Mapping the Discrete Logarithm

Daniel R. Cloutier
Rose-Hulman Institute of Techonology
Terre Haute, IN 47803
Daniel.R.Cloutier@Rose-Hulman.edu

July 3, 2005

Abstract

The discrete logarithm is a problem that surfaces frequently in the field of cryptography as a result of using the transformation $g^a \mod n$. This paper focuses on a prime modulus, p, for which it is shown that the basic structure of the functional graph is largely dependent on an interaction between g and $p-1$. In fact, there are precisely as many different functional graph structures as there are divisors of $p-1$. This paper extracts two of these structures, permutations and binary functional graphs. Estimates exist for the shape of a random permutation, but similar estimates must be created for the binary functional graphs. Experimental data suggests that both the permutations and binary functional graphs correspond well to the theoretical data which provides motivation to extend this to larger divisors of $p-1$ and study the impact this forced structure has on the many cryptographic algorithms that rely on the discrete logarithm for their security. This is especially applicable to those algorithms that require a "safe" prime ($p = 2q+1$, where q is prime) modulus since all non-trivial functional graphs generated using a safe prime modulus can be analyzed by the framework presented here.

1 Introduction

Just a few decades ago, cryptography was considered a domain exclusive to national governments and militaries. However, the computer explosion has changed that. Every day, millions of people trust that their privacy will be protected as they make online purchases or communicate privately with a friend. Many of the cryptographic algorithms they will use are built upon a common transformation, namely

$$g^x \equiv y \mod n. \tag{1}$$

For instance, Diffie-Hellman key exchange, RSA and the Blum-Micali pseudorandom bit generator all use (1). This paper will examine some of the properties exhibited by this sort of transformation and provide theoretical and experimental data describing how the interaction between g and the modulus impacts the behavior of this function. First, though, we give a brief look at each of the algorithms listed above to provide an illustration of the wide range of use (1) has found in the field of cryptography and the extent to which many of these algorithms base their security on the properties of (1).

In 1976, Whitfield Diffie and Martin Hellman proposed a way to use (1) as part of a method for key-exchange. They presented a scheme in [3] where two users compute a shared secret key over a public communication channel using (1). Using a publicly specified p and

g, which is a primitive root (also referred to as a generator) modulo p, Alice computes $g^a \mod p$ and Bob computes $g^b \mod p$. They can exchange results and each can compute the same shared key, namely $g^{ab} \mod p$. The most obvious attack to this system involves solving directly for a (or equivalently, b). Fortunately, this problem appears to be quite difficult and is generally known as the discrete logarithm problem. Schneier [11, Section 11.6] gives an introduction to the problem for the unfamiliar reader.

Two years after Diffie and Hellman announced their key-exchange system, Ronald Rivest, Adi Shamir and Leonard Adleman used (1) as the backbone of a new public-key encryption method dubbed RSA [10]. RSA is a simple and elegant algorithm, yet it appears to provide an extremely high level of security. If Alice intends to send Bob a message, then she looks up Bob's public key which consists of two numbers, e and n. She then computes $C = M^e \mod n$ where M is the message she wishes to encrypt. Bob then takes Alice's encrypted message C and combined with his secret key d, computes $M = C^d \mod n$. The most obvious attack on this system is to compute d, Bob's secret key. The authors argue that this is equivalent to the problem of factoring n since the prime factorization of n can be found using e and d. In general, however, factoring a large number has proven to be a hard problem. Pomerance [9], describes the evolution of our attempts to find an efficient solution to this problem.

In 1984, Blum and Micali introduced the first pseudorandom bit generator (PRBG) designed to be secure enough for cryptographic applications [2]. Schneier [11, Section 17.9] gives the following explanation of the algorithm. Let g be a prime and p be an odd prime. A seed, x_0 is used to start the process with each following bit computed using $x_{i+1} = g^{x_i} \mod p$. The output is 1 if $x_i < (p-1)/2$ and 0 otherwise. Assuming that g and p are known, predicting the next bit with greater than 50% certainty appears to require solving the discrete logarithm problem. Blum and Micali's bit generator, however, was rather slow, taking one entire modular exponentiation per bit. It has since been improved in a number of ways to increase its efficiency while maintaining its cryptographic security. For instance, the PRBG presented by Gennaro in [6] can produce $n - c - 1$ bits per exponentiation with an n bit modulus and a c bit exponent.

2 Terminology and Background

A mapping relates elements in one set, the domain, to elements in another set, the codomain. It does this by means of what will be referred to here as a transition function. For instance, let A be the domain and B the codomain. If φ is the transition function, then for each $a \in A$, $\varphi(a) = b$ for some $b \in B$. In this paper, we will examine mappings generated with (1) as the transition function and restrict the values of n to primes. For a more natural notation, p will hereafter denote the prime modulus.

In some instances, it will prove to be useful to interpret the mappings as functional graphs. A graph is a set of vertices (or nodes) and a set of edges where each edge is a directed path from one vertex to another (or possibly the same) vertex. A functional graph simply restricts the edges such that each vertex must have exactly one edge directed out from it. Equivalently, the out-degree of each vertex must be one. The relationship between the mappings which interest us and functional graphs is straightforward. In the mappings of interest here, the domain and codomain are the same set, namely

$$S = \{1, 2, ..., p-1\}.$$

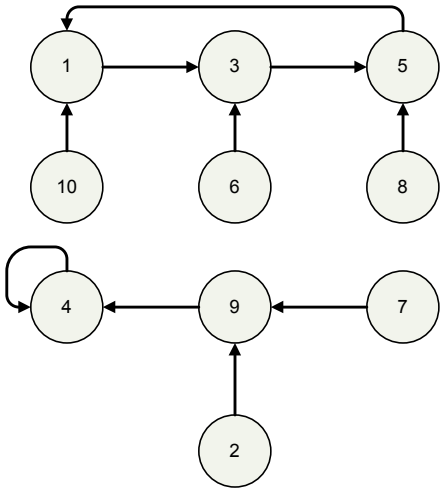

Figure 1: The graph generated using $\varphi(x) = 3^x \mod 11$. This graph has two connected components: one containing a cycle of length three (1,3,5) and the other containing a cycle of length one (fixed point) at 4.

Each element in S can then be interpreted as a vertex. The edges are defined simply for $a, b \in S$ where an edge $\langle a, b\rangle$ is in the graph if and only if $\varphi(a) = b$.

There are a number of statistics of interest derived from functional graphs. Following the convention of [5], which treats random mappings in detail, let $\varphi : S \to S$ be the transition function so that the edges in the functional graph can be expressed as the ordered pair $\langle x, \varphi(x)\rangle$ for $x, \varphi(x) \in S$. By applying the pigeonhole principle and noting that the cardinality of S is $p - 1$ we can say that by starting at any random point u_0 and following the sequence $u_1 = \varphi(u_0)$, $u_2 = \varphi(u_1)$, ..., there must be a $u_i = u_j$ after at most p iterations. Suppose u_i occurs before u_j in the sequence of nodes. In this case, the tail length is the number of iterations from u_0 to u_i. The cycle length is the number of iterations from u_i to u_j. In more natural graphical terms, the cycle length is the number of edges (or equivalently nodes) involved in the directed path from u_i to itself. The tail length is the number of edges from u_0 to u_i. Additionally, a terminal node is one with no pre-image, or more formally, x is a terminal node if $\varphi^{-1}(x) = \emptyset$. A node is an image node if it is not a terminal node. Since each node has an out-degree of exactly one, each cycle with the trees grafted onto its nodes will form a connected component. An example of a small functional graph can be found in Figure 1.

The value of g plays a major role in determining the basic structure of the graph. In fact, as Theorem 1 formalizes, the interaction among g and $p - 1$ will effectively fix the in-degrees of the nodes in the graph. First, though, define an m-ary functional graph to be a graph where each node has in-degree of exactly zero or m.

Theorem 1. *Let m be any positive integer that divides $p-1$. Then there are $\phi(\frac{p-1}{m})$ m-ary functional graphs produced by the map φ for a given g and p. Furthermore, if r is any primitive root modulo p, and $g \equiv r^a \mod p$, then the values of g that produce an m-ary graph are precisely those for which $\gcd(a, p-1) = m$.*

Proof. Let r be any primitive root modulo p. Choose a and b so that $g \equiv r^a \mod p$ and

$y \equiv r^b \mod p$ for g and y in (1). Then (1) can be written in terms of r as

$$r^{ax} \equiv r^b \mod p.$$

Since r is a primitive root, the solutions are the same as in

$$ax \equiv b \mod p-1.$$

Let $m = \gcd(a, p-1)$. By [8, Theorem 2.17], there are m solutions if $m|b$ and no solutions otherwise. Since each solution is one directed edge to y, the in-degree of y is either m or 0 depending as $m|b$.[1]

It then remains, then, only to count the m-ary graphs. It should be clear that the values of a which satisfy $m = \gcd(a, p-1)$ are precisely the same as those that satisfy

$$1 = \gcd(\frac{a}{m}, \frac{p-1}{m}).$$

There are $\phi(\frac{p-1}{m})$ values of $\frac{a}{m}$ which satisfy this equation and are less than $\frac{p-1}{m}$. By multiplying the numbers relatively prime to and less than $\frac{p-1}{m}$ by m, the values a which satisfy $m = \gcd(a, p-1)$ are obtained and the proof is complete. □

Theorem 1 gives a strong indication that the graphs generated by (1) have to be considered separately for different values of m. In Section 4.1, experimental data is provided to support this claim. It should be noted, though, that there are some values of m which lead to completely predicable graphs. For instance, there is one $(p-1)$-ary graph that corresponds to $g \equiv 1 \mod p$. There is also one $(\frac{p-1}{2})$-ary graph that corresponds to $g \equiv -1 \mod p$. In general, however, an m-ary graph is not trivially predictable. This paper will restrict its focus to unary functional graphs (which will be referred to as permutations since they simply permute the numbers 1, ..., $p-1$) and binary functional graphs. The values of g which produce a permutation are precisely those which are primitive roots modulo p. Figure 1 can now be identified more precisely as a binary functional graph since the in-degree of each node is either zero or two.

In cryptography, it is common to look for primes where $p-1$ has at least one large prime factor. For instance, the PRBG described by Gennaro in [6] and mentioned in Section 1 requires the modulus to be of the form $p = 2q+1$ where q is also prime. A prime of this form is known as a safe prime (q is also known as a Sophie Germain prime). These primes are of interest here not only because of their extensive use in cryptography, but also because $p-1$ has only four divisors, namely 1, 2, q and $2q$. It can be quickly verified that there is only one q-ary ($g \equiv -1 \mod p$) and one $2q$-ary ($g \equiv 1 \mod p$) graph generated. More importantly, there are $\phi(q)$ permutations and $\phi(q)$ binary functional graphs which represent the remaining values of g (since $\phi(q)$ is $q-1$). Thus, not only do safe primes provide large numbers of permutations and binary functional graphs, but every graph generated by a safe prime is either trivial (the graphs where g is either 1 or -1) or fits into the theoretical framework presented in Section 3.

3 Theoretical Results

In Theorem 1, it is shown that the in-degree of each node is dependent on the value of both g and p. This is clearly imposing a structure on any functional graphs generated

[1] The structure of this proof is due to personal communication with Joshua Holden.

using (1). It seems reasonable, though, that a large collection of functional graphs generated by using (1) as the transition function would tend toward exhibiting behavior similar to that of a collection of random functional graphs. At a minimum, a complex factorization for $p-1$ would certainly seem to hide the structure imposed by Theorem 1 since the many divisors of $p-1$ would each contribute some graphs. Section 4.1 will give evidence that this is not the case. However, the methods used to obtain the theoretical bounds for the random functional graphs can be extended to analyze m-ary graphs for specific m.

While most of the parameters that are of interest depend on the exact graph generated, the number of image nodes can be computed directly from the values of g and p.

