

January 2014

# Algorithms for Solving the Discrete Logarithm Problem

Ryan Edward Whaley  
*Eastern Kentucky University*

Follow this and additional works at: <https://encompass.eku.edu/etd>



Part of the [Mathematics Commons](#)

---

## Recommended Citation

Whaley, Ryan Edward, "Algorithms for Solving the Discrete Logarithm Problem" (2014). *Online Theses and Dissertations*. 235.  
<https://encompass.eku.edu/etd/235>

This Open Access Thesis is brought to you for free and open access by the Student Scholarship at Encompass. It has been accepted for inclusion in Online Theses and Dissertations by an authorized administrator of Encompass. For more information, please contact [Linda.Sizemore@eku.edu](mailto:Linda.Sizemore@eku.edu).

Algorithms for Solving the Discrete Logarithm Problem

---

---

By

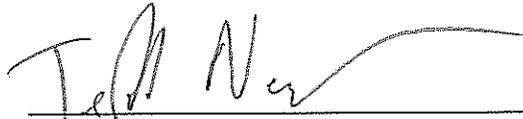
Ryan Whaley

Thesis Approved:



---

Patrick Costello, Ph.D., Chair, Advisory Committee



---

Jeffrey Neugebauer, Ph.D., Member, Advisory Committee



---

Mathew Cropper, Ph.D., Member, Advisory Committee



---

Jerry Pogatschnik, Ph.D., Dean, Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a MS degree at Eastern Kentucky University, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made. Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in [his/her] absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature Ryan Whaley

Date 4/14/2014

Algorithms for Solving the Discrete Logarithm Problem

By

Ryan Whaley

Bachelor of Science  
Eastern Kentucky University  
Richmond, Kentucky  
2012

Submitted to the Faculty of the Graduate School of  
Eastern Kentucky University  
in partial fulfillment of the requirements  
for the degree of  
MASTER OF SCIENCE  
May, 2014

Copyright © Ryan Whaley, 2014  
All rights reserved

## DEDICATION

This thesis is dedicated to my fiancée, Kristin Eppinghoff, for always supporting me through everything that I do.

## ACKNOWLEDGMENTS

I would like to give thanks to Dr. Pat Costello for working with me on this thesis, and his great patience. I would also like to thank the other two committee members, Dr. Jeffrey Neugebauer and Dr. Mathew Cropper, for their support and opinions. The last of the people I would like to thank would be all of the wonderful faculty members of the ECU mathematics and statistics department. I would not be where I am today if it weren't for them.

## ABSTRACT

In mathematics, there are often many procedures to solve or prove the same problem. The discrete logarithm is one of these problems. The baby step, giant step algorithm and Pollard's kangaroo algorithm are two algorithms that attempt to solve discrete logarithm problems. Explanations on what these two algorithms are will be discussed as well as examples of each algorithm. In addition to these two algorithms, a modified form of Pollard's kangaroo algorithm will be provided with results. Throughout the text, Mathematica programs will be presented that simulate each of the three algorithms above.

## TABLE OF CONTENTS

CHAPTER	PAGE
I. Introduction.....	1
Discrete Logarithm Problem.....	1
Applications for the DLP .....	3
II. Baby Step, Giant Step.....	5
Examples of Baby Step, Giant Step.....	6
III. Pollard's Kangaroo Algorithm.....	10
Examples of the Kangaroo Algorithm.....	11
IV. Comparing Two Algorithms.....	16
V. Modified Algorithm .....	19
Examples of the Modified Algorithm.....	20
VI. Conclusion .....	24
List of References.....	25
Appendices.....	26
A. Mathematica Commands.....	26
B. Mathematica Example Output.....	29

## Chapter I

### INTRODUCTION

In mathematics, there are often many procedures to solve or prove the same problem. The discrete logarithm problem is one of these problems. Within this paper, two algorithms will be discussed that solve the discrete logarithm problem. This paper will also discuss the programs created to simulate these two algorithms. Examples and comparisons of each algorithm will also be provided. Finally, a comparison between the two algorithms will be shown as well as a modified version of the second program. Throughout the text, you will see many examples that do not show all of the output. The complete output will be in Appendix B at the end of this paper. Also, many Mathematica commands were used within the programs. Explanations for more uncommon commands will be in Appendix A.

### DISCRETE LOGARITHM PROBLEM

Let  $p$  be a prime number and let  $\alpha, \beta \in \mathbb{Z}_p^\times$ . Let  $\beta \equiv \alpha^x \pmod{p}$ . Then the process of solving for  $x$  is called the discrete logarithm problem (DLP for short). We restrict ourselves to  $0 \leq x < n$  where  $n$  is the smallest positive exponent such that  $\alpha^x \equiv 1 \pmod{p}$ . In the case of  $p$  being prime,  $p - 1$  would be the smallest  $n$ . Otherwise, we could have multiple values for  $x \pmod{p - 1}$ .

The way to restrict  $x$  between 0 and  $n$  would be to use a proposition that says if  $r \equiv s \pmod{p-1}$ , where  $r, s \in \mathbb{Z}^+$ , then  $a^r \equiv a^s \pmod{p}$  for all  $a \in \mathbb{Z}$ . Fermat's Little Theorem is used to prove this proposition. Fermat's Little Theorem says if  $p$  is prime, then  $a^{p-1} \equiv 1 \pmod{p}$  for  $a \geq 1$  and where  $p \nmid a$  (Fermat's Little Theorem, 2014). The proof to the proposition is as follows: Suppose  $r \equiv s \pmod{p-1}$ . Then  $r = b(p-1) + s$  for some integer  $b$ . Then  $a^r = a^{b(p-1)+s} = (a^{p-1})^b \cdot a^s$ . By Fermat's Little Theorem,  $(a^{p-1})^b \equiv 1^b$ . So,  $a^r = (a^{p-1})^b \cdot a^s = 1^b \cdot a^s \equiv a^s \pmod{p}$ . Hence,  $a^r \equiv a^s \pmod{p}$ .

