STUDIES ON GOPALA-HEMACHANDRA CODES AND THEIR APPLICATIONS.

by

Logan Childers

December, 2020

Director of Thesis: Krishnan Gopalakrishnan, PhD

Major Department: Computer Science

Gopala-Hemachandra codes are a variation of the Fibonacci universal code and have applications in data compression and cryptography. We study a specific parameterization of Gopala-Hemachandra codes and present several results pertaining to these codes. We show that $GH_a(n)$ always exists for $n \geq 1$ when $-2 \geq a \geq -4$, meaning that these are universal codes. We develop two new algorithms to determine whether a GH code exists for a given a and n, and to construct them if they exist. We also prove that when $a = -(4 + k)$ where $k \geq 1$, that there are at most k consecutive integers for which GH codes do not exist. In 2014, Nalli and Ozyilmaz proposed a stream cipher based on GH codes. We show that this cipher is insecure and provide experimental results on the performance of our program that cracks this cipher.

STUDIES ON GOPALA-HEMACHANDRA CODES AND THEIR APPLICATIONS.

A Thesis

Presented to The Faculty of the Department of Computer Science

East Carolina University

In Partial Fulfillment of the Requirements for the Degree

Master of Science in Computer Science

by

Logan Childers

December, 2020

STUDIES ON GOPALA-HEMACHANDRA CODES AND THEIR APPLICATIONS.

by

Logan Childers

APPROVED BY:

DIRECTOR OF THESIS:

Krishnan Gopalakrishnan, PhD

COMMITTEE MEMBER:

Venkat Gudivada, PhD

COMMITTEE MEMBER:

Karl Abrahamson, PhD

CHAIR OF THE DEPARTMENT OF COMPUTER SCIENCE:

Venkat Gudivada, PhD

DEAN OF THE GRADUATE SCHOOL:

Paul J. Gemperline, PhD

Table of Contents

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Data Compression, Universal Codes, and Fibonacci Codes

In computer science, data compression is the process of converting a given set of data into a compressed, smaller format. There are many reasons to compress data into smaller sized files, including reducing the storage space required for large files and decreasing the size of a message prior to sending it over a network [14]. In this way, the field of data compression is concerned with improving the efficiency of storing and communicating data. Different types of data, including text, images, and videos, require different types of algorithms to compress and decompress, due to the fact that data compression involves eliminating redundancies specific to a given data type. In this work, we will be concerned exclusively with text compression.

1.1 Fixed-Length Codes

If one wishes to send a message over a network, one must have a method by which they may encode each character in such a way that they can then be decoded correctly by the receiver. More precisely, let us have an alphabet Σ from which we take symbols to compose our message, then we must construct an encoding for each symbol which allows the receiver to then decode each codeword, such that they can retrieve the original message. ASCII encoding is one such scheme by which this may be done.

ASCII is the abbreviation for the American Standard Code for Information Interchange, and is a character encoding standard which assigns 128 characters to 7 bit binary represen-

tations between 0 and 127 [2]. ASCII characters include the capital and lowercase English alphabet, the digits 0 through 9, as well as other symbols such as parentheses and punctuation. Typically, a parity bit is added to the end of each encoding, such that each character utilizes a full byte for representation.

If the alphabet used by our message is a subset of the ASCII characters, then we may encode each character in our message using the provided encodings, resulting in an encoded message which is $8 * l$ bits long, where l is the length of the message. ASCII represents a fixed-length solution to the encoding/decoding of an alphabet, in that we are encoding all symbols to be the same length, so that the receiver may then decode the resulting codewords by taking a fixed amount of bits, in our case 8, and looking up the corresponding codeword. Fixed-length encodings thus provide us with one important feature for sending a message, and for the general encoding of a message: unique decodability.

Definition 1.1.1. *An encoding scheme for an alphabet Σ is uniquely decodable if, for any message m composed of symbols from Σ, the resulting encoded message may only be decoded in one way, that being into the original message m.*

Without this property, it cannot be guaranteed that an encoded message m will not be read out as an alternate message m', and thus any useful text-encoding scheme must be uniquely-decodable. However, ASCII codes do not consider the probability with which symbols will appear in a given message. ASCII, being a general encoding scheme, seeks only to provide encodings for a comprehensive alphabet. If we wanted to try to send our message using fewer bits, we could do better by creating a shorter collection of encodings tailored to that specific message. Further, we could attempt to provide encodings for symbols based on how often they appear in our message. As an example, consider the message "therefore." It would make sense to try to represent e with as short of a code as possible, while less common symbols like t and f can be allowed to have longer encodings. To this end, we can utilize variable-length encodings.

1.2 Huffman Coding

One type of useful variable-length code for text compression is a prefix code.

Definition 1.2.1. *An encoding scheme for an alphabet Σ is a prefix code if no codeword is a prefix of any other codeword. A message encoded with a prefix code is uniquely-decodable, and can be decoded from left to right by reading until you have a complete codeword.*

David Huffman developed an algorithm for producing minimum-length prefix codes for a finite set of symbols based on the probability with which they appear in a message [8]. That is, he developed an algorithm which allows one to produce an optimally compressed representation for some set of symbols, whereby more frequent symbols are assigned shorter encodings, and less frequent symbols are left with longer encodings.

The algorithm can be viewed as building a tree based on the probabilities associated with the symbols, or based on the number of occurrences of each symbol which imply probabilities. One begins to build this tree by considering all symbols to be their own separate tree, with each symbol being the only node in that tree. Next, the two lowest probabilities trees become children of a new node, where one of the children is assigned the value 0, and the other is assigned the value 1. The new tree has a probability equal to the sum of the two children's probabilities, and the combining step is repeated until one tree remains. The final tree is then able to yield an encoding for each symbol, which can be constructed by concatenating the bits at each node required to reach the symbol from the root of the tree. This algorithm bypasses the need for fixed-length encodings by producing a prefix-code over the symbols utilized.

Consider the following toy example. Let our alphabet be e, f, and k, with probabilities .5, .3, and .2 respectively. Considering each symbol as a separate tree, we combine the trees with symbols f and k because they have the lowest probabilities, assigning f to 0 and k to 1. Now, we have two trees, each with probability of .5: the tree which consists of only e as the root node, and the tree which has an empty root node and f and k as its children. We

can combine these last two trees by assigning the tree e to 0, and the tree with f and k to 1. Thus, our resulting code book encodes e to be 0, f to be 10, and k to be 11.

Huffman's algorithm produces optimal Huffman Codes over a given alphabet by greedily constructing the codebook for the message. This is illustrated in our example above, where symbols with lower frequencies appear further down in the final tree, and thus obtain larger encodings than the most frequent symbols, which are added to the tree towards the end and obtain shorter encodings. Unlike ASCII encoding, which is a fixed-encoding, the Huffman Code for a message depends on the codebook used to encode it, and so the codebook must be sent along with the message so that the receiver has the codebook to decode the message with, adding to the overhead of transmission.

Huffman Coding is not suitable for infinite sets of symbols due to the fact that an infinite alphabet would prevent the algorithm from terminating, and the storage and transmission of an infinite codebook would not be feasible. The solution to finding small representations for an infinite set of symbols comes in the form of universal codes.

1.3 Universal Codes

A universal code maps an infinite alphabet Σ onto an infinite set of encodings, such that the resulting scheme is uniquely-decodable [6, 14, 15]. Universal codes accomplish this by mapping a symbol x to a positive integer i, such that encoding all of the positive integers is equivalent to encoding an infinite alphabet of symbols which are indexed to the positive integers. Further, we assume that for two symbols indexed at i and j such that $i < j$, the symbol represented by i is more common in our message than the symbol represented by j. Hence, higher indices correspond to symbols with lesser probability of occurrence. This section covers two universal codes, Unary Coding and Elias Coding.

Universal codes may possess many other properties, one of which is robustness. A code can be considered to have some degree of robustness if it has some ability to limit the corruption of a message caused by introducing an error into the encoded version. Some

universal codes have little to no robustness, such that some errors can propagate indefinitely throughout the message, while some universal codes, such as the Fibonacci Code, can limit error propagation from a single error to a small number of codewords [9].

1.3.1 Unary Coding

Unary coding is an exceedingly simple universal coding convention by which an integer n is represented by n 1's followed by a 0, or equivalently, n 0's followed by a 1 [15]. As an example, 5 would be represented by the unary code 111110. This coding scheme is indeed a prefix code, and hence it is uniquely decodable, but it is exceedingly inefficient, since it requires $n + 1$ bits to encode n, versus the $\lfloor log_2(n) \rfloor + 1$ bits required for a direct binary encoding. We could not utilize direct binary encodings for text encoding, however, due to binary encodings not being uniquely decodable.

To give an example of how inefficient unary coding can be, consider n = 1000, whose binary encoding is 1111101000, and whose unary encoding is 1000 1's followed by a 0. The unary encoding is indeed 100 times larger than the binary encoding. It is, however, useful in constructing coding schemes which are substantially more efficient. One such scheme is Elias's gamma coding scheme, which is described in the next section.

1.3.2 Elias Gamma Coding

Peter Elias published three universal codes in his 1975 paper [6]. Here, we give Elias gamma coding as an example of a more efficient universal code.

Elias gamma coding encodes an integer based on it's binary representation. Let $\beta(n)$ be the binary representation of the integer n to be encoded, and let $\alpha(\beta(n))$ be the length of the binary representation of n. Then the gamma code of n, $\gamma(n)$, is $\alpha(\beta(n)) - 1$ 0's prepended onto $\beta(n)$. Let us take, as an example, $n = 25$. The binary encoding of 25 is 11001, which is 5 bits long, and so $\gamma(25) = 000011001$. In order to decode the code, one reads and counts the 0's from left to right until a 1 is encountered. Letting the number of zero's counted be

x, we then calculate n as $2^x + y$, where y is the next x bits read after the first 1 as an integer in binary form [14].

