Breaking Diffie-Hellman

Cristian Vergara, Kaifu Yung, Adolfas Lapsys

November 29, 2016

Contents

1 Disclaimer

To avoid cluttering the paper with mathematical definitions and theorems which our target audience already likely understand, we assume that the reader possesses an understanding of number theory equivalent to that of understanding Chapter 3 in [**TrappeWashington2005**].

2 Diffie-Hellman Key Exchange Protocol

Before introducing how the Diffie-Hellman key exchange protocol works, we will first briefly talk about the Discrete Log Problem, as it is considered to be the backbone of the protocol's security and a key subject of analysis throughout the paper.

2.1 The Discrete Log Problem

Let p be a prime. Let g and β be nonzero integers $\pmod p$. Consider the congruence

$$g^x \equiv \beta \pmod p.$$

In this context, the integer x is called the **discrete logarithm** of β (sometimes referred to as the **index** of β), and the problem of finding it is (unsurprisingly) called the **discrete logarithm problem** (DLP). An efficient algorithm (Polynomial time or better) to calculate $log_g\beta = x$ given p, g, and β is yet to be published, and the most powerful known attempts are of sub-exponential run time complexity at best. The problem is therefore believed to be computationally hard. We discuss some of the known approaches in further sections.

Because we refer to discrete logarithms often in this paper, we use one of the several conventional notations to express it:

$$x = log_g\beta \iff g^x \equiv \beta \pmod p$$

2.2 The Protocol

Diffie-Hellman protocol (DH) was first described in a 1976 paper by W.Diffie and M.Hellman [**DiffieHellman1976**] and is one of the first public key exchange protocols. It is still widely in use to date.

Suppose that Alice wants to agree on a secret key with Bob to later use it in their favorite message encryption scheme. Quite obviously, they don't want Eve to figure out the key in case she does intercept any data transfer between them. Here's how the DH protocol helps us exchange the key:

- Alice generates a large, secure (i.e. not vulnerable to the attacks we discuss below) prime p and some integer $g \pmod p$ and sends both of them to Bob.
- Alice generates some integer $a \pmod{p-1}$ but keeps it secret. She calculates $g^a \equiv A \pmod p$ and sends it to Bob.
- Bob generates some integer $b \pmod{p-1}$ and keeps it secret too. He calculates $g^b \equiv B \pmod p$ and sends it to Alice.
- Observe that both Bob and Alice can now calculate $g^{ab} \equiv K \pmod p$ (B^a and A^b from the perspectives of Alice and Bob, respectively). They use K as their secret key.
- Eve the eavesdropper intercepts the communication between Alice and Bob. She observes the exchange of prime modulus p, the base g, and the two integers A and B. Unless Eve knows a way to (efficiently) calculate $a \equiv log_gA$ or $b \equiv log_gB$ (we believe that she doesn't), Alice and Bob can communicate safely knowing that their key K is not compromised. But, as we can see, finding a or b is equivalent to solving the DLP.

3 The Challenge

Here is the challenge we were given:

Two persons (no doubt evil) are employing a Diffie-Hellman scheme to fix up a session key for their (no doubt evil) communications. They are using a prime modulus of

3217014639198601771090467299986349868436393574029172456674199

and a base value of 5.

Your operatives report that they have witnessed the Diffie-Hellman exhanges of the following information:

244241057144443665472449725715508406620552440771362355600491

and

794175985233932171488379184551301257494458499937502344155004.

Find the shared secret key.

3.1 A Few Immediate Observations

The prime modulus p given to use is a 202-bit number, or 61 digits in base 10. We are given the values g^a and g^b, and we're asked to find $g^{ab} \equiv K \pmod{p}$. This means that we only need to solve for either a or b, because $(g^a)^b \equiv (g^b)^a \equiv g^{ab} \pmod{p}$.

There's a lot more to be said about this problem, but the discoveries that we made will be explained where appropriate.

4 Initial Analysis

We didn't assume more than we were given, so we started analyzing the numbers we were presented with in the problem. Here are two observations:

- 5 is a quadratic residue $\pmod{p}$ (we used Euler's criterion to determine that) and is therefore not a primitive root $\pmod{p}$.
- The modulus p decomposes to $2q + 1$, where q is a prime. We used Miller-Rabin algorithm to confirm that to a high degree of confidence.

4.1 Shanks - Tonelli Algorithm

Shanks - Tonelli algorithm is a way to find square roots of a quadratic residue $n \mod p$. The algorithm works as follows:

1. Factor out powers of two from $p - 1$ and express it as:

$$p - 1 = q \cdot 2^s$$

   If $s = 1 \implies p \equiv 3 \pmod{4}$, then the algorithm is done and the quadratic roots are

$$R \equiv \pm n^{\frac{p+1}{4}} \pmod{p}$$

2. Pick some quadratic nonresidue $z \pmod{p}$ and set $c \equiv z^q$.
3. Let $R \equiv n^{\frac{q+1}{2}}$, $t \equiv n^q$, $m = s$.
4. Repeat the following steps until $t \equiv 1$
   (a) Find the lowest positive integer i such that $t^{2^i} \equiv 1 \pmod{p}$
   (b) Let $b \equiv c^{2^{(m-i-1)}}$ and set $R \equiv Rb$, $t \equiv tb^2$, $c \equiv b^2$, $m = i$.

   Once the loop is done, $(\pm R)^2 \equiv n \pmod{p}$.

We used this algorithm to find the square roots of our base 5.

4.2 Law of Quadratic Reciprocity

Let p and q be distinct odd primes. The Law of Quadratic reciprocity states that if $q \equiv p \equiv 3 \pmod 4$, then

$$\left(\frac{q}{p}\right) = -\left(\frac{p}{q}\right)$$

Otherwise,

$$\left(\frac{q}{p}\right) = \left(\frac{p}{q}\right)$$

With the help of this law, we were able to identify that the two square roots of 5 differ in an important way: one of them is a quadratic residue mod p and the other one isn't. After finding the two square roots with the help of Shanks-Tonelli algorithm, we checked them both and identified the non-quadratic one with the help of Euler's criterion.

Finding a quadratic nonresidue root was an important step in our attempt at using the Index Calculus Algorithm. Using 5 or the square root of 5 that's a quadratic residue would require us to use a stripped version of the factor base (defined in Index Calculus chapter) consisting of only quadratic roots.

4.3 Smooth Integers

The Fundamental Theorem of Arithmetic states that every positive integer $n > 1$ can be represented uniquely as a product of prime powers:

$$n = p_1^{r_1} \cdot p_2^{r_2} \cdots p_k^{r_k} = \prod_{i=1}^{k} p_i^{r_i}$$

We say that an integer n is *B-smooth* if none of its prime factors p_i are larger than some fixed positive integer B (which we may refer to as the "smoothness bound"), i.e.

$$n = \prod_{i=1}^{k} p_i^{r_i} \mid p_i \leq B$$

We will see how integer smoothness comes into play in further sections.

