Implementation of Depth First Search to Solve the Discrete Logarithm Problem with a Multiple-Element Generating Set

Zeki Amani – 13524082
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: zekiamanioke@gmail.com, 13524082@std.stei.itb.ac.id

***Abstract*—This paper discusses the application of Depth First Search (DFS) to solve a modified form of the Discrete Logarithm Problem in a finite group. Instead of using one generator, the problem uses a generating set consisting of multiple elements and asks for a sequence of generator operations that produces a target element. The group is modeled as a Cayley graph, where every group element is a vertex and every generator operation is an edge. The proposed DFS method starts from the identity element, explores one possible path deeply, and stores parent information to reconstruct a valid representation when the target is found. This paper aim to give better approach to Discrete Logarithm Problem than brute focre.**

***Index Terms*—Discrete Logarithm Problem, Cayley graph, Depth First Search**

I. Introduction

The Discrete Logarithm Problem (DLP) is an important problem in modern cryptography. The difficulty of solving discrete logarithms in large finite groups is closely related to the security of several public-key cryptographic systems, including Diffie–Hellman key exchange [2]. Therefore, finding an efficient algorithm for solving DLP would have a significant impact on the security of many cryptographic applications.

Many algorithms have been developed to solve the Discrete Logarithm Problem. However, solving DLP in groups with cryptographic-sized parameters remains computationally difficult. This paper does not aim to propose a practical attack against real cryptographic systems. Instead, it explores how a variation of the problem in a finite group with a relatively small order can be modeled and solved using a graph search algorithm.

The problem discussed in this paper extends the usual representation of an element using one generator. Instead, the group is described using a generating set consisting of multiple elements. The objective is to find a representation of a target element using the elements in the generating set.

The purpose of this paper is to demonstrate the application of DFS to a problem involving cyclic groups and generating sets. DFS is used to explore one possible sequence of generator operations at a time and to backtrack when that sequence cannot reach an unvisited element. The objective of this paper is to find one valid representation of the target element. The proposed method is evaluated only on finite groups with small orders because the traversal may still require substantial time and memory when the number of group elements becomes large. Even so, later will be shown that this approach can be better than brute force approach.

II. Theoretical Foundations

This section explains several mathematical and algorithmic concepts that are used in this paper. These concepts include group, cyclic group and generator, Discrete Logarithm Problem, Cayley graph, and Depth First Search.

A. Group

A group is a mathematical structure consisting of a set of elements and a binary operation. A group is usually denoted as $(G, \circ)$, where G is a set and $\circ$ is an operation between two elements of G.

For $(G, \circ)$ to be called a group, it must satisfy four properties [1].

1) *Closure*. For every $a, b \in G$, the result of $a \circ b$ must also be an element of G.
2) *Associativity*. For every $a, b, c \in G$,
$$(a \circ b) \circ c = a \circ (b \circ c). \tag{1}$$
3) *Identity element*. There exists an element $e \in G$ such that, for every $a \in G$,
$$a \circ e = e \circ a = a. \tag{2}$$
4) *Inverse element*. For every $a \in G$, there exists an element $a^{-1} \in G$ such that
$$a \circ a^{-1} = a^{-1} \circ a = e. \tag{3}$$

One example of a group is $(\mathbb{Z}, +)$, which is the set of integers with addition as its operation. The identity element is 0 because

$$a + 0 = 0 + a = a \tag{4}$$

for every integer a. Every element also has an inverse. For example, the inverse of 5 is -5 because

$$5 + (-5) = 0. \tag{5}$$

Another example is the multiplicative group modulo a prime number p, written as

$$\mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}. \tag{6}$$

The operation in this group is multiplication modulo p. For example, in $\mathbb{Z}_7^*$,

$$3 \cdot 5 \equiv 1 \pmod 7. \tag{7}$$

Therefore, 5 is the multiplicative inverse of 3 modulo 7.

B. Cyclic Group, Generator, and Generating Set

A cyclic group is a group in which every element can be generated from one particular element through repeated applications of the group operation. This particular element is called a generator [1].

If g is a generator of a group G, then the group can be written as

$$G = \langle g \rangle. \tag{8}$$

For a multiplicative group, the cyclic group generated by g is

$$\langle g \rangle = \{g^k \mid k \in \mathbb{Z}\}. \tag{9}$$

Meanwhile, for an additive group, the cyclic group generated by g is

$$\langle g \rangle = \{kg \mid k \in \mathbb{Z}\}. \tag{10}$$

For example, consider the additive group $\mathbb{Z}_6 = \{0, 1, 2, 3, 4, 5\}$ under addition modulo 6. The element 1 is a generator because repeated addition of 1 produces every element in the group:

$$\begin{aligned} 0 \cdot 1 &\equiv 0 \pmod 6, \\ 1 \cdot 1 &\equiv 1 \pmod 6, \\ 2 \cdot 1 &\equiv 2 \pmod 6, \\ 3 \cdot 1 &\equiv 3 \pmod 6, \\ 4 \cdot 1 &\equiv 4 \pmod 6, \\ 5 \cdot 1 &\equiv 5 \pmod 6. \end{aligned} \tag{11}$$

