewed as a polynomial over $\mathbb{F}_2$, the elements have irreducible factors of degree at most b. Tuning b correctly, this occurs with a high enough probability, so that we can generate many relations, which can be converted into a system of linear equations where the unknowns are the logarithms of the elements represented by an irreducible polynomial of degree at most b. One important difference with the factorization and prime field case is that testing the smoothness is much easier: this boils down to factoring univariate polynomials over $\mathbb{F}_2$, which can be done in polynomial time, and very efficiently in practice.

The main remaining question is about the probability that a polynomial of degree less than n is b-smooth. This is given by the following general result, which is an equivalent of the theorem of Canfield–Erdős–Pomerance.

Theorem 5 (Panario – Gourdon – Flajolet [20]). *Let $N_q(n, m)$ be the number of monic polynomials over $\mathbb{F}_q$, of degree n that are m-smooth.*

Then we have

$$N_q(n, m)/q^n = u^{-u(1+o(1))},$$

where $u = n/m$.

The direct consequence is that by setting $b = \log_2 L_{2^n}(1/2, \sqrt{2}/2)$, the probability of getting a b-smooth element is in $L_{2^n}(1/2, \sqrt{2}/2 + o(1))^{-1}$, so that the analysis is exactly the same as in the prime field case, and the final complexity is again in $L_{2^n}(1/2, \sqrt{2} + o(1))$.

This concludes our description of basic algorithms using the combination of congruences. We are now ready to zoom-in on two more recent topics, in order to highlight links with symbolic computation.

3 Discrete logarithms in finite fields of small characteristic

In 2013, many improvements were discovered for discrete logarithm computation in the case of finite fields of small characteristic, starting with an algorithm with an $L_{2^n}(1/4)$ complexity due to Joux [15], that was then modified to get a quasi-polynomial complexity [7]. By quasi-polynomial, we mean a complexity of $n^{O(\log n)}$ where n is the number of bits of the input, which would correspond to $L_{2^n}(o(1))$ in the L-notation.

In this lecture, we present these algorithms in reverse-chronological order.

3.1 The BaGaJoTh quasi-polynomial algorithm

The canonical setting for the quasi-polynomial algorithm is unusual, since we require that the finite field is of the form $\mathbb{F}_{q^{2k}}$, where k is less than or equal to $q + 2$, but still close to q, so that in

the complexity estimates we can replace k by q without losing too much. The size of the input is $\log(q^{2k}) \approx q \log q$. Therefore, any algorithmic step whose complexity is in $q^{O(1)}$ is polynomial time, and can therefore be counted as one unit of time in the analysis, since in the end we will get something that is quasi-polynomial, i.e. $q^{O(\log q)}$. Also, in the following, we will freely assume that the discrete logarithms of the elements of $\mathbb{F}_{q^2}$ have been precomputed (in polynomial time).

The starting point of the algorithm is to choose an appropriate representation for the field that we call a "sparse medium subfield representation": we assume that there exist two quadratic polynomials h_0 and h_1 in $\mathbb{F}_{q^2}[X]$ such that $h_1(X)X^q - h_0(X)$ has an irreducible factor $\varphi(X)$ of degree k. Although we know no proof for the existence of such a representation, heuristic arguments and practical experiments for small cases suggest that if they are exceptions, they must be quite rare.

We now describe a method that allows to express the discrete logarithm of any element in terms of discrete logarithms of "smaller" elements. Here, the size of an element is its degree, assuming that the element is represented as a polynomial over $\mathbb{F}_{q^2}[X]$, modulo $\varphi(X)$. Let $P(X)$ be such an element. We start with the main equation of the finite field $\mathbb{F}_q$:

$$X^q - X = \prod_{\alpha \in \mathbb{F}_q} X - \alpha, \tag{1}$$

which is valid as a polynomial equation in $\mathbb{F}_{q^2}[X]$. For any quadruple (a, b, c, d) of elements in $\mathbb{F}_{q^2}$, we replace X by $(aP + b)/(cP + d)$ in the equation. It is easier to write things projectively, and we get

$$(aP + b)^q(cP + d) - (aP + b)(cP + d)^q = \prod_{(\alpha:\beta) \in \mathbb{P}^1(\mathbb{F}_q)} (-c\alpha + a\beta)P - (d\alpha - b\beta),$$

where the set of representatives $(\alpha : \beta)$ for $\mathbb{P}^1(\mathbb{F}_q)$ is chosen so that no correcting factor is needed. We will study both sides of the equation in turn.

Left-hand-side. This part of the equation has high degree, but using the sparse representation of the field extension, we can rewrite it as an equation of smaller degree modulo φ. Indeed, by q-linearity, $(aP + b)^q = (a^q P^q + b^q)$, and $P(X)^q = \tilde{P}(X^q)$, where $\tilde{P}$ is the same polynomial as P, where all the coefficients have been raised to the power q. In particular, $\tilde{P}$ has the same degree as P. Now, modulo $\varphi(X)$, we can rewrite X^q as $h_0(X)/h_1(X)$, hence

$$(aP + b)^q \equiv a^q \tilde{P}(h_0/h_1) + b^q \mod \varphi.$$

After multiplication by $h_1^{\deg P}$, the denominator gets clear, and the expression becomes a polynomial of degree $2 \deg P$. Doing this also for $(cP + d)^q$, we see finally that the left-hand-side can be rewritten, modulo φ, as a polynomial of degree $3 \deg P$ divided by a large power of h_1. Assuming that it behaves like a random polynomial of the same degree, it splits in polynomials of degree at most $\frac{1}{2} \deg P$ with a constant probability.

Right-hand-side. This part of the equation does not need any transformation: we just remark that all the factors are, up to multiplication by an elements of $\mathbb{F}_{q^2}$ (for which the discrete logarithms have been precomputed), translates of P by an element of $\mathbb{F}_{q^2}$. More precisely, the set of translations is given by the image of $\mathbb{P}^1(\mathbb{F}_q)$ by the inverse of the homography described by the matrix $m = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$.