Now, the value of  $\alpha$  in the DLP is usually assigned to be a primitive root so that there is a solution for every  $\beta \in \mathbb{Z}_p^\times$ . An element  $\alpha \in G$  is a primitive root, or generator, of  $G$  when the powers of  $\alpha$  result in every non-zero element within the group  $G$ . So in the DLP, we have a solution for every  $\beta$  since  $\beta$  is nonzero and a power of  $\alpha$ . If we were to restrict ourselves to any  $\alpha \in \mathbb{Z}_p$ , there would be no solution to the discrete logarithm problem for certain  $\beta$  (Trappe & Washington, 2006). Within Mathematica, there is a command called *PrimitiveRoot[n]* where Mathematica will provide you with the primitive root for any integer  $n$ , provided a primitive root is defined for that particular  $n$ . In the examples of the algorithms that will be discussed later,  $n$  will be prime, which will always have a primitive root.

## APPLICATIONS FOR THE DLP

One of the most widely used applications of the discrete logarithm problem is in the field of public key cryptography. More specifically, we will look at the Diffie-Hellman key exchange. Say two people want to share a message with each other, but they want to make sure the message is kept secret. Let us say the encoding key is  $z$ . First, both people decide on a prime  $p$  and a primitive root  $\alpha$  for, in our case,  $\mathbb{Z}_p^\times$ . These two things are made public. Person A, for some  $x \in \mathbb{Z}^+$ , calculates  $\alpha^x \pmod{p} \equiv g$  and  $g \in \mathbb{Z}_p^\times$  where  $x$  is not public. Person A then sends the solution  $g$  to person B. Person B does the same thing except person B calculates  $\alpha^y \pmod{p} \equiv h$  for some  $y \in \mathbb{Z}^+$  and  $h \in \mathbb{Z}_p^\times$  where  $y$  is not public. Note that  $x$  and  $y$  are both large. Person B then sends that solution  $h$  to person A. Now, Person A simply has to calculate  $h^x \pmod{p} \equiv z$  and person B has to calculate  $g^y \pmod{p} \equiv z$ . This works since  $h^x \pmod{p} \equiv (\alpha^y)^x = (\alpha^x)^y \equiv g^y \pmod{p}$ . Hence, both people received the message. How does one on the outside find the secret message then?

For someone on the outside who does not know the values of  $x$  and  $y$ , the only way they could figure out the secret code,  $z$ , is to solve the DLP. They would have to find the  $x$  such that  $\alpha^x \equiv g \pmod{p}$ , or they would have to find the  $y$  such that  $\alpha^y \equiv h \pmod{p}$ . Only then will the outside person be able to find the secret message  $z \equiv (\alpha^x)^y \pmod{p}$ . When  $x$  and  $y$  are very large, the DLP becomes very difficult to solve (Das, 2013).

Other applications for the DLP come into play when you look at elliptic-curves and extension fields of odd characteristic  $p$ . These will not be discussed in this paper.

## Chapter II

### BABY STEP, GIANT STEP

Let  $\mathbb{Z}_p^\times$  be a finite cyclic group mod  $p$ . Suppose that we wish to solve  $\alpha^x \equiv \beta \pmod{p}$  where  $\alpha, \beta \in \mathbb{Z}_p^\times, x \in \mathbb{Z}^+$ , and  $x < p - 1$ . The baby step, giant step algorithm starts out with choosing an  $n \in \mathbb{Z}$  such that  $n^2 \geq p - 1$ . For all of the examples of this algorithm,  $n = \lfloor \sqrt{p - 1} \rfloor + 1$ . Next, you make two lists. The first list contains all  $f(j) = \alpha^j \pmod{p}$  for  $0 \leq j < n$ . Once you compute all  $f(j)$ , you then start making the second list. The second list contains all  $h(k) = \beta \alpha^{-nk} \pmod{p}$  for  $0 \leq k < n$ . You only compute values for  $h(k)$  until you have found a match when you compare the  $h(k)$  value just computed to each  $f(j)$  in the first list. Once you find a match, you stop constructing the second list. Suppose you find a match, say  $\alpha^r \equiv \beta \alpha^{-ns} \pmod{p}$ . Then  $\alpha^{r+ns} \equiv \beta$ . Thus  $r + ns = x$ , which solves the discrete logarithm problem (Trappe, 2006). If you exhaust all  $h(k)$  with no match to any  $f(j)$ , the algorithm fails.

The baby step, giant step algorithm needs a lot of storage to run. Notice that when you run this algorithm, you need to store all of the first list so that you can compare the second list to it. Now,  $0 \leq j < n$  which means that for the first list alone, you must store  $n$  values. This does not include the values you will need to store for other operations within the algorithm. At the very least, you will have to store  $n$  values. For large values

of  $p$ , this algorithm is very impractical. It is limited based on the storage the computer has and how long it takes the computer to get through both lists. The following examples in the next section will demonstrate this idea.

