Index Calculus Algorithms

Eli Vigneron

July 2020

1 Introduction

In this short note we will discuss the discrete logarithm problem and also look at an implementation of a few probabilistic algorithms for computing discrete logarithms. These algorithms are collectively classified as index calculus algorithms.

The problem of computing discrete logarithms can be summarized as follows. Letting q be a prime power, the multiplicative subgroup $\mathbb{F}_q^* \leq \mathbb{F}_q$ is cyclic. Elements that generate this cyclic subgroup are called primitive. If we are given a primitive element $g \in \mathbb{F}_q$ as well as any $h \in \mathbb{F}_q^*$, then the discrete logarithm of h with respect to g is the integer $x, 0 \leq x \leq q-1$ such that $g^x = h$. We will denote this by the usual $x = \log_g h \mod q$. The goal is to find this x given q, g and h.

We will start by looking at some naïve implementations of index calculi and then refine our approaches in later sections.

2 The Index Calculus Algorithm (A first look) [1]

We start with some psuedocode illustrating a basic index calculus type algorithm. Throughout, let **Procedure 1** be a smoothness checking algorithm which takes in an integer and a factor base and returns a boolean.

[1] The corresponding Jupyter notebook for this section is titled index_calculus_trial_division.ipynb

Procedure 1 Index Calculus

```
input:
    q a prime
    g a generator of F_q^*
    h an argument
    b a bound for a factor base {2, 3, ..., p_r}
output:
    x such that g^x ≡ h  mod q

factor_base ← list of primes up to b
relations ← empty list

for k = 1, 2, ... do
    g_k ← g^k  mod q
    call Procedure 1 on g_k and factor_base
    if Procedure 1 returns True then
        factorize g_k as 2^{e_0} 3^{e_1} ··· p_r^{e_r}
        rel_k ← (e_0, e_1, ..., e_r, k)
        if rel_k is linearly independent with relations then
            relations ← relations + rel_k
            if length of relations ≥ r + 1 then
                break

R ← reduced row echelon of relations
for s = 1, 2, ..., do
    g_s h ← g^s h  mod q
    call Procedure 1 on g_s h and factor_base
    if Procedure 1 returns True then
        factorize g_s h as 2^{f_0} 3^{f_1} ··· p_r^{f_r}
        return x ← f_0 · R_{0,r+1} + f_1 · R_{1,r+1} + ··· + f_r · R_{r,r+1} − s
```

To illustrate what this algorithm is doing, we look at a concrete example.

Let $q = 83, g = 2$ and $b = 10$ so that our factor base is $2, 3, 5, 7$. Since our factor base has four primes, we are looking for four relations. We have the following:

$$
\begin{aligned}
\star \qquad & 2^1 \equiv 2 \\
& \vdots \\
\star \qquad & 2^7 \equiv 45 = 3^2 \cdot 5 \\
& 2^8 \equiv 7 \\
\star \qquad & 2^9 \equiv 14 = 2 \cdot 7 \\
& 2^{10} \equiv 28 = 2^2 \cdot 7 \\
& 2^{11} \equiv 56 = 2^3 \cdot 7 \\
& 2^{13} \equiv 58 = 2 \cdot 29 \\
& 2^{14} \equiv 33 = 2 \cdot 11 \\
& 2^{15} \equiv 66 = 2^2 \cdot 11 \\
& 2^{16} \equiv 49 = 7^2 \\
\star \qquad & 2^{17} \equiv 15 = 3 \cdot 2
\end{aligned}
$$

The $\star$'s indicate that we split over the factor base and the relation vector is linearly independent with the relations already added.

From this we obtain the matrix of relations i.e. the linear system over $\mathbb{Z}/82\mathbb{Z}$:

$$
\begin{array}{c}
\begin{array}{ccccc} 2 & 3 & 5 & 7 & \end{array} \\
\begin{pmatrix}
1 & 0 & 0 & 0 & 1 \\
0 & 2 & 1 & 0 & 7 \\
0 & 0 & 0 & 1 & 8 \\
1 & 0 & 0 & 1 & 9 \\
0 & 1 & 1 & 0 & 17
\end{pmatrix}
\end{array}
$$

Putting this matrix into reduced row echelon form allows us to determine the logarithms of the elements in the factor base. For example, $(0\ 1\ 0\ 0\ 72) \equiv (0\ 1\ 0\ 0\ -10) = R_2 - R_4$ so we find that $\log_2(3) \equiv 72$.

3 Smoothness Checking [2] [3]

One of the important steps in index calculus type algorithms is a smoothness checking algorithm. This is where a large portion of the running time of the overall algorithm will be spent. The most straightforward way to go about checking if an integer is smooth over a factor base is via trial division.

