# Report for the MAGMA project

Giacomo Borin

August 18, 2022

**Abstract**

The report contains the descriptions of 5 algorithms:

- ECDSA, a digital signature scheme based on algebraic curves.
- Pohlig Hellman algorithm for solving discrete logarithm problem on Algebraic curves. Particularly efficient when the base point has order with only small primes in the factorization
- Index Calculus algorithm for solving discrete logarithm problem in finite fields of prime order using linear algebra and B-smooth sieving.
- Solovay-Strassen primality test, based on the evaluation of the Jacobi Symbol.
- Lehman factorization algorithm, based on a modification of the Fermat factorization algorithm.

## General considerations

In general functions already implemented in MAGMA are significantly faster than ours, so when possible we have used them.

Also I will assume that you will have the code near, since I will refer to it and to the variables.

All the calculation times insertend in the report are realtive to the following machine:

`Apple MacBook Pro "M1 Pro" 8-Core CPU/14-Core GPU 14-Inch (2021)` with a `3.2 GHz Apple M1 Pro processor with 8-cores` and `16 GB of onboard RAM`.

## 1 ECDSA

ECDSA is a Digital Signature Algorithm based on elliptic curves proposed by [JM99] based on the difficulty of the discrete logarithm on them.

The public setup of the scheme is composed by the finite field $\mathbb{F}_q$, an elliptic curve $E = E(\mathbb{F}_q)$ and a base point $G$ with order a large prime $r$. In our case $q$ is a large prime, so we are working with a prime field in the form of integers modulo $q$. The other possibility for $q$ is a power of 2. Then Alice can generate a private key $d \in \{2, .., r-1\}$ and publish the public key $Q := d \cdot G$. The goal of the scheme is to create a signature for an integer modulo $q$, in our case represented by $m$ (usually it is the hash of the message). To do this Alice has to perform:

1. Choose a random $k$ with $1 \leq k < r$, the ephimeral key.
2. Evaluate $R = k \cdot G$.
3. Evaluate $s = (m + d \cdot R_x)k^{-1} \mod r$, where $R = (R_x : R_y : 1)$.

The signature of $m$ is then $(R, s)$.

To verify $(m, R, s)$ Bob has to:

1. Compute $u_1 = s^{-1}m \mod r$ and $u_2 = s^{-1}R_x \mod r$.
2. Evaluate $V = u_1 \cdot G + u_2 \cdot Q$.
3. The signature is valid if $V = R$.

**Theorem 1.1** (Correctness)**.** *Alice is generating a valid signature for each choices of the parameters*

*Proof.* We simply do the computation recalling that the calculations on the coefficients are done modulo the order $r$:

$$\begin{aligned} V = u_1 \cdot G + u_2 \cdot Q = s^{-1} \cdot m \cdot G + s^{-1} \cdot R_x \cdot (d \cdot G) = \\ = s^{-1}(m + d \cdot R_x) \cdot G = k \cdot (m + d \cdot R_x)^{-1}(m + d \cdot R_x) \cdot G = \\ = k \cdot G = R \end{aligned}$$

□

## 1.1 Implementation choices

We have implemented the algorithm from [Was08, Section 6.6].

I've personally implemented all the code and a good part of the test vectors (it didn't took a lot of time), while Dario helped with debugging. The code is the plain transposition of the previous instructions in MAGMA, using the `EllipticCurve` operations, so there isn't much to say. I will recall only some small observations.

We can see that it would be faster to use a fixed small value for the ephimeral key $k$, since it simplifies a lot the evaluation of $kG$ (for example $k = 4$ would require only two doublings) but it would be very bad for security. For example a vulnerability like this was exploited years ago by a group of hacker to find the PS3's ECDSA signature ([]).

However I have tried to use again the idea, considering $k \in [-\frac{r}{2}, \frac{r}{2}]$ (instead of $k \in \{1, .., r-1\}$, so that the modulus is slightly smaller and we save some doublings. To do this I have also defined the function `mmod(a,m)` that returns an integer in $[-\frac{r}{2}, \frac{r}{2}]$ equal to $a$ modulo $m$. This idea saves only a small time (0.1 seconds)[1].

Another observation is that we use `Modinv` to find the inverses modulo $r$ and in the verification algorithm we evaluate $s^{-1}$ only one time, by saving it. Sadly we had to use two casts, since the coordinates of the point `R` are in the finite field, but we need type `Integers()` to perform the multiplications.

## 2 Pohlig-Hellman

The Pohlig-Hellman method is an algorithm for the evaluation of the discrete logarithm , i.e. given $P$ and $Q$ in a group $G$ such that $P$ has finite order $N$ and exists $k \in \{0, ..., N-1\}$ such that $kP = Q$ the algorithm finds $k$.

**Notation.** I will use additive notation for the group operation with neutral element $O$ since we implemented this algorithm for Elliptic Curves.

*Idea* 1. Consider the factorization of $N$, the order of the point:

$$N = \prod_{i=1}^{r} p_i^{e_i}.$$

Solve now the discrete logarithm modulo $p_i^{e_i}$ for all $i = 1, ..., r$ and use these values to evaluate $k$ using the Chinese Reminder Theorem.