Theorem 2. *The number of image nodes in any m-ary graph is $\frac{p-1}{m}$.*

Proof. The values of y in (1) that have solutions are precisely the image nodes in a functional graph. Then, using the same machinery as in the proof of Theorem 1, let r be any primitive root modulo p and choose a and b such that $g \equiv r^a \mod p$ and $y \equiv r^b \mod p$ for g and y in (1). If, $m = \gcd(a, p-1)$ (or equivalently, the functional graph is m-ary) there is a solution only if $m|b$. Thus, it is sufficient to count the values of b that are multiples of m. A moment's reflection should demonstrate that of the first $p-1$ integers, precisely $\frac{p-1}{m}$ of them are multiples of m. □

Theorem 2 helps to quantify the repercussions of Theorem 1 and the restrictions on in-degree in m-ary graphs. The number of image nodes is a direct function of m which can greatly limit the shapes each graph can take on. None of the other parameters appear to have a generalization as convenient as the image nodes and will be treated as specific parameters in permutations and binary functional graphs.

3.1 Random Functional Graphs

Flajolet and Odlyzko do a thorough analysis of functional graphs in [5]. While none of these results are original, Flajolet and Odlyzko demonstrate that all of these parameters can be estimated through a singularity analysis of generating functions. This appears to be the first method that can be applied to all of these parameters. Their methods can then be adapted for any fixed value of m to estimate the parameters of interest for an m-ary graph. Specifically, the methods will be used to confirm some permutation results and to develop all of the binary functional graph results. The results from [5] are summarized below in Theorem 3.

Theorem 3. *The asymptotic values for the parameters of interest in a random functional*

graph of size n are:

$$\begin{aligned}
&\textit{Number of components} && \frac{\log(2n)+\gamma}{2} && \text{(i)}\\
&\textit{Number of cyclic nodes} && \sqrt{\pi n/2}-\frac{1}{3} && \text{(ii)}\\
&\textit{Number of tail nodes} && n-\sqrt{\pi n/2}+\frac{1}{3} && \text{(iii)}\\
&\textit{Number of terminal nodes} && e^{-1}n && \text{(iv)}\\
&\textit{Number of image nodes} && (1-e^{-1})n && \text{(v)}\\
&\textit{Average cycle length} && \sqrt{\pi n/8} && \text{(vi)}\\
&\textit{Average tail length} && \sqrt{\pi n/8} && \text{(vii)}\\
&\textit{Maximum cycle length} && \sqrt{\frac{\pi n}{2}}\int_0^\infty \left[1-\exp\left(-\int_v^\infty e^{-u}\frac{du}{u}\right)\right] dv && \\
& && \approx 0.78248\sqrt{n} && \text{(viii)}
\end{aligned} \tag{2}$$

In part (i), γ refers to the Euler constant which is approximately 0.57721566. The second order terms for parts (i), (ii), and (iii) were not given in [5], but can be computed with a careful singularity analysis using precisely the same methods used there.

3.2 Permutations

Predicting the behavior of the permutations is, in many ways, much easier than other m-ary graphs. The most important reason for this is that there are no terminal nodes or tail nodes. This follows quickly from the definition of a permutation as a unary functional graph and the fact that the sum of the in-degrees must be the same as the sum of the out-degrees. Each node has an out-degree of exactly one, and if any node were to have an in-degree of zero, then, by the pigeon-hole principle, at least one node must have an in-degree of more than one. This is not allowed so each node must have in-degree of exactly one. Furthermore, since every tail must contain at least one terminal node, this also implies that every node is cyclic. The parameters that can then be determined from the definition of a permutation are given below.

$$\begin{array}{ll}
\text{Number of cyclic nodes} & n\\
\text{Number of tail nodes} & 0\\
\text{Number of terminal nodes} & 0\\
\text{Number of image nodes} & n\\
\text{Average tail length} & 0
\end{array} \tag{3}$$

There are three non-trivial parameters of interest. They are expressed in Theorem 4.

Theorem 4. *The asymptotic values for the number of components, the average cycle length as seen from a random node and the maximum cycle length in a random permutation of*

size n have the following values:

$$\textit{Number of components} \quad \sum_{i=1}^{n} \frac{1}{i} \tag{i}$$

$$\textit{Average cycle length} \quad \frac{n+1}{2} \tag{ii}$$

$$\textit{Maximum cycle length} \quad n \int_0^\infty \left[1 - \exp\left(-\int_v^\infty e^{-u} \frac{du}{u}\right)\right] dv \approx 0.62432965n \tag{iii}$$

Proof. In e.g., [1, Theorem 1], it is shown that the expected number of cycles of length i in a random permutation is asymptotic to $1/i$. Part (i) follows immediately as

$$\text{Number of components} = \sum_{i=1}^{n} \frac{1}{i}.$$

Part (ii) of the Theorem follows from the observation that for each cycle of length i, each of the i nodes will contribute a weight of i for the cycle. Thus, the solution follows

$$\text{Average cycle length} = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{i} i^2 = \frac{n+1}{2}.$$

Part (iii) seems to have first been solved by Shepp and Lloyd in 1966 [12]. An alternative solution and proof more similar to the methods used here is offered by Flajolet and Odlyzko in [4]. □

The relative simplicity of permutations provides an opportunity to introduce the methods that will be developed more fully in Section 3.3. While all of Theorem 4 could be shown using these methods (as well as the parameters that followed from the definition of a permutation), an alternative proof is given only for part (i).

As in [5], the first order of business is to enumerate the permutations in a form that can be converted into generating functions. This can be done as

Permutation	= set(Components)
Component	= cycle(Tree)
Tree	= Node
Node	= Atomic Unit.

A permutation is a set of connected components. Each component is nothing more than a cycle of trees where each tree is only a single node. Clearly, the structure for tree could be omitted in this case. However, this definition adapts more easily to that used in Section 3.3 and provides no further complications. The notation, especially in this case, is fairly intuitive. The conversion into generating functions is quite mechanical (see [5] for the details). The resulting generating functions are

$$f(z) = e^{c(z)} = \frac{1}{1-z} \tag{4}$$

$$c(z) = \log \frac{1}{1-z} \tag{5}$$

$$t(z) = \frac{z}{1-z} \tag{6}$$

where f counts the functional graphs, or permutations in this case, c represented the components and t represents the trees.

From these definitions, the bivariate generating function $\xi(u, z)$ can be determined by taking f and using u to mark the parameter of interest. In this case, the parameter of interest is a component so the generating function is simply

$$\xi(u, z) = e^{u \cdot c(z)} = \exp\left(u \log \frac{1}{1-z}\right).$$

Then, $\Xi(z)$ which is the generating function for the mean of ξ can be defined by simply taking the partial derivative of ξ with respect to u and evaluating at $u = 1$. Performing these operations gives the definition of $\Xi(z)$ to be

$$\Xi(z) = \frac{1}{1-z} \log \frac{1}{1-z}.$$

Performing a singularity analysis[2] of this generating function leads to

$$[z^n]\Xi(z) = \log n + \gamma + O\left(\frac{2 + \log n}{n}\right)$$

where $[z^n]\Xi(z)$ denotes the coefficient of z^n in $\Xi(z)$. This gives the number of components to be asymptotic to $\log n + \gamma$ where γ is the Euler constant defined as

$$\gamma = \lim_{n \to \infty} \left(-\log n + \sum_{i=1}^{n} \frac{1}{i}\right).$$

Thus, $\log n + \gamma$ is asymptotically the same result derived in the proof of Theorem 4(i).

A careful reader may notice that $\Xi(z)$ is an exponential generating function and the coefficient was taken without regard for the $n!$ inherent to all exponential generating functions. Additionally, $\Xi(z)$ contains cumulative data on all of the possible permutations. This should necessitate scaling the coefficient by the number of graphs to obtain the expected results for a single graph. However, since there are $n!$ possible permutations, these two factors cancel themselves out and, in this case, can be ignored. In the following derivations, this is not the case and this normalizing process adds a small complication to the process described above.

3.3 Binary Functional Graphs

While estimates for the parameters investigated here exist in literature for the random functional graphs and permutations, it does not appear similar estimates exist for binary functional graphs. However, the methods in [5] that were introduced with the alternate proof of Theorem 4(i) can be extended to develop these estimates. Imitating the methods of [5], we first need to convert our ideas of a binary functional graph into corresponding generating functions. The machinery is fairly straightforward once we define the following as in [5]:

[2]The analysis in this paper have been performed using the computer algebra program Maple and the packages created as part of the Algorithms Project at INRIA, Rocquencourt, France. The packages can be found online at http://pauillac.inria.fr/algo/libraries/software.html.

BinFunGraph	= set(Components)
Component	= cycle(Node*BinaryTree)
BinaryTree	= Node + Node*set(BinaryTree, cardinality = 2)
Node	= Atomic Unit

This implies that a binary functional graph is a set of components. Each component is a cycle of nodes with each node having an attached binary tree to bring its in-degree to two. A binary tree is either a node (terminal node) or a node with two binary trees attached. Finally, a node is simply an atomic unit. A moment's reflection should indicate that this natural specification does, in fact, specify a binary functional graph. Imitating the transformations in [5, Section 2.1], the generating functions of interest are

$$f(z) = e^{c(z)} = \frac{1}{1 - zb(z)} \tag{7}$$

$$c(z) = \log \frac{1}{1 - zb(z)} \tag{8}$$

$$b(z) = z + \frac{1}{2} z b^2(z) \tag{9}$$

Here f generates the number of binary functional graphs, c generates the number of components, and b generates the number of binary trees of a given size. Solving the quadratic formula for (9), we can produce the following equations f and c which simplify some of the cases:

$$f^*(z) = \frac{1}{\sqrt{1 - 2z^2}} \tag{10}$$

$$c^*(z) = \log \frac{1}{\sqrt{1 - 2z^2}} \tag{11}$$

In order to compute asymptotic forms of any of the statistics of interest, we must first compute an asymptotic form for f or f^* to normalize results. The following derivations give only a highlight of the methods used by Flajolet and Odlyzko. The interested reader is encouraged to see [4, 5] for detailed proofs.

From equation (10) it is clear that there is a singularity at $z = 1/\sqrt{2}$. Performing the analysis as in [5, Section 2], the asymptotic form for f^* falls out quickly as

$$f^*(z) \sim \frac{2^{n/2}}{\sqrt{2\pi n}}. \tag{12}$$

In at least one case, there are some second-order interactions between the error terms of the number of graphs and the appropriate statistic. In these cases, a more exact form of (12) must be used. Expanding one more term in the expansion of f^* gives

$$f^*(z) \sim \frac{2^{n/2}}{\sqrt{2\pi n}} - \frac{2^{n/2}}{4n\sqrt{2\pi n}} = \frac{2^{n/2}(4n-1)}{4n\sqrt{2\pi n}} \tag{13}$$

In most cases, using this more precise expansion of f is not necessary and does not change the results. Therefore, in all but the necessary cases, (12) will be used.

We begin by deriving the results for the most simple parameters.

Theorem 5. *The asymptotic forms for the number of components, number of cyclic nodes, number of tail nodes, number of terminal nodes and number of image nodes in a random binary functional graph of size n, as $n \to \infty$ are*

$$\begin{aligned}
&\textit{Number of components} && \frac{\log(2n)+\gamma}{2} && \text{(i)}\\
&\textit{Number of cyclic nodes} && \sqrt{\pi n/2}-1 && \text{(ii)}\\
&\textit{Number of tail nodes} && n-\sqrt{\pi n/2}+1 && \text{(iii)}\\
&\textit{Number of terminal nodes} && n/2 && \text{(iv)}\\
&\textit{Number of image nodes} && n/2 && \text{(v)}
\end{aligned}$$

In part (i), γ represents the Euler constant which is approximately 0.57721566. The highlights of the proofs as they differ from those in [5] follow.