4.4 Dickman Function

The Dickman function ρ is a function that estimates distribution of smooth numbers up to a given bound.
Given $x > 0, y \geq 2$:

$$\Psi(x, y)$$

denotes the number of integers less than x that are y-smooth.
The Dickman function defines this proportion with respect to u where $y = x^{1/u}$ [**DeBruijn1951**], then:

$$\rho(u) = \Psi(x, y)$$

In other words, when y is the uth root of our bound x, $\rho(u)$ is the probability a random element in the range of (2, x] is ($x^{1/u}$)-smooth.

We refer to the Dickman function to have a rough estimate of the chances that "randomly" chosen integers of certain sizes are smooth.

5 The Infeasible Approaches

5.1 Brute Force

It is very easy to observe that a Brute Force approach to this problem is not a smart idea. Trying every single exponent x for $0 \leq x < p - 1$ would result in $O(p)$ operations. Suppose that we were using a supercomputer, where each operation takes 1 picosecond ($10^{-12}s \approx 2^{-30}s$) to execute. Then the amount of time the algorithm would take in the worst case would be in the order of $2^{172}s$, which is an unmanageable amount of time.

5.2 Pohlig-Hellman Algorithm

In 1978, S. Pohlig and M. Hellman published (and, seemingly, forgot to give it a simple name to) a deterministic attack against the Discrete Log Problem. The method is intended to be used when, given a setup $g^x \equiv \beta \pmod{p}$, the modulus p has a particular vulnerability in that $\phi(p) = p - 1$ is smooth (i.e. p-1 factors to small primes).

5.2.1 Explanation

Here is how the algorithm works:

1. We obtain the prime factors of $p-1$: $q_1^{r_1}, q_2^{r_2}, \ldots, q_n^{r_n}$.
2. We compute discrete logarithm of x under every one of the prime factors $q_i^{r_i}$:

   $$log_b x \equiv x_i \pmod{q_i^{r_i}}$$

   Depending on the size of moduli $q_i^{r_i}$, each one of these logarithms can either be found by using brute force, baby step - giant step algorithm (described below) or using the method described in the original paper where this entire algorithm was introduced [**Pohlig2006**].
3. By Chinese Remainder Theorem, we know that the congruence system we obtained in step 2

   $$\begin{aligned} x_1 &\pmod{q_1^{r_1}} \\ x_2 &\pmod{q_2^{r_2}} \\ x_3 &\pmod{q_3^{r_3}} \\ &\ldots \end{aligned}$$

   has one unique solution $x \pmod{p-1}$, therefore we can finding the discrete logarithm by solving the system.

5.2.2 Complexity

A rigorous analysis of the complexity of this algorithm is presented in [**Pohlig2006**]. In short, the expected runtime given an arbitrary prime p, a base value g and its order n is $O(\sqrt{n})$; however, when $p-1$ has only small prime factors and we express it as

$$p - 1 = \prod_i p_i^{r_i}$$

then the algorithm is expected to find the discrete log much faster, where its complexity can be expressed as:

$$O\left(\sum_i r_i(log\ n + \sqrt{p_i})\right)$$

5.2.3 Why it's a bad choice

Because the Pohlig-Hellman algorithm relies on an assumption that $\phi(p)$ is smooth, we conclude that it's not a feasible algorithm to use in our situation as that property doesn't hold for our prime modulus p (i.e. $\phi(p) = 2 \cdot q$, where q is a 201-bit prime). The expected complexity would end up being $O(\sqrt{p})$, which, as described below in Baby Step - Giant Step section, is simply not an option in our case.

5.3 Baby Step - Giant Step Algorithm

The Baby Step - Giant Step algorithm for computing discrete logarithms was introduced by D. Shanks in 1971 [**Shanks1971**]. The algorithm is a meet-in-the-middle attack that works for any modulus p (not necessarily prime) and any base g (not necessarily a primitive root, unlike some other algorithms). When implemented correctly, it's significantly faster than the brute force approach (see analysis of time and space complexity further below).

5.3.1 Explanation with an example

We provide a quick rundown with a small example. Suppose we are given a modulus $p = 37$ and a base value $b = 5$ that is raised to some exponent x, and suppose that our $\beta = 18$, i.e.:

$$5^x \equiv 18 \pmod{37}$$

To find the discrete log x using the Baby Step - Giant Step algorithm, we take the following steps:

1. We let $m = \lceil \sqrt{p} \rceil$. We then generate a list $\{g^0, g^1, \ldots, g^{m-1}\} \pmod{p}$. We store this list in a form of a hash table for $O(1)$ lookup (see below why). In our example, $m = \lceil \sqrt{37} \rceil = 7$, so we calculate the following values $\pmod{37}$:

$$\begin{aligned} 5^0 &\equiv 1 \\ 5^1 &\equiv 5 \\ 5^2 &\equiv 25 \\ 5^3 &\equiv 14 \\ 5^4 &\equiv 33 \\ 5^5 &\equiv 17 \\ 5^6 &\equiv 11 \end{aligned}$$

2. We then start generating a second list $\{\beta(g^{-m})^0, \beta(g^{-m})^1, \ldots\} \pmod{p}$ and for each generated value $\beta(g^{-m})^i$, we check if it's congruent to some value g^j from the first list (i.e. we're looking for a collision – that's why it's useful to have the first list stored as a hash table). It is usually convenient to first calculate the modular inverse g^{-1} of the base value using the Extended Euclidean Algorithm and then use properties of exponents to calculate $g^{-m} \equiv (g^{-1})^m$ and store it for the repeated use in this stage. We stop once the collision $\beta(g^{-m})^i \equiv g^j \pmod{p}$ is found.

   Continuing with our example, we first find 5^{-7}:

$$5^{-1} \equiv 15 \pmod{37} \implies 5^{-7} \equiv 15^7 \equiv 35 \pmod{37}$$

   We then start generating values for the second list $\pmod{37}$ until a collision with the first list occurs:

$$\begin{gathered} 18 \cdot (5^{-7})^0 \equiv 18 * 35^0 \equiv 18 \\ \mathbf{18 \cdot (5^{-7})^1 \equiv 18 * 35^1 \equiv 1} \end{gathered}$$

3. Observe that if a collision is found, the following holds:

$$\beta(g^{-m})^i \equiv g^{x-mi} \equiv g^j \pmod{p} \implies x - mi \equiv j \pmod{p-1}$$

   We can therefore obtain x by simply rewriting the congruence above:

$$x \equiv j + mi \pmod{p-1}$$

   In our example, we found that $18 \cdot (5^{-7})^1 \equiv 1 \equiv 5^0 \pmod{37}$, so $i = 1$ and $j = 0$, which yields:

$$5^x \cdot 5^{-7 \cdot 1} \equiv 5^0 \pmod{37} \implies x - 7 \equiv 0 \pmod{36} \implies x = 7$$

   We confirm that $5^7 \equiv 18 \pmod{37}$ and we're done!

