Breaking the Discrete Log problem

Karlo Delić

(Project assignment — Programming for Contemporary Processors)

1. Introduction and motivation

The discrete logarithm problem is a fundamental problem in modern cryptography because it forms the basis of many popular encryption techniques, including the Diffie-Hellman key exchange, ElGamal encryption, and digital signature schemes. In this work, we will explore several algorithms used for solving this problem.

The discrete logarithm problem (DLP) is the problem of finding an exponent in a group, where the group is generated by a fixed element. More formally, given a group G of order n, a generator g of G, and an element h in G, the problem is to find an integer x such that $g^x = h \pmod{n}$. DLP is believed to be computationally infeasible[1] on classical computers for large prime-order groups, which makes it an attractive choice for cryptographic applications.

The history of the discrete logarithm problem dates back to the mid-1970s, when Whitfield Diffie and Martin Hellman proposed the Diffie-Hellman key exchange protocol, which is a method for two parties to agree on a shared secret over an insecure communication channel. The protocol relies on the difficulty of the discrete logarithm problem in finite fields to ensure that an eavesdropper cannot determine the shared secret in a viable time period.

Since then, the discrete logarithm problem has become a cornerstone of modern cryptography. The ElGamal encryption scheme, which is a public-key cryptosystem, also relies on the intractability of the discrete logarithm problem in finite fields. In this scheme, the sender encrypts a message using the recipient's public key, and the recipient decrypts the message using their private key, which is based on the discrete logarithm problem.

This problem is also used in digital signature schemes, such as the Digital Signature Algorithm (DSA), which is widely used for authentication and data integrity in secure communication protocols. The DSA scheme relies on the discrete logarithm problem in finite fields to ensure that a signature cannot be forged.

In recent years, the development of quantum computers has raised concerns about the security of cryptographic schemes that rely on DLP. Quantum computers can solve DLP in polynomial time using Shor's algorithm, which poses a significant threat to the security of existing cryptographic systems. As a result, researchers have been exploring new cryptographic schemes that are resistant to quantum attacks, such as post-quantum cryptography.

[1] Not solvable in polynomial time.

2. Classical algorithms

2.1. Baby-steps giant-steps algorithm

2.1.1. High level description

The *Baby-steps giant-steps* algorithm works by dividing the search space for the discrete logarithm into two parts: the first part consists of a set of small *baby* steps, where we compute the values of $g^0, g^1, g^2, \ldots, g^{m-1}$ for some small m. The second part consists of a set of larger *giant* steps, where we compute the values of h^{-im} for $i \in \{0, 1, 2, \ldots, m-1\}$. Then, we check whether any of these giant steps land on one of the baby steps we computed earlier. If we find a match, then we can solve for the discrete logarithm by computing $x = im + j$, where j is the index of the baby step that matched the giant step of index i.

The intuition behind this algorithm is that if we randomly choose x as a candidate solution for the discrete logarithm, then the probability that $h * g^{-x}$ coincides with one of the baby steps is roughly $1/m$. Therefore, by computing the giant steps for all i, we increase our chances of finding a match, without having to compute all possible values of g^x and $h * g^{-x}$.

2.1.2. Pseudocode

```
function BabyStepGiantStep($g, h, p$)
    $m \leftarrow \lceil p^{0.5} \rceil$
    $baby\_steps \leftarrow \{\}$, $baby\_steps\_inverse \leftarrow \{\}$
    $x \leftarrow 1$
    for $j \leftarrow 0$ to $m - 1$ do
        $baby\_steps[j] \leftarrow x$
        $baby\_steps\_inverse[x] \leftarrow j$
        $x \leftarrow (x \cdot g) \bmod p$
    $giant\_step \leftarrow g^{-m} \bmod p$
    $y \leftarrow h$
    for $i \leftarrow 0$ to $m - 1$ do
        if $y \in baby\_steps.values()$ then
            return $i \cdot m + baby\_steps\_inverse[y]$
        $y \leftarrow (y \cdot giant\_step) \bmod p$
    return "No solution found"
```

2.1.3. Example

As can be seen in file **bsgs.py**, our implementation was run on example with $g = 2$, $h = 15$, $n = 29$ and the obtained result was $x = 27$, in accordance with the fact that $2^{27} \equiv 15 \pmod{29}$.