Let's see a first naive version of the strategy[2] for finding $k \bmod p^e$:

**Algorithm 2.1.** Here we are pre-computing the multiples of $\frac{N}{p}P$ saving their index.

1. Compute $T = \{(j, j\left(\frac{N}{p}P\right)) \mid j = 0, ..., p-1\}$.
2. Assign $i := 0$, $Q_0 := Q$

Now in the following step we use the pre-computed values to solve the discrete logarithm for a point $R$.

3. find $(j, R) \in T$ such that $R = \frac{N}{p^{i+1}}Q_i$ and assign $k_i := j$
4. Increment $i := i + 1$, then:
5. If $i < e$ assign $Q_i := Q_{i-1} - k_{i-1}p^{i-1}P$ and return to step 3

Using the values $k_i$ we can retrieve $k \mod p^e$.

6. Then we have $k \equiv k_0 + k_1 p + ... + k_{e-1}p^{e-1} \mod p^e$ .

Obviously the procedure terminates, but we still need to prove that the step 6 is correct.

[1]Some evaluation can be seen in the associated pull request

[2]I know that this is indeed very similar to a real code, but for me it is easier to understand

**Proposition 2.2.** *Consider $k$ written in base $p$ as $k = \hat{k}_0 + \hat{k}_1 p + \hat{k}_2 p^2 + \dots$, then we have that:*

*for each $i \in \{0, \dots, e-1\}$ at the step 3 hold that $\hat{k}_i \ = k_i$ and $Q_i = (\hat{k}_i p^i + \hat{k}_{i+1} p^{i+1} + \dots)P$*

Essentially the step 5 with the multiplication by $\frac{N}{p^{i+1}}$ are necessary to find $\hat{k}_i$.

*Proof.* We will prove this by induction on $i \in \{0, \dots, e-1\}$.

**Base case**: $i = 0$. Trivially we have that $Q_0 = Q = kP = (\hat{k}_0 + \hat{k}_1 p + \hat{k}_2 p^2 + \dots)P$. Observe now that:

$$\frac{N}{p^{0+1}} Q_0 = \frac{N}{p} \hat{k}_0 P + N(\hat{k}_1 + \hat{k}_2 p + \dots)P \overset{*}{=} \frac{N}{p} k_0 P + O = \hat{k}_0 \frac{N}{p} P \tag{1}$$

where in $*$ we have used that $NP = O$. Observe that $p$ divides $N$. Since $j$ from step 3 is in $\{0, \dots, p-1\}$ we have $\hat{k}_0 = j = k_0$.

**Induction step**: prove the result for $i \in \{1, \dots, e-1\}$ assuming that it holds for $i-1$.

By the step 5 we have that:

$$\begin{aligned} Q_i := Q_{i-1} - k_{i-1} p^{i-1} P &= (\text{induction on } i-1 \text{ for } \ k_{i-1} \text{ and } \ Q_{i-1}) \\ &= (\hat{k}_{i-1} p^{i-1} + \hat{k}_i p^i + \dots)P - \hat{k}_{i-1} p^{i-1} P = (\hat{k}_i p^i + \hat{k}_{i+1} p^{i+1} + \dots)P. \end{aligned}$$

So we can evaluate $R$ :

$$\begin{aligned} \frac{N}{p^{i+1}} Q_i = \frac{N}{p^{i+1}} (\hat{k}_i p^i + \hat{k}_{i+1} p^{i+1} + \dots)P = \\ = \hat{k}_i \frac{N}{p} P + N(\hat{k}_{i+1} + \hat{k}_{i+1} p + \dots)P = \hat{k}_i \frac{N}{p} P + O = \hat{k}_i \frac{N}{p} P \end{aligned}$$

So in the same way $\hat{k}_i = j = k_i$. Observe that these are the same calculation of the base case and also we have that $p^{i+1}$ divides $N$ since $i < e$. □

Trivially it follows that:

**Corollary 2.3.** *The result in step 6 is correct.*

So to find the discrete logarithm we only have to repeat the previous steps for each $p_i^{e_i}$. We can then solve the system

$$k \mod p_i^{e_i} \quad \text{for } i = 1, \dots, r$$

using the Chinese Reminder Theorem and get the desired $k \in \{0, \dots, N-1\}$.

The strength of this algorithm is that we are performing the discrete logarithm by linear search $\sum e_i$ times in sets of cardinality $p_i$ (having then a complexity of $O(e_j p_j)$ where $p_j$ is the greatest prime factor) instead of one time for all the $N$ values in $\langle P \rangle$. So obviously if the factors of $N$ are small we obtain a result much better than $O(\sqrt{N})$ of other general algorithms (like Shanks or Pollard-Rho).

For the description (and implementation) of the algorithm I have used [Was08, Section 5.2.3] and [SP19, Section 7.2.3] (written in multiplicative notation and with a more constructive description).

## 2.1 Implementation

For now on I will use the same notation used in the code of `ECDLP_PohligHellman(q, E, P, r, Q)`.

Let's start with considering what we have **not** implemented, i.e. the functions already contained in MAGMA:

- `Factorization`, necessary to factor the order of the point `r`
- `CRT`, that at the end evaluate modulo `r` the solution of the system of congruences resulting after the for loop.
- `IsPrimitive`, I will explain later why we need it.
- The `EllipticCurve` structure and all its functions (in particular the sum and the doubling).

The first working implementation (2416fa5) of the code used `Factorization` on $r$ and then followed step by step the Algorithm 2.1 on every factor $p^e$. It was very slow because of two principal reasons:

1. Extensive use of the command `Append`, slow since memory allocation for sequences is expensive, particularly in MAGMA.
2. The use of the command `map<|>`, similar to the dictionary in Python, but sadly not efficient in MAGMA.