The algorithm then proceeds as follows: for all possible quadruple (a, b, c, d), we check whether the left-hand-side is $\frac{1}{2}\deg P$-smooth. If yes, we collect this as a relation. The right-hand-sides of all the collected relations involve only translates of P, and there are only q^2+1 of them (projectively). Hence, if we get more than that number of relations, we can hope to be able to eliminate all these unknowns except for the one we are interested in, namely P itself. Since the left-hand-sides involve only polynomials of degree at most $\frac{1}{2}\deg P$, we obtain an equation relating P to elements of degree half of its size.

Is there any chance to get enough relations? At first sight, there seems to be q^8 choices for (a, b, c, d). However, some of them yield useless relations, and many of them yield the same relation. In the end, this is not too difficult to check that the matrix $m = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ should visit only one representative in each coset of $\mathcal{P} = \mathrm{PGL}_2(\mathbb{F}_{q^2})/\mathrm{PGL}_2(\mathbb{F}_q)$ to avoid obvious duplicates. Since the cardinality of $\mathcal{P}$ is q^3+q and since the probability of getting a smooth left-hand-side is constant, we expect to get $\Theta(q^3)$ relations, so that there is a fair chance that the matrix with only q^2+1 columns has full rank. We will assume that this is indeed the case.

All the operations we have just mentioned take a time complexity within $q^{O(1)}$: they are polynomial-time. It remains to call this building-block recursively, and to estimate the size of the recursion tree. The depth of the tree is the logarithm of the degree of the initial P, that is itself bounded by $k \approx q$. The arity of the tree is bounded by the number of factors we have in the left-hand-side times the number of relations we have to combine to eliminate the translates of P. This is within $q^{O(1)}$ (for a constant in the $O()$ that is the same all along the tree). Therefore, the number of nodes in the tree of arity $q^{O(1)}$ and depth $\log q$ is $q^{O(\log q)}$, that is quasi-polynomial.

One last comment is about the end of the recursion. Getting the logarithms of the linear factors is done with the same strategy of collecting relations with the action of $\mathcal{P} = \mathrm{PGL}_2(\mathbb{F}_{q^2})/\mathrm{PGL}_2(\mathbb{F}_q)$. The only difference is that the linear algebra step will involve an homogeneous system instead of an inhomogeneous one. We skip the details and refer to the paper.

How can we handle finite fields of small characteristic that are not in the appropriate shape? Let $\mathbb{F}_{p^n}$ be the field in which we want to compute discrete logarithms, where p is small, i.e. $p < n$. If $p \approx n$, then we can set $q = p$ and $k = n$, embed our discrete logarithm problem into $\mathbb{F}_{q^{2k}} = \mathbb{F}_{p^{2n}}$ which is only twice as large, and therefore the quasi-polynomial complexity will still hold. The surprising thing is that this strategy of embedding in a larger field also preserves the quasi-polynomial complexity when p is much smaller than n. Consider the worst case, where $p = 2$ and n is prime. Let q be the smallest power of 2 that is larger than n and set $k = n$. The initial field cardinality is $Q = 2^n$, while the BaGaJoTh algorithm is run in a field of cardinality $2^{2n\lceil \log n \rceil}$. In that case, the time complexity is $n^{O(\log n)}$, that is also quasi-polynomial in the original input size.

3.2 In practice: Joux's $L(1/4)$ descent based on Gröbner basis

The main drawback of the BaGaJoTh algorithm is that the arity of the descent tree is very large (on the order of q^2), and therefore it does not behave well in practice. Joux's $L(1/4)$ algorithm, that was invented before the BaGaJoTh algorithm, is a nice way to circumvent this difficulty. This works well only at the bottom of the descent tree, when the degree of the polynomial P is less than $\sqrt{q}$.

The setting is the same as before: we assume that the finite field $\mathbb{F}_{q^{2k}}$ is represented by $\mathbb{F}_{q^2}[X]/\varphi(X)$, where φ is an irreducible factor of degree k of $h_1(X)X^q - h_0(X)$, with h_0 and h_1 two

quadratic polynomials over $\mathbb{F}_{q^2}$.

Let P be an irreducible polynomial of degree D over $\mathbb{F}_{q^2}$ the logarithm of which we want to express in terms of logarithms of polynomials of smaller degrees. The strategy is to look for a rational fraction $k_1(X)/k_2(X)$ to plug into Equation 1: we wish to enforce the left-hand-side to be divisible by P, while keeping k_1 and k_2 of sufficiently small degrees so that both sides can be hoped to be smooth for a smoothness bound yet to be determined, but that should be smaller than D.

Let us start from Equation 1 where we have replaced X by $k_1(X)/k_2(X)$ for two polynomials k_1 and k_2 over $\mathbb{F}_{q^2}$ of respective degrees d_1 and d_2. After clearing denominators, we obtain

$$k_1(X)^q k_2(X) - k_1(X)k_2(X)^q = \prod_{(\alpha:\beta)\in\mathbb{P}^1(\mathbb{F}_q)} \beta k_1(X) - \alpha k_2(X).$$

The right-hand-side raises no difficulty: it comes readily in a factored form, with factors of degree at most $\max(d_1, d_2)$. On the left-hand-side, we will use the plain expression to write a system of equations that encode the divisibility by P, while using the equality $h_1(X)X^q - h_0(X) = 0$, we will be able to control its degree.

Modeling divisibility by a bilinear system. In a finite field that is an extension of $\mathbb{F}_q$, raising to the power q is a q-linear operation. In order to make it explicit, we write $\mathbb{F}_{q^2}$ with a polynomial basis over $\mathbb{F}_q$:

$$\mathbb{F}_{q^2} = \mathbb{F}_q[t]/(t^2 + \tau_1 t + \tau_0),$$

where τ_0 and τ_1 are elements of $\mathbb{F}_q$ such that the polynomial $t^2 + \tau_1 t + \tau_0$ is irreducible. Let us write the coefficients of k_1 and k_2 on this basis:

$$\begin{aligned} k_1(X) &= a_0 + a_1 X + \cdots + a_{d_1} X^{d_1} \\ &= (a'_0 + a''_0 t) + (a'_1 + a''_1 t)X + \cdots + (a'_{d_1} + a''_{d_1} t)X^{d_1}, \\ k_2(X) &= b_0 + b_1 X + \cdots + b_{d_2} X^{d_2} \\ &= (b'_0 + b''_0 t) + (b'_1 + b''_1 t)X + \cdots + (b'_{d_2} + b''_{d_2} t)X^{d_2}. \end{aligned}$$