#### EXAMPLES OF BABY STEP, GIANT STEP

This first example is here to demonstrate how the baby step, giant step algorithm works. For this particular example, we use the group  $\mathbb{Z}_7^\times$  with primitive root  $\alpha = 3$ . We will also choose  $\beta = 5$ . Then  $n = \lfloor \sqrt{6} \rfloor + 1 = 3$ . So,  $0 \leq j, k < 3$ . We then start computing the first list.  $f(0) = 3^0 \pmod{7} \equiv 1$ ,  $f(1) = 3^1 \pmod{7} \equiv 3$ , and  $f(2) = 3^2 \pmod{7} \equiv 2$ . Once the first list is made, start the second list.  $h(0) = 5 * 3^{-7*3*0} \pmod{7} \equiv 5$ . This one doesn't match any of the  $f(j)$ , so we keep going.  $h(1) = 5 * 3^{-7*3*1} \pmod{7} \equiv 2$ . This time,  $h(1) = f(2)$ . Hence,  $x = 2 + 3 * 1 \pmod{6} \equiv 5$ . Thus, we stop constructing the second list. To check ourselves, we see that  $\alpha^x = 3^5 = 243 \equiv 5 \pmod{7}$ , which was our  $\beta$ . Hence, we found a solution.

The following is a program written in Mathematica to perform the baby step, giant step algorithm:

## Baby Step, Giant Step Program

```
Clear[g, f, a,  $\alpha$ ,  $\beta$ , j, k, b, c, x, y];
z = DateList[ ];
g = ;
 $\alpha$  = ;
 $\beta$  = ;
n = Floor[ Sqrt[ g-1 ] ] + 1;
Array[f, n - 1, 0];
Array[h, n - 1, 0];
b = PowerMod[  $\alpha$ , - 1, g];
j = 0;
k = 0;
flag = 0;
While[ j < n, f[ j ] = Mod[  $\alpha$  ^ j, g ]; Print[ "f[" , j, "]" = ", f[ j ]]; j++];
While[ (k < n) && (flag == 0), h[ k ] = Mod[  $\beta$  * b^ ( n * k ), g ]; For[ j = 0, j < n,
  j++, If[ h[ k ] == f[ j ], flag = 1; Print["h[" , k, "]" = ", h[k]]; x = j + n * k ]; k++];
y = DateList[ ];
If[ flag == 0, Print[ "Did not match" ]];
If[ flag == 1, Print[ "x=", z ]];
Print[ "Difference in seconds: ", Last[ y ] - Last[ z ] ];
```

The program statements above start out by clearing all the variable assignments

Mathematica might have stored. The two *DateList[]* commands are in the program to record the time before the program started to run and the time after the program was finished running. You start out by filling in what  $g$  you want to have where  $g$  is the order of the group  $\mathbb{Z}_p^\times$ . You then fill in the primitive root  $\alpha$  and whichever  $\beta$  you want to have where  $\beta \in \mathbb{Z}_p^\times$ . The  $n$  in the program is the same  $n$  that was discussed in the previous section. The two arrays create storage space so that while the program is running, it can store those values it computes for each  $f$  and  $h$ . We start the program at  $j, k = 0$ . The heart of the program lies in the while loop commands. The first while loop goes through

each  $j < n$  and computes what  $f(j)$  is, prints that  $f(j)$  value, and then stores that value inside the array. Note that printing the  $f(j)$  could be removed. Once the program computes all  $f$ , the second while loop starts computing  $g$ 's for the second list. For each  $k < n$ , the program then compares each  $h(k)$  to all values of  $f$  to see if there is a match. If there is a match, the program outputs the answer to the DLP. If no match occurs, the program proceeds to the next  $k$ . The program keeps doing this until all of the  $h(k)$  are exhausted, or when it has found a match. If there are no matches found, we have failed to solve the DLP. If a match was found, we have a solution. The last line of code will tell us how long it took the program to perform the baby step, giant step algorithm for the particular group,  $\alpha$ , and  $\beta$  we chose. This comes into play later.

Now that the program that simulates the baby step, giant step algorithm has been introduced, we will now go through some examples that involve much larger groups of prime order that we would not want to do by hand. This example will be from the cyclic group  $\mathbb{Z}_{1,000,003}^{\times}$ . This group has a primitive root  $\alpha = 2$ , and we would like to make  $\beta = 4,000$ . Note that I used Mathematica's *NextPrime[]* function to get the next prime above 1,000,000. Then I used Mathematica's *PrimitiveRoot[]* function to find the primitive root of that prime. So, we would like to use the baby step, giant step algorithm to solve the DLP  $2^x \equiv 4,000 \pmod{1,000,003}$ . When I put these values into the Mathematica program above, it took the baby step, giant step algorithm 43.42 seconds to compute the value of  $x$ . The program was run on an HP ENVY m6 Notebook with an Intel Core i5 processor and 8 gigs of RAM. For all other examples, this same computer will be used to run the programs. The value of  $x$  that solves the DLP is 877,142.

Another example involves a larger group to show how much longer it takes the baby step, giant step program to compute the value of  $x$ . The cyclic group this time is  $\mathbb{Z}_{10,000,019}^{\times}$ . The primitive root for this group is 6, so  $\alpha = 6$ . Also, we will assign  $\beta = 256$ . This example will require solving the DLP  $6^x \equiv 256 \pmod{10,000,019}$ . This time around, it took Mathematica 2,165.36 seconds to compute the value of  $x$ . The value of  $x$  that solves this DLP is 8,954,372. As you can see, just adding an extra digit has increased the time to compute a huge amount. This shows the idea that the baby step, giant step algorithm very well depends on your computer's memory and running capability.