Therefore,

$$\mathbb{Z}_6 = \langle 1 \rangle. \tag{12}$$

In a finite cyclic group G, an element g is a generator if its order is equal to the number of elements in the group. This can be written as

$$\operatorname{ord}(g) = |G|. \tag{13}$$

The order of an element g is the smallest positive integer k such that

$$g^k = e, \tag{14}$$

where e is the identity element of the group. For an additive group, the equivalent condition is

$$kg = e. \tag{15}$$

A group can also be generated by more than one element. Let

$$S = \{s_1, s_2, \ldots, s_m\} \subseteq G. \tag{16}$$

The set S is called a *generating set* of G if every element in G can be obtained by combining elements of S and their inverses [1]. The group generated by S is denoted by

$$G = \langle S \rangle = \langle s_1, s_2, \ldots, s_m \rangle. \tag{17}$$

For an additive group, the subgroup generated by S can be written as

$$\langle S \rangle = \{x_1 s_1 + x_2 s_2 + \cdots + x_m s_m \mid x_1, x_2, \ldots, x_m \in \mathbb{Z}\}. \tag{18}$$

The members of a generating set do not have to be generators individually. For example, consider again the additive group $\mathbb{Z}_6$. The element 2 is not a generator because repeated addition of 2 only produces

$$\langle 2 \rangle = \{0, 2, 4\}. \tag{19}$$

Similarly, the element 3 is not a generator because

$$\langle 3 \rangle = \{0, 3\}. \tag{20}$$

However, the set $\{2, 3\}$ generates the entire group $\mathbb{Z}_6$. Every element of the group can be expressed as a combination of 2 and 3 modulo 6:

$$\begin{aligned} 0 &\equiv 0 \pmod 6, \\ 1 &\equiv 3 - 2 \pmod 6, \\ 2 &\equiv 2 \pmod 6, \\ 3 &\equiv 3 \pmod 6, \\ 4 &\equiv 2 + 2 \pmod 6, \\ 5 &\equiv 2 + 3 \pmod 6. \end{aligned} \tag{21}$$

Therefore,

$$\mathbb{Z}_6 = \langle 2, 3 \rangle. \tag{22}$$

Thus, a generator is a single element that generates the entire group, whereas a generating set is a collection of elements that generates the group together. A cyclic group always has at least one generator, but it can also be described using a generating set with more than one element.

The concept of a generating set gives a more general way to express an element of a group. If $S = \{s_1, s_2, \ldots, s_m\}$

generates a group G, then every element $h \in G$ can be represented as a product of elements from S and their inverses. In general, this representation can be written as

$$h = s_{i_1}^{\epsilon_1} s_{i_2}^{\epsilon_2} \cdots s_{i_k}^{\epsilon_k}, \tag{23}$$

where $s_{i_j} \in S$ and $\epsilon_j \in \{-1, 1\}$.

If the group is abelian, the order of multiplication does not affect the result. Therefore, the representation in (23) can be written more compactly as

$$h = s_1^{x_1} s_2^{x_2} \cdots s_m^{x_m}, \tag{24}$$

where $x_1, x_2, \ldots, x_m \in \mathbb{Z}$.

C. Cayley Graph

A Cayley graph is a graph constructed from a group and a generating set. It represents the elements of a group as vertices and represents group operations as edges. Therefore, a Cayley graph provides a way to visualize the structure of a group using graph theory [3].

Let G be a group and let

$$S = \{s_1, s_2, \ldots, s_m\} \tag{25}$$

be a generating set of G. The directed Cayley graph of G with respect to S is denoted by

$$\mathrm{Cay}(G, S). \tag{26}$$

The vertices in the Cayley graph are all elements of the group. Thus, the vertex set is defined as

$$V = G. \tag{27}$$

For every vertex $u \in G$ and every element $s \in S$, there is a directed edge from u to another vertex v if

$$v = u \cdot s. \tag{28}$$

Therefore, every edge represents the result of applying one element from the generating set. The edge can be labeled by the corresponding element s to indicate which group operation is used.

