Theses and Dissertations   1. Thesis and Dissertation Collection, all items

2006-09

Absolute position measurement for automated guided vehicles using the Greedy DeBruijn Sequence

Ortiz, John E.

Monterey California. Naval Postgraduate School

http://hdl.handle.net/10945/2612

NAVAL
POSTGRADUATE
SCHOOL

MONTEREY, CALIFORNIA

THESIS

ABSOLUTE POSITION MEASUREMENT FOR
AUTOMATED GUIDED VEHICLES USING THE GREEDY
DEBRUIJN SEQUENCE

by

John E. Ortiz

September 2006

Co- Advisors: Harold M. Fredricksen
                                        Jon T. Butler

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE		*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE September 2006	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE Absolute Position Measurement for Automated Guided Vehicles using the Greedy DeBruijn Sequence	5. FUNDING NUMBERS N/A
6. AUTHOR(S) John E. Ortiz	

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER N/A

9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	10. SPONSORING/MONITORING AGENCY REPORT NUMBER N/A

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	12b. DISTRIBUTION CODE N/A

13. ABSTRACT (maximum 200 words)

Automated Guided Vehicles (AGVs) use different techniques to help locate their position with respect to a point of origin. This thesis compares two approaches that utilize a binary track laid on the floor for the AGV to follow. Both approaches use equally spaced n-tuples on the track that the AGV can use to compute its position. Both approaches also have the special feature that every n-tuple on the binary track is unique and can be used to designate the position of an AGV. The first approach, developed by E.M. Petriu, uses a Pseudo-Random Binary Sequence (PRBS) as a model for the binary track. In the second approach, we use a Greedy DeBruijn Sequence (GDBS) as a model for the binary track. Unlike the PRBS model, the GDBS model has a natural ordering which can be used to determine the position of the AGV more quickly and efficiently than the PRBS model.

14. SUBJECT TERMS Automated Guided Vehicle (AGV), DeBruijn Sequence, Necklace, Necklace Algorithm, Pseudo-Random Binary Sequence, Absolute Position Measurement, Discrete Logarithm Problem	15. NUMBER OF PAGES 169
	16. PRICE CODE N/A

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

i

THIS PAGE INTENTIONALLY LEFT BLANK

ABSOLUTE POSITION MEASUREMENT FOR AUTOMATED GUIDED VEHICLES USING THE GREEDY DEBRUIJN SEQUENCE

John E. Ortiz
Lieutenant, United States Navy
B.S., University of Central Florida, 2000

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 2006

Author: John E. Ortiz

Approved by: Harold M. Fredricksen
                    Co-Advisor

                    Jon T. Butler
                    Co-Advisor

                    Jeffrey B. Knorr
                    Chairman, Department of Electrical Engineering

iii

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Automated Guided Vehicles (AGVs) use different techniques to help locate their position with respect to a point of origin. This thesis compares two approaches that utilize a binary track laid on the floor for the AGV to follow. Both approaches use equally spaced n-tuples on the track that the AGV can use to compute its position. Both approaches also have the special feature that every n-tuple on the binary track is unique and can be used to designate the position of an AGV. The first approach, developed by E.M. Petriu, uses a Pseudo-Random Binary Sequence (PRBS) as a model for the binary track. In the second approach, we use a Greedy DeBruijn Sequence (GDBS) as a model for the binary track. Unlike the PRBS model, the GDBS model has a natural ordering which can be used to determine the position of the AGV more quickly and efficiently than the PRBS model.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

ix

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I want to thank my wife, Danielle, for her support and encouragement during our time here. I know that it has not been easy for her and I know she looks forward to having me available at nights and on the weekends again. I want to thank the rest of my family for their support and help whenver my wife and I needed it. I want to thank my advisors, Dr. Fredricksen and Dr. Butler, for their availability, instruction and insight. Both of you helped me craft my thesis into a far better product than I could have done on my own. Mostly, I want to thank the Lord for allowing me to have the opportunity to study at the Naval Postgraduate School. I will continue to trust in Him as He uses whatever means to shape me into the person He wants me to become.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

The greedy DeBruijn sequence (GDBS) of span n is a binary string of length 2^n where every n-tuple in the sequence is unique. Originally the sequence was defined by the rule: Start with n-1 zeros and then add a 1. Continue adding 1's as long as the n-tuple created has not been seen before in the sequence. If it has already appeared, add a zero instead. Continue by adding 1s. If neither a 0 nor a 1 can be added, then stop. The process will generate a sequence of length 2^n [Mar]. The sequence can also be constructed more efficiently by concatenating a list of combinatorial objects called necklaces in lexicographic order. [Fr] The motivation behind this project is to try to determine a mapping between the set of necklaces for the greedy DeBruijn sequence and the natural code, much like the mapping that exists between the Gray code strings and the natural code. Finding this necklace-to-natural code mapping would enable us to determine the absolute position of any arbitrary n-tuple in a greedy DeBruijn sequence. Although we do not solve the mathematical problem in determining this mapping, we are able to apply the properties of this DeBruijn sequence to construct a very efficient implementation in helping an Automated Guided Vehicle (AGV) determine its position on a long DeBruijn track. Since every n-tuple in the DeBruijn sequence is unique, the n-tuples can be used to designate unique positions for the AGV.

In this thesis, we compare two approaches to the AGV implementation. E.M. Petriu developed the first approach. He uses a Pseudo-Random Binary Sequence (PRBS) generated by a linear shift register as the basis of his binary track. He implements a series of n-tuples, called milestones, on the track that provide the AGV with information about its location. We advocate the use of the GDBS as the binary track since it has certain advantages over the PRBS. Like Petriu, we use certain n-tuples, called signposts, to give the AGV information about its position. However, the way the signposts are generated and they way they are used differs significantly from Petriu's approach. With the GDBS, the AGV can determine its location in a far more efficient manner than can the method used by Petriu.

Chapter II gives the background on the mathematical structures of necklaces. There the Necklace Algorithm developed by Fredricksen, to generate necklaces efficiently, is introduced. [Fr] This algorithm plays an important role in helping the AGV determine its absolute position.   There the GDBS is constructed from the necklaces as generated by the Necklace Algorithm.

Chapter III introduces Petriu's method. It describes the sequential-parallel approach Petriu uses to determine the AGV's absolute position. We also discuss the equipment and time costs associated with his scheme and the computation needed to determine the optimum number of milestones that will be used.

Chapter IV describes our method and the results obtained. We introduce the initial steps the AGV must take to determine its location. We describe how signposts are generated and where they are placed on the GDBS. We compare the number of milestones versus the number of signposts and find they are similar in both methods but the GDBS method has computational advantages over Petriu's PRBS scheme. The GDBS scheme can also be used to potentially optimize the distribution of more than one AGV on the GDBS track. Finally, we describe an alternate means of signpost generation and compare it to the approach we decided to use.

Chapter V describes current problems of a mathematical nature that simultaneously give insight into necklaces and their structure and significantly improve the AGV implementation. By solving the necklace-to-natural code-mapping problem, the AGV would not even need an external binary track to follow. There are two fruitful approaches that were initiated to try to solve this problem but more work needs to be done.

Appendix A provides a background on linear shift registers and how they generate the PBRS. Appendix B contains the code used to execute the Necklace Algorithm, generate necklaces and signposts, and gather statistics needed to analyze the structure of necklaces.

I.   WHAT IS A NECKLACE?

A.   BACKGROUND

This chapter provides background material on using a greedy DeBruijn sequence, instead of a PBRS, as a bit track for the Automated Guided Vehicle (AGV). This DeBruijn sequence, generated via the Necklace Algorithm, has some properties that give it an advantage over the PBRS in helping determine the position of an arbitrary n-tuple of that sequence.

B.   STRUCTURE OF NECKLACES

Let A_n represent the set of all n-bit binary strings. The total number of possible strings in A_n is 2^n. We define a mapping $\phi : A_n \to B_n$ where the set B_n is the set of all *necklaces* over A_n. By a necklace, we mean a linear representation of the collection of all circular permutations of a binary n-tuple into another binary n-tuple. For example, for $n = 5$, let $a_1 = 01101$ be an arbitrary binary string in A_5. Then the following binary strings are all left circular permutations of a_1:

01101

1101**0**

101**0**1

01**0**11

1**0**110
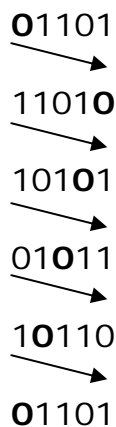
01101

Figure 1.   Circular Permutations of 01101

Note that each binary string is a result of cyclically rotating the previous string one position to the left. Also, note that the last cyclic permutation just reproduces the original string a_1. From this, we can see that each binary string in this list is some number of cyclical shifts of any other binary string in the list. Although each binary

1

string is distinct in A_5, we can represent this whole list of binary 5-tuples as an equivalence class (the elements of B_n) of some binary string in the list. We pick a particular binary string to be the linear representation of this equivalence class. We call this representative a *necklace,* and we define it to be the element of A_n that has the largest decimal value. (See Table 1.)

Binary String	Decimal Equivalent
01101_2	13
11010_2	26
10101_2	21
01011_2	11
10110_2	22

Table 1.   Decimal Equivalents of Elements of Equivalence Class for 01101

We see that 11010_2 has the greatest decimal equivalent so this is the "necklace" for this equivalence class. Note that the necklace always ends in a "0" (except for the necklace of all "1"s) since a "1" at the end can always be cyclically shifted to the front of the binary string to give a larger decimal equivalent. Counting from the Most Significant Bit (MSB), or the leftmost bit, a necklace representative contains its greatest number of leading ones before the first zero.

The crucial point here is that every possible binary string in A_n (for every *n*) will be a member of some necklace's equivalence class. Further, it is possible to produce a list of necklaces whose equivalence classes are disjoint (no two distinct equivalence classes can contain any binary string in common). One obvious way to tell if two necklaces are from two different equivalence classes is to see if they have different densities (different number of ones). The certain way to ensure that two equivalence classes are disjoint is to test if one necklace can be circularly permuted into the other one. For example, let $b_1 = 110010$ and $b_2 = 110100$ be two necklaces for *n* = 6. Neither can be

2

obtained from the other by circular permutation although both have the same density.

Both have different equivalence classes as seen in Table 2:

Equivalence Class for 110010	Equivalence Class for 110100
110010	110100
100101	101001
001011	010011
010110	100110
101100	001101
011001	011010

Table 2.   Comparison of Equivalence Classes

One could randomly choose binary strings in A_n and search for necklaces, but this would be an inefficient procedure at best (especially for large n). Fortunately, an elegant algorithm has been developed to generate all the necklaces for any n [Fr].

C.   NECKLACE ALGORITHM

The Necklace Algorithm [Fr] generates the list of all necklaces in lexicographic order from the all ones necklace to the all zeros necklace. We use a subroutine, called the Θ−Algorithm, to generate a binary string, which may be a necklace.

Θ−Algorithm [Fr]**:**

1.   Given an arbitrary binary sequence of length n (where the indices ascend from left to right), read the binary string from right to left and find the largest index corresponding to the bit that is equal to "1" and any remaining subsequent bits (bits of a higher index) are all 0s. Denote this largest index by j. If j is equal to n, there will be no zeros following it.

2.   Denote this binary n-tuple as the parent string. Copy the contents of this parent string into another string called the child string. Subtract "1" from the bit corresponding to the jth index in the child string. This bit will become a zero. Discard the remaining n-j bits (all zeros) of higher index following this bit.

3.   Since we discarded the remaining n-j bits, we may need to fill them with bits so that the child string has a length of n. We do this by copying n-j

3

bits from the beginning of the child string into the remaining *n-j* "spaces" as many times as necessary to complete an *n*-long string.

The following theorem [Fr] states:

If $a = a_1, a_2, ..., a_n$ is a necklace, $\neq 000...0$, then one of $\theta(a), \theta^2(a), ..., \theta^{[(n-1)/2]}(a)$ is a necklace.

As an example, let the 11110 be the parent string. We copy the contents of this parent string into a child string. Then, $j = 4$. Subtracting "1" from the bit corresponding to this index, we have

$$11110 \text{ (child string)}$$

$$\underline{-1}$$

$$1110\_ \text{ (child string)}.$$

To fill the remaining position, we need to copy one bit from the beginning of the child string so we have:

$$11101.$$

Necklace Algorithm [Fr]:

1. Start with the *n*-bit binary string of all ones denoted by 1^n. This is the first necklace and the first binary parent string.

2. To find the $(i+1)^{st}$ necklace for $i \geq 1$, apply the $\Theta-$algorithm to the ith necklace and denote the execution of this algorithm as θ^1.

3. If the jth index of the parent string divides *n*, the child string is a necklace. Then increment *i* and return to step 2. Otherwise it is nota necklace. The child string now becomes the parent string for the next execution of the Θ-Algorithm.

4. If *j* does not divide n, then apply $\theta^2, \theta^3, ..., \theta^k$, where k denotes the least number of times needed to apply the $\Theta-$algorithm to the ith necklace until *j* divides *n*. The resulting child string is a necklace so increment *i*.

5. If the all zeros necklace 0^n has not appeared in steps 2,3 and 4 above, return to step 2.

The number of bad n-tuples (or non-necklaces), B_n, between any pair of consecutive necklaces generated by the Necklace Algorithm is bounded by $B_n \le \left[\frac{1}{2}(n-1) \right] Z_n$ where

$$Z_n = \frac{1}{n} \sum_{d|n} 2^{\frac{n}{d}} \varphi(d) \qquad (1.1)$$

.

Z_n is the number of necklaces, d is a divisor of n, and $\varphi(d)$ is Euler's totient function which represents the number of integers that are relatively prime to d, including "1" [Go].

By the Necklace Algorithm, we generate a lexicographic list of necklaces whose equivalence classes together contain all possible 2^n binary strings of length n. The following example demonstrates the Necklace Algorithm.

Let $n = 6$. Starting with the initial necklace and parent string:

$$111111$$

note that the index $j = 6$ and there are no zeros following the last bit.

Also, the index of this parent string ($j = 6$) divides 6, so the child formed from the all ones parent string will also be a necklace. Subtracting 1 from the last bit we have:

$$111110$$

Since the string has 6 bits, the Θ–Algorithm need not fill in any missing bits. This is the second necklace and the new parent string. Note that the index of the last non-zero bit is 5. This index does not divide 6 so the child string formed from the parent string 111110 is not a necklace. The Θ–Algorithm procedure yields:
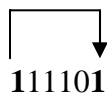
$$11111\mathbf{0} \ (\text{parent string})$$

$$\underline{-\mathbf{1}}$$

$$1111\mathbf{0}\_ \ (\text{child string})$$

To fill the remaining position, we copy one bit from the beginning of the child string so we have:

$$\mathbf{1}11101$$

To see how this child string is not a necklace, just cyclically shift the last bit to the beginning of the string to produce

<div align="center">111110</div>

Note that this is the parent string that was a necklace. Remember that a necklace is the representative of the equivalence class that has the largest decimal value. It is easily seen that 111101 belongs to the equivalence class of 111110 and that $111110_2 > 111101_2$. To continue, note that for the string 111101, $j = 6$. This obviously divides six so the child string of 111101 is a necklace although the string 111101 is not a necklace. To see the necklace, subtract "1" from 111101 to produce

<div align="center">111100</div>

which is clearly a necklace. (Further this necklace cannot be cyclically rotated to produce 111110. So the equivalence class of 111100 is distinct from the equivalence class of 111110).

Continuing in the same manner, we generate a list containing both binary necklaces and non-necklaces from 111111 to 00000.

111111 - necklace (j^{th} index = 6)

111110 - necklace (j^{th} index = 5)

111101 - non-necklace (j^{th} index = 6)

111100 - necklace (j^{th} index = 4)

111011 - non-necklace (j^{th} index = 6)

111010 - necklace (j^{th} index = 5)

111001 - non-necklace (j^{th} index = 6)

111000 - necklace (j^{th} index = 3)

110110 - necklace (j^{th} index = 5)

110101 – non-necklace (j^{th} index = 6)

110100 - necklace (j^{th} index = 4)

110011 - non-necklace (j^{th} index = 6)

110010 - necklace (j^{th} index = 5)

110001 - non-necklace (j^{th} index = 6)

110000 - necklace (j^{th} index = 2)

101010 – necklace (j^{th} index = 5)

101001 - non-necklace (j^{th} index = 6)

101000 - necklace (j^{th} index = 3)

100100 - necklace (j^{th} index = 4)

100010 - non-necklace (j^{th} index = 5)

100001 - non-necklace (j^{th} index = 6)

100000 - necklace (j^{th} index = 1)

000000 - necklace (end of algorithm)

With the necklace algorithm, we only need to generate and search through 23 (instead of $2^6 = 64$) binary strings to find out which ones were necklaces. Of these 23, exactly 14 are necklaces.

D.   DEBRUIJN SEQUENCES

A binary DeBruijn sequence of span n contains every possible n-tuple of length n in a cycle of length 2^n. The necklace $x_1 x_2 ... x_n$ $\left(where\, each\, x_i \in \{0.1\} \right)$ has a *period* of length d if $x_1 x_2 ... x_n = x_{d+1} x_{d+2} ... x_n x_1 x_2 ... x_d$. So

$x_1 x_2 ... x_d = x_{d+1} x_{d+2} ... x_{2d} = x_{2d+1} x_{2d+2} ... x_{3d} = ...$ is the periodic sub-necklace of period d. Lyndon words are necklaces that exhibit no sub-period behaviour or necklaces with multiple cycles removed. We are only interested in concatenating all the Lyndon words from the Necklace Algorithm, in lexicographic order, to generate a DeBruijn sequence. The length of a Lyndon word is equal to one period of a necklace. Table 3 lists all the necklaces along with their periods and respective Lyndon words for the case where $n = 6$. (We list those Lyndon words in bold face that are shortened by virtue of being periodic necklaces).

Necklace	Lyndon Word	Period
111111	**1**	**1**
111110	111110	6
111100	111100	6
111010	111010	6
111000	111000	6
110110	**110**	**3**

7

Necklace	Lyndon Word	Period
110100	110100	6
110010	110010	6
110000	110000	6
101010	**10**	**2**
101000	101000	6
100100	**100**	**3**
100000	100000	6
000000	**0**	**1**

Table 3.   Necklaces and Lyndon Words for n = 6

To determine the period of a necklace, simply count the number of bits until the pattern of ones and zeros begins to repeat. It is easy to see this process if one concatenates copies of the necklace with itself (where the carat "^" operator is used for concatenation) and count how many bits are needed before the same pattern of zeros and ones is repeated. In the following examples

111111^111111 = 111111111111
000000^000000 = 000000000000

the pattern repeats itself after only one bit so the period is one. The following sequence

110110^110110 = 110110110110

repeats the patterns of ones and zeros after three bits. So the period is three. However, the following sequence

111100^111100 = 111100111100

requires six bits before the pattern repeats itself. So its period is six. We can now concatenate in order the Lyndon words found by the Necklace Algorithm to find the DeBruijn sequence. For $n = 6$ the DeBruijn sequence is:

1^111110^111100^111010^111000^110^110100^110010^110000^10^101000^100^100000^0

The resulting sequence is $2^6 = 64$ bits in length. Reading from left to right, the first string is 111111. Cycling one bit to the right and reading six bits yields the next string 111110. Continuing in this manner until the last bit is reached yields all possible binary strings of length six that are contained in the above sequence exactly once. The binary strings 000001, 000011, 000111, 001111, and 011111 are read from the sequence as we cycle one bit at a time to the right from 000000 to 111111.

The reason that we concatenate the Lyndon words, and not the necklaces themselves, is to ensure that every possible binary sequence of length six occurs only once. To see this, imagine if we concatenated the entire necklaces and not just the Lyndon words:

111111^111110^111100^111010^111000^110110^110100^110010^110000 ^ 101010 ^ 101000^100100^100000^000000 =

111111111110111100111010111000**110110**110100110010110000**101010**101000**100100**1**000000**00000 (84 bits)

If we read the sequence formed by concatenation from left to right, we see that the binary strings 111111 and 000000 occur multiple times. The binary strings 110110, 101010 and 100100 occur multiply. By using the Lyndon words, every possible binary string of length six occurs exactly once in the DeBruijn sequence for $n = 6$. This is the same sequence as that formed by the greedy algorithm described in Chapter I.

Note that $n = 6$ is a composite number (it has more factors other than one and itself) and that the factors of six (one, two, three and six) correspond to the periods of the necklaces (or the length of the Lyndon words). This is always the case for any $n \geq 0$. If n is prime (itself and 1 are the only factors), then the only necklaces that have periods less than n are the all 1s and the all 0s necklace. All other necklaces have periods equal to n. Table 4 lists the necklaces for $n = 5$:

Necklace	Lyndon Words	Period
11111	1	1
11110	11110	5
11100	11100	5
11010	11010	5
11000	11000	5
10100	10100	5
10000	10000	5
00000	0	1

Table 4.   Necklaces and Lyndon Words for n = 5

Concatenating only the Lyndon words

1^11110^11100^11010^11000^10100^10000^0 =

11111011100110101100010100100000

is the DeBruijn sequence of length $2^5 = 32$ bits.

E.   ENUMERATION OF NECKLACES

We organize the necklaces into classes according to their class number. Reading the necklace from left to right, the class number corresponds to the number of ones preceeding the first zero. When we list the necklaces for $n = 6$, they are listed in a decreasing order from 111111_2 to 000000_2. This is one of the special features produced by the necklace algorithm that we will use later on. Table 5 contains the necklaces, their class numbers and decimal equivalents for $n = 6$:

Necklace	Class Number	Decimal Equivalent
111111	6	63
111110	5	62
111100	4	60
111010	3	58
111000	3	56
110110	2	54
110100	2	52
110010	2	50
110000	2	48
101010	1	42
101000	1	40
100100	1	36
100000	1	32
000000	0	0

Table 5.   Necklaces, Class Numbers and Decimal Equivalents for $n = 6$

For purposes of this presentation, we do not shorten the necklaces to their respective Lyndon words for those necklaces whose periods are less than n since we want the true decimal equivalence of the binary number. Note that the decimal equivalents (except for the first number) decrease by two until we reach the string 110000. The decimal equivalents of 46 and 44 are "missing". The reason 46 and 44 are missing is that their binary equivalents are 101110_2 and 101100_2 respectively, and these strings are not necklaces since they can be cyclically rotated to produce 111010 and 110010 respectively. As seen in the Table, the necklaces 111010 and 110010 have already appeared in the list earlier. So the strings 101110 and 101100 belong to the respective equivalence classes of 111010 and 110010.   Recall that the list of necklaces will contain only one representative (i.e., the necklace) from each *distinct* equivalent class. All decimal equivalents that are "missing" as we continue on correspond to binary strings that are in equivalence classes of necklaces that have already appeared on the list. At

11

times, we may have more than one even decimal "missing" between two adjacent necklaces. We call this group of consecutive missing even decimals a "clump".

Define the *threshold* as the necklace that has the smallest decimal equivalent from the beginning of the list of necklaces and for which no "missing" has yet been encountered. This necklace is designated by p ones followed by all zeros where

$$p = \left\lfloor \frac{n-1}{2} \right\rfloor \tag{1.2}$$

For $n = 6$, $p = \left\lfloor \frac{6-1}{2} \right\rfloor = 2$ so our threshold is 110000 (highlighted in the Table above).

The decimal equivalents have no missing values until the class number of the necklace is strictly less than p. Any necklace whose class number is strictly less than p (i.e., one and zero for $n = 6$) will have "missing" decimal equivalents.

Define *cardinality* of a class as the number of necklaces in a given class. For $n = 6$, the cardinality of class one is four. The entries in the Table below are the cardinalities for a given class number $k = n-q$ and a given necklace length n and an index q. The totals represent the total number of necklaces generated by the necklace algorithm for a given n. For n = 6, the class numbers and their cardinalities are given in Table 6.