Compared to just using the binary form of an integer, which requires $\lfloor log_2(n) \rfloor + 1$ bits, Elias gamma coding requires $2\lfloor log_2(n) \rfloor + 1$ bits [14]. Unlike the binary representation of an integer, however, gamma coding produces a prefix code for the integers. This property is ensured by the prepending of 0's to the binary representation. If two encodings are the same length, they may not be a prefix of each other. Otherwise, if two encodings are of different length, it must be because the binary representations of the two integers which are being encoded have different lengths. This means that one code begins with more 0's than the other, meaning one may not be the prefix of the other.

This is only the basic code provided by Elias in his paper. In [6], he also produces a delta code, which invokes his gamma encoding scheme, and an omega code, which is self-contained and recursively calls itself. Rather than looking at these, we will next examine the Fibonacci Code, which is important in understanding its generalization, Gopala-Hemachandra Codes, the primary concern of this thesis.

1.4 Fibonacci Coding

Fibonacci Coding is a universal coding scheme first defined by Apostolico and Fraenkel [1], and involves encoding an integer using its representation using the Fibonacci Sequence. Therefore, we first discuss the Fibonacci Sequence its properties.

1.4.1 The Fibonacci Sequence

The Fibonacci sequence is a sequence of positive integers whose terms are defined by the recurrence relation $F[n] = F[n-1] + F[n-2]$ for all $n > 2$ with the initial conditions $F[1] = 1$ and $F[2] = 2$, where $F[n]$ indicates the n^{th} term of the Fibonacci Sequence. In order to generate the sequence from the two initial terms, we would first generate $F[3]$ as

follows:

$$F[3] = F[2] + F[1] = 2 + 1 = 3$$

and then $F[4]$ as:

$$F[4] = F[3] + F[2] = 3 + 2 = 5$$

The first few terms of the sequence are then:

$$1, 2, 3, 5, 8, 13, 21, 34, 55, ...$$

One property of the Fibonacci Sequence, which we use a generalization of later, is found in [11]. For the sake of completeness, we prove this property here.

Lemma 1.4.1. *Let r be an integer greater than or equal to 1. Then the sum of the first r terms of the Fibonacci Sequence is equal to $F[r+2] - 2$.*

$$\sum_{i=1}^{r} F[i] = F[r+2] - 2$$

Proof. We will prove the above identity by induction. As a base case, let $r = 1$, then

$$\sum_{i=1}^{1} F[i] = F[1] = 1 = 3 - 2 = F[1+2] - 2$$

and we have shown our base case.

Now, assume $\sum_{i=1}^{k} F[i] = F[k+2] - 2$, and let us try to prove $\sum_{i=1}^{k+1} F[i] = F[k+3] - 2$. Now we have:

$$\begin{aligned} \sum_{i=1}^{k+1} F[i] &= \sum_{i=1}^{k} F[i] + F[k+1] \\ &= F[k+2] - 2 + F[k+1] \\ &= F[k+3] - 2 \end{aligned}$$

Thus, we have proven $\sum_{i=1}^{r} F[i] = F[r+2] - 2$ by induction. □

Just as every positive integer n has a binary representation where n is equal to the sum of some powers of 2, every positive integer (as well as zero) may also be represented as the sum of some terms of the Fibonacci Sequence. Zeckendorf's Theorem [18], states that every positive integer can be represented uniquely as the sum of some set of non-consecutive Fibonacci numbers, and such a representation is called an integer's Zeckendorf representation. It turns out that the Zeckendorf representation can be constructed greedily.

Definition 1.4.2 (Greedy Method)**.** *The greedy construction of the Zeckendorf representation of an integer n involves taking the largest Fibonacci term up to and including n as part of the Zeckendorf representation, subtracting this term from n, and then iterating the process on the remainder until the remainder is 0. The terms taken by this process form the Zeckendorf representation.*

Lemma 1.4.3. *Let n be a positive integer, then the Zeckendorf representation of n may be constructed greedily.*

Proof. We will prove that the Zeckendorf representation of n may be constructed greedily by induction. As a base case, we have for free that 1, 2, and 3 are in the Fibonacci Sequence, and thus, we can represent them as follows:

$$\begin{aligned} 1: &\quad 1 \\ 2: &\quad 01 \\ 3: &\quad 001 \end{aligned}$$

Since these representations have no consecutive 1's and are constructable via the greedy method, we have shown our base case.

Next, we prove our induction step, which states that if all integers from 1 to $n-1$ have a Zeckendorf representation, then n also must have a Zeckendorf representation.

If $n = F[i]$ for some i, then we are done, as n is in the Fibonacci Sequence. We may then represent n by taking $F[i]$, setting the bits at all indices to 0 except i, which will be set to 1.

Otherwise, let us apply the greedy technique to the construction of the Zeckendorf representation for n by taking as part of the sum the largest Fibonacci term up to n. Let

$F[i] < n < F[i+1]$, and let us take $F[i]$ as part of the Zeckendorf representation. Then, examining the remainder $r = n - F[i]$.

$$\begin{aligned} r &= n - F[i] \\ &< F[i+1] - F[i] \\ &= F[i] - F[i] + F[i-1] \\ &= F[i-1] \end{aligned}$$

We have shown that if we are to take $F[i]$, the remainder to be encoded is less than $F[i-1]$, meaning that we will not take $F[i-1]$, and there will not be consecutive 1's in our representation at this point. Further, we know that $0 \leq r \leq n-1$, and so, by the induction hypothesis, we have a Zeckendorf representation for r. Then, the Zeckendorf representation of n is the representation of r, along with the bit at index i equal to 1, with all bits in-between set to zero. □

Now that we have shown the greedy technique to work, we will prove Zeckendorf's Theorem.

Theorem 1.4.4 (Zeckendorf's Theorem)**.** *Any positive integer n can be written uniquely as $\sum_{i=1}^{l} \alpha_i F[i]$, where $F[i]$ represents the i^{th} term of the Fibonacci Sequence, α_i is either 0 or 1, and $\alpha_l = 1$. Moreover, for any α_i and α_{i+1}, it is not the case that both are* 1*.*

Proof. In order to prove Zeckendorf's Theorem, we must show that any positive integer n has a Zeckendorf representation, and that that representation is unique. By Lemma 1.4.3, we know that for any n, the greedy method will produce the Zeckendorf Representation of n. Thus, all that remains is to prove uniqueness.

We now show that Zeckendorf representation for any integer is unique. We know that we can construct the representation of n greedily, so let us call the Zeckendorf representation constructed by the greedy method S. If T is indeed a distinct Zeckendorf representation, then at some point, it must choose a different term from the greedy method, which is to say that at some remainder r from the greedy method, it must refuse to take the largest term up to and including r. Let $F[i] \leq r < F[i+1]$ be the first remainder r at which we deviate from

the greedy method. If it does not use $F[i]$, then the greatest term we could use is $F[i-1]$, as $F[i+1]$ will give us a sum greater than r at this step, and the final sum will be greater than n. Let us then try to construct the largest number r' that we can form by taking $F[i-1]$ and not taking any two consecutive terms. Without loss of generality, assume 1 is the last index that we use.

$$\begin{aligned} r' &= F[i-1] + F[i-3] + ... + F[1] \\ &= F[i-2] + F[i-3] + F[i-4] + F[i-5] + ... + F[2] + 2 * F[1] \\ &= F[i] - 2 + F[1] \quad \text{By Lemma 1.4.1} \\ &= F[i] - 1 \\ &< F[i] \\ &\leq r \end{aligned}$$

Therefore, if we do not take the greedy step at every point, we cannot possibly construct a Zeckendorf representation for n. Therefore, we cannot deviate from the greedy method's choices in constructing the Zeckendorf representation of n, and hence the Zeckendorf representation of n is unique.

Since we have proven the existence and the uniqueness of the Zeckendorf representation for all positive integers, we have proven Zeckendorf's Theorem. □

Although attributed by name to Edouard Zeckendorf, the same property of the Fibonacci Sequence was published 20 years earlier by Cornelis Gerrit Lekkerkerker [10], in 1951.

1.4.2 The Fibonacci Code

Utilizing Zeckendorf's Theorem, and in particular the property that these representations possess no consecutive 1's, it is then possible to produce a Fibonacci universal code. Defined in [1], the Fibonacci Coding Scheme encodes any integer into a binary codeword by appending a 1 to the end of the Zeckendorf representation of n. That is, given a Zeckendorf representation $n = \sum_{i=1}^{l} \alpha_i F[i]$, the Fibonacci Code for n is $\alpha_1\alpha_2\alpha_3...\alpha_l 1$. Since the Zeckendorf representation does not use consecutive terms of the Fibonacci Sequence, the only time it is possible to see consecutive 1's in the Fibonacci code is at the end of the encoding,

meaning that the Fibonacci Coding Scheme is a variable length prefix code, and thus is uniquely decodable. Fibonacci encodings for small values of n can be found in Table 1.1. We use the notation $F(n)$ to refer to the Fibonacci Code of n, and some examples are that $F(1) = 11, F(5) = 00011$, and $F(25) = 10100011$.

In [7], the Fibonacci Coding Scheme is said to represent n using $\lfloor \log_\phi(\sqrt{5}n) + 1 \rfloor$ bits, where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio. The Fibonacci Code is slightly less efficient at compressing symbols than Elias Delta Coding, as an example, which uses $\lfloor \log_2(n) \rfloor + 2\lfloor \log_2(\lfloor \log_2(n) \rfloor + 1) + 1 \rfloor$ bits, but more efficient than Elias Gamma Coding. Fibonacci Coding has the interesting property that it is quite robust against errors introduced into an encoded message, significantly more than other coding schemes like Elias' codes, in which certain errors can propagate indefinitely, invalidating the decoding of the entire portion of the message which comes after the error. Single-bit errors introduced into an encoding include insertion of a bit into the encoding which does not belong, deletion of some bit within the encoding, and substitution of the value of one bit for another. A single bit error introduced into a message encoded with Fibonacci Codes may only propagate over at most 3 codewords. This property arises from the use of two consecutive 1's to terminate each code. If an error does not change one of these terminating 1's or the bit leading up to these 1's, then the error is entirely contained in one codeblock. Otherwise, if the error is introduced into the terminating area of some codeword, it has the ability to add false information to the next codeword, but it cannot corrupt more than three symbols, including the current one [7].