In general, the Cayley graph can be written as

$$\mathrm{Cay}(G, S) = (V, E), \tag{29}$$

where the edge set is

$$E = \{(u, u \cdot s) \mid u \in G,\ s \in S\}. \tag{30}$$

For additive groups, the group operation is addition. Thus, the edge rule in (28) becomes

$$v = u + s. \tag{31}$$

Consider the additive group

$$G = \mathbb{Z}_6 = \{0, 1, 2, 3, 4, 5\}, \tag{32}$$

with the generating set

$$S = \{1\}. \tag{33}$$

Since repeated addition of 1 modulo 6 produces every element in $\mathbb{Z}_6$, the element 1 is a generator of the group. The Cayley graph of this group is

$$\mathrm{Cay}(\mathbb{Z}_6, \{1\}). \tag{34}$$

Every vertex u has one outgoing edge to the vertex

$$v \equiv u + 1 \pmod 6. \tag{35}$$

For example, several edges in the graph represent the following operations:

$$\begin{aligned} 0 + 1 &\equiv 1 \pmod 6, \\ 1 + 1 &\equiv 2 \pmod 6, \\ 2 + 1 &\equiv 3 \pmod 6, \\ 5 + 1 &\equiv 0 \pmod 6. \end{aligned} \tag{36}$$

The last equation shows that the graph returns from vertex 5 to vertex 0. Therefore, the Cayley graph forms a directed cycle, as shown in Fig. 1.

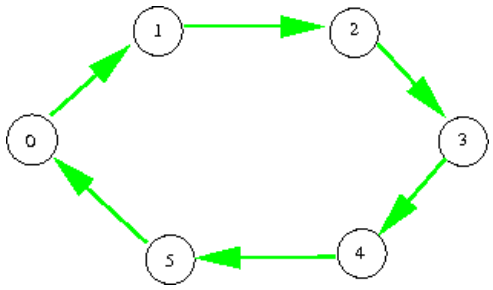

Fig. 1. Directed Cayley graph $\mathrm{Cay}(\mathbb{Z}_6, \{1\})$. Each edge from u to v represents the operation $v \equiv u + 1 \pmod 6$.

A path in a Cayley graph corresponds to a sequence of group operations. For example, the path

$$0 \to 1 \to 2 \to 3 \tag{37}$$

represents repeated addition of the generator 1. Thus, the path corresponds to

$$3 \equiv 1 + 1 + 1 \pmod 6. \tag{38}$$

More generally, suppose that a path from the identity element e to a target element h is labeled by

$$a_1, a_2, \ldots, a_l, \tag{39}$$

where every $a_i \in S$. The path represents the group equation

$$h = a_1 a_2 \cdots a_l. \tag{40}$$

The number of edges in the path is equal to the number of generator operations used to obtain h from the identity element.

The example in Fig. 1 uses only one generator, so each vertex has only one possible outgoing edge. In the proposed method, the generating set may contain more than one element. If

$$S = \{s_1, s_2, \ldots, s_m\}, \tag{41}$$

then every vertex u has one possible transition for every element in S:

$$u \to u \cdot s_1, \quad u \to u \cdot s_2, \quad \ldots, \quad u \to u \cdot s_m. \tag{42}$$

This creates multiple possible paths from the identity element to a target element.

D. Discrete Logarithm Problem

The Discrete Logarithm Problem, commonly abbreviated as DLP, is a problem of finding an exponent when the base and the result of an exponentiation are known. Let $G = \langle g \rangle$ be a cyclic group and let $h \in G$. The DLP asks for an integer x such that [2]

$$g^x = h. \tag{43}$$

Since g is a generator of G, every element $h \in G$ can be written as a power of g. If G is finite and has order n, the exponent x is usually considered modulo n. Therefore, the solution can be represented by a value in the range

$$x \in \{0, 1, \ldots, n-1\}. \tag{44}$$

In a multiplicative group modulo a prime number p, the DLP is commonly written as

$$g^x \equiv h \pmod{p}. \tag{45}$$

For example, consider the multiplicative group $\mathbb{Z}_7^*$ with generator $g = 3$. Suppose the target element is $h = 6$. The problem is to find x such that

$$3^x \equiv 6 \pmod{7}. \tag{46}$$

The solution is $x = 3$, because

$$3^3 = 27 \equiv 6 \pmod{7}. \tag{47}$$

Calculating the value of $g^x \bmod p$ is relatively easy using modular exponentiation. However, finding x when only g, h, and p are known becomes computationally difficult when the group has a very large number of elements. This property makes DLP important in many cryptographic applications [2].

The standard DLP uses one generator g. This can be viewed as a special case of the more general representation of an element using a generating set. Let

$$S = \{s_1, s_2, \ldots, s_m\} \tag{48}$$

be a generating set of an abelian group G. An element $h \in G$ can be represented as

$$h = s_1^{x_1} s_2^{x_2} \cdots s_m^{x_m}. \tag{49}$$

The objective in this representation is to find an exponent vector

$$(x_1, x_2, \ldots, x_m) \tag{50}$$

that satisfies (49). Unlike the standard DLP, the exponent vector does not necessarily have a unique solution because different combinations of elements in S may produce the same target element.