5.3.2 Time and Space Complexity

In the first stage of the algorithm, we compute and store m values of the first list, yielding $O(m) = O(\lceil\sqrt{p}\rceil)$ runtime and space requirement. Before the second stage, we calculate g^{-1} followed by g^{-m}, which takes $O(log_2 n)$ operations thanks to the fast exponentiation method described earlier in this paper. We then start calculating the values for the second list and checking them for a collision with the first list, but how many times to we have to do it in the worst case? Having in mind that we stop once we hit the desired exponent i, we can rearrange the equation $x = j + mi$ to obtain:

$$i = \frac{x-j}{m} \leq \frac{x}{m} \leq \frac{p}{\sqrt{p}} = \sqrt{p}$$

So the second stage will require $O(\sqrt{p})$ operations (which may be less than the first stage) and add no extra space requirement (because we don't really need to store the values after checking them). The final calculation of x is trivial in comparison and will run in $O(1)$ time, so the final time and space complexity and of the algorithm is dictated by te complexity of stage 1, which is $O(\lceil\sqrt{p}\rceil)$.

5.3.3 Why it's a Bad Choice in Our Situation

While Baby Step - Giant Step is a significant improvement over a brute-force attack, it's still not a feasible choice for the problem we were given, and it is easy to observe that on a rather intuitive level. Our modulus p is a 202-bit number, so $\lceil\sqrt{p}\rceil$ is approximately a 101-bit number. That means that we'll need to perform about $a \cdot 2^{101}$ operations and use about $b \cdot 2^{101}$ bits of memory to solve our problem (where a and b are some positive constants). Suppose that we're using some highly sophisticated version of this algorithm, so we allow ourselves to say that $b = 1$ and $a = 1$, and imagine that we're running it on a supercomputer with a 1THz processor on which each operation takes a mere picosecond ($10^{-12}s$). Then the amount of time this algorithm would take in the worst case would be:

$$2^{101} \cdot 10^{-12}s \approx 10^{30} \cdot 10^{-12}s = 10^{18}s \approx 32 \textit{ billion years}$$

That's more than three times the expected life span of the Sun. The memory requirement is not comforting either:

$$2^{101} bits = 2^{98} bytes = 2^{28} ZB \approx 256 \textit{ million Zettabytes}$$

That blows recent estimates of the total amount of data stored on the earth out of proportion [3]. So even using secondary memory instead of RAM is not an option. We can conclude that the algorithm would neither run fast enough (unless we relax the definition of "fast" to allow tens of billions of years) nor would it have enough memory to operate in on any modern machine. Unsurprisingly, our attempt to run it on an Amazon Web Services instance with 3.75GB of RAM ended up with a crash due to exceeding the memory limit.

5.4 Pollard's Rho Algorithm

Pollard's Rho for DLP is a variant of J.M. Pollard's original Rho algorithm used for factoring. Like the original algorithm, the version of Rho method for solving the discrete logarithm takes advantage of what is commonly known as the birthday paradox. The birthday paradox states that given N uniformally distributed, numbered elements, one only needs to randomly choose $\approx \sqrt{N}$ elements in the set to likely end up picking an element twice [**Pollard1974**]

The algorithm is easily parallelizeable, though doing so only yields linear increases in speed.

5.4.1 The Basic Idea

Given the discrete log problem $g^x \equiv h \pmod{p}$:

- Let g be a generator for the finite cyclic group G, which has an order of N.
- For some integer x, where $0 \leq x \leq N$, $g^x = h$ where h is a an element from G.
- The integer x is called the discrete logarithm of h to generator g, denoted $log_g h$.

We choose integers a, b in the range of [0, $N-1$] such that:

- We obtain a random element of G, $g^a h^b$.
- We compute random group elements until we get two elements, $g^{a_i} h^{b_i}$ and $g^{a_j} h^{b_j}$, where $a_i b_i \equiv a_j b_j \pmod{N}$.
- We can then derive the equation: $x \equiv (a_j - a_i)(b_i - b_j)^{-1}$ for $b_i \neq b_j$.

5.4.2 The problem with the naive approach

If we simply consider the basic approach above, it's simple to see that the space complexity of the solution would scale directly with the time complexity, as we're looking for a repeat element in any one of the expected $\approx \sqrt{N}$ elements we previously generated.

The Rho method minimizes storage requirement by utilizing Floyd's Cycle Finding algorithm [**BaiBrent2008**]. Floyd's algorithm uses two pointers (thus constant space) to traverse a periodic sequence where one pointer takes one step (the "tortoise" pointer) while the other pointer takes two (the "hare" pointer). Donald E. Knuth's theorem on Floyd's algorithm claims a repeat element can be found in the sequence within a minimum range of iterations (i) where i lies in the range of $\mu \leq i \leq \mu + \lambda$ [**BaiBrent2008**].

5.4.3 The algorithm

Let:
G be a cyclic group of order p
$g, h \in G$, and a partition of $G = S_0 \cup S_1 \cup S_2$
$f : G \to G$ be a map:

1. Initialize tortoise pointer $a_0 \leftarrow 0, b_0 \leftarrow 0, x_0 \leftarrow 1 \in G$.
2. Initialize hare pointer $A_0 \leftarrow 0, B_0 \leftarrow 0, X_0 \leftarrow 1 \in G$.
3. Initialize $i \leftarrow 1$.
4. Let

$$f(x_i, a_i, b_i) = \begin{cases} x_{i+1} = hx_i, \;\; a_{i+1} = a_i, \;\; b_{i+1} = b_i + 1 \pmod p & if \;\; x_i \in S_0 \\ x_{i+1} = x_i^2, \;\; a = 2a, \;\; b = 2b & if \;\; x_i \in S_1 \\ x_{i+1} = gx_i, \;\; a_{i+1} = a_i + 1 \pmod p, \;\; b_{i+1} = b_i & if \;\; x_i \in S_2 \end{cases}$$

5. Loop until $i = p$:

$$x_{i+1}, a_{i+1}, b_{i+1} = f(x_i, a_i, b_i)$$
$$X_{i+1}, A_{i+1}, B_{i+1} = f(f(X_i, A_i, B_i))$$

if $X = x$:

if $b - B = 0$:
return fail

else:
return $(b - B)^{-1}(A - a) \pmod p$

5.4.4 Possible improvements for Pollard's Rho

Because Pollard's Rho looks for a single collision in the same group, several processors can be used to parallelize if communication between the processors is fostered correctly. One method to do this is with the use of *distinguished points*. Distinguished points in the context of our problem, are going to be group elements that have some sort of easily checkable property. Numbers that have a fixed number of leading zeroes is a common property used. The property is adjusted with respect to the given problem, in such a way so that there are enough distinguished points reduce the number of steps taken before cycle in the sequence is reached, but not so many points so that storage of these points would become a problem [**OorschotWiener1999**]. The actual process of parallelizing Pollard's Rho is as follows:

- Each processor initialize with different exponenents $a_0, b_0 \in [0, p)$ where p is the order of the group.
- A hash table is used to store triples consisting of $\{x_i, a_i, b_i\}$, x_i being the key, where $x_i = g^{a_i} h^{b_i}$