2.1.4. Use and complexity analysis

This algorithm works for a generic finite cyclic group, although it works most efficiently on groups for which the group order is relatively small and has a known factorization, or prime. In particular, if the order of the group can be written as $n = p - 1 = q_1^{e_1} \cdot q_2^{e_2} \cdots q_k^{e_k}$, where p is a prime number and $q_1, q_2, \ldots, q_k$ are distinct prime factors, then the algorithm can be applied using the largest prime factor q_k as the baby step size, and the product of the other prime factors $q_1^{e_1} \cdot q_2^{e_2} \cdots q_{k-1}^{e_{k-1}}$ as the giant step size.

When one uses an efficient lookup scheme, such as hash table, the algorithm has a running time of $\sum_{k=1}^{\sqrt{n}} \log k = \mathcal{O}(\sqrt{n} \log n)$. Since it requires $\mathcal{O}(\sqrt{n})$ memory to store the baby steps, it is well-suited for groups where the order is small enough to fit in memory, but not too small to make the algorithm impractical.

If the order of the group is composite then the, soon to be discussed, Pohlig–Hellman algorithm is more efficient.

2.2. Pohlig-Hellman algorithm

Building on BSGS algorithm described in previous section 2.1, we construct a more sophisticated one: *Pohlig-Hellman algorithm*.

This algorithm is a way to solve the discrete logarithm problem (DLP) in a finite cyclic group of composite order $n = \Pi_{i=1}^{r} p_i^{e_i}$. As before, given a generator g and an element h of the group, the DLP asks to find the integer x such that $g^x = h \pmod{n}$.

2.2.1. High level description

The key idea behind the algorithm is that, by using the prime factorization of n, we can reduce the initial group of order n into a set of groups of smaller orders for which the DLP can be solved more easily — using the BSGS algorithm. Then, by using the Chinese Remainder Theorem, we can combine the solutions for each factor to obtain the solution for the original group.

2.2.2. Pseudocode

```
function ChineseRemainderTheorem($a, m$)
    $sum \leftarrow 0$
    $M \leftarrow$ reduce(mul, m, 1)
    for $i \leftarrow 0$ to len($a$) $- 1$ do
        $p \leftarrow M/m[i]$
        $sum \leftarrow sum + a[i] \cdot p \cdot (p^{-1} \bmod m[i])$
    return $sum \bmod M$
function Factors($n$)
    $factorisation \leftarrow []$
    $f \leftarrow 2$
    while $f^2 \leq n$ do
```

```
        **if** $n \bmod f = 0$ **then**
            $exp \leftarrow 0$
            **while** $n \bmod f = 0$ **do**
                $exp \leftarrow exp + 1$
                $n \leftarrow n/f$
            $factorisation$.append([$f$, $exp$])
        $f \leftarrow f + 1$
    **if** $n > 1$ **then**
        $factorisation$.append([$n$, 1])
    **return** $factorisation$
**function** PohligHellman($g, h, p, factors$)
    $x \leftarrow [], pe \leftarrow []$
    **for** $i \leftarrow 0$ to length($factors$) $- 1$ **do**
        $pe$.append($factors[i][0]^{factors[i][1]}$)
        $g_i \leftarrow g^{p/pe[i]} \bmod p$
        $h_i \leftarrow h^{p/pe[i]} \bmod p$
        $x$.append(BabyStepGiantStep($g_i$, $h_i$, $p$) $\bmod pe[i]$)
    $dl \leftarrow$ ChineseRemainderTheorem($x$, $pe$)
    **return** $dl$
```

2.2.3. Example

As shown in file **ph.py**, we first note that our implementation calls the implemented BSGS script. Furthermore, this algorithm was run on example with slightly larger numbers than BSGS (and composite as well): $g = 6$, $h = 7531$, $n = 8101$ and the obtained result was $x = 6689$, in accordance with the fact that $6^{6689} \equiv 17531 \pmod{8101}$.

2.2.4. Use and complexity analysis

To solve the DLP in a group of order $p_i^{e_i}$, the algorithm uses the BSGS algorithm, which has a time complexity of $\mathcal{O}(\sqrt{p_i^{e_i}} \log p_i^{e_i}) = \mathcal{O}(p_i^{e_i/2} \log p_i^{e_i})$. From this, we can also see that PH algorithm degrades to BSGS algorithm for groups of prime order and has a $\mathcal{O}(\sqrt{n} \log n)$ time complexity.