1.4.3 Higher Order Fibonacci Sequences

The Fibonacci Code as has been discussed so far may also be called the second order Fibonacci Code, because it is based on the second order Fibonacci Sequence. The Fibonacci Sequence is "second order" in that it is constructed from a second order recurrence relation: $F[n] = F[n-1] + F[n-2]$. In the same way, one may consider the general m^{th} order Fibonacci Sequence, which is given in [1]. The m^{th} order Fibonacci Sequence uses the

n	$F(n)$
1	11
2	011
3	0011
4	1011
5	00011
6	10011
7	01011
8	000011
9	100011
10	010011
11	001011
12	101011

Table 1.1: Fibonacci Encodings for small n

recurrence relation $F^m[n] = F^m[n-1] + F^m[n-2] + ... + F^m[n-m]$ for all $n > 1$ with the initial conditions $F^m[0] = F^m[1] = 1$ and $F^m[-1] = F^m[-2] = ... = F^m[-m+2] = 0$. Here, we have added a superscript, m, to indicate the order of the relation used to construct the Fibonacci Sequence.

As an example, we present the third order Fibonacci Sequence. The third order Fibonacci Sequence, which is constructed from the recurrence relation $F^3[n] = F^3[n-1] + F^3[n-2] + F^3[n-3]$ for all $n > 1$ with the initial conditions $F^3[-1] = 0$, $F^3[0] = 1$ and $F^3[1] = 1$. Therefore, the first few terms of the third order Fibonacci Sequence are

$$1, 2, 4, 7, 13, 24, 44, 81, ...$$

From [5, 9], we know that the concept of a Zeckendorf Representation generalizes to order 3. That is, we have for every positive integer n, that it can be written uniquely as $n = \sum_{i=1}^{l} \alpha_i F^3[i]$, where $F^3[i]$ represents the i^{th} term of the third order Fibonacci Sequence, α_i is either 0 or 1, and $\alpha_l = 1$. Further, we have that for any α_i, α_{i+1} and α_{i+2}, it is not the case that all three are 1.

The presence of the third order Zeckendorf Theorem allows one to consider the possibility

of a generalization from the second order Fibonacci Code to the third order Fibonacci Code. Given a Zeckendorf representation $n = \sum_{i=1}^{l} \alpha_i F^3[i]$, the third order Fibonacci Code for n can be defined as $\alpha_1\alpha_2\alpha_3...\alpha_l 11$. In the case of the third order Fibonacci Code, we append 2 1's onto the end of the Zeckendorf representation of n on the premise that 3 consecutive 1's should now only appear at the end of an encoding, courtesy of Zeckendorf's Theorem for the third order.

However, in [9], it is made apparent that this is not sufficient to produce a uniquely-decodable coding scheme. The problem is that while Zeckendorf's Theorem guarantees there shall be no instance of 3 consecutive 1's, it certainly allows for 2 consecutive 1's to appear anywhere in the code, particularly at the end of the representation. Attempting to decode a third order code in the same way one would decode a second order code could then possibly result in outputting a different message from what was encoded. Consider the following two codewords which encode 6 and 5 respectively :

01111 10111

The problem is that, were these to be concatenated together, these two codewords would be indistinguishable from the following two codewords:

0111 110111

which represent 2 and 10. This counterexample, of which there are many, shows that while the third order Fibonacci Code is universal in the sense that it too provides unique representations for all positive integers, it is not uniquely decodable. Additionally, it is easy to see that the problem generalizes to higher order Fibonacci Codes, since even in the presence of higher-order Zeckendorf Theorems, it is still possible for two or more 1's to appear prior to the terminating ones and thwart a second-order style decoding algorithm.

Chapter 2

Gopala-Hemachandra Codes

The Fibonacci Sequence is named for Leonardo of Pisa, also known by the name Fibonacci, who published the sequence in 1202. However, according to [16], the sequence in question was studied in India far prior to its publication by Fibonacci. Among those who studied the sequence were Gopala and Hemachandra, between the years 1135 and 1150. In addition to studying and setting out the rules for the formation of the sequence, they also studied variations of the Fibonacci Sequence. One such variation named for them is the Gopala-Hemachandra Sequence, which generalizes the Fibonacci Sequence by allowing the same recursive construction of the Fibonacci Sequence to be used with arbitrary starting terms. That is, one can define $\mathrm{GH}_{a,b}$ to be the sequence:

$$a, b, a+b, a+2b, 2a+3b, 3a+5b, ...$$

An interesting note about this sequence is that if one lets $a = b = 1$, or $a = 1, b = 2$, then one obtains the Fibonacci Sequence, with either two 1's or a single 1 at the beginning respectively.

In 2007, J.H. Thomas proposed a specific variation on the Gopala-Hemachandra Sequence which restricts the parameterization of the sequence to one variable [17]. The variation he proposed involves letting $b = 1 - a$ and letting $a \leq -2$. Otherwise, the sequence is formed

using the same recursive construction, which yields the following general sequence:

$$a, 1-a, 1, 2-a, 3-a, 5-2a, ...$$

As an example, one could choose $a = -2$ to yield the sequence:

$$-2, 3, 1, 4, 5, 9, ...$$

A few interesting properties that follow from the choice of the b parameter and the restriction on a are that 1 will always be included as the third term of the sequence, a itself is the only negative term in the sequence, and that the sequence is monotonically increasing beginning at 1, the third term.

J.H. Thomas extended the concept of a Zeckendorf representation to these variant Fibonacci Sequences in order to consider Gopala-Hemachandra Codes (GH) over these sequences, formed in a similar way to Fibonacci Codes. We denote $GH_a[n]$ to be the n^{th} term of the GH sequence parameterized by a. Further, we define $GH_a(n)$ to be a GH code for an integer n in the sequence parameterized by a. We then denote a Zeckendorf representation n to be: $n = \sum_{i=1}^{l} \alpha_i GH_a[i]$, where α_i is either 0 or 1, and $\alpha_l = 1$. Moreover, for any α_i and α_{i+1}, it is not the case that both are 1. Then, in the same way as for Fibonacci Codes, a Gopala-Hemachandra Code for an integer n is $\alpha_1\alpha_2\alpha_3...\alpha_l 1$, where α_i is the coefficient of $GH_a[i]$ from the above Zeckendorf representation.

J.H. Thomas observed that $GH_a(n)$ may not exist and may not be unique if it does exist, unlike Fibonacci Codes, which always exist and are unique. Some examples provided by [17] include that $GH_{-2}(3)$ can be written either as 011 or 100011, and that $GH_{-5}(5)$ does not have a representation. When $GH_a(n)$ does not exist, we say that $GH_a(n) = \text{N/A}$, meaning there is no applicable code for n under the parameter a.

As with the Fibonacci Sequence, there are sequences in the family GH_a which permit universal codes. Let a be a particular integer, then we refer to the GH_a code as universal

when all positive integers possess GH_a codes.

We present three major results on second order GH sequences as defined by J.H. Thomas: a new universality proof for GH_a when $-2 \geq a \geq -4$; two new algorithms for the determining the existence $GH_a(n)$ for any positive integer n, and constructing a code if it exists; and a general proof bounding the number of consecutive N/A codes for a given a.

2.1 Universality of GH_a for $-2 \geq a \geq -4$

In [17], Thomas presented tables showing codes for $GH_a(n)$ where $-2 \geq a \geq -5$ and $1 \leq n \leq 15$, when applicable. In [4], Basu and Prasad continued this experimental approach by publishing tables for $GH_a(n)$ where $-2 \geq a \geq -20$ and $1 \leq n \leq 100$, when they exist. In [3], Basu and Das present a proof of universality for GH_a when $-2 \geq a \geq -4$ which uses the technique of strong induction. We present their proof below.

Theorem 2.1.1. *$GH_a(n)$ always exists when $-2 \geq a \geq -4$.*

Proof. Adapted from [3]. We will do a proof by strong induction. Our base case will require that all positive integers be constructable up to the integer n we seek to construct. This allows us to start at $n = 101$ where the tables in [4] end, and use those tables as the base case.

Now, let us assume we have GH_a codes for 1 through $n-1$, and try to construct the $GH_a(n)$. If $n = GH_a[l]$ for some index l, then we are done, and $GH_a(n)$ is $l-1$ 0's followed by 11. Otherwise, we have that, for some l, $GH_a[l] < n < GH_a[l+1]$. We can then attempt to construct n by taking $GH_a[l]$ as part of our sum. Now, we must try to construct $n - GH_a[l]$. Since $n > GH_a[l]$, we have $n - GH_a[l] > 0$, and by our induction hypothesis, we have a code for $n - GH_a[l]$. Therefore, we can construct the code for n by taking the Zeckendorf Representation from the code for $n - GH_a[l]$ and setting the l^{th} index to 1, filling in 0's for all indices between l and the representation of $n - GH_a[l]$, and adding the ending 1 to complete the code. □

Unfortunately, this proof does not always construct the Zeckendorf representation for n. It does; however, construct a representation for n, which can then be turned into a Zeckendorf representation.

As an example of a bad GH code constructed by this method, we take $n = 135$ and $a = -4$. First, we construct the first few terms of GH_{-4}:

$$-4, 5, 1, 6, 7, 13, 20, 33, 53, 86, 139$$

We certainly have that 135 is not in the GH_{-4} sequence, so we start by taking 86 as the first element of the sum. Next, we subtract 86 from 135, and obtain 49. 49 is well within the base case provided by [4], and the GH code for 49 is 1000000011. Thus, we are done, and we construct our final code by taking the Zeckendorf Representation of 49 setting the index of 86 (10) to be 1, and adding the ending 1. This results in the code 10000000111, which is not a Zeckendorf representation, due to the fact that it features two consecutive 1's before the end of the code.