Proof. It should first be noted that part (ii) and part (iii) are complements of each other. Likewise, parts (iv) and (v) must sum to n. The forms for parts (iii) and (v) follow from the derivation of their partners. As in [5], the following bivariate generating functions need to be defined with parameter u marking the elements of interest. The generating functions for the number of components, number of cyclic nodes and number of terminal nodes are respectively:

$$\xi_1(u,z) = \exp\left(u\log\frac{1}{1-zb(z)}\right) \tag{14}$$

$$\xi_2(u,z) = \frac{1}{1-uzb(z)} \tag{15}$$

$$\xi_3(u,z) = \frac{1}{\sqrt{1-2uz^2}} \tag{16}$$

Equation (16) follows from marking the appropriate element in (9), solving the quadratic formula and substituting into (7). Imitating the methods in [5], the mean value generating function, $\Xi(z)$, is found by taking the partial derivative of $\xi(u,z)$ with respect to u and evaluating at $u=1$. This yields the following results

$$\Xi_1(z) = \frac{1}{1-zb(z)}\log\left(\frac{1}{1-zb(z)}\right) \tag{17}$$

$$\Xi_2(z) = \frac{zb(z)}{(1-zb(z))^2} \tag{18}$$

$$\Xi_3(z) = \frac{z^2}{(1-2z^2)^{3/2}}. \tag{19}$$

Equations (17), (18), and (19) lead to the following expansion around the singularity $z = 1/\sqrt{2}$.

$$\Xi_1(z) = \frac{\log\left(\frac{1}{\sqrt{2}\sqrt{1-z\sqrt{2}}}\right)}{\sqrt{2}\sqrt{1-z\sqrt{2}}} + O\left(\sqrt{1-z\sqrt{2}}\left(\log\left|\frac{1}{1-z\sqrt{2}}\right|+2\right)\right) \tag{20}$$

$$\Xi_2(z) = \frac{1}{2(1-z\sqrt{2})} - \frac{\sqrt{2}}{2\sqrt{1-z\sqrt{2}}} + O(1) \tag{21}$$

$$\Xi_3(z) = \frac{\sqrt{2}}{8(1-z\sqrt{2})^{3/2}} - \frac{5\sqrt{2}}{32\sqrt{1-z\sqrt{2}}} + O\left(\sqrt{1-z\sqrt{2}}\right) \tag{22}$$

Applying singularity analysis as in [5], Equations (20) through (22), lead to the following:

$$[z^n]\Xi_1(z) = \frac{2^{n/2}\log n}{2\sqrt{2\pi n}} + \frac{2^{n/2}(\gamma + \log 2)}{2\sqrt{2\pi n}} + O\left(\frac{2^{n/2}(\log|n| + 2)}{n^{3/2}}\right) \tag{23}$$

$$[z^n]\Xi_2(z) = \frac{2^{n/2}(\sqrt{\pi n} - \sqrt{2})}{2\sqrt{\pi n}} + O\left(\frac{2^{n/2}}{n^{3/2}}\right) \tag{24}$$

$$[z^n]\Xi_3(z) = \frac{2^{n/2}(4n-1)}{8\sqrt{2\pi n}} + O\left(\frac{2^{n/2}}{n^{3/2}}\right) \tag{25}$$

The forms in the statement of the proof follow by normalizing (23) and (24) by (12) and (25) by (13). Parts (iii) and (v) follow from parts (ii) and (iv) respectively since the respective pairs must sum to n.

□

The asymptotic values for the length of a cycle and tail as seen from a random point in the graph are also interesting. The asymptotic forms of these values are given in Theorem 6.

Theorem 6. *The expected values for the cycle size and tail length as seen from a random node in a random binary functional graph of size n are asymptotic to*

$$\textit{Average cycle length} \quad \sqrt{\pi n/8} \tag{i}$$

$$\textit{Average tail length} \quad \sqrt{\pi n/8} \tag{ii}$$

Proof. In order to calculate the average cycle length and average tail length, the generating functions must be manipulated to account for each node in the cycle or tail. This can be done by using the same methods as in the previous proof, but on the component function and taking an additional derivative with respect to z to weight each cycle and tail by the nodes involved. Multiplying again by z replaces the factor lost in the differentiation and by $1/(1-b(z))$ cumulates over all of the components. This strategy is used to prove the result for average cycle size in [5]. More background on the method can be found there.

Let $\xi_1(z)$ be the exponential generating function for the average cycle length and $\xi_2(z)$ be the exponential generating function for the average tail length. Then, they can be defined as

$$\xi_1 = \frac{z}{1-\sqrt{1-2z^2}}\left[\frac{\partial^2}{\partial z \partial u}\log\frac{1}{1-u(1-\sqrt{1-2z^2})}\right]_{u=1} \tag{26}$$

$$\xi_2(z) = \frac{z}{1-\sqrt{1-2z^2}}\left[\frac{\partial^2}{\partial z \partial u}\log\frac{1}{\sqrt{1-2uz^2}}\right]_{u=1} \tag{27}$$

In order to properly mark ξ_2, the marking was done in $b(z)$. The quadratic equation could then be solved and the result inserted back into $c(z)$. Performing a singularity analysis of the two generating functions and normalization by

$$\frac{2^{n/2}}{n\sqrt{2\pi n}}$$

as done in the previous Theorems, lead to the statement of the Theorem. The additional factor of n in the denominator is needed to compensate for the fact that the parameters were estimated across all nodes in the graph and the goal is to determine them from any single random node in the graph. □

The final parameter that needs to be calculated is the maximum cycle length. The result is given as Theorem 7. The proof for this result follows precisely the methods of [5] with substitution of the proper generating function f. Therefore, the proof of Theorem 7 is omitted.

Theorem 7. *The expected length of the largest cycle in a random binary functional graph of size n has the following asymptotic form*

$$\sqrt{\frac{\pi n}{2}} \int_0^\infty \left[1 - \exp\left(-\int_v^\infty e^{-u}\frac{du}{u}\right)\right] dv \approx 0.78248\sqrt{n}$$

4 Observed Results

In [7], heuristics and observed values for the number of small cycles (fixed points and two-cycles) in graphs of the type investigated here are given. Our methods build on this to generate experimental data for the parameters described by the theoretical predictions in Section 3. The method of data collection was straightforward. A prime was chosen as the modulus and then for each $g \in \{1, 2, 3, ..., p-1\}$, the corresponding map or permutation was generated. The results were then computed as averages over all $p-1$ graphs observed. The permutations and binary functional graphs were noted and their results were also tabulated separately. In this manner, the data can be examined in its complete form over all graphs and individually over the permutations and binary functional graphs. The generation and analysis of each of the graphs was handled by C++ code written by the author.

The primes chosen for these calculations were

$$\begin{aligned} 100043 &= 2 \cdot 50021 + 1, \\ 100057 &= 2^3 \cdot 3 \cdot 11 \cdot 379 + 1, \text{ and} \\ 106261 &= 2^2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 23 + 1. \end{aligned}$$

The total number of graphs, permutations and binary functional graphs can be computed using Theorem 1 and are shown in Table 1. The combined results of all functional graphs

	100043	**100057**	**106261**
Permutations	50020	30240	21120
Binary Functional Graphs	50020	15120	10560
Total Functional Graphs	100042	100056	106260

Table 1: The number of permutations, binary functional graphs and total functional graphs associated with $p = 100043$, $p = 100057$, and $p = 106260$.

will be examined first in Section 4.1 where the observed results will be compared to the theoretical framework for random functional graphs given in Theorem 3. In Section 4.2, the observed results for the permutations will be compared to the theoretical results given in Theorem 4. Finally, the observed results for the binary functional graphs will be examined in Section 4.3. Theorems 5 through 7 will provide the theoretical predictions for these values. Since the terminal nodes and tail nodes can be directly computed from the image nodes and cyclic nodes, including them in the collected data does not add any insight. For

this reason, they have both been excluded from the analysis conducted in the following sections. Appendix D gives some of the interesting extremal data such as the longest cycle observed for each prime.

4.1 Combined Results

It would seem that by combining better than one hundred thousand functional graphs generated by (1), the results would tend toward a random functional graph. Theorem 1 shows that the modular exponentiation function imposes some structure onto the functional graphs, but especially if $p-1$ has a complex factorization, the large number of graphs should be able to overcome the structure. However, as Table 2 clearly shows, these graphs are not tending toward a random functional graph. A more complete table which includes the

	100043		**100057**		**106261**	
	Observed	**Error**	**Observed**	**Error**	**Observed**	**Error**
Components	9.235	44.481%	7.603	18.947%	6.742	4.983%
Cyclic Nodes	50271.600	12578.567%	30399.400	7574.478%	21268.600	5110.130%
Image Nodes	75029.000	18.644%	47838.800	24.363%	69435.300	3.374%
Avg Cycle	25088.934	12557.883%	15249.500	7593.148%	10629.500	5103.529%
Avg Tail	197.951	0.130%	114.215	42.380%	92.590	54.674%
Max Cycle	31320.700	12555.466%	19027.821	7587.860%	13259.600	5098.564%

Table 2: The observed results for the three primes over all functional graphs generated and the corresponding percent errors.

predicted values can be found in Appendix A.

There are only a few of the statistics that give a reasonable percent error. The first that stands out is the average tail length for $p = 100043$. This, however, can be easily explained. Since there are only four divisors of $p - 1$, nearly half of the graphs ($(p - 3)/2$ graphs) are permutations. These permutations have no tails, and therefore, the only tail contributions are from binary functional graphs and the $g = 1$ and $g = p - 1$ graphs. Thus, this result is nearly the same as the average tail length statistic that will be presented in Section 4.3.

The only other statistics that have under 15 percent error are the number of components and image nodes for $p = 106261$. This appears to be a result of a more complex factorization for $p - 1$ leading to more variety in the m-ary graphs. The increasing complexity does not appear to hide the graph structure demonstrated by the other statistics, though.

As was stated above, the permutations are notably different than other m-ary graphs due to the fact that each node must be cyclic. If the permutations are removed from the results shown in Table 2, the percent errors do, in fact, decrease in a number of instances. Table 3 gives the results with the permutations and graphs with $g = 1$ and $g = p - 1$ removed. Again, a more complete table can be found in Appendix A. For $p = 100043$ this leaves only the binary functional graphs, but for the other primes the graphs are still quite diverse. This explains why the percent errors nearly go to zero for $p = 100043$ on most of the statistics. The predicted values for a binary functional graph are nearly identical (or even identical) to a random functional graph in many cases. Since there are only binary functional graphs left for $p = 100043$ it is to be expected that the results follow closely those

	100043		100057		106261	
	Observed	**Error**	**Observed**	**Error**	**Observed**	**Error**
Components	6.389	0.045%	5.675	11.217%	5.406	15.821%
Cyclic Nodes	395.303	0.197%	228.389	42.342%	185.412	54.580%
Image Nodes	50021	20.901%	25222.257	60.121%	19600.94	70.819%
Avg Cycle	198.319	0.056%	114.832	42.069%	93.103	54.423%
Avg Tail	197.961	0.125%	114.218	42.379%	92.592	54.673%
Max Cycle	247.261	0.092%	143.023	42.215%	116.054	54.501%

Table 3: The observed results for the three primes over the m-ary graphs for $2 \leq m < \frac{p-1}{2}$ and the corresponding percent errors.

that will be presented in Section 4.3. Namely, the data conforms well to the predictions for binary functional graphs.

The more complex cases, however, do not fare as well. In fact, $p = 106261$ actually has more graphs that are more varied, but the results are actually worse than for $p = 100057$. The data seems to imply that as more graphs are added with a generally higher value of m, there are fewer cyclic nodes, which lead to shorter cycles. Additionally, the tails also seem to become shorter in these graphs. This fits with the results of Theorem 2 which claims that a larger value of m leads to fewer image nodes with which to work.