The product $k_1(X)^q k_2(X)$ can then be expanded: any coefficient in X of this product is a sum of terms of the form $(a'_i + a''_i t)^q (b'_j + b''_j t)$. By q-linearity, this can be rewritten as $(a'_i + a''_i t^q)(b'_j + b''_j t)$. This expression is a polynomial in t, where each coefficient is a linear combination of monomials AB, where A is one of the a'_i or a''_i, and B is one of the b'_j or b''_j. This polynomial in t can be reduced modulo $t^2 + \tau_1 t + \tau_0$, and this preserves the bilinear nature of its coefficients. At the end of the line, $k_1(X)^q k_2(X)$ is a polynomial in X whose coefficients are linear polynomials in t over the polynomial ring over $\mathbb{F}_q$ with all the a'_i, a''_i, b'_j, b''_j as indeterminates, and these coefficients are bilinear expressions with the two sets of indeterminates $\{a'_i, a''_i\}$ on one side, and $\{b'_j, b''_j\}$ on the other side.

The same is true for the product $k_1(X)k_2(X)^q$, so that it is also true for their difference, which is the left-hand-side that we want to be divisible by P. This divisibility can be expressed by $D = \deg P$ linear conditions over $\mathbb{F}_{q^2}$, involving the coefficients of the left-hand-side. Since a linear combination of bilinear expressions is still a bilinear expression, we obtain the key to the efficiency of Joux's algorithm: the divisibility by P can be encoded by a system of bilinear equations.

Degrees of freedom. The number of $\mathbb{F}_{q^2}$-conditions is D, and therefore, the number of $\mathbb{F}_q$-bilinear equations is $2D$. The number of indeterminates is $2d_1 + 2$ for k_1 and $2d_2 + 2$ for k_2. As usual, there are some redundancies that must be taken into account. In this case, this amounts to first forcing one of the polynomials to be monic, and second to ensure that one of the coefficients (say, the leading coefficient of the second one) is set to a fix element of $\mathbb{F}_{q^2} - \mathbb{F}_q$. The remaining number of unknowns is then $2d_1 + 2d_2$. Finally, we need a bit of margin, in order to have several solutions among which one is expected to yield a relation. Hence the degrees of k_1 and k_2 are chosen with the following constraint:

$$d_1 + d_2 = D + 1.$$

Algorithm and complexity analysis. The algorithm proceeds as follows: for a given P of degree D, we write the system of bilinear equations over $\mathbb{F}_q$ encoding the fact that two polynomials k_1 and k_2 yield an equation for which P divides the left-hand-side. This bilinear system can be solved using Gröbner basis techniques. When a solution is found, the left-hand-side for the corresponding (k_1, k_2) pair can be rewritten using the equation $X^q = h_0(X)/h_1(X)$. After clearing denominators, this gives a polynomial of degree $d_1 + d_2 + \max(d_1, d_2) = D + 1 + \max(d_1, d_2)$, that is at most $2D$. We test its smoothness for a smoothness bound that is $\max(d_1, d_2)$, in order to reach the same level of smoothness as on the right-hand-side. Under the heuristic that this behaves like a random polynomial, the smoothness probability is constant. If it is not smooth, we look for another solution (k_1, k_2) and try again.

We have therefore explained a building-block that allows to express the logarithm of a polynomial of degree D in terms of logarithms of polynomials of degrees at most $\max(d_1, d_2)$, under the condition $d_1+d_2 = D+1$. If the bilinear system could be solved in polynomial time, then choosing $d_1 \approx d_2 \approx \frac{D}{2}$ would yield another quasi-polynomial time algorithm. This is unfortunately not the case, but recent progress in structured polynomial system solving provide a complexity estimate that is better that for general polynomial systems. In [10], it is proven that for a 0-dimensional affine bilinear system involving n_x and n_y unknowns, the complexity is in

$$O\left(\binom{n_x + n_y + \min(n_x + 1, n_y + 1)}{\min(n_x + 1, n_y + 1)}^{\omega}\right).$$

In our case, we have $n_x = 2d_1$ and $n_y = 2d_2 = 2(D + 1 - d_1)$ and, assuming $d_1 < d_2$, the formula becomes $O\left(\binom{2D+3+2d_1}{2d_1+1}^{\omega}\right)$. From here, it is possible to optimize d_1: taking a smaller value makes the polynomial system solving easier, but since the smoothness bound is then $D - d_1$, we get a tree with a larger depth (the arity is always close to q). In Joux's paper, this Gröbner basis trick is used only for polynomials of degree at most $\sqrt{q}$, and in that case, it can be shown that taking $d_1 \approx \sqrt[4]{q}$ yields the appropriate balance in the costs. Together with a more classical strategy to deal with polynomials of degrees from q to $\sqrt{q}$, this combines into an algorithm with a complexity of

$$L_{q^{2k}}(1/4 + o(1)),$$

which, at the time it was made public was a big surprise in the community.

This algorithm is a very good example where knowing and using the latest results in symbolic computation was necessary to design and analyze a new technique: sticking to the basic worst-case doubly-exponential estimate would end-up in a non-competitive complexity.

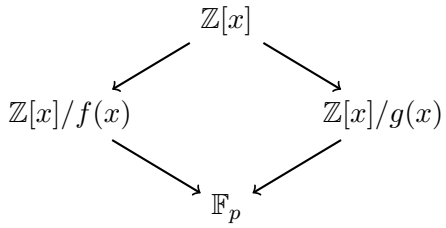
It is also interesting to note that the bilinear systems that occur in this discrete logarithm algorithm are not really random. This could mean that in fact the complexity estimate of [10] that is given for random systems is not valid here. However, we view this as a chance to potentially further optimize the practical resolution.

4 The number field sieve for prime field discrete logarithm

4.1 Overview of the NFS algorithm

The Number Field Sieve algorithm falls in the category of the discrete logarithm algorithms based on combining congruences. As such it follows the 3 main phases listed in Section 2.2: collect relations, solve a linear algebra problem, solve the individual logarithm question. The first 2 phases do not depend on the element whose discrete logarithm is wanted and can be seen as a precomputation if many logarithms must be computed.