We now prove that if we have some representation of n which is not a Zeckendorf representation, it can be converted into a Zeckendorf representation using the following lemma:

Lemma 2.1.2. *Let n be representable as the sum of some terms of the GH_a sequence. In other words, let $n = \sum_{i=1}^{l} \alpha_i GH_a[i]$, where α_i is either 0 or 1, and α_l is 1. Then there exists a Zeckendorf representation of n. In other words, there exists $n = \sum_{i=1}^{k} \beta_i GH_a[i]$, where β_i is either 0 or 1, β_k is 1, and for any β_i and β_{i+1}, at most one of them is 1.*

Proof. Let n be represented by the string $\alpha_1\alpha_2\alpha_3...\alpha_l$. If the string is not already a Zeckendorf representation of n, then there must be two consecutive 1's within the string. Scan from right to left until you find the first incidence of the substring 11, and replace the substring 110 with 001. This preserves the value of the sum represented by the string, since $GH_a[i] = GH_a[i-1] + GH_a[i-2]$, and we are replacing two consecutive terms with the term which they sum to. Repeat the process as long as two consecutive 1's may be found in the string.

n	$a=-2$	$a=-3$	$a=-4$
0	00000	00000	00000
1	00100	00100	00100
2	10010	10010	10010
3	10001	10001	10001
4	10101	10101	10101
5	00001	00010	01000
6	00101	00001	00010
7	01010	00101	00001
8	01001	10011	00101
9	-	01010	10011
10	-	01001	10111
11	-	-	01010
12	-	-	01001

Table 2.1: Representations for $0 \leq n \leq GH_a[6]$ for $-2 \geq a \geq -4$.

Once there are no longer any consecutive 1's, the new string $\beta_1\beta_2\beta_3...\beta_k$ will be a Zeckendorf representation of n. We will have $k = l + 1$ if $\alpha_l = \alpha_{l-1} = 1$, such that the last two indices of the representation must be replaced. Otherwise, we have $k = l$. □

By Lemma 2.1.2, it is sufficient that every integer n have a representation for there to be a Zeckendorf representation of n. Using this Lemma, we can repair our example of a bad GH code. Our bad code for 135 in GH_{-4} was 10000000111, with a representation of 1000000011 when the terminating 1 is removed. Lemma 2.1.2 can be applied by converting the two consecutive 1's in this representation into 0's, and then placing a 1 after them, to yield the Zeckendorf representation 10000000001, and the GH code 100000000011. In addition to repairing this proof, we also present a new proof of Theorem 2.1.1, which is a constructive proof.

Alternative Proof of Theorem 2.1.1. We present explicit codes for the integers $0 \leq n < GH[6]$ for $-2 \geq a \geq 4$ in Table 2.1. We use these concrete codes at the end of our general construction.

Now, let n be the integer one wishes to encode. If it is between 0 and $GH_a[6]$, we are done, as we have those codes prewritten. If $n = GH_a[l]$ for some integer l, then we are done, and the

code is $i-1$ 0's followed by 11. Otherwise we have that, for some l, $GH_a[l] < n < GH_a(l+1)$. We can then attempt to construct n by taking $GH_a[l]$ as part of our sum. Now, we must try to construct $n - GH_a[l]$, which we will call n' now. If $n' < GH_a[6]$, then we have the encoding of our current remainder, and we can construct the code using the representation of the remainder and the terms we took so far. Otherwise, $n' \geq GH_a[6]$, and we may again find the greatest term up to n' and include it in our sum. We repeat the process until, eventually, our remainder is less than $GH_a[6]$, and we have an encoding for the remainder. We may then construct a representation by setting all the bits corresponding to the terms selected to 1, including those of the remainder's representation. If this is not a Zeckendorf Representation, we may apply Lemma 2.1.2 in order to make it a Zeckendorf representation, and then add an additional 1 at the end to obtain the $GH_a(n)$. □

2.2 Algorithms to construct GH Codes

We know from [4, 17] that there are integers for which GH_a codes do not exist where $-5 \geq a \geq -20$. We take a moment to prove a generalization of this observation stated in the following theorem.

Theorem 2.2.1. *If $a \leq -5$, then GH_a codes do not exist for all integers.*

Proof. We can prove that no GH_a for $a \leq -5$ is universal by finding at least one integer for each a that does not possess an encoding, and so, we will show that $n = 5$ may never be encoded when $a \leq -5$.

Consider the general form of the GH sequence:

$$a,\ 1-a,\ 1,\ 2-a,\ 3-a,\ 5-2a, ...$$

and let us rule out ways which will not allow us to add terms to equal 5. Firstly, we may not use any terms greater than or equal to $GH_a[6]$, which is $5-2a$. If $a \leq -5$, then any term at or beyond $5-2a$ will be greater than or equal to 15, and thus greater than 5. Second,

we may not use only a single term from the sequence. If we use just 1 or just a, the sum will be less than 5, and if we use $1-a, 2-a$, or $3-a$, then the sum must be greater than 5 if $a \leq -5$. Finally, we cannot use two or more terms which feature $-a$ in them, or else we encounter the same problem as using a term at or beyond $GH_a[6]$.

This leaves us to try to construct 5 using exactly one term from the set $\{1-a, 2-a, 3-a\}$, and at least one term from the set $\{a, 1\}$. If we do not include a, the resulting sum will be greater than or equal to $2-a$, whose smallest possible value is 7 at $a=-5$. If we only use a, then we are only able to construct $1, 2$, and 3. If we use both 1 and a and then we can only construct $2, 3$, and 4

Since the only integers we can construct when $a \leq -5$ are either less than or greater than 5, it is not possible to construct 5 from GH_a when $a \leq -5$. □

Since we know that for no $a \leq -5$, GH_a forms a universal code, the natural next question is when does a code exist for an integer n, and if it does exist, how do we find one. In [13], Pal and Das provide an algorithm for constructing $GH_a(n)$, when it exists, for an arbitrary a and n, which is shown here as Algorithm 1. Their algorithm works by trying to reduce the construction of a GH_a code for n down to construction of $GH_a(k)$ for a smaller integer k while accumulating a list of indices for additional terms to add to the representation of $GH_a(k)$.

Their algorithm, like the universality proof from [3], produces some incorrect results in that they are not necessarily GH codes, but are representations of the n that they tried to encode. As an example. we attempt to construct $GH_{-6}(649)$ using their algorithm. The GH sequence for $a=-6$ is shown below.

$$-6, 7, 1, 8, 9, 17, 26, 43, 69, 112, 181, 293, 474, 767, ...$$

To begin, we have $r = 1$ and we have the greatest GH term up to 649 is 474. Then, $m - m_r = 649 - 474 = 175$. The algorithm then checks whether $175 > 38 - 13 * -6$,

or $175 > 116$. The condition is true, so the algorithm takes 474 as part of the sum, and then tries to construct 175. The greatest GH term up to 175 is 112, and so we compute $m - m_r = 175 - 112 = 63$. $63 > 116$ is not true, and so the algorithm would like us to try to find the code which corresponds to 63 from the tables, which is 1000000011. We can then construct the code as 10000000110011. Indeed, the final "code" has two consecutive 1's and is thus not a GH code. It can however, be turned into a GH code through the use of Lemma 2.1.2.

In addition to not necessarily producing GH codes as outputs, the authors did not present a proof of correctness of their algorithm. They also presented the algorithm in an unclear manner which makes it difficult to implement. We provide two new algorithms which construct $GH_a(n)$ if it exists, and we then prove the correctness of these two algorithms. We first present several theorems and lemmas that aid in proving the correctness of our algorithms.

Lemma 2.2.2. *Let r be an integer greater than or equal to 2. Then $GH_a[r+2]-1$ is equal to the sum of the terms from $GH_a[2]$ to $GH_a[r]$.*

$$\sum_{i=2}^{r} GH_a[i] = GH_a[r+2] - 1$$

Proof. We will prove this lemma by induction. Firstly, recall that the general form of the sequence GH_a is:

$$a,\ 1-a,\ 1,\ 2-a,\ 3-a,\ 5-2a, ...$$

As a base case, let $r = 2$. Then, we have

$$\sum_{i=2}^{r} GH_a[i] = GH_a[2] = 1 - a$$

$$GH_a[2+2] - 1 = 2 - a - 1 = 1 - a$$

Thus, we have proven that our lemma is true for $r = 2$. For our induction step, assume the

Figure 2.1: Finding $GH_a(n)$ from [13]

```
Let r = 1, m = n;
if m ∈ GH_a then
    Write the representation with the index of m to be 1, all other indices 0, and add
     the ending 1, then stop.
end
if m − a ∈ GH_a then
    Write the representation with the index of m and a to be 1, all other indices 0,
     and add the ending 1, then stop.
end
if m − a − 1 ∈ GH_a then
    Write the representation with the index of m, 1 and a to be 1, all other indices 0,
     and add the ending 1, then stop.
end
while true do
    Let m_r = GH_a[i_r] be the greatest term in GH_a less than m;
    if m − m_r > 38 − 13a then
        m = m − m_r  r = r + 1
    else
        if m − m_r satisfies a straight line property from [13] then
            Lookup the codeword associated with the applicable straight line
             property, delete the ending 1, and then fill in 1's in all positions i_r
             chosen by the algorithm, zeros in all other positions, then add the
             ending 1, and stop.
        else
            Let m_{r+1} = GH_a[i_{r+1}] be the greatest term in GH_a less than or equal to
             m_r ;
            if m − m_{r+1} ≥ m_{r+1} then
                The code does not exist for GH_a(n), exit.
            else
                r = r + 1
            end
        end
    end
end
```

lemma is true for some k. Then, we have:

$$\sum_{i=2}^{k} GH_a[i] = GH_a[k+2] - 1$$

. We now need to prove the statement in the lemma is true for $k+1$, or

$$\sum_{i=2}^{k+1} GH_a[i] = GH_a[k+3] - 1$$

Starting from the left-hand side:

$$\begin{aligned}\sum_{i=2}^{k+1} GH_a[i] &= GH_a[k+1] + \sum_{i=2}^{k} GH_a[i] \\ &= GH_a[k+1] + GH_a[k+2] - 1 \\ &= GH_a[k+3] - 1\end{aligned}$$

Since we have proven both the base case and the induction step, we have proven our theorem by induction. □

Next, we present a theorem concerning the composition of representations which forms the basis for our algorithms.