The second command was used to perform the steps 1 and 3 of the procedure, that we have individuated as the most time consuming parts of the method:

**Problem 2.4** (Evaluation)**.** Evaluate and store all the values $\{(j, j\left(\frac{N}{p}P\right)) \mid j = 0, ..., p-1\}$ (step 1). In the code $\frac{N}{p}P$ is `coeffP`.

**Problem 2.5** (Search)**.** Search into the evaluated points and return the corresponding index (step 3)

To solve these problems I have observed that we can consider two different cases: $e = 1$ and $e > 1$ (where $e$ is the exponent of the prime in the factorization).

*Remark* 2. Observe that the goal of Algorithm 2.1 is to find the values $\hat{k}_i$, that are the digits of the logarithm modulo $p^e$. It is completely fine to use other techniques, if more efficient.

### 2.2 Case e=1

Since we only need $k$ modulo $p$ (in the code it is called `prime`) here we can solve the two problems together performing in parallel the evaluation of `j*coeffP` (`coeffP` is $\frac{r}{p}P$) while checking if it is equal to `Qs` ($= \frac{r}{p}Q$). In this way we the memory allocation is negligible and we will need on average `prime`/2 operations. while checking if it is equal to `Qs`. In this way we the memory allocation is negligible and we will need on average `prime`/2 operations.

Let's look more in details to the implementation choices in this case. Obviously we start evaluating `Qs` (corresponding to $\frac{N}{r}Q$). If it is the infinity point we return directly `k`$= k_0 = 0$ otherwise we distinguish two other cases.

In the second one we do the simple evaluation starting from $k = 1$ and `Point` = `coeffP`, then increment $k$ by 1 and `Point` by `coeffP` until we find `Point eq Qs`.

In the first we look if 2 is primitive in $\mathbb{Z}_p$, in this way to get all the multiples of `coeffP` we can evaluate instead $2^j$`coeffP` for $j = 0, ..., p-2$.

So instead of incrementing $k$ we multiply it by $2 \mod p$ and we double `Point`, that is slightly faster with respect to adding points. (commit d2d2692)

### 2.3 Case e>1

In this case we go straight with the computation and the storing of points. The only two improvements added where the pre-initialization of the array `T` to reduce new memory allocation and the use of the function `Position` to find the index.

Observe that also `K` and `Qs` where pre-initialized with buffer values.

*Remark* 3. In a real implementation of the code is possible to improve the speed doing a combination with other methods[3]: for example we could use Shanks method to perform the step 3 (avoiding the step 1), resulting with a complexity of $O(e_i\sqrt{p_i})$ where $p_i$ is the greatest prime factor. In this implementation another possible improvement would be to save and reuse the first vector of Shanks algorithm when $e > 1$.

In our case the primes of the factorization where too small to see a real speed up in using this approach (and also it would had been against the rules of the competition).

## 3 Index Calculus

The Index Calculus is a method based on linear algebra to solve the discrete logarithm problem in the finite prime fields, i.e. given a generator $g$ and an element $h$ in $\mathbb{F}_p$ find $x \in \mathbb{Z}$ such that $g^x = h$.

The idea is to fix a factor base $\mathcal{B}$ of primes[4] and looking for linear relations of $\log_g(b)$ for every $b \in \mathcal{B}$. To do this we compute $\{0, ..., p-1\} \ni g_i \equiv g^i \mod p$

[3]idea from [SP19], at page 265

[4]i.e. a finite set of primes

for some random integers $i$, if $g_i$ is $\mathcal{B}$-smooth[5] we have that it factors as

$$g_i = \prod_{b\in\mathcal{B}} b^{e_b} \equiv \prod_{b\in\mathcal{B}} g^{\log_g(b)e_b} \mod p,$$

thus looking at the exponents we have that:

$$i \equiv \sum_{b\in\mathcal{B}} e_b \log_g(b) \mod p-1. \tag{2}$$

So picking random $i$ we can generate enough linear equation (2), find a solution with Gaussian Elimination and assume it to contain the logarithms of the primes.

*Remarks* 4. Surely the vector with the logarithms is a solution, so if we have an overdetermined system (i.e. with more than $\#\mathcal{B}$ equations) is possible for it to be the unique solution. Thus a possible way to retrieve it is to consider $\#\mathcal{B}+1$ equations. We will see later in the implementation that we have done something different, both for correctness and efficiency reason.

Also we have that $p-1$ is not a prime, but we need a field to perform correctly Gauss Elimination. We can work out this by doing linear algebra on $\mathbb{Z}_q$ for all the prime factors $q$ of $p-1$ and combine them with the Chinese Reminder Theorem. We don't have problems with the powers of the factors since from the specifications we got that $p$ is a safeprime, i.e. $p-1=2q$ with $q$ prime.