The results in Tables 2 and 3 give evidence that the graphs should be broken up and analyzed separately by the value of m. The results in Table 3 for $p = 100043$ especially lead to that conclusion as the error goes nearly to zero in the cases where the expected values remained nearly unchanged for random functional graphs to binary functional graphs.

4.2 Permutation Results

The results in Section 4.1 imply that the graphs should be split based on the value of m, or the possible in-degrees of each node. It is also clear from Section 4.1 that the permutations differ greatly from the other functional graphs by the change in percent error which occurred by removing them. Section 3.2 and Theorem 4 clearly indicate the differences in the expected value of many of the parameters between random functional graphs and binary functional graphs. The results of looking at only the values of g that were a primitive root modulo p can be found in Table 4. As in the previous section, a more detailed table which includes the predicted values computed using Theorem 4 can be found in Appendix B.

	100043		100057		106261	
	Observed	**Error**	**Observed**	**Error**	**Observed**	**Error**
Components	12.081	0.083%	12.054	0.306%	12.126	0.205%
Avg Cycle	49980.551	0.082%	50191.352	0.326%	53105.104	0.048%
Max Cycle	62395.488	0.102%	62627.745	0.256%	66245.807	0.144%

Table 4: The observed results for the three primes over the permutations and the corresponding percent errors.

Unlike the results in the previous section, the percent error here is nearly zero in every instance. This seems to indicate that there are no obvious structural differences between a random permutation and a permutation generated by the process used here.

4.3 Binary Functional Graph Results

The binary functional graphs should prove more interesting than the permutations examined in the previous section. Unlike permutations, binary functional graphs do not appear to have been previously studied in detail. The statistics derived from the binary functional graphs and the error when compared to the results derived in Section 3.3 can be found in Table 5. As with the previous two sections, more detailed results which include the actual

	100043		**100057**		**106261**	
	Observed	**Error**	**Observed**	**Error**	**Observed**	**Error**
Components	6.389	0.047%	6.364	0.437%	6.370	0.810%
Cyclic Nodes	395.303	0.029%	395.858	0.105%	408.433	0.217%
Image Nodes	50021	0%	50028	0%	53130	0%
Avg Cycle	198.319	0.056%	197.766	0.230%	202.651	0.795%
Avg Tail	197.961	0.125%	197.550	0.339%	202.422	0.907%
Max Cycle	247.261	0.094%	247.302	0.082%	256.986	0.754%

Table 5: The observed results for the three primes over all binary functional graphs generated and the corresponding percent errors.

predicted values generated using Theorems 5 through 7 can be found in Appendix C.

The number of image nodes came out exactly as expected and predicted by Theorem 2. However, in many other cases the results were nearly as good. In no case is any percent error over one percent and in many cases it is less than 0.1%. The relative size of the error follows the number of binary functional graphs for each prime. This is especially worth noting for $p = 100043$ which has over fifty thousand binary functional graphs while 100057 and 106261 have approximately fifteen thousand and ten thousand respectively. Since having more graphs appears to push the results closer to those derived in Section 3.3, this seems to further support the claim that the results hold for any binary functional graph produced by our mapping.

5 Conclusions and Future Work

The transformation used here to generate functional graphs and permutations is an exceedingly important transformation in cryptography. If the output of the function were to fall into a predictable pattern, it could be an exploitable flaw in many algorithms considered secure today. For instance, the average cycle length seems particularly important for PRBGs since, in many cases, it relates directly to the predictability of the PRBG. As Theorem 1 demonstrates, the use of (1) repeatedly forces a non-trivial structure onto the graphs generated. This is certainly worth investigating as any imposed structure may be open to an exploit. In fact, as Section 4.1 demonstrates, this structure is visible even through a relatively large number of graphs. The notable differences in permutations from other m-ary graphs contributes a large amount of this error since the structure of a permutation

does not allow terminal nodes or tails whereas other m-ary graphs have both. Even after removing the permutations, however, the error is still prevalent and appears to only get worse as the factorization of $p-1$ gets more complex and there are more functional graphs with larger possible in-degrees.

The advantage of using a safe prime is that every non-trivial graph can be analyzed by the theoretical framework laid out in this paper. Their use is also very prevalent in cryptographic applications. For instance, the pseudo-random bit generator specified in [6] requires the use of a safe prime to defend against other attacks. However, the methods used for binary functional graphs in Section 3.3 can and should be extended to larger values of m. In an ideal case, they should be extended in the general case for an m-ary graph that can be specified by

FunctionalGraph	= set(Components)
Component	= cycle(Node*Set(Tree, cardinality = $m-1$))
Tree	= Node + Node*set(Tree, cardinality = m)
Node	= Atomic Unit

The associated generating functions for these functional graphs would be

$$\begin{aligned} f(z) &= e^{c(z)} \\ c(z) &= \log\left(1 - \frac{z}{(m-1)!}t^{m-1}(z)\right)^{-1} \\ t(z) &= z + \frac{z}{m!}t^m(z) \end{aligned}$$

where $f(z)$ is the exponential generating function associated to the functional graphs, $c(z)$ is the exponential generating function associated to the connected components and $t(z)$ is associated to the trees. The methods in Section 3.3 could also be extended to obtain values for additional parameters such as the maximum tail length.

This paper has focused on the graphs generated when the modulus is prime. In practice, though, this is not always the case. For this reason, it could be worthwhile to attempt to extend the type of analysis done here to a composite modulus.

While the data generated for this project appears to confirm that the graphs do tend toward the shape and structure of a random graph of the appropriate type, no data was collected on the distribution of the different parameters. This data could help to give a clearer picture of how closely individual graphs may be expected to exhibit the characteristics of a random graph.

This paper has shown that the functional graphs generated by using (1) do not conform well to the expected shape of a random functional graph. In fact, it has shown that the structure of the graph is largely dependent on a predictable interaction between g and $p-1$. Once this interaction is accounted for, the graphs can be divided into categories and analyzed separately. The number of categories is simply the number of divisors of $p-1$ and there will be $\phi(\frac{p-1}{m})$ graphs of that variety for each m that divides $p-1$. For $m=1$ the result are permutations. These appear to conform well to established theory on the shape of random permutation. If $m=2$, the graphs appear to converge on a general shape suggested by theoretical results derived in Section 3.3 for binary functional graphs. This provides motivation to extend the results to m-ary graphs with larger values of m, as well as to investigate the impact this imposed structure can have on the many cryptographic algorithms that rely on the discrete log and modular exponentiation for their security.

Acknowledgements The author would like to thank his thesis advisor, Joshua Holden, for his help and support throughout this project.

A Combined Data and Results

NOTE: The results marked No Permutations have had the permutations and the graphs generated by $g = 1$ and $g = p - 1$ removed.

Data for $p = 100043$

	Including Permutations			No Permutations		
	Predicted	**Observed**	**Error**	**Predicted**	**Observed**	**Error**
Components	6.392	9.235	44.481%	6.392	6.389	0.045%
Cyclic Nodes	396.083	50217.600	12578.567%	396.083	395.303	0.197%
Image Nodes	63238.605	75029.000	18.644%	50021	63238.605	20.901%
Avg Cycle	198.208	25088.934	12557.883%	198.208	198.319	0.056%
Avg Tail	198.208	197.951	0.130%	198.208	197.961	0.125%
Max Cycle	247.488	31320.700	12555.466%	247.488	247.261	0.092%

Data for $p = 100057$

	Including Permutations			No Permutations		
	Predicted	**Observed**	**Error**	**Predicted**	**Observed**	**Error**
Components	6.392	7.603	18.947%	6.392	5.675	11.217%
Cyclic Nodes	396.110	30399.400	7575.478%	396.111	228.389	42.342%
Image Nodes	63247.455	47838.800	24.362%	63247.455	25222.257	60.121%
Avg Cycle	198.222	15249.500	7593.148%	198.222	114.832	42.069%
Avg Tail	198.222	114.215	42.380%	198.222	114.218	42.379%
Max Cycle	247.505	19027.824	7587.860%	247.511	143.023	42.215%

Data for $p = 106261$

	Including Permutations			No Permutations		
	Predicted	**Observed**	**Error**	**Predicted**	**Observed**	**Error**
Components	6.422	6.742	4.983%	6.422	5.406	15.821%
Cyclic Nodes	408.216	21268.600	5110.130%	408.216	185.412	54.580%
Image Nodes	67169.131	69435.300	3.374%	67169.131	19600.940	70.819%
Avg Cycle	204.275	10629.500	5103.529%	204.275	93.103	54.423%
Avg Tail	204.275	92.590	54.674%	204.275	95.592	54.673%
Max Cycle	255.063	13259.600	5098.564%	255.069	116.054	54.501%

B Permutations Data and Results

Data for $p = 100043$

	Predicted	**Observed**	**Error**
Components	12.091	12.081	0.083%
Avg Cycle	50021.500	49980.551	0.082%
Max Cycle	62459.187	62395.488	0.102%

Data for $p = 100057$

	Predicted	**Observed**	**Error**
Components	12.091	12.054	0.306%
Avg Cycle	50028.500	50191.352	0.326%
Max Cycle	62467.927	62627.745	0.256%

Data for $p = 106261$

	Predicted	**Observed**	**Error**
Components	12.151	12.126	0.205%
Avg Cycle	53130.500	53105.104	0.048%
Max Cycle	66341.269	66245.807	0.144%

C Binary Functional Graphs Data and Results

Data for $p = 100043$

	Predicted	**Observed**	**Error**
Components	6.392	6.389	0.047%
Cyclic Nodes	395.416	395.303	0.029%
Image Nodes	50021	50021	0%
Avg Cycle	198.208	198.319	0.056%
Avg Tail	198.208	197.961	0.125%
Max Cycle	247.494	247.261	0.094%

Data for $p = 100057$

	Predicted	**Observed**	**Error**
Components	6.392	6.364	0.437%
Cyclic Nodes	395.447	395.858	0.105%
Image Nodes	50028	50028	0%
Avg Cycle	198.222	198.766	0.230%
Avg Tail	198.222	197.550	0.339%
Max Cycle	247.505	247.302	0.082%

Data for $p = 106261$

	Predicted	**Observed**	**Error**
Components	6.422	6.370	0.810%
Cyclic Nodes	407.550	408.433	0.217%
Image Nodes	53130	53130	0%
Avg Cycle	204.275	202.651	0.795%
Avg Tail	204.275	202.422	0.907%
Max Cycle	255.063	256.986	0.754%

D Extremal Data

For $p = 100043$, the longest cycle observed was 100042 which occurred for two different values of g. They were $g = 20812$ and $g = 94034$. The longest tail had a length of 1448 and was observed when $g = 89339$. There were five instances where the graphs contained no cycles longer than one which occurred for $g = 1$, 72116, 91980, 95997, and 100042.

The graphs generated by $p = 100057$ had an overall longest cycle of 100052 when $g = 58303$. The longest tail observed was 1589 when $g = 18115$. There were also 26 different values of g that produced a graph that did not have a cycle longer than one.

The largest cycle observed in graphs generated using $p = 106261$ was 106257 when $g = 102141$. The longest tail was 35822 when $g = 1480$. There were 92 different values of g that produced graphs with no cycles longer than a fixed point.