**POLLARD'S KANGAROO ALGORITHM**

For Pollard's kangaroo algorithm, we are going to let  $\mathbb{Z}_p^\times$  be a finite cyclic group mod  $p$  just like in the baby step, giant step algorithm. Also, suppose we wish to solve  $\alpha^x \equiv \beta \pmod{p}$  where  $\alpha, \beta \in \mathbb{Z}_p^\times, x \in \mathbb{Z}^+, \text{ and } x < p - 1$ . For the sake of comparison later, we will be choosing the same  $n$  as we did for the baby step, giant step algorithm. So,  $n = \lfloor \sqrt{p-1} \rfloor + 1$ . Within this paper, we will be using an interval  $\{b, \dots, c\} \subset \mathbb{Z}_p^\times$  such that  $b = 0$  and  $c = \lfloor \frac{p}{2} \rfloor$ , where  $\alpha^c \pmod{p}$  is where the first list we will calculate will come from (Galbraith, 2013). Note that we could use the interval  $\{0, \dots, p - 1\}$  but the point of this algorithm is to try to solve the DLP more efficiently. Increasing this interval will increase the time it takes for this algorithm to solve the DLP. The downside to decreasing this interval is that a smaller interval invites more failures to occur when you use the kangaroo algorithm. The next thing to set up is a pseudorandom walk. This random walk is what drives the kangaroo algorithm. For each  $r \in \mathbb{Z}_p^\times$ , you pair a random integer so that you get a map  $f: \mathbb{Z}_p^\times \rightarrow S$  where  $S$  is the set of random integers created.

Now, there are two lists that we start computing. One is called the "tame kangaroo" and the other is called the "wild kangaroo." The "tame kangaroo" will be all  $x_i$  such that  $i \in \{0, 1, \dots, n - 1\}$ . The value of  $x_0 = \alpha^c$  and all other  $x_{i+1} = x_i \alpha^{f(x_i)}$ . Note that the first value of the first list starts with the primitive root raised to the middle

element of the group we are in. The next thing to compute is  $d = \sum_{i=0}^{n-1} f(x_i)$ . This adds up all of the random integers that the original group elements were mapped to with the pseudorandom mapping. Now that the first list was made, you start constructing a second list, or the “wild kangaroo.” This time, the first step is  $y_0 = \beta$ . For every step afterward,  $y_{i+1} = y_i \alpha^{f(y_i)}$ . Just like with the  $x_i$ 's, we then form a  $d_n = \sum_{i=0}^{n-1} f(y_i)$  which sums up the random integers used when computing  $f(y_i)$  for each  $i$ .

We keep creating  $y_i$ 's until one of two conditions are met. The first condition is that the algorithm finds  $y_j = x_n$  for some  $j$ . If this happens, we have found a solution to the DLP and  $x = c + d - d_j$ . The second condition to check for would be if  $d_i > c - b + d$ . If this happens, the algorithm fails to find the  $x$  (“Pollard’s kangaroo algorithm”, 2014).

When this happens, you must rerun the algorithm again with a different pseudorandom walk to see if the algorithm is successful. Note that unlike the baby step, giant step algorithm, you only need to store the last step in the first list,  $x_n$ . You then compare every  $y_i$  to that one value. This algorithm is much more efficient in the aspect that very little storage space is required. Therefore, you don’t have as much of a time or resource constraint that you did with the previous algorithm with the kangaroo algorithm.

#### EXAMPLES OF THE KANGAROO ALGORITHM

An example of the DLP will now be demonstrated that uses the kangaroo algorithm. This first example will walk you through a simple example that can easily be

done by hand. Suppose that we are working in the cyclic group  $\mathbb{Z}_{13}^*$ . The primitive root for this group is  $\alpha = 2$  and we choose our  $\beta = 9$ . Then  $n = \lfloor \sqrt{12} \rfloor + 1 = 4$ . The interval that is created is  $[0,6]$ . The pseudorandom mapping that was created is as follows:

$f(1) = 3, f(2) = 8, f(3) = 9, f(4) = 2, f(5) = 11, f(6) = 4, f(7) = 7, f(8) = 12, f(9) = 7, f(10) = 8, f(11) = 1, f(12) = 11$ . The next thing to do is to create all of the  $x_i$ 's for  $i = 1, \dots, n$ .  $x_0 = \alpha^b \pmod{13} = 2^6 \pmod{13} \equiv 12$ . Then  $x_1 = x_0 * \alpha^{f(x_0)} \pmod{13} = 12 * 2^{11} \pmod{13} \equiv 6$ . Continuing the same recursive formula for the other three  $x_i$ 's, we get  $x_2 = 5, x_3 = 9$ , and  $x_4 = 8$ . We compute the  $d$  value as  $d = f(x_0) + f(x_1) + f(x_2) + f(x_3) = 11 + 4 + 11 + 7 = 33$ . Now that these are computed, we will now compute the  $y_i$ 's. Remember, we stop the calculations when either we find  $y_i = x_n$  or if  $d_i > b - a + d = 6 - 0 + 33 = 39$ . The beginning value is  $y_0 = \beta = 9$ . Then  $y_1 = y_0 * \alpha^{f(y_0)} \pmod{13} = 9 * 2^7 \pmod{13} \equiv 8$  and  $d_1 = f(y_0) = 7$ . We found a match between  $y_1$  and  $x_4$ . Thus  $x = b + d - d_1 = 6 + 33 - 7 = 32 \pmod{12} \equiv 8$ . Checking ourselves, we see that  $\alpha^x = 2^8 = 256 \equiv 9 \pmod{13}$ .