In a generic case, however, the time complexity calculation is similar to BSGS one and based on $n = \Pi_{i=1}^r p_i^{e_i}$ and it reduces to: $\mathcal{O}\left(\sum_{i=0}^r e_i(\log n + \sqrt{p_i})\right)$.

Regarding the space complexity, it stems from the BSGS part of the algorithm and is thus equal to $\mathcal{O}\left(\max\left\{p_i : i \in [1 \cdot \cdot r]\right\}\right)$.

2.3. Nonexhaustive overview of classical algorithms

- Baby-Step Giant-Step algorithm:

  This algorithm has a time complexity of $\mathcal{O}(\sqrt{\log n})$, where n is the order of the group. It is based on the observation that any element g^x in the group can be written as g^{i*m+j}, where m is the smallest integer such that $m > \sqrt{n}$, and i and j are integers between 0 and $m-1$. The algorithm works by precomputing a table of m baby steps g^j and then using giant steps g^{im} to find a match between a baby step and a giant step.

- Pollard rho algorithm:

  This algorithm also has a time complexity of $\mathcal{O}(\sqrt{n})$. It works by randomly generating a sequence of elements in the group and using a function to produce a pseudorandom sequence. The algorithm looks for matches in the sequence, which correspond to two elements of the group that have the same discrete logarithm. The time complexity of the algorithm depends on the expected number of iterations required to find a collision, which is proportional to the square root of the order of the group.

- Pohlig-Hellman algorithm:

  This algorithm has a time complexity of $\mathcal{O}\left(\sum_{i=0}^{r} e_i(\log n + \sqrt{p_i})\right)$, where $n = \Pi_{i=1}^{r} p_i^{e_i}$ is the order of the group. The algorithm works by reducing the DLP in the prime order group n to a series of DLPs in smaller order groups using n's factorization. It solves DLP in each of these smaller groups using the Baby-Step Giant-Step algorithm.

- Index calculus algorithm:

  This algorithm has a time complexity of $\mathcal{O}(\exp\left((c + o(1))(\log n \log\log n)^{1/2}\right))$, where n is the order of the group and c is a constant, dependent upon algorithm details[1]. The algorithm works by constructing a system of linear equations involving the discrete logarithms of a set of "smooth" elements of the group. After this, it solves the system using linear algebra. The time complexity of the algorithm depends on the number of equations required to solve the system, which is proportional to the size of the "smooth" factor base.

In general, the Baby-Step Giant-Step algorithm and Pollard rho algorithm are the most efficient classical algorithms for solving the DLP in small to moderate size groups. The index calculus algorithm is more efficient for larger groups, but is often impractical due to its high memory requirements. The Pohlig-Hellman algorithm is most efficient when the prime factors of the order of the group are small.

[1]There are several algorithms in the Index calculus family.

3. Quantum algorithms

As a complete paradigm shift, we ought to venture outside of classical world: there is no known efficient (polynomial) classical algorithm for the discrete log problem that works on all inputs. Thus, we turn our attention to quantum mechanics. Starting with Deutsch and Josza, there were several exponentially costing (although somewhat contrived) problems shown to be solvable on a quantum computer in polynomial time. However, a true breakthrough occurred in 1994 when Peter Shor found an algorithm for number factorisation with polynomial time complexity. This problem and several other problems are actually just cases of the *hidden subgroup problem*, and one of them is Shor's algorithm for the discrete logarithm as well.

3.1. Shor's algorithm

The basic idea behind Shor's algorithm is to use a quantum Fourier transform to find the period of a certain function. This function is chosen such that its period is related to the discrete logarithm we want to find. Once the period is found, we can use classical algorithms to find the factors of the integer, which can then be used to solve the discrete logarithm problem.

The time complexity of Shor's algorithm is $\mathcal{O}(\log^3 n)$, where n is the group order. This is exponentially faster than the best-known classical algorithms for DLP (e.g. those in index calculus familiy), which have a time complexity of $\mathcal{O}(\exp\left((c + o(1))(\log n \log\log n)^{1/2}\right))$.