The main difference lies in the way the algorithm produces relations. Let p be a prime defining the finite field $\mathbb{F}_p$, and let ℓ be a large prime divided $p-1$. We want to solve the discrete logarithm problem in the subgroup of order ℓ of $\mathbb{F}_p^*$. We start by selecting two irreducible polynomials f and g over $\mathbb{Z}$ with appropriate degrees and coefficients sizes, such that f and g have a common root m modulo p. To facilitate the exposition, we assume furthermore that f and g are monic, but this is not necessary. By construction, the following diagram, where arrows are ring homomorphisms, commutes:



The strategy to produce relations is to plug at the top of the diagram many linear polynomials $a - bx$ and try to smooth them on both paths. Since the algebraic structures on the f- and on the g-sides have a priori nothing to do with each others, this should yield a non-trivial equality at the bottom of the diagram, namely in the target field $\mathbb{F}_p$. Before discussing the notion of smoothness that we are going to use, we explain how f and g can be constructed. They are many strategies for that task, and this is still an active research topic [6]. However, a very basic method gives the appropriate complexity and is enough for our purpose. Let us fix a degree d for the polynomial f and set m to an integer close to $p^{1/d}$. Then the integer p can be written in base m as

$$\begin{aligned} p &= f_0 + f_1 m + \cdots + f_{d-1} m^{d-1} + m^d \\ &= f(m) \ , \end{aligned}$$

where the f_i are bounded by m. With tremendous probability, f is irreducible over $\mathbb{Z}$, and if not, then we try again with another value for m. The polynomial g is just $x - m$, and by construction, f and g have a common root modulo p, namely m. The degree d of f has to be tuned after a complexity analysis that we will not include here. The optimal asymptotic value for d is the closest integer to $(3 \log p / \log\log p)^{1/3}$. This grows slowly to infinity, and for all practical cases considered nowadays, d is 4, 5 or 6, depending on size of p.

Smoothness in number fields. Since g is monic and linear, the ring $\mathbb{Z}[x]/g(x)$ is isomorphic to $\mathbb{Z}$ in which factorization is well defined, and we can use the usual notion of smoothness. On the f-side, however, we need to work in the ring $\mathbb{Z}[x]/f(x)$ which has no reason to be a principal ideal domain, and not even a unique factorization domain. In fact, cases where $\mathbb{Z}[x]/f(x)$ is a UFD are rare enough so that we can consider that they can not occur by accident and we have no way to force this property unless for very particular choices of p.

As a subring of the ring of integers of a number field, $\mathbb{Z}[x]/f(x)$ is a Dedekind domain, i.e. every non-zero ideal factors into a product of prime ideals, and this factorization is unique up to reordering. The norm of an ideal is an integer that drives its factorization as an ideal: the norm of a prime ideal is a prime number or a prime power, and since the norm is multiplicative, by testing the smoothness of the norm of an ideal, one can detect if it is a product of prime ideals of small norms (some care must be taken with powers, but this can be delt with).

Hence, the algorithm proceeds as announced: for many linear polynomials $a-bx$, we send them on both paths of the diagram. On the g-side, it becomes the integer $a - bm$ that we test for B-smoothness in a classical way. If it is smooth, we go on with the f-side and study the ideal generated by $a-b\alpha$, where α is a root of f in the ring $\mathbb{Z}[x]/f(x)$. For this, we compute its norm, which in this case is just the integer $b^d f(a/b)$, and test its B-smoothness in $\mathbb{Z}$. If the norm is smooth, then we can write the principal ideal $(a - b\alpha)$ as a product of prime ideals of norm less than B. Therefore, we have the two factorizations:

$$\begin{aligned} (a - b\alpha) &= \prod_{\mathrm{Norm}(\mathfrak{q})<B} \mathfrak{q}^{e_\mathfrak{q}} \\ a - bm &= \prod_{q<B} q^{e_q}. \end{aligned}$$

Finally, we would like to map those two factorizations in $\mathbb{F}_p$, which is no problem on the g-side, but does not directly make sense on the f-side. Indeed, the ring homomorphism in the diagram can operate on elements, but not on ideals. This difficulty caused many problems during the genesis of the algorithm, in the 90's, until Schirokauer found an elegant way of solving it. The general idea is to convert the equality between ideals into an equality between elements. For this, we need first to make the ideal principals by raising the equation to the cardinality of the class group of the number field $\mathbb{Q}[x]/f(x)$, and then to adjust the contribution of the unit group. However, the polynomial f is in general way too complicated to leave a hope to compute the class group and the unit group explicitly. Schirokauer noticed that we just need to ensure that the equality between elements holds up to ℓ-powers. Indeed, once we have mapped a relation into $\mathbb{F}_p$, we are only interested in taking its logarithm modulo ℓ, and any ℓ-power contributes to 0 in such a setting.

The **Schirokauer maps** are characters from elements of $\mathbb{Q}[x]/f(x)$ coprime to ℓ to $\mathbb{Z}/\ell\mathbb{Z}$, that can be computed in polynomial time, with the property that if all the characters vanish, then a factorization of ideals can be mapped in the finite field with a meaningful sense. Hence modulo the use of Schirokauer maps, it makes sense to talk about the logarithm of an ideal $\mathfrak{q}$ of $\mathbb{Z}[x]/f(x)$ mapped into $\mathbb{F}_p$. This is called a virtual logarithm in the literature [24]. The number of Schirokauer maps (SM) to use is exactly the unit rank as given by Dirichlet theorem: if the polynomial f has r_1 real roots and $2r_2$ complex roots, then we need $r_1 + r_2 - 1$ SMs.

Finally, any $a - bx$ for which the two sides are B-smooth becomes a row of a matrix, where columns are labelled by prime numbers less than B and prime ideals of norm less than B, plus a few columns for the SMs. The entries are the valuations e_q and $e_\mathfrak{q}$ that occur in both factorizations, and the evaluation of the SMs at $a - bx$.

Once we have reached a point where we get enough relations so that the matrix has maximal rank (which is full-rank minus 1), computing a non-zero vector of its right-kernel gives the logarithms of

all the primes below B (and also the virtual logarithms of the prime ideals). We skip completely the last phase of the algorithm which is to rewrite the target element h in terms of these small elements in order to get its logarithm. Although this is less simple as for the basic combination of congruences, this still takes only a small amount of time compared to the rest of the algorithm.