When we have the logarithms of the base we can start to compute $h\cdot g^{-k}$ mod $p$ for random integers $k$, until we get a $\mathcal{B}$-smooth number, such that:

$$h\cdot g^{-k} \equiv \prod_{b\in\mathcal{B}} b^{l_b} \mod p$$

At this point we can compute:

$$\log_g(h) \equiv \sum_{b\in\mathcal{B}} l_b \cdot \log_g(b) \mod p-1 \tag{3}$$

This algorithms works, with some attentions to the remark during the implementation, but I have found a better version of it in these lecture notes. Here in fact the two steps are united together: instead of considering $g^i$ to find the equations for the linear system we consider directly $h\cdot g^{-k}$ mod $p$. If it is $\mathcal{B}$-smooth from the factorization we get the equation:

$$k \equiv \log_g(h) - \sum_{b\in\mathcal{B}} e_b \cdot \log_g(b) \mod p-1 \tag{4}$$

So if we consider a system of linear equations of the form (4) surely it contains as solution a vector with first component equal to $\log_g(h)$. Geometrically we

[5]recall that a number is $\mathcal{B}$-smooth if we can completely factor it using primes in $\mathcal{B}$

are computing an affine space that contains at least the solution (represented by a single point).

I have verified that this method is computationally more efficient, and I think that there are two reasons:

- It is more compact, with a single linear system we can directly get a candidate solution.
- We need only to look for the first component of the solution[6], so even if all the others are wrong, but the orthogonal projection on the line $\langle(1, 0, ..., 0)\rangle$ is a single point we are sure that it is the solution.

*Remark* 5. A question that I asked myself was: *but what about the other solutions? Can I try to use them?* The answer is no, in fact the projection of an affine subspace on a line is an affine subspace that can have only dimension 0 (so it's a single point) or dimension 1, thus it is the line itself (of cardinality $p-1$). So looking to the other solutions (if not unique) is equivalent to use brute force.

Reference for *index calculus* can be found in [SP19, Section 7.2.4].

## 3.1 Implementation

Dario had done the first implementation, using the first version, but there were some problems and some possible improvements. The principal one was that not every solution of the linear system gives the discrete logarithm. I had tried to solve in two ways:

- Removing columns with all zeros. In fact in this case we don't have condition on the corresponding coordinate, so all the possible values are acceptable. The idea at this point is to consider a factor base that contains only the primes with non zero columns modulo $q$ and 2. A possible way to implement this can be seen in the commit 4b0b67d.
- Since the previous solution is not enough I have inserted a check to see if the solution is effectively the discrete logarithm, if not the function generate more equations and tries to find a new solution.

When I had implemented the second version the filtering on the columns was no longer necessary, but still we need to use an iterative strategy.

Let's look at the final algorithm. It starts with the creation of the factor base, in particular it is done by choosing a bound `TopB` and using $\mathcal{B}=\{\, p \,|\, p$ prime and $p \leq$ `TopB`$\}$ (evaluated with the MAGMA function `PrimesUpTo` ). You can read more on this in the section 3.1.2.

Then we can initialize some bounds (explained in section 3.1.1), two counter, `rows` and `newrows`, a random `exponent` and then start the algorithm. Using `Factorization` we factor $h \cdot g^{-k} \mod p$ and memorize its decomposition. Then

[6]technically in the implementation it will be the last, but it doesn't matter

we can see if it is $\mathcal{B}$-smooth directly verifying the containment of the primes in the decomposition into `B`.

*Remark* 6. Mathematically, to see if a number $n$ is $\mathcal{B}$-smooth and get its factorization it should be better to divide for the primes of $\mathcal{B}$, memorizing the $p$-adic evaluation[7]. Then if after all the divisions we get 1 $n$ is $\mathcal{B}$-smooth and we have also its factorization. We have in fact both tried this possibility (80399ac), but using `Factorization` is still more efficient, so we have abandoned it.

If $h \cdot g^{-k} \mod p$ is $\mathcal{B}$-smooth then we increment `rows` and `newrows`, we save $k$ (`exponent`) and the exponents of the factorization. To do this we need to convert it as a vector in $\mathbb{Z}^{\#\mathcal{B}}$ and it is done with the function `IntermediateZeroPadding` (done by Dario). In the function a sequence of zeros `seq` is initialized, then we iterate on the primes of $\mathcal{B}$, if we find one that it is also in the factorization we write the exponent value in `seq`. Observe that we append a coefficient 1 at the end, necessary to find $\log_g(h)$.

When we have enough rows we try to see if we can find a solution. To do this we use the function `SolveSystem` that solve the corresponding linear system in `GF(prime)` for `prime`$\in \{2, (p-1)/2\}$. The function `Matrix` generates of the matrix in the finite field and `Solution` solves the linear system. Then we consider the last coordinate of the two vectors and retrieve the solution modulo $p-1$ using the Chinese reminder theorem (`CRT`).

Then we check if $g^{\texttt{sol}} \equiv h \mod p$, if not we restart the generation of new rows. We will stop the generation when the number of new rows exceed the second bound, then we look again for a solution and repeat.

### 3.1.1 Bound for iterative search

To actually implement the strategy we need to choose `bound_rows`, that decides when to call `SolveSystem` the first time, and `bound_new_rows`, i.e. the new equations added before a new search for solutions. The most naive solution is to start from $\# \mathcal{B}+1$ and add one equations at each step. Obviously this isn't very efficient, in particular the one step choice. First of all to define them I have decided to consider a percentage of $\#\mathcal{B}$, then I have tried several combinations of the two bounds, in particular with `bound_rows` less than $\# \mathcal{B}$. The final values chosen are:

```
bound_rows := Floor((#B)*(0.78));
bound_new_rows :=Floor((#B)*(0.32));.
```

To find these I have tried all the possible combinations of fractions of $\#\mathcal{B}$ for `bound_rows` and `bound_new_rows`. The idea is to execute the algorithm on some tests and measure the time necessary, as you can see in the commit f3dbf27, and then visualize the data using Sagemath, in order to find patterns. In Figure 2 you can see a 3D plot of the data, that gives an idea of the regions where you can find the best combinations. In Figure 1 we have instead a more precise 2D plot, where the blue points represent the fastest combinations (the slowest are red).