An illustration of a smoothness checking algorithm using trial division is presented below in pseudocode.

[2] The corresponding Jupyter notebooks for this section are titled smoothness_check_trial_division.ipynb and bernstein.ipynb

[3] Code for a modified sieve of Eratosthenes which can be used to find all smooth numbers and their factorizations can be found in the Jupyter notebook titled eratosthenes_smoothness_sieve.ipynb

Procedure 2 Smoothness Check

```
1: input:
      factor_base ← [2, 3, ..., p_r]
      n ← number to be tested
2: output:
      True if n is smooth
3:
4: k ← ∏_{p ∈ factor base} p
5:  g = gcd(n, k)
6:
7: if g > 1 then n = r g^e for some e ≥ 1
8:     e ← 0
9:     while r ∉ ℤ do
10:        e ← e + 1
11:        r = n/g^e
12:    if r = 1 then n is smooth
13:    else n ← r goto step 5
14: else  return False
```

Using this smoothness checking algorithm together with the index calculus algorithm outlined in the previous section will yield a sub-exponential algorithm for computing discrete logarithms in finite fields of characteristic p. This algorithm will take on the order of $b^{1+o(1)}$ operations, where as before b is the bound for the factor base [Cra05].

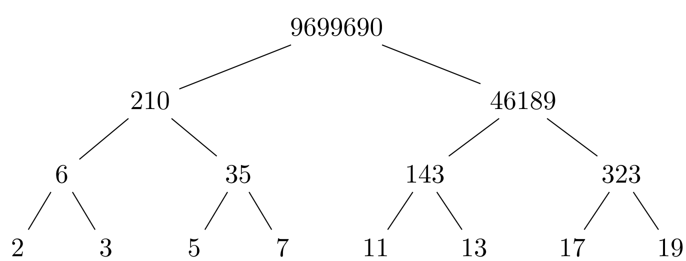
We can improve our smoothness checking step with a method due to Bernstein [Ber04]. Bernstein's algorithm returns all smooth numbers between a range. Note that the interval we are looking in does not have to consist of consecutive integers. If we let M be the maximum element of the interval we are examining then the number of operations is on the order of $(\log^2 b \log M)^{1+o(1)}$ operations.

We begin by illustrating a motivating example. Let $b = 20$ and let our interval be

$$I = [1001, 1002, \ldots, 1008]$$

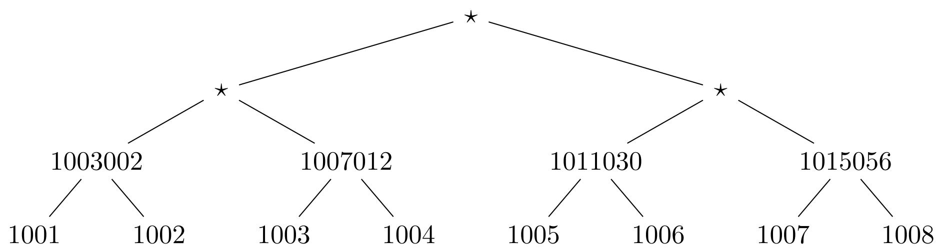
(again the algorithm will work for non-consecutive integer intervals, it is just for the sake of this example that we make use of a consecutive interval).

The first step is to find the product of all the primes up to our bound $b = 20$. This is done by way of a product tree as illustrated below.

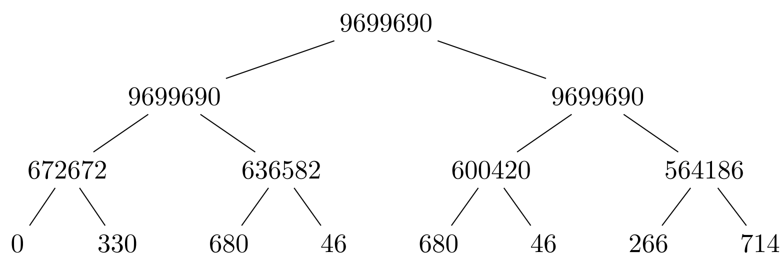


This tree is generated by starting with the leaves, i.e. the factor base and then multiplying up. The root $P = 9699690$ is the product of the elements in the factor base.