Class No.	6	5	4	3	2	1	0
Cardinality	1	1	1	2	4	4	1

Table 6.   Class Number vs. Cardinality for $n = 6$

Summing the cardinalities produces 14 necklaces for $n = 6$, which, not surprisingly, corresponds to the number of necklaces generated by the Necklace Algorithm. We can also calculate the number of necklaces of length n without running the Necklace Algorithm by using equation 1.1 on page 5 to calculate Z_n. Although the number of necklaces grows with increasing n, it is small compared to the number of total binary strings 2^n since $Z_n \sim \dfrac{2^n}{n}$ [Go]. In other words, we can represent the entire binary space

containing 2^n strings with $\dfrac{1}{n}th$ of its members (represented as the % entries in Table 7 and Table 8). This difference between Z_n and 2^n is also displayed in Figure 2.

q	n=3	n=5	n=7	n=9	n=11	n=13	n=15	n=17	n=19	n=21
0	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	1	1	1
3	1	2	2	2	2	2	2	2	2	2
4		2	4	4	4	4	4	4	4	4
5		1	6	8	8	8	8	8	8	8
6			4	14	16	16	16	16	16	16
7			1	19	30	32	32	32	32	32
8				9	50	62	64	64	64	64
9				1	56	114	126	128	128	128
10					18	178	242	254	256	256
11					1	172	433	498	510	512
12						40	635	944	1010	1022
13						1	533	1640	1968	2034
14							93	2262	3682	4016
15							1	1646	6222	7777
16								210	8072	14363
17								1	5126	23610
18									492	28828
19									1	16035
20										1169
21										1
Total	**4**	**8**	**20**	**60**	**188**	**632**	**2192**	**7712**	**27596**	**99880**
2^N	**8**	**32**	**128**	**512**	**2048**	**8192**	**32768**	**131072**	**524288**	**2097152**
%	**50**	**25**	**15.6**	**11.7**	**9.18**	**7.71**	**6.69**	**5.88**	**5.26**	**4.76**

Table 7.   Necklace Enumeration for Odd n

q	n=4	n=6	n=8	n=10	n=12	n=14	n=16	n=18	n=20	n=22
0	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	1	1	1
3	2	2	2	2	2	2	2	2	2	2
4	1	4	4	4	4	4	4	4	4	4
5		4	8	8	8	8	8	8	8	8
6		1	11	16	16	16	16	16	16	16
7			7	27	32	32	32	32	32	32
8			1	33	59	64	64	64	64	64
9				14	96	123	128	128	128	128
10				1	101	223	251	256	256	256
11					30	338	479	507	512	512
12					1	305	844	991	1019	1024
13						63	1202	1867	2015	2043
14						1	940	3199	3914	4063
15							142	4281	7276	8010
16							1	2915	12128	15462
17								328	15267	28374
18								1	9078	46005
19									765	54511
20									1	28418
21										1810
22										1
Total	6	14	32	108	352	1182	4116	14602	52488	190746
2^N	16	64	256	1024	4096	16384	65536	262144	1048576	4194304
%	37.5	21.9	12.5	10.5	8.59	7.21	6.28	5.57	5.01	4.55

Table 8.   Necklace Enumeration for Even n
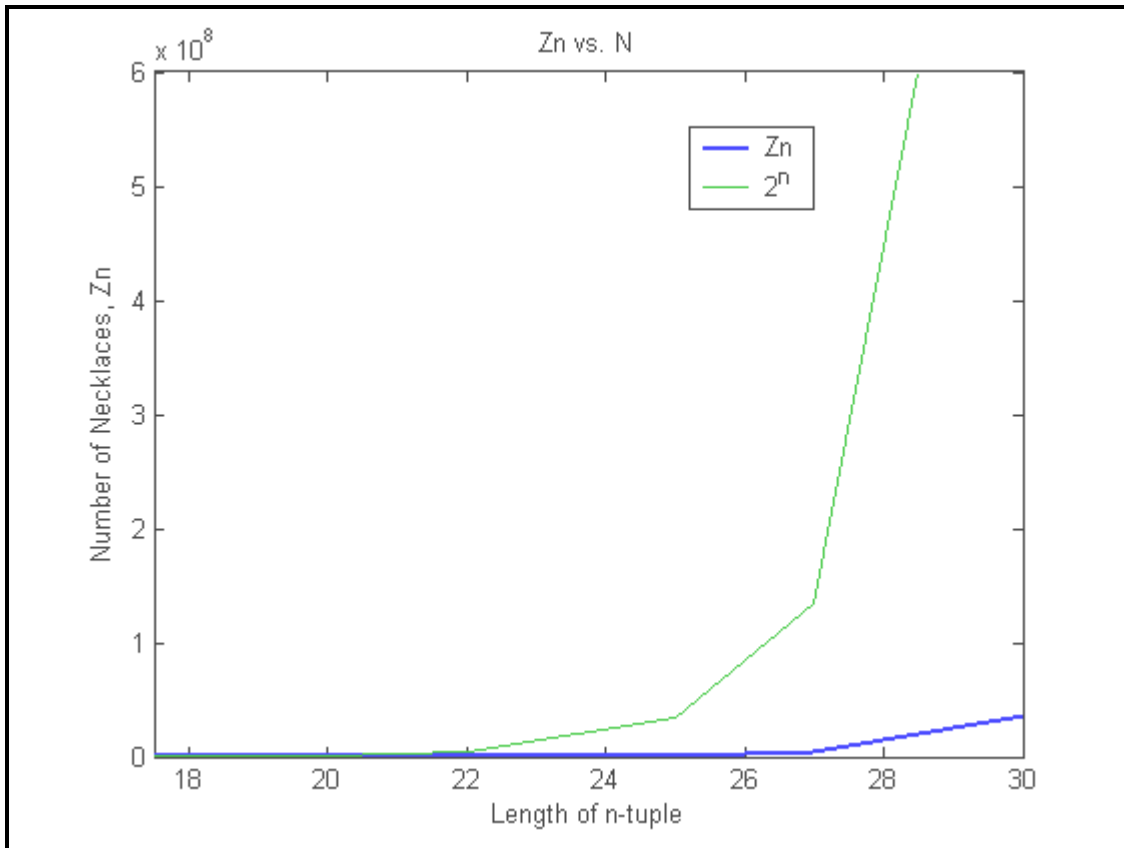
Figure 2. Z_n vs. 2^n

THIS PAGE INTENTIONALLY LEFT BLANK

II.   PETRIU'S METHOD

A.   BACKGROUND

In this chapter we describe Petriu's method for computing the Automated Guided Vehicle's (AGV) position from a point of origin on a binary track. We also describe the hardware and software efficiencies of his method as well as its limitations. Petriu's scheme only requires a 1-bit wide code track to be physically laid on the floor. This is an attractive alternative for absolute position measurement that requires a very long track. This method is used by high-resolution shaft encoders and Automated Guided Vehicles (AGVs) [Pe2]. He uses a pseudorandom binary sequence (PRBS) as the basis for the track that the AGV will follow on the floor. The PBRS is a maximal length linear sequence of length $2^n - 1$ generated by a linear shift register (see Appendix A for more details). If we scan every possible n-tuple in this sequence with a window of length n, we see that every possible n-tuple (except the all 0 n-tuple) appears uniquely. If the AGV's position is designated by an n-tuple, then it can use this uniqueness property to determine its exact position on this track [Pe2].

B.   ABSOLUTE POSITION MEASUREMENT

There are different methods to convert these pseudorandom n-tuples into a natural code (the binary equivalent of the exact distance from a point of origin). One conversion method uses a strictly parallel solution by implementing a ROM-stored lookup Table. This can increase hardware costs for large values of n [Pe2]. At the other extreme is a strictly serial solution by using a reverse feedback shift register to count the number of reverse feedback shifts it takes to get to the "zero position" code pattern. This is equivalent to counting the number of bits the n-tuple is from the origin. Although the hardware costs are not as high, it becomes prohibitively time consuming as n gets large [Pe1]. Petriu uses a combination of the parallel and serial methods to achieve a more economic approach. This approach implements a set of evenly spaced n-tuples, called milestones, where each milestone and its distance from the point of origin (designated to be the MSB of the PBRS), is paired with a binary equivalent of the distance of the milestone's MSB from the point of origin [Pe2].

17

In the PBRS, we have a total of $2^n - 1$ positions, or n-tuples, that the AGV can occupy (the PBRS does not include the n-tuple consisting of all zeros). See Appendix A for further details. Since the milestones are uniformly distributed with a period of t bits, the total number of milestones needed is

$$w = \left\lceil (2^{n-1})/t \right\rceil \tag{2.1}$$

Let $p = m*t + r$ designate the position of any n-tuple where $m*t$ represents the position of the nearest "down the track" milestone, Q(m), and r represents the relative distance between this milestone and the n-tuple representing the AGV's initial position [Pe2].

Petriu uses a sequential algorithm for the code conversion of the relative distance r and a parallel code conversion method for the milestone position $m*t$. The sequential algorithm counts the number of steps needed for the AGV to move from its initial position toward the origin (p=0) until it reaches the MSB of a milestone. The parallel code conversion scheme compares each unique n-tuple encountered during this "back stepping" against all possible milestones to see if the binary patterns match [Pe2]. The parallel "milestone" recognition method can be implemented using a field-programmable logic array (FPLA) since it has n inputs, $n+1$ outputs and w products. This is seen in the schematic below [Pe3]:
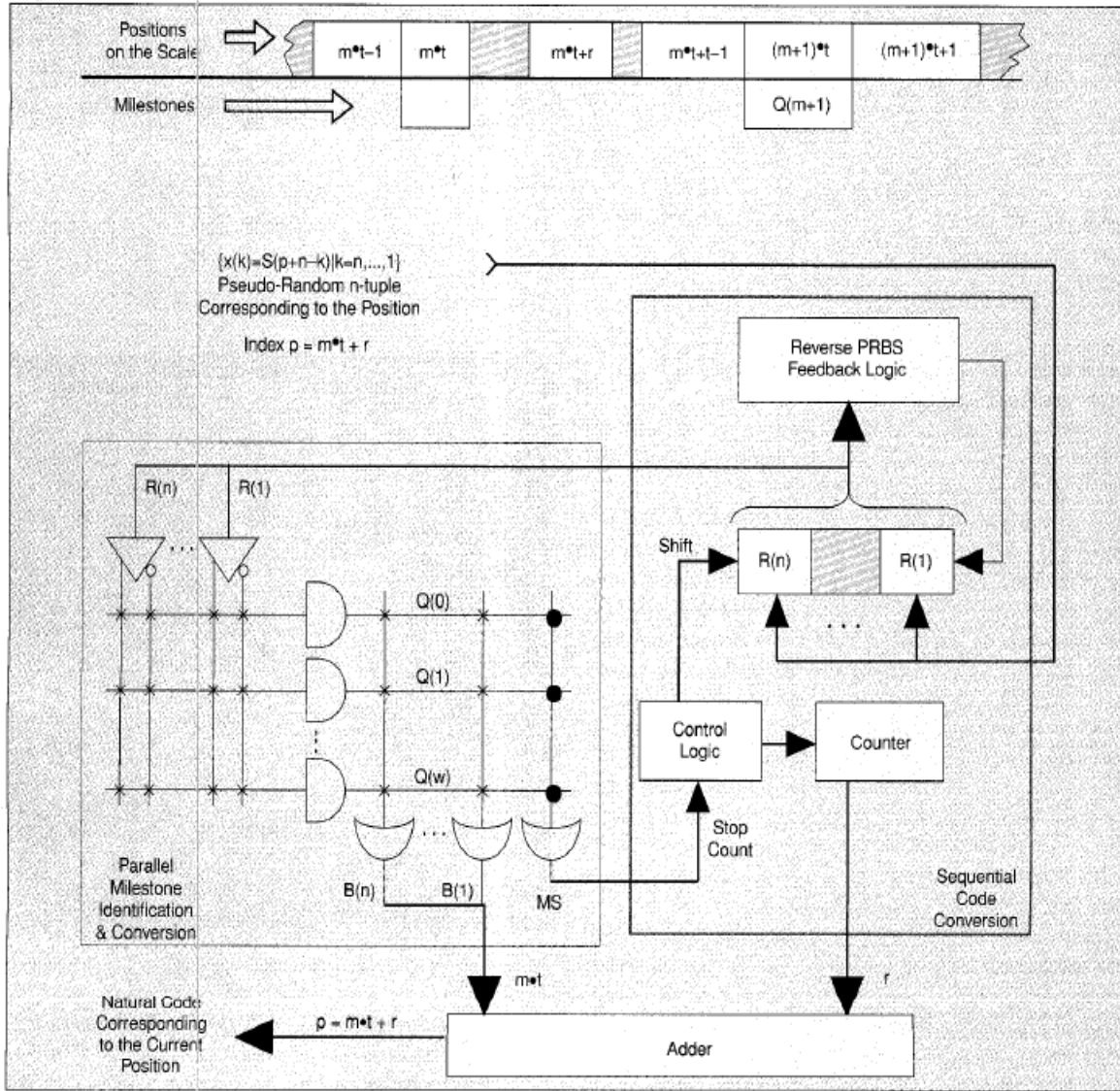
Figure 3.   Serial-parallel PRBS to Natural Code Conversion (From: [Pe3])

The outputs B(n), B(n-1),…,B(1) yield the natural binary code for $m*t$ which is the position of the recognized milestone Q(m). The output MS signals the control logic that a milestone as been detected. The control logic then stops the sequence of "back shifts" in the sequential code conversion method. The AGV stops traveling and computes the natural binary code of its position p by adding the binary representations of r and $m*t$ using an n-bit adder [Pe2].

For example, let n=5. Then the PBRS track is $2^5 - 1 = 31$ bits long. If the period is $t = 8$, then the number of milestones is $w = \lceil (2^{5-1})/8 \rceil = 4$. The Figure below illustrates this clearly [Pe2].

PSEUDORANDOM CODE TRACK
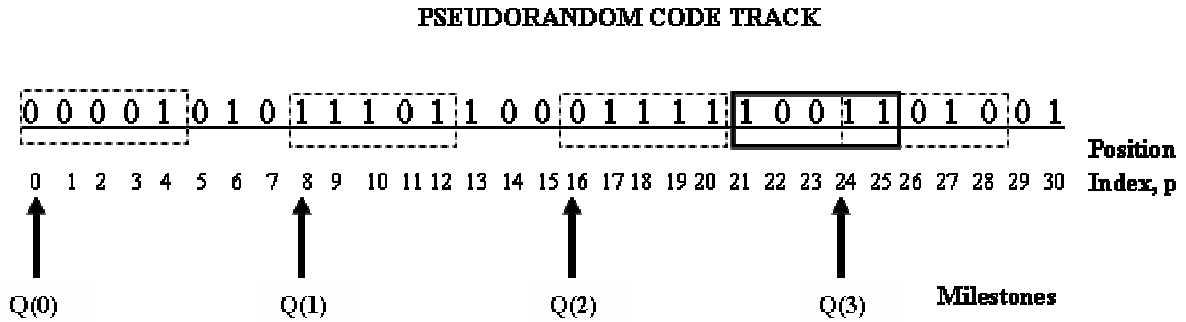
Figure 4.   PBRS Track with Milestones (From: [Pe2])

Each $Q(i)\,(0 \le i \le 3)$ (boxed in with dashed lines) is a milestone and is associated with the following pseudorandom / natural code conversion scheme:

Milestone	Natural code position	Code decimal equivalent
00001	00000	0
11101	01000	8
01111	10000	16
11010	11000	24

Table 9.   Milestone-to-Natural Code Conversion (From [Pe2])

Assume the AGV's initial position is 10011 (boxed in with solid lines). The AGV begins to travel one bit at a time towards position $p = 0$, comparing each n-tuple in parallel with all milestones. Once it reaches the milestone 01111, it obtains the position of this milestone to be $m * t = 16$, where the natural binary code for this is 10000 (as seen in the Table above). Since the AGV needed to travel $r = 5 = 00101_2$ bits to reach this

milestone, the absolute position of this n-tuple $p = m * t + r = 10000_2 + 00101_2 = 10101_2 = $ 21 bits which corresponds to the location of the n-tuple in the PBRS above.

C. PERFORMANCE COSTS

The performance cost of the previous example can be estimated as follows:

Hardware cost: 4 words x 5 bits

Time cost: 7 clock periods

This cost is efficient when compared to a strictly parallel approach (31 words x 5 bits) or a strictly sequential approach (31 clock periods) [Pe2]. This comparison is demonstrated graphically below [Pe1]:
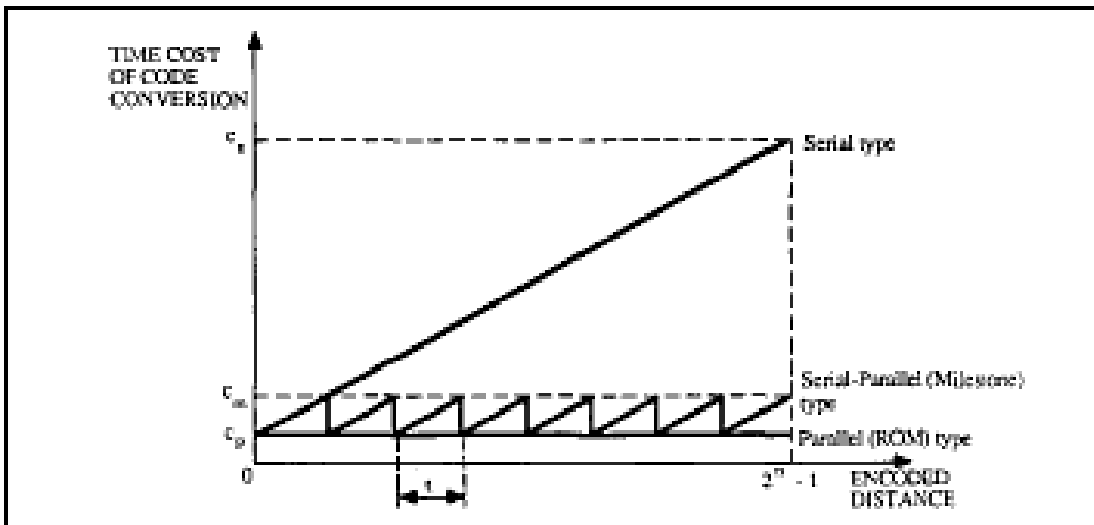
Figure 5.   Relative Time Performance Of Different Pseudorandom / Natural Code
Conversion Methods (From [Pe1])

The following are equipment and temporal cost equations for the serial-parallel method employed by Petriu [Pe1]:

$$Equipment\ Cost = k_1 \bullet number\ of\ milestones + k_2$$
$$= k_1 \bullet \left\lceil (2^{n-1})/t \right\rceil + k_2$$

(2.2)

21

$$Temporal\ Cost = k_3 + k_4 \bullet t \qquad\qquad\qquad (2.3)$$

$$Total\ Cost = Equipment\ Cost + Temporal\ Cost = k_1 \bullet \left\lceil (2^{n-1})/t \right\rceil + k_2 + k_3 + k_4 \bullet t \quad (2.4)$$

where

k_1 = equipment cost associated with each milestone

k_2 = basal equipment cost for the serial back shift operations

k_3 = basal temporal cost for a fully parallel solution

k_4 = temporal cost associated with each back shift operation

and k_1, k_2, k_3 and k_4 are constant for a given n and are functions of the technology being used [Pe1]. The following graph illustrates this relationship:
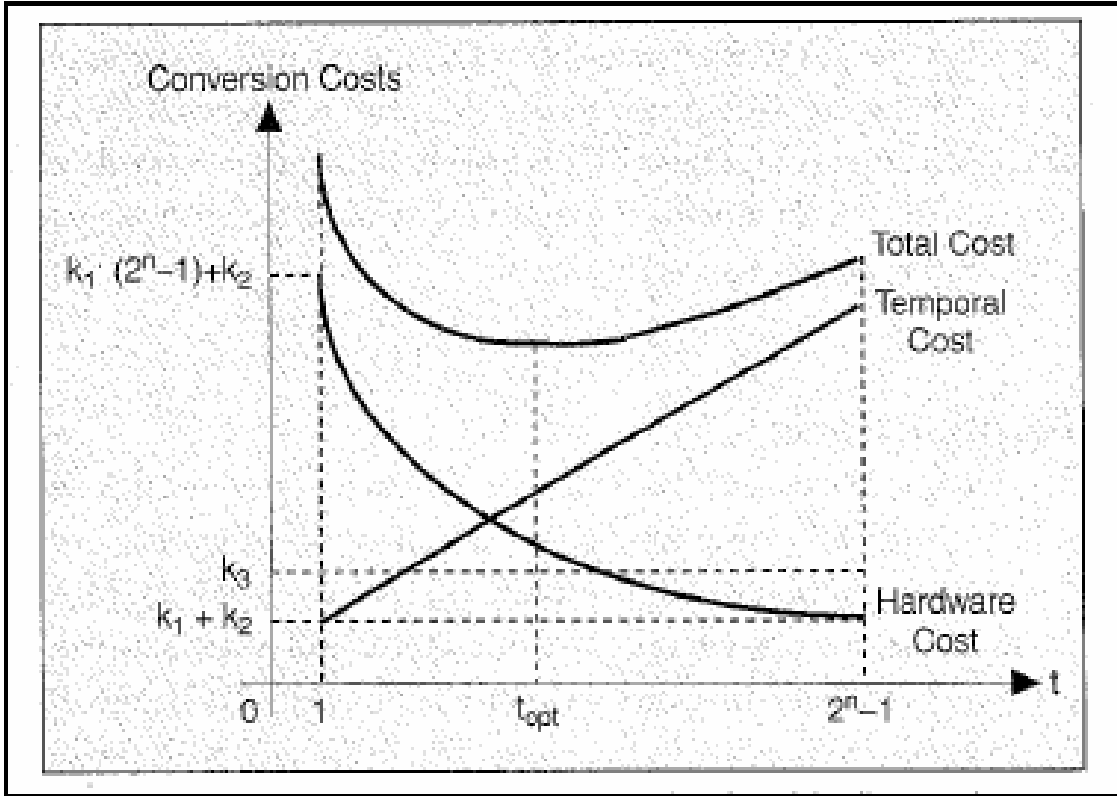
Figure 6.   Serial-Parallel Code Conversion Costs as a Function of Distance (From [Pe1])

The total cost has a minimum at t_{opt} and can be found by using the following formula:

$$t_{opt} = \left\lceil \left[(k_1/k_4) \bullet (2^{n-1}) \right]^{\frac{1}{2}} \right\rceil \qquad (2.5)$$

the optimal distance, t_{opt}, between milestones depending on the values of k_1 and k_4.

More parallelism is needed as the temporal cost begins to exceed the hardware costs

$k_4 > k_1$. As the measuring resolution increases, we will need more milestones to maintain

the same code conversion speed [Pe1].

THIS PAGE INTENTIONALLY LEFT BLANK

III.   GREEDY DEBRUIJN SEQUENCE (GDBS) APPROACH AND RESULTS

A.   BACKGROUND

In Petriu's scheme, the AGV monitors its position on a binary track that is physically laid out on the floor. This track is modeled after a Pseudo-Random Binary Sequence (PRBS) that is generated using a linear shift register. Our scheme also uses a binary track laid out on the floor that is encoded using the greedy DeBruijn sequence. This sequence can be constructed from the concatenation of lexicographically ordered Lyndon words generated by the Necklace algorithm. The objective of this chapter is to demonstrate an alternate scheme of measuring the absolute position of any Automated Guided Vehicle (AGV) whose position is given by an n-tuple on a greedy DeBruijn sequence.

There are two ways to measure the absolute position of an n-tuple depending on whether the position is above or below a threshold. The first method applies a direct computation for those n-tuples which are Lyndon words or which lie between two adjacent necklaces that are equal to or greater than the threshold. Since there are no missing even decimals above the threshold, we can directly calculate the position of any n-tuple above the threshold without any need of signposts or without running the Necklace Algorithm. The second method uses lexicographically ordered signposts that serve a similar function to Petriu's milestones. These signposts are evenly distributed throughout all the necklaces below the threshold. The signpost data are $2n$ bits long where the first n bits correspond to the necklace designated as a signpost and the last n bits indicate the binary equivalent of the numerical value of the distance from a point of origin. We define the LSB of the greedy DeBruijn sequence whose position number is designated as "1"as the LSB (or the rightmost bit) of the 000…0 n-tuple.

B.   ABSOLUTE POSITION MEASUREMENT ABOVE THRESHOLD

In our approach, an AGV travels either to the left or right tracking at most n bits until the AGV's position matches that of a necklace. We call a nearby necklace the *position of reference (POR).* While the AGV moves towards the POR it tallies the number of bits it has traveled. The subroutine below, called testforNecklace, is

embedded in another routine that searches the vicinity of the original n-tuple and tests each new *n*-tuple as the AGV moves to see if it is a necklace. The routine stops executing when testforNecklace returns a true Boolean value indicating a necklace was found.

```
bool Necklaces::testforNecklace(long long bininput);
{
bool foundnecklace = false;
long long ndecimal = bininput; // binary decimal associated with n-tuple
nstring = decToBin(ndecimal); // subroutine that converts decimal  value into a binary
                                // sequence
shiftSequenceToNecklace(); // subroutine that shifts a binary sequence into its necklace
                                // representative
shiftdecimal = binToDec();   // subroutine that converts a binary string into its decimal
                                // equivalent
if(shiftdecimal = = ndecimal)  // test to see if current n-tuple is a necklace
{
  foundnecklace = true;
}
else
{
  foundnecklace = false;
}
return foundnecklace;
// end testforNecklace
```

The key subroutine in testforNecklace is shiftSequenceToNecklace. Further details of the coding implementation can be found in Appendix B.

The absolute distance of any n-tuple above the threshold is given by:

$$D = 2^n - n * \left[\frac{2^n - n_d}{2}\right] + \sum_{i=1}^{k-1}(n - p_i) \pm b, \qquad (3.1)$$

where D is the distance of the most significant (leftmost) bit of the n-tuple from the point of origin (including the point of origin), n_d is the decimal value of the POR, p_i is the period of the ith necklace generated by the Necklace Algorithm and b is the number of bits traveled to reach the necklace (where $-b$ represents the AGV moving b bits to the left to reach a POR). For example, given the greedy DeBruijn sequence for $n = 7$

1^1111110^1111100^1111010^1111000^1110<u>10^1110100</u>^1110010^<mark>1110000</mark>^1101100^
1101010^1101000^1100100^1100010^1100000^1010100^1010000^1001000^1000000^0

we determine the number of leading ones the threshold (highlighted above) to be

$p = \left\lfloor\frac{7-1}{2}\right\rfloor = \left\lfloor\frac{6}{2}\right\rfloor = 3$. Assume 1011101 (underlined above) is the AGV's initial position

(shown using an underline). According to the above subroutine, the decimal value of 1011101 is 93 when the point of origin is the LSB position of the 0000000 n-tuple in the above sequence. Shifting the string to its necklace representative

$$1011101 \rightarrow 1110110$$

and determining its decimal value we have $118 \neq 93$. So the AGV keeps moving to the left until the decimal values match. After moving five bits to the left it reaches the necklace 1110110, which is designated as the POR. Using the formula for D, we have the position of the 7-tuple 1011101 to be:

$$D = 2^7 - 7 * \left[\frac{2^7 - 118}{2}\right] + \sum_{i=1}^{5}(7 - p_i) - 5$$

$$D = 128 - 35 + 6 + 0 + 0 + 0 + 0 - 5 = 94$$

which corresponds to the location of the sequence above.

In the above example, we used $n = 7$, a prime number. When n is composite (e.g $n = 6$) we can see more clearly the effects of shortened periods. (See Table 10)

27

Necklace	Lyndon Word	Period	Decimal Equivalent
111111	**1**	**1**	**63**
111110	111110	6	62
111100	111100	6	60
111010	111010	6	58
111000	111000	6	56
110110	**110**	**3**	**54**
110100	110100	6	52
110010	110010	6	50
110000	110000	6	48
101010	**10**	**2**	**42**
101000	101000	6	40
100100	**100**	**3**	**36**
100000	100000	6	32
000000	**0**	**1**	**0**

Table 10.   Necklaces and Lyndon Words for the Greedy DeBruijn Sequence

We construct the greedy DeBruijn sequence given for $n = 6$:

1^111110^111100^111010^111000^**110**^110<u>100</u>^110010^<mark>110000</mark>^**10**^101000^**100**
^100000^**0**

The threshold has $p = \left\lfloor \dfrac{n-1}{2} \right\rfloor = \left\lfloor \dfrac{6-1}{2} \right\rfloor = \lfloor 2.5 \rfloor = 2$ leading ones and is highlighted in our

sequence. If the AGV's current position is 100110 and we move three bits to the right to
reach the POR 110010 we obtain

$$D = 2^6 - 6 * \left\lceil \frac{2^6 - 50}{2} \right\rceil + \sum_{i=1}^{7}(6 - p_i) + 3$$

$$D = 64 - 42 + 5 + 3 + 3 = 33.$$

This location is also confirmed by the above sequence.

28

It is not too difficult to determine the necklaces that have shortened periods for a given composite n before you run the Necklace Algorithm. First, determine the prime divisors of n. Partition the n-length sequence of all ones into subsequences whose lengths are the cofactors of these prime divisors (numbers resulting from the divisions by the prime divisors). We run the Necklace Algorithm on each of the partitions simultaneously until all of the necklaces have been generated for each of these identical partitions. Finally, we concatenate all identical partitions into a sequence of length n. This results in all the necklaces that have shortened periods.

For example, let n =12. The prime divisors of 12 are 2 and 3. Dividing 12 by 2 and 3 yields 6 and 4. Partitioning the 12-long sequence 111111111111 into the subsequences of lengths six and four yields 111111^111111 and 1111^1111^1111.   We then generate Table 11 for the cofactor $d = 6$ and Table 12 for the cofactor $d = 4$.