[7]i.e. the maximum exponent $v$ such that $p^v$ divides the integer, but not $p^{v+1}$

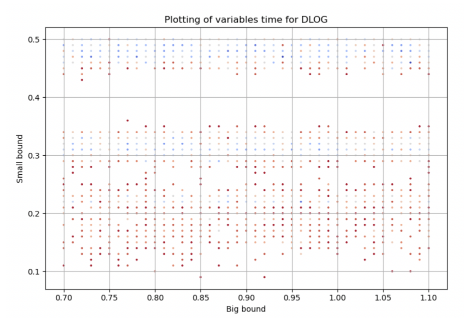


Figure 1: Bounds in 2D, blue is faster.

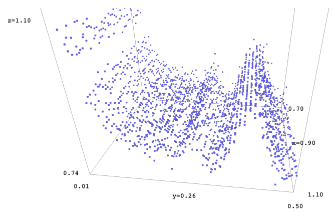


Figure 2: Bounds in 3D.

You can see that the better results are for *small bound* (corresponding to `bound_new_rows`) around 0.32 or 0.46. Different values for the *big bound* correspond to marginal improvements, in fact you can see that in Figure 2 there are two constant strips. However some values are slightly better than others.

Moreover looking at the evaluations it is possible to see that in around 74% of the cases we find the discrete logarithm calling `SolveSystem` only one time, while in all the others we only need to call `SolveSystem` one more time.

#### 3.1.2 The factor base

To find an optimal factor base we need to choose `TobB` properly. Dario handled this by looking at the notes cited before and found a possible form as:

$$e^{c\cdot(\log(p)+\log(\log(p))^2)^{1/3}} \tag{5}$$

This bound depends on $p$ (as expected) and on a positive real constant $c$. Dario has done it by testing all possible values for $c$ in an interval using brute force. Also other possible analytical forms for the bound where tried by Dario, but the form (5) was the fastest.

## 4 Solovay-Strassen

The Solovay-Strassen test is a primality test based on the Euler's criterion:

**Theorem 4.1.** *If $p$ is a prime integer, than for any integer $a \in \mathbb{Z}$ we have:*

$$a^{\frac{p-1}{2}} \equiv \left(\frac{a}{p}\right) \mod p$$

The term on the right is the Legendre Symbol, that is equal to 0 if $p$ divides $a$, to 1 if it is a quadratic residue modulo $p$[8] and $-1$ otherwise. The generalization of the Legendre Symbol for all integers $n = p_1^{e_1} \cdots p_r^{e_r}$ is the Jacobi Symbol:

[8] if exists $b$ integer such that $a \equiv b^2 \mod p$

$$\left(\frac{a}{n}\right) := \left(\frac{a}{p_1}\right)^{e_1} \cdots \left(\frac{a}{p_r}\right)^{e_r}$$

Since for general composite integers the theorem 4.1 will not always hold we can exploit this for a primality test. Given an odd integer $n$ the idea is to evaluate the two terms on a random integer $a \in \{2, ..., n-1\}$ separately, see if they are equal, if not then $n$ is composite, otherwise we change $a$ and repeat. We can conclude that $n$ is composite also if the Jacobi Symbol is equal to 0, in fact this would imply that exists a prime $p_i$ dividing $a < n$ and $n$, so $n$ has a proper factor (in common with $a$).

**Problem 4.2.** How do we evaluate the Jacobi Symbol without knowing the factorization of $n$?

**Problem 4.3.** What is the probability for a composite number to pass the test $m$ times?

It is possible to prove that no composite number pass the test for all $a \in \{0, ..., n-1\}$ and in particular it fails in at least 1/2 of them. So we have that

$$\mathbb{P}(n \text{ pass the SS test } m \text{ times}|n \text{ is composite}) \leq \frac{1}{2^m}. \tag{6}$$

Observe that what we want is $\mathbb{P}(n \text{ is composite}|n \text{ pass the SS test } m \text{ times})$, but it is possible to use Bayes' theorem to compute:

$$\mathbb{P}(n \text{ is composite}|n \text{ pass the SS test } m \text{ times}) \leq \frac{\log(n) - 2}{\log(n) - 2 + 2^{m+1}} \tag{7}$$

You can see the proof of this at page 208 of [SP19, Section 6.4.2].

In our implementation we repeat the test $m = 10$ times and the test vectors are integers $n \approx 2^20 \approx e^{13.9}$, we we get a pseudoprime with probability:

$$\leq \frac{13.9 - 2}{13.9 - 2 + 2^{11}} \approx 0.00577$$

In my opinion the real probability is smaller, since I was unable to find a random counterexample when the test is applied 10 times.

So we have partially solved problem 4.3, thus we focus now on problem 4.2. In the code I have solved this problem using the native MAGMA function `JacobiSymbol`, clearly very fast, but I want however to show a simple way to evaluate it (also implemented in the file, but not used) without the factorization to prove we can solve problem 4.2.