E Code to Produce Experimental Data

E.1 bn_prime.h

```
/* Auto generated by bn_prime.pl */
/* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
 * All rights reserved.
 *
 * This package is an SSL implementation written
 * by Eric Young (eay@cryptsoft.com).
 * The implementation was written so as to conform with Netscapes SSL.
 *
 * This library is free for commercial and non-commercial use as long as
 * the following conditions are aheared to.  The following conditions
 * apply to all code found in this distribution, be it the RC4, RSA,
 * lhash, DES, etc., code; not just the SSL code.  The SSL documentation
 * included with this distribution is covered by the same copyright terms
 * except that the holder is Tim Hudson (tjh@cryptsoft.com).
 *
 * Copyright remains Eric Young's, and as such any Copyright notices in
 * the code are not to be removed.
 * If this package is used in a product, Eric Young should be given attribution
 * as the author of the parts of the library used.
 * This can be in the form of a textual message at program startup or
 * in documentation (online or textual) provided with the package.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 *    "This product includes cryptographic software written by
 *     Eric Young (eay@cryptsoft.com)"
 *    The word 'cryptographic' can be left out if the rouines from the library
 *    being used are not cryptographic related :-).
 * 4. If you include any Windows specific code (or a derivative thereof) from
 *    the apps directory (application code) you must include an acknowledgement:
 *    "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
 *
 * THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
```

```
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * The licence and distribution terms for any publically available version or
 * derivative of this code cannot be changed.  i.e. this code cannot simply be
 * copied and put under another distribution licence
 * [including the GNU Public Licence.]
 */

#ifndef EIGHT_BIT
#define NUMPRIMES 2048
#else
#define NUMPRIMES 54
#endif
static const unsigned int primes[NUMPRIMES]=
{
   2,   3,   5,   7,  11,  13,  17,  19,
  23,  29,  31,  37,  41,  43,  47,  53,
  59,  61,  67,  71,  73,  79,  83,  89,
  97, 101, 103, 107, 109, 113, 127, 131,
 137, 139, 149, 151, 157, 163, 167, 173,
 179, 181, 191, 193, 197, 199, 211, 223,
 227, 229, 233, 239, 241, 251,
#ifndef EIGHT_BIT
 257, 263,
 269, 271, 277, 281, 283, 293, 307, 311,
 313, 317, 331, 337, 347, 349, 353, 359,
 367, 373, 379, 383, 389, 397, 401, 409,
 419, 421, 431, 433, 439, 443, 449, 457,
 461, 463, 467, 479, 487, 491, 499, 503,
 509, 521, 523, 541, 547, 557, 563, 569,
 571, 577, 587, 593, 599, 601, 607, 613,
 617, 619, 631, 641, 643, 647, 653, 659,
 661, 673, 677, 683, 691, 701, 709, 719,
 727, 733, 739, 743, 751, 757, 761, 769,
 773, 787, 797, 809, 811, 821, 823, 827,
 829, 839, 853, 857, 859, 863, 877, 881,
 883, 887, 907, 911, 919, 929, 937, 941,
 947, 953, 967, 971, 977, 983, 991, 997,
1009,1013,1019,1021,1031,1033,1039,1049,
1051,1061,1063,1069,1087,1091,1093,1097,
1103,1109,1117,1123,1129,1151,1153,1163,
1171,1181,1187,1193,1201,1213,1217,1223,
```

1229,1231,1237,1249,1259,1277,1279,1283,
1289,1291,1297,1301,1303,1307,1319,1321,
1327,1361,1367,1373,1381,1399,1409,1423,
1427,1429,1433,1439,1447,1451,1453,1459,
1471,1481,1483,1487,1489,1493,1499,1511,
1523,1531,1543,1549,1553,1559,1567,1571,
1579,1583,1597,1601,1607,1609,1613,1619,
1621,1627,1637,1657,1663,1667,1669,1693,
1697,1699,1709,1721,1723,1733,1741,1747,
1753,1759,1777,1783,1787,1789,1801,1811,
1823,1831,1847,1861,1867,1871,1873,1877,
1879,1889,1901,1907,1913,1931,1933,1949,
1951,1973,1979,1987,1993,1997,1999,2003,
2011,2017,2027,2029,2039,2053,2063,2069,
2081,2083,2087,2089,2099,2111,2113,2129,
2131,2137,2141,2143,2153,2161,2179,2203,
2207,2213,2221,2237,2239,2243,2251,2267,
2269,2273,2281,2287,2293,2297,2309,2311,
2333,2339,2341,2347,2351,2357,2371,2377,
2381,2383,2389,2393,2399,2411,2417,2423,
2437,2441,2447,2459,2467,2473,2477,2503,
2521,2531,2539,2543,2549,2551,2557,2579,
2591,2593,2609,2617,2621,2633,2647,2657,
2659,2663,2671,2677,2683,2687,2689,2693,
2699,2707,2711,2713,2719,2729,2731,2741,
2749,2753,2767,2777,2789,2791,2797,2801,
2803,2819,2833,2837,2843,2851,2857,2861,
2879,2887,2897,2903,2909,2917,2927,2939,
2953,2957,2963,2969,2971,2999,3001,3011,
3019,3023,3037,3041,3049,3061,3067,3079,
3083,3089,3109,3119,3121,3137,3163,3167,
3169,3181,3187,3191,3203,3209,3217,3221,
3229,3251,3253,3257,3259,3271,3299,3301,
3307,3313,3319,3323,3329,3331,3343,3347,
3359,3361,3371,3373,3389,3391,3407,3413,
3433,3449,3457,3461,3463,3467,3469,3491,
3499,3511,3517,3527,3529,3533,3539,3541,
3547,3557,3559,3571,3581,3583,3593,3607,
3613,3617,3623,3631,3637,3643,3659,3671,
3673,3677,3691,3697,3701,3709,3719,3727,
3733,3739,3761,3767,3769,3779,3793,3797,
3803,3821,3823,3833,3847,3851,3853,3863,
3877,3881,3889,3907,3911,3917,3919,3923,
3929,3931,3943,3947,3967,3989,4001,4003,
4007,4013,4019,4021,4027,4049,4051,4057,
4073,4079,4091,4093,4099,4111,4127,4129,
4133,4139,4153,4157,4159,4177,4201,4211,
4217,4219,4229,4231,4241,4243,4253,4259,

4261,4271,4273,4283,4289,4297,4327,4337,
4339,4349,4357,4363,4373,4391,4397,4409,
4421,4423,4441,4447,4451,4457,4463,4481,
4483,4493,4507,4513,4517,4519,4523,4547,
4549,4561,4567,4583,4591,4597,4603,4621,
4637,4639,4643,4649,4651,4657,4663,4673,
4679,4691,4703,4721,4723,4729,4733,4751,
4759,4783,4787,4789,4793,4799,4801,4813,
4817,4831,4861,4871,4877,4889,4903,4909,
4919,4931,4933,4937,4943,4951,4957,4967,
4969,4973,4987,4993,4999,5003,5009,5011,
5021,5023,5039,5051,5059,5077,5081,5087,
5099,5101,5107,5113,5119,5147,5153,5167,
5171,5179,5189,5197,5209,5227,5231,5233,
5237,5261,5273,5279,5281,5297,5303,5309,
5323,5333,5347,5351,5381,5387,5393,5399,
5407,5413,5417,5419,5431,5437,5441,5443,
5449,5471,5477,5479,5483,5501,5503,5507,
5519,5521,5527,5531,5557,5563,5569,5573,
5581,5591,5623,5639,5641,5647,5651,5653,
5657,5659,5669,5683,5689,5693,5701,5711,
5717,5737,5741,5743,5749,5779,5783,5791,
5801,5807,5813,5821,5827,5839,5843,5849,
5851,5857,5861,5867,5869,5879,5881,5897,
5903,5923,5927,5939,5953,5981,5987,6007,
6011,6029,6037,6043,6047,6053,6067,6073,
6079,6089,6091,6101,6113,6121,6131,6133,
6143,6151,6163,6173,6197,6199,6203,6211,
6217,6221,6229,6247,6257,6263,6269,6271,
6277,6287,6299,6301,6311,6317,6323,6329,
6337,6343,6353,6359,6361,6367,6373,6379,
6389,6397,6421,6427,6449,6451,6469,6473,
6481,6491,6521,6529,6547,6551,6553,6563,
6569,6571,6577,6581,6599,6607,6619,6637,
6653,6659,6661,6673,6679,6689,6691,6701,
6703,6709,6719,6733,6737,6761,6763,6779,
6781,6791,6793,6803,6823,6827,6829,6833,
6841,6857,6863,6869,6871,6883,6899,6907,
6911,6917,6947,6949,6959,6961,6967,6971,
6977,6983,6991,6997,7001,7013,7019,7027,
7039,7043,7057,7069,7079,7103,7109,7121,
7127,7129,7151,7159,7177,7187,7193,7207,
7211,7213,7219,7229,7237,7243,7247,7253,
7283,7297,7307,7309,7321,7331,7333,7349,
7351,7369,7393,7411,7417,7433,7451,7457,
7459,7477,7481,7487,7489,7499,7507,7517,
7523,7529,7537,7541,7547,7549,7559,7561,
7573,7577,7583,7589,7591,7603,7607,7621,

7639,7643,7649,7669,7673,7681,7687,7691,
7699,7703,7717,7723,7727,7741,7753,7757,
7759,7789,7793,7817,7823,7829,7841,7853,
7867,7873,7877,7879,7883,7901,7907,7919,
7927,7933,7937,7949,7951,7963,7993,8009,
8011,8017,8039,8053,8059,8069,8081,8087,
8089,8093,8101,8111,8117,8123,8147,8161,
8167,8171,8179,8191,8209,8219,8221,8231,
8233,8237,8243,8263,8269,8273,8287,8291,
8293,8297,8311,8317,8329,8353,8363,8369,
8377,8387,8389,8419,8423,8429,8431,8443,
8447,8461,8467,8501,8513,8521,8527,8537,
8539,8543,8563,8573,8581,8597,8599,8609,
8623,8627,8629,8641,8647,8663,8669,8677,
8681,8689,8693,8699,8707,8713,8719,8731,
8737,8741,8747,8753,8761,8779,8783,8803,
8807,8819,8821,8831,8837,8839,8849,8861,
8863,8867,8887,8893,8923,8929,8933,8941,
8951,8963,8969,8971,8999,9001,9007,9011,
9013,9029,9041,9043,9049,9059,9067,9091,
9103,9109,9127,9133,9137,9151,9157,9161,
9173,9181,9187,9199,9203,9209,9221,9227,
9239,9241,9257,9277,9281,9283,9293,9311,
9319,9323,9337,9341,9343,9349,9371,9377,
9391,9397,9403,9413,9419,9421,9431,9433,
9437,9439,9461,9463,9467,9473,9479,9491,
9497,9511,9521,9533,9539,9547,9551,9587,
9601,9613,9619,9623,9629,9631,9643,9649,
9661,9677,9679,9689,9697,9719,9721,9733,
9739,9743,9749,9767,9769,9781,9787,9791,
9803,9811,9817,9829,9833,9839,9851,9857,
9859,9871,9883,9887,9901,9907,9923,9929,
9931,9941,9949,9967,9973,10007,10009,10037,
10039,10061,10067,10069,10079,10091,10093,10099,
10103,10111,10133,10139,10141,10151,10159,10163,
10169,10177,10181,10193,10211,10223,10243,10247,
10253,10259,10267,10271,10273,10289,10301,10303,
10313,10321,10331,10333,10337,10343,10357,10369,
10391,10399,10427,10429,10433,10453,10457,10459,
10463,10477,10487,10499,10501,10513,10529,10531,
10559,10567,10589,10597,10601,10607,10613,10627,
10631,10639,10651,10657,10663,10667,10687,10691,
10709,10711,10723,10729,10733,10739,10753,10771,
10781,10789,10799,10831,10837,10847,10853,10859,
10861,10867,10883,10889,10891,10903,10909,10937,
10939,10949,10957,10973,10979,10987,10993,11003,
11027,11047,11057,11059,11069,11071,11083,11087,
11093,11113,11117,11119,11131,11149,11159,11161,