However, it is important to note that the implementation of Shor's algorithm requires a large-scale quantum computer with a sufficient number of qubits and low error rates: this is the main technical difficulty in implementing Shor's algorithm to actually be useful — the need for a large-scale, fault-tolerant quantum computer. The algorithm requires the ability to perform a large number of quantum gates with high accuracy, and any errors in the computation can quickly cause the output to become meaningless. Additionally, the algorithm requires the ability to prepare and measure quantum states with high fidelity, which can also be difficult in practice. Finally, it also requires a number of quantum gates that grows exponentially with the size of the input, which becomes another nontrivial problem for large numbers.

Shor's algorithm is most efficient for integers n that are large and have many small prime factors. In particular, the algorithm can be used to efficiently factor integers that are the product of two large prime numbers. For integers with larger prime factors or with few prime factors, the algorithm is less efficient and requires a larger number of qubits and quantum gates. Additionally, the algorithm is not known to be efficient for non-prime moduli, although recent research has suggested that it may be possible to adapt the algorithm for use in this case.

3.1.1. High level overview

For this algorithm, we do not write the explicit pseudocode since it requires a lot of quantum concepts to be introduced. However, we do provide a high level overview of its main steps.

1. Initialization: The first step of the algorithm is to initialize two quantum registers, one to store the inputs and the other to store the output of the algorithm (the discrete logarithm). The size of the quantum registers depends on the size of the input number and the desired level of precision.
2. Superposition: The next step is to put the first quantum register into a superposition of all possible values using the Hadamard gate. This creates a uniform distribution of probabilities for all possible values of the input numbers.
3. Modular exponentiation: The key step in Shor's algorithm is to perform modular exponentiation on the input generator g. This is done using a unitary operator that takes the form $U|x\rangle|y\rangle = |x\rangle|y \oplus f(x)\rangle$, where N is the order of the group and $|x\rangle$ and $|y\rangle$ are the states in two quantum registers. Function $f(x)$ includes modular arithmetic operations and modular exponentiation and can be efficiently implemented using a quantum circuit.
4. Quantum Fourier transform: After applying the modular exponentiation operator, we apply the quantum Fourier transform to the first quantum register. This transforms the superposition of states into a Fourier basis, which enables us to extract information about the period of the function.
5. Measurement: Once the quantum Fourier transform has been applied, we measure the outputs of the first register and obtain two numbers of interest.
6. Classical post-processing: After obtaining the aforementioned two numbers, generalized continued fractions algorithm is applied to them, and then, with a high probability, the desired solution to the discrete logarithm problem.

3.1.2. Example

In the example[1] provided in Github repository of this assignment: **q.ipynb**, with $g = 13$ and $n = 15$, we see that the outputs of register of interest (containing two numbers) yield the following pairs $(x, h) \in \{(12, 1), (14, 4), (1, 13)\}$, in accordance with: $13^{12} \equiv 1 \pmod{15}, 13^{14} \equiv 4 \pmod{15}, 13^{1} \equiv 13 \pmod{15}$, respectively.

[1]Original URL: https://github.com/DavidNadaly/Discrete-Logarithm

4. Conclusion and future prospects

The discrete logarithm problem is an important mathematical problem that has numerous applications in fields such as cryptography and computer science. In this assignment, we described the DLP and presented three algorithms for solving it: two classical algorithms, namely, the baby-step giant-step and Pohlig-Hellman algorithsm, and one quantum algorithm, namely, Shor's algorithm.

For each of the algorithms, we provided an implementation, an example of use, and a complexity and useful range overview. The baby-step giant-step algorithm uses a memory-efficient approach to solve the DLP, with a time complexity of $\mathcal{O}(\sqrt{n}\log n)$, making it more efficient than the brute-force algorithm for larger inputs, which is of complexity $\mathcal{O}(n)$. Pohlig-Hellman algorithm provides an even better average time complexity, however it reduces to BSGS for prime-order groups.

On the other hand, Shor's algorithm is a quantum algorithm that can efficiently solve the DLP using a quantum computer. It has a polynomial time complexity of $\mathcal{O}(\log^3 N)$, making it significantly faster than the classical algorithms for large inputs, since the best classical algorithms have time complexity of $\mathcal{O}(\exp\big((c+o(1))(\log n \log\log n)^{1/2}\big))$.

As a possible future direction for this assignment to be expanded, one could implement the other classical algorithms, along with their analysis. Similarly, the quantum algorithm could be generalized for a wider range of groups and a circuit scheme, along with detailed explanation of its quantum aspects, could be included in the report.