The idea is to use some of the properties of the Jacobi Symbol (inherited from the corresponding ones of the Legendre Symbol) to reduce the problem to it's evaluation on 0, 1 (trivial) and 2. I insert the property in the order in which they are used.

1. We can evaluate $\left(\dfrac{2}{n}\right)$ as 1 for $n \equiv \pm 1 \mod 8$ and $-1$ otherwise. Also we have $\left(\dfrac{ac}{n}\right) = \left(\dfrac{a}{n}\right)\left(\dfrac{c}{n}\right)$ from the definition. So if $a = 2^k m$ (with $m$ odd) we have that:
   - $\left(\dfrac{a}{n}\right) = \left(\dfrac{m}{n}\right)$ for $k$ even or $n \equiv \pm 1 \mod 8$
   - $\left(\dfrac{a}{n}\right) = -\left(\dfrac{m}{n}\right)$ otherwise
2. $\left(\dfrac{a}{n}\right) = -\left(\dfrac{n}{a}\right)$ for $m \equiv n \equiv 3 \mod 4$ and $\left(\dfrac{a}{n}\right) = \left(\dfrac{n}{a}\right)$ otherwise
3. $\left(\dfrac{a}{n}\right) = \left(\dfrac{a \mod n}{n}\right)$

We repeat these steps until $a \in \{0, 1\}$ memorizing the sign changes. We then return the sign if $a = 1$, and 0 otherwise.

Since the case in which we return 0 is equivalent to $a$ having a common factor we can avoid calculation verifying in advance if $\gcd(a, n) \neq 1$, eventually returning 0.

### 4.1 Final remark on the implementation

There isn't much more to say with respect to the implementation, in fact the only two expensive parts are the evaluations of the Jacobi Symbol (very efficient using the native function) and of $a^{\frac{n-1}{2}} \mod n$ (done with `Modexp`). All the other changes that I have tried, for example the way to check $x \equiv y \mod n$, did not improve the time.

## 5 Lehman

The Lehman factorization algorithm is a modification of the Fermat factorization algorithm proposed in [Leh74]. It has complexity of $O(n^{1/3})$ and it was the first historical example of an algorithm with time complexity less than $O(n^{1/2})$.

The principal result of the article is the following theorem:

**Theorem 5.1.** *Suppose that $n = pq$ is a positive odd integer, with $p, q$ primes, and consider and integer $r \in \{\ 1, ..., \lfloor n^{1/2} \rfloor\}$. Suppose that it holds:*

$$\left(\frac{n}{r+1}\right)^{1/2} < p \leq n^{1/2} \tag{8}$$

*Then there exists non negative integers $x, y$ and $k \in \{1, ..., r\}$ such that:*

$$x^2 - y^2 = 4kn, \tag{9a}$$
$$x \equiv k + 1 \mod 2, \tag{9b}$$
$$x \equiv k + n \mod 4 \text{ if } k \text{ is odd}, \tag{9c}$$
$$0 \leq x - (4kn)^{1/2} \leq \frac{1}{4(r+1)} \left(\frac{n}{k}\right)^{1/2} \tag{9d}$$

*and we can use $x, y$ to factor $n$:*

$$\{p, q\} = \{\gcd(x+y, n), \gcd(x-y, n)\} \tag{10}$$

*Instead if $n$ is prime no integers satisfy the equations in 9.*

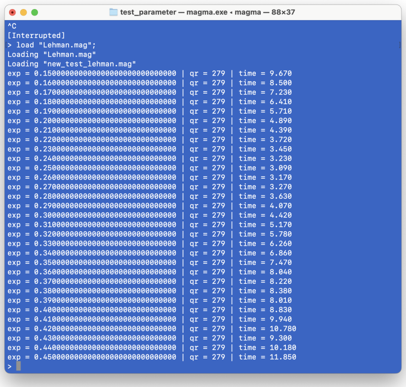
So using the theorem we can write a factorization algorithm for semiprimes:

1. Do trial division up to $\left(\frac{n}{r+1}\right)^{1/2}$, since we can find only primes that satisfy equation 9.
2. Iterate for $k \in \{ 1, ..., r\}$
3. Iterate for $x \in \{ \lfloor (4kn)^{1/2} \rfloor, ..., \lceil \frac{1}{4(r+1)} \left(\frac{n}{k}\right)^{1/2} + (4kn)^{1/2} \rceil\}$ such that $x \equiv k + 1 \mod 2$ and, if $k$ is odd, $x \equiv k + n \mod 4$
4. If $x^2 - 4kn$ is a perfect square $y^2$ return the factorization:
$$\{\gcd(x+y, n), \gcd(x-y, n)\}$$
5. If we can't find any factor after all the iterations we return that $n$ is prime.