- Each processor then iterates through algorithm normally, storing the triples in the hash table if x_i fulfills the distinguishable property requirement. If a hash collision is encountered, then the result is calculated from the other collision values $(b_i - b_j)^{-1}(a_j - a_i) \pmod p$

5.4.5 Analysis of Pollard's Rho

On a single processor, a collision is expected to occur in $\sqrt{\pi p/2}$ iterations [**BaiBrent2008**]. When the work is divided among m processors generating points independently, this time can be reduced linearly to $\sqrt{\pi p/2}/m$. If we denote the probability of hitting a distinguished point among our iterations as θ, the number of steps from a collision to its detection is a geometrically distributed (as we're simply looking for the first distinguished point after a collision occurs) random variable with an expected value of $1/\theta$. Thus the overall runtime expected runtime of parallelized Pollard's Rho would be [**OorschotWiener1999**]:

$$\sqrt{\pi p/2}/m + 1/\theta$$

5.5 Pollard's Kangaroo Algorithm

The Kangaroo method (referred to by some as λ method) for computing discrete logarithms was introduced by J.M.Pollard in 1978 along with his ρ method [**Pollard1978**]. The method is probabilistic, and its key component closely resembles Kruskal's "card trick" [**Gardner1970**]. It is also easy to parallelize, and is particularly effective if, given the standard DLP setup $g^x \equiv \beta \pmod p$ the the exponent x is known to lie in a certain range $a \le x \le b$.

The general idea of the method (and the reason why it is named after the animal) is to let two Kangaroos – a wild one and a tame one – travel on their own until their paths cross and they fall into the same trap.

Converting the analogy into mathematical terms, this is how the method works:

1. Let G be the set of unique elements $G_i \equiv g^i \pmod p$.
2. Let S be a set of integers $S = \{S_1, S_2, \ldots, S_k\}$.

   This is a set of jumps that either of the Kangaroos may do on their path. Pollard performs some analysis in [**Pollard1978**] and [**Pollard2000**] to make a few recommendations regarding the set:

   - The size of the set, $|S|$, should be significantly smaller than $\sqrt{|G|}$
   - The set's mean, m, should be approximately $\sqrt{|G|}$
   - The set may (and, more often than not, should) contain duplicates as long as its elements are in a non-decreasing order
   - The set's elements should be powers of 2 (which happen to be easy to generate).
   - The set need not be stored in memory – each of the values can be calculated dynamically as long as they don't change.

3. Choose a hash function H. This function will provide a pseudorandom map $H : G \to S$. The goal of using this hash function is to pick "random" jumps depending on the Kangaroo's position on the path. Even though not mentioned in Pollard's paper, it is easy to reason that the efficiency of H is more important than its other qualities (such as "randomness"), because in practice it affects the runtime of the entire algorithm linearly (see below why).
4. Release the Tame kangaroo. Pick a number N (which may be some multiple of m, as suggested by Pollard in [1]) and compute the sequence $\{x_0, x_1, \ldots, x_N\}$ using a recursive rule:

   - $x_0 \equiv g^b \pmod p$ (where b is the upper bound of the range in which we expect the exponent to lie)
   - $x_i \equiv x_{i-1} \times g^{h(x_{i-1})} \pmod p$

   Observe:

   $$x_N \equiv g^b \times g^{h(x_0)} \times g^{h(x_1)} \times \ldots \times g^{h(x_{N-1})} \equiv g^{b+h(x_0)+h(x_1)+\ldots+h(x_{N-1})}$$

   Putting it into context, x_N resembles the trap which we want the Wild kangaroo to fall into.

5. We now calculate d, which will stand for the distance travelled by the Tame kangaroo:

$$d = h(x_0) + h(x_1) + \ldots + h(x_{N-1}) = \sum_{i=0}^{N-1} h(x_i)$$

   Observe that $x_N = g^{b+d}$ (in other words, the position of the Tame kangaroo is equivalent to its starting point b plus the distance travelled d)

6. Release the Wild kangaroo. Begin computing the sequence $\{y_0, y_1, \ldots\}$ using a recursive rule similar to the one in step 3:

   - $y_0 \equiv \beta \equiv a^x \pmod p$
   - $y_i \equiv y_{i-1} \times a^{h(y_i-1)} \pmod p$

   Keep track of the Wild kangaroo's travel distance after each jump:

$$d'_n = h(y_0) + h(y_1) + \ldots + h(y_n) = \sum_{i=0}^{n} h(y_i)$$

7. Stop once the Wild kangaroo either falls into or passes the trap which the Tame kangaroo is in:

   - If $y_i = x_N$, then the trap worked and the Wild kangaroo was caught. The exponent we are looking for can now be obtained using the formula: $x = d'_i - d + b$
   - If $d'_i > b - a + d$, then the Wild kangaroo passed the trap (i.e. the algorithm failed)

5.5.1 Complexity

The algorithm is probabilistic and may finish without finding a solution, but its time complexity, as described by Pollard, is $O(\sqrt{b-a})$.

5.5.2 Why it won't work

The Kangaroo Method is intended to be used when the range in which the exponent lies is known. Since we don't know that range, the complexity is $O(\sqrt{p-2-1}) \approx O(\sqrt{p})$, which we already proved to be infeasible.

6 Index Calculus Algorithm

Index Calculus (IC) is a probabilistic algorithm for computing Discrete Logarithms introduced in by [**Adelman1979**]. The algorithm runs in subexponential time and its runtime complexity is usually expressed using the expression

$$L[\mathit{something}] = O(\mathit{something})$$

Here we introduce the classic, generic version of IC and follow up with a few modifications that we employed for individual parts.

6.1 The Classic version

Speaking generically, the algorithm consists of four stages. Stage one and three are independent from each other and both very easily parallelizeable, and stage four consists of one calculation so trivial that it barely deserves to be called a stage.

The algorithm relies on the following observations. Suppose that, given a base g and modulus p, we raise g to a "random" exponent c and express it in two ways:

$$g^c \equiv \beta \pmod p$$
$$g^c \equiv p - \beta \pmod p$$

We can express β or $p - \beta$ by its unique prime factors p_i to their respective powers r_i multiplied by $(-1)^{r_0}$ (where we set $r_0 = 1$ or 0 if we want to express a negative or positive result).

$$g^c \equiv (-1)^{r_0} \cdot p_1^{r_1} \cdot p_2^{r_2} \cdots p_n^{r_n} \pmod p$$

Taking log of both sides gives us:

$$c \equiv log_g((-1)^{r_0} \cdot p_1^{r_1} \cdot p_2^{r_2} \cdots p_n^{r_n}) \pmod{p-1}$$
$$c \equiv r_0 log_g(-1) + r_1 log_g p_1 + r_2 log_g p_2 + \ldots + r_n log_g p_n \pmod{p-1}$$

In this situation, the exponent c is known and is used to express logarithms of first n primes and -1. What if we had the reverse? Supppose that the exponent c is not known but log_g is known for -1 and first n primes. In that case, given a standard DLP setup $g^x \equiv \beta \pmod p$, we can calculate x as long as β factors into first n primes (i.e. it's *B-smooth*, where B is the n-th prime). TODO add more

6.1.1 First Stage

The first stage is considered to be a "data collection" stage.