11171,11173,11177,11197,11213,11239,11243,11251,
11257,11261,11273,11279,11287,11299,11311,11317,
11321,11329,11351,11353,11369,11383,11393,11399,
11411,11423,11437,11443,11447,11467,11471,11483,
11489,11491,11497,11503,11519,11527,11549,11551,
11579,11587,11593,11597,11617,11621,11633,11657,
11677,11681,11689,11699,11701,11717,11719,11731,
11743,11777,11779,11783,11789,11801,11807,11813,
11821,11827,11831,11833,11839,11863,11867,11887,
11897,11903,11909,11923,11927,11933,11939,11941,
11953,11959,11969,11971,11981,11987,12007,12011,
12037,12041,12043,12049,12071,12073,12097,12101,
12107,12109,12113,12119,12143,12149,12157,12161,
12163,12197,12203,12211,12227,12239,12241,12251,
12253,12263,12269,12277,12281,12289,12301,12323,
12329,12343,12347,12373,12377,12379,12391,12401,
12409,12413,12421,12433,12437,12451,12457,12473,
12479,12487,12491,12497,12503,12511,12517,12527,
12539,12541,12547,12553,12569,12577,12583,12589,
12601,12611,12613,12619,12637,12641,12647,12653,
12659,12671,12689,12697,12703,12713,12721,12739,
12743,12757,12763,12781,12791,12799,12809,12821,
12823,12829,12841,12853,12889,12893,12899,12907,
12911,12917,12919,12923,12941,12953,12959,12967,
12973,12979,12983,13001,13003,13007,13009,13033,
13037,13043,13049,13063,13093,13099,13103,13109,
13121,13127,13147,13151,13159,13163,13171,13177,
13183,13187,13217,13219,13229,13241,13249,13259,
13267,13291,13297,13309,13313,13327,13331,13337,
13339,13367,13381,13397,13399,13411,13417,13421,
13441,13451,13457,13463,13469,13477,13487,13499,
13513,13523,13537,13553,13567,13577,13591,13597,
13613,13619,13627,13633,13649,13669,13679,13681,
13687,13691,13693,13697,13709,13711,13721,13723,
13729,13751,13757,13759,13763,13781,13789,13799,
13807,13829,13831,13841,13859,13873,13877,13879,
13883,13901,13903,13907,13913,13921,13931,13933,
13963,13967,13997,13999,14009,14011,14029,14033,
14051,14057,14071,14081,14083,14087,14107,14143,
14149,14153,14159,14173,14177,14197,14207,14221,
14243,14249,14251,14281,14293,14303,14321,14323,
14327,14341,14347,14369,14387,14389,14401,14407,
14411,14419,14423,14431,14437,14447,14449,14461,
14479,14489,14503,14519,14533,14537,14543,14549,
14551,14557,14561,14563,14591,14593,14621,14627,
14629,14633,14639,14653,14657,14669,14683,14699,
14713,14717,14723,14731,14737,14741,14747,14753,
14759,14767,14771,14779,14783,14797,14813,14821,

```
14827,14831,14843,14851,14867,14869,14879,14887,
14891,14897,14923,14929,14939,14947,14951,14957,
14969,14983,15013,15017,15031,15053,15061,15073,
15077,15083,15091,15101,15107,15121,15131,15137,
15139,15149,15161,15173,15187,15193,15199,15217,
15227,15233,15241,15259,15263,15269,15271,15277,
15287,15289,15299,15307,15313,15319,15329,15331,
15349,15359,15361,15373,15377,15383,15391,15401,
15413,15427,15439,15443,15451,15461,15467,15473,
15493,15497,15511,15527,15541,15551,15559,15569,
15581,15583,15601,15607,15619,15629,15641,15643,
15647,15649,15661,15667,15671,15679,15683,15727,
15731,15733,15737,15739,15749,15761,15767,15773,
15787,15791,15797,15803,15809,15817,15823,15859,
15877,15881,15887,15889,15901,15907,15913,15919,
15923,15937,15959,15971,15973,15991,16001,16007,
16033,16057,16061,16063,16067,16069,16073,16087,
16091,16097,16103,16111,16127,16139,16141,16183,
16187,16189,16193,16217,16223,16229,16231,16249,
16253,16267,16273,16301,16319,16333,16339,16349,
16361,16363,16369,16381,16411,16417,16421,16427,
16433,16447,16451,16453,16477,16481,16487,16493,
16519,16529,16547,16553,16561,16567,16573,16603,
16607,16619,16631,16633,16649,16651,16657,16661,
16673,16691,16693,16699,16703,16729,16741,16747,
16759,16763,16787,16811,16823,16829,16831,16843,
16871,16879,16883,16889,16901,16903,16921,16927,
16931,16937,16943,16963,16979,16981,16987,16993,
17011,17021,17027,17029,17033,17041,17047,17053,
17077,17093,17099,17107,17117,17123,17137,17159,
17167,17183,17189,17191,17203,17207,17209,17231,
17239,17257,17291,17293,17299,17317,17321,17327,
17333,17341,17351,17359,17377,17383,17387,17389,
17393,17401,17417,17419,17431,17443,17449,17467,
17471,17477,17483,17489,17491,17497,17509,17519,
17539,17551,17569,17573,17579,17581,17597,17599,
17609,17623,17627,17657,17659,17669,17681,17683,
17707,17713,17729,17737,17747,17749,17761,17783,
17789,17791,17807,17827,17837,17839,17851,17863,
#endif
};
```

E.2 LinkedList

This is a linked list class that I implemented to provide a dynamic structure and allow explicit control over the allocation and deallocation of memory.

E.2.1 LinkedList.h

```
#ifndef LINKEDLIST_HEADER
#define LINKEDLIST_HEADER

#define NULL 0

#include<iostream>
#include<fstream>
#include "mpi.h"

typedef struct Node {
    int value;
    Node * next;
} Node;

class LinkedList {

public:

    LinkedList();

    void makeEmpty();
    void insert(int);
    int get(int);
    int length();
    int find(int);
    int* toArray();
    void destroyArray();

private:

    Node* head;
    Node* tail;
    int size;
    int * array;

    int get_r(int, Node);
    int find_r(int, Node, int);
    void deleteList(Node*);


};

class IndexOutofBoundsException {
public:
    IndexOutofBoundsException()
```

```
        : message("attempted access to an invalid data structure element"){}
        const char *what() const {return message;}
private:
    const char *message;
};

#endif
```

E.2.2 LinkedList.cpp

```
#include "LinkedList.h"

LinkedList::LinkedList() {
   head = tail = NULL;
   size = 0;
}

void LinkedList::makeEmpty() {
   if(head != NULL) deleteList(head);
   head = tail = NULL;
   size = 0;
}

void LinkedList::insert(int val) {
   if (size == 0) {
      head = new Node();
      if(head == NULL) {
         std::cerr<<"Error allocating new nodes.\n";
         ofstream status("status.txt");
         status << "Exiting\n";
         status.close();
         MPI_Finalize();
         exit(1);
      }
      (*head).value = val;
      (*head).next = NULL;
      tail = head;
      size++;
      return;
   }
   Node* temp = new Node();;
   if(temp == NULL) {
      std::cerr<<"Error allocating new nodes.\n";
      ofstream status("status.txt");
      status << "Exiting\n";
      status.close();
```

```
      MPI_Finalize();
      exit(1);
   }
   (*temp).next = NULL;
   (*temp).value = val;
   (*tail).next = temp;
   tail = temp;
   temp = NULL;
   delete temp;
   size++;
}

int LinkedList::length() {
   return size;
}

int LinkedList::get(int m){
   if (m > size) throw IndexOutofBoundsException();
   return get_r(m, *head);
}

int LinkedList::get_r(int m, Node node){
   if(m == 0) return node.value;
      return get_r(m-1, *node.next);
}

int LinkedList::find(int n) {
   return find_r(n, *head, 0);
}

int LinkedList::find_r(int n, Node node, int index) {
   if(node.value == n) return index;
   if(node.next == NULL) return -1;
   return find_r(n, *node.next, index+1);
}

int* LinkedList::toArray() {
   array = new int[size];
   if(array == NULL) {
      std::cout<<"Error allocating array for list\n";
      ofstream status("status.txt");
      status << "Exiting\n";
      status.close();
      MPI_Finalize();
      exit(1);
      return array;
   }
   if(size==0) return array;
```

```
   Node node = *head;
   for(int i = 0; i < size; i++) {
      array[i] = node.value;
      if(node.next != NULL)
         node = *node.next;
   }
   return array;
}

void LinkedList::destroyArray() {
   delete[] array;
}

void LinkedList::deleteList(Node* n) {
   Node* next;
   while(n != NULL) {
      next = (*n).next;
      delete n;
      n = next;
   }
}
```

E.3 Version 1

This version of the program will generate all graphs for the given value of p. It will then track the statistics for the results of all graphs, permutations and other functional graphs.

E.3.1 Version1.cpp

```
#include "LinkedList.h"
#include "ca1.h"
#include <exception>
#include <string>
#include<fstream>
#include "mpi.h"

using std::exception;
using std::cout;
using std::endl;
using std::string;


int main(int argc, char* argv[]) {
   MPI_Init(&argc, &argv);

   try {
```

```
      run();

   } catch (...) {
     //do nothing, just let MPI_Finalize() run
   }
   MPI_Finalize();
   ofstream status("status.txt");
   status << "Exiting program\n";
   status.close();
   return 0;
}
```

E.3.2 ca1.h

```
#ifndef CA1
#define CA1

#include "LinkedList.h"
#include <fstream>
#include <string>
#include <sstream>
#include <iostream>
#include <math.h>
#include <time.h>
#include "bn_prime.h"

using std::string;
using std::ofstream;
using std::ifstream;



#define n 106261

void run();
int m_exp(int, int, int);
long long ml_exp(long long, int, long long);
void computeResults(int, const int*, const int*, int*, int*,
                    int*, int*, int*, int*);
void writeResults(int, int*, int *, int*);
void setArrays(int*,bool*, int*, bool*);
bool isPrimRoot(int);
bool isRelPrime(int);
void writeTotalResults(const int*, const int*, const int*,
                       const int*, const int*, const int*,
                       const int*, const int*, const int*,
```

```
                        const int*, const int*, const int*,
                        const int*, const int*, const int*,
                        const int*, const int*, const int*,
                        const int*, const int*, const int*,
                        const int*, const int*, const int*,
                        const int*, const int*, const int*,
                        const int*, const int*, const int*);
int gcd(int, int);
bool isPrime(int);
bool MillerRabin(int, int, int, int);


template<class F1, class F2> void convert(F1 in, F2 out){
   std::stringstream ss;
   ss << in;
   ss>> out;
}

#endif
```