The trial division require $O\left(\left(\frac{n}{r}\right)^{1/2}\right)$ operations, while for each $k$ we have $O\left(\frac{1}{r}\left(\frac{n}{k}\right)^{1/2}\right)$ possible $x$ to check (there is also a constant quantity of operations to perform in order to evaluate the interval). So the total operations are:

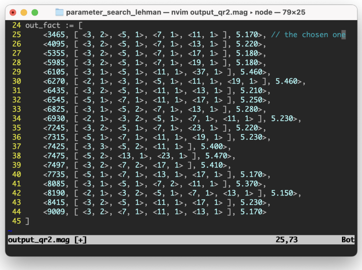
$$O\left(\left(\frac{n}{r}\right)^{1/2}\right) + \sum_{k=1}^{r} O\left(\frac{1}{r}\left(\frac{n}{k}\right)^{1/2} + 1\right) \tag{11}$$

that for $r = O(n^{1/3})$ can be approximated to a complexity of $O(n^{1/3})$. This choice is not proven to be optimal, in fact for the calculation it is used that $\sum_{i=1}^{r} O\left(\frac{1}{k^{1/2}}\right) \subset O(r^{1/2})$, and I think that we can do better (at least empirically I have observed that for $r = n^{1/4}$ we can do better).



```
^C
[Interrupted]
> load "Lehman.mag";
Loading "Lehman.mag"
Loading "new_test_lehman.mag"
exp = 0.150000000000000000000000000000 | qr = 279 | time = 9.670
exp = 0.160000000000000000000000000000 | qr = 279 | time = 8.500
exp = 0.170000000000000000000000000000 | qr = 279 | time = 7.230
exp = 0.180000000000000000000000000000 | qr = 279 | time = 6.410
exp = 0.190000000000000000000000000000 | qr = 279 | time = 5.710
exp = 0.200000000000000000000000000000 | qr = 279 | time = 4.890
exp = 0.210000000000000000000000000000 | qr = 279 | time = 4.390
exp = 0.220000000000000000000000000000 | qr = 279 | time = 3.720
exp = 0.230000000000000000000000000000 | qr = 279 | time = 3.450
exp = 0.240000000000000000000000000000 | qr = 279 | time = 3.230
exp = 0.250000000000000000000000000000 | qr = 279 | time = 3.090
exp = 0.260000000000000000000000000000 | qr = 279 | time = 3.170
exp = 0.270000000000000000000000000000 | qr = 279 | time = 3.270
exp = 0.280000000000000000000000000000 | qr = 279 | time = 3.630
exp = 0.290000000000000000000000000000 | qr = 279 | time = 4.070
exp = 0.300000000000000000000000000000 | qr = 279 | time = 4.420
exp = 0.310000000000000000000000000000 | qr = 279 | time = 5.170
exp = 0.320000000000000000000000000000 | qr = 279 | time = 5.780
exp = 0.330000000000000000000000000000 | qr = 279 | time = 6.260
exp = 0.340000000000000000000000000000 | qr = 279 | time = 6.860
exp = 0.350000000000000000000000000000 | qr = 279 | time = 7.470
exp = 0.360000000000000000000000000000 | qr = 279 | time = 8.040
exp = 0.370000000000000000000000000000 | qr = 279 | time = 8.220
exp = 0.380000000000000000000000000000 | qr = 279 | time = 8.380
exp = 0.390000000000000000000000000000 | qr = 279 | time = 8.010
exp = 0.400000000000000000000000000000 | qr = 279 | time = 8.830
exp = 0.410000000000000000000000000000 | qr = 279 | time = 9.940
exp = 0.420000000000000000000000000000 | qr = 279 | time = 10.780
exp = 0.430000000000000000000000000000 | qr = 279 | time = 9.300
exp = 0.440000000000000000000000000000 | qr = 279 | time = 10.180
exp = 0.450000000000000000000000000000 | qr = 279 | time = 11.850
>
```

Figure 3: Time for $r = n^{exp}$


parameter_search_lehman — nvim output_qr2.mag ▸ node — 79×25

```
out_fact := [
    <3465, [ <3, 2>, <5, 1>, <7, 1>, <11, 1> ], 5.170>, // the chosen one
    <4095, [ <3, 2>, <5, 1>, <7, 1>, <13, 1> ], 5.220>,
    <5355, [ <3, 2>, <5, 1>, <7, 1>, <17, 1> ], 5.180>,
    <5985, [ <3, 2>, <5, 1>, <7, 1>, <19, 1> ], 5.180>,
    <6105, [ <3, 1>, <5, 1>, <11, 1>, <37, 1> ], 5.460>,
    <6270, [ <2, 1>, <3, 1>, <5, 1>, <11, 1>, <19, 1> ], 5.460>,
    <6435, [ <3, 2>, <5, 1>, <11, 1>, <13, 1> ], 5.210>,
    <6545, [ <5, 1>, <7, 1>, <11, 1>, <17, 1> ], 5.250>,
    <6825, [ <3, 1>, <5, 2>, <7, 1>, <13, 1> ], 5.280>,
    <6930, [ <2, 1>, <3, 2>, <5, 1>, <7, 1>, <11, 1> ], 5.230>,
    <7245, [ <3, 2>, <5, 1>, <7, 1>, <23, 1> ], 5.220>,
    <7315, [ <5, 1>, <7, 1>, <11, 1>, <19, 1> ], 5.230>,
    <7425, [ <3, 3>, <5, 2>, <11, 1> ], 5.400>,
    <7475, [ <5, 2>, <13, 1>, <23, 1> ], 5.470>,
    <7497, [ <3, 2>, <7, 2>, <17, 1> ], 5.410>,
    <7735, [ <5, 1>, <7, 1>, <13, 1>, <17, 1> ], 5.170>,
    <8085, [ <3, 1>, <5, 1>, <7, 2>, <11, 1> ], 5.370>,
    <8190, [ <2, 1>, <3, 2>, <5, 1>, <7, 1>, <13, 1> ], 5.150>,
    <8415, [ <3, 2>, <5, 1>, <11, 1>, <17, 1> ], 5.230>,
    <9009, [ <3, 2>, <7, 1>, <11, 1>, <13, 1> ], 5.170>
]
output_qr2.mag [+]                                   25,73          Bot
```

Figure 4: optimal qr mod with factorization and time

## 5.1 Implementation

In [Leh74, Section 5] is also inserted the algorithm written in `Algol` with some ideas to improve velocity, that I have followed in the MAGMA implementation.