A few words on the complexity. With the choice of $d = \deg f$ already mentioned, the value of m is in $L_p(2/3, c)$, for a constant c that we do not make explicit, because the formulae quickly get ugly. In fact, in this paragraph, we will only write the first parameter in the L-notation. The sizes of a and b can not be decided in advance, because while we do not find enough relations, we need to take larger and larger values for a, b. So we just pretend that we try with a bound in $L_p(1/3)$ for a and b, and we check in the end that this leaves enough room for getting a matrix of maximal rank. On the g-side, the element that we test for smoothness is $a - bm$, which is in $L_p(2/3)$, while on the f-side, we test $b^d f(a/b)$, which is also bounded by an expression of the form $L_p(2/3)$. Now, we set a smoothness bound B of the form $B = L_p(1/3)$. Using Canfield–Erdős–Pomerance's theorem, we can check that on both sides the probability of being smooth is of the form $L_p(1/3)^{-1}$. Hence, after $L_p(1/3)$ trials, we get more relations than unknowns, and we can solve the linear algebra problem, again in time $L_p(1/3)$. All the exponent constants can be made explicit and optimized. This is the typical exercise that is best left to the reader! We just recall the final complexity that we already gave in Section 2.2: $L_p(1/3, \sqrt[3]{64/9} + o(1))$.

Beside the formulae of the analysis, it is legitimate to ask ourselves where did the improvement from $L_p(1/2)$ to $L_p(1/3)$ come from. The most important change is due to the splitting of the smoothness condition into two other conditions. In the basic congruence combination algorithm, in order to find a relation, we need to be lucky enough to get an element of roughly the same size as p that is smooth. By using the two-paths diagram, the condition to get a relation is now to have two elements simultaneously smooth, which sounds harder at first sight, but since those elements are much smaller – they are in $L_p(2/3)$ – this is actually an easier condition.

Can we go further, and get a better complexity, like $L_p(1/4)$? This is of course an important open question to which there is no obvious answer. One idea that did not yet succeed would be to replace the NFS diagram with two sides with a more complicated one, involving more than one indeterminates. Restating the "common root modulo p" condition would probably involve some ideal-theoretic point of view in a multivariate ring of polynomials.

4.2 The linear algebra step

We resume our description of the classical NFS algorithm, and explain with more details what is done in the linear algebra step. Indeed, although this is a simple kernel computation for a sparse matrix defined over a finite field, the specificities of the matrix and the large sizes of the problem we consider in record computations make it interesting or even necessary not to content ourselves with using an existing library as a black-box.

The general strategy is rather classical: we start by a few steps of Gaussian elimination with heuristics trying to keep the matrix as sparse as possible, and then we switch to an iterative method, usually block-Wiedemann. In the factorization and discrete logarithm community, the first step is usually called **filtering**.

In order to fix ideas, we will give some examples (sizes, densities, running times), coming from a record computation that we did in June 2014, for an integer p of 180 decimal digits [8] (we refer

to this as the `p180` example).

Properties of the initial matrix. The matrix, as it comes from the first step of the algorithm is just a set of files containing relations in a format that is close to their number-theoretic meaning. For the `p180` examples, there were about 253 millions (we write 253M) of relations, each of them taking about 80 bytes on average in compressed format. That sums-up to more than 20 GB of data. At this stage, we have not yet computed the 3 Schirokauer maps that are required in this case. On this example, this would add more than 50 GB of data. Since they can be computed on the fly and since they are not needed until we enter the iterative step, it is better to postpone their computation.

Here is the list of the specificities of this input matrix:

- The number of non-zero entries per row is very low (say 20). This is essentially the number of prime factors in the factorizations on each side. The variance for this quantity is also reasonably low: there is no row with only a couple of non-zero entries, nor is there with hundreds of entries.
- Most of the non-zero entries are 1's. For the smallest primes and prime ideals (the heaviest columns), 2's and 3's are also frequent, but they become rare when the primes get larger, since this correspond to a multiplicity in the prime decomposition.
- The density of the columns is highly variable: a column labelled by a tiny prime or prime ideal is dense, while a column labelled by a prime or prime ideal at the limit of the smoothness bound is almost empty (or even completely empty). Sorting the columns by the size of the prime or ideal they represent gives a gradient of density, from very dense on the left part of the matrix to very sparse on the right part.
- There is no structural property that can be expected a priori (no natural domain decomposition).
- There are rows that occur two or more times. This is due to the way the relations are produced with so-called special-q strategy. On the `p180` example, there were only 175M unique relations, so there were about 30% of duplicates.
- There are empty columns. To be more precise, in a relation, we store information about the primes and prime ideals, but the correspondence between these prime and prime ideals and column indices is not yet done. This correspondence is not done a priori, because some primes occur in no relation.
- There is a lot a redundancy: usually we collect much more rows than what we expect to be needed for getting maximal rank. Indeed, heuristically, this will allow the filtering step to produce a better matrix.

Filtering, interpreted as operations on sparse matrices. Let M_0 be the input matrix (that is stored as raw relations, in various files). The first step is to remove duplicate relations. Although it is an easy task in theory, the software implementation should be robust enough to handle a large amount of data, if possible working as a set of filters that take files or pipes in input and output, and use as less memory as possible. This is rather standard "big data" processing. In terms of

matrices, this amount to finding a matrix S_0, an identity matrix with some missing rows, such that $M_1 = S_0 M_0$ has no duplicate rows. On the `p180` example, the matrix M_0 has 175M rows and 82M non-empty columns.

The second step is to identify empty columns, and columns with only a single non-zero entry. Indeed, in this case, the row and the column corresponding to this singleton entry can be removed (or more precisely, kept for later use). This removal can generate new empty columns or singletons, so that the process should be repeated until we have reached a point where all columns have at least weight 2. This part of the filtering requires to keep efficient data-structures for the matrix, so that searching for columns of smallest weights should be quick, and everything should be updated easily after a row removal. This is again a rather standard searching-sorting problem, but here each and every byte should be saved, since this is a point where we need most of the information in central memory. The particular shape of the matrix can be used, here. Usually, this singleton-removal phase of the filtering is performed only on the sparse part of the matrix: the heaviest columns are not stored in memory. In terms of matrix operations, this step is a first step of row-echelon computation where we select pivots that are alone on their column, and stop when there is no such event. So, we compute P_1 and Q_1 two permutations such that $M_2 = P_1 M_1 Q_1$, and M_2 is a block matrix of the form $\begin{pmatrix} A_2 & B_2 \\ 0 & C_2 \end{pmatrix}$, where A_2 is in row-echelon form. In fact, the matrix A_2 is very close to the identity matrix. Finding a kernel vector of the initial matrix is therefore reduced to finding a kernel vector of C_2. On the `p180` example, the matrix C_2 has 171M rows and 78M columns.