Theorem 2.2.3. *Let n be a positive integer. If n can be realized as the sum of some terms of the sequence GH_a, then there exist integers n_0 and n_1 which satisfy the following conditions:*

1. $n = n_0 + n_1$
2. $n_0 = \sum_{i=1}^{5} \alpha_i GH_a[i]$ *where α_i are either 0 or 1 and* $0 \leq n_0 < GH_a[6]$
3. $n_1 = \sum_{i=6}^{k} \alpha_i GH_a[i]$ *where α_i are either 0 or 1 and forms a Zeckendorf representation of n_1 which can be constructed greedily.*

Proof. Assume that n can be realized as some sum of terms from the sequence GH_a such that $\sum_{i=1}^{k} \beta_i GH_a[i]$, and let $n_0' = \sum_{i=1}^{5} \beta_i GH_a[i]$ and $n_1' = \sum_{i=6}^{k} \beta_i GH_a[i]$, and let us first focus on n_0'.

Consider the first five terms of the general GH sequence:

$$a,\ 1-a,\ 1,\ 2-a,\ 3-a$$

If $0 \leq n_0' < GH_a[6]$, then we can set $\alpha_i = \beta_i$ for $1 \leq i \leq 5$, and we have a valid n_0.

If $n_0' < 0$, then it must be the case that $\beta_1 = 1$, $\beta_2 = \beta_4 = \beta_5 = 0$, and β_3 is either 0 or 1. We remedy this issue by using the recurrence relation $GH_a[n] = GH_a[n-1] + GH_a[n-2]$ to move a portion of the sum from n_1' to n_0'. If $n_0' < 0$, then $n_1' > 0$, since n is a positive integer. Since $n_1' > 0$, there must exist some smallest index j such that $\beta_j = 1$. Using the recurrence relation $GH_a[n] = GH_a[n-1] + GH_a[n-2]$, we can change the bit β_j to be zero, and set $\beta_{j-1} = \beta_{j-2} = 1$. We know β_{j-1} and β_{j-2} were zero to begin with, since β_j was the first 1 in n_1'. Since the last two indices of n_0' are guaranteed to be 0 by the fact that $n_0' < 0$, this method can then be repeated until the bits which are replaced fall within n_0'. The final replacement will set $\beta_5 = 1$, and β_4 equal to either 0 or 1. Since β_5 is now 1 in n_0', it is no longer possible for n_0' to be negative, and we have preserved the sum $n = n_0' + n_1'$ by usage of the GH recurrence relation. Thus, we may now set $\alpha_i = \beta_i$ for $1 \leq i \leq 5$, and we have a valid n_0.

If $n_0' \geq GH_a[6]$, then $\beta_4 = \beta_5 = 1$, or n_0' is represented by 01101, in which case the GH recurrence relation can be used to convert the representation such that $\beta_4 = \beta_5 = 1$. Therefore, let us assume that $\beta_4 = \beta_5 = 1$. Since we have two consecutive 1's, we can remove at least β_5 and from n_0' using Lemma 2.1.2, depending on where the first 0 index lies in n_1'. Once we have removed β_5 from n_0', the new β terms will represent a sum less than $GH_a[6]$, and we may now set $\alpha_i = \beta_i$ for $1 \leq i \leq 5$, and we have a valid n_0. This takes care of all the possibilities for which n_0' could deviate from our criteria.

Now, we must consider n_1, for which our main requirement is that it constitutes a Zeckendorf representation which may be constructed greedily. In order to do this, we set $n_1 = n - n_0$, so that condition 1 is satisfied. We know that n_1 must be the sum of some

terms of the remaining segment of the GH sequence, and we know that if we have some representation of n_1 which is not a Zeckendorf representation, we may use Lemma 2.1.2 to turn it into a Zeckendorf representation. With this in mind, we will show that this representation may be found using the Greedy Approach.

When we refer to constructing a code for n_1 greedily, we refer to the process of taking the largest GH term up to n_1 as being part of the sum, subtract that term from n_1, and then iterating the process on the difference until we have all of the terms of the code. Formally, let r be the current remainder we are trying to construct, and let $GH_a[m] \leq r < GH_a[m+1]$, then we must choose $GH_a[m]$.

We prove by contradiction that the Zeckendorf representation must be constructed greedily for n_1. Assume that the sum is not being constructed greedily, meaning that there is some first r where $GH_a[m] \leq r < GH_a[m+1]$, and we do not choose $GH_a[m]$. One cannot choose $GH_a[m+1]$ or greater terms because the resulting sum would be greater than the remainder which we are trying to construct. Therefore, the largest term we may then choose for a non-greedy step is $GH_a[m-1]$. Now, consider the largest Zeckendorf representation r' that we may form by taking this non-greedy step:

$$\begin{aligned}
r' &= GH_a[m-1] + GH_a[m-3] + ... + GH_a[6] \\
&= GH_a[m-2] + GH_a[m-3] + GH_a[m-4] + GH_a[m-5] + ... + GH_a[5] + GH_a[4] \\
&= \sum_{i=4}^{m-2} GH_a[i] \\
&< \sum_{i=2}^{m-2} GH_a[i] \\
&= GH_a[m] - 1 \qquad \text{By Lemma 2.2.2} \\
&< GH_a[m] \\
&\leq r
\end{aligned}$$

Thus, we have shown that if at all one were to take a non-greedy step to try to construct n_1, it will no longer be possible to add up terms to the remainder at that step while still forming a Zeckendorf representation. Thus, the greedy construction is the only method which will terminate correctly, and thus construct the representation for n_1. □

Now, we present our simple algorithm, which is a consequence of Theorem 2.2.3. The idea is as follows: for a given a, there are a maximum of 13 constructable n_0's. If an integer is encodable, then its encoding must utilize one of these n_0. Therefore, we can construct a GH code for an integer n by testing different possibilities for n_0 until we either find one that permits a valid n_1, or until we run out of possibilities for n_0 which we can try, from which we can conclude that there is no encoding of n. The possible options for n_0 are listed in Table 2.2 for $-2 \geq a \geq -4$, and in Table 2.3 for $a \leq -5$.

Figure 2.2: Simple Algorithm for Finding $GH_a(n)$

```
for n_0 := 0 to GH_a[6] − 1 do
    if n_0 can be represented using GH_a[1] through GH_a[5] then
        n_1 = n − n_0;
        if n_1 can be constructed greedily using GH_a[6] and up then
            GH_a(n) exists;
            Concatenate the representations of n_0 and n_1;
            Use Lemma 2.1.2 to ensure the result is a Zeckendorf Representation;
            Add the ending 1, print the code, and stop;
        end
    end
end
GH_a(n) is N/A
```

The correctness of Algorithm 2 follows from Theorem 2.2.3. If at all there is a representation of n in GH_a, then there is one which must use one of the n_0 representations available for a, and we can construct the corresponding n_1 greedily. Otherwise, if there is no representation which uses a valid n_0 for a, then there can be no representation of n in GH_a. Regarding complexity, the actual work of the algorithm is in the greedy attempts to construct n_1, which is a linear-time operation in the length of the Zeckendorf representation of n. Additionally, the invocation of Lemma 2.1.2 is a linear time operation. The loop runs a constant amount of times, and so Algorithm 2 is linear in the length of the Zeckendorf representation of n.

Next, we provide a more efficient algorithm which exploits characteristics of non-greedy codes in order to prevent us from checking up to 13 cases. Algorithm 3 relies on the following

n	$a=-2$	$a=-3$	$a=-4$
0	00000	00000	00000
1	00100	00100	00100
2	10010	10010	10010
3	10001	10001	10001
4	10101	10101	10101
5	00001	00010	01000
6	00101	00001	00010
7	01010	00101	00001
8	01001	10011	00101
9	-	01010	10011
10	-	01001	10111
11	-	-	01010
12	-	-	01001

Table 2.2: Representations for $0 \leq n_0 < GH_a[6]$ for $-2 \geq a \geq -4$.

n	$a=-(4+k)$
0	00000
1	00100
2	10010
3	10001
4	10101
$k+5$	01000
$k+6$	00010
$k+7$	00001
$k+8$	00101
$k+9$	10011
$k+10$	10111
$2k+11$	01010
$2k+12$	01001

Table 2.3: Representations for $0 \leq n_0 < GH_a[6]$ for $a=-(4+k)$ where $k \geq 1$

lemmas.

Lemma 2.2.4. *Let n be a positive integer. If n has an encoding which is produced by a non-greedy method, then the second bit of the encoding must be 1.*

Proof. Let n be a positive integer whose encoding is not produced by the greedy method. Recall that the greedy method works by picking the largest term which may be included in the sum, continuing this for the remainder until we reach a known encoding. If n is not produced greedily, then at some point it must ignore the greedy step and choose an alternative next term. Let r be the remainder at which we deviate from the greedy algorithm.