First we define the function `isqrt` that returns $\lceil n^{1/2} \rceil$ and $\lceil n^{1/2} \rceil^2 - n$. It uses both `Sqrt` and `Ceiling`.

Then we initialize a boolean array `QR` (outside of the function), the value of $r$ as $\lceil n^{1/4} \ \rceil$ and a sequence of arrays `C`.

*Remark* 7. Statistical tests showed that $r = n^{1/4}$ achieves better performances, as it can be seen in Figure 5.1. I have tried them because I wasn't convinced by the big $O$ calculations in [Leh74] and, at least empirically, I can claim $n^{1/4}$ to be better.

Observe that in our case we have $p \approx q \approx 2^{20}$, so $n \approx 2^{40}$ and $r = n^{1/4} \ \approx 2^{10}$. For the theorem 5.1 we need equation (9) to be satisfied. We can in fact easily see:

$$\left(\frac{n}{r+1}\right)^{1/2} \approx \left(2^{40-10}\right)^{1/2} \ \approx 2^{15} \ << 2^{20} \approx p$$

Also since we have this condition met we can avoid the trial division (useless for RSA semiprimes).

In [Leh74] the authors observed that $k$ with an higher number of divisors are more likely to find a factorization, thus we should try them first. The arrays in `C` are necessary to iterate over $k \in \{1, ..., r\}$ passing before from numbers with more divisors, for example the multiple of 30 and of 24. Each `array` contains a starting value for $k$ `array[2]` that it is incremented using the elements of `array[3]` cycled modulo `array[1]` (the length of `array[3]`). When $k > r$ then we change array. Observe that the odd $k$ values are done last (we will use this later).

Then we define $x := \lceil (4kn)^{1/2} \rceil$ and $u := x^2 - 4kn$ (it is the candidate to $y^2$ as in (9a)). The next two `if` conditions increment $x$ to meet the conditions (9b) and (9c). Observe that also $u$ is incremented using that:

$$(x+b)^2 - 4kn = \underbrace{(x^2 - 4kn)}_{\text{previous } u} + 2 \cdot b \cdot x + b^2 \tag{12}$$

So we don't have to evaluate other squares.

When we have the first $x$ we can start the iteration, incrementing by `jump` $=$ 2 for $k$ even and 4 otherwise, using again the technique in equation (12). By the condition (9d) we have to iterate over `[i1..(j+1) by jump]` where `i1` is the number of $x$ skipped previously.

The only thing we miss now is to check whether or not $u$ is a perfect square $y^2$. Here we will use the boolean array `Qr`, that it was initialized by performing square modulo `qr_mod` and setting to `true` only the quadratic residues. The idea is to use that if $u$ is a perfect square, then it is a quadratic residue modulo `qr_mod`. So if `Qr[u mod qr_mod]` is `false` surely it is not a perfect square. Since in MAGMA the arrays start from 1 we need to slightly modify the modulo using some $-1$ and $+1$.

If we have that it is a quadratic residue we check if it is a perfect square $u = y^2$, if yes we factor $n$ evaluating $\gcd(x - y, n)$. Then it returns the two factors in increasing order.

### 5.1.1 The choice of qr mod

In the article `qr_mod`$= 729 = 3^6$ is used because the percentage of quadratic residues is approximately 38%, but I have observed that we can do better. For example $385 = 5 \cdot 7 \cdot 11$ has only 18.7% of quadratic residues and gives more efficient results during computations. I have actually tried all the possible `qr_mod` up to 10000, finding better choices (inserted in the code), and I have individuated as more efficient the value $3465 = 3^2 \cdot 5 \cdot 7 \cdot 11$. There where other possible good values, as you can see in Figure 4, all odd (but one) and with small factors. They have all small factors because this reduce the quantity of quadratic residues (remind that for the Chinese reminder theorem an integer is a quadratic residue for $n$ if and only if it is a quadratic residue modulo all the factors $p^e$ of $n$).

I think that they are odd because we start the iteration on the even $k$, so $x \equiv k + 1 \equiv 1 \mod 2$ is odd, thus $u = x^2 - 4kn$ is odd, hence the distribution of the non quadratic residue is able to spot more non-squares.

## References

[] *PS3 hacked through poor cryptography implementation.* `https://arstechnica.com/gaming/2010/12/ps3-hacked-through-poor-implementation-of-cryptography/`. Accessed: 2022-08-18.

[JM99] D. Johnson and A. Menezes. *The Elliptic Curve Digital Signature Algorithm (ECDSA).* Tech. rep. 1999.

[Leh74] S. R. Lehman. “Factoring large integers”. English. In: *Math. Comput.* 28 (1974), pp. 637–646. DOI: `10.2307/2005940`.

[SP19] D. R. Stinson and M. B. Paterson. *Cryptography: theory and practice.* English. 4th edition. Textb. Math. Boca Raton, FL: CRC Press, 2019.

[Was08] L. C. Washington. *Elliptic curves. Number theory and cryptography.* English. 2nd ed. Boca Raton, FL: Chapman and Hall/CRC, 2008.