The third-step of the filtering is improperly called clique-removal. Due to the redundancy, we expect C_2 to have much more rows than columns, whereas, with a reasonable heuristic, a square matrix should be enough to get the appropriate rank. Therefore, we can remove some rows. The goal of this step is to select the rows that we remove, so that we maximize our chances to play again the singleton-removal game, and further reduce the size of the matrix. The strategy is as follows: we build a graph, where the nodes are the rows. For each column of weight 2, we add a vertex between the two rows that contribute entries for this column. Then we compute a decomposition in connected components for this graph, and select the heaviest component. Removing one row from this component, will generate a singleton in the column that was of weight 2, so that the rows directly connected to it can be removed, and this propagates to the whole component. Therefore, for each excess row that we have, we can remove one connected component. This heuristic strategy works pretty well, and can be improved by adding further information in the graph; typically, it is interesting to select the component with not only the largest number of nodes, but also the ones for which the nodes correspond to heavy rows, or row that have entries in low-weight columns, so that this increases the chance to create more weight-2 columns. In terms of implementation, this step is not difficult from a theoretical point of view (the graph-theoretic part is very basic), but again, we are at a stage where the memory footprint is problematic: no shortcoming in the way the data structures are handled can be accepted. In terms of matrix operation on C_2, this boils down to a combination of row deletion and row-echelon form computation, again based on pivot that are alone on their column. So this means that we find a row-removal matrix S_2 and permutation matrices P_2 and Q_2 such that the matrix $P_2 S_2 C_2 Q_2$ is a block matrix $\begin{pmatrix} A_3 & B_3 \\ 0 & C_3 \end{pmatrix}$, where A_3 is in row-echelon form and C_3 is a square matrix whose columns have weight at least 2. On the `p180` example, the matrix C_3 has 21M rows and columns.

The final step of the filtering, called "merge", is again a partial row-echelon form computation. Until now, all the pivots were selected so that no pivoting is actually needed, so that the rows were

not combined. In merge, we allow row combinations. The selection of the pivot is now based on heuristics that try to keep the densification of the matrix under control. It is clear that this can not work for long. In order to estimate when to stop, we use the theoretical complexity of iterative methods as a first hint: the cost of the next step is supposed to be proportional to the number of rows/columns times the total weight of the matrix. This quantity can easily be computed during the merge phase. Usually, it is decreasing quickly at the beginning, and at some point, it starts to increase again (unfortunately, this is more chaotic, but the general picture is clear). As for the heuristics used to choose the pivot, these are based again on working with columns with minimal weight. Classical tools from graph theory, like minimal spanning trees are also involved, but we do not enter the details. They are of the same nature as the ones of the previous steps. On the `p180` example, the final matrix after the filtering step has 7.2M rows and columns, with an average of 150 non-zero entries per row.

A final remark on the whole filtering process is that its running time is only a small fraction of the overall running time of a discrete logarithm computation. Indeed, with appropriate data structures, each step can be performed in quasi-linear time. The main issues are the memory footprint, and the quality of the output.

Iterative methods taking SMs into account. We come to the next stage, where we have a square matrix M for which we want to find a non-zero kernel vector, and the matrix is much smaller, but slightly denser that the one we started with. We assume that we have also computed the SM columns for this matrix. The main part of the matrix have only tiny non-zero entries: these are (signed) integers of value less than a hundred. On the other hand the columns of SM are dense columns with entries that are random integer modulo ℓ, the prime modulo which the linear algebra must be performed. Usually ℓ is large and fits from 2 to a dozen of machine words (for the `p180` example, ℓ has almost 600 bits).

The two available methods for this linear algebra step are Lanczos and Wiedemann algorithms. Indeed, any attempt to use a direct method would result in a dense matrix that does not fit in memory, while an iterative method keeps the matrix untouched, with the main operation being a sparse matrix times vector product (SpMV).

We recall briefly Wiedemann's algorithm. Let x and y be random vectors, and consider the sequence $a_i = {}^t x M^i y$ of field elements. It verifies a linear recurrence relation, the minimal polynomial of which $\chi_{M,x,y}$ is a divisor of the minimal polynomial χ_M of the matrix M, so its degree is bounded by N, the dimension of M. It is therefore enough to know $2N$ coefficients of the a_i sequence to be able to compute $\chi_{M,x,y}$, with the rational reconstruction procedure (also known as continued fraction method, lattice reduction in dimension 2, Berlekamp-Massey, or just Euclidean algorithm stopped in the middle) that takes a quasi-linear time in N. Let us assume that $\chi_{M,x,y}$ is exactly χ_M. Since M has a non-trivial kernel (by construction), $\chi_M(X)$ is a polynomial divisible by X, and we write $\chi_M(X) = X^k P(X)$, where k is the valuation of χ_M. For a random vector v, we then compute $w = P(M)v$, that we assume to be non-zero. Then, the smallest integer i such that $M^i w = 0$ is at most k, and is easily computed. Finally the vector $M^{i-1}w$ is a non-trivial kernel vector. This algorithm is probabilistic, and controlling the probability of failure is not so simple. In our context, however, the modulus ℓ is so large that there is no practical problem. And since, on one hand, verifying that the discrete logarithms are correct at the end is easy, and on the second hand, the rest of the NFS algorithm is very heuristic, there is no point in trying to be rigorous here.

Computing the a_i amounts to $2N$ applications of the SpMV operation, while the construction

of the kernel vector from the minimal polynomial requires N of them. Therefore the cost of the algorithm is in $O(N)$ SpMV, each of them having a cost proportional to the weight of the matrix.