We define a *factor base* to be the set of first n primes and -1. We call the nth prime our *smoothness bound B*.

We repeatedly raise the base g to "random" exponents c to obtain congruences $g^c \equiv \beta \pmod p$. For each one of the resulting β values, we attempt factoring it in hopes that β can be represented as a product of members of our factor base. In other words, we hope that β is *B-smooth*. If it is, we represent the relation

$$c \equiv r_0 log_g(-1) + r_1 log_g p_1 + r_2 log_g p_2 + \ldots + r_n log_g p_n \pmod{p-1}$$

as a row vector

$$v = \{r_0, r_1, \ldots, r_n, c\}$$

and save it for later use.

The goal of this stage is to generate a linear system (a matrix) M containing enough linearly independent relations for it to be fully solvable.

This stage is considered to be "embarassingly parallel" because the relations can be generated independently of each other. This part can therefore be executed on many computing units at the same time, yielding an opportunity for a significant linear speedup.

6.1.2 Second Stage

The second stage solves the linear system M. We obtain the discrete logarithms log_g for each one of the primes from our factor base and -1.

6.1.3 Third Stage

By the observations made eariler above, we know that if, given $g^x \equiv \beta \pmod p$, the resulting β is *B-smooth*, then x can be computed easily if we know log_b of each of β's prime factors. But what if β is not *B-smooth*?

We then employ a technique similar to that of the first stage: we keep generating "random" exponents c and calculating

$$\beta g^c \equiv g^{x+c} \pmod p$$

until g^{x+c} is *B-smooth*:

$$g^{x+c} \equiv (-1)^{r_0} \cdot p_1^{r_1} \cdot p_2^{r_2} \cdots p_n^{r_n} \pmod p$$

We stop once we find at least one such relation. We take log_b of both sides and obtain

$$x + c \equiv r_0 log_g(-1) + r_1 log_g p_1 + r_2 log_g p_2 + \ldots + r_n log_g p_n \pmod{p-1}$$

This stage is independent of the first two stages and, just like the first one, can be massively parallelized.

6.1.4 Fourth Stage

Once the first three stages are done, observe that we have computed the following information:

- Discrete logarithms of first n primes and -1 (obtained in first two stages):

$$log_b(-1), log_b p_1, log_b p_2, \ldots, log_b p_n$$

- A congruence (obtained in the third stage) with x unknown:

$$x + c \equiv r_0 log_g(-1) + r_1 log_g p_1 + r_2 log_g p_2 + \ldots + r_n log_g p_n \pmod{p-1}$$

All that's left to do is plug in the discrete logs solve for x:

$$x \equiv r_0 log_g(-1) + r_1 log_g p_1 + r_2 log_g p_2 + \ldots + r_n log_g p_n - c \pmod{p-1}$$

That solves the discrete logarithm problem.

6.2 Modified Extended Euclidean Algorithm

Supppse we are in the first stage of the Index Calculus, generating the linear relations. The general idea in the original version this stage is to repeatedly raise our base g to "randomly" chosen exponents $\{c_0, c_1, \ldots\}$ and to attempt factoring the resulting numbers $\{\beta_1, \beta_2, \ldots\}$ in hopes that they are "smooth enough" (Smoothness explained in section X). However, more often than not, the resulting number β_i will be a relatively large number (by "large" we mean that its size in bits will be close to that of our modulus), so the chances that it will fall under a desirable smoothness bound once factored will be fairly slim (see the section about Dickman function for more rigorous reasoning). On top of that, factoring β_i would be a relatively slow process. However, as you will see below, with the help of a modification to the Extended Euclidean Algorithm, we will be able to quickly reduce the problem of factoring β_i to that of factoring two much smaller numbers which will have a much bigger chance of satisfying our smoothness requirement.

The Extended Euclidean Algorithm (EEA), described in more detail in [**Rosen2002**], provides a fast way to calculate integers x and y such that the greatest common divisor of two integers a and b can be expressed by the equation $ax + by = gcd(a, b)$. When $gcd(a, b) = 1$ and $a > b$, it becomes a convenient way to calculate the modular inverse b^{-1} of b under the modulus a.

In its original form, the algorithm consists of two stages:

1. The first stage (also known as just the Euclidean Algorithm) is used to find the greatest common divisor of a and b;
2. The second stage – the Extended part – is the "reversal" stage where x and y are obtained such that $ax + by = gcd(a, b)$ equation is satisfied.

6.2.1 An Example And a Comparison

Suppose that we are runnning the first stage of Index Calculus algorithm to generate relations, and, we are, say, given a prime modulus $p = 1123$, a base $g = 3$, and we generate an exponent $c = 16$ (These numbers are obviously very small, but we use them purely for an example). We obtain a congruence $3^{16} \equiv 1008 \pmod{1123}$. As expected, 1008 is a relatively large number (11 bits – same as our modulus!), so we conclude that its chances of being smooth are relatively low. How could EEA help us in this case? We could use it to find the inverse of 1008. Let's try an example of it to see if that's useful:

- We let $a = 1123$ and $b = 1008$
- The first stage of the algorithm finds the greatest common divisor:

$$1123 = 1 \cdot 1008 + 115$$

$$1008 = 8 \cdot 115 + 88$$

$$115 = 1 \cdot 88 + 27$$

$$88 = 3 \cdot 27 + 7$$
$$27 = 3 \cdot 7 + 6$$
$$7 = 1 \cdot 6 + 1$$

So $gcd(1123, 1008) = 1$ (which we obviously knew because 1123 is prime).

- We then use the second stage (the "reversal") to obtain x and y:

$$1 = 7 - 1 \cdot 6$$
$$1 = 7 - 1 \cdot (27 - 3 \cdot 7) = 4 \cdot 7 - 1 \cdot 27$$
$$1 = 4 \cdot (88 - 3 \cdot 27) - 1 \cdot 27 = 4 \cdot 88 - 13 \cdot 27$$
$$1 = 4 \cdot 88 - 13 \cdot (115 - 1 \cdot 88) = 17 \cdot 88 - 13 \cdot 115$$
$$1 = 17 \cdot (1008 - 8 \cdot 115) - 13 \cdot 115 = 17 \cdot 1008 - 149 \cdot 115$$
$$1 = 17 \cdot 1008 - 149 \cdot (1123 - 1 \cdot 1008) = 166 \cdot 1008 - 149 \cdot 1123$$

So $gcd(1123, 1008) = 166 \cdot 1008 - 149 \cdot 1123 = 1$.