For example, consider the multiplicative group $\mathbb{Z}_7^*$ and the generating set

$$S = \{2, 6\}. \tag{51}$$

The element 2 is not a generator of $\mathbb{Z}_7^*$ because

$$\langle 2 \rangle = \{1, 2, 4\}. \tag{52}$$

Similarly, the element 6 is not a generator because

$$\langle 6 \rangle = \{1, 6\}. \tag{53}$$

However, together they generate the entire group:

$$\mathbb{Z}_7^* = \langle 2, 6 \rangle. \tag{54}$$

For the target element $h = 3$, one possible representation is

$$3 \equiv 2^2 \cdot 6^1 \pmod{7}, \tag{55}$$

because

$$2^2 \cdot 6 = 24 \equiv 3 \pmod{7}. \tag{56}$$

Thus, one valid exponent vector is

$$(x_1, x_2) = (2, 1). \tag{57}$$

If $G = \langle g \rangle$ is a cyclic group of order n, every element of a generating set can be expressed as a power of g. Suppose that

$$s_i = g^{a_i} \tag{58}$$

for every $s_i \in S$, and suppose that the target element can be written as

$$h = g^y. \tag{59}$$

Then the representation in (49) can be rewritten as

$$g^y = g^{a_1 x_1} g^{a_2 x_2} \cdots g^{a_m x_m}. \tag{60}$$

Therefore, the exponent vector must satisfy the following modular linear congruence:

$$a_1 x_1 + a_2 x_2 + \cdots + a_m x_m \equiv y \pmod{n}. \tag{61}$$

This shows that the ordinary DLP is the case where the generating set contains only one element. When a generating set contains more than one element, an element of the group may have multiple possible representations.

E. Depth First Search

Depth First Search, commonly abbreviated as DFS, is a graph traversal algorithm that explores a path as deeply as possible before it returns to a previous vertex and explores another path. Starting from a selected vertex, DFS visits one unvisited adjacent vertex and continues from that vertex. When the current vertex has no unvisited adjacent vertex, DFS backtracks to the most recent vertex that still has an unvisited adjacent vertex [4].

DFS can be implemented using recursion or an explicit *stack* data structure. A stack follows the Last In First Out (LIFO) principle, which means that the most recently inserted vertex is the first vertex that will be processed again. In the proposed implementation, a *visited* set is used so that a group element is not explored more than once. This is necessary because a finite Cayley graph may contain cycles.

The general procedure of DFS is as follows [4].

1) Mark the starting vertex as visited.
2) Check whether the current vertex is the target vertex.
3) Visit one unvisited adjacent vertex of the current vertex.
4) Continue the search recursively from that adjacent vertex.
5) If the current vertex has no unvisited adjacent vertex, return to its parent vertex.
6) Repeat the process until the target is found or every reachable vertex has been visited.

For example, consider a graph where vertex A is connected to vertices B and C. Vertex B is connected to D, while vertex C is connected to E. If DFS starts from vertex A and processes B before C, one possible traversal order is

$$A \to B \to D \to C \to E. \tag{62}$$

The algorithm first follows the path from A to B and then to D. Only after it can no longer continue from D does it backtrack to A and explore the branch that begins at C.

For a graph with vertex set V and edge set E, DFS has time complexity

$$O(|V| + |E|), \tag{63}$$

when every vertex and edge is processed at most once. The memory complexity is $O(|V|)$ in the worst case because DFS stores the visited vertices, parent information, and the current recursion or stack path. Unlike BFS, DFS does not find the path with the minimum number of edges. However, DFS is sufficient for this paper because the required output is one valid representation of the target element, not necessarily the shortest representation.

III. Proposed Solution

The main idea of this method is to traverse the Cayley graph without constructing the whole graph in advance. Each group element is generated only when DFS needs to explore it.

A. Problem Formulation

Let p be a prime number and let

$$S = g_0, g_1, \ldots, g_{m-1} \tag{64}$$

be a set of generators in $\mathbb{Z}_p^*$. Given a target element h, the objective is to find nonnegative exponents

$$x_0, x_1, \ldots, x_{m-1} \tag{65}$$

such that

$$h \equiv g_0^{x_0} g_1^{x_1} \cdots g_{m-1}^{x_{m-1}} \pmod{p}. \tag{66}$$

B. Cayley Graph Representation

The problem can be represented as a directed Cayley graph. Each vertex represents an element of $\mathbb{Z}_p^*$ that can be reached from the identity element 1. For every current vertex u and every generator $g_i \in S$, there is a directed edge from u to another vertex v, where

$$v \equiv u \cdot g_i \pmod{p}. \tag{67}$$

The traversal starts from the identity element 1 because

$$1 = g_0^0 g_1^0 \cdots g_{m-1}^0. \tag{68}$$

Every edge represents one multiplication by a generator. Therefore, a path from 1 to h represents a valid expression of h using the generators in S.