We now prove by contradiction that taking the non-greedy step requires us to include $GH_a[2]$ in our sum. If $GH_a[m] \leq r < GH_a[m+1]$, taking $GH_a[m]$ for the sum would be the greedy step. We cannot take a term greater than $GH_a[m]$, as this would create a sum greater than our remainder. Therefore, the largest term we may take for our sum is $GH_a[m-1]$. Let r' be the largest possible Zeckendorf representation that we can create by not taking $GH_a[2]$ and $GH_a[m]$. We have two possible cases for our ending term, depending on the parity of m. Without loss of generality, we assume the ending term is $GH_a[4]$.

$$\begin{aligned} r' &= GH_a[m-1] + GH_a[m-3] + ... + GH_a[4] \\ &= GH_a[m-2] + GH_a[m-3] + GH_a[m-4] + GH_a[m-5] + ... + GH_a[3] + GH_a[2] \\ &= \sum_{i=2}^{m-2} GH_a[i] \\ &= GH_a[m] - 1 \qquad \text{By Lemma 2.2.2} \\ &< GH_a[m] \\ &\leq r \end{aligned}$$

Therefore, if we are not allowed to include $GH_a[2]$ in our non-greedy construction, it is not possible to construct n. □

Lemma 2.2.5. *Let n be a positive integer. If n has an encoding which is produced by a non-greedy method, then the fourth bit of the encoding must be 1.*

Proof. Let n be a positive integer whose encoding is not produced by the greedy method. If

n is not produced greedily, then at some point it must ignore the greedy step and choose an alternative next term.

We now prove by contradiction that taking the non-greedy step requires us to include $GH_a[4]$ in our sum. Let r be the remainder at which we deviate from the greedy algorithm. If $GH_a[m] \leq r < GH_a[m+1]$, taking $GH_a[m]$ for the sum would be the greedy step. We cannot take a term greater than $GH_a[m]$, as this would create a sum greater than our remainder. Therefore, the largest term we may take for our sum is $GH_a[m-1]$. Let us now try to construct the largest possible Zeckendorf representation for r' that we can create by not taking $GH_a[4]$ and $GH_a[m]$. By Lemma 2.2.4, we must include $GH_a[2]$ in our sum, and since we are working with a Zeckendorf representation, we may not use $GH_a[1]$ or $GH_a[3]$. We have two possible cases for our ending term, depending on the parity of m. Without loss of generality, we assume the ending term is $GH_a[5]$.

$$\begin{aligned}
r' &= GH_a[m-1] + GH_a[m-3] + ... + GH_a[5] + GH_a[2] \\
&= GH_a[m-2] + GH_a[m-3] + ... + GH_a[4] + GH_a[3] + GH_a[2] \\
&= \sum_{i=2}^{m-2} GH_a[i] \\
&= GH_a[m] - 1 \qquad \text{By Lemma 2.2.2} \\
&< GH_a[m] \\
&\leq r
\end{aligned}$$

Therefore, if we are not allowed to include $GH_a[4]$ in our non-greedy construction, it is not possible to construct n. □

Lemma 2.2.4 and Lemma 2.2.5 together then claim that $GH_a[2]$ and $GH_a[4]$ must both be utilized if the code for n is constructed non-greedily. However, if we are creating a Zeckendorf representation, then we are not allowed to use $GH_a[1]$, $GH_a[3]$ or $GH_a[5]$. Thus, n_0 must be 01010 if the code we are constructing requires us to deviate from the greedy method. This gives rise to the principal of our second algorithm. We can assume that the encoding for n can be constructed by using the greedy method until the remainder $n' \leq GH_a[6]$, at which point we can then lookup whether the remainder has an encoding n_0. If it does not, we can

conclude that the encoding for n may not be constructed by the greedy method, and we can then try to construct it using $n_0 = 01010$ as the only n_0 for Algorithm 2. In other words, we set $n_0 = 01010$, compute $n_1 = n - n_0$, and then construct n_1 greedily. If this method does not work, then we can conclude that the encoding does not exist. The resulting algorithm is outlined as Algorithm 3.

Figure 2.3: More Efficient Algorithm for Finding $GH_a(n)$

```
Attempt to construct n greedily using terms GH_a[6] and up;
Let n_1 be the sum of the numbers picked by the greedy attempt;
n_0 = n − n_1;
if n_0 has an encoding then
    GH_a(n) exists;
    Concatenate the representations of n_0 and n_1;
    Use Lemma 2.1.2 to ensure the result is a Zeckendorf Representation;
    Add the ending 1, print the code, and stop;
else
    n_0 = GH_a[2] + GH_a[4];
    n_1 = n − n_0;
    if n_1 can be constructed greedily using GH_a[6] and up then
        GH_a(n) exists;
        Concatenate 01010 and the encoding for n_1;
        Use Lemma 2.1.2 to ensure the result is a Zeckendorf Representation;
        Add the ending 1, print the code, and stop;
    else
        GH_a(n) is N/A
    end
end
```

The correctness of Algorithm 3 follows from Lemma 2.2.4 and Lemma 2.2.5. If a code can be constructed greedily from n, then we simply attempt to construct it greedily until the remainder is less than $GH_a[6]$, at which point we can check whether the remainder n_0 is encodable. Otherwise, by Lemma 2.2.4 and Lemma 2.2.5, a constructable code which is not greedy-constructable must possess $n_0 = 01010$, and so it can either be constructed in this way, or not at all.

Both Algorithm 2 and 3 run in the asymptotically linear time in the length of the Zeckendorf representation of n, but in practice, Algorithm 2 may take up to 13 attempts to

construct a code, while Algorithm 3 makes only 2 attempts to construct a code, making it significantly more efficient.

2.3 Non-existence of GH codes for consecutive integers

In [4], Basu and Prasad produced tables of encodings for $GH_a(n)$, where $-2 \geq a \geq -20$, and $1 \leq n \leq 100$. They observed that for $a = -(4 + k)$, there were a maximum of k consecutive integers which could not be encoded for GH_a. That is, the largest block of integers whose encodings were N/A was of length k. We prove that, for general a and for unlimited values of n, there are a maximum of k consecutive integers that cannot be encoded for GH_a, where $a = -(4+k)$ and $k \geq 1$. We first prove a lemma which connects the existence of N/A encodings greater than $GH_a[6]$ to N/A encodings less than $GH_a[6]$.

Lemma 2.3.1. *If an integer $n > GH_a[6]$ does not possess a code, then there is some integer less than $GH_a[6]$ which does not possess a GH code.*

Proof. Assume some integer $n > GH_a[6]$ is N/A, meaning it does not possess a GH code, and attempt to construct it greedily by taking the largest term up to the n as part of the sum, subtracting it from n, and repeating this process continually on the difference n'. Eventually, the remainder, which we can call r, must be less than $GH_a[6]$. If $GH_a(r)$ is not N/A, then it has some valid encoding such that, combined with the terms chosen by the greedy method, would represent n. Since this would be a contradiction, we can conclude that $GH_a(r)$ does not have an encoding, and is thus N/A. Therefore, if $n > GH_a[6]$ does not have an encoding, there is also an integer $r < GH_a[6]$ which also does not have an encoding. □

Theorem 2.3.2. *Let k be a positive integer. Then, there exists at most k consecutive integers for which $GH_{-(4+k)}$ codes are N/A.*

Proof. Our proof relies on where encodings exist when $n < GH_a[6]$. Recall that, according to Table 2.3, we can only encode the integers 0 through 4, $k + 5$ through $k + 10$, $2k + 11$ and $2k + 12$, for $a \leq -5$, where $a = -(4 + k)$. Therefore, the integers from 5 to $k + 4$, and

$k + 11$ to $2k + 10$, have no encodings. Note how each of these ranges contains exactly k integers which we are unable to encode. Also, recall that if an integer n does not possess an encoding, there is some integer less than $GH_a[6]$ which also does not have an encoding by Lemma 2.3.1.

Consider two consecutive integers n and n' which do not have encodings. Let i be the index such that $GH_a[i]$ is the greatest GH term up to n, then $GH_a[i] ¡ n ¡ GH_a[i+1]$. This must also be the case for n', as it is either $n+1$ or $n-1$, and n' must not have an encoding. Now, attempt to construct n and n' greedily using terms greater than or equal to $GH_a[6]$. At this first step, both must choose $GH_a[i]$, since this is the greatest term less than both n and n'. Since both constructions choose the same step, the remainders will differ only by 1, and must also fall within some range $GH_a[i'] ¡ r, r' ¡ GH_a[i'+1]$, where r and r' are the current remainders of n and n' respectively. Therefore, both greedy constructions must take all of the same steps until both remainders fall between 0 and $GH_a[6]$. By Lemma 2.3.1, we know both n and n' rely on some n_0 less than $GH_a[6]$ not being encodable, and since the same steps were taken in both constructions, the value of the n_0's created by both constructions only differ by 1. Therefore, for these two consecutive integers to not be encodable, there must be two consecutive n_0's greater than 0 and less than $GH_a[6]$ which are not encodable.

We can argue in the same way that if $k + 1$ consecutive integers lack encodings, then there must be $k + 1$ consecutive integers less than $GH_a[6]$ which do not possess encodings. However, according to our analysis of Table 2.3 from above, there are only k consecutive integers less than $GH_a[6]$ which are not encodable. Thus, we can conclude that there are at most k consecutive integers which are not encodable for GH_a, where $a = -(4 + k)$. □

Chapter 3

Cryptanaylsis of a stream cipher based on GH codes

The GH sequence as defined by J.H. Thomas can be described as being of order 2, meaning it utilizes a recurrence relation which generates the next term by adding the past two terms. In [12], Nalli and Ozyilmaz put forth a generalization to the third order for Gopala-Hemachandra Sequences, as well as third order GH codes. We denote $GH_a^3[n]$ to be the n^{th} index of the 3^{rd} order GH code, and we define 3^{rd} order GH codes with the notation GH_a^3, where $GH_a^3(n)$ represents a code for n from the sequence GH_a^3, if it exists. The third order Gopala-Hemachandra code is based on the third order Gopala-Hemachandra sequence, which utilizes the recurrence relation $GH_a^3[n] = GH_a^3[n-1]+GH_a^3[n-2]+GH_a^3[n-3]$ for all $n > 2$ with the initial conditions $GH_a^3[0] = 0$, $GH_a^3[1] = a$ and $GH_a^3[2] = 1-a$, and $a \leq -2$. In general, we have GH_a^3 as:

$$a, 1-a, 1, 2, 4-a, 7-a, 13-2a, ...$$

As an example, one could choose $a = -2$ to yield the sequence:

$$-2, 3, 1, 2, 6, 9, 17, ...$$

While the concept of a third order Zeckendorf representation is not defined explicitly for GH_a^3 in [12], their article uses the same type of Zeckendorf representation as the third order Fibonacci Sequence presented earlier from [1], save that we are now working with terms

of the GH_a^3 sequence. The third order Zeckendorf representation for n under GH_a^3 is then $n = \sum_{i=1}^{l} \alpha_i GH_a^3[i]$, where $GH_a^3[i]$ represents the i^{th} term of the third order GH_a^3 Sequence, α_i is either 0 or 1, and $\alpha_l = 1$. Further, we require that for any α_i, α_{i+1} and α_{i+2}, it is not the case that all three are 1.