We also need the find the residues, $x \mod P$ for each $x \in I$. It would take to long to do this separately for each x, so instead we first find the product $\prod_{x \in I} x$. Again this is done using a product tree, however we note that it is not necessary to form any products larger than P. Let $\star$ denote these unknown products. Then we have the following tree formed from the elements of I:



The next step is to reduce each node in the above tree by P thus creating a 'remainder tree'. This remainder tree is produced by replacing the root R of the tree by $P \mod R$, and then moving towards the leaves replacing each node along the way with its remainder upon division with its parent. This is illustrated as follows (where stars are replaced with P):



Now for each $x \in I$ i.e. the values that we are checking for smoothness, the value in the immediately above remainder tree is $P \mod x$. Using this residue, we can sequentially square and reduce mod x, finally taking the gcd of the end result with x. If this gcd is 0, then x is smooth over the factor base.

Here is the pseudocode:

Procedure 3 Bernstein

input:

factor_base $\leftarrow [2, 3, \ldots, p_r]$

$X \leftarrow$ *set of integers*

output:

smooth numbers in X

Step 1
$\mathcal{P} \leftarrow$ product tree for factor base
$P \leftarrow$ root of $\mathcal{P}$
$\mathcal{T} \leftarrow$ product tree for X for products at most P

Step 2
$\mathcal{R} \leftarrow$ remainder tree $P \mod \mathcal{T}$

Step 3
$e \leftarrow \sup \{e : \max S \leq 2^{2^e}\}$
for $x \in X$ **do**
  $r \leftarrow P \mod s$ get this from $\mathcal{R}$
  $k \leftarrow r^{2^e} \mod x$
  $g \leftarrow \gcd(k, x)$

4 Index Calculus (A second look) [4]

The flavour of index calculus that we will be looking at in this section is known as the linear sieve and it is due to Coppersmith, Odlzyko, and Schroeppel [COS86].

Let p be an odd prime and $K = GF(p)$. Let b be the bound for the factor base and take g to be a generator of K. The goal is to compute the discrete logarithms of as many elements in the 'extended' factor base as possible, where the 'extended' factor base is to be described.

The plan is to use a sieve to collect the relations that involve the logarithms of the factor base elements. This is similar to the first step in the naïve index calculus algorithm from section 1.

Let $H = \lfloor \sqrt{p} \rfloor + 1$ and $J = H^2 - p$. Pick an upper bound, call it climit such that climit is considerably smaller than H. We are looking to find pairs (c_1, c_2) with $2 \leq c_1, c_2 \leq$ climit such that

$$(H + c_1)(H + c_2) \mod p = J + (c_1 + c_2)H + c_1 c_2$$

is smooth with respect to the factor base. Adding the $H + c_i$ to the factor base, we form the extended factor base and we have a relation modulo $p - 1$ involving the logarithms of the terms of this extended factor base.

To check for smoothness, we use the following sieve technique.

Fix c_1, and initialize an array full of zeros. This array will be indexed by c_2. For each prime q in the factor base and each sufficiently small prime power q^h we compute

$$d = (J + c_1 H)(H + c_1)^{-1} \mod q^h$$

Then for each $c_2 \equiv d \mod q^h$ we have

[4] The corresponding Jupyter notebook to this section is titled linear_sieve.ipynb

$$(H + c_1)(H + c_2) \equiv 0 \mod q^h.$$

Then for each $c_2 \in [c_1, \text{climit}]$ where $c_2 \equiv d \mod q^h$, we increment the c_2 entry in the array by $\log q$. The condition that $c_1 \leq c_2$ allows us to avoid redundant relations. After this is done for each of the q in the factor base, we then perform trial division to get the relations corresponding to the c_2's which have a sieve value below a preset threshold.

This process is the repeated for another value of c_1 until we have collected more relations that elements of the factor base. Finally, solving the linear system modulo $p-1$ yields the logarithms we are after. If there are two unknowns in one equation then we will unfortunately be unable to determine the logarithm using this method.