Necklace	Period	Decimal
111111^111111 → 111111111111	**1**	**4095**
111110^111110 → 111110111110	6	4030
111100^111100 → 111100111100	6	3900
111010^111010 → 111010111010	6	3770
111000^111000 → 111000111000	6	3640
110110^110110 → 10110110110	3	1462
110100^110100 → 110100110100	6	3380
110010^110010 → 110010110010	6	3250
110000^110000 → 110000110000	6	3120
101010^101010 → 101010101010	**2**	**2730**
101000^101000 →101000101000	6	2600
100100^100100 → 100100100100	3	2340
100000^100000 → 100000100000	6	2080
000000^000000 → 000000000000	**1**	**0**

Table 11.   Generating Shortened Necklaces for $d = 6$

Similarly,

Necklace	Period	Decimal
1111^1111^1111 → 111111111111	**1**	**4095**
1110^1110^1110 → 111011101110	4	3822
1100^1100^1100 → 110011001100	4	3276
1010^1010^1010→ 101010101010	**2**	**2730**
1000^1000^1000 → 100010001000	4	2184
0000^0000^0000 → 000000000000	**1**	**0**

Table 12.   Generating Shortened Necklaces for $d = 4$

Note that since 6 and 4 have a greatest common divisor (g.c.d.) of 2, they share the same common factors of 1 and 2. Thus, they also share the necklaces whose periods are one and two, namely: 111111111111, 000000000000 and 101010101010. All the other necklaces for the two divisors are distinct. Since 3 is a divisor of 6 and not of 4, all necklaces with shortened periods of length 3 are contained in the list for $d = 6$ as well as $d = 3$. Since all necklaces are lexicographically ordered, we know which of the necklaces with shortened periods will precede the POR so we can make the necessary adjustments in calculating the AGV's position. Of course, we can make things simpler by insisting that n be a prime number.

C.   SIGNPOST GENERATION BELOW THRESHOLD

When measuring the absolute position of an n-tuple below the threshold we have a more difficult situation.   Since there are "missing" even decimals to contend with as described in Chapter II we cannot apply a direct computation. We choose to select some of the necklaces below the threshold as "signposts" to contain embedded information about their distance from the origin. This procedure dominates absolute position measurement as n increases since the ratio of necklaces above the threshold to those below gets very small. This is confirmed through mathematical derivation and supported by experimental data illustrated in Table 16 and Figures 11 and 12.   Although these signposts are distributed evenly within the DeBruijn sequence below the threshold, each

class will contain a different number of signposts since each class contains a different number of necklaces. We describe a placement of signposts for $n = 7$ in Table 13.

Necklace	Lyndon Words	Class	Period	Decimal Equivalent
1 1 1 1 1 1 1	1	7	1	127
1 1 1 1 1 1 0	1 1 1 1 1 1 0	6	7	126
1 1 1 1 1 0 0	1 1 1 1 1 0 0	5	7	124
1 1 1 1 0 1 0	1 1 1 1 0 1 0	4	7	122
1 1 1 1 0 0 0	1 1 1 1 0 0 0	4	7	120
1 1 1 0 1 1 0	1 1 1 0 1 1 0	3	7	118
1 1 1 0 1 0 0	1 1 1 0 1 0 0	3	7	116
1 1 1 0 0 1 0	1 1 1 0 0 1 0	3	7	114
1 1 1 0 0 0 0	1 1 1 0 0 0 0	3	7	112
1 1 0 1 1 0 0	**1 1 0 1 1 0 0**	**2**	**7**	**108**
1 1 0 1 0 1 0	1 1 0 1 0 1 0	2	7	106
1 1 0 1 0 0 0	**1 1 0 1 0 0 0**	**2**	**7**	**104**
1 1 0 0 1 0 0	1 1 0 0 1 0 0	2	7	100
1 1 0 0 0 1 0	**1 1 0 0 0 1 0**	**2**	**7**	**98**
1 1 0 0 0 0 0	1 1 0 0 0 0 0	2	7	96
1 0 1 0 1 0 0	**1 0 1 0 1 0 0**	**1**	**7**	**84**
1 0 1 0 0 0 0	1 0 1 0 0 0 0	1	7	80
1 0 0 1 0 0 0	**1 0 0 1 0 0 0**	**1**	**7**	**72**
1 0 0 0 0 0 0	1 0 0 0 0 0 0	1	7	64
0 0 0 0 0 0 0	**0**	**0**	**1**	**0**

Table 13.   Necklaces and Signpost Insertion for n = 7

Below the threshold (highlighted row), there are only 11 remaining necklaces within which we insert signposts. If we arbitrarily designate six necklaces as signposts (highlighted in

bold), we see class 2 has three signposts, class 1 has two signposts and class 0 has one signpost (the all 0s n-tuple). Although we would not likely put the signposts in such close proximity when n is larger, this example serves to illustrate the point of equal placement. As n increases, the results are more dramatic and there are more necklaces between adjacent signposts.

For any n, we generate the signposts as we run the Necklace Algorithm to generate the necklaces. To find the number of signposts, N_s, we want to insert, we first need to calculate how many remaining necklaces, N_r, there are below the threshold. We use the following formula:

$$N_r = \left(Z_n - \left[\frac{(2^n - 2) - (2^n - 1) - (2^{n-p} - 1)}{2} + 1\right]\right)$$
$$= Z_n - 2^{n-p-1} - 1 \qquad (3.2)$$
$$= Z_n - N_T,$$

where $p = \left\lfloor \frac{n-1}{2} \right\rfloor$. Z_n indicates the total number of necklaces for a given n and

$N_T = 2^{n-p-1} + 1$ is the number of necklaces above and including the threshold. The number of signposts, N_s, is

$$N_s = \left\lceil \frac{N_r}{d+1} \right\rceil, \qquad (3.3)$$

where $d \le N_r$, $d \ne 0$ and d represents the number of necklaces we want the necklace algorithm to generate between signposts on the list. We define the distance between signposts as N_d. If n is prime, then $N_d = d*n$. Otherwise, since $N_d \le d*n$, we need to take into account necklaces that have periods less than n. In our example,

$p = \left\lfloor \frac{7-1}{2} \right\rfloor = \left\lfloor \frac{6}{2} \right\rfloor = 3$ and $N_r = Z_7 - 2^{7-3-1} - 1 = 11$ which agrees with the example above.

We choose $d = 1$ yielding a total of six signposts and the distance between signposts is $N_d = 1*7 = 7$ bits.

Recalling that we only concatenate the Lyndon words derived from the necklaces, we have the following greedy DeBruijn sequence for n=7:

1^1111110^1111100^1111010^1111000^1110110^1110100^1110010 ^1110000
^**1101100**^1101010^**1101000**^1100100^**1100010**^1100000^**1010100**^1010000
^**1001000**^1**000000**^0.

The sequence is of length $2^7 = 128$ with the signposts indicated in bold and the threshold highlighted. If we measure the point of origin from the zero bit at the LSB position of the DeBruijn sequence, then the number of bits needed to reach the MSB of our signpost is the absolute distance D. By beginning with a count of $2^7 = 128$, we subtract the period of each necklace we generate to determine this distance, convert it into binary, and concatenate it to the necklace. This is illustrated in the Table below:

Necklace	Distance (bits)	Signpost
1101100	$71 = 1000111_2$	1101100^1000111
1101000	$57 = 0111001_2$	1101000^0111001
1100010	$43 = 0101011_2$	1100010^0101011
1010100	$29 = 0011101_2$	1010100^0011101
1001000	$15 = 0001111_2$	1001000^0001111
0000000	$7 = 0000111_2$	0000000^0000111

Table 14.   Signposts for n = 7

The signpost information has length $2n$, since it is a concatenation of the necklace designated as a signpost and the binary equivalent of its numerical distance from the point of origin. After we generate the signposts, the AGV can store them in its memory to use in determining its location. Like Petriu's scheme, the AGV will find itself in an arbitrary position designated by a unique n-tuple. In Petriu's implementation, the AGV must travel back towards the origin one bit at a time, using the window property of the PRBS to see if each n-tuple encountered matches a milestone. Since Petriu simultaneously uses a serial and a parallel operation to detect his milestones, this is an O(m) x O(l) operation, where *m* represents the distance between the *n*-tuple and the

33

milestone and *l* represents the number of milestones. Since the number of bits between milestones grows significantly as *n* increases, this can take a while and the work grows accordingly.

Our signposts contain the same information as Petriu's milestones. The main difference in the two schemes is that our signposts are lexicographically ordered, so their relative spatial relationship to each other is clear. The same cannot be said about Petriu's milestones unless one performs the pseudorandom to natural code conversion described in Chapter III.

Unlike in Petriu's scheme, in our scheme each *n*-tuple examined is not compared with each signpost until a match is found. Instead, we search for a POR to compare against signposts so we can determine between which signposts the POR resides. Since the signposts are lexicographically ordered and organized by class number, we can narrow the search to a smaller subset of the list of signposts. As we compare the POR to our signposts, we keep track of the current and the last signpost we encounter on our list. When the first signpost is found whose decimal value is smaller than that of our POR, we stop the search. The AGV then runs the Necklace Algorithm from the last signpost whose decimal value is greater than the POR's to the POR while simultaneously computing how many bits lie between them. This is accomplished by determining the period of each necklace generated and then summing the periods. This number is subtracted from the decimal equivalent of the distance information in the signpost. If the AGV needed to travel to the left to reach the POR, then we would subtract the number of bits traveled. Otherwise, we add them. We summarize this in the following formula:

$$D = D_j - \sum_{i=j}^{m} p_i \pm b \tag{3.4}$$

D is the absolute distance of the n-tuple, D_j is the distance of the jth signpost from the origin, p_i is the period of a necklace between the jth signpost and the mth necklace that precedes the POR, and *b* is the number of bits traveled to reach the POR.

For example, given the DeBruijn sequence for n=7,

34

1^1111110^1111100^1111010^1111000^1110110^1110100^111 0010^<mark>1110000</mark>
^**1101100**^1101010^**1101000**^1100100^**1100010**^1100000^**1010100**^1010000^**1001000**
^1**000000^0**

If the AGV's initial position is below the threshold, for example 0011001 (underlined above), we need to use the distance information contained in the signposts. If we choose to move to the left five bits, the AGV's designated position will be the POR 1101000. Since this POR is already a signpost, we do need not run the Necklace Algorithm to find the separation distance between a signpost and the POR. Using the formula and the distance corresponding to 110100 from the Table above,

$$D = 57 - 0 - 5 = 52 \text{ bits from the point of origin}$$

There are no necklaces between the signpost and the POR (since they are the same) and the AGV needed to move five bits to the left. This matches the count we would obtain from the above sequence. If the POR is not a signpost, we need to search through all the signposts in the same class as our POR to find out between which signposts our POR resided. We then run the Necklace Algorithm to get the information needed to compute the absolute position.

For example, assume we only choose to insert four signposts (indicated in bold) in our sequence below:

1^1111110^1111100^1111010^1111000^1110110^1110100^111 0010^<mark>1110000</mark>
^**1101100**^1101010^1101000^**1100100**^1100010^1100000^**1010100**^1010000
^1001000^**1000000**^0

If we have the same POR as before, 1101000 is not a signpost. We compare this necklace to the list of signposts below

<div align="center">

1101100

1100100

</div>

1010100

1000000

and we see that $1101100 < 1101000 < 1100100$. Running the Necklace Algorithm from 1101100 to 1101000, we find that they are 14 bits apart. Using formula 3.4

$$D = D_j - \sum_{i=j}^{m} p_i \pm b$$

$$D = 71 - (7 + 7) - 5 = 71 - 19 = 52 \text{ bits from the point of origin}$$

This corresponds with the previous result we had for this n-tuple.

With more than one AGV on the DeBruijn track, we initially space them uniformly apart where their initial positions correspond to a signpost. Then, as each AGV begins to move we update their current locations by adding new signposts to the list or by changing the identities of the signposts. If the AGVs are not initially on signposts and not spaced evenly, we could move each of them to their respective PORs and easily find out how far apart they are from each other. Even without knowing the exact location of each POR, we can have a sense of where each AGV is on the DeBruijn track and where they are relative to each other since the PORs are lexicographically ordered. (A higher decimal value of the POR corresponds to a longer distance from the point of origin). This also leads to an optimized solution on the AGV placement since we can calculate (without moving them beyond the POR) which one is closest to a particular signpost or AGV. This is not possible with Petriu's milestones, since in his formulation the milestones are not lexicographically ordered with respect to their distance from the point of origin. In addition, an AGV on Petriu's track needs to move much further than n bits (when n is sufficiently large) to find its location to get a sense of how far it is from the point of origin. The problem becomes more expensive with more than one AGV on the track for a sufficiently large value of n.

D.   PERFORMANCE EVALUATION

Petriu uses a pseudorandom-to-natural code conversion with his milestones. Depending on the value of n that is used to generate the PRBS of length $2^n - 1$, the number of milestones can become large, or if we limit the number of milestones, the

distance between adjacent milestones can become large. Figures 7 and 8 indicate how the value of t_{opt}, the distance between milestones and w, the number of milestones, grows with the value of n.
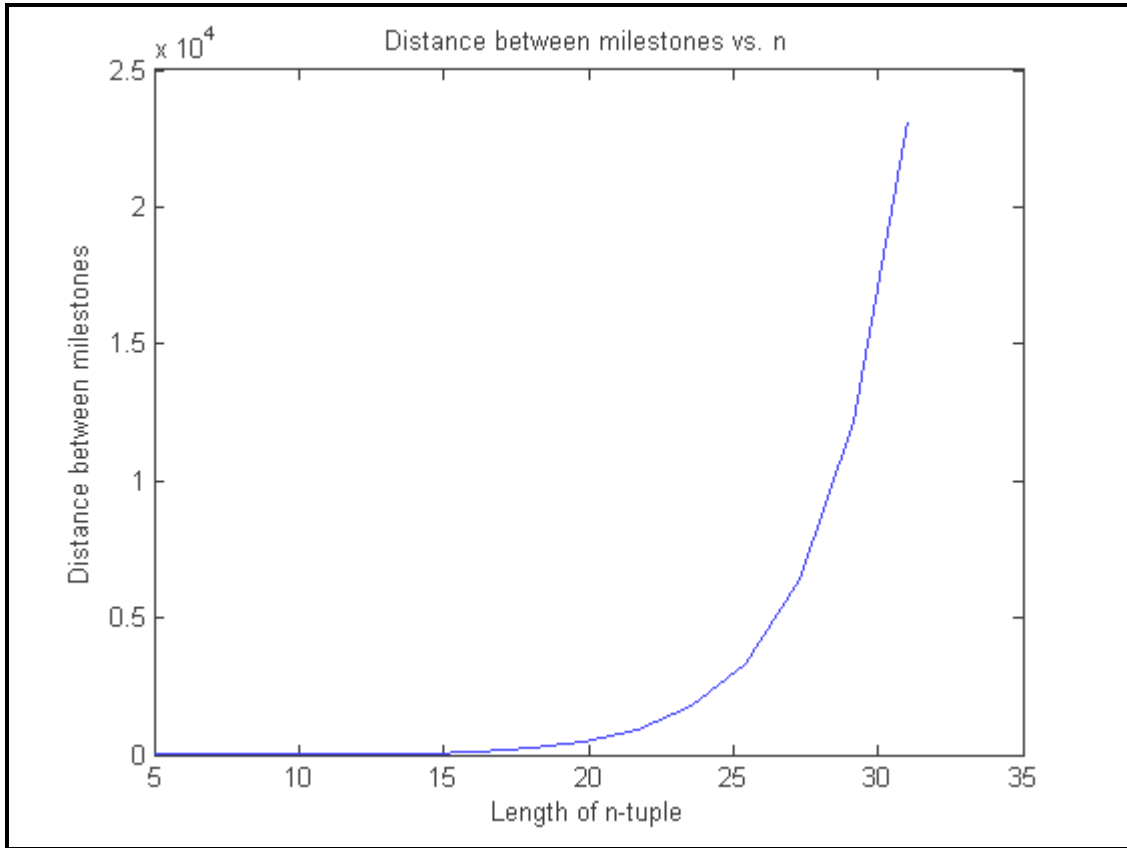
Figure 7.   Distance between Milestones vs. n
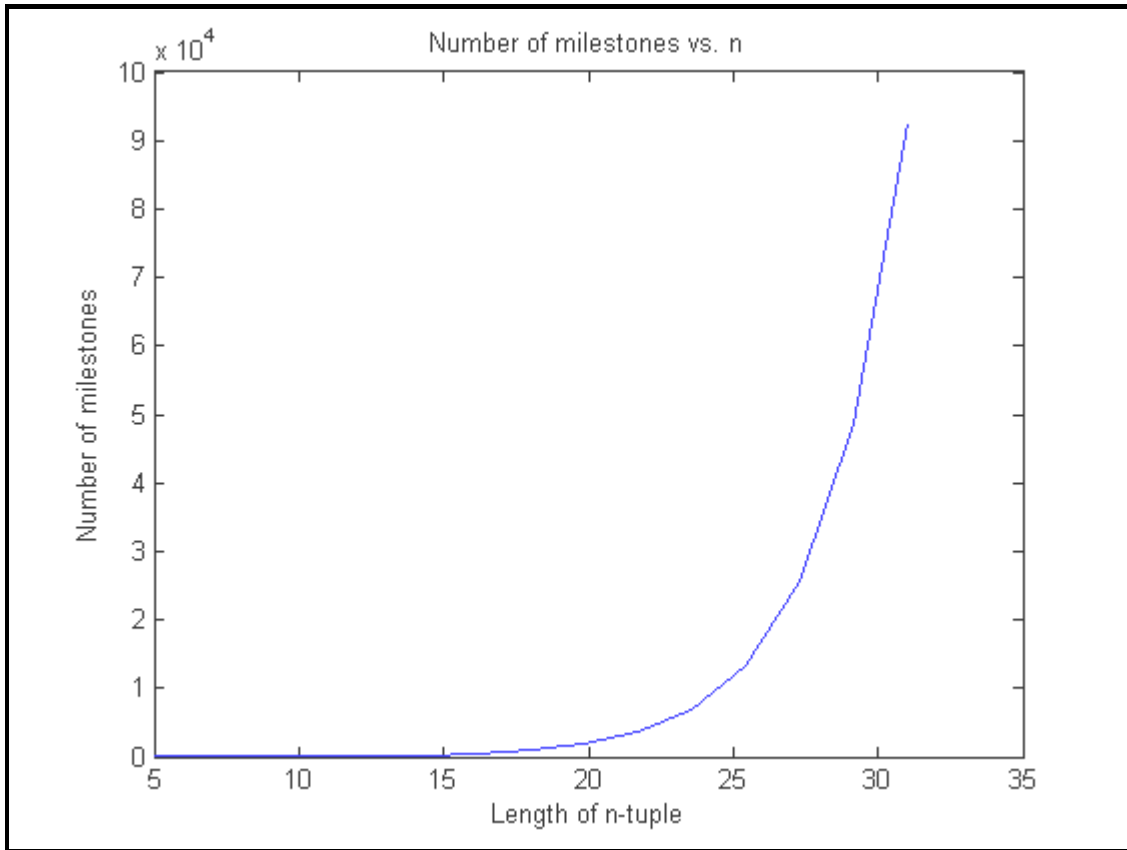
Figure 8.   Number of Milestones vs. *n*

Figures 9 and 10 compare the number of signposts versus *n* and the distance between signposts versus *n*.
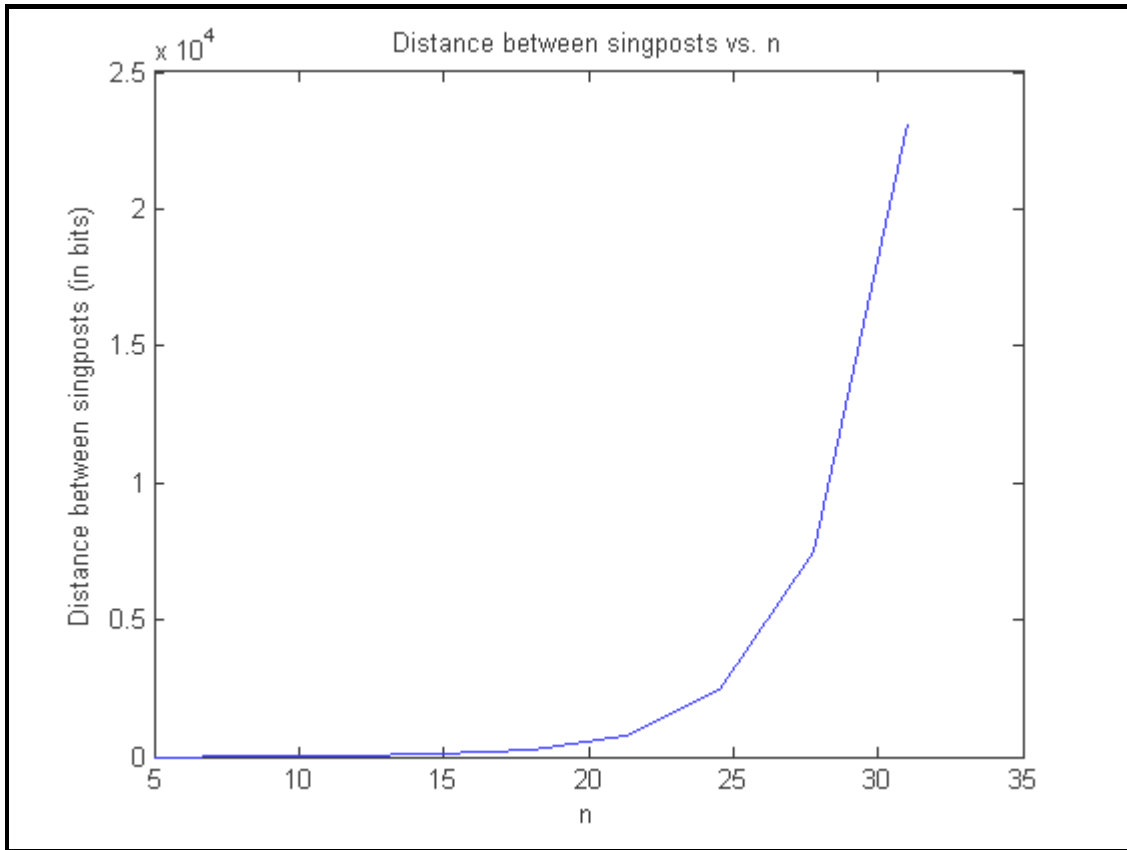
Figure 9.   Distance between Signposts vs. *n*
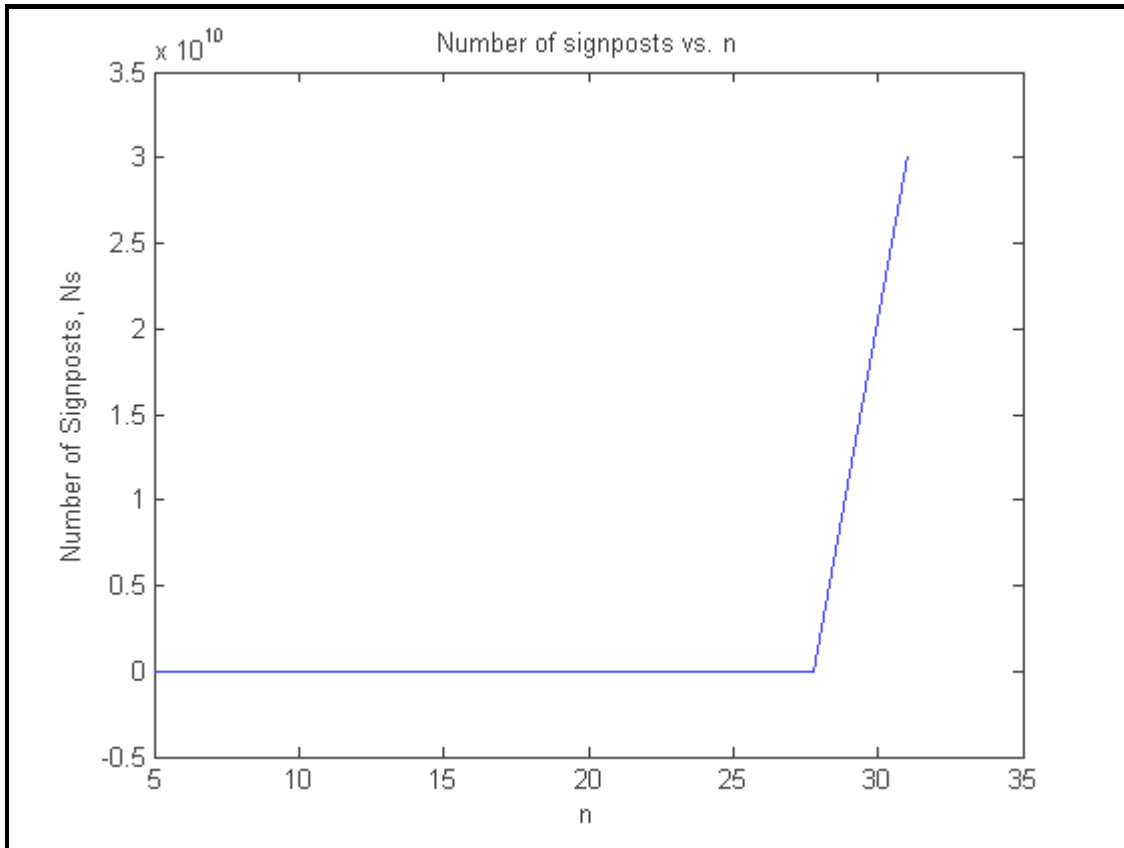
Figure 10.   Number of Signposts vs. *n*

Petriu's milestones are fixed once they are chosen for a given *n* as they are designed into the hardware. Our signposts are stored in memory and, each time an AGV moves to a new location, we can determine a new POR and add a new signpost among existing signposts while the AGVs are in operation. We have the flexibility of adding or removing as many signposts as we want or changing which POR we choose to be a signpost. Table 15 compares the number of signposts and their distances between them with Petriu's statistics.

Greedy DeBruijn Sequence Track							
n	**5**	**7**	**8**	**16**	**19**	**24**	**31**	**50**
2^n	32	128	256	65,536	524,288	16,777,216	2,147,483,648	1.12589990684262e+015
Z_n	8	20	36	4116	27,596	699,252	69,273,668	22,517,998,808,028
N_r	3	11	19	3859	27,083	695,155	69,240,899	22,517,965,253,595
N_T	5	9	17	257	513	4097	32,769	33,554,433
d	1	1	1	8	20	86	748	335,545
D_d	0	0	0	112	361	2040	23,157	16,777,200
N_s	3	11	19	483	1355	8084	92,569	67,108,631
Petriu's Track							
t_{opt}	3	6	8	128	363	2048	23,171	16,777,216
w	4	10	16	456	1373	8098	92,556	67,108,665

Table 15.   Statistics on GDBS vs. PRBS Performance

$D_d = \lceil (d-1)*n \rceil$ refers to the distance between the LSB of one necklace to the MSB of

the adjacent necklace. The variable $d = \left\lceil \dfrac{t_{opt}}{n} \right\rceil$ is chosen so that every d^{th} necklace could

be selected as a signpost so that the distances between necklaces would match Petriu's

optimal distances. Both approaches (highlighted above) perform about the same. The

number of signposts, $N_s = \left\lceil \dfrac{N_r}{d} \right\rceil$, is about the same as the number of milestones, w.