It is easy to see that we don't gain much from this calculation: our only achievement is that we found 166 to be the modular inverse of 1008 – in other words, $b^{-16} \equiv 166 \pmod{1123}$, and 166 is an 8-bit number, and it's still relatively large in relation to our modulus. However, here's where our modification comes into use. Suppose that instead of computing the greatest common divisor in the first stage of the algorithm, we stop mid-way. We obtain an equation $r = s \cdot t + k$, starting from which we can run the second stage of the algorithm, except that this time, $ax + by$ will end up expressing the number k instead of $gcd(a, b)$. Using the example above, we can apply this modification to express any of the remainders obtained in the first stage of the algorithm in terms of a and b:

- We start the first stage the regular way but stop early:

$$1123 = 1 \cdot 1008 + 115$$
$$1008 = 8 \cdot 115 + 88$$
$$115 = 1 \cdot 88 + 27$$

So $k = 27$.

- We express k in terms of a and b:

$$27 = 115 - 1 \cdot 88$$
$$27 = 115 - 1 \cdot (1008 - 8 \cdot 115) = 9 \cdot 115 - 1 \cdot 1008$$
$$27 = 9 \cdot (1123 - 1 \cdot 1008) - 1 \cdot 1008 = 9 \cdot 1123 - 10 \cdot 1008$$

So we end up with $27 = 9 \cdot 1123 - 10 \cdot 1008$. In other words, $k = 27$, $x = 9$, and $y = -10$. This equation can now be rewritten in a way that is going to be more useful to us:

$$27 = 9 \cdot 1123 - 10 \cdot 1008 \iff 27 \equiv -10 \cdot 1008 \pmod{1123}$$

Having in mind that $\phi(1123) = 1122$, we can take log_3 of both sides and rearrange the expression as follows:

$$log_3 27 \equiv log_3(-10 \cdot 1008) \pmod{1122}$$
$$log_3 27 \equiv log_3(-10) + log_3 1008 \pmod{1122}$$
$$log_3 27 \equiv log_3(-10) + 16 \pmod{1122}$$
$$log_3 27 - log_3(-10) \equiv 16 \pmod{1122}$$

Notice that 27 and -10 are **much** easier to work with because they are significantly smaller than 1008 or 166 in relation to the modulus 1123. We can therefore conclude that it's worth spending the time attempting to factor them and hope that they will both fall under our desired smoothness bound. In fact, in our example we find that both numbers turn out to be 5-smooth, which, given modulus 1023, is a reasonable achievement:

$$27 = 3^3$$
$$-10 = (-1)^1 \cdot 2^1 \cdot 5^1$$

We can now use the properties of logarithms to rewrite 16 (mod 1122):

$$log_3 27 \equiv log_3 3^3 \equiv 3 \cdot log_3 3 \pmod{1122}$$
$$log_3(-10) \equiv log_3(-1^1 \cdot 2^1 \cdot 5^1) \equiv log_3(-1) + log_3 2 + log_3 5 \pmod{1122}$$
$$log_3 27 - log_3(-10) \equiv log_3(-1) + log_3 2 - 3 \cdot log_3 3 + log_3 5 \equiv 16 \pmod{1122}$$

This gives us a 5-smooth relation for our first stage of the Index Calculus algorithm.

6.3 Finding Smooth Parts of Integers in Batches

Finding Smooth Parts of Integers in Batches (will be referred to from now as Batch-Find or BSF) is a variation of Daniel J. Bernstein's Batch Integer Division (also designed by DJB) that, given a finite sequence of integers, S, and a finite set of primes P, finds the of each integer in S with respect to the set P [**Bernstein2004**].

The algorithm finds the "smooth-part" of integers in our set, but what is a "smooth-part?"The smooth part of an integer with respect to a set of primes is the largest divisor of that integer that divides over the set of primes (which means that divisor is P-smooth). For instance:

Let's suppose a number's prime factorization is $2^3 5^1 7^1 23^1$.
Given the set of primes P = $\{2, 3, 5\}$, the P-smooth-part of the above number is $2^3 5^1$.

In the context of Index Calculus, BSF provides an efficient method of data collection in phase 1 of Index Calculus. When used with regular Index Calculus we can test if candidates of g^x are factorable over our factor base in large batches. Additionally we can use BSF for testing r (and s values if a smooth candidate is found from the batch of r values) from the Modified Extended Euclidean Algorithm for smoothness for further speed up in phase 1.

We ended up not using BSF due to timing discrepancy between its implementation and that of the Linear Sieve method (additionally BSF can't be pipelined with the Linear Sieve method like it can with MEEA). BSF was tested in combination with MEEA successfully with smaller moduli on a single thread machine, but was not tested on our main platform.

6.3.1 The subroutines

Because BSF does its computations in batches, several subroutines are used to take advantage of the batch structure.

1. Product Tree

   - Due to CPU and compiler optimization, on most CPUs multiplication is a constant time process as long as the multiplier and multiplicand can each fit in a single register. However once numbers become too large and some bignum library is required, multiplication is then almost completely "handled by software", and multiplication actually becomes an expensive operation that runs in $n \times m$ time where n and m are the number of digits in the multiplier and multiplicand. Time spent on multiplication is referred to as $\mu - time$.
   - When dealing with the product of a set of integers a product tree is used to avoid the having to sequentially multiply over and over again throughout the entire set.
   - Conceptually, the product tree is very simple. The product tree is a binary tree where the leaves of the product tree are the numbers in the set we wish to multiply. Each non-leaf node of the tree holds the product of its two children, with the root of the tree being the product of every number in the set.
   - Computation of the product tree of a set of elements: $\{p_1, p_2\ ...p_m\}$ takes at most $kb\mu$ time, where k is the smallest integer such that $2^k \geq m$, b = number of bits in $\{p_1\ ...\ p_m\}$, and $\mu = \mu - time$ [**Bernstein2004**].

2. Remainder Tree

   - Similar to the the product tree, computing an integer $h \pmod{x_k}$ for $k \in \{1...n\}$ can be a costly process when done sequentially done over the set of $\{x_1...x_n\}$, and can be sped up using a binary tree structure.
   - A remainder tree is binary tree where the leaves are h modulo each element of $\{x_1...x_n\}$, and every nonleaf node is h modulo the product of its two children.
   - Unlike a product tree, where we construct the tree bottom-up, we construct the remainder tree top-down, using the product tree of $\{x_1...x_n\}$ to get the intermediate products to use as the moduli in the non-leaf nodes of the remainder tree.
   - Computation of the remainder tree of a set of elements, using the product tree of $\{x_1...x_n\}$ to get the intermediate products to use as the moduli in the non-leaf nodes of the remainder tree.
   - Computation of the remainder tree as described above takes $O(b(log_2 b)^2 log_2(log_2 b))$ time where $b =$ number of bits in $\{x_1...x_n\}$ [**Bernstein2004**][**Bernstein2008**].