Due to the size of the problems we want to consider, parallelism is a most-wanted feature. Using several cores available on a computer is the easiest part. This might be not enough, either for performance reasons, or just because the central memory available on one computing node is not enough to store the matrix and the few temporary vectors. In that case, using parallelism between several nodes connected if possible with a fast network like Infiniband is also quite important. This can be done for instance with a message-passing modelling (and the MPI API). All this is very classical and implemented in most of the linear algebra libraries.

A less classical implementation is a block-version of the Wiedemann algorithm. From the theoretical point of view, this means that the random vectors x and y in the non-block description above are replaced by blocks of m vectors for x and n vectors for y (for small values of m and n). The a_i's are now small $m \times n$ matrices, and the rational reconstruction step becomes much more involved. Also, the construction of the final vector can enjoy the same kind of block-effect. This strategy does not change the number of SpMV that have to be computed. However, it brings another possibility for parallelism, since the n vectors forming y can be processed in parallel, without any interaction.

The concrete consequence is that the costly part of the linear algebra, namely the $3N$ applications of SpMV can be split in n independent computations, with just one point of synchronization after $2/3$ of the total computation, in order to do the rational reconstruction. This rational reconstruction step is still an operation that is quasi-linear in the dimension N of the matrix, but the dependency in the blocking factors m and n is not so nice, so that we can not hope to let m and n go to infinity. Still, there have been progress on this topic that make this step practical for m and n up to a dozen [26, 13].

The other specificity of the matrix we are dealing with is the presence of the 3 columns of SMs. It is obvious that they should not be handled in the same way as the other entries of the matrix that are small integers. There is no canonical way to solve this question, since this is getting too close to the implementation issue on the target platform. We mention just one example that was recently experimented by Hamza Jeljeli during his PhD. When implementing the SpMV operation on a GPU, the RNS representation of integers modulo ℓ is attractive. Indeed, this is some kind of Chinese Remainder Theorem representation where computations take place modulo small primes that fit in one machine word (whatever this means for a GPU that tend to prefer floating point operations). This offers great opportunities for parallelism in a SIMD manner, which is appropriate for the programming paradigm of GPUs. During the SpMV, most of the basic operations are of the type $x \leftarrow x + ay$, where x and y are integers modulo ℓ, while a is a small integer. Therefore, it is possible to have an accumulator that is only one word larger than what is required for ℓ in order to delay the reductions. In RNS representation, this translates into a certain amount of small moduli. When the computation arrives to the SM columns, a typical operation is of the same type, but then a is a full integer modulo ℓ. Therefore, it is necessary to enlarge the RNS representation on-the-fly, so that the accumulator becomes large enough. This demonstrates that optimizing the implementation for the specificities of the matrix can be really intrusive and can hardly be done in a generic library that would be able to handle any matrix, like `LinBox`.

For the `p180` example, the matrix was slightly too large to fit in the memory of the graphic cards we had access to, so we fell back to a CPU implementation. We used $m = 24$ and $n = 12$ as blocking factors, meaning that 12 sequences were run in parallel, each of them using 4 nodes with

16 cores each, with an FDR Infiniband interconnect. The total time for the SpMV part was around 80 core-years. The linear generator phase took 15 hours on 144 cores, which remains negligible.

5 Conclusion

In this lecture, we have explained the basic ideas behind modern algorithms for integer factorization and discrete logarithms in finite fields. We have tried to emphasize a few points where using tools from computer algebra and symbolic computation as black-box was far from optimal.

Polynomial system solving plays an important role in discrete logarithm computation in small characteristic, but this is really just one of its application to cryptography, where it has become an ubiquitous tool for cryptanalysis, both in symmetric and asymmetric cryptography, even more than lattice reduction. Although we had no time to explain it, the case of elliptic curves defined over a finite field of characteristic 2 deserves a special mention, since an algorithm based on combining congruences can be designed in that case, but the smoothing part is replaced by some polynomial system solving. Depending on the properties of these systems, there exist or not a subexponential algorithm for the discrete logarithm in that case [9, 21]. The last place, but not the least, where Gröbner basis have not yet found an application is integer factorization.

As for linear algebra, this is an important step in all the algorithms based on combining congruences. Historically, factorization was the topic that was the most interesting for the public key cryptography community, and therefore there was a focus on sparse linear algebra modulo 2. At that time (80's and 90's), there were no efficient library available to solve huge sparse systems over $\mathbb{F}_2$, and a lot of work was done independently of the community that was focusing on linear algebra with floating point computations. In the end, a lot of tools were invented that can certainly be interpreted as (now) standard operations in other communities, like preconditioning. It is also unclear to us whether there are other applications that require solving huge exact linear algebra problems. Who, apart from a cryptanalyst, is ready to spend 6 months of a 1000-core cluster on just a single kernel computation?

Appendix: literature and software

Besides the articles mentioned in the references, here is a list of recommended reading.

- *Mathematics of Public Key Cryptography* by Steven Galbraith (CUP, 2012).

  This is an excellent book on many mathematical aspects of public key cryptography. A long part (5 chapters) is dedicated to integer factorization and discrete logarithm. It includes also a lot of material on algebraic curves and lattices.

- *Prime Numbers: A Computational Perspective* by Richard Crandall and Carl Pomerance (Springer, 2001).

  Despite the title, this book contains many pages on integer factorization, including the Number Field Sieve. This is one of the best reference on the topic, written by both a pure mathematician and a strong implementor.

- *Algorithmic cryptanalysis* by Antoine Joux (CRC, 2009).

A very good source of insight for anyone interested in cryptanalysis. This contains a survey of attacks that use polynomial system solving. Integer factorization and discrete logarithms take also an important place, but of course the latest improvements are not included.

- *The Development of the Number Field Sieve* by Arjen Lenstra and Hendrik Lenstra (Springer, 1993).

  This book collects the articles of the genesis of the Number Field Sieve for integer factorization. This is not only of historical interest, since in many places it contains material that is not covered anywhere else.

We also list some free software that is related to our lecture.

- CADO-NFS. `http://cado-nfs.gforge.inria.fr`

  A complete implementation of the Number Field Sieve for integer factorization, with increasingly more features for discrete logarithm as well.

  It is maybe not the fastest implementation (Kleinjung's siever is faster), but is easier to use and to read. It also contains an efficient implementation of the Block-Wiedemann algorithm.

- GMP-ECM. `http://ecm.gforge.inria.fr`

  This is the reference implementation of the ECM factoring algorithm.