We restrict our signpost selection to those necklaces below the threshold. However, the

percentage of necklaces, $\% N_T$, above the threshold decreases rapidly as n becomes large.

Given $N_r = Z_n - 2^{n-p-1} - 1 = Z_n - N_T$, then

$$\frac{N_r}{Z_n} = \frac{Z_n - N_T}{Z_n} = 1 - \frac{N_T}{Z_n} = 1 - \frac{2^{n-p-1}+1}{\dfrac{2^n}{n}} = 1 - \left(\frac{n}{2^{\frac{n-1}{2}}} + \frac{n}{2^n} \right),$$

41

since $p = \left\lfloor \dfrac{n-1}{2} \right\rfloor \approx \dfrac{n-1}{2}$, and $Z_n \sim \dfrac{2^n}{n}$. Then as $n \to \infty$, $2^n \gg n$ and $\dfrac{N_r}{Z_n} \to 1$.

Since $N_T = Z_n - N_r$, we have $N_T \to 0$ as $n \to \infty$. This is also verified experimentally in Table 16 and in Figures 11 and 12.

n	5	7	8	16	19	24	31	50
$\% N_r$	37.5	55	52.78	93.76	98.14	99.414	99.95	99.99985
$\% N_T$	62.5	45	47.22	6.24	1.86	0.586	0.05	0.00015

Table 16. $\%N_r$ and $\%N_T$ vs. n
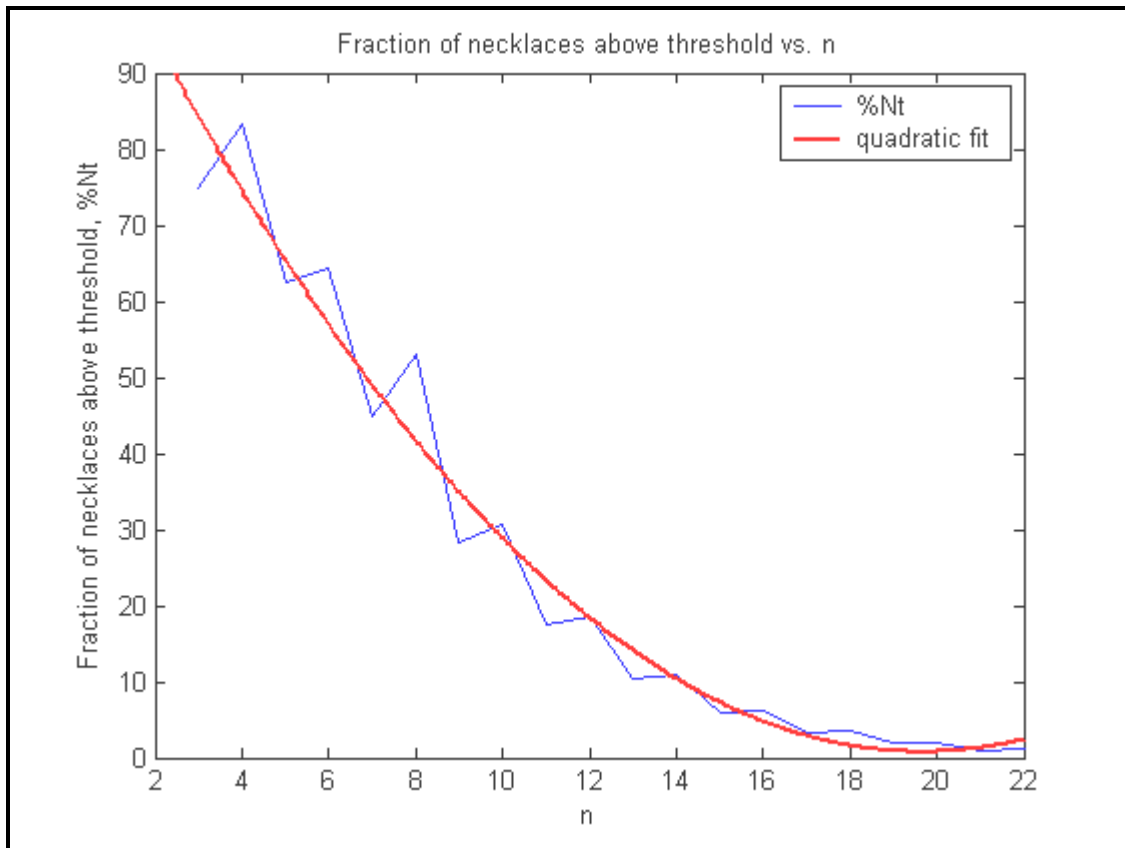
Figure 11. $\%N_t$ vs. n
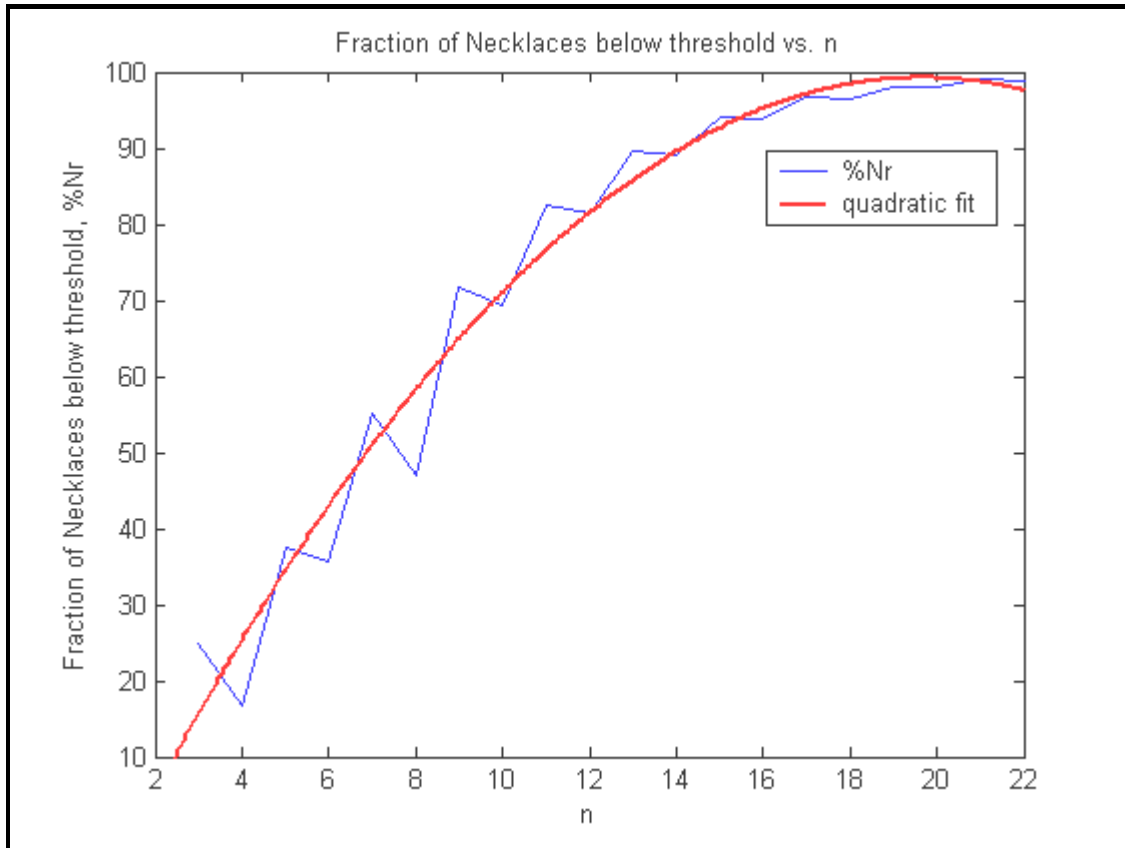
Figure 12.   %N_r vs. *n*

This explains why the performance between Petiu's approach and ours are similar. Our biggest savings occur during the AGV's movement to its POR to test whether or not each n-tuple examined is a necklace. The number of comparisons is at most *n*, while in Petriu's case, for *n* = 50, there is a potential of 67,108,665 comparisons. As mentioned previously, the AGV on Petriu's track needs to travel at most 16,777,216 bits in order to know its location. The AGV on our DeBruijn track would need to travel at most 50 bits. Of course, if the AGV does not find a signpost at the POR, then more computation is needed.   If our POR lies between two signposts, for *n* = 50 the number of necklaces between adjacent signposts is *d* = 335,545 necklaces. Using the Necklace Algorithm to generate this number of necklaces for *n* = 50 is still less work than generating necklaces for *n* = 30 (since the number of total necklaces for *n* = 30 is 364,724). Once we have our POR, we know in which class to begin our search. The

number of comparisons needed for our necklace will be less than 67,108,631 since all of the signposts are spread out among $p - 1 = \left\lfloor \dfrac{50 - 1}{2} \right\rfloor - 1 = 23$ classes. Assuming, as a baseline measure, that these signposts were evenly distributed among these classes we would have $\left\lceil \dfrac{67,108,631}{23} \right\rceil = 2,917,767$ signposts over which to compare our necklace. In reality, some classes have more signposts than others. In Petriu's scheme, comparisons against the milestones occur numerous times while in our scheme comparisons against our signposts happens only once.

E.   SHIFTING ONES SIGNPOSTS

A way to measure the effectiveness of the signpost scheme is to measure the average distance between the signposts and where along the greedy DeBruijn track they are distributed. This gives an idea of how much computation the AGV will need to perform in order to locate its POR within a list of signposts. An alternate technique to generate signposts is termed the shifting ones signposts. The idea is to see if a different distribution of signposts throughout the DeBruijn track below the threshold could produce better results than evenly distributing them. We compare the average distance between the shifting ones signposts with the distance between evenly spaced signposts.

First, we select the last necklace associated with a given class number. For example, with $n = 5$, 11000 is the last necklace associated with class 2. We then shift the "1" associated with the largest index one space to the right, each time checking that we still have a necklace. These necklaces will be associated with our signposts. For example:

1**1**000
10**1**00
100**1**0   Not a necklace
10000

For larger n, there are many necklaces that demonstrate this pattern but we only select those necklaces that are associated with a clump (a group of consecutive even

44

decimals that are "missing" between two adjacent necklaces). The reason for this is explained more clearly in Chapter IV. In the example above, if we select only those necklaces associated with "missing" decimal equivalents to be signposts, then the final list of signpost values is:

11000

10100

10000.

The value $n = 5$ is too small to make a big difference. However, for $n = 15$ when we run the Necklace Algorithm we obtain the following nine signposts out of a potential 97 necklaces which demonstrate this shifting ones pattern:

11000000000**1**00: clump size = 1 even decimal "missing"

10**1**000000000000: clump size = 1 even decimal "missing"

100**1**00000000000: clump size = 3 even decimals "missing"

1000**1**0000000000: clump size = 7 even decimals "missing"

10000**1**000000000: clump size = 7 even decimals "missing"

100000**1**00000000: clump size = 127 even decimals "missing"

1000000**1**0000000: clump size = 63 even decimals "missing"

100000000000000: c1ump size = 63 even decimals "missing"

The clump size refers to the number of "missing" decimal equivalents. These signposts are not evenly spaced apart as can be seen from the Table 17:

Necklaces	Distances Between Necklaces
	31,359
110000000000**1**00	
	1085
10**1**000000000000	

Necklaces	Distances Between Necklaces
	183
100**1**00000000000	
	60
1000**1**0000000000	
	20
10000**1**000000000	
	15
100000**1**00000000	
	15
1000000**1**0000000	
	15
100000000000000	
	15
000000000000000	

Table 17.   Distances between Shifting Ones Signposts (weight = 1) for n = 15

The average distance between signposts is 3640 bits and the first distance corresponds to the distance from the MSB of the DeBruijn sequence to the MSB of the first signpost. The distance between our evenly spaced signposts is 90 bits and we need 295 of them. The DeBruijn sequence itself is of length $2^{15} = 32,768$ bits. The beginning gap is 31,359 bits long so the percentage of the DeBruijn sequence covered by signposts is

$$\left(\frac{32,768 - 31,359}{32,768} \right) * 100 = 4.3\%$$

which is very inefficient since only a small percentage of the DeBruijn track is covered. If we decrease our searching range by increasing the weight to 2, we obtain 39 (containing a weight of one or two) signpost ts out of a potential 361 necklaces:

Signposts	Distances between Signposts
	23,336
1 1 1 0 0 0 0 0 0 0 0 **1 1** 0 0	
	4778
1 1 0 **1** 0 0 0 0 0 0 0 0 **1** 0 0	
	1620
1 1 0 0 **1** 0 0 0 0 0 0 0 **1** 0 0	
	725
1 1 0 0 0 **1** 0 0 0 0 0 0 **1** 0 0	
	375
1 1 0 0 0 0 **1** 0 0 0 0 0 **1** 0 0	
	45
1 1 0 0 0 0 0 **1 1** 0 0 0 0 0 0	
	165
1 1 0 0 0 0 0 **1** 0 0 0 0 **1** 0 0	
	120
1 1 0 0 0 0 0 0 **1** 0 0 0 **1** 0 0	
	75
1 1 0 0 0 0 0 0 0 **1** 0 0 **1** 0 0	
	45
1 1 0 0 0 0 0 0 0 0 **1** 0 **1** 0 0	
	45
1 1 0 0 0 0 0 0 0 0 0 **1** 0 **1** 0	
	30
1 1 0 0 0 0 0 0 0 0 0 0 0 **1** 0 0	
	525
1 0 **1** 0 **1** 0 0 0 0 0 0 0 0 0 0	
	230
1 0 **1** 0 0 **1** 0 0 0 0 0 0 0 0 0	

Signposts	Distances between Signposts
	120
1 0 **1** 0 0 0 **1** 0 0 0 0 0 0 0	
	75
1 0 **1** 0 0 0 0 **1** 0 0 0 0 0 0	
	45
1 0 **1** 0 0 0 0 0 **1** 0 0 0 0 0	
	30
1 0 **1** 0 0 0 0 0 0 **1** 0 0 0 0	
	15
1 0 **1** 0 0 0 0 0 0 0 **1** 0 0 0 0	
	15
1 0 **1** 0 0 0 0 0 0 0 0 **1** 0 0 0	
	15
1 0 **1** 0 0 0 0 0 0 0 0 0 **1** 0 0	
	15
1 0 **1** 0 0 0 0 0 0 0 0 0 0 0 0	
	63
1 0 0 **1** 0 0 **1** 0 0 0 0 0 0 0 0	
	45
1 0 0 **1** 0 0 0 **1** 0 0 0 0 0 0 0	
	15
1 0 0 **1** 0 0 0 0 **1** 0 0 0 0 0 0	
	15
1 0 0 **1** 0 0 0 0 0 **1** 0 0 0 0 0	
	15
1 0 0 **1** 0 0 0 0 0 0 **1** 0 0 0 0	
	15
1 0 0 **1** 0 0 0 0 0 0 0 **1** 0 0 0	

Signposts	Distances between Signposts
	15
1 0 0 **1** 0 0 0 0 0 0 0 0 0 0 0	
	15
1 0 0 0 **1** 0 0 0 **1** 0 0 0 0 0 0	
	15
1 0 0 0 **1** 0 0 0 0 **1** 0 0 0 0 0	
	15
1 0 0 0 **1** 0 0 0 0 0 **1** 0 0 0 0	
	15
1 0 0 0 **1** 0 0 0 0 0 0 0 0 0 0	
	15
1 0 0 0 0 **1** 0 0 0 0 **1** 0 0 0 0	
	5
1 0 0 0 0 **1** 0 0 0 0 0 0 0 0 0	
	15
1 0 0 0 0 0 **1** 0 0 0 0 0 0 0 0	
	15
1 0 0 0 0 0 0 **1** 0 0 0 0 0 0 0	
	15
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0	
	15
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	

Table 18.   Distances between Shifting Ones Signposts (weight = 2) for n = 15

The average distance is 840 bits. The DeBruijn sequence covered by signposts is

$$\left(\frac{32,768 - 23,336}{32,768} \right) *100 = 28.78\% \, ,$$

which is a significant improvement.   However, the percentage of the

DeBruijn sequence covered by equally spaced signposts is

$$\left(\frac{2^n - n * N_T}{2^n} \right) * 100 .$$

(3.5)

For n = 15, we have

$$\left(\frac{2^{15} - 15 * 129}{2^{15}} \right) * 100 = 94.10\% ,$$

which is much more efficient. The following Table summarizes the performance of the shifting ones signposts as the weight is increased:

Weight	Distance to first signpost (bits)	Number of Signposts	% of DB sequence
1	31,359	9	4.3
2	23,336	39	28.78
3	13,701	116	58.19
4	6976	244	78.71
5	2866	370	91.25
6	1921	432	94.14
7	1921	447	94.14
8	1921	450	94.14

Table 19.   Number of Shifting Ones Signposts vs. % of DeBruijn Sequence for Various Weights

The gap between the first signpost and the beginning of the DeBruijn sequence levels off to 1921 bits yieldig the highest efficiency of 94% with at least 432 shifting ones signposts. Evenly spacing the signposts throughout the DeBruijn sequence achieves the same efficiency but with 295 signposts. Clearly, the evenly spaced signpost scheme has a better performance.

IV.   CONSIDERATIONS AND FUTURE WORK

A.   BACKGROUND

In Petrui's scheme, the AGV needs to examine every n-tuple sequentially until it reaches a milestone. There is no pattern in the sequence it traverses. In our scheme, the AGV needs to initially examine at most n bits sequentially, and then it can examine n bits simultaneously as it compares the POR to the signposts because of the pattern inherent in the list of necklaces. Although our scheme is efficient in finding the absolute position of an n-tuple, we have attempted to improve on its efficiency by a couple of methods. The first method involves trying to find a pattern in the number of missing necklaces that allows us to calculate the number of "missings" in a given class for any n. The second method involves trying to map the necklaces of length n to a class of a larger group of necklaces of length $n+k$, where k is some class number for this larger group of necklaces. Rather than scan through the list of signposts one at a time to find the location of a POR, either method would allow us to make calculable leaps over a group of signposts and speed up the search problem. Determining a method of computing the position of an n-tuple in a greedy DeBruijn sequence would eliminate the need to use the Necklace Algorithm to calculate the position. It would also eliminate the need for having a physically laid out binary track for the AGV to follow. The AGV would know the location of the n-tuple that designates its position without having to travel to its POR on the binary track. Although there are readily apparent patterns, finding a mathematical relationship to describe these patterns has been challenging and would be an appropriate topic for future work.

B.   RECURRENCE RELATION FOR MISSINGS

A linear recurrence relation is of the form $h_{n+i} = c_{n-1}h_{n+i-1} + ... + c_1 h_{i+1}$ where the c_i's $(1 \le i \le n-1)$ are integers and the h_i's $(1 \le i \le n)$ (in our presentation) are positive integers representing the number of "missings" for a given group of necklaces of length i. We attempt to predict the number of "missings" for a given value of n knowing the number of "missings" for smaller values of n.

We ran the Necklace Algorithm and collected and organized the following information on the number of "missings" per class for a given n. (The class number is determined by $k = n-q$).

q	n=2	n=4	n=6	n=8	n=10	n=12	n=14	n=16	n=18	n=20	n=22
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0
3		0	0	0	0	0	0	0	0	0	0
4		3	0	0	0	0	0	0	0	0	0
5			3	0	0	0	0	0	0	0	0
6			15	**5**	0	0	0	0	0	0	0
7				25	**5**	0	0	0	0	0	0
8				63	31	**5**	0	0	0	0	0
9					114	32	**5**	0	0	0	0
10					255	155	**33**	5	0	0	0
11						482	174	**33**	5	0	0
12						1023	719	180	**33**	5	0
13							1985	846	181	**33**	5
14							4095	3156	897	**182**	33
15								8050	3911	916	**182**
16								16383	13469	4256	922
17									32440	17501	4394
18									65535	56458	19531
19										130307	76561
20										262143	233726
21											522478
22											1048580

Table 20.   Number of "Missings" for Even n

Notice that as n increases the number of "missings" per class approaches a steady state value. From the Table above, our steady state values (highlighted in bold) are 5, 33 and 182.

q	n=1	n=3	n=5	n=7	n=9	n=11	n=13	n=15	n=17	n=19	n=21
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0
2		0	0	0	0	0	0	0	0	0	0
3		1	0	0	0	0	0	0	0	0	0
4			2	0	0	0	0	0	0	0	0
5			7	2	0	0	0	0	0	0	0
6				12	2	0	0	0	0	0	0
7				31	13	2	0	0	0	0	0
8					55	14	2	0	0	0	0
9					127	72	14	2	0	0	0
10						238	78	14	2	0	0
11						511	340	79	14	2	0
12							984	389	80	14	2
13							2047	1515	408	80	14
14								4003	1834	414	80
15								8191	6546	1970	415
16									16174	8312	2021
17									32767	27642	9158
18										65044	36708
19										131071	115037
20											260975
21											524287

Table 21.   Number of "Missings" for Odd *n*

The steady state values in the Table for n odd are 2, 14 and 80. There is an additional pattern that is shared by both Tables. These values are verified by listing the necklaces for a given *n* and a given class *k = n-q*. For example, if *n* = 7 we have the following information in Table 22:

Necklace	Class	Decimal Equivalent
1 1 1 1 1 1 1	7	127
1 1 1 1 1 1 0	6	126
1 1 1 1 1 0 0	5	124
1 1 1 1 0 1 0	4	122
1 1 1 1 0 0 0	4	120
1 1 1 0 1 1 0	3	118
1 1 1 0 1 0 0	3	116
1 1 1 0 0 1 0	3	114
1 1 1 0 0 0 0	3	112
1 1 0 1 1 0 0	2	108
1 1 0 1 0 1 0	2	106
1 1 0 1 0 0 0	2	104
1 1 0 0 1 0 0	2	100
1 1 0 0 0 1 0	2	98
1 1 0 0 0 0 0	2	96
1 0 1 0 1 0 0	1	84
1 0 1 0 0 0 0	1	80
1 0 0 1 0 0 0	1	72
1 0 0 0 0 0 0	1	64
0 0 0 0 0 0 0	0	0

Table 22.   Necklaces and Their Classes for $n = 7$ Used in Analyzing "Missings"

Note for class 7-5 = 2 there are two missing decimal equivalents: 110 and 102, corresponding to the strings 1101110 and 1100110, thus the entry (highlighted) in the Table above of 2 "missing". When considering class 9-6 = 3 (for $n = 9$) and class 11-7 = 4 (for $n =$ 11) there are also 2 "missings": 111011110 and 111001110 (for $n = 9$),

11110111110 and 11110011110 (for $n = 11$). With knowledge of how many "missings" there are between necklaces, we can use the following formula in determining how many necklaces, N_n, there are between any two necklaces

$$N_n = \left(\frac{n_{d_1} - n_{d_2}}{2} \right) - N_m - 1 ,\qquad (4.1)$$

where n_{d_1}, n_{d_2} are the decimal value of the necklaces and $n_{d_1} \geq n_{d_2}$, and N_m represents the number of "missings" between these two necklaces ($N_m = 0$ if $n_{d_1} = n_{d_2}$). For example, given the necklaces 1110000 and 1100000, we have

$$N_n = \left(\frac{112 - 96}{2} \right) - 2 - 1 = 5 \; necklaces ,$$

which corresponds with the Table above. Without knowledge of the number of "missings", we would have needed to run the Necklace Algorithm in order to determine this information.

Although the steady state values differ whether n is even or odd, they do share an additional common pattern.   Comparing the first few diagonal elements as we approach the steady state values we have:

For n even:

3, 3, 5, 5, 5, … steady state value = 5

15, 25, 31, 32, 33, 33, 33, … steady state value = 33

63, 114, 155, 174, 180, 181, 182, 182, 182, … steady state value = 182

For n odd:

1, 2, 2, 2, … steady state value = 2

7, 12, 13, 14, 14, 14, … steady state value = 14

31, 55, 72, 78, 79, 80, 80, 80, … steady state value = 80

If we form the *difference sequence* between adjacent values we have:

For n even:

2, 0, 0, 0, …

10, 6, 1, 1, 0, 0, …

51, 41, 19, 6, 1, 1, 0, 0, …

For n odd:

1, 0, 0, 0, …

5, 1, 1, 0, 0, 0, …

24, 17, 6, 1, 1, 0, 0, 0, …

Based on the difference sequences, we can conjecture what the steady state value should be for a given n, class number and the last few values approaching the steady state. In the case of $n = 24$ and class number $= 24 - 17 = 7$, the steady state value should be $922 + 2 = 924$. For $n = 23$ and class number $= 23 - 16 = 7$, the steady state value should be $415 + 1 = 416$.

Running the data for higher values of n, we notice the difference sequences for n odd or even are the same, namely: 1, 1, 6, 19, 51, 138, … Comparing these values to an online integer sequence database [Sl] to see if this pattern matched any other mathematical structure, we found there was no match. Evidently, this problem has not been studied previously. More work needs to be done to uncover the meaning of this pattern since it unifies the data for both n odd and even.

C.   MAPPING OF SUBSEQUENCES OF NECKLACES

A second approach to understand the missing n-tuples involves trying to enumerate the number of necklaces in a given class $k = n-q$, by creating a mapping of these necklaces to those of length $n-k$, where the total number of necklaces was known. If we know a lot of information (such as number and location of "missings") for necklaces of length $n-k$, we want to know how much information we can predict about necklaces of length n, class k. For group of missings associated with steady state, there seems to be a lot of predictability.

Define an "*m*-missing" to be a binary string in the group of "missings "associated with steady state value *m* for a particular *n*. For example, the "2-missings" for $n = 7$ and $n = 9$ is shown in Table 23.

n =7 "2-missing" (Class 2)	*n* = 9 "2-missing" (Class 3)
$1101110_2 = 110_{10}$	$111011110_2 = 478_{10}$
$1100110_2 = 102_{10}$	$111001110_2 = 462_{10}$

Table 23.   Comparison of $n = 7$ and $n = 9$ "2-missings"

The binary and deciemal values are shown. Note that the "2-missings" for $n = 9$ is obtained by adding a 1 to the two longest runs of 1s for $n = 7$ (in this case there are only two runs in which to add a 1).

n =8 "5-missing" (Class 2)	*n* = 10 "5-missing" (Class 3)
$11011110_2 = 222_{10}$	$1110111110_2 = 958_{10}$
$11011100_2 = 220_{10}$	$1110111100_2 = 956_{10}$
$11010110_2 = 214_{10}$	$1110101110_2 = 942_{10}$
$11001110_2 = 206_{10}$	$1110011110_2 = 926_{10}$
$11000110_2 = 198_{10}$	$1110001110_2 = 910_{10}$

Table 24.   Comparison of $n = 8$ and $n = 10$ "5-missings"

In both Tables the length *n* increased by two and 1 bit was added to each of the longest run of ones. Since we have a one-to-one mapping between both sets of "missings", the number of "missings" does not change and that is why we have steady state values for the "missings".