6.3.2 The algorithm

Let:
$P = \{p_1, p_2, p_3...p_m\}$ be a set of distinct prime numbers
$S = \{x_1, x_2, x_3...x_n\}$ be a set of positive integers

1. Compute $z = p_1 \times p_2 \times p_3 \times \ ... \ \times p_m$ using a Product Tree
2. Compute $z \pmod{x_1}$ ... $z \pmod{x_n}$ using a Remainder Tree
3. For each $k \in \{1\ ...\ n\}$: Compute $y_k \leftarrow (z \pmod{x_k})^{2^e} \pmod{x_k}$ by repeated squaring, where e is the smallest integer such that $2^{2^e} \geq x_k$
4. For each $k \in \{1\ ...\ n\}$: Print $\{x_k, y_k\}$. These are the smooth parts of each x_k

6.3.3 Possible improvements for BSF

The basic method of BSF as it turns out provides more information than we want for our problem. BSF generates the smooth parts of integers (The "portion" of the integer that fulfills the smoothness requirement defined by our factor base), but we only care if the entire number fulfills our smoothness requirement (finding the smooth part of integers has applications outside of Index Calculus). Thus we can skip step 4 of the algorithm entirely and simply compare each y_k generated from step 3 to the corresponding x_k in our batch of candidates. If they're equal, then x_k divides by our factor base. If we had continued with step 4, then $gcd(x_k, y_k)$ would have been x_k, meaning the smooth part of x_k would've been itself.

6.3.4 Analysis of BSF

Let b be the number of bits in the total input (the collective number of bits in the batch of candidates). The runtime for each step is as follows:

1. Step 1 takes $O(b(log_2 b)^2 log_2(log_2 b))$ [**Bernstein2008**]
2. Step 2 takes $O(b(log_2 b)^2 log_2(log_2 b))$ [**Bernstein2008**]
3. Step 3 takes $O(b_k(log_2 b)^2 log_2(log_2 b))$ where b_k is the number of bits in x_k, since e is bounded by $log_2 b$. Step 3 run over the entire batch would therefore take $O(b(log_2 b)^2 log_2(log_2 b))$
4. Step 4 we were able to avoid doing altogether due to the goal of our algorithm.

Altogether the algorithm takes

$$O(b(log_2 b)^2 log_2(log_2 b))$$

time.

- When dealing with a 500000-prime factor base, getting the sequential product of the factor base took 77693 milliseconds, while the product tree yielded the result in 329 milliseconds
- Using the remainder tree vs sequential modulo operations also yielded significant speed gains, though the speed gains varied heavily with different batch size and batch inputs

6.3.5 Implementation Details

Because Product and Remainder Trees are always complete balanced binary trees, we were able to implement via arrays doing computations in place, similar to the standard binary heap implementation.

6.4 Pollard's p-1 Factorization Method

Pollard's p-1 factorization algorithm is a method of factoring a composite integer n using ideas established in Fermat's Little Theorem which states [**Pollard1974**]:

$$a^{p-1} \equiv 1 \pmod p$$

where p is a prime number.

We set an integer $d \equiv a^{p-1} - 1 \equiv 0 \pmod p$, where a is relatively prime to p. Thus, $gcd(d, n) = p$. p was the prime factor of n we were looking for.
Let $m = p - 1$. The problem then reduces to how select a "good" m.
The trick is realizing m doesn't have be exactly equal to $p - 1$, as the equation would still be valid if m is a multiple of $p - 1$:

$$a^m - 1 = a^{c(p-1)} - 1 = a^{(p-1)^c} - 1 = 1^c \equiv 0 \pmod p$$

Thus we want to iterate m in such a way to maximize the chance of the above holding. We do so by choosing m as a product of many small primes.

6.4.1 The algorithm

1. Choose a smoothness bound B to choose primes from.
2. Compute
$$\prod_{1 \le p \le B} p^{logB/logp}$$
3. Choose a random a between 1 and n.
4. Compute $d = gcd(m, n)$.
   If $d = 1$, go to step 5.
   If $d \neq 1$, return d.
5. Compute $a^m \pmod n$.
6. Compute $e = gcd(a^m - 1, n)$.
   If $e = 1$, increase B, goto step 1.
   If $e = n$, choose a new a and goto step 3.
   If $e \neq 1$ & $e \neq n$ return d.

6.4.2 Viability of the p-1 method

Because the p-1 method is relient on selecting a number ,m , which $p - 1$ divides, the performance of the algorithm is directly dependent on how smooth the prime factors of our number n is. If we're trying to factor out a p which is extremely unsmooth from n, then two situations can occur.

- We'll have to continually keep increasing our smoothness bound after each iteration of p-1 until prime factor is found

- We start with an extremely large smoothness bound, which would hurt the performance of each individual iteration of p-1

Either way, Pollard's p-1 is an incredibly poor tool to use if we're trying to factor a number that's known to be unlikely smooth.

6.5 Linear Sieve

The Linear Sieve is an alternative approach to stage 1 of Index Calculus. [**Coppersmith1986**]

6.5.1 Introducing new variables

Given a DLP where a is our generator, b is the residue, and p is a prime modulus:

1. Let $H = \lfloor\sqrt{p}\rfloor + 1$, $J = H^2 - p$
2. Let S be an extended factor base containing not only our set of small primes, but also integers "near" H
3. Select some bound C and two constants c_1, c_2, such that $0 \leq c_1, c_2 \leq C$
4. If $(H + c_1)(H + c_2) \pmod{p}$ is B-Smooth then:
   - $(H + c_1)(H + c_2) \equiv \prod_{q_i \in B} q_i^k \pmod{p} forsomek \geq 0$
   - We can see that $(H + c_1)(H + c_2)$ produces a residue that can be expressed as the product of small primes
   - Taking log base a of both sides:

   $$log_a(H + c_1) + log_a(H + c_2) \equiv \sum_{q_i \in B} klog_a(q_i) \pmod{p-1}$$

   We get one of the equations we'll want in stage 2 of Index Calculus

6.5.2 Explanation

The value of $(H + c_1)(H + c_2) \pmod{p}$ is $J + H(c_1 + c_2) + c_1c_2$. This value is only slightly larger than $\sqrt{p}$, and is therefore more likely to be smooth than p.

6.5.3 Choosing appropriate parameters

Using the algorithm as described above to collect relations, we can see that in addition to selecting the size of our prime factor base, we also need to select an appropriate bound C for which to choose our constants $c1, c2$ from.