As you can see from the example above, there are quite a few calculations that are involved with this algorithm, and this example was with a small group. Imagine the amount of calculations one would have to do in order to solve a DLP for a larger group. That is why I created a program through Mathematica that will simulate Pollard's kangaroo algorithm without having to do the calculations by hand. The program is as follows:

## Pollard's Kangaroo Algorithm

```
Clear[  $\alpha$ ,  $\beta$ ,  $g$ ,  $n$ ,  $b$ ,  $c$ ,  $s$ ,  $y$ ,  $dt$ ,  $h$ ,  $f$ ,  $a$ ,  $d$ ];
 $\alpha$  = ;
 $\beta$  = ;
 $g$  = ;
 $n$  = Floor[ Sqrt[  $g - 1$  ] ] + 1;
 $b$  = 0;
 $c$  = Floor[  $g/2$  ];
f[t_] := RandomInteger[ {1, $g$  } ];
Do[ a[ t ] = f[ t ]; Print[ "a[" , t, "]" = " , a[ t ] ], { t,  $g$  } ];
h[ 0 ] = Mod[  $\alpha^c$ ,  $g$  ];
Print[ "x_i's" ];
d = 0;
Do[ h[ j ] = Mod[ h[ j - 1 ] *  $\alpha$  ^ ( a[ h[ j - 1 ] ] ),  $g$  ];
  Print[ "h[" , j, "]" = " , h[ j ] ]; d = d + a[h[ j - 1 ] ], {j, n} ];
Print[ "d = " , d ];
Clear[ r, s, y ];
y[ 0 ] =  $\beta$ ;
flag = 0;
s = 1;
Print[ "y_i's and d_i's" ];
While[ ( flag == 0 ) && ( y[ s - 1 ]  $\neq$  h[ n ] ),
  s ++ ;
  y[ s - 1 ] = Mod[ y[ s - 2 ] *  $\alpha$  ^ ( a[ y[ s - 2 ] ] ),  $g$  ];
  Print[ "y[" , s - 1, "]" = " , y[ s - 1 ] ];
  dt[ s - 1 ] =  $\sum_{i=0}^{s-2} a[y[i]]$  ;
  If[ dt[ s - 1 ]  $\leq$  ( c - d + b ), flag = 0, flag = 1 ];
  Print[ "dt[" , s - 1, "]" = " , dt[ s - 1 ] ] ];
  Print[ "value of x" ];
  If[ y[ s - 1 ] == h[ n ], Print[ c + d - dt[ s - 1 ] ], Print[ "failed" ] ]];
```

The program starts out by clearing all of the variables so that nothing is stored for those variables each time you run the program. You then specify the values of  $\alpha$ ,  $\beta$ , and  $g$  where these variables are the same as the variables in the baby step, giant step program. The value of  $n$  is then created along with  $b$  and  $c$ . The function  $f$  is used to call upon random integers between 1 and  $g$ . The do loop comes after. This assigns the

function  $a$  to actually be the pseudorandom walk. This creates  $a(1), \dots, a(g)$  by using values of the function  $f$ . Once those are created, we start constructing the lists. The function  $h$  represents the  $x_i$ 's that were explained in the kangaroo algorithm in the previous section. So  $h[0] = x_0$ . The program then tells you to print the words "x\_i's". The do loop that follows creates the rest of the  $x_i$ 's along with printing each one in the output Mathematica provides. The variable  $d$  is created next which was defined in the last section. The print command tells Mathematica to list what  $d$  is equal to. We then clear the variables  $r, s$ , and  $y$  of any stored values they might have had.  $y[0]$  is the same as  $y_0$  in the previous section. The while loop that follows  $y[0]$  is the loop that creates  $y_i$ 's until one of the two conditions are met. Once the while loop ends, Mathematica then prints the words "value of x." Finally, the if statement that follows tells Mathematica that if it found a  $y_i$  that matches  $x_n$ , print  $c + d - d_i$ , which is the value of  $x$ . Otherwise, Mathematica will print "failed", which means that the algorithm failed to solve the DLP. The next thing to do is to do more examples of solving the DLP, since we have a program that simulates the kangaroo algorithm now.

This first example will be from the cyclic group  $\mathbb{Z}_{1,000,507}^{\times}$ . The primitive root for this group is  $\alpha = 2$ . We wish to solve the DLP  $2^x \equiv 3,000 \pmod{1,000,507}$ . When first running the program, the algorithm failed twice due to the fact the second condition of the algorithm did not get met. The third time running the program resulted in a success. The program found that  $x = 72,339,521$ . When reducing this  $x \pmod{1,000,506}$ , we result in  $x = 303,089$ . Upon checking,  $2^{303,089} \equiv 3,000 \pmod{1,000,507}$ . The run time of this program for this particular example was only 7.34 seconds. Therefore, even

though there were two failures when running the program, the combined time for all failures and the success was only roughly 22 seconds. That is not bad at all.

The next example involves the cyclic group  $\mathbb{Z}_{1,500,007}^{\times}$ . The primitive root for this group is  $\alpha = 3$ . We wish to solve the DLP  $3^x \equiv 1,250 \pmod{1,500,007}$ . This time, when running the program for the first time, we got a success. The program took only 20.64 seconds to run. The value that we got for  $x$  was  $x = 137,811,996$ . Reducing  $x \pmod{1,500,007}$ ,  $x = 1,311,450$ . Upon checking, we find  $\alpha^x = 3^{1,311,450} \equiv 1,250 \pmod{1,500,007}$ .