Given a Zeckendorf representation $n = \sum_{i=1}^{l} \alpha_i GH_a^3[i]$, the GH_a^3 Code for n is $\alpha_1\alpha_2...\alpha_l 11$. In the same way as the third order Fibonacci Code, we append two 1's onto the end of the Zeckendorf representation of n on the premise that three consecutive 1's should now only appear at the end of an encoding. Again, this approach is plagued with the same problems as the third order Fibonacci Sequence, in that it is not uniquely decodable.

One application which has been explored in the literature besides data compression is the use of GH codes in cryptography, particularly in the design of stream ciphers. In 2014, Nalli and Ozyilmaz presented a stream cipher in the context of order 2 and order 3 GH codes [12]. We cryptanalyze the stream cipher presented in [12] here.

3.1 The GH Cipher

A stream cipher takes a plaintext composed from some plaintext alphabet P, which is a set of symbols that may be encoded, and encodes them into a ciphertext composed from some ciphertext alphabet C. It does so by generating a keystream using a keystream generator g, and an initializing key from the keyspace K of possible keys. The keystream itself also has an alphabet, L, of possible symbols which may compose the keystream. Finally, there must be an encryption and decryption rule which enciphers an arbitrary plaintext into a ciphertext, and a valid ciphertext into a plaintext respectively.

The cipher presented in [12] uses English alphabetic characters as the plaintext alphabet. The plaintext is processed prior to encryption by replacing each plaintext letter with its index in the alphabet (a number between 1 and 26), and then by converting that number into a GH code for a given sequence. Once all the GH Codes have been obtained, each one is padded with zeros so that they are all of the same length, and then the codes are concatenated

together. Encryption is done by taking the keystream generated by a key and performing the bit-wise exclusive or operation on the encoded plaintext string and the keystream.

The key for the stream cipher has three components. These components are the order m of the recurrence relation (2 or 3), the specific parameter a used, and the length of the longest GH code used in the message. Using the tables from [4], the only parameters a for which GH_a^2 encode all of the integers 1 through 26 are $-2 \leq a \leq -4$, and using the tables from [12], the only parameters a for which GH_a^3 encode all of the integers 1 through 26 are $-2 \leq a \leq -10$. For order 2, the shortest code that can be used is of length 3, and for order 3, the shortest code is of length 4. For both orders 2 and 3, the longest code is of length 9. In order to get the key, one looks up or determines the order m Fibonacci code for $-a$, and then pads the end of the code with zeroes until its length matches the length of the longest GH code used to encode the plaintext. Note that this implies possessing just the order m and the parameter a is not enough to fully determine the key, and thus the final key used is dependent on the message. Once the key has been found, the keystream is generated by simply repeating the key until it is as long as the processed plaintext, after which encryption is performed by taking the bit-wise exclusive or of the keystream and the plaintext.

The receiver generates the keystream as above, and then performs the bit-wise exclusive or operation on the ciphertext and the keystream, which returns the encoded version of the plaintext. To decode the plaintext, the receiver will then split the string into blocks of the length of the key, remove all zero padding, and then convert each GH_a^m code into its associated integer. Once the list of integers is obtained, all that remains is to convert each integer back into a letter by finding which letter is at that index in the alphabet.

As an example, let us encipher the word "secure". Firstly, we must convert each letter to its index in the alphabet. In this case, we have:

19 5 3 21 18 5

Now, we must convert each of these numbers into a GH encoding for some sequence. Let us use order $m = 3$ and $a = -4$, then the codes associated with each number are, from [12, Table 3]:

100100111 0111 001111 000000111 101000111 0111

Next, we must pad each code with zeroes in order for every code to be the same length. Since the longest code in our plaintext comes from s, with a length of 9, each code must be padded to length 9 with 0's:

100100111 011100000 001111000 000000111 101000111 011100000

Now, we have everything we need to find the total key and construct the keystream. Since $m = 3$ and $a = -4$, we look up the order 3 Fibonacci Encoding of 4 in [12, Table 1]. We find the encoding to be 00111. After padding with zeroes, we get the key as 001110000. This gives us a plaintext string of:

100100111011100000001111000000000111101000111011100000

and a keystream of:

001110000001110000001110000001110000001110000001110000

Performing the exclusive or operation, we get the ciphertext as:

101010111010010000000010000011101111001101110100100000

Now, in order to decipher the ciphertext, we use the key to again generate the keystream, and perform the exclusive or of the ciphertext and the keystream, yielding the encoded plaintext

string:

$$100100111011100000001111000000000111101000111011100000$$

From here, we split the string into blocks of length 9 since our key is of length 9, yielding the following blocks of codes:

$$100100111 \;\; 011100000 \;\; 001111000 \;\; 000000111 \;\; 101000111 \;\; 011100000$$

We then remove all zero padding, and get:

$$100100111 \;\; 0111 \;\; 001111 \;\; 000000111 \;\; 101000111 \;\; 0111$$

We then use the GH^3_{-4} table [12, Table 3] to determine the integers encoded by these codes:

$$19 \;\; 15 \;\; 3 \;\; 21 \;\; 18 \;\; 5$$

Finally, we convert these integers back into letters, and obtain "secure".

3.2 Cryptanalysis

The relatively small range of valid a parameters over order 2 and order 3 raises the question of how effective a brute force attack might be on this stream cipher. When we refer to brute force, we refer to an attack on the cipher by which we try to decrypt a ciphertext with all possible keys, eventually obtaining the correct decryption in the process. In order to carry out such an attack, we must first know how many valid keys there are for this cipher.

The two orders of GH code used by the authors are $m = 2$ and $m = 3$. Since the size of the key is dependent on the size of the largest code used, we must know how small and how large a message can be. Order 2 GH codes have a minimum length of 3 and a maximum length of 9. There are 3 values for the a parameter for order 2 which encode the integers 1

through 26, and so we have $3 * 7 = 21$ possible keys. Order 3 GH codes have a minimum length of 4 and a maximum length of 9. There are 9 values for the a parameter for order 3 which encode the integers 1 through 26, and so we have $9 * 6 = 54$ possible keys. This gives us a total of 75 keys. That lends itself kindly to a brute-force approach, in which we simply try all these possible key combinations on the ciphertext and analyze the outputs of these 75 decryptions.

While a keyspace of 75 keys is rather small, the brute force approach for this stream cipher can be further optimized based on two principles. Firstly, some keys may be disqualified before even attempting decryption due to the length of the key relative to the ciphertext. Second, not all keys will lend themselves to a valid decryption of the ciphertext.

Before examining the contents of the ciphertext, one may examine the length of the ciphertext to disqualify a wide range of keys. Again, take our ciphertext for the word "secure." The length of that ciphertext is 54 bits because we had 6 codeblocks of length 9. Since the key is padded to the length of each codeblock, the length of the key must divide the length of the ciphertext. Since 4,5,7 and 8 do not divide the length of the ciphertext evenly, no key with a block length of 4,5,7 or 8 could possibly have generated the ciphertext in question.

Of the remaining keys which can be used to decrypt a ciphertext, most keys are unable to yield a valid decryption, in that a codeblock in the decrypted ciphertext will often not be interpretable as a GH code under the given key. We consider a key to be a valid key relative to a ciphertext if it can be decrypted into some plaintext (independent of whether it is meaningful or not), and we consider a key to be invalid for a ciphertext if it is unable to be used to decrypt a ciphertext into a sequence of characters from the English alphabet. For example, take our ciphertext for the word "secure" encoded using the key $a = -4$, $m = 3$, and block length 9:

101010111010010000000001000001110111100110111010010000

Let us not change the order and a parameter, and instead only change the block length to be 6 instead of 9. The key would now be 001110, and the first ciphertext block we would examine would be 101010. Then 100100 would be the first GH codeword to be interpreted from the GH^3_{-4} table [12, Table 3]. Since this code does not end in three 1's, the first codeblock of this ciphertext does not correspond to a valid GH^3_{-4} code. This is just one instance of an invalid key which can be disqualified very quickly without performing the full decryption. These two characteristics of this particular cipher substantially speed up brute force attacks.

We developed an encryption program and a decryption program for experimenting with the cipher. The encryption program takes a message, a parameter a, and an order m, and generates a ciphertext with a block length determined by a and m, just as described. The decryption program takes a ciphertext, the parameter a, the order m, and the block length of the message (which fully determines the key), and produces the resulting plaintext provided the ciphertext is a valid ciphertext produced by the key.

Next, we developed a program which cracks a single ciphertext. The cracking program implements the brute force approach described above, using the length of the ciphertext to disqualify several keys before feeding it to the decryption function, which stops immediately if it tries to decode an invalid code for a given key. Finally, we developed a program which streamlines the cracking of multiple ciphertexts and gathers statistics on these cracking attempts.