For example, suppose a path uses the generators in the following order:

$$g_0, g_2, g_0, g_1. \tag{69}$$

Then, the target element produced by this path is

$$h \equiv g_0 \cdot g_2 \cdot g_0 \cdot g_1 \equiv g_0^2 g_1 g_2 \pmod{p}. \tag{70}$$

Thus, the exponent vector for this path is

$$(x_0, x_1, x_2) = (2, 1, 1). \tag{71}$$

C. Graph Construction

As already said before, the Cayley graph is not constructed completely before the DFS traversal begins. Instead, the graph is generated dynamically when DFS is about to explore a vertex.

Suppose DFS is currently at vertex u. At this point, the outgoing edges of u are generated one by one by multiplying u with every generator $g_i \in S$. The resulting next vertex is calculated using

$$v \equiv u \cdot g_i \pmod{p}. \tag{72}$$

Therefore, only vertices and edges that are actually needed during the search are generated. If DFS finds the target element before exploring all reachable vertices, the remaining vertices and edges do not need to be generated.

D. DFS Traversal

DFS explores one branch of the Cayley graph as deeply as possible before returning to explore another branch. Starting from a current element u, DFS first checks whether u is equal to the target element h.

If u is not equal to h, DFS generates and explores its neighboring vertices. For every generator g_i, DFS calculates

$$v \equiv u \cdot g_i \pmod{p}. \tag{73}$$

Before continuing to v, the exponent of g_i is increased by one and the selected generator is added to the current path. DFS then continues recursively from v.

If a branch reaches the target element, the current path and exponent vector are returned as the solution. Otherwise, DFS performs backtracking. The last generator is removed from the current path, its exponent is decreased by one, and another generator is tried from the same vertex.

The general procedure is as follows.

1) Start from the identity element 1.
2) Check whether the current element is equal to the target h.
3) Mark the current element as visited.
4) Generate the outgoing edges of the current element one by one.
5) For every generator g_i, calculate [v $\equiv u \cdot g_i \pmod{p}$.]
5) Increase the exponent of g_i and add the generator to the current path.
6) Continue DFS from v.
7) If the target is found, return the current path and exponent vector.
8) Otherwise, undo the previous step and try another generator.

E. Cycle Prevention

Since the group is finite, repeated multiplication by generators may create cycles. For example, multiplying an element repeatedly by the same generator may eventually return to a previously explored element.

To prevent DFS from traversing the same cycle indefinitely, a visited set is used. Once a vertex has been explored, it is marked as visited. If DFS reaches a vertex that has already been visited, that vertex is not explored again.

Using the visited set guarantees that every reachable vertex is explored at most once. Therefore, the traversal terminates even when the target element does not belong to the subgroup generated by S.

F. Solution Representation

The result of the search consists of a path and an exponent vector. The path records the order of generator multiplications used to reach the target. The exponent vector records the total number of times each generator appears in that path.

Suppose the returned exponent vector is

$$(x_0, x_1, \ldots, x_{m-1}). \tag{74}$$

The solution is valid if

$$g_0^{x_0} g_1^{x_1} \cdots g_{m-1}^{x_{m-1}} \equiv h \pmod{p}. \tag{75}$$

Since multiplication in $\mathbb{Z}_p^*$ is commutative, the order of generators in the path does not affect the final product. However, the path is still useful because it shows the sequence of vertices explored by DFS.

G. Complexity Analysis

Let R be the number of elements reachable from the identity element 1 using the generator set S, and let m be the number of generators.

In DFS, every reachable element is marked as visited after it is explored. Therefore, the same element will not be explored repeatedly even when it can be obtained through different sequences of generators. For every reachable element, DFS tries at most m generator multiplications. Thus, the worst-case time complexity of DFS is

$$O(Rm). \tag{76}$$

For a problem with two generators g_1 and g_2, the value of m is equal to 2. Therefore, the time complexity becomes

$$O(2R) = O(R). \tag{77}$$

If g_1 and g_2 generate the entire group $\mathbb{Z}_p^*$, then there are at most $p-1$ reachable elements. In this case, the worst-case complexity of DFS is

$$O(p-1). \tag{78}$$

This can be compared with a brute force approach. Brute force searches for every possible pair of exponents (i, j) such that

$$g_1^i g_2^j \equiv h \pmod{p}. \tag{79}$$

Let r_1 be the order of g_1 and let r_2 be the order of g_2. Since exponent values repeat after reaching the order of a generator, brute force only needs to check

$$0 \le i < r_1 \tag{80}$$

and

$$0 \le j < r_2. \tag{81}$$

Thus, brute force must examine at most

$$r_1 r_2 \tag{82}$$

pairs of exponents. Its time complexity is therefore

$$O(r_1 r_2). \tag{83}$$

In the worst case, both generators may have order $p-1$. Therefore, brute force may require

$$O((p-1)^2) \tag{84}$$

candidate checks.