- Choosing bound C is a balancing act.
- By selecting a larger C, the search field for $c1, c2$ will increase quadratically since we are considering $c1, c2$ pairwise, and we'll be more likely to find a relation
- The drawback from selecting a large C however is it would greatly increase the number of $(H + c_i)$ residues we compute that don't factor out over our factor base, thus not contributing to our solution

6.5.4 The "sieving" in the Linear Sieve

To efficiently find smooth residues in $(H + c_i)$ form, we first pick a $c1$, then we solve for $c2$ in the following equation for each q_i:

$$J + H(c_1 + c_2) + c_1c_2 \pmod{q}_i$$

Solving for c_2:

$$J + Hc_1 + Hc_2 + c_1c_2 \equiv J + Hc_1 + c_2(H + c_1) \pmod{q_i}$$

$$-(J + Hc_1) \equiv c_2(H + c_1) \pmod{q_i}$$

$$-(J + Hc_1)(H + c_2)^{-1} \equiv c_2 \pmod{q_i}$$

6.5.5 The algorithm

The sieving process is as follows:

1. Fix a $c1$
2. Initialize an array of size $|C|$ to θ
3. For each prime power:

   - Compute $d \equiv -(J + Hc_1)(H + c_2)^{-1} \pmod{q_i}$
   - Add lnq_i to each index in θ that is congruent to $d \pmod{q_i}$:

4. Factor each element of θ that has a value close to $ln(J + H(c_1 + c_2) + c_1c_2)$ over the prime factor base.

6.5.6 Implementation Details

Parallelization is fairly simple. We simply delegate different fixed values of c_1 to different processes and threads. It is possible that $(H + c_2)^{-1} \pmod{q_i}$ is not defined. In these cases, a separate process exists that involves factoring q_i from $(H + c_2)^{-1}$ and $(J + Hc_1)$. Though we implemented this process, in practice it did not help us much. It would have been more effective to simply ignore these values and continue with the algorithm.

We also ignored values of c_2 for which we've already fixed at c_1. This meant that threads assigned earlier values of c_1 had more work to do than threads assigned later values of c_2. A better method of allocating work would have sped up the process significantly.

In all, it took us 12 hours to generate a list of relations over 450k prime numbers and 150k values of $H + c$. In all, we had 1.2 million relations for a matrix of size 1.2 million by 600 thousand.

6.5.7 Caveats

It is often assumed when you do a Linear Sieve that your generator is a small prime, so $logg = 1$ is known. This way, we can make the matrix solvable. However, our only known prime is $log5 = 2$.

6.6 Structured Gaussian Elimination

The matrix generated in the Linear Sieve is far too large to process. Structured Gaussian Elimination is not a process to solve the matrix, but to reduce the size of the problem to something that can be managed.[**Coppersmith1986**] [**LaMacchia1991**] If we are trying to find an individual logarithm, we don't necessarily need 450k primes to do so. We can safely discard many of the relations in our set.

Structured Gaussian Elimination describes the following four operations.

1. Designate the columns with the most entries as "heavy", the rest as "light". Assign each row a weight being the number of light columns in that row.
2. Select a row of weight 1. Add multiples of that row to other rows as to eliminate the column, then delete the row.
3. Delete columns with only a single entry, and the associated row of that entry.
4. Delete rows of highest weight

All implementations of SGE are different combinations of these operations. In our implementation, we started by deleting singular columns, assigned weights, deleted rows of weight 1, then deleted rows of the highest weight. We repeated the process until we reached a managable size. The resulting matrix of 190607 relations with 107746 unknowns. Of the unknowns, 60k were prime. We started our search for the individual logarithm with these primes as a factor base using the modified extended euclidean algorithm method described previously.

6.6.1 Implementation Details

In our implementation, we wrote our program to perform all steps in sequence. This required a very complex structure to keep track of the matrix, including the weights and columns. Had we done this again, we would implement this process as four separate programs that act using each operation. This would have simplified the code at each step.

6.7 Lanczros Algorithm

To solve a large matrix, we used the Lanczos method[**LaMaccia1991**]. This method is an extension of his method for solving the eigenproblem. To see how it works, let's see how it works in the real numbers.

Let A be a symmetric matrix. We're trying to solve $Ax = w$ with w known and x unknown. The Lanczos Algorithm builds an orthogonal set of vectors and iteratively constructs the solution.

1. Set $w_0 = w$
2. Set $v_1 = Aw_0$
3. Set $w_1 = v_1 - w_0(v_1^T Aw_0)/(w_o^T Aw_0)$
4. Then until $w_i^T Aw_i = 0$:
    - Set $v_{i+1} = Aw_i$
    - Set $w_{i+1} = v_{i+1} - w_i(v_i^T Aw_i)/(w_i^T Aw_i) - w_{i-1}(v_{i+1}^T Aw_{i-1})/(w_{i-1}^T Aw_{i-1})$
5. If the algorithm ends at iteration j, If $w_j = 0$, then $x = \sum_0^{j-1} b_i w_i$ where $b_i = (w_i^T w)/(w_i^T Aw_i)$. Otherwise if $w_0 \neq 0$ the algorithm fails.

To adapt this algorithm to work in our case, we need to convert our matrix into a symmetric matrix. Since we're trying to solve $Bx = u$, we convert the problem by multiplying both sides by B^T such that $A = B^T B$ and $w = B^T u$. The solution for x will remain the same.

We do not need to calculate $A = B^T B$ directly. Instead, we can caculate $v_i = B^T(Bw_{i-1})$ at each iteration.

6.7.1 Implementation details

The slowest part of our implementation is the multiplication of a scalar by a vector. The multiplication of a large matrix took 5s, and since it has to be done twice (once for B^T and again for B), it would take 10s to do an iteration. In the initial implementation, the sparse matrix was an array of pointers to an array of row elements. This caused an issue when trying to spread the work across threads. In the transpose, the rows representing smaller primes were very dense, so threads working in those rows did far more work than the other threads.

The matrix was reworked to facilitate spreading the work evenly across threads. We represented the matrix as a linear array of 64-bit unsigned integers. Each element was given 6 cells of the array. The first cell gave the row, the second gave the column, and the remaining four were the entry in the cell. Each thread was given a range of the array. The thread would take the coefficient stored in the element and multiply it by the column index of the input vector. It would then add the sum to the output vector at the row index. The row output vector elements were guarded by a mutex to prevent any race condition from occurring with the write.

With this implementation, iterations took a little under 1 second to run. We founded roughly 70 iterations every minute. In all, it took a little over 24 hours to find a solution for the matrix.

7 Conclusion

At the end of our Phase 2, we found that the results were inconsistent, so we were unable to solve the problem in the time given. The Lanczros implementation was tested on smaller matrices, and would return inconsistent results if the matrix either did not have full rank, or the matrix was inconsistent.

One issue we found afterwards was in how we defined the relations for use with the matrix. The relations defined by the Linear Sieve all equal 0, and it is usually assumed that you know the log of a small prime (often the generator). We knew $log 5 = 2$, but when we converted the matrix, we assigned $log 5 = -2$. I do not think this alone would cause issues, since we should be able to multiply the result by -1 and get the correct answer.

Another issue is that $A = B^T B$ is not guaranteed to have the same rank as B. I think this is likely what caused our problem. A solution to this is to define $A = B^T D^2 B$ for a random diagonal matrix D and solve. We skipped this step because that technique is more often used to handle the situation where the height of A is not significantly smaller than the modulus that we're attempting to solve, which did not apply to us.

We wound up cutting it too close to the deadline to rerun. The matrix was solved the morning of the due date for this paper. We're interested in trying again with corrections, but the solution would come after we've already submitted this report.