3.3 Experimental Results

Our experimental testing of the program that cracks the stream cipher from [12] sought to quantify how few valid decryptions that plaintexts of various sizes yield on average. Plaintexts for testing were generated by breaking up the King James Bible found on Project Gutenberg after removing the license information added to the document. Plaintexts were generated in lengths of 2, 3, 4, 5, 10, 20, 50, 100, 1000, and 10000 characters, with 1000

Instance Length	Avg Time (s)	Avg # of Valid Keys	Max # of Valid Keys
2	0.000753983	1.441	5
3	0.000982375	1.149	4
4	0.001160898	1.048	3
5	0.001347399	1.031	3
10	0.002342737	1.004	2
20	0.002612016	1	1
50	0.002767603	1	1
100	0.003831668	1	1
1000	0.013544820	1	1
10000	0.113696467	1	1

Table 3.1: Experimental results for cracking ciphertexts of varying lengths

instances for all lengths below 10000 and 332 instances for length 10000. 332 instances was the largest number of 10000 character plaintexts that could be generated from the full-text of the bible. These plaintexts were encrypted en mass by choosing a random m and a combination for each plaintext, and then running the encryption program. Next, the resulting ciphertexts were cracked en mass. The average time in seconds of cracking a ciphertext of a given length is shown in Table 3.1. Also shown in the table are the average number of valid keys and maximum number of valid keys found when cracking a ciphertext produced from a plaintext of a given length. The timing of a cracking attempt starts when the program feeds the loaded in ciphertext to the cracking function, after which the program begins testing possible keys. The timing of a cracking attempt stops after all keys have either been disqualified or attempted.

We observe that even when the original plaintext is almost trivially small, down around two characters total, the maximum number of valid keys is 5. This means that in the absolute worst case, less than 7% of the possible 75 keys are going to allow decryption to complete and yield a valid output, regardless of whether the outputs of these keys make sense to a human observer. The most striking result here is that by the time the message length reaches 10 characters, the average number of valid keys is almost 1, and by 20 characters, there were no instances found which had more than 1 valid key. This means that there is

really only one key which can produce a valid output. Using this fact, we were able to further optimize our cracking algorithm. Rather than examining the whole ciphertext, we choose never to examine more than the first 10 characters of ciphertext for the purposes of running a decryption to find the key. Only when we have a list of valid keys do we then go back and perform the full decryption on the whole ciphertext. This drastically cuts the time that it takes to crack a ciphertext obtained from a plaintext which is longer than 10 characters, and is particularly noticeable when the message size is larger than 1000 characters. Table 3.1 presents the average time for the optimized version of the cracking program.

In summary, the time to crack even a large ciphertext is less than a second, given nothing but the ciphertext itself. Further, small ciphertexts have on average less than 2 potential keys to choose from before even considering whether a decryption makes sense as a message. Hence, we conclude that this stream cipher is quite insecure. Further, were one to use a standard stream cipher with a random binary key of length 8, which is the smallest block length generally used by a meaningful message in the analyzed GH cipher, there would be 2^8, or 256 possible valid keys. Comparatively, the keyspace of the GH stream cipher over all lengths is limited to 75, making brute force attacks significantly faster.

In 2016, Basu and Das presented a different design for a stream cipher in their paper utilizing GH_a^2 sequences and codes [3]. We did not analyze the construction in [3], since their design is equivalent to a standard stream cipher which simply uses GH codes to encode the plaintext into binary. Their scheme utilizes an arbitrary codebook in which they assign each character to specific GH codes. The key in their scheme is a random binary key, and has no relation to GH sequences or codes themselves. Since the encodings from plaintext to GH code must be public knowledge by Kerckhoffs' principle, their scheme cannot provide additional security beyond that of a standard stream cipher.

Chapter 4

Conclusion and Open Problems

Our work here consisted of proving theoretical results on GH_a^2 sequences, thus providing more solid ground upon which further investigations of GH_a^2 sequences may be conducted. On universality, we provided a new proof of universality for GH_a^2 when $-2 \geq a \geq -4$, and provided a means to correct a flawed proof in the literature. We provided two new algorithms for determining whether a $GH_a^2(n)$ code exists for given parameters a and n and constructing one if it indeed exists. Our algorithms are provably correct and run in linear time in the length of the code output and are thus asymptotically optimal. We also provide a general proof of a bound the maximum number of consecutive integers whose encodings do not exist, relative to a. Finally, we cryptanalyze a stream cipher based on GH codes from [12], developing an optimized brute force approach which allows arbitrary ciphertexts to be easily cracked without knowledge of the key.

While we now have several general results for GH_a^2, as far as we know, [12] is the only paper in the literature which examines the GH_a^3 sequence and its codes in any detail. Further, said paper consists of providing a definition for GH_a^3, providing $GH_a^3(n)$ encodings for $-2 \geq a \geq -20$ and $1 \leq n \leq 100$, and presenting a stream cipher which utilizes these codes. This means that there are essentially no general results on GH_a^3 as a whole, let alone higher order GH sequences and codes. It would be interesting to see if results similar to ours might be proven for order 3 GH codes, or in general for order m GH codes. In particular, it would be interesting to prove whether some range of a exists for order 3 such that GH_a^3 permits a universal code, to develop an algorithm which determines whether a $GH_a^3(n)$ code exists for

a given set of parameters a and n, and construct one if it exists, and to prove a similar bound on the maximum number of consecutive integers with N/A codes relative to a in order 3.

The usefulness of sequences GH_a^m for $m > 2$ for data compression is questionable though, as these sequences are not necessarily uniquely decodable. From the tables presented in [12], we can easily find concatenations of GH_a^3 encodings which are not uniquely decodable, an important property that is needed for use in data compression. For example, take the encodings of 15 and 13 in GH_{-4}^3:

10001111 00010111

We know that those decode to 15 and 13 because we were the ones to encode them. However, because 15 has four consecutive 1's, and the concatenation of these two codes could be interpreted in two different ways by the receiver. As above, or as

1000111 100010111

which for GH_{-4}^3, would correspond to the integers 4 and 25. This example shows that, while it may be possible to prove that some range of a allows universal GH_a^3 codes, these codes do not guarantee unique decoding. We note that in [9], Klein et al made a similar observation that third order Fibonacci Codes are not uniquely decodable.

The above example illustrates the need in general to obtain a fixed-length code if one intends to use the GH_a^3 code in the cipher from [12]. Initially, the idea of padding out a variable length code seems like a strange choice, both for the sake of security and the size of the encodings. The advantages of having small encodings for small integers such as 1 are lost once one has to pad out the encoding with zeros for the cipher, and the concept of breaking up the ciphertext into codeblocks significantly streamlined the process of testing keys, even if the associated padding had a multiplicative effect on the size of the keyspace. However, as GH_a^3 is not uniquely decodable, there is no guarantee that the parameters a and m will

be sufficient for deciphering the ciphertext, and thus it was required that zero padding be added to the codeblocks.

Despite the fact that higher order GH codes may not be useful in the area of data compression, there is still room also to produce even more general results for order m, such as a finding a function $f(m)$, if it exists, such that for GH_a^m, $-2 \geq a \geq f(m)$ permits universal codes. Another interesting problem is to develop an algorithm which determines whether a $GH_a^m(n)$ code exists for a given set of parameters m, a, and n, and construct one if it exists. That is, it takes a, m, and n as inputs, and produces an encoding of n using the order m GH sequence parameterized with a, if it exists.

Finally, our work on GH sequences and codes focuses primarily on the specific case of the general Gopala-Hemachandra sequence in which $b = 1 - a$ for $a \leq -2$. This family of sequences is interesting due to the fact that it can permit universal codes, but there may be other interesting parameterizations of the Gopala-Hemachandra Sequence to investigate, as well as the possibility of general results on GH sequences with arbitrary initial values of a and b.

BIBLIOGRAPHY

[1] A., A., and A., F. Robust transmission of unbounded strings using fibonacci representations. *IEEE Transactions on Information Theory 33*, 2 (1987), 238–245.

[2] American National Standards Institute. *ASCII Graphic Character Set*, 1975.

[3] Basu, M., and Das, M. Uses of second order variant fibonacci universal code in cryptography. *Control and Cybernetics 45* (2016), 239–257.

[4] Basu, M., and Prasad, B. Long range variations on the fibonacci universal code. *Journal of Number Theory 130*, 9 (2010), 1925 – 1931.

[5] Carlitz, L., Scoville, R., and Hoggatt Jr, V. Fibonacci representations of higher order-ii. *Fibonacci Quart 10*, 1 (1972), 43–69.

[6] Elias, P. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory 21*, 2 (1975), 194–203.

[7] Fraenkel, A. S., and Klein, S. T. Robust universal complete codes for transmission and compression. *Discrete Applied Mathematics 64*, 31 (1996), 55.

[8] Huffman, D. A. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE 40*, 9 (1952), 1098–1101.

[9] Klein, S. T., and Ben-Nissan, M. K. On the usefulness of fibonacci compression codes. *The Computer Journal 53*, 6 (2010), 701–716.

[10] Lekkerkerker, C. G. Voorstelling van natuurlijke getallen door een som van getallen van fibonacci. *Stichting Mathematisch Centrum. Zuivere Wiskunde*, ZW 30/51 (1951).

[11] Lucas, E. *Théorie des nombres*, vol. 1. Gauthier-Villars, 1891.

[12] Nalli, A., and Ozyilmaz, C. The third order variations on the fibonacci universal code. *Journal of Number Theory 149* (2014), 15 – 32.

[13] Pal, J., and Das, M. The gopala-hemachandra universal code determined by straight lines. *Journal of Mathematics and Computer Science 19* (2019), 158–170.

[14] Salomon, D., and Motta, G. *Handbook of Data Compression*. Springer Science & Business Media, 2010.

[15] Sayood, K. *Lossless Compression Handbook*. Elsevier, 2002.

[16] Singh, P. The so-called fibonacci numbers in ancient and medieval india. *Historia Mathematica 12*, 3 (1985), 229 – 244.

[17] Thomas, J. H. Variations on the fibonacci universal code, 2007.

[18] Zeckendorf, E. Representation des nombres naturels par une somme des nombres de fibonacci ou de nombres de lucas. *Bull. Soc. Roy. Sci. Liege 41* (1972), 179–182.