E.3.3 ca1.cpp

```
#include "ca1.h"


int m_exp(int b, int r, int num = n) {
   return (int) ml_exp((long long) b, r, (long long) num);
}

long long ml_exp(long long b, int r, long long num) {
   if (r == 0) return 1;
   if(r % 2 == 0) {
      long long result = ml_exp(b,r/2,num);
      return result * result % num;
   }
   long long result = ml_exp(b,r/2,num);
   return (b * result % num) * result % num;
}

void computeResults(int base, const int* cycle, const int* toCycle,
                    int* allCResults, int* allTResults, int* allToCycleResults,
                    int* allSplitCResults, int* allSplitTResults,
                    int* allSplitToCResults) {

   for(int i = 0; i < n; i++) {
      allSplitToCResults[toCycle[i]]++;
```

```
      allToCycleResults[toCycle[i]]++;
      if(cycle[i] < 0) {
         allSplitTResults[-1*cycle[i]]++;
         allTResults[-1*cycle[i]]++;
      }
      else if (cycle[i] > 0) {
         allSplitCResults[cycle[i]]++;
         allCResults[cycle[i]]++;
      }
   }
}


void writeResults(int base, int* cycleResults, int* tailResults,
                  int* toCycleResults){
   bool primRoot = isPrimRoot(base);
   bool relPrime = isRelPrime(base);
   char* file = new char[20];
   convert<int, char*>(n, file);
   string fileStr = file;
   if (primRoot && relPrime)
      fileStr+="PRRP";
   else if (primRoot)
      fileStr+="PR";
   else if (relPrime)
      fileStr+= "RP";
   else
      fileStr += "NONE";
   fileStr += ".txt";
   ifstream in(fileStr.c_str());
   int prevC, prevT, prevToC;
   for(int i = 1; i <= n; i++){
      in>>prevC;
      in.ignore(1,0);
      in>> prevT;
      in.ignore(1,0);
      in>> prevToC;
      in.ignore(1,0);
      cycleResults[i]+= prevC;
      tailResults[i]+= prevT;
      toCycleResults[i] += prevToC;
   }
   in.close();
   ofstream out(fileStr.c_str());
   for(int i = 1; i <= n; i++)
      out<<cycleResults[i] << " " <<tailResults[i]<< " " << toCycleResults[i]
         << '\n';
   out.close();
```

```
   delete file;
}


bool isPrimRoot(int base) {
   if(!isPrime(n)) return false;

   int n_1 = n-1;

   if ((unsigned)n_1 > (primes[NUMPRIMES-1]*primes[NUMPRIMES-1]))
      std::cerr<<"Error in Primitive Root Testing, n could "
               <<"have prime factor too large for testing."<<std::endl;
   int n1 = n_1;
   int index = 0;
   int p;
   while(n1 > 1 && index < NUMPRIMES) {
      if((n1 % primes[index]) == 0) {
         p = primes[index];
         while((n1 % primes[index] == 0)) n1/=primes[index];
         if(m_exp(base,n_1/p) == 1) return false;
         if(n1 == 50021) return (m_exp(base,n_1/50021) != 1);
      }
      index++;
   }
   return true;
}

bool isPrime(int num) {
   for(int i = 0; i < 50;i++) {
      if(primes[i] > (unsigned)num) return true;
      if((num % primes[i] == 0) && (num!=primes[i])) return false;
   }

   int k = 0;
   int q = num-1;
   while(q % 2 == 0) {
      k++;
      q >>= 1;
   }
   srand(time(0));
   int a;
   for(int i = 0; i < 10; i++) {
   a = (rand() % (num-2)) + 1;
   if(!MillerRabin(num, k, q, a)) return false;
   }

   return true;
}
```

```
bool MillerRabin(int num, int k, int q, int a) {
   int n1 = num-1;
   if(m_exp(a,q, num) == 1) return true;
   for(int i = 0; i < k; i++)
   if(m_exp(a,(int)pow(2,i)*q, num) == n1) return true;
   return false;
}


bool isRelPrime(int base) {
   return gcd(base, n-1) == 1;
}

int gcd(int a, int b) {
   int r = a % b;
   int d = b;
   int c;
   while (r > 0) {
      c = d;
      d = r;
      r = c % d;
   }
   return d;
}

void setArrays(int * cycle, bool* visit, int* toCycle, bool* image){
   for(int i = 0; i < n; i++){
      visit[i] = false;
      cycle[i] = 0;
      toCycle[i] = 0;
      image[i] = false;
   }
}


void writeTotalResults(const int* allCResults, const int* allTResults,
                       const int* allToCycleResults, const int* allPRCResults,
                       const int* allPRTResults, const int* allPRToCResults,
                       const int* allNONECResults, const int* allNONETResults,
                       const int* allNONEToCResults, const int* maxTAll,
                       const int* maxTPR,  const int* maxTNone,
                       const int* maxCAll, const int* maxCPR,  const int* maxCNone,
                       const int* terminalAll, const int* terminalPR,
                       const int* terminalNone) {
```

```
   char* file = new char[20];
   convert<int, char*>(n, file);
   string fileStr = file;
   fileStr += "all.txt";
   ofstream out(fileStr.c_str());
   for(int i = 1; i <= n; i++) {
      if(i == 1 || i == n)
         out<< allCResults[i]/i << " " << allTResults[i] << " "
            << allToCycleResults[i] << " " << 0 <<" " << 0 << " "
            << 0 <<'\n';
      else
         out<< allCResults[i]/i << " " << allTResults[i] << " "
            << allToCycleResults[i] << " " << terminalAll[i] << " "
            << maxCAll[i] << " " << maxTAll[i] << '\n';
   }
   out.close();
   convert<int, char*>(n, file);
   string filePR = strcat(file,"PR.txt");
   convert<int, char*>(n, file);
   string fileNotPR = strcat(file, "NotPR.txt");
   ofstream outPR(filePR.c_str());
   ofstream outNotPR(fileNotPR.c_str());
   for(int i = 1; i <= n; i++) {
      if(i == 1 || i == n) {
         outPR << allPRCResults[i]/i << " " << allPRTResults[i]
               << " " << allPRToCResults[i] << " " << 0 << " " << 0 << " "
               << 0 << '\n';
         outNotPR << allNONECResults[i]/i << " " << allNONETResults[i] << " "
                  << allNONEToCResults[i] << " " << 0 << " "
                  << 0 << " " << 0 << '\n';
      } else {
         outPR << allPRCResults[i]/i << " " << allPRTResults[i] << " "
               << allPRToCResults[i] << " " << terminalPR[i] <<
               " " << maxCPR[i] << " " << maxTPR[i] << '\n';
         outNotPR << allNONECResults[i]/i << " " << allNONETResults[i] << " "
                  << allNONEToCResults[i] << " " << terminalNone[i] << " "
                  << maxCNone[i] << " " << maxTNone[i] << '\n';
      }
   }
   outPR.close();
   outNotPR.close();
   delete file;
}


void run() {
   ofstream status("status2.txt");
```

```
status << "Allocating...\n";
status.close();
bool* visit = new bool[n];
bool* image = new bool[n];
int* maxTAll = new int[n];
int* maxTPR = new int[n];
int* maxTNone = new int[n];
int* maxCAll = new int[n];
int* maxCPR = new int[n];
int* maxCNone = new int[n];
int* terminalAll = new int[n];
int* terminalPR = new int[n];
int* terminalNone = new int[n];
int *cycle= new int[n];
int *toCycle = new int[n];
int *allCResults = new int[n+1];
int *allTResults = new int[n+1];
int *allToCycleResults = new int[n+1];
int *allPRCResults = new int[n+1];
int *allNONECResults = new int[n+1];
int *allPRTResults = new int[n+1];
int *allNONETResults = new int[n+1];
int *allPRToCResults = new int[n+1];
int *allNONEToCResults = new int[n+1];

int next, size, loc, baseTail, cycleLength, mC, mT, terminal;
LinkedList list;
int* listArray;
bool PR;
for(int i = 0; i <= n; i++){
   allCResults[i] = 0;
   allTResults[i] = 0;
   allToCycleResults[i] = 0;
   allPRCResults[i] = 0;
   allNONECResults[i] = 0;
   allPRTResults[i] = 0;
   allNONETResults[i] = 0;
   allPRToCResults[i] = 0;
   allNONEToCResults[i] = 0;
   if(i < n) {
      maxTAll[i] = 0;
      maxTPR[i] = 0;
      maxTNone[i] = 0;
      maxCAll[i] = 0;
      maxCPR[i] = 0;
      maxCNone[i] = 0;
      terminalAll[i] = 0;
      terminalPR[i] = 0;
```

```
      terminalNone[i] = 0;
   }
}
status.open("status2.txt", fstream::app);
status << "Starting...\n";
status.close();
for(int base = 2; base < n-1; base++) {
   mC = 0;
   mT = 0;
   setArrays(cycle, visit, toCycle, image);
   if(base % 100 == 0) {
      status.open("status2.txt");
      status << "Base is " << base << std::endl;
      status.close();
   }
   for(int i = 1; i < n; i++) {
      list.makeEmpty();
      next = i;
      list.insert(next);
      while(!visit[next]){
         visit[next] = true;
         next = m_exp(base,next);
         image[next] = true;
         list.insert(next);
      }
      if(cycle[next] != 0) {
         if(cycle[next] > 0) {//all tail into cycle
            listArray = list.toArray();
            size = list.length();
            loc = list.find(next);
            cycleLength = cycle[listArray[size-1]];
            if(size - 1 > mT) mT = size - 1;
            for(int j = 0; j < loc; j++){
               cycle[listArray[j]] = 1+j-size;
               toCycle[listArray[j]] = cycleLength;
            }
         } else {//extension of tail
            listArray = list.toArray();
            size = list.length();
            baseTail = cycle[listArray[size-1]];
            cycleLength = toCycle[listArray[size-1]];
            if(size-1-baseTail > mT) mT = size-1-baseTail;
            for(int j = 0; j < size-1; j++) {
               cycle[listArray[j]] = j+1-size+baseTail;
               toCycle[listArray[j]] = cycleLength;
            }
         }
      } else {//new cycle found
```

```
            listArray = list.toArray();
            loc = list.find(next);
            size = list.length();
            cycleLength = size - loc - 1;
            if(cycleLength > mC) mC = cycleLength;
            if(loc > mT) mT = loc;
            for(int j = 0; j <= loc - 1; j++) {
               cycle[listArray[j]] = -loc+j;
               toCycle[listArray[j]] = cycleLength;
            }
            for(int j = loc; j < size - 1; j++)
               cycle[listArray[j]] = cycleLength;
         }
         list.destroyArray();
      }

      PR = isPrimRoot(base);
      terminal=0;
      for(int i = 1; i < n; i++)
         if(!image[i]) terminal++;
      maxTAll[base] = mT;
      maxCAll[base] = mC;
      terminalAll[base] = terminal;
      if(!PR) {
         maxTNone[base]=mT;
         maxCNone[base] = mC;
         terminalNone[base] = terminal;
         computeResults(base, cycle, toCycle, allCResults, allTResults,
            allToCycleResults, allNONECResults, allNONETResults,
            allNONEToCResults);

      } else {
         maxTPR[base] = mT;
         maxCPR[base] = mC;
         terminalPR[base] = terminal;

         computeResults(base, cycle, toCycle, allCResults, allTResults,
            allToCycleResults, allPRCResults, allPRTResults,
            allPRToCResults);

      }

   }
   status.open("status2.txt", fstream::app);
   status << "Writing Results...\n";
   status.close();
   writeTotalResults(allCResults, allTResults, allToCycleResults,
      allPRCResults, allPRTResults, allPRToCResults,
```

```
      allNONECResults, allNONETResults, allNONEToCResults,
      maxTAll,  maxTPR,  maxTNone,
      maxCAll,  maxCPR,  maxCNone,
      terminalAll,  terminalPR,  terminalNone);
   status.open("status2.txt", fstream::app);
   status << "Cleaning up...\n";
   status.close();
   delete [] visit;
   delete [] cycle;
   delete [] toCycle;
   delete [] allCResults;
   delete [] allTResults;
   delete [] allToCycleResults;
   delete [] allPRCResults;
   delete [] allNONECResults;
   delete [] allPRTResults;
   delete [] allNONETResults;
   delete [] allPRToCResults;
   delete [] allNONEToCResults;
   delete[] maxTAll;
   delete[] maxTPR;
   delete[] maxTNone;
   delete[] maxCAll;
   delete[] maxCPR;
   delete[] maxCNone;
   delete [] terminalAll;
   delete [] terminalPR;
   delete [] terminalNone;
}
```

E.4 Version 2

This version of the program will only examine the m-ary graphs for the given value of m and p. It will not generate any of the other graphs.