The main difference is that brute force explores exponent pairs, while DFS explores group elements. Different exponent pairs may produce the same group element. For example, two different pairs (i, j) and (a, b) may satisfy

$$g_1^i g_2^j \equiv g_1^a g_2^b \pmod{p}. \tag{85}$$

Brute force still checks both pairs separately. On the other hand, DFS marks the resulting group element as visited, so that element is explored only once.

Therefore, DFS can reduce repeated searches caused by different exponent combinations that lead to the same residue. In the case where both generators generate the entire group, DFS has complexity $O(p-1)$, while naive brute force can have complexity $O((p-1)^2)$.

However, DFS requires additional memory to store visited elements and the current traversal path. Its space complexity is

$$O(R+m). \tag{86}$$

Brute force only stores the current exponent values, so its space complexity is

$$O(1). \tag{87}$$

Thus, DFS uses more memory but can be more time efficient than brute force.

IV. Implementation

The implementation of this algorithm is made in golang and is separated into 2 part which is main, and solver.

A. *solver.go*

This file contains the solving process using graph tranversal as discussed before.

```
package main

import "math/big"

type Step struct {
    generatorIndex int
    value          int64
}

type SearchResult struct {
    exponents []int
    path      []Step
}

func multiplyModulo(left int64, right int64, modulus int64) int64 {
    product := new(big.Int).Mul(big.NewInt(left), big.NewInt(right))
    product.Mod(product, big.NewInt(modulus))
    return product.Int64()
}

func solveDLP(target int64, generators []int64, modulus int64) (SearchResult, bool)
      {
    exponents := make([]int, len(generators))
    path := []Step{}
    visited := map[int64]bool{}
    return dfs(target, modulus, generators, 1, exponents, path, visited)
}

func dfs(target int64, modulus int64, generators []int64, current int64, exponents
      []int, path []Step, visited map[int64]bool) (SearchResult, bool) {
    if current == target {
        return SearchResult{
            exponents: copyInts(exponents),
            path:      copySteps(path),
        }, true
    }
    if visited[current] {
        return SearchResult{}, false
    }
    visited[current] = true
    for i, generator := range generators {
        nextValue := multiplyModulo(current, generator, modulus)
        exponents[i]++
        path = append(path, Step{
            generatorIndex: i,
            value:          nextValue,
        })
        result, found := dfs(target, modulus, generators, nextValue, exponents, path
      , visited)
        if found {
            return result, true
        }
        path = path[:len(path)-1]
        exponents[i]--
    }
    return SearchResult{}, false
}

func copyInts(values []int) []int {
    copied := make([]int, len(values))
    copy(copied, values)
    return copied
}

func copySteps(values []Step) []Step {
    copied := make([]Step, len(values))
    copy(copied, values)
    return copied
}

func gcd(left int64, right int64) int64 {
    if left < 0 {
        left = -left
    }
    if right < 0 {
        right = -right
    }
    for right != 0 {
        left, right = right, left%right
    }
    return left
}
```

Listing 1. solver.go

B. *main.go*

This file contain the user interaction part where it take inputs from user and output the result back

```
package main

import "fmt"

func main() {
    fmt.Println("DLP solver")

    var modulus int64
    fmt.Print("modulus: ")
    if _, err := fmt.Scan(&modulus); err != nil {
        fmt.Println("Invalid input")
        return
    }
    if modulus <= 1 {
        fmt.Println("modulus must be greater than 1")
        return
    }

    var target int64
    fmt.Print("target: ")
    if _, err := fmt.Scan(&target); err != nil {
        fmt.Println("Invalid input")
        return
    }
    if gcd(target, modulus) != 1 {
        fmt.Println("target is not a unit modulo modulus")
        return
    }

    var generatorCount int
    fmt.Print("number of generators: ")
    if _, err := fmt.Scan(&generatorCount); err != nil {
        fmt.Println("Invalid input")
        return
    }
    if generatorCount <= 0 {
        fmt.Println("generator count must be positive")
        return
    }

    generators := make([]int64, generatorCount)
    for i := 0; i < generatorCount; i++ {
```

```
        fmt.Printf("generator %d: ", i+1)
        if _, err := fmt.Scan(&generators[i]); err != nil {
            fmt.Println("Invalid input")
            return
        }
        if gcd(generators[i], modulus) != 1 {
            fmt.Printf("generator %d is not a unit modulo modulus\n", i+1)
            return
        }
    }

    result, found := solveDLP(target, generators, modulus)
    if !found {
        fmt.Println("solution not found")
        return
    }

    fmt.Println("solution found")
    fmt.Print("exponents: ")
    for i, exponent := range result.exponents {
        if i > 0 {
            fmt.Print(" ")
        }
        fmt.Printf("x%d=%d", i+1, exponent)
    }
    fmt.Println()
    fmt.Println("path:")
    current := int64(1)
    fmt.Printf("start: %d\n", current)
    for i, step := range result.path {
        fmt.Printf("%d. multiply by %d -> %d\n", i+1, generators[step.generatorIndex
      ], step.value)
    }
}
```