There is another means of evaluating a mapping from a set of missings for length *n-k* to another set of missings of length *n*. Consider the clump that is generated as the Necklace Algorithm transitions from one class to another while generating necklaces. The last

element of a class k will have k ones followed by all zeros: $\underbrace{111...1}_{k\,ones}0....0$. The next

necklace produced will be of the form $\underbrace{111...1}_{k-1ones}0\underbrace{111...1}_{k-1ones}0...0\underbrace{111...1}_{k-1ones}0 \underbrace{11..0}_{n-r(2k-1)}$ where r is the

number of times $\underbrace{111...1}_{k-1ones}0$ occurs in the sequence. Using the following formula, we can

find the number of missings between two consecutive necklaces generated by the

Necklace Algorithm:

$$N_m = \left(\frac{\left|N_{d_1} - N_{d_2}\right| - 2}{2}\right) \tag{4.2}$$

.

N_m is the number of missings and N_{d_1}, N_{d_2} are the decimal values of the necklaces. We

can then determine the clump size, or number of necklaces, between the smallest

necklace of class k and the laragest necklace of class k-1. If we remove the first k bits

from the left of the necklace $\underbrace{111...1}_{k-1ones}0\underbrace{111...1}_{k-1ones}0...0\underbrace{111...1}_{k-1ones}0 \underbrace{11..0}_{n-r(2k-1)}$, then we obtain a smaller

version of this necklace that is derived from the necklace $\underbrace{111...1}_{k\,ones}0....0$ of length n-k.

Knowing the size of the first clump for the n-k case should give us insight into knowing

the size of the first clump for the n case. For example, let $n = 32$. Then for $k = 6$ we have

$$11111100000000000000000000000000.$$

Applying the Θ-Algorithm, we obtain

$$11111011111011111011111011111010.$$

Removing the first 6 bits from 11111011111011111011111011111010 we obtain

$$11111011111011111011111010 \; (n = 26).$$

58

This is derived from (using the Necklace Algorithm)

11111100000000000000000000

The following Table summarizes the number of missings between these consecutive necklaces for n = 32, 26, 20, 14 and 8.

	$n = 32$	$n = 26$	$n = 20$	$n = 14$	$n = 8$
6 1's followed by all 0s	4227858432	66060288	1032192	16128	252
θ^1	4226793210	66043642	1031930	16122	250
No. Missings	532610	8322	130	2	0

Table 25.   Comparison of number of "missings" for $n = 32$, 26, 20, 14 and 8.

If we take the following ratios

$$\left\lfloor \frac{532610}{8322} \right\rfloor = 64 = 2^6, \left\lfloor \frac{8322}{130} \right\rfloor = 64 = 2^6, \text{ and } \left\lfloor \frac{130}{2} \right\rfloor = 65 \approx 2^6$$

it seems as n increases by 6, the number of missings for the first clump increases by a factor of 2^6. This type of information can be useful in reducing the number of computations needed when generating necklaces. More work needs to be done to understand the nature of this mapping between various classes for necklaces of a given length n and other families of necklaces of length $< n$.

D.   CONCLUSION

Improvement in Petriu's scheme involved improving the hardware synchronization design. Improving our scheme is of a mathematical nature. Gaining more theoretical understanding of these issues will allow us to make significant computational gains. This potential benefit is not possible through Petriu's scheme.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A: PSEUDO-RANDOM BINARY SEQUENCES

A.   SHIFT REGISTER GENERATION OF A PSEUDO RANDOM BINARY SEQUENCE (PBRS)

A Pseudo-Random Binary Sequence (PBRS) is a shortened De Bruijn sequence since it contains all possible 2^n n-tuples except the all zeros sequence [Mi]. So the length of the PBRS is $2^n - 1$. A PBRS contains 2^{n-1} ones and $2^{n-1} - 1$ zeros. There are 2^{n-1} odd numbers (binary numbers ending in 1) and $2^{n-1} - 1$ even numbers (binary numbers ending in 0) between 1 and 2^{n-1}. So the total is

$$2^{n-1} + 2^{n-1} - 1 = \frac{1}{2}(2^n + 2^n) - 1 = 2^n - 1 \text{ total n-tuples}$$

The number of runs (consecutive sequence of identical integers) of ones and zeros are approximately the same. For any given n, we have ½ of the runs have length 1, ¼ of the runs have length 2, 1/8 have length 3 and 1/16 have length 4, etc. as long as the fraction makes sense [Go].

One can generate a PBRS using a linear shift register that is modeled using a primitive polynomial $h(x)$ of degree n [Ma]. A primitive polynomial is one that is irreducible and has maximum period. The coefficients of the primitive polynomial come from the field of $\mathbb{Z}_2 = \{0,1\}$. An example is the following:

$$h(x) = 1x^4 + 0x^3 + 0x^2 + 1x^1 + 1x^0 = x^4 + x + 1 \tag{A.1}$$

The diagram below gives an illustration of a *linear* shift register that generates a maximal length sequence of length $2^4 - 1$. Each stage in the register can contain a value of "0" or "1". The only non-zero coefficients of $h(x)$ are those corresponding to x^4, x^1 and x^0. The values in the stages corresponding to x^1 and x^0 are the only ones that are "tapped" to be added. Since $x^4 = x^1 + x^0$, the value associated with x^4 is inserted into the stage associated with x^3 at the next time unit when all of the constants shift to the right. The all zeros input is excluded so that we do not produce a continuous sequence of zeros. Starting with an initial state of 1 0 0 0 loaded into the shift register in Figure 13, we generate the

various state values as a result of adding the stages corresponding to x^1 and x^0, and then shifting the contents of the stages one step to the right [Ma]. (See Table 25).
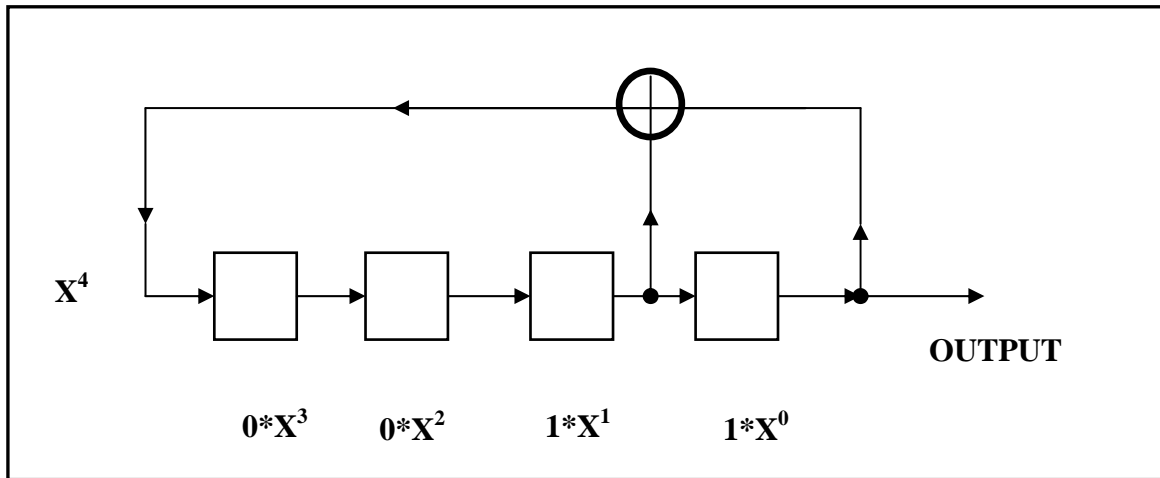
Figure 13.   Feedback Shift Register Corresponding to $x^4 + x + 1$ (From [Ma])

State Number	State				Output
	MSB			LSB	
	x^3	x^2	x^1	x^0	
0	1	0	0	0	0
1	0	1	0	0	0
2	0	0	1	0	0
3	1	0	0	1	1
4	1	1	0	0	0
5	0	1	1	0	0
6	1	0	1	1	1
7	0	1	0	1	1
8	1	0	1	0	0
9	1	1	0	1	1
10	1	1	1	0	0
11	1	1	1	1	1
12	0	1	1	1	1

State Number	State MSB x^3	x^2	x^1	LSB x^0	Output
13	0	0	1	1	1
14	0	0	0	1	1
15	1	0	0	0	0
16	0	1	0	0	0

Table 26.   16 Feedback Shift Register States Corresponding to $x^4 + x + 1$ (From [Ma])

Note that there are 15 states corresponding to all the possible $2^4 - 1$ different non-zero binary 4-tuples. Since states 15 and 16 are just repeats of states 0 and 1, we have generated a cycle of length $2^4 - 1$. The value for Most Significant Bit (MSB) column in a row is just the sum of the LSB value (the value for x^0) and the x^1 value from the previous row. The output corresponds to the last column of states (the Least Significant Bit (LSB) position). Since the shift register essentially generates a cycle of length $2^n - 1$, any one of the states we load into the shift register will produce a cyclically shifted version of the output of various initial states as can be seen in Table 26 [Ma].

STATE x^3	x^2	x^1	x^0	OUTPUT
1	0	0	0	0 0 0 1 0 0 1 1 0 1 0 1 1 1 1
0	1	0	0	0 0 1 0 0 1 1 0 1 0 1 1 1 1 0
0	0	1	0	0 1 0 0 1 1 0 1 0 1 1 1 1 0 0
1	0	0	1	1 0 0 1 1 0 1 0 1 1 1 1 0 0 0
1	1	0	0	0 0 1 1 0 1 0 1 1 1 1 0 0 0 1
0	1	1	0	0 1 1 0 1 0 1 1 1 1 0 0 0 1 0
1	0	1	1	1 1 0 1 0 1 1 1 1 0 0 0 1 0 0
0	1	0	1	1 0 1 0 1 1 1 1 0 0 0 1 0 0 1

STATE				OUTPUT
x^3	x^2	x^1	x^0	
1	0	1	0	0 1 0 1 1 1 1 0 0 0 0 1 0 0 1 1
1	1	0	1	1 0 1 1 1 1 0 0 0 1 0 0 1 1 0
1	1	1	0	0 1 1 1 1 0 0 0 1 0 0 1 1 0 1
1	1	1	1	1 1 1 1 0 0 0 1 0 0 1 1 0 1 0
0	1	1	1	1 1 1 0 0 0 1 0 0 1 1 0 1 0 1
0	0	1	1	1 1 0 0 0 1 0 0 1 1 0 1 0 1 1
0	0	0	1	1 0 0 0 1 0 0 1 1 0 1 0 1 1 1

Table 27.   16 Output Sequences Corresponding to $x^4 + x + 1$ (From [])

Thus there are $2^4 - 1$ different pseudo-random binary sequences, each one corresponding to a state of the linear shift register. (We can generate a sequence of length 2^n, called a De Bruijn sequence only if we use a non-linear shift register [Go]). The important point to make is that the sequence is a result of how we define $h(x)$. If we change $h(x)$ to be another irreducible polynomial $x^4 + x^3 + 1$, then we reverse the sequences in the previous Table.   Appropriate polynomials exist for every value of *n* [Ma].

B.   FINITE FIELDS AND SHIFT REGISTERS

Let a, b and c be elements of a set F with two binary operations: addition and multiplication. Then F is a field if the following properties hold [Ma]:

(i) $a + b = b + a$

(ii) $ab = ba$

(iii) $a + (b + c) = (a + b) + c$

(iv) $a(bc) = (ab)c$

(v) $a(b + c) = ab + ac$

Properties (i) and (ii) are the commutative properties, properties (iii) and (iv) are the associative properties and property (v) is the distributive property. Further, there must exist elements 0, 1 in the set and for every non-zero element ($a \neq 0$), the additive and

64

multiplicative inverses, $-a$ and a^{-1} respectively, must also exist. Then the following hold [Ma]:

(vi) $0 + a = a$

(vii) $(-a) + a = 0$

(viii) $0a = 0$

(ix) $1a = a$

(x) $(a^{-1})a = 1$

A finite field is one that contains a finite number of elements. A field of q elements is known as a *Galois field* and is denoted by $GF(q)$ where q is a power of a prime number. An example of a simple field is the integers modulo p , $GF(p)$, where p is a prime number. The elements of this field are $\{0,1,...,p-1\}$ and addition, subtraction, multiplication and division (by non-zero elements) are carried out modulo p . So if $p = 2$, then $GF(2) = \{0,1\} = \mathbb{Z}_2$. We can construct a field with p^n elements, where n is any integer and p is a prime number, by selecting an irreducible polynomial $h(x)$ of degree n. The elements of this field will all be polynomials in x of degree $<n$ with coefficients from $GF(p)$. We will illustrate with $p = 2$ and $n = 4$. We construct a field of $2^4 = 16$ elements. In this case, we let the irreducible polynomial be $h(x) = x^4 + x + 1$. This leads to the results in Table 27 [Ma].

Polynomial	Polynomial Coefficients	Power of ξ
1	0 0 0 1	$\xi^0 = 1$
x	0 0 1 0	ξ^1
$xx = x^2$	0 1 0 0	ξ^2
x^3	1 0 0 0	ξ^3
$x + 1$	0 0 1 1	ξ^4
$x^2 + x$	0 1 1 0	ξ^5

Polynomial	Polynomial Coefficients	Power of ξ
$x^3 + x^2$	1 1 0 0	ξ^6
$x^3 + x + 1$	1 0 1 1	ξ^7
$x^2 + 1$	0 1 0 1	ξ^8
$x^3 + x$	1 0 1 0	ξ^9
$x^2 + x + 1$	0 1 1 1	ξ^{10}
$x^3 + x^2 + x$	1 1 1 0	ξ^{11}
$x^3 + x^2 + x + 1$	1 1 1 1	ξ^{12}
$x^3 + x^2 + 1$	1 1 0 1	ξ^{13}
$x^3 + 1$	1 0 0 1	ξ^{14}
1	0 0 0 1	$\xi^{15} = 1 = \xi^0$

Table 28.   Polynomials Associated with 16 States of the Shift Register (From [Ma])

Note that the coefficients of the polynomials correspond to the different possible states of the shift register. (The 16th state, not appearing in the sequence, is the all zeros state). Note that the third column of the Table has corresponding powers of ξ. We denote ξ as the primitive element that is a root of the primitive polynomial $h(x) = x^4 + x + 1$. If we have $h(x) = x^4 + x + 1 = 0$ and ξ is a root, then $\xi^4 + \xi + 1 = 0$ or $\xi^4 = \xi + 1$. From this relation we can derive all other entries in the Table. We can then find the powers of any polynomial or the products of any of the polynomials above simply by performing exponential addition on the powers of ξ.

APPENDIX B: CODE

A.   BACKGROUND

The most important subroutine is the runNecklaceAlgorithm code. The algorithm of runNecklace Algorithm had to be modified from Matty's version [Mat]. Since the number of necklaces grows prohibitively large, as n gets large, it was necessary to gather statistical information on necklaces (such as number of "missings") and insert signposts while executing the Necklace Algorithm to generate necklaces. The level of detail analysis is also limited by the amount of memory needed to store information on "missings", clump sizes and other pertinent information. The runNecklaceAlgorithm is also limited to generate necklaces of a given binary string length n at a particular time. So comparison of data for different binary string lengths can only occur by storing the data separately.   Fortunately, it is not necessary to generate all the necklaces using the necklace algorithm so detailed analysis can be performed on a small range of necklaces. The Automated Guided Vehicle (AGV) determines its position in a greedy DeBruijn track by using this feature.

B.   HEADER FILES

1.   Necklace Header File

```
//***********************************************************************
// Necklaces.h
//
// LT John Ortiz
//
// Project: Thesis
// Operating Environment: Windows XP Home
// Compiler: Visual Studio .NET
// Date:
// Description:
//
//***********************************************************************
#ifndef NECKLACES_H
#define NECKLACES_H
```

```
//**************************************
// Class:   Necklaces
// Purpose:
//**************************************
class Necklaces {

  public:
  const static int MAX_COLUMNS = 400;

// Keep in mind the limits of memory when displaying missings.
// You might not display them all if there are too many.
// Adjusting the threshold and percentage may help you see most of them.

  const static int DEFAULT_THRESHOLD = 0;

  const static int DEFAULT_PERCENT = 0;
  const static int MAX_LENGTH = 40;
  const static int OUTPUT_CELL_SIZE = 4;  //output size

  //**m = 14 is the largest dimension I can use here without running out of memory

  //**************************************
  // Method       Necklaces--Default Constructor
  // Return value  none
  // Parameters    none
  // Purpose
  //**************************************
  Necklaces();

  //**************************************
  // Method       Necklaces-- Constructor
  // Return value  none
  // Parameters    int size, int thresh, float per
  // Purpose
  //**************************************
  Necklaces(int size, int thresh, int per);

  //**************************************
  // Method        initNecklace
  // Return value   decimal equivalent
  // Parameters     int ones
  // Purpose        to create a sequence consisting of a certain number of leading ones
  //                with the remaining sequence consisting of zeros
  //**************************************
  long long initNecklace(int ones);
```

```
//****************************************
// Method        printSequence
// Return value   none
// Parameters      none
// Purpose        displays necklaces, theta generated and unobserved binary sequences
//****************************************
void printSequence();

//****************************************
// Method        binToDec
// Return value   none
// Parameters      none
// Purpose        converts binary number of a sequence into its decimal equivalent
//****************************************
long long binToDec();

//****************************************
// Method        decToBin
// Return value   none
// Parameters      int input
// Purpose        converts binary number of a sequence into its
//               decimal equivalent and returns a pointer to binary sequence
//****************************************
void decToBin(long long input);

//****************************************
// Method        createUnobservedSequences
// Return value   none
// Parameters      int dec1, int dec2
// Purpose        generates all even binary sequences between any two even sequences
//****************************************
long createUnobservedSequences(int a, int b);

//****************************************
// Method        findDivisors
// Return value   none
// Parameters      int num
// Purpose        find all divisors for an integer n
//****************************************
void findDivisors(int num);

//****************************************
// Method        printDivisors
// Return value   none
```

```
// Parameters    int num
// Purpose       prints all divisors for an integer n
//****************************************
void printDivisors(int num);


//****************************************
// Method        findGCD
// Return value   int gcd
// Parameters    int num1, num2
// Purpose       find greatest common divisor (gcd) for two integers
//****************************************
int findGCD(int num1, int num2);


//****************************************
// Method        eulerTotient
// Return value   long long Zn
// Parameters    int num
// Purpose       Calculate number of necklaces for a given n
//****************************************
long long eulerTotient(int num);


//****************************************
// Method        generatePower2
// Return value   long long power2val
// Parameters    long long exponent
// Purpose       generates 2 raised to any integer
//****************************************
long long generatePower2(long long exponent);


//****************************************
// Method        runNecklaceAlgorithm
// Return value   none
// Parameters    long long initial, long long final, long long input, bool numflag
// Purpose       executes necklace algorithm
//****************************************
void runNecklaceAlgorithm(long long initial, long long final, long long input, bool
numflag);


//****************************************
// Method        inputBinaryString
// Return value   none
// Parameters    none
// Purpose       checks whether user input binary string is valid
//****************************************
void inputBinaryString();
```

```
//***************************************
// Method        countRunningOnes
// Return value  int nclass, largest number of running ones
// Parameters    int start, int finish
// Purpose       counts the number of running ones in each necklace
//***************************************
int countRunningOnes(int start, int finish);


//***************************************
// Method        countRunningZeros
// Return value  int nclass, largest number of runnning zeros
// Parameters    int start, int finish
// Purpose       counts the largest number of running ones in each sequence
//***************************************
int countRunningZeros(int start, int finish);


//***************************************
// Method        shiftSequenceToNecklace
// Return value  none
// Parameters    none
// Purpose       given an arbitrary n-tuple, it rotates the sequence until it generates
//               necklace
//***************************************
void shiftSequenceToNecklace();


//***************************************
// Method        generateShiftingOnesSequence
// Return value  none
// Parameters    long long neckdec, decimal equivalent of necklace
//               long long numclump, number of missings after necklace
//               long distance, distance between adjacent signposts
//               long necknum, necklace number associated with each signpost
// Purpose       generates shifting ones sequence to use as signposts
// Note: Entries are stored in a matrix format that is read from left to right where the last
// entry of the previous row precedes the first entry of the current row. This ensures I
// have enough memory to collect enough signposts
//***************************************
void generateShiftingOnesSequence(long long necdec, long long clmpsz, long distance,
long necknum);


//***************************************
// Method        getWeight
// Return value  void
// Parameters    int num
```

```
// Purpose        obtain user weight input for shifting ones sigposts
//*****************************************
void getWeight(int num);


//*****************************************
// Method          cycleshiftSequence
// Return value    none
// Parameters      int index, index of largest sequence of running ones
// Purpose         cyclically rotate sequence until largest number of
//                 running ones is at the leftmost position of the sequence
//*****************************************
void cycleshiftSequence(int index);


//*****************************************
// Method          createInputString
// Return value    none
// Parameters      long long input
// Purpose         creates binary string version of user input decimal
//                 to be matched to a subsection of the DeBruijn sequence
//*****************************************
void createInputString(long long input);


//*****************************************
// Method          createFirstNecklace
// Return value    none
// Parameters      long long input
// Purpose         creates first necklace to be used in creating a
//                 subsection of the DeBruijn sequence
//*****************************************
void createFirstNecklace(long long input);


//*****************************************
// Method          createMiddleNecklace
// Return value    none
// Parameters      long long input
// Purpose         creates middle necklace to be used in creating a
//                 subsection of the DeBruijn sequence
//*****************************************
void createMiddleNecklace(long long input);


//*****************************************
// Method          createLastNecklace
// Return value    none
// Parameters      long long input
// Purpose         creates last necklace to be used in creating a
```

```
//              subsection of the DeBruijn sequence
//****************************************
void createLastNecklace(long long input);

//****************************************
// Method       createDeBruijnSection
// Return value   none
// Parameters     none
// Purpose        concatenates nonperiodic portions of three adjacent necklaces
//****************************************
void createDeBruijnSection();

//****************************************
// Method       createWraparoundDeBruijnSection
// Return value   none
// Parameters     none
// Purpose        concatenates nonperiodic portions of four adjacent necklaces:
//                100...00 ^ 0 ^ 1 ^ 111...10
//****************************************
void createWraparoundDeBruijnSection();

//****************************************
// Method       getInputDec
// Return value   none
// Parameters     long long input
// Purpose        obtains user input decimal value that yields binary string
//                to be searched for in DeBruijn sequence
//****************************************
void getInputDec(long long input);

//****************************************
// Method       searchDeBruijnString
// Return value   long long position
// Parameters     none
// Purpose        searches for user input binary string in DeBruijn sequence
//                and returns its location
//****************************************
long long searchDeBruijnString();

//****************************************
// Method       findSequencePeriod
// Return value   int period
// Parameters     none
// Purpose        find period of a necklace
//****************************************
```

```
int findSequencePeriod();

//****************************************
// Method        computeDensity
// Return value   int nclass, largest number of runnning zeros
// Parameters     int start, int finish
// Purpose        counts the total number of ones in the sequence
// ** Note:  To compute number of zeros = length of sequence - computeDensity
//****************************************
int computeDensity(int start, int finish);

//****************************************
// Method        testforNecklace
// Return value   bool foundnecklace
// Parameters     long long bininput
// Purpose        tests n-tuple to see if it is a necklace
//****************************************
bool testforNecklace(long long bininput);

//****************************************
// Method        countNecklacesPerClass
// Return value   none
// Parameters     int ones, number of leading ones
// Purpose        separates the number of leading ones in each necklace
//               into distinct classes
//****************************************
void countNecklacesPerClass(int ones);

//****************************************
// Method        generateChangeSequence
// Return value   none
// Parameters     long long oldec, decimal value of previous necklace
//               long long newdec, decimal value of current necklace
//               int indx, array index for sequence
// Purpose        counts the number of bits that differ between each necklace
//               and inserts these values into a sequence
//****************************************
void generateChangeSequence(long long olddec, long long newdec, int indx);

//****************************************
// Method        displayNecklaceInfo
// Return value   none
// Parameters     int col, classnumber, long long value, numclump
// Purpose        displays all information related to a necklace
//****************************************
```

```cpp
    void displayNecklaceInfo(int a, int b, long long c, long long d);

    //****************************************
    // Method       displayPower2Array
    // Return value   none
    // Parameters     none
    // Purpose        displays power2[i] array
    //****************************************
    void displayPower2Array();

    //****************************************
    // Method       displayDifference
    // Return value   none
    // Parameters     none
    // Purpose        calculates and displays difference between power2 approximation and
    number of necklaces
    //****************************************
    void displayDifference();

    //****************************************
    // Method       displayNumbNeck
    // Return value   none
    // Parameters     none
    // Purpose        displays number of necklaces per class
    //****************************************
    void displayNumbNeck();

    //****************************************
    // Method       displayClassNumbers
    // Return value   none
    // Parameters     none
    // Purpose        displays class numbers of necklaces in decreasing order
    //****************************************
    void displayClassNumbers();

    //****************************************
    // Method       displayNcount
    // Return value   none
    // Parameters     none
    // Purpose        displays total number of necklaces
    //****************************************
    void displayNcount();

    //****************************************
    // Method       displayClumpsPerClass
```

```
// Return value   none
// Parameters     none
// Purpose        displays total number of missing even n-tuples per class
//****************************************
void displayClumpsPerClass();


//****************************************
// Method         displayClumpDistribution
// Return value   none
// Parameters     none
// Purpose        displays various clump sizes per class
//****************************************
void displayClumpDistribution();


//****************************************
// Method         displayDecimalArray
// Return value   none
// Parameters     none
// Purpose        displays decimal equivalent of necklaces associated with clumps per
class
//****************************************
void displayDecimalArray();


//****************************************
// Method         displayShiftingOnesSignposts
// Return value   none
// Parameters     none
// Purpose        displays decimal and binary equivalents of shifting signposts
//****************************************
void displayShiftingOnesSignposts();


//****************************************
// Method         displayShiftingOnesDistances
// Return value   none
// Parameters     none
// Purpose        displays distances between shifting ones signposts
//****************************************
void displayShiftingOnesDistances();


//****************************************
// Method         displayBitChangeSequence
// Return value   none
// Parameters     none
// Purpose        displays bit changes between necklaces
//****************************************
```

```
void displayBitChangeSequence();

//*****************************************
// Method         displayFirstNecklace
// Return value   none
// Parameters     none
// Purpose        displays prior necklace in deBruijn section
//*****************************************
void displayFirstNecklace();

//*****************************************
// Method         displayMiddleNecklace
// Return value   none
// Parameters     none
// Purpose        displays input necklace in deBruijn section
//*****************************************
void displayMiddleNecklace();

//*****************************************
// Method         displayLastNecklace
// Return value   none
// Parameters     none
// Purpose        displays post necklace in deBruijn section
//*****************************************
void displayLastNecklace();

//*****************************************
// Method         displaydeBruijnSection
// Return value   none
// Parameters     none
// Purpose        displays deBruijn sectional associated with user input necklace
//*****************************************
void displaydeBruijnSection();

//*****************************************
// Method         displayUnobservedSequences
// Return value   none
// Parameters     none
// Purpose        displays all missings (even binary n-tuples) between any 2 nunmbers
//*****************************************
void displayUnobservedSequences();

//*****************************************
// Method         outputData
// Return value   none
```

```cpp
// Parameters
// Purpose        display numerical results of algorithm output
//************************************
void outputData();


private:

//****************************************
// Method          initSequences
// Return value    none
// Parameters      int size, int thresh, float per
// Purpose         to initialize all the sequences involved in the analysis of the algorithm
//****************************************
void initSequences(int size, int thresh, int per);

private:

//variables
  int m; // size of sequence
  int k; // number of leading ones input by the user
  int q; //index for onescount array
  int mark; // marker for shifting ones array
  int dim; //dimension for decimalarray
  int begin; //index to begin counting ones or zeros
  int threshold; // smallest size of missings displayed
  int num; // integral numerator of percentage
  int numevensignposts; // number of evenly spaced signposts
  int ncountdiff; // counts the number of necklaces between adjacent signposts
  int firstperiod; // period of first adjacent necklace in a deBruijn section
  int middleperiod; // period of middle adjacent necklace in a deBruijn section
  int lastperiod; // period of last adjacent necklace in a deBruijn section
  int sumperiod; // sum of last three periods
  int weight; // weight for shifting ones signposts
  int rowindex; // shifting ones row index <= MAX_LENGTH
  int colindex; // shifting ones column index <= MAX_COLUMNS
  long distance; // computes distance between initial and final necklace
  long olddistance; // computes distance of last necklace
  long totaldistance; //distance from end of deBruijn sequence (111.11) until first
                      //necklace in deBruijn section
  float percent; // percentage of maximum clump size in a given class
  long ncount; // necklace number on theta generated list
  long noncount; // theta-generated non-necklace count
  long shiftonesnoclump; //number of shifting ones sequence NOT associated with
                         //missings
```

```cpp
long long absposition; // // absolute position from origin or end of 000...0 necklace
long long totmissing; // number of missing even n-tuples in a given class
long long maxclumpsize;  // maximum clump size for all the classes
long long inputstrdec; // decimal equivalent of input string
double shiftonesclump; //number of shifting ones sequence associated with missings
double binlength; // binary length of 2^m-1
double petriumilestones; // number of petriu milsetones assuming max period of m
double milestonedistance; // petriu distance given number of signposts
double topt; // optimal petriudistance between milestones

//arrays
int sequence[MAX_LENGTH]; // binary sequence for necklaces
int divisors[MAX_LENGTH+1];  // all divisors for a given n
int bitchangeseq[MAX_COLUMNS]; // stores the number of bit changes between
                              // necklaces
int inputstring[MAX_LENGTH]; // binary sequence of user input decimal
int firstnecklace[MAX_LENGTH]; // necklace listed before user input necklace
int middlenecklace[MAX_LENGTH]; // user input necklace
int lastnecklace[MAX_LENGTH]; // necklace listed after user input necklace
int debruijnsection[MAX_LENGTH]; // section of DB sequence containing prior
                                // three arrays
int wrapdebruijnsection[MAX_LENGTH]; // section of DB sequence that includes
                                    // strings between 00...0 and 11...1
long onescount[MAX_LENGTH+1]; //collects number of necklaces per class
long power2[MAX_LENGTH+1]; // power of 2 approximation to onescount
long diff[MAX_LENGTH+1];  // difference between power2 and onescount
long ncountarray[MAX_LENGTH][MAX_COLUMNS]; // numbered count of each
                                          //necklace on theta list per class
//long long decimalarray[MAX_LENGTH][MAX_COLUMNS]; // decimal
                                          //equivalent of each necklace per class
long clumpcntarray[MAX_LENGTH+1]; //number of clumps per class
long long missingarray[MAX_LENGTH][MAX_COLUMNS];
long long maxclumpsizearray[MAX_LENGTH]; // maximum clump size for each
                                        // class
long long totalmissings[MAX_LENGTH];
long long missingbelowthreshold[MAX_LENGTH]; // total number of missing for a
                                            // given class less than threshold
long long clumpsizearray[MAX_LENGTH][MAX_COLUMNS];
long long evenspacedsignpostarray[MAX_LENGTH][MAX_COLUMNS]; //contains
                                    // decimal value of evenly spaced signposts
long long shiftonesarray[MAX_LENGTH][MAX_COLUMNS]; // contains decimal
                              //value of necklaces with shifting ones in a given class
long shiftonesdistancearray[MAX_LENGTH][MAX_COLUMNS]; // contains
                              // distances (in bits) between adjacent signposts
long shiftonesncountarray[MAX_LENGTH][MAX_COLUMNS]; // contains
```

```cpp
                                              // necklace number of each signpost
    long long shiftonesclumpsizearray[MAX_LENGTH][MAX_COLUMNS]; // contains
                                              // clumpsize associated with shifting ones array
}; //end Class Necklaces

#endif  // end of file Necklaces.h
```