The following is an implementation of this algorithm in SageMath, it is adapted from an algorithm due to Allan Steel which was in turn based on code from Benjamin Costello.

```
from sage.rings.factorint import factor_trial_division


f sieve(K, qlimit, climit, ratio=1.1):
  """
  Input: K = GF(p), qlimit is the uppper bound for the factor base,
         climit is the limit for c_1's (should be much smaller than
         H = floor(sqrt(p)) + 1)
  Output: list containing pairs of elements of the extended factor
          base and their respective discre logarithms
  """

  p = len(K)

  # get the generator of K
  a = K.multiplicative_generator()

  H = floor(sqrt(p)) + 1
  J = H^2 - p

  # Get factor basis of all primes smaller than qlimit
  fb_primes = list(primes(qlimit))

  FB = fb_primes.copy()

  # Initialize extended factor base FB to fb_primes (starting with a)
  # note that starting with a ensures that when the vector mod p-1 is
  # normalized, the logarithms will be wrt to a
  if a in FB:
      FB.insert(0, FB.pop(FB.index(a)))

  if a not in FB:
      FB = [a] + fb_primes
```

```
# Initialize A to 0 by 0, this will be the sparse matrix.
A = []

# Get logs of all factor basis primes
logqs = [float(log(q, 2)) for q in fb_primes]

for c1 in range(1, climit+1):

    # stop if ratio of relations to unknowns is sufficient default 1.1
    if len(A) / len(FB) >= ratio:
        break

    # initialize sieve
    sieve = [log(1) for i in range(1, climit+1)]

    den = H + c1           # denominator of relation
    num = -(J + c1*H)      # numerator

    for i in range(0, len(fb_primes)): #needs to be 0
        # For each prime q in factor base...
        q = fb_primes[i]
        logq = logqs[i]

        qpow = q
        while qpow <= qlimit:
            # For all powers qpow of q up to qlimit
            if den % qpow == 0:
                break

            c2 = num * inverse_mod(den, qpow) % qpow

            if c2 == 0:
                c2 = qpow

            nextqpow = qpow * q

            # ensure c1 <= c2 to ignore redundant relations
            while c2 < c1:
                c2 += qpow

            while c2 <= len(sieve):
                # Add logq into sieve for c2
                sieve[c2-1] += float(logq)

                # Test higher powers of q if nextqpow is too large
                if nextqpow > qlimit:
                    prod = (J + (c1 + c2)*H + c1*c2) % p
                    nextp = nextqpow

```

```
                    while prod % nextp == 0:
                        sieve[c2-1] += float(logq)
                        nextp *= q

                c2 += qpow
            qpow = nextqpow

    # look in sieve for factorization
    rel = den * (H + 1)
    rel_inc = H + c1        # add this to get next rel

    for c2 in range(1, len(sieve)+1):
        n = rel % p

        if abs(sieve[c2-1] - floor(log(n, 2))) < 1:

            fact = factor_trial_division(n, qlimit)

            # need to check if this is the full factorization
            r = 0
            if eval(str(fact)) == 1:
                fact = [(1,2)]

            if fact[-1][1] == 1:
                if fact[-1][0] not in Primes():
                    r = fact[-1][0]

            # check if biggest prime is less than qlimit
            if r == 0 and (fact[-1][0] < qlimit):

                # Include each H + c_i in extended factor basis at end
                if H + c1 not in FB:
                    FB.append(H + c1)
                if H + c2 not in FB:
                    FB.append(H + c2)

                # initialize a sparse row
                row = [0 for b in range(0, len(FB))]

                # now we update it

                # Include relation (H + c1)*(H + c2) = fact
                for pk, e in list(fact):
                    if pk == 1:
                        continue
                    # update A_i,j to e
                    j_1 = FB.index(pk)

                    # stick expont at j
```

```
                    row[j_1] = e

                if c1 == c2:
                    j_2 = FB.index(H + c1)
                    row[j_2] = -2

                else:
                    j_2 = FB.index(H + c1)
                    j_3 = FB.index(H + c2)

                    row[j_2] = -1
                    row[j_3] = -1

                A.append(row)

        rel += rel_inc

# fill out the matrix with sparse entries
for row in A:
    sparse_bit = [0]*(len(FB) - len(row))
    row += sparse_bit

# this block of code is mostly formatting to
# make a call to the Modular Solution method
nrows = str(len(A))
ncols = str(len(A[0]))
B = str(A)
B = B.replace("(", "").replace(")", "")
v = magma_free("C := Matrix(IntegerRing(),"+nrows+","+ncols+","+B+
               "); \n ModularSolution(SparseMatrix(C), "+str(p-1)+");")
v = str(v).replace("(", "").replace(")", "")
v2 = [int(j) for j in str(v).split()]
value_log = list(zip(FB, v2))
return value_log
```

References

[BCFS13] Wieb Bosma, John Cannon, Claus Fieker, and Allan Steel. Handbook of magma functions. *J. Symbolic Comput.*, 2013. Computational algebra and number theory (London, 1993).

[Ber04] D Bernstein. Fast multiplication and its applications. *J. Buhler and P. Stevenhagen, editors Cornerstones in algorithmic number theory, a Mathematical Sciences Research Institute Publication*, 2004.

[COS86] Don Coppersmith, Andrew M. Odlzyko, and Richard Schroeppel. Discrete logarithms inGF(p). *Algorithmica*, 1(1-4):1–15, November 1986.

[Cra05] Richard Crandall. *Prime numbers : a computational perspective.* Springer, New York, NY, 2005.