E.4.1 Version2.cpp

```
#include "LinkedList.h"
#include "ca2.h"
#include <exception>
#include <string>
#include<fstream>
#include "mpi.h"

using std::exception;
using std::cout;
```

```
using std::endl;
using std::string;


int main(int argc, char* argv[]) {

  MPI_Init(&argc, &argv);

  try {
      run();

  } catch (...) {
    //do nothing, just let MPI_Finalize() run
  }
  MPI_Finalize();

  return 0;
}
```

E.4.2 ca2.h

```
#ifndef CA2
#define CA2

#include "LinkedList.h"
#include <fstream>
#include <string>
#include<sstream>
#include<iostream>
#include<math.h>
#include<time.h>
#include "bn_prime.h"

using std::string;
using std::ofstream;
using std::ifstream;


#define STATUS "status_3_100057.txt"
#define n 100057
#define M_ARY 3

void run();
int m_exp(int, int, int);
long long ml_exp(long long, int, long long);
void computeResults(int, const int*, const int*, int*, int*, int* );
```

```
void writeResults(int, int*, int *, int*);
void setArrays(int*,bool*, int*, bool*);
bool isPrimRoot(int);
bool isRelPrime(int);
void writeTotalResults(const int*, const int*, const int*,
                       const int*, const int*, const int*);

int gcd(int, int);
bool isPrime(int);
bool MillerRabin(int, int, int, int);


template<class F1, class F2> void convert(F1 in, F2 out){
   std::stringstream ss;
   ss << in;
   ss>> out;
}

#endif
```

E.4.3 ca2.cpp

```
#include "ca2.h"


int m_exp(int b, int r, int num = n) {
   return (int) ml_exp((long long) b, r, (long long) num);
}

long long ml_exp(long long b, int r, long long num) {
   if (r == 0) return 1;
   if(r % 2 == 0) {
      long long result = ml_exp(b,r/2,num);
      return result * result % num;
   }
   long long result = ml_exp(b,r/2,num);
   return (b * result % num) * result % num;
}

void computeResults(int base, const int* cycle, const int* toCycle,
                    int* allCResults, int* allTResults,
                    int* allToCycleResults) {

   for(int i = 0; i < n; i++) {
      allToCycleResults[toCycle[i]]++;
      if(cycle[i] < 0) {
```

```
         allTResults[-1*cycle[i]]++;
      } else if (cycle[i] > 0) {
         allCResults[cycle[i]]++;
      }
   }
}


void writeResults(int base, int* cycleResults, int* tailResults,
                  int* toCycleResults){
   bool primRoot = isPrimRoot(base);
   bool relPrime = isRelPrime(base);
   char* file = new char[20];
   convert<int, char*>(n, file);
   string fileStr = file;
   if (primRoot && relPrime)
      fileStr+="PRRP";
   else if (primRoot)
      fileStr+="PR";
   else if (relPrime)
      fileStr+= "RP";
   else
      fileStr += "NONE";
   fileStr += ".txt";
   ifstream in(fileStr.c_str());
   int prevC, prevT, prevToC;
   for(int i = 1; i <= n; i++){
      in>>prevC;
      in.ignore(1,0);
      in>> prevT;
      in.ignore(1,0);
      in>> prevToC;
      in.ignore(1,0);
      cycleResults[i]+= prevC;
      tailResults[i]+= prevT;
      toCycleResults[i] += prevToC;
   }
   in.close();
   ofstream out(fileStr.c_str());
   for(int i = 1; i <= n; i++)
      out<<cycleResults[i] << " " <<tailResults[i]<< " "
         << toCycleResults[i] << '\n';
   out.close();
   delete file;
}


bool isPrimRoot(int base) {
```

```
   if(!isPrime(n)) return false;

   int n_1 = n-1;

   if ((unsigned)n_1 > (primes[NUMPRIMES-1]*primes[NUMPRIMES-1]))
      std::cerr<<"Error in Primitive Root Testing, n could "
               <<"have prime factor too large for testing."<<std::endl;
   int n1 = n_1;
   int index = 0;
   int p;
   while(n1 > 1 && index < NUMPRIMES) {
      if((n1 % primes[index]) == 0) {
         p = primes[index];
         while((n1 % primes[index] == 0)) n1/=primes[index];
         if(m_exp(base,n_1/p) == 1) return false;
         if(n1 == 50021) return (m_exp(base,n_1/50021) != 1);
      }
      index++;
   }
   return true;
}

bool isPrime(int num) {
   for(int i = 0; i < 50;i++) {
      if(primes[i] > (unsigned)num) return true;
      if((num % primes[i] == 0) && (num!=primes[i])) return false;
   }

   int k = 0;
   int q = num-1;
   while(q % 2 == 0) {
      k++;
      q >>= 1;
   }
   srand(time(0));
   int a;
   for(int i = 0; i < 10; i++) {
      a = (rand() % (num-2)) + 1;
      if(!MillerRabin(num, k, q, a)) return false;
   }

   return true;
}


bool MillerRabin(int num, int k, int q, int a) {
   int n1 = num-1;
   if(m_exp(a,q, num) == 1) return true;
```

```
   for(int i = 0; i < k; i++)
      if(m_exp(a,(int)pow(2,i)*q, num) == n1) return true;
   return false;
}


bool isRelPrime(int base) {
   return gcd(base, n-1) == 1;
}

int gcd(int a, int b) {
   if(a== 0) return b;
   if(b==0) return a;
   int r = a % b;
   int d = b;
   int c;
   while (r > 0) {
      c = d;
      d = r;
      r = c % d;
   }
   return d;
}

void setArrays(int * cycle, bool* visit, int* toCycle, bool* image){
   for(int i = 0; i < n; i++){
      visit[i] = false;
      cycle[i] = 0;
      toCycle[i] = 0;
      image[i] = false;
   }
}


void writeTotalResults(const int* allCResults, const int* allTResults,
                       const int* allToCycleResults,
       const int* maxTAll,
       const int* maxCAll,
       const int* terminalAll) {

   char* file = new char[20];
   convert<int, char*>(n, file);
   string fileStr = file;
   fileStr += "_";
   char* m_ary = new char[10];
   convert<int, char*>(M_ARY, m_ary);
   fileStr += m_ary;
   fileStr += ".dat";
```

```
   ofstream out(fileStr.c_str());
   for(int i = 1; i <= n; i++) {
      if(i == 1 || i == n)
         out<< allCResults[i]/i << " " << allTResults[i] << " "
            << allToCycleResults[i] << " " << 0 <<" " << 0 << " "
            << 0 <<'\n';
      else
         out<< allCResults[i]/i << " " << allTResults[i] << " "
            << allToCycleResults[i] << " " << terminalAll[i] << " "
            << maxCAll[i] << " " << maxTAll[i] << '\n';
   }
   out.close();
   delete file;
}


void run() {
   ofstream status(STATUS);
   status << "Allocating...\n";
   status.close();
   bool* visit = new bool[n];
   bool* image = new bool[n];
   int* maxTAll = new int[n];
   int* maxCAll = new int[n];
   int* terminalAll = new int[n];
   int *cycle= new int[n];
   int *toCycle = new int[n];
   int *allCResults = new int[n+1];
   int *allTResults = new int[n+1];
   int *allToCycleResults = new int[n+1];

   int next, size, loc, baseTail, cycleLength, mC, mT, terminal;
   int root, exp, base;
   LinkedList list;
   int* listArray;

   for(int i = 0; i <= n; i++){
      allCResults[i] = 0;
      allTResults[i] = 0;
      allToCycleResults[i] = 0;
      if(i < n) {
         maxTAll[i] = 0;
         maxCAll[i] = 0;
         terminalAll[i] = 0;
      }
   }
   status.open(STATUS, fstream::app);
   status << "Finding a PR...\n";
```

```
status.close();

for(root = 1; !isPrimRoot(root); root++);

status.open(STATUS, fstream::app);
status <<"Prim root is "<<root <<"...\n";
status.close();

for(exp = 0; exp < n; exp++) {
   if(exp % 100 == 0) {
      status.open(STATUS);
      status << "Exp is " << exp << std::endl;
      status.close();
   }
   if(gcd(exp,n-1) != M_ARY) continue;
   base = m_exp(root,exp);
   mC = 0;
   mT = 0;
   setArrays(cycle, visit, toCycle, image);

   for(int i = 1; i < n; i++) {
      list.makeEmpty();
      next = i;
      list.insert(next);
      while(!visit[next]){
         visit[next] = true;
         next = m_exp(base,next);
         image[next] = true;
         list.insert(next);
      }
      if(cycle[next] != 0) {
         if(cycle[next] > 0) {//all tail into cycle
            listArray = list.toArray();
            size = list.length();
            loc = list.find(next);
            cycleLength = cycle[listArray[size-1]];
            if(size - 1 > mT) mT = size - 1;
            for(int j = 0; j < loc; j++){
               cycle[listArray[j]] = 1+j-size;
               toCycle[listArray[j]] = cycleLength;
            }
         } else {//extension of tail
            listArray = list.toArray();
            size = list.length();
            baseTail = cycle[listArray[size-1]];
            cycleLength = toCycle[listArray[size-1]];
            if(size-1-baseTail > mT) mT = size-1-baseTail;
            for(int j = 0; j < size-1; j++) {
```

```
                cycle[listArray[j]] = j+1-size+baseTail;
                toCycle[listArray[j]] = cycleLength;
             }
          }
       } else {//new cycle found
          listArray = list.toArray();
          loc = list.find(next);
          size = list.length();
          cycleLength = size - loc - 1;
          if(cycleLength > mC) mC = cycleLength;
          if(loc > mT) mT = loc;
          for(int j = 0; j <= loc - 1; j++) {
             cycle[listArray[j]] = -loc+j;
             toCycle[listArray[j]] = cycleLength;
          }
          for(int j = loc; j < size - 1; j++)
             cycle[listArray[j]] = cycleLength;
       }
       list.destroyArray();
    }

    terminal=0;
    for(int i = 1; i < n; i++)
       if(!image[i]) terminal++;
    maxTAll[base] = mT;
    maxCAll[base] = mC;
    terminalAll[base] = terminal;
    computeResults(base, cycle, toCycle, allCResults, allTResults,
       allToCycleResults);

}
status.open(STATUS, fstream::app);
status << "Writing Results...\n";
status.close();
writeTotalResults(allCResults, allTResults, allToCycleResults,
      maxTAll, maxCAll, terminalAll);
status.open(STATUS, fstream::app);
status << "Cleaning up...\n";
status.close();
delete [] visit;
delete [] cycle;
delete [] toCycle;
delete [] allCResults;
delete [] allTResults;
delete [] allToCycleResults;
delete[] maxTAll;
delete[] maxCAll;
delete [] terminalAll;
```

```
   status.open(STATUS, fstream::app);
   status<<"Exiting...\n";
   status.close();
}
```

References

[1] R. Arratia and S. Tavaré. The cycle structure of random permutations. *Ann. Probab.*, 20(3):1567–1591, 1992.

[2] M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudorandom bits. *SIAM Journal on Computing*, 13(4):850–864, 1984.

[3] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[4] P. Flajolet and A. Odlyzko. Singularity analysis of generating functions. *SIAM J. Discrete Math.*, 3(2):216–240, 1990.

[5] P. Flajolet and A. M. Odlyzko. Random mapping statistics. In *Advances in cryptology—EUROCRYPT '89 (Houthalen, 1989)*, volume 434 of *Lecture Notes in Comput. Sci.*, pages 329–354. Springer, Berlin, 1990.

[6] R. Gennaro. An improved pseudo-random generator based on the discrete logarithm problem. *Journal of Cryptology*, 18(2):91–110, 2005.

[7] J. Holden. Fixed points and two-cycles of the discrete logarithm. In *Algorithmic number theory (Sydney, 2002)*, volume 2369 of *Lecture Notes in Comput. Sci.*, pages 405–415. Springer, Berlin, 2002.

[8] I. Niven, H. S. Zuckerman, and H. L. Montgomery. *An Introduction to the Theory of Numbers*. John Wiley & Sons, Inc, 1991.

[9] C. Pomerance. A tale of two sieves. *Notices Amer. Math. Soc.*, 43(12):1473–1485, 1996.

[10] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Comm. ACM*, 21(2):120–126, 1978.

[11] B. Schneier. *Applied Cryptography: Protocols, Algorithms and Source Code in C*. John Wiley & Sons, Inc, 2 edition, 1996.

[12] L. A. Shepp and S. P. Lloyd. Ordered cycle lengths in a random permutation. *Trans. Amer. Math. Soc.*, 121:340–357, 1966.