2. IO Thesis, FileOpeningException and SieveSizeException Header Files

```cpp
//**********************************************************************
***
// IOThesis.h
//
// LT John Ortiz
//
//
// Course: CS2971 2006 Q1
// Project: Generating Necklaces
// Operating Environment: Windows XP Home
// Compiler: Visual Studio .NET
// Date: 08 December 2005
// Description:
//
//**********************************************************************
***
#ifndef IOTHESIS_H
#define IOTHESIS_H

#include <string>
#include <sstream>
#include "NecklaceTable.h"
#include <fstream>

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

const int EXIT_WITH_ERROR = 1; // global constant that enables a program to exit
when there is an error.
const int EXIT_NORMALLY = 0; // global constant that enables a program to exit when
there is no error.

//***************************************
// Class:   IOThesis
// Purpose: Display a 20x20 Matrix of integers where multiples of a filter
//          provided by the user are replaced by blank spaces.
//***************************************
```

```cpp
class IOThesis {

public:

    //*****************************************
    // Method        displayMessage
    // Return value   none
    // Parameters     none
    // Purpose        prompts user for dimension input
    //*****************************************
    static void displayMessage(string message);

    //*****************************************
    // Method openOutputFile()
    // Return value
    // Parameters: const char *FILE_NAME the name of the file to open for writing
    // Purpose: dynamically create a new ofstream and returns a pointer if successful
    //*****************************************
    static ofstream* openOutputFile(const char *FILE_NAME);

    //*****************************************
    // Method openInputFile
    // Return value ifstream*
    // Parameters const char *FILE_NAME, the file to open for reading
    // Purpose
    //*****************************************
    static ifstream* openInputFile(const char *FILE_NAME);

    //*****************************************
    // Method        pauseBeforeExit
    // Return value   none
    // Parameters     int condition (EXIT_WITH_ERROR = 1, global defined in
    IOController.h)
    //                (EXIT_NORMALLY = 0, global defined in IOController.h)
    // Purpose        keeps display window open long enough to see results
    //*****************************************
    static void pauseBeforeExit(int condition);

    //*****************************************
    // Method        inputInt
    // Return value   int
    // Parameters     none, function will ask for input via I/O
    // Purpose        take string input and determine if valid for an int
    //              use inputFilter = IOController::inputInt(); vice cin >> inputFilter
    //*****************************************
```

```cpp
    static int inputInt();

    //*****************************************
    // Method        inputIntWithLimits
    // Return value   int
    // Parameters    (string message, int lower, int upper)
    // Purpose       calls input int for a valid int, then checks limits
    //               used similiarly to inputInt
    //*****************************************
    static int inputIntWithLimits(string message, int lower, int upper);

    //*****************************************
    // Method        clearInputBuffer
    // Return value   bool TRUE if cin buffer has no additional stuff in it
    // Parameters    none
    // Purpose       utility function for processing inputs
    //*****************************************
    static bool clearInputBuffer();

}; //end Class IOThesis

#endif  // end of file IOThesis.h

//*************************************************************************
// FileOpeningException.h
//
// LT John Ortiz
//
//
// Course: CS2971 2006 Q1
// Project:
// Operating Environment: Windows XP Pro
// Compiler: Visual Studio .NET
// Date:       08 December 2005
// Description: Provide exception handling for file opening.
//*************************************************************************
***
#include <stdexcept>
using std::runtime_error;

//*****************************************
// Class:   FileOpeningException
// Purpose: Provide exception handling for file opening.
//*****************************************
class FileOpeningException : public runtime_error {
```

```cpp
public:

  //*********************************
  // Method      FileOpeningException--Constructor
  // Return value  none
  // Parameters    none
  // Purpose
  //***********************************
  FileOpeningException::FileOpeningException(): runtime_error("Unable to open file")
  {
    //nothing else to initialize
  }

  //variables
  private:
    //none
}; //end Class FileOpeningException


//****************************************************************************
// SieveSizeException.h
//
// LT Ortiz
//
//
// Project: Thesis
// Operating Environment: Windows XP Home
// Compiler: Visual Studio .NET
// Date:
// Description: Provide exception handling for incorrect necklace size requests
//             There is no default constructor. Coders must provide an argument.
//             Choose the appropriate enumerated constant
//****************************************************************************
#ifndef SIEVESIZEEXCEPTION_H
#define SIEVESIZEEXCEPTION_H

#include <string>
#include <stdexcept>
using namespace std;


//*****************************************
// Class:   SieveSizeException
// Purpose: Provide exception handling for incorrect sieve size requests
//*****************************************
```

```cpp
class SieveSizeException : public runtime_error {

public:
  const static enum { TOO_SMALL, TOO_BIG };

  //**************************************
  // Method       SieveSizeException--Constructor
  // Return value  none
  // Parameters    const int ERROR_ID
  // Purpose
  //**************************************
  SieveSizeException(const int ERROR_ID) : runtime_error(
resolveErrorType(ERROR_ID) ) {
    //nothing else to initialize
  }


  //*****************************************
  // Method       resolveErrorType
  // Return value   string  The exception message
  // Parameters     const int errorID
  // Purpose        generate exception text
  //*****************************************
  string resolveErrorType(const int ERROR_ID) {

    switch (ERROR_ID) {
      case TOO_SMALL:
        return "A necklace needs to have positive dimensionality";
        break;

      case TOO_BIG:
        return "That would produce a necklace too large to output in the console";
        break;

      default:
        return "Really bad things are happening here...function was not provided
TOO_SMALL or TOO_BIG as an argument";
    }

  }//end function resolveErrorType


  //variables
  private:
    //none
```

```
};  //end Class SieveSizeException

#endif  // end of file SieveSizeException.h
```

C.   CPP FILES

1.   Necklace CPP File

```cpp
//************************************************************************
***
// Necklaces.cpp
//
// LT John Ortiz
//
//
// Project: Thesis
// Operating Environment: Windows XP Home
// Compiler: Visual Studio .NET
// Date:
// Description:
//
//************************************************************************
***

#include <iostream>
#include <iomanip>
#include <cctype>
#include "math.h"
#include "Necklaces.h"
#include "IOThesis.h"

using namespace std;



//**********************************
// Method      Necklaces--Default Constructor
// Return value  none
// Parameters    none
// Purpose
//**********************************
Necklaces::Necklaces()
{
   initSequences(MAX_COLUMNS,DEFAULT_THRESHOLD,DEFAULT_PERCENT);
// MAX_COLUMNS = 34
```

```cpp
}


//************************************
// Method       Necklaces-- Constructor
// Return value  none
// Parameters    int size, int thresh, float per
// Purpose
//************************************
Necklaces::Necklaces(int size, int thresh, int per)
{
  initSequences(size, thresh, per);
}//end constructor



//*****************************************
// Method        initSequences
// Return value   none
// Parameters     int size, int thresh, float per
// Purpose        to initialize all the sequences involved in the analysis of the algorithm
//*****************************************
void Necklaces::initSequences(int size, int thresh, int per)
{
  m = size; // defined only once for all functions
  threshold = thresh;
  begin = 0;
  num = per;
  percent = num /100;

  if ( m < 1 )
  {
    cout << "A necklace needs to have a length > = 1 \n"
       << "exiting the program.\n" << endl;

    IOThesis::pauseBeforeExit(EXIT_WITH_ERROR);
  }

  if ( m > MAX_LENGTH) // MAX_COLUMNS = 400
  {
    cout << "That would produce a necklace too large to output in the console \n"
       << "exiting the program.\n" << endl;

    IOThesis::pauseBeforeExit(EXIT_WITH_ERROR);
  }
```

```
initNecklace(0); //initializes necklace to be all zeros

for (int i = 0; i < MAX_LENGTH; i++)
{
  maxclumpsizearray[i]= 0;
  totalmissings[i] = 0;
  missingbelowthreshold[i] = 0;
  firstnecklace[i] = 9;
  middlenecklace[i] = 9;
  lastnecklace[i] = 9;
  debruijnsection[i]= 9;
  wrapdebruijnsection[i] = 9;
}

for (int i = 0; i <= MAX_LENGTH; i++)
{
  power2[i] = 1;
  diff[i] = 0;
  onescount[i] = 1;
  clumpcntarray[i] = 0;
  divisors[i] = -1;
}

for (int i = 0; i < MAX_COLUMNS; i++)
{
  bitchangeseq[i] = 0;
}

mark = -1;
for (int i = 0; i < MAX_LENGTH; i++)
{
  for(int j = 0; j < MAX_COLUMNS; j++)
  {
    missingarray[i][j] = 0;
    ncountarray[i][j] = 0;
    //decimalarray[i][j]= 9; // a necklace will always have an even decimal equivalent
    clumpsizearray[i][j] = 0;
    shiftonesarray[i][j] = mark;
    shiftonesdistancearray[i][j] = 0;
    shiftonesclumpsizearray[i][j] = 0;
    evenspacedsignpostarray[i][j] = mark;
  }
}

for (int i = 0; i < MAX_LENGTH; i++)
```

```
  {
    for(int j = 0; j < MAX_LENGTH; j++)
    {
      evenspacedsignpostarray[i][j] = 0;
    }
  }
} // end initSequences


//*****************************************
// Method       initNecklace
// Return value   decimal equivalent
// Parameters     int ones
// Purpose       to create a sequence consisting of a certain number of leading ones
//             with the remaining sequence consisting of zeros
//*****************************************
long long Necklaces::initNecklace(int ones)
{
  long long initdec = 0;

  for (int i=0; i < m; i++)
  {
    sequence[i] = 0; // sequence initially set to all zeros
  }

  k = ones;
  int x = 1;
  for(int i=0; i< k; i++) // initializing initial fill with given number of leading ones
  {
    sequence[i]= x;
  }
  initdec = binToDec();

  return initdec; //returns decimal equivalent of initial necklace
}// end initNecklace

//*****************************************
// Method       printSequence
// Return value   none
// Parameters     none
// Purpose       displays binary sequences into three parts
//*****************************************
void Necklaces::printSequence()
{
  int n = 0;
```

```cpp
    int i = 0;
    while(sequence[i] != 0 && i < m) // determine number of leading ones
    {
       n++;
       i++;
    }

    for(int i = 0; i < m; i++)
    {
       if(i == n)
       {
          cout << sequence[i] << " ";
          // this next line is optional and can be commented out if desired
          //cout << setw(OUTPUT_CELL_SIZE - 1)<< " "; //print 2 spaces after first run of
ones and a zero
       }
       else if(i == m-2)
       {
          cout << sequence[i] << " ";
          // this next line is optional and can be commented out if desired
          //cout << setw(OUTPUT_CELL_SIZE - 1)<< " "; //print 2 spaces before
terminating zero
       }
       else
       {
          cout << sequence[i] << " "; //print 1 within middle part
       }
    }
}// end printSequence


//****************************************
// Method       runNecklaceAlgorithm
// Return value   none
// Parameters     long long initial, long long final, long long input, bool numflag
// Purpose        executes necklace algorithm
//****************************************
void Necklaces::runNecklaceAlgorithm(long long initial, long long final, long long input,
bool numflag)
//void Necklaces::runNecklaceAlgorithm(int ones, long long numneck, bool numflag)
{
   ncount = 0;
   noncount = 0;
   totmissing = 0;
   maxclumpsize = 0;
```

```cpp
distance = 0;
olddistance = 0;
ncountdiff = 0;
shiftonesclump = 0;
shiftonesnoclump = 0;
firstperiod = 0;
middleperiod = 0;
lastperiod = 0;
sumperiod = 0;
int classnumber = 0;
int j = m-1; //largest index such that aj=1 and ak=0 for k=j+1 to m
//k = ones;
q = 0;
long inputncount = 0; // necklace number associated with user input necklace
long long nondecimal = 0; // decimal equivalent of current non-necklace1
long long ndecimal = 0; // decimal equivalent of current necklace
long long oldndecimal = 0; // decimal equivalent of previous necklace
long long oldnondecimal = 0; // decimal equivalent of previous non-necklace
long long priorndecimal = 0;
long long maxnum = final - initial + 1;
long long value = 0;
long long inputdec = input;
long long nummissing = 0; //difference between decimal equivalences of 2 adjacent
necklaces
int col = 0; // column number for arrays
int idx = 0;  // index number
int idx2 = 0; // second index number
int oldclassnumber = 0;
int numclump = 0; // number of clumps in a given class corresponding to number of
signposts
int numevensignposts = 0; // number of evenly spaced signposts
bool printflag = numflag;
rowindex = 0;
colindex = 0;

//initNecklace(k);

ncount = 1;
shiftonesnoclump++;
oldndecimal = initial;
decToBin(oldndecimal); // initial necklace
distance = findSequencePeriod(); // initial calculation

if (printflag == true) // print statement for initial necklace
{
```

```cpp
      cout << endl;
      cout << "\nNecklace " << ncount << ":" << endl;
      printSequence();
      cout << "  decimal equivalent = " << oldndecimal << endl;
      cout << "\n" ;
   }


  do
  {
     nummissing = 0; //resetting the count
     j=m-1; //always start at the end of the current sequence and work backwards to find
largest aj = 1
     while (sequence[j]==0) // keep decrementing index until aj = 1
     {
        j=j-1;
     }
     sequence[j]=sequence[j]-1; //subtracts 1 from jth position

     if(j!=m-1)// checks to see if we can copy beginning portion to end
     {
        for(int i=1; i < m -j; i++)
        {
          sequence[j+i] = sequence[i-1]; //copies the first portion onto the remainder of
necklace
        }
     } // end if

     if(m %(j+1)!=0) // Checking for generated nonnecklaces: (j+1) ensures we do not
have division by zero; m = real length, j+1 = real index
     {
        noncount = noncount++;
        nondecimal = binToDec();
        //if (printflag == true)
        {
          //cout << endl;
          //cout << "\nNon-Necklace " << noncount << ":" << endl;
          //printSequence();
          //cout << "decimal equivalent = " << nondecimal << endl;
          //cout << "\n" ;
        }
     } // end if

     if(m %(j+1)==0) // Checking for necklaces: (j+1) ensures we do not have division by
zero
     {
```

```cpp
      ncount = ncount++;
      ndecimal = binToDec();
      distance += findSequencePeriod(); // computing and summing up successive
                                        //  periods
      olddistance = distance - findSequencePeriod();

      if(ncount == maxnum) // allows us to exit after a given number of thetasteps
      {
        break;
      }

      if(ndecimal == inputdec) // case where missing n-tuple is close to its necklace
      {
        //cout << "\noldndecimal for first necklace = " << oldndecimal << endl;
        createFirstNecklace(oldndecimal);
        firstperiod = findSequencePeriod();
        //cout << "\nndecimal for middle necklace = " << ndecimal << endl;
        createMiddleNecklace(ndecimal);
        middleperiod = findSequencePeriod();
        inputncount = ncount;
        totaldistance = olddistance - firstperiod; // need to adjust for first necklace
      }

      if(final < inputdec)
      {
        if(ncount == inputncount + 1)
        {
          //cout << "\nndecimal for last necklace = " << ndecimal << endl;
          createLastNecklace(ndecimal);
          lastperiod = findSequencePeriod();
          sumperiod = firstperiod + middleperiod + lastperiod;
        }
      }
      if (idx <= MAX_COLUMNS) // prevents access violations in memory
      {

        generateChangeSequence(value, ndecimal, idx);
        idx++;

      }
    if(computeDensity(0, m-1) <= weight + classnumber)// tests for shifting ones in
substring: changed classnumber to 0
    {
      //generateShiftingOnesSequence(ndecimal, nummissing, distance, ncount);
       shiftonesnoclump++;
    } // end if(computeDensity)
    //{
```

```
//   evenspacedsignpostarray[col][col] = ndecimal;
//}
if (printflag == true)
{
    cout << endl;
    cout << "\nNecklace " << ncount << ":" << endl;
    cout << "\n";
    printSequence();
    cout << "  decimal equivalent = " << ndecimal << endl;
    //cout << "\nmaxdistance = " << distance << endl;
    cout << "\n" ;
} // end if(printflag)
classnumber = countRunningOnes(begin,j);
countNecklacesPerClass(classnumber);
nummissing = (oldndecimal - ndecimal - 2)/2;
if ((nummissing) > 0) // indicates there exists a clump
{
    value = 0; //resetting value to accept current value
    value = oldndecimal; // sets "value" to previous value of ndecimal
    if (printflag == true)
    {
        cout << "number missing between Necklaces " << ncount - 1 << " and " <<
ncount << " = " << nummissing << endl;
        cout << "\n" ;
    } // end if(printflag)
    totmissing += nummissing;

    if(oldclassnumber != classnumber) // verifies if class number has changed
    {
        oldclassnumber = classnumber;
        col = 0;
        numclump = 0; // reset when class number changes
        //idx = 0;  // check to see if you need it here
    }
    else
    {
        col++; // change to allow new info in a given class to be recorded
        totalmissings[classnumber]+= nummissing; // sums total missing for a given
class
    } // end if-else
    if (nummissing > maxclumpsize) // only records largest clump size per class
    {
        maxclumpsize = nummissing;
        maxclumpsizearray[classnumber]= maxclumpsize;
    }
```

```
        if(nummissing >= threshold && col <= MAX_COLUMNS) // restricting to
collecting signpost information
        {
          clumpsizearray[classnumber][col] = nummissing;
          //decimalarray[classnumber][col] = value; //stores decimal equivalent of
sequence after missings
          //cout << "decimalarray[" << classnumber << "][" << col  << "] = " << value;
          //cout << "\n" ;
          missingarray[classnumber][col] = nummissing; // records instances of all
missings > threshold for a given class
          idx++;
        }
        if(nummissing <= threshold)
        {
          missingbelowthreshold[classnumber]+= nummissing; // records only the sum of
missings below threshold
        }
        if(nummissing >= threshold) // counts the total number of clumps for a given class
        {
          numclump++;
          clumpcntarray[classnumber]= numclump;
        }
        if(computeDensity(0, m-1) <= weight + classnumber)// tests for shifting ones in
substring: changed classnumber to 0
        {
        generateShiftingOnesSequence(ndecimal, nummissing, olddistance, ncount);
        shiftonesclump++;
        } // end if(computeDensity)
        ncountarray[classnumber][col] = ncount - 1; // why do I have this here??
        if (printflag == true)
        {
          //displayNecklaceInfo(col, classnumber, value, numclump);
        }
      }// end outside if(nummissing)
      oldndecimal = ndecimal; // setting value which is the old "value" of ndecimal to
current ndecimal
      ///value = 0; //resetting value to accept current value
    } // end if(m %(j+1)==0)
  }while (ndecimal != final); // end of necklace algorithm

}// end runNecklaceAlgorithm

//***************************************
// Method       inputBinaryString
```

```cpp
// Return value   none
// Parameters     none
// Purpose        checks whether user input binary string is valid
//*****************************************
void Necklaces::inputBinaryString()
{
  const int SIZE = MAX_LENGTH;
  char ch[SIZE]; // temporary buffer for input characters
  for(int i=0; i< SIZE; i++) //initializing ch array
  {
    ch[i] = 9;
  }
  bool state = true;
  int limit = m; // limits size of user input binary string to match with constructor

  while(state == true) // This loop prompts a user to enter in only one integer that is less
than 12 characters long
  {
    int q = 0;
    int total = 0;
    char element = cin.get();
    while(element != '\n')// reads elements into a character array
    {
      ch[q] = element;
      total++;
      if (total == limit) // checks if total exceeds 12 characters
      {
        cout << "You have reached the maximum number of characters allowed" <<
endl;
        cout << "We will truncate the additional digits" << endl;
        state = false;
        break;
      }
      if (cin.eof()) // checks if ctrl-z in input as a character
      {
        cout << "\nInvalid character.";
        cin.clear(); //resets input stream so it can keep processing
        cin.putback('\n');
        state = !IOThesis::clearInputBuffer();
        break;
      }
      if (ch[q] != '0' || ch[q] != '1') // checks if there is an invalid element
      {
        cout << "\nInvalid character.";
        state = !IOThesis::clearInputBuffer();
```

```cpp
            break;
          }
          else
          {
            element = cin.get(); // gets the next element
            q++;
          }
      } // end inside while
    } // end outside while

    for (int i=0; i <= m-1; i++)
    {
      if(ch[i]!= 9)
      {
        sequence[i] = ch[i];
      }
    }


} // end inputBinaryString
//****************************************
// Method        countRunningOnes
// Return value   int numones, largest number of runnning ones
// Parameters     int start, int finish
// Purpose        counts the largest number of running ones in each sequence
//****************************************

int Necklaces::countRunningOnes(int start, int finish)
{
    int alpha = start;
    int omega = finish;
    int tempcnt = 0;
    int numones = 0;

    for(int i = alpha; i <= omega; i++)
    {
      if(sequence[i] != 0)
      {
        tempcnt++;
      }
      else
      {
        tempcnt = 0; //resets tempcnt
      }
```

```cpp
    if(tempcnt > numones)
    {
      numones = tempcnt; //ensures largest string of ones is preserved
    }
  }

  if(alpha == omega)
  {
    numones = sequence[alpha];
  }

  return numones;
} // end countRunningOnes


//****************************************
// Method       countRunningZeros
// Return value   int numzeros, largest number of runnning zeros
// Parameters     int start, int finish
// Purpose        counts the largest number of running ones in each sequence
//****************************************

int Necklaces::countRunningZeros(int start, int finish)
{
  int alpha = start;
  int omega = finish;
  int tempcnt = 0;
  int numzeros = 0;

  for(int i = alpha; i <= omega; i++)
  {
    if(sequence[i] != 1)
    {
      tempcnt++;
    }
    else
    {
      tempcnt = 0; //resets tempcnt
    }

    if(tempcnt > numzeros)
    {
      numzeros = tempcnt; //ensures largest string of zeros is preserved
    }
  }
```

```cpp
    if(alpha == omega)
    {
      if(sequence[alpha]== '0')
      {
        numzeros = 1;
      }
      else
      {
        numzeros = 0;
      }
    }

    return numzeros;
}// end countRunningZeros



//***************************************
// Method        cycleshiftSequence
// Return value   none
// Parameters     int index, index of largest sequence of running ones
// Purpose        cyclically rotate sequence a number of positions to the left
//***************************************

void Necklaces::cycleshiftSequence(int index)
{
  int temp = 0; // temporary place holder
  int seqindx = index;
  for (int i = 0; i < seqindx; i++) //outer for loop only keeps track of number of shifts
  {
    temp = sequence[0]; // beginning of sequence stored until it can be placed at the end
    for (int j = 0; j < m-1; j++)
    {
      sequence[j] = sequence[j+1]; // all bits shift one place to the left
    }
    sequence[m-1]= temp;
  }
} // end cycleshiftSequence



//***************************************
// Method        createInputString
// Return value   none
// Parameters     long long input
// Purpose        creates binary string version of user input decimal
```

```cpp
//                  to be matched to a subsection of the DeBruijn sequence
//*****************************************
void Necklaces::createInputString(long long input)
{
  long long inputdecimal = input;

  decToBin(inputdecimal);

  for(int i = 0; i < m; i++)
  {
    inputstring[i] = sequence[i];
  }

} // end createInputString


//*****************************************
// Method        createFirstNecklace
// Return value   none
// Parameters     long long input
// Purpose        creates first necklace to be used in creating a
//                subsection of the DeBruijn sequence
//*****************************************
void Necklaces::createFirstNecklace(long long input)
{
  long long firstdecimal = input;

  decToBin(firstdecimal);

  for(int i = 0; i < m; i++)
  {
    firstnecklace[i] = sequence[i];
  }
} // end createFirstNecklace

//*****************************************
// Method        createMiddleNecklace
// Return value   none
// Parameters     long long input
// Purpose        creates middle necklace to be used in creating a
//                subsection of the DeBruijn sequence
//*****************************************
void Necklaces::createMiddleNecklace(long long input)
{
  long long middledecimal = input;
```

```
    decToBin(middledecimal);

    for(int i = 0; i < m; i++)
    {
       middlenecklace[i] = sequence[i];
    }
} // end createMiddleNecklace


//****************************************
// Method        createLastNecklace
// Return value   none
// Parameters     long long input
// Purpose        creates last necklace to be used in creating a
//                subsection of the DeBruijn sequence
//****************************************
void Necklaces::createLastNecklace(long long input)
{
    long long lastdecimal = input;

    decToBin(lastdecimal);

    for(int i = 0; i < m; i++)
    {
       lastnecklace[i] = sequence[i];
    }
} // end createLastNecklace


//****************************************
// Method        createDeBruijnSection
// Return value   none
// Parameters     none
// Purpose        concatenates nonperiodic portions of three adjacent necklaces
//****************************************
void Necklaces::createDeBruijnSection()
{
    int p1 = firstperiod; // period of firstnecklace
    int p2 = middleperiod; // period of middlenecklace
    int p3 = lastperiod; // period of lastnecklace
    int p4 = 0; // period of last necklace during wraparound case
    long long testinteger = 0;  //testing for wraparound case

    for(int i = 0; i < p1; i++)
    {
```

```cpp
      debruijnsection[i] = firstnecklace[i]; // first third of deBruijn section
      //cout << "\ndebruijnsection[" << i << "] = " << debruijnsection[i] << endl;
      //cout << "\nfirstnecklace[" << i << "] = " << firstnecklace[i] << endl;
  }

  //testinteger = binToDec();
  //cout << "\ntestinteger = " << testinteger << endl;
  //if(testinteger == generatePower2(m-1) || testinteger == 0) // wraparound case
  //{
  //   initNecklace(0); // all zeros
  //   middleperiod = findSequencePeriod();
  //   initNecklace(1); // all ones
  //   lastperiod = findSequencePeriod();

  //   debruijnsection[0] = 1;
  //   debruijnsection[2*m + 1] = 0;
  //   for(int i = 1; i <= m ; i++)
  //   {
  //      debruijnsection[i] = 0;
  //   }
  //   for(int i = m+1; i <= 2*m; i++)
  //   {
  //      debruijnsection[i] = 1;
  //   }
  // } // end if
  // else
  // {
      for(int i = 0; i < p2; i++)
      {
        debruijnsection[i + p1] = middlenecklace[i]; // middle third of deBruijn section
      }
      for(int i = 0; i < p3; i++)
      {
        debruijnsection[i + p1 + p2] = lastnecklace[i]; // last third of deBruijn section
      }
  // } // end else
} // end createDeBruijnSection


//*****************************************
// Method         createWraparoundDeBruijnSection
// Return value   none
// Parameters     none
// Purpose        concatenates nonperiodic portions of four adjacent necklaces:
//                100...00 ^ 0 ^ 1 ^ 111...10
```