### COMPARING TWO ALGORITHMS

In this section, we will now compare the baby step, giant step algorithm to the kangaroo algorithm. The first example that we will be comparing will be one that we looked at in chapter 2. The group is  $\mathbb{Z}_{1,000,003}^{\times}$  with primitive root  $\alpha = 2$ . We wish to solve the DLP  $2^x \equiv 4,000 \pmod{1,000,003}$ . When we performed the baby step, giant step algorithm in chapter 2 for this example, we found that  $x = 877,142$ . This process took 43.42 seconds to complete. Now, when I ran the same example through the kangaroo algorithm program, the time was consistently around 8 to 10.5 seconds to complete the entire algorithm. These times are a lot better than the times for the baby step, giant step algorithm. This is due to the fact that the baby step, giant step algorithm is storage intensive to where the kangaroo algorithm is not. However, I ran this example through the kangaroo algorithm ten times and got a failure all ten times. In this case, the baby step, giant step algorithm was able to solve this example to where the kangaroo algorithm failed multiple times.

Another example we wish to look at is from the cyclic group  $\mathbb{Z}_{10,000,019}^{\times}$ . Note that this was the third example we did in chapter 2. When we ran this example in the program for the baby step, giant step algorithm, the time it took to finish was 2,165.36 seconds. The value of  $x = 8,954,372$ , which solve the DLP. If you look at the time though, 2,165.36 seconds is a long time to just solve one DLP. Now we will compare the

kangaroo algorithm to the baby step, giant step algorithm using the same example. This time, when we ran the example through the program for the kangaroo algorithm, the time it took was only 355.58 seconds. That is way less than 2,156.36 seconds.

Furthermore, it took only one time through the program to solve this example. The program ended up getting a value of

$x = 688,955,596 \equiv 8,954,372 \pmod{10,000,018}$ . This time, the kangaroo algorithm was a lot more efficient than the baby step giant step.

You may be asking why you can't use the baby step, giant step algorithm all the time since using an  $n = \lfloor \sqrt{p-1} \rfloor + 1$  will give you a success when using this algorithm.

One reason why you should consider using the kangaroo algorithm is because the kangaroo algorithm isn't storage intensive. You can solve DLP's with much higher prime numbers with the kangaroo algorithm than you ever could with the baby step, giant step algorithm. Another reason is based on time. Even though the kangaroo algorithm can fail a lot more often due to the random walk with its algorithm, the time to solve DLP's is a lot shorter than the baby step, giant step algorithm's time to solve.

Another reason to consider using the kangaroo algorithm is because you do not need to limit yourself to the size of  $n$  we have been dealing with. What if we wanted to decrease the size of  $n$  so that the amount of storage and time we need decrease? For example, let  $n = \lfloor \sqrt{p}/2 \rfloor$ . This will definitely decrease the amount of storage we need for the baby step, giant step algorithm. Now, let the group we use be  $\mathbb{Z}_{1,000,033}^{\times}$  with primitive root  $\alpha = 5$ . We wish to solve the DLP  $5^x \equiv 1,500 \pmod{1,000,033}$  Now, the

modification that we have to make to the baby step, giant step program is that we change what  $n$  is equal to. Once we change that, we will then put this example in like usual. The time that it took to run the baby step, giant step algorithm was only 20.59 seconds, but no match was found between the two lists. This means that  $x$  cannot be found using the baby step, giant step algorithm with this size  $n$ . This algorithm will always fail with this example. When we take this example and run it in the program for the kangaroo algorithm, there were 12 failures at first with times ranging from 9.5 to 10.8 seconds. Upon running the program for the 13<sup>th</sup> time, a value of  $x$  was found, which was  $58,769,379 \equiv 767,523 \pmod{1,000,032}$ . Checking this, we get  $\alpha^x = 5^{767,523} \equiv 1,500 \pmod{1,000,033}$ . The 13<sup>th</sup> time took 9.8 seconds to complete. Even though there were a lot of failures, eventually the kangaroo algorithm solved the DLP to where the baby step, giant step did not.

## Chapter V

### MODIFIED ALGORITHM

There have been some modifications to Pollard's Kangaroo Algorithm over the years. The most famous ones involve increasing the number of kangaroos that the algorithm has, whether it be increasing the number of tame kangaroos or increasing the number of wild kangaroos the algorithm might have (Galbraith, 2013). Instead of increasing the number of kangaroos within the algorithm, we decided to keep two kangaroos, one tame and one wild, and keep the last two steps of the tame kangaroo instead of just the last step. In other words, the original algorithm states that you are to keep the last step of the tame kangaroo's jump. You then compare the  $y_i$ 's to that last step to see if you get a match. Now, we will keep not only the last step but the second to last step as well. This means that we will be comparing the  $y_i$ 's to both of these steps instead of just the last step. So, instead of explaining the whole process again, the only additional step in this modified algorithm is to store  $x_{n-1}$  as well as  $x_n$ . Then, every time you find a  $y_i$ , compare it to both of the  $h$  values. If you get a match to either one, you are finished. If not, repeat the process for the next  $y_i$ . The algorithm still ends when one of the two conditions of the original kangaroo algorithm is met.