  Although it does not includes the latest improvements (especially for small sizes), this remains a very good implementation, used by well-known computer algebra systems, like Sage or Magma.

- LinBox. `http://www.linalg.org`

  A powerful library for exact linear algebra.

  It contains many functionalities (system solving, determinant, Smith normal form, ...) and support various types of matrices: dense or sparse, over integers, finite fields, rationals. Many advanced linear algebra algorithms have been implemented by their inventors in LinBox.

References

[1] GMP-ECM. Elliptic Curve Method for Integer Factorization, available at `http://ecm.gforge.inria.fr/`.

[2] L. M. Adleman and M.-D. A. Huang. *Primality testing and Abelian varieties over finite fields*, volume 1512 of *Lecture Notes in Math.* Springer–Verlag, 1992.

[3] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. *Annals of mathematics*, pages 781–793, 2004.

[4] A. O. L. Atkin and D. J. Bernstein. Prime sieves using binary quadratic forms. *Math. Comp.*, 73(246):1023–1030, 2004.

[5] A. O. L. Atkin and F. Morain. Elliptic curves and primality proving. *Math. Comp.*, 61(203):29–68, July 1993.

[6] Shi Bai, Cyril Bouvier, Alexander Kruppa, and Paul Zimmermann. Better polynomials for GNFS. Preprint available at `http://www.loria.fr/~zimmerma/papers/sopt-20140905.pdf`.

[7] Razvan Barbulescu, Pierrick Gaudry, Antoine Joux, and Emmanuel Thomé. A heuristic quasi-polynomial algorithm for discrete logarithm in finite fields of small characteristic. In *Advances in Cryptology–EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Comput. Sci.*, pages 1–16. Springer, 2014.

[8] C. Bouvier, P. Gaudry, L. Imbert, H. Jeljeli, and E. Thomé. Discrete logarithms in GF(p) — 180 digits, 2014. Announcement available at the NMBRTHRY archives, item 004703.

[9] Jean-Charles Faugère, Ludovic Perret, Christophe Petit, and Guénaël Renault. Improving the complexity of index calculus algorithms in elliptic curves over binary fields. In *Advances in Cryptology - EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Comput. Sci.*, pages 27–44. Springer, 2012.

[10] Jean-Charles Faugère, Mohab Safey El Din, and Pierre-Jean Spaenlehauer. Gröbner bases of bihomogeneous ideals generated by polynomials of bidegree (1, 1): Algorithms and complexity. *Journal of Symbolic Computation*, 46(4):406–437, 2011.

[11] Steven D Galbraith. *Mathematics of public key cryptography*. Cambridge University Press, 2012.

[12] William F. Galway. Dissecting a sieve to cut its need for space. In *Algorithmic number theory*, volume 1838 of *Lecture Notes in Comput. Sci.*, pages 297–312. Springer, 2000.

[13] Pascal Giorgi and Romain Lebreton. Online order basis algorithm and its impact on the block Wiedemann algorithm. In *International Symposium on Symbolic and Algebraic Computation, ISSAC'14*, pages 202–209. ACM, 2014.

[14] Daniel M Gordon. Discrete logarithms in GF(P) using the number field sieve. *SIAM Journal on Discrete Mathematics*, 6(1):124–138, 1993.

[15] Antoine Joux. A new index calculus algorithm with complexity $L(1/4 + o(1))$ in small characteristic. In *Selected Areas in Cryptography - SAC 2013*, volume 8282 of *Lecture Notes in Comput. Sci.*, pages 355–379. Springer, 2014.

[16] A. K. Lenstra and H. W. Lenstra, Jr., editors. *The development of the number field sieve*, volume 1554 of *Lecture Notes in Math.* Springer Verlag, 1993.

[17] H. W. Lenstra, Jr. Factoring integers with elliptic curves. *Ann. of Math.*, 126:649–673, 1987.

[18] H. W. Lenstra Jr. and C. Pomerance. A rigorous time bound for factoring integers. *J. Amer. Math. Soc.*, 5(3):483–516, 1992.

[19] V. Nechaev. Complexity of a determinate algorithm for the discrete logarithm. *Mathematical Notes*, 55(2):165–172, 1994.

[20] Daniel Panario, Xavier Gourdon, and Philippe Flajolet. An analytic approach to smooth polynomials over finite fields. In *Algorithmic number theory – ANTS III*, volume 1423 of *Lecture Notes in Comput. Sci.*, pages 226–236. Springer, 1998.

[21] Christophe Petit and Jean-Jacques Quisquater. On polynomial systems arising from a Weil descent. In *Advances in Cryptology–ASIACRYPT 2012*, volume 7658 of *Lecture Notes in Comput. Sci.*, pages 451–466. Springer, 2012.

[22] C. Pomerance. Fast, rigorous factorization and discrete logarithm algorithms. In D. S. Johnson, T. Nishizeki, A. Nozaki, and H. S. Wolf, editors, *Discrete Algorithms and Complexity, Proceedings of the Japan–US Joint Seminar, June 4–6, 1986, Kyoto, Japan*, Perspectives in Computing, pages 119–143, Orlando, 1987. Academic Press.

[23] Oliver Schirokauer. Using number fields to compute logarithms in finite fields. *Math. Comp.*, 69(231):1267–1283, 2000.

[24] Oliver Schirokauer. Virtual logarithms. *Journal of Algorithms*, 57(2):140–147, 2005.

[25] V. Shoup. Lower bounds for discrete logarithms and related problems. In W. Fumy, editor, *Advances in Cryptology – EUROCRYPT '97*, volume 1233 of *Lecture Notes in Comput. Sci.*, pages 256–266. Springer–Verlag, 1997.

[26] Emmanuel Thomé. Subquadratic computation of vector generating polynomials and improvement of the block Wiedemann algorithm. *Journal of Symbolic Computation*, 33(5):757–775, 2002.

[27] P. C. van Oorschot and M. J. Wiener. Parallel collision search with cryptanalytic applications. *J. of Cryptology*, 12:1–28, 1999.

[28] Paul Zimmermann and Bruce Dodson. 20 years of ecm. In *Algorithmic number theory*, volume 4076 of *Lecture Notes in Comput. Sci.*, pages 525–542. Springer, 2006.