101

```cpp
//*****************************************
void Necklaces::createWraparoundDeBruijnSection()
{
   wrapdebruijnsection[0] = 1;
   wrapdebruijnsection[2*m + 1] = 0;

   for(int i = 1; i <= m ; i++)
   {
     wrapdebruijnsection[i] = 0;
   }

   for(int i = m+1; i <= 2*m; i++)
   {
     wrapdebruijnsection[i] = 1;
   }

} // end createWraparoundDeBruijnSection


//*****************************************
// Method        getInputDec
// Return value   none
// Parameters     long long input
// Purpose        obtains user input decimal value that yields binary string
//                to be searched for in DeBruijn sequence
//*****************************************
void Necklaces::getInputDec(long long input)
{
   inputstrdec = input;
} //end getInputDec


//*****************************************
// Method        searchDeBruijnString
// Return value   long long position
// Parameters     none
// Purpose        searches for user input binary string in DeBruijn sequence
//                and returns its location
//*****************************************
long long Necklaces::searchDeBruijnString()
{
   long position = 0; // position from beginning of 111...1 necklace
   absposition = 0;
   int matchcount = 0; // counts how many matches there are between n-tuple and
deBruijn string
   long stepcount = 0; // counts how many step you take thru deBruijn section until there
is a match
```

```cpp
    cout << "\nsumperiod = " << sumperiod << endl;

    for(int i = 0; i < sumperiod; i++)
    {
      for(int j = 0; j < m; j++)
      {
        if(inputstring[j] == debruijnsection[i+j])
        {
          matchcount++;
        }
        else
        {
          matchcount = 0; //reset
          break;
        }
      } // end inner for loop

      if(matchcount = = m) //section of deBruijn sequence fully and uniquely matches user
input
      {
        break;
      }
      stepcount++; // keep counting number of steps needed to find a match
    }
    position = stepcount + totaldistance;

    if(inputstrdec = = generatePower2(m)-1) // accounts for all ones necklace
    {
      absposition = generatePower2(m); //last position in deBruin sequence read from right
to left
    }
    else if (inputstrdec = = 0) //accounts for all zeros necklace
    {
      absposition = 1; //first position in deBruin sequence read from right to left
    }
    else //everything else in between two extremes
    {
      absposition = generatePower2(m) - position;
    }

    return position;
} // end searchDeBruijnString
```

103

```cpp
//******************************************
// Method        findSequencePeriod
// Return value  int period
// Parameters    none
// Purpose       find period of a necklace
//******************************************

int Necklaces::findSequencePeriod()
{
  int period = 0;

  findDivisors(m); // need to find divisors first

  for(int i = 0; i < m; i++)
  {
    if(divisors[i]!=0)
    {
      for(int j = 0; j < (m / divisors[i]) - 1; j++)
      {
        period = 0; //reset
        for(int k = 0; k < divisors[i]; k++)
        {
          if(sequence[k] == sequence[k + (j+1)*divisors[i]])
          {
            period++; // period value keeps accumulating as long as there is a match
          }
          else
          {
            period = m;
            break; // exit k for loop if there is no match
          }
        }// end k for loop
        if(period == m)
        {
          break; // do not bother comparing remaining instances
        }
      } // end j for loop
    } // end outer if statement
    if(period == divisors[i])
    {
      break; // no need to check remaining divisors since we found period
    }
  } // end i for loop

  return period;
```

```
} // findSequencePeriod

//****************************************
// Method        shiftSequenceToNecklace
// Return value  none
// Parameters    none
// Purpose       given an arbitrary n-tuple, it rotates the sequence until it generates
necklace
//****************************************

void Necklaces::shiftSequenceToNecklace()
{
  int w = 0; // value of largest run of ones for cyclical rotation
  int w1 = 0; // window equal to running ones without wraparound
  int w2 = 0; //first half of wraparound window
  int w3 = 0; // second half of wraparound window

  int firstsum = 0; // number of ones from i to m-1 when i+w-1 > m-1
  int secondsum = 0; // number of ones from 0 to i+w-1
  int onessum = 0;
  long long temp1 = 0;
  long long temp2 = 0;

  temp1 = binToDec(); // initial decimal value

  w1 = countRunningOnes(0,m-1); //window size consisting of all ones

  for(int i = 0; i < w1; i++)
  {
    if(sequence[i]==1)
    {
      w2++;
    }
    else
    {
      break;
    }
  }

  for(int i = m-1; i >= m-w1; i--)
  {
    if(sequence[i]==1)
    {
      w3++;
    }
```

```cpp
      else
       {
         break;
       }
    }

  if(w1 > w2+w3) // need to find largest window size under cyclical rotation
  {
    w = w1;
  }
  else
  {
    w = w2+w3;
  }

  for(int i = 0; i < m; i++)
  {
    if(i+w-1 < m-1) // checks if end of window has reached end of sequence
    {
      if(countRunningOnes(i,i+w-1)== w)
      {
        decToBin(temp1);
        cycleshiftSequence(i);
        temp2 = binToDec(); // converts rotated sequence into a decimal value
        if(temp2 > temp1)
        {
          temp1 = temp2; //stores only largest decimal equivalent associated with "w"
ones
        }
      } // end if(i+w-1 <= m-1) statement
      else // may not need to use this
      {
        continue; // ignores all windows which have less than "w" running ones
      }
    }
    else if (i+w-1 >= m-1) //window exceeds sequence length and wraps around to
beginning
    {
      //cout << "\nInside if( " << i+w-1 << " >= " << m-1 << " ) statement..." << endl;
      //cout << "\n  i = " << i << endl;
      //cout << "\n  temp1 before cycleshift = " << temp1 << endl;
      //cout << "\n  Binary value of temp1: ";
      decToBin(temp1);
      //printSequence();
      //cout << "\n" << endl;
```

106

```cpp
      cycleshiftSequence(i);
      temp2 = binToDec(); // converts rotated sequence into a decimal value
      //cout << "\n  Binary value of temp1 after shifting by "<< i << ": ";
      //printSequence();
      //cout << "\n";
      //cout << "\n  temp2 = " << temp2 << endl;
      //cout << "\n  temp1 = " << temp1 << endl;
      if(temp2 > temp1)
      {
        //cout << "\n   ";
        //cout << temp2 << " > " << temp1 << endl;
        temp1 = temp2; //stores only largest decimal equivalent associated with "w" ones
      }
    }
  }// end for loop

  decToBin(temp1); // converts decimal equivalent to binary
} // end shiftSequenceToNecklace


//*****************************************
// Method      generateShiftingOnesSequence
// Return value   none
// Parameters    long long neckdec, decimal equivalent of necklace
//            long long numclump, number of missings after necklace
//            long distance, distance between adjacent signposts
//            long necknum, necklace number associated with each signpost
// Purpose      generates shifting ones sequence to use as signposts
// Note: Entries are stored in a matrix format that is read from left
//      to right where the last entry of the previous row precedes
//      the first entry of the current row. This ensures I have enough
//      memory to collect enough signposts
//*****************************************

void Necklaces::generateShiftingOnesSequence(long long necdec, long long clmpsz,
long distance, long necknum)
{
  long long nval = necdec;
  long long nclump = clmpsz;
  long dist = distance;
  long ncnt = necknum;

  if(colindex <= MAX_COLUMNS)
  {
    shiftonesarray[rowindex][colindex] = nval;
```

```
          shiftonesclumpsizearray[rowindex][colindex] = clmpsz;
          shiftonesdistancearray[rowindex][colindex] = distance;
          shiftonesncountarray[rowindex][colindex] = ncnt;
          colindex++; // increment column index
      }
    else
    {
      colindex = 0; // reset column index
      rowindex++; //increment row index
      if(rowindex <= MAX_LENGTH)
      {
        shiftonesarray[rowindex][colindex] = nval;
        shiftonesclumpsizearray[rowindex][colindex] = clmpsz;
        shiftonesdistancearray[rowindex][colindex] = distance;
        shiftonesncountarray[rowindex][colindex] = ncnt;
        colindex++; // increment column index
      }
    }
}// end generateShiftingOnesSequence


//****************************************
// Method       getWeight
// Return value   void
// Parameters     int num
// Purpose       obtain user weight input for shifting ones sigposts
//****************************************
void Necklaces::getWeight(int num)
{
  weight = num;
} // end getWeight


//****************************************
// Method       getNumEvenlySpacedSignposts
// Return value   void
// Parameters     int num
// Purpose       obtain number of evenly spaced sigposts from user
//****************************************
void Necklaces::getNumEvenlySpacedSignposts(int num)
{
  numevensignposts = num;
}// end getNumEvenlySpacedSignposts
```

```
//***************************************
// Method        computeDensity
// Return value   int nclass, largest number of runnning zeros
// Parameters     int start, int finish
// Purpose        counts the total number of ones in the sequence
// ** Note:  To compute number of zeros = length of sequence - computeDensity
//***************************************

int Necklaces::computeDensity(int start, int finish)
{
  int alpha = start;
  int omega = finish;
  int density = 0;

  for(int i = alpha; i <= omega; i++)
  {
    if(sequence[i] == 1)
    {
      density++;
    }
  }
  return density;
} // end computeDensity


//***************************************
// Method        testforNecklace
// Return value   bool foundnecklace
// Parameters     long long bininput
// Purpose        tests n-tuple to see if it is a necklace
//***************************************

bool Necklaces::testforNecklace(long long bininput);

{

bool foundnecklace = false;

long long ndecimal = bininput; // binary decimal associated with n-tuple

nstring = decToBin(ndecimal); // subroutine that converts decimal  value into a binary
                              // sequence

shiftSequenceToNecklace(); // subroutine that shifts a binary sequence into its necklace
                           // representative

shiftdecimal = binToDec();  // subroutine that converts a binary string into its decimal
```

109

```
                          // equivalent

if(shiftdecimal = = ndecimal)  // test to see if current n-tuple is a necklace

{

  foundnecklace = true;

}

else

{

  foundnecklace = false;

}

return foundnecklace;

// end testforNecklace

//****************************************
// Method        countNecklacesPerClass
// Return value   none
// Parameters     int ones, number of leading ones
// Purpose        separates the number of leading ones in each necklace
//                into distinct classes
//****************************************

void Necklaces::countNecklacesPerClass(int ones)
{
  int p = ones; //number of leading ones

  if(p != m-q)//checks if current ones count is different from last ones count
  {
    onescount[m-p]= 1; // ensures unique number of ones is counted only once
    q++;
  }
  else
  {
    onescount[m-p]++; //increase number of necklaces with same number of leading ones
  }

  if(p == 0) //accounts for all zeros string
  {
    onescount[m-p]= 1;
  }
}// end countNecklacesPerClass
```

```
//****************************************
// Method        generateChangeSequence
// Return value  none
// Parameters    long long oldec, decimal value of previous necklace
//               long long newdec, decimal value of current necklace
//               int indx, array index for sequence
// Purpose       counts the number of bits that differ between each necklace
//               and inserts these values into a sequence
//****************************************
void Necklaces::generateChangeSequence(long long olddec, long long newdec, int indx)
{
  int oldseq[MAX_LENGTH]; // binary sequence for previous necklace
  int newseq[MAX_LENGTH]; // binary sequence for current necklace
  int bitchange = 0; // counts the number of bit changes between both sequences
  int idx = indx; // column index for bit change sequence

  decToBin(olddec);
  for(int i=0; i < m; i++)
  {
    oldseq[i] = sequence[i];
  }
  decToBin(newdec);
  for(int i=0; i < m; i++)
  {
    newseq[i] = sequence[i];
  }

  for(int i=0; i < m; i++)
  {
    if(oldseq[i]!= newseq[i])
    {
      bitchange++;
    }
  }
  bitchangeseq[idx]= bitchange;
}// end generateChangeSequence


//****************************************
// Method        binToDec
// Return value  int dec
// Parameters    none
// Purpose       converts binary number of a sequence into its decimal equivalent
//****************************************
```

111

```cpp
long long Necklaces::binToDec()
{
  long long dec = 0;
  for(int i = 0; i < m; i++)
  {
    dec = dec + generatePower2(m-1-i)*sequence[i];
  }
  return dec;
}// end binToDec



//****************************************
// Method        decToBin
// Return value  none
// Parameters    long long decimal input
// Purpose       converts decimal input into its binary equivalent sequence
//****************************************

void Necklaces::decToBin(long long input)
{
  int r = 0; //number of leading ones
  int j = m-1; //start at LSB
  initNecklace(r);
  long long quotient = input;  //find a way to change to long long
  while(quotient !=0 && j >= 0)
  {
    if (quotient == 0)
    {
      sequence[j] = 1; // accounts for the MSB where 1/2 = 0 + (rem=1)
    }
    else
    {
      sequence[j] = quotient%2; //remainder should be 0 or 1
    }
    j--;
    quotient = quotient /2;
  }
}// end decToBin



//****************************************
// Method        createUnobservedSequences
// Return value  none
// Parameters    int dec1, int dec2
// Purpose       generates preabsorbed necklaces that are skipped by theta
```

```cpp
//          and which exist between the last necklace in a given class
//          and the first necklace in the next adjacent class
//***************************************

long Necklaces::createUnobservedSequences(int dec1, int dec2)
{
  long index = 0;
  long index1 = dec1; // decimal equivalent of last necklace in a given class
  long index2 = dec2; // decimal equivalent of first necklace in next class
  long seqcount = 0; //counts sequences

  //index = index1-2;
  index = index1;
  while (index >= index2)
  {
    decToBin(index); //generates binary equivalent of decimal
    printSequence();
    cout << "decimal value = " << index << endl;
    index = index - 2; // substract 2 to skip over binary strings ending in 1
    seqcount++;
    cout << "\n";
  }
  return seqcount;
} // end createUnobservedSequences


//***************************************
// Method       findDivisors
// Return value   none
// Parameters     int num
// Purpose       find all divisors for an integer n
//***************************************

void Necklaces::findDivisors(int num)
{
  for(int i=0; i < MAX_LENGTH+1; i++)
  {
    divisors[i] = 0;
  }

  int number = num;
  for(int i=0; i < number; i++) // does not include m itself
  {
    if(number%(i+1)== 0) // checks for divisors; i+1 ensures no division by zero
    {
```

113

```cpp
        divisors[i] = i+1; // assigns divisors to the (i-1)th place in array
      }
    }
}// end findDivisors



//****************************************
// Method       printDivisors
// Return value   none
// Parameters    int num
// Purpose       prints all divisors for an integer n
//****************************************

void Necklaces::printDivisors(int num)
{
  int number = num;
  cout << "\nDivisors of " << number << "...\n" << endl;
  for(int i=0; i < number; i++)
  {
    if(divisors[i]!=0)
    {
      cout << setw(3) << divisors[i] << " "; // divisors printed
    }
  }
} // end printDivisors



//****************************************
// Method       findGCD
// Return value   int gcd
// Parameters    int num1, num2
// Purpose       find greatest common divisor (gcd) for two integers
//****************************************

int Necklaces::findGCD(int num1, int num2)
{
  int first = num1;
  int second = num2;
  int factor1[MAX_LENGTH + 1];
  int factor2[MAX_LENGTH + 1];

  for(int i=0; i < MAX_LENGTH + 1; i++)
  {
    factor1[i] = 0;
    factor2[i] = 0;
```

114

```
}

int factor1cnt = 0;
findDivisors(first);
for(int i=0; i < MAX_LENGTH + 1; i++)
{
  if(divisors[i]!=0) //only collect non-zero divisors
  {
    factor1[i]= divisors[i];
    factor1cnt++;
  }
}

int factor2cnt = 0;
findDivisors(second);
for(int i=0; i < MAX_LENGTH + 1; i++)
{
  if(divisors[i] !=0) //only collect non-zero divisors
  {
    factor2[i]= divisors[i];
    factor2cnt++;
  }
}

int factorcnt = 0;
if(factor1cnt > factor2cnt)
{
  factorcnt = factor1cnt;
}
else
{
  factorcnt = factor2cnt;
}

int gcd = 0;

for(int i=0; i < factorcnt; i++)
{
  for(int j=0; j < factorcnt; j++)
  {
    if(factor2[j] == factor1[i])
    {
      if(factor2[j] > gcd)
      {
        gcd = factor2[j];
```

```
            }
          }
        }
    }

    return gcd;
} // end findGCD

//****************************************
// Method       eulerTotient
// Return value   long long Zn
// Parameters     int num
// Purpose        Calculate number of necklaces for a given n
//****************************************

long long Necklaces::eulerTotient(int num)
//double Necklaces::eulerTotient(int num)
{
    int size = num;
    long long Zn = 0;
    long long totient[MAX_LENGTH+1]; // contains number of integers relatively prime
to the divisor including "1"
    //double Zn = 0;
    //double totient[MAX_LENGTH+1]; // contains number of integers relatively prime to
the divisor including "1"

    int divisorarray[MAX_LENGTH+1];

    for(int i=0; i <= MAX_LENGTH; i++)
    {
        totient[i] = 0;
        divisorarray[i] = 0;
    }

    int divisorcnt = 0;
    findDivisors(size);
    for(int i=0; i <= MAX_LENGTH; i++)
    {
        if(divisors[i]!=0) //only collect non-zero divisors
        {
            divisorarray[i] = divisors[i];
            divisorcnt++;
        }
    }
```

```cpp
    for(int i=0; i <= size; i++)
    {
      if(divisorarray[i] != 0)
      {
        for(int j = 1; j < divisorarray[i]; j++)
        {
          if(findGCD(divisorarray[i],j)== 1)
          {
            totient[i]++;
          }
        }
      }
    }

    totient[0] = 1; // to include "1"
    long long sum = 0;
    //double sum = 0;
    for(int i=0; i <= size; i++)
    {
      if(totient[i]!=0)
      {
        sum = sum + totient[i]*generatePower2(m/divisorarray[i]);
      }
    }
    Zn = sum / size;

    cout << "\n";
    return Zn;
}// end eulerTotient


//*****************************************
// Method        generatePower2
// Return value   long long power2val
// Parameters     long long exponent
// Purpose        generates 2 raised to any integer
//*****************************************

long long Necklaces::generatePower2(long long exponent)
{
    long long power2val = 1;
    long long expint = exponent;
    for(int i = 0; i < expint; i++)
    {
     power2val *= 2;  //calculates power of 2
```

```
    }

    return power2val;

}// end generatePower2
```

```
void Necklaces::displayNecklaceInfo(int a, int b, long long c, long long d)
{
  int column = a;
  int clssnum = b;
  long long val = c;
  long long totalclumps = d;

  cout << "\nclass number = " << clssnum << endl;
  cout << "\nnumber of clumps so far = " << totalclumps << endl;
  cout << "\nclumpcntarray[" << clssnum << "]= " << totalclumps << endl;
  cout << "\nncountarray["<< clssnum << "][" << column << "] = " << ncount <<
endl;
  //cout << "\ndecimalarray["<< clssnum << "][" << column << "] = " <<
decimalarray[clssnum][column] << endl;
  cout << "\n";
}// end displayNecklaceInfo
```

```
void Necklaces::displayPower2Array()
{
  int a1 = 3;
  int a2 = 2;
  for(int i=a1; i<=m; i++)
  {
```

118

```cpp
      for(int j=a2; j<i; j++)
      {
        power2[i] *= 2;  //calculates power of 2
      }
    }

    cout << "\n\nPower of 2 approximation for each class number:\n" << endl;
    for(int i=0; i <= m; i++)
    {
      cout << setw(OUTPUT_CELL_SIZE)<< power2[i] <<" ";
    }
}// end displayPower2Array


//****************************************
// Method       displayDifference
// Return value   none
// Parameters     none
// Purpose       calculates and displays difference between power2 approximation and
number of necklaces
//****************************************

void Necklaces::displayDifference()
{
  for(int i = 0; i <= m; i++)
  {
    diff[i] = power2[i] - onescount[i];
  }

  cout << "\n\nNumber of differing necklaces per class number:\n" << endl;
  for(int i=0; i <= m; i++)
  {
    cout << setw(OUTPUT_CELL_SIZE)<< diff[i] <<" ";
  }
}// end displayDifference


//****************************************
// Method       displayNumbNeck
// Return value   none
// Parameters     none
// Purpose       displays number of necklaces per class
//****************************************

void Necklaces::displayNumbNeck()
```

```cpp
{
  cout << "\nNumber of necklaces per class number:\n" << endl;
  for(int i=0; i <= m; i++)
  {
    cout << setw(OUTPUT_CELL_SIZE)<< onescount[i] <<" ";
  }
}// end displayNumbNeck
//****************************************
// Method        displayClassNumbers
// Return value   none
// Parameters     none
// Purpose        displays class numbers of necklaces in decreasing order
//****************************************

void Necklaces::displayClassNumbers()
{
  cout << "\nDecreasing class numbers:\n" << endl;
  for(int i = m; i >= 0; i--)
  {
    cout << setw(OUTPUT_CELL_SIZE) << i <<" "; // displays decreasing leading ones
  }
  cout << endl;
}// end displayClassNumbers



//****************************************
// Method        displayNcount
// Return value   none
// Parameters     none
// Purpose        displays total number of necklaces
//****************************************

void Necklaces::displayNcount()
{
  cout << "\n\n*** Total number of theta generated binary " << m << "-tuples = "
<< ncount + noncount << " ***" <<endl;
  cout << "\nNumber of necklaces = " << ncount << endl;
  cout << "\nNumber of theta generated non-necklaces = " << noncount << endl;
  cout << "\nNumber of missing even " << m << "-tuples = " << totmissing << endl;
  cout << "\nNumber of missing even " << m << "-tuples + number of necklaces = "
<< totmissing + ncount << endl;
  cout << "\nMaximum distance of necklaces = " << distance << endl;
  cout << "\n";
}// end displayNcount
```

```
//*****************************************
// Method        displayClumpsPerClass
// Return value   none
// Parameters     none
// Purpose        displays total number of missing even n-tuples per class
//*****************************************

void Necklaces::displayClumpsPerClass()
{
  long clumptotal = 0;
  cout << "\n\nNumber of missing clumps per class number:\n" << endl;
  for(int i = m; i >= 0; i--)
  {
    cout << setw(OUTPUT_CELL_SIZE)<< clumpcntarray[i] <<" ";
    clumptotal +=  clumpcntarray[i];
  }
  cout << "\n\nTotal number of clumps = " << clumptotal << " for threshold = " <<
threshold << endl;

}// end displayClumpsPerClass


//*****************************************
// Method        displayClumpDistribution
// Return value   none
// Parameters     none
// Purpose        displays various clump sizes per class
//*****************************************

void Necklaces::displayClumpDistribution()
{
  cout << "\n\n\nDistribution of missing necklaces for n = " << m << " and threshold
= " << threshold << ":" << endl;
  cout << "\nEach number below represents the size of the clump." << endl;
  cout << "\n";

  long long summissing = 0;
  long long nummiss = 0;
  int numclump = 0;
  int listwidth = 4; // displays only so many triples across
  // triple = (Necklace Number, Decimal Equivalent of Necklace, Number of Missings
after Necklace)

  // percent = percentage of the maximum clump size
```