## EXAMPLES OF THE MODIFIED ALGORITHM

For the first example, we are going to use the same example that was in chapter 3 to start out with. The cyclic group is  $\mathbb{Z}_{13}^\times$ . Recall that the primitive root was  $\alpha = 2$  and we chose our  $\beta = 9$ . The value  $n = \lfloor \sqrt{12} \rfloor + 1 = 4$ . The interval that we had was  $[0,6]$ . The pseudorandom mapping that was created was:  $f(1) = 3, f(2) = 8, f(3) = 9, f(4) = 2, f(5) = 11, f(6) = 4, f(7) = 7, f(8) = 12, f(9) = 7, f(10) = 8, f(11) = 1, f(12) = 11$ . The next thing we did was create all the  $x_i$ 's. We found that  $x_0 = 12, x_1 = 6, x_2 = 5, x_3 = 9, x_4 = 8$ . The next thing that is different is the value of  $d$ . This time, we have a  $D_1$  value which is the  $d$  value for the  $x_{n-1}$  step and  $D_2$  is the  $d$  value for the  $x_n$  step. So,  $D_1 = 26$  and  $D_2 = 33$ . Once the first list is computed, we keep  $x_3 = 9$  and  $x_4 = 8$  to compare the  $y_i$ 's to. Remember that this time we stop the calculations of the  $y_i$ 's if either  $y_i = x_3$  or  $y_i = x_4$  or if  $d_i > c - b + d = 39$ . The starting value is  $y_0 = \beta = 9$ . We can automatically stop there since  $y_0 = x_3$ . So  $x = c + D_1 - d_0 = 6 + 26 - 0 = 32 \equiv 8 \pmod{12}$ . If there was a match between a  $y_i$  and the last step  $x_n$ , we would have used  $D_2$  instead of  $D_1$  when finding  $x$ .

The Mathematica program that was created for this modified algorithm is almost exactly the same as the program for the kangaroo algorithm. However, there are a few minor changes to certain parts of the original program. The program for the modified algorithm is below.

## Modified Kangaroo Algorithm

```
Clear[  $\alpha$ ,  $\beta$ , g, n, b, c, s, y, dt, h, f, a, d, d1, d2];
 $\alpha$  = ;
 $\beta$  = ;
g = ;
n = Floor[ Sqrt[ g - 1 ] ] + 1;
b = 0;
c = Floor[ g/2 ];
f[t_] := RandomInteger[ {1,g} ];
Do[ a[ t ] = f[ t ]; Print[ "a[" , t, "]" = ", a[ t ] ], { t, g } ];
h[ 0 ] = Mod[  $\alpha^c$ , g ];
Print[ "x_i's" ];
d = 0;
Do[ h[ j ] = Mod[ h[ j - 1 ] *  $\alpha^{( a[ h[ j - 1 ] ] )}$ , g ];
  Print[ "h[" , j, "]" = ", h[ j ] ], {j, n} ];
d1 =  $\sum_{k=0}^{k-2} a[h[k]]$ ;
d2 =  $\sum_{k=0}^{k-1} a[h[k]]$ ;
Print[ "d1 = " , d1 ];
Print[ "d2 = " , d2 ];
Clear[ r, s, y ];
y[ 0 ] =  $\beta$ ;
flag = 0;
s = 1;
Print[ "y_i's and d_i's" ];
While[ ( flag == 0 ) && ( y[ s - 1 ]  $\neq$  h[ n ] ) && ( y[ s - 1 ]  $\neq$  h[ n - 1 ] ),
  s ++ ;
  y[ s - 1 ] = Mod[ y[ s - 2 ] *  $\alpha^{( a[ y[ s - 2 ] ] )}$ , g ];
  Print[ "y[" , s - 1, "]" = ", y[ s - 1 ] ];
  dt[ s - 1 ] =  $\sum_{i=0}^{s-2} a[y[i]]$ ;
  If[ dt[ s - 1 ]  $\leq$  ( c - d + b ), flag = 0, flag = 1 ];
  Print[ "dt[" , s - 1, "]" = ", dt[ s - 1 ] ] ];
  Print[ "value of x" ];
  If[ y[ s - 1 ] == h[ n ], Print[ c + d2 - dt[ s - 1 ] ] ];
  If[ y[ s - 1 ] == h[ n - 1 ], Print[ c + d1 - dt[ s - 1 ] ], Print[ "failed" ] ];
```

This program has the same explanation as the one we gave in chapter 3 until you get down to the values of  $D1$  and  $D2$ . The reason we must compute two different  $d$  values

instead of just one is because we are keeping two steps of the tame kangaroo instead of one now. Once you get past that part of the program, it continues the same way until you hit the while loop. Since we now have to check to make sure the  $y_i$ 's do not match  $x_n$  also, we have to put another  $\&\&$  check statement inside the while loop. Finally, the program continues on as usual until you get to the last two statements. The first statement tells you that if the  $y_i = x_n$ , print what  $x$  equals. The second one tells you if the  $y_i = x_{n-1}$ , print what  $x$  equals. Otherwise, the program prints failed.

Now that the program has been explained, we will run through an example coming from  $\mathbb{Z}_{1,000,003}^\times$  with primitive root  $\alpha = 2$ . We wish to solve the DLP  $2^x \equiv 269 \pmod{1,000,003}$ . The first five times of running the modified program resulted in a failure. The 6<sup>th</sup> time running it resulted in a success. It took 9.67 seconds for the program to get an  $x = 279,617,009 \equiv 616,451 \pmod{1,000,003}$ . Checking this, we get  $\alpha^x = 2^{616,451} \equiv 269 \pmod{1,000,003}$ .

Now we will revisit the example from chapter 4 where the kangaroo algorithm failed ten times before we quit trying to find a solution. The group was  $\mathbb{Z}_{1,000,013}^\times$  with primitive root  $\alpha = 2$ . We wish to solve the DLP  $2^x \equiv 4,000 \pmod{1,000,013}$  by trying to run this example through the modified algorithm. Again, like the kangaroo algorithm, the modified algorithm failed to solve this DLP. Note that even though both programs failed to solve this DLP, theoretically the modified version of the kangaroo algorithm should solve more problems than the original kangaroo algorithm. This is due to the fact that not only does the modified algorithm compare each  $y_i$  to the  $x_n$ , just like the original kangaroo algorithm, it also compares each  $y_i$  to the  $x_{n-1}$ . This means that the

modified algorithm has twice as many chances to find a solution than the original algorithm does.