Listing 2. main.go

V. Test

A. Test Case 1

Generators = $6, 2$ Target = 3 Modulus = 7
Result:

```
DLP solver
modulus: 7
target: 3
number of generators: 2
generator 1: 6
generator 2: 2
solution found
exponents: x1=3 x2=2
path:
start: 1
1. multiply by 6 -> 6
2. multiply by 2 -> 5
3. multiply by 6 -> 2
4. multiply by 2 -> 4
5. multiply by 6 -> 3
```

Fig. 2. Test Case 1 result

B. Test Case 2

Generators = $3, 5$ Target = 2 Modulus = 13
Result:

```
DLP solver
modulus: 19
target: 2
number of generators: 2
generator 1: 18
generator 2: 4
solution found
exponents: x1=5 x2=5
path:
start: 1
1. multiply by 18 -> 18
2. multiply by 4 -> 15
3. multiply by 18 -> 4
4. multiply by 4 -> 16
5. multiply by 18 -> 3
6. multiply by 4 -> 12
7. multiply by 18 -> 7
8. multiply by 4 -> 9
9. multiply by 18 -> 10
10. multiply by 4 -> 2
```

Fig. 3. Test Case 2 result

C. Test Case 3

Generators = $30, 25, 16$ Target = 3 Modulus = 31
Result:

```
DLP solver
modulus: 31
target: 3
number of generators: 3
generator 1: 30
generator 2: 25
generator 3: 16
solution found
exponents: x1=5 x2=4 x3=1
path:
start: 1
1. multiply by 30 -> 30
2. multiply by 25 -> 6
3. multiply by 30 -> 25
4. multiply by 25 -> 5
5. multiply by 30 -> 26
6. multiply by 16 -> 13
7. multiply by 30 -> 18
8. multiply by 25 -> 16
9. multiply by 30 -> 15
10. multiply by 25 -> 3
```

Fig. 4. Test Case 3 result

D. Test Case 4

Generators = $210, 196, 107, 171$ Target = 2 Modulus = 211
Result:

```
DLP solver
modulus: 211
target: 2
number of generators: 4
generator 1: 210
generator 2: 196
generator 3: 107
generator 4: 171
solution found
exponents: x1=69 x2=46 x3=18 x4=4
path:
start: 1
1. multiply by 210 -> 210
2. multiply by 196 -> 15
3. multiply by 210 -> 196
4. multiply by 196 -> 14
5. multiply by 210 -> 197
6. multiply by 107 -> 190
7. multiply by 210 -> 21
8. multiply by 196 -> 107
9. multiply by 210 -> 104
10. multiply by 196 -> 128
11. multiply by 210 -> 83
12. multiply by 107 -> 19
13. multiply by 210 -> 192
14. multiply by 196 -> 74
15. multiply by 210 -> 137
```

Fig. 5. Test Case 4 result part 1

```
15. multiply by 210 -> 137
16. multiply by 196 -> 55
17. multiply by 210 -> 156
18. multiply by 107 -> 23
19. multiply by 210 -> 188
20. multiply by 196 -> 134
21. multiply by 210 -> 77
22. multiply by 196 -> 111
23. multiply by 210 -> 100
24. multiply by 107 -> 150
25. multiply by 210 -> 61
26. multiply by 196 -> 140
27. multiply by 210 -> 71
28. multiply by 196 -> 201
29. multiply by 210 -> 10
30. multiply by 171 -> 22
31. multiply by 210 -> 189
32. multiply by 196 -> 119
33. multiply by 210 -> 92
34. multiply by 196 -> 97
35. multiply by 210 -> 114
36. multiply by 107 -> 171
37. multiply by 210 -> 40
38. multiply by 196 -> 33
39. multiply by 210 -> 178
40. multiply by 196 -> 73
41. multiply by 210 -> 138
42. multiply by 107 -> 207
43. multiply by 210 -> 4
```

Fig. 6. Test Case 4 result part 2

```
44. multiply by 196 -> 151
45. multiply by 210 -> 60
46. multiply by 196 -> 155
47. multiply by 210 -> 56
48. multiply by 107 -> 84
49. multiply by 210 -> 127
50. multiply by 196 -> 205
51. multiply by 210 -> 6
52. multiply by 196 -> 121
53. multiply by 210 -> 90
54. multiply by 107 -> 135
55. multiply by 210 -> 76
56. multiply by 196 -> 126
57. multiply by 210 -> 85
58. multiply by 196 -> 202
59. multiply by 210 -> 9
60. multiply by 171 -> 62
61. multiply by 210 -> 149
62. multiply by 196 -> 86
63. multiply by 210 -> 125
64. multiply by 196 -> 24
65. multiply by 210 -> 187
66. multiply by 107 -> 175
67. multiply by 210 -> 36
68. multiply by 196 -> 93
69. multiply by 210 -> 118
70. multiply by 196 -> 129
71. multiply by 210 -> 82
72. multiply by 107 -> 123
73. multiply by 210 -> 88
```

Fig. 7. Test Case 4 result part 3

```
74. multiply by 196 -> 157
75. multiply by 210 -> 54
76. multiply by 196 -> 34
77. multiply by 210 -> 177
78. multiply by 107 -> 160
79. multiply by 210 -> 51
80. multiply by 196 -> 79
81. multiply by 210 -> 132
82. multiply by 196 -> 130
83. multiply by 210 -> 81
84. multiply by 107 -> 16
85. multiply by 210 -> 195
86. multiply by 196 -> 29
87. multiply by 210 -> 182
88. multiply by 196 -> 13
89. multiply by 210 -> 198
90. multiply by 171 -> 98
91. multiply by 210 -> 113
92. multiply by 196 -> 204
93. multiply by 210 -> 7
94. multiply by 196 -> 106
95. multiply by 210 -> 105
96. multiply by 107 -> 52
97. multiply by 210 -> 159
98. multiply by 196 -> 147
99. multiply by 210 -> 64
100. multiply by 196 -> 95
101. multiply by 210 -> 116
102. multiply by 107 -> 174
103. multiply by 210 -> 37
```

Fig. 8. Test Case 4 result part 4

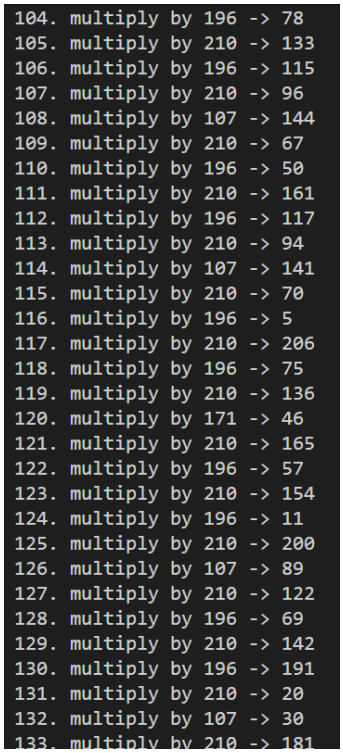

Fig. 9. Test Case 4 result part 5

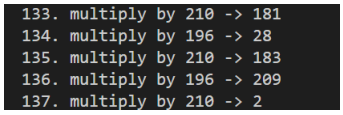

Fig. 10. Test Case 4 result part 6

VI. Acknowledgment

Praise be to God Almighty for His grace, the paper with the title "Implementation of Depth First Search to Solve the Discrete Logarithm Problem with a Multiple-Element Generating Set" can be completed properly. The author would also like to thank the lecturer of K02 IF2211 Strategi Algoritma, Ir. Rila Mandala, M.Eng., Ph.D. for the knowledge that has been taught during lectures so that the author can complete this paper smoothly. In addition, the author also wants to thank everybody who has given their support to the author in completing this research paper.

VII. Appendix

Github Repository: Github

VIII. Conclusion

In this paper, we have demonstrated that Depth-First Search can be used to solve a discrete logarithm problem with more than one generator by traversing a Cayley graph. Starting from the identity element, DFS explores the graph by repeatedly multiplying the current element with each available generator until the target element is found.

The program successfully found valid exponent vector such that the product of the generators raised to their respective exponents is congruent to the target element modulo p. Compared with a brute force approach that checks combinations of exponents, DFS explores each reachable group element at most once. Therefore, DFS can reduce repeated searches caused by different exponent combinations that produce the same value.

References

[1] D. S. Dummit and R. M. Foote, *Abstract Algebra*, 3rd ed. Hoboken, NJ, USA: John Wiley & Sons, 2004.

[2] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. Boca Raton, FL, USA: CRC Press, 1996.

[3] N. Biggs, *Algebraic Graph Theory*, 2nd ed. Cambridge, U.K.: Cambridge University Press, 1993.

[4] R. Munir, "Breadth First Search (BFS) dan Depth First Search (DFS) – Bagian 1," materi kuliah Strategi Algoritma, Institut Teknologi Bandung, 2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/13-BFS-DFS-(2026)-Bagian1.pdf. [Accessed: Jun. 18, 2026].

IX. Pernyataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, june 19th 2026

Zeki Amani

NIM 13524082