## Chapter VI

### **CONCLUSION**

The three algorithms presented within this paper are only some out of many ways to solve the discrete algorithm problem. Some are more efficient than the ones discussed, and some are not. It is up to the type of DLP you wish to solve to determine which method is the most effective and reliable to use when dealing with these problems. Also, you can certainly take an already existing algorithm and modify it to see how it affects the effectiveness of the overall process, like we did with the modified version of the kangaroo algorithm. The ideas are limitless.

## REFERENCES

1. Das , A. (2013). *Computational number theory*. (1st ed., p. 347). Boca Raton, FL: Chapman and Hall/CRC.
2. *Fermat's Little Theorem*. (2014, 02 04). Retrieved from [http://en.wikipedia.org/wiki/Fermat's\\_little\\_theorem](http://en.wikipedia.org/wiki/Fermat's_little_theorem)
3. Galbraith, S. D., Pollard, J. M., & Ruprai, R. S. (2013). COMPUTING DISCRETE LOGARITHMS IN AN INTERVAL. *Mathematics Of Computation*, 82(282), 1182-1183.
4. *Pollard's kangaroo algorithm*. (2014, 02 20). Retrieved from [http://en.wikipedia.org/wiki/Pollard's\\_kangaroo\\_algorithm](http://en.wikipedia.org/wiki/Pollard's_kangaroo_algorithm)
5. Trappe, W., & Washington, L. C. (2006). *Introduction to cryptography with coding theory*. (2nd ed., pp. 201-202). Upper Saddle River, NJ: Pearson Education, Inc.

APPENDIX A:  
Mathematica Commands

This appendix is here to show all of the Mathematica commands used throughout this paper. In addition this appendix give formats you would have to use in order to use these commands in Mathematica.

### **Array[ f, n, r ]**

- This array command generates a list of length n using the starting point of r with elements f[i].

### **Clear[ symbol\_1, symbol\_2, ... ]**

- Clears all values and definitions for each symbol specified within the Clear[] command.

### **DateList[]**

- This gives the current local time and date in the form of { year, month, day, hour, minute, second}.

### **Do[ expression, { i, i<sub>min</sub>, i<sub>max</sub> } ]**

- This tells you to evaluate the expression with variable i from i<sub>min</sub> to i<sub>max</sub>.

### **Floor[n]**

- Performs the floor function on n.

### **If[ condition, t, f ]**

- This gives the output of t if the condition is true and an output of f if the condition is false.

### **Mod[ m, n ]**

- This gives the remainder on division of m by n.

**NextPrime[n]**

- This gives the next prime number after the number n specified,

**PowerMod[ a, - 1 , m ]**

- This finds the modular inverse of a mod m.

**PrimitiveRoot[n]**

- This gives the smallest primitive root of the specified n.

**Print[]**

- Prints whatever you write within the brackets in the output.

**RandomInteger[ { i<sub>min</sub>, i<sub>max</sub> } ]**

- This gives a pseudorandom integer in the range specified.

**Sqrt[n]**

- This gives the square root of n.

**While[ test, body ]**

- This loop evaluates whatever is in the test, then the body, repetitively, until the test first fails to give a true statement.

APPENDIX B:  
Mathematica Example Output

Appendix B is here to show some of the output that Mathematica printed when running the examples throughout the paper. An example from each program will be listed below, along with what Mathematica prints when you run the example with the program.

### **Example for the Baby Step, Giant Step Program**

- This example is for the group  $\mathbb{Z}_{1,000,013}^{\times}$  where we wished to solve the DLP  $2^x \equiv 4,000$ . When put into the baby step, giant step program, Mathematica prints the following:

```
f[0]=1
f[1]=2
f[2]=4
f[3]=8
.
.
.
f[997]=813833
f[998]=627663
f[999]=255323
f[1000]=510646
h[876]=408096
x=877142
Difference in seconds: 44.0754184
```

### Example for the Kangaroo Algorithm Program

- This example is for the group  $\mathbb{Z}_{1,000,507}^{\times}$  where we wished to solve the DLP  
 $2^x \equiv 3,000 \pmod{1,000,507}$ . When put into the kangaroo algorithm program,

Mathematica prints the following:

```
x_i's
h[1]=793966
h[2]=662920
h[3]=27923
.
.
.
h[999]=240407
h[1000]=21551
h[1001]=837242
d= 502131756
y_i's and d_i's
y[1]= 928917
dt[1]= 42256
y[2]= 653373
dt[2]= 448240
.
.
.
y[880]= 21551
dt[880]= 429608346
y[881]= 837242
dt[881]= 430292488
value of x
72339521
```

### Example of the Modified Program

- This example is for the group  $\mathbb{Z}_{1,000,003}^{\times}$  where we wished to solve the DLP

$2^x \equiv 4,000 \pmod{1,000,003}$ . When put into the modified program,

Mathematica prints the following:

```
x_i's
h[1]=765027
h[2]=163213
h[3]=399728
.
.
.
h[999]=903447
h[1000]=646019
h[1001]=373658
d1= 496588904
d2= 496742024
y_i's and d_i's
y[1]= 68893
dt[1]= 886604
y[2]= 181220
dt[2]= 1340191
.
.
.
y[540]= 163051
dt[540]= 260521290
y[541]= 57004
dt[541]= 261505971
value of x
failed
```