121

```cpp
    cout << "Clump sizes which are greater than " << num << " percent of the
maximum clump size are shown. " << endl;
    cout << "\nMaximum clumpsize for all the classes = " << maxclumpsize << endl;
    cout << "\nNote: Lower necklace decimal value = Upper necklace decimal value -
2*(number of missings)\n" << endl;
  cout << "\n";
  for (int i = 0; i < m; i++)
  {
    cout << "\nClass "<< i << ":  ";
    summissing = 0; //resetting count
    nummiss = 0;
    for (int j = 0; j < MAX_COLUMNS; j++)
    {
      if(clumpsizearray[i][j]!= 0)
      {
        if(clumpsizearray[i][j] >=  maxclumpsizearray[i]*percent)
        {
          cout << setw(OUTPUT_CELL_SIZE) << clumpsizearray[i][j] << " " ;
          summissing += clumpsizearray[i][j];
          nummiss ++;
        }
      }
    }
    if(nummiss!=0)
    {
      cout << "\n\nTotal number of missing even " << m << "-tuples exceeding
threshold of " << threshold << " = " << summissing << endl;
      cout << "\nTotal number of missing even " << m << "-tuples below threshold
of " << threshold << " = " << missingbelowthreshold[i] << endl;
      cout << "\nTotal number of clumps = " << nummiss << endl;
      cout << "\nMaximum clump size for this class = " << maxclumpsizearray[i] <<
endl;
      cout << "\n(Necklace Number, Decimal Equivalent of Necklace, Number of
Missings after Necklace)"<< endl;
      cout << "\n";
      numclump = 0; //resetting count
      for (int j = 0; j < MAX_COLUMNS; j++)
      {
        //if(ncountarray[i][j] != 0 && missingarray[i][j]!= 0 && decimalarray[i][j]!= 9)
        {
          //cout << setw(OUTPUT_CELL_SIZE) << "(" << ncountarray[i][j] << " , " <<
decimalarray[i][j]<< " , " << missingarray[i][j] << ")" ;
          numclump++;
          if((j+1)%listwidth == 0)
          {
```

```cpp
                cout << "\n\n";
            }
        }
      } // end for
    } // end if
    else
    {
      cout << "No missings in this class." << endl;
    }
    cout << "\n\n" << endl;
  }//end outside for
}// end displayClumpDistribution


//****************************************
// Method        displayDecimalArray
// Return value   none
// Parameters     none
// Purpose        displays decimal equivalent of necklaces associated with clumps per class
//****************************************

void Necklaces::displayDecimalArray()
{
  int listwidth = 4; // displays only so many triples across
  cout << "\n\n\nDistribution of decimal equivalents for missing clumps for n = " <<
m << " and threshold = " << threshold << ":" << endl;
  cout << "\n";
  for (int i = 0; i < m; i++)
  {
    for (int j = 0; j < MAX_COLUMNS; j++)
    {
      //if(decimalarray[i][j]!= 9)
      {
        //cout << setw(OUTPUT_CELL_SIZE) << decimalarray[i][j]<< " " ;
        if((j+1)%listwidth == 0)
        {
          cout << "\n\n";
        }
      }
    }
  } // end for

  cout << "\n\n\nList of necklaces associated with clumps for n = " << m << " and
threshold = " << threshold << ":" << endl;
  cout << "\n";
```

123

```cpp
      for (int i = m-1; i >= 0; i--)
      {
        for (int j = 0; j < MAX_COLUMNS; j++)
        {
          //if(decimalarray[i][j]!= 9)
          {
            //decToBin(decimalarray[i][j]);
            printSequence();
            cout << "\n\n";
          }
        }
      }
}// end displayDecimalArray


//****************************************
// Method        displayShiftingOnesSignposts
// Return value   none
// Parameters     none
// Purpose        displays decimal and binary equivalents of shifting ones signposts
//                along with clump sizes
//****************************************

void Necklaces::displayShiftingOnesSignposts()
{
  int numshiftones = 0; // total number of shifting ones signposts
  int loopcnt = 0; // keeps track of long long for loop count
  long long shiftonesmissingsum = 0; //sums missings
  char answer1;
  char answer2;
  cout << "\n\n\nSignposts with shifting ones for n = " << m << endl;
  cout << "\n";

  cout << "Do you want to see the signposts? (Enter 'y' or 'n'): ";
  cin >> answer1;
  cout << "\n";

  cout << "Do you want to see the various clump sizes? (Enter 'y' or 'n'): ";
  cin >> answer2;
  cout << "\n";

  for(int i = m-1; i >= 0; i--)
  {
    for (int j = 0; j < MAX_COLUMNS; j++)
    {
```

```cpp
        if (shiftonesarray[i][j]!= mark)
        {
          numshiftones++;
          if(answer1 == 'Y' || answer1 == 'y') // displays binary strings, decimal values and
clump sizes
          {
            decToBin(shiftonesarray[i][j]);
            printSequence();
            cout << "   decimal value = "<< shiftonesarray[i][j] << ",   ";
            cout << setw(OUTPUT_CELL_SIZE) << "clump size = "<<
shiftonesclumpsizearray[i][j] << endl;
            shiftonesmissingsum += shiftonesclumpsizearray[i][j];
            cout << "\n";
          }
          else // displays only various clump sizes
          {
          loopcnt++;
          if (shiftonesclumpsizearray[i][j]!= 0)
          {
            if(answer2 == 'Y' || answer2 == 'y')
            {
              if (loopcnt <= 1) // ensures output statement is not repeated more than once
              {
                cout << "Clump sizes for shifting ones signposts: " << endl;
                cout <<"\n";
              }
              cout << setw(OUTPUT_CELL_SIZE) << shiftonesclumpsizearray[i][j] << "
";
            }
            shiftonesmissingsum += shiftonesclumpsizearray[i][j];
          }
        }
        }
      else
      {
        continue;
      }
    }
  }

  binlength = generatePower2(m-1);
  int k1 = 4; // equipment cost
  int k4 = 1; // temporal cost

  petriumilestones = floor(binlength / m); // assume max period = m
```

125

```
   milestonedistance = floor(binlength /shiftonesclump); // distance between milestones
using number of shifting ones signposts
   topt = ceil(sqrt((k1/k4)*binlength));// gives bad answer

   cout << "\n\nNumber of shifting ones signposts associated with missings (using
array) = " << numshiftones << endl;
   cout << "\nNumber of shifting ones signposts associated with missings (using pure
count) = " << shiftonesclump << endl;
   cout << "\nNumber of shifting ones signposts NOT associated with missings = " <<
shiftonesnoclump << endl;
   cout << "\nTotal number of missings accounted for by shifting ones = " <<
shiftonesmissingsum << endl;
   cout << "\n\nPetriu length is 2^" << m << " - 1 = " << generatePower2(m) - 1 <<
endl;
   cout << "\nTotal number of Petriu's signposts (assuming the max separation
distance is " << m << " bits) = " << petriumilestones << endl;
   cout << "\nPetriu distance between milestones (assuming the number of milestones
is " << shiftonesclump << ") = " << milestonedistance << endl;
   cout << "\nOptimal Petriu distance between milestones = " << topt << endl;
}// end displayShiftingOnesSignposts


//****************************************
// Method        displayShiftingOnesDistances
// Return value   none
// Parameters     none
// Purpose        displays distances between shifting ones signposts
//****************************************

void Necklaces::displayShiftingOnesDistances()
{
   long reldistance = 0; // relative distance
   long currentdistance = 0;
   long sumdistance = 0; // sum of rleative distance elements
   long numdistance = 0; // number of distance elements
   long avedistance = 0;
   int colspread = 15; // width of data elements
   int spreadcount = 0; // number of data elements used to spread data matrix evenly
   long shiftonespetriunumbigger = 0; // number of distances greater than petriu distance
   long shiftonespetriunumlesser = 0; // number of distances less than or equal to petriu
distance

   cout << " "; // adjustment for first row to make entries look even

   for(int i = 0; i < MAX_LENGTH; i++)
```

126

```cpp
  {
    for(int j = 0; j < MAX_COLUMNS; j++)
    {
      if(shiftonesdistancearray[i][j] != 0)
      {
        reldistance = shiftonesdistancearray[i][j] - currentdistance;
        cout << setw(OUTPUT_CELL_SIZE) << reldistance;
        currentdistance = shiftonesdistancearray[i][j];
        sumdistance += reldistance;
        numdistance++;
        spreadcount++;
        //if(reldistance > milestonedistance)
        if(reldistance > topt)
        {
          shiftonespetriunumbigger++;
        }
        else
        {
          shiftonespetriunumlesser++;
        }
        if(spreadcount == colspread)
        {
          cout << "\n " ;
          spreadcount = 0; //reset
        }
        else
        {
          cout << setw(OUTPUT_CELL_SIZE) << " " ;
        }
      }
    }
  }

  avedistance = sumdistance / numdistance;
  cout << "\n\navedistance = " << avedistance;
  //cout << "\n\nNumber of signposts whose distance is greater than " <<
milestonedistance << " = " << shiftonespetriunumbigger << endl;
  //cout << "\n\nNumber of signposts whose distance is less than or equal to " <<
milestonedistance << " = " << shiftonespetriunumlesser << endl;
  cout << "\n\nNumber of signposts whose distance is greater than " << topt << " = "
<< shiftonespetriunumbigger << endl;
  cout << "\n\nNumber of signposts whose distance is less than or equal to " <<
topt<< " = " << shiftonespetriunumlesser << endl;

}// end displayShiftingOnesDistances
```

```cpp
//****************************************
// Method        displayBitChangeSequence
// Return value  none
// Parameters    none
// Purpose       displays bit changes between necklaces
//****************************************

void Necklaces::displayBitChangeSequence()
{
  cout << "\n\nThe following sequence shows the number of bit changes between
each pair of necklaces. " << endl;
  cout << "\n";

  for(int i=0; i < MAX_COLUMNS; i++)
  {
    if (bitchangeseq[i]!= 0)
    {
      cout << setw(OUTPUT_CELL_SIZE) << bitchangeseq[i] << " ";
    }
  }
}// end displayBitChangeSequence


//****************************************
// Method        displayFirstNecklace
// Return value  none
// Parameters    none
// Purpose       displays prior necklace in deBruijn section
//****************************************

void Necklaces::displayFirstNecklace()
{
  for(int i = 0; i < m; i++)
  {
    if(firstnecklace[i]!= 9)
    {
      cout << firstnecklace[i] << " ";
    }
  }
  cout <<"\n";
  cout << "\nfirstperiod = " << firstperiod << endl;

}// end displayFirstNecklace
```

128

```cpp
//*****************************************
// Method        displayMiddleNecklace
// Return value  none
// Parameters    none
// Purpose       displays input necklace in deBruijn section
//*****************************************

void Necklaces::displayMiddleNecklace()
{
  for(int i = 0; i < m; i++)
  {
    if(middlenecklace[i]!= 9 )
    {
      cout << middlenecklace[i] << " ";
    }
  }
  cout <<"\n";
  cout << "\nmiddleperiod = " << middleperiod<< endl;

}// end displayMiddleNecklace


//*****************************************
// Method        displayLastNecklace
// Return value  none
// Parameters    none
// Purpose       displays post necklace in deBruijn section
//*****************************************

void Necklaces::displayLastNecklace()
{
  for(int i = 0; i < m; i++)
  {
    if(lastnecklace[i]!= 9 )
    {
      cout << lastnecklace[i] << " ";
    }
  }
  cout <<"\n";
  cout << "\nlastperiod = " << lastperiod << endl;
  cout << "\nsumperiod = " << sumperiod << endl;

}// end displayLastNecklace
```

```cpp
//****************************************
// Method        displaydeBruijnSection
// Return value   none
// Parameters     none
// Purpose        displays deBruijn sectional associated with user input necklace
//****************************************

void Necklaces::displaydeBruijnSection()
{
    for(int i = 0; i < firstperiod; i++)
    {
      if(debruijnsection[i]!= 9)
      {
        cout << debruijnsection [i] << " "; // prints the first third
      }
    } // end for loop
    cout << "^ ";

    for(int i = 0 ; i < middleperiod ; i++)
    {
      if(debruijnsection[i + firstperiod]!= 9)
      {
        cout << debruijnsection [i + firstperiod] << " "; // prints the second third
      }
    } // end for loop
    cout << "^ ";

    for(int i = 0; i < lastperiod; i++)
    {
      if(debruijnsection[i + firstperiod + middleperiod]!= 9)
      {
        cout << debruijnsection [i + firstperiod + middleperiod] << " "; // prints the last
third
      }
    } // end for loop
  //} // end else

    cout << "\n"<< endl;

} // end displaydeBruijnSection

//****************************************
// Method        displayUnobservedSequences
// Return value   none
// Parameters     none
```

```cpp
// Purpose      displays all missings (even binary n-tuples) between any 2 nunmbers
//****************************************

void Necklaces::displayUnobservedSequences()
{
  char answer;
  int input1 = m ; //need to change to long long
  int input2 = 0; //need to change to long long
  long totseq = 0;

  cout << "\n\nWould you like to see all the binary sequences between 2 numbers? (y
or n) ";
  cin >> answer;
  if (answer == 'Y' || answer == 'y')
  {
    cout << "\nThis section will help you see all the sequences between 2 numbers."
<< endl;
    cout << "\nPlease enter 0 to quit or an even positive number less than " <<
generatePower2(input1)-1 << ": ";
    cin >> input1;

    while(input1%2 !=0)
    {
      cout << "\nPlease enter another positive number: ";
      cin >> input1;
    }

    if(input1 != 0)
    {
    cout << "\nPlease enter another number less than or equal to " << input1 << ": ";
    cin >> input2;
    cout << "\nGenerating unobserved sequences..." << endl;
    cout << "\n";
    totseq = createUnobservedSequences(input1,input2);
    cout << "\n\nTotal number of sequences = " << totseq;
    }
  }
}// end displayUnobservedSequences


//****************************************
// Method       outputData
// Return value   none
// Parameters     none
// Purpose       provide numerical data useful for analysis
```

131

```
//***************************************

void Necklaces::outputData()
{
  displayNcount();

  displayClassNumbers();

  displayNumbNeck();

  displayPower2Array();

  displayDifference();

  displayClumpsPerClass();

  displayDecimalArray();

  displayClumpDistribution();

  displayShiftingOnesSignposts();

  displayBitChangeSequence();

 displayUnobservedSequences();

}// end outputData

// end of file Necklaces.cpp
```

2.   Thesismain CPP File

```
//**********************************************************************
// Thesismain.cpp
//
// LT John Ortiz
//
//
// Project: Thesis
// Operating Environment: Windows XP Home
// Compiler: Visual Studio .NET
// Date: 08 December 2005
// Description: This program generates necklaces and performs analysis on them
//
// Inputs: This program takes as inputs the length of the necklace
// Outputs: This program displays a table of necklaces
```

```cpp
// Processes: none
// Assumptions: none
//
// Warnings: none
//
//**********************************************************************

#include "Necklaces.h"
#include "IOThesis.h"
#include "SieveSizeException.h"
#include "math.h"
#include <iostream>
#include <iomanip>
#include <cctype>
#include <fstream>

using std::ostream;
using std::ofstream; //output file stream
using namespace std;

//****************************************
// Method        checkInput(input2);
// Return value  int input2
// Parameters    int input2
// Purpose       checks if input2 is valid
//****************************************
int checkInput(int input1, int input2)
{
  int oneslength = input2;
  int limit = input1;
  while (oneslength < 0 || oneslength > limit)
  {
    cout << "Please enter a number between 0 and " << limit << " for the initial
sequence." << endl;
    cin >> oneslength;
  }
  return oneslength;
} // end checkInput

int main()
{
  long long input1 = 0;
  long long input2 = 0;
  long long input3 = 0;
```

```cpp
    char input;
    char answer;
    long long inputstep = 0;
    int maxinput  = 63;
    int num = 0;
    int num1 = 0;
    int num2 = 0;
    int intshift = 0;
    int onesrun = 0;
    int decimal = 0;
    int necknum = 0;
    int cellsize = 3;
    int bound = 0; // value of threshold
    int length = 0; //length of binary string
    int permaxclump = 0; // percentage of the maximum clump size
    long long expinput = 0;
    long long expoutput = 0;
    bool flag = false;

     cout << "This program will help you generate a list of necklaces" << endl;
    cout << "to enable you to perform analysis on them." << endl;
    flag = false;
    while (flag == false)
    {
      cout << "\n\nPlease enter an integer for your necklace length between 1 and 50: ";
      try
        {
        length = IOThesis::inputInt(); // verifies user input dimension is valid
         flag = true;
      }
        catch(SieveSizeException &sieveSizeException)
        {
          cout << "Error occured: " << sieveSizeException.what() << endl;
           flag = false;
        }
    }
  cout << "\n\nControlling the threshold (smallest size of missings displayed) and " <<
endl;
  cout << "\nthe percentage of the maximum clump size can help manage the limitations
" << endl;
  cout << "\nof memory when displaying the missings.  If you don't want to adjust the "
<< endl;
  cout << "\nthreshold or percentage, enter '0'. Else, hit any key: ";
  cin >> answer;
```

```cpp
    if (answer != '0')
    {
      cout << "\n\nPlease enter the threshold: ";
      cin >> bound;

      cout << "\nPlease enter the percentage: ";
      cin >> permaxclump;
    }

    Necklaces necklace = Necklaces(length, bound, permaxclump); //Necklace class
instantiated

    // Relatively prime test
    cout << "\n" << endl;
    cout << "Would you like to know whether two integers are relatively prime? ";
    cin >> answer;
    while (answer == 'y' || answer == 'Y')
    {
      cout << "\nPlease enter your first integer: ";
      cin >> input1;
      cout << "\nPlease enter your second integer: ";
      cin >> input2;
      cout <<"\nThe greatest common divisor between " << input1 << " and " << input2 <<
" is " << necklace.findGCD(input1,input2);
      if(necklace.findGCD(input1,input2)==1)
      {
        cout << "\n\nThe integers are relatively prime." << endl;
      }
      else
      {
        cout << "\n\nThe integers are not relatively prime." << endl;
      }
      cout << "\n\nWould you like to know whether another two integers are relatively
prime? ";
      cin >> answer;
    } // end relatively prime test

    //Calculating divsiors of binary string length
    cout << "\n" << endl;
    cout << "Would you like to know number of necklaces for and the divisors of n = " <<
length << " ? ";
    cin >> answer;
    if (answer == 'y' || answer == 'Y')
    {
      necklace.findDivisors(length);
```

```cpp
      necklace.printDivisors(length);
      cout << "\nCalculated value of number of necklaces = " <<
necklace.eulerTotient(length) << endl;
   }

   cout << "\nWould you like run the Necklace algorithm? ";
   cin >> answer;
   if (answer == 'y' || answer == 'Y')
   {
   cout << "\nWould you like to generate signposts? ";
   cin >> answer;

   if (answer == 'y' || answer == 'Y')
   {
   cout << "\n\nThis next section enables you to generate signposts using the shifting ones
strategy."<< endl;
   cout << "\nPlease enter a '1' if you want to use the shifting ones signposts: " ;
   cin >> answer;
   if (answer == '1')
   {
      cout << "\n\nPlease enter the weight you would want to use: ";
      cin >> num;
      necklace.getWeight(num);
   } // end if
   cout << "\n" << endl;

   cout << "If you want to generate all the necklaces, enter '1' or else hit any other key: ";
   cin >> answer;
   if (answer != '1')
   {
      cout << "\nPlease enter a number between 0 and " <<
necklace.generatePower2(length)-1 << " for the intial decimal: ";
      cin >> input1;
      cout << "\nPlease enter a number between 0 and " <<
necklace.generatePower2(length)-1 << " for the final decimal: ";
      cin >> input2;
      cout << "\nPlease enter a number between 0 and " <<
necklace.generatePower2(length)-1 << " for the input decimal: ";
      cin >> input3;
   }
   else //default settings to generate all the necklaces
   {
      input1 = necklace.generatePower2(length)-1;
      input2 = 0;
      input3 = 0;
```

136

```cpp
        }

    cout << "\n\nPlease enter 'y' or 'Y' if you would like to see the necklaces. Otherwise,
enter any key: ";
    cin >> input;

    if (input == 'y' || input == 'Y')
    {
        flag = true;
        cout << "\nGenerating Necklaces..." << endl;
    }
    else
    {
        flag = false;
    }
    necklace.runNecklaceAlgorithm(input1, input2, input3, flag);
    necklace.outputData();
    }

    cout << "\n\nDo you want to test cycle shifting a sequence? ";
    cin >> answer;

    while (answer == 'Y' || answer == 'y')
    {
        cout << "\nPlease enter a decimal to generate a binary sequence: ";
        cin >> input1;
        necklace.decToBin(input1);
        cout << "\n";
        necklace.printSequence();
        cout << "\n";
        cout << "\nPlease enter an integer to test the cyclic shift: ";
        cin >> intshift;
        necklace.cycleshiftSequence(intshift);
        cout << "\n";
        necklace.printSequence();
        cout << "\n";
        cout << "\n\nDo you want to test cycle shifting another sequence? ";
        cin >> answer;
    }

    cout << "\n\nDo you want to compute the density of a sequence? ";
    cin >> answer;
    while (answer == 'Y' || answer == 'y')
    {
        cout << "\nPlease enter a decimal to generate a binary sequence: ";
```

```cpp
    cin >> input1;
    necklace.decToBin(input1);
    cout << "\n";
    necklace.printSequence();
    cout << "\n";
   cout << "Density = " << necklace.computeDensity(0, length-1) << endl;
   cout << "\n\nDo you want to compute the density of another sequence? ";
   cin >> answer;
}

cout << "\n\nDo you want to find the period of a sequence? ";
cin >> answer;
while (answer == 'Y' || answer == 'y')
{
  cout << "\nPlease enter a decimal to generate a binary sequence: ";
   cin >> input1;
   necklace.decToBin(input1);
    cout << "\n";
   necklace.printSequence();
  necklace.findDivisors(length);
   cout << "\n\n";
  cout << "Period = " << necklace.findSequencePeriod() << endl;
   cout << "\n\nDo you want to find the period of another sequence? ";
   cin >> answer;
}

cout << "\n\nDo you want to find the necklace of a sequence? ";
cin >> answer;
while (answer == 'Y' || answer == 'y')
{
  cout << "\nPlease enter a decimal to generate a binary sequence: ";
   cin >> input1;
   necklace.decToBin(input1);
  cout << "\nSequence prior to shifting: ";
  necklace.printSequence();
  cout << "\n";
  necklace.shiftSequenceToNecklace();
  cout << "\nNecklace: " << endl;
  cout << "\n";
  necklace.printSequence();
  cout << "\n\nDo you want to find the necklace of another sequence? ";
  cin >> answer;
}

cout << "\n\nDo you want to see the distances between the shifting ones signposts? ";
```

```cpp
    cin >> answer;
    if (answer == 'Y' || answer == 'y')
    {
      cout << "\n";
      necklace.displayShiftingOnesDistances();
    }

  cout << "\n\nDo you want to see the deBruijn section surrounding your necklace input?
";
    cin >> answer;
    while (answer == 'Y' || answer == 'y')
    {
      cout << "\nPlease enter a decimal between 0 and " <<
necklace.generatePower2(length)-1 << " to generate a binary sequence: ";
      cin >> input3;
      necklace.decToBin(input3);
      cout << "\nSequence prior to shifting: ";
      necklace.printSequence();
      cout << "\n";
      necklace.shiftSequenceToNecklace();
      cout << "\nNecklace: " << endl;
      cout << "\n";
      necklace.printSequence();
      input3 = necklace.binToDec();
      cout << "\n\ndecimal associated with necklace = " << input3 << endl;

      cout << "\n\nPlease enter another decimal greater than " << input3 << ": ";
      cin >> input1;
      necklace.decToBin(input1);
      cout << "\nSequence prior to shifting: ";
      necklace.printSequence();
      cout << "\n";
      necklace.shiftSequenceToNecklace();
      cout << "\nNecklace: " << endl;
      cout << "\n";
      necklace.printSequence();
      input1 = necklace.binToDec();
      cout << "\n\ndecimal associated with necklace = " << input1 << endl;

      cout << "\n\nPlease enter another decimal less than " << input3 << ": ";
      cin >> input2;
      necklace.decToBin(input2);
      cout << "\nSequence prior to shifting: ";
      necklace.printSequence();
      cout << "\n";
```

```cpp
    necklace.shiftSequenceToNecklace();
    cout << "\nNecklace: " << endl;
    cout << "\n";
    necklace.printSequence();

    cout << "\n\nRunning Necklace Algorithm..." << endl;
    necklace.runNecklaceAlgorithm(input1, input2, input3, false);

    cout << "\nFirst necklace: " << endl;
    necklace.displayFirstNecklace();

    cout << "\nMiddle necklace: " << endl;
    necklace.displayMiddleNecklace();

    cout << "\nLast necklace: " << endl;
    necklace.displayLastNecklace();

    necklace.createDeBruijnSection();

    cout << "\nDeBruijn Section..." << endl;
    cout << "\n";
    necklace.displaydeBruijnSection();

    cout << "\n\nDo you want to search the deBruijn section for a particular " << length
<< "-tuple? ";
    cin >> answer;
    while (answer == 'Y' || answer == 'y')
    {
      int pos = 0;
      int per = 0;
      cout << "\nPlease enter a decimal value between " << input1 << " and " << input2
<< ": ";
      cin >> input3;
      necklace.getInputDec(input3);
      cout << "\nSequence prior to shifting: ";
      necklace.printSequence();
      cout << "\n";
      necklace.shiftSequenceToNecklace();
      cout << "\nNecklace: " << endl;
      cout << "\n";
      necklace.printSequence();
      input3 = necklace.binToDec();
      cout << "\n\ndecimal associated with necklace = " << input3 << endl;

      necklace.runNecklaceAlgorithm(input1, input2, input3, true);
```

140

```cpp
        cout << "\nFirst necklace in deBruijn section: " << endl;
        necklace.displayFirstNecklace();
        necklace.createInputString(input3);
        cout << "\nBinary string associated with decimal input: ";
        necklace.printSequence();
        cout << "\n";
        necklace.createDeBruijnSection();
        pos = necklace.searchDeBruijnString();
        cout << "\n\nPositon associated with binary string input = " << pos << endl;
        cout <<"\nDo you want to search for another sequence? " << endl;
        cin >> answer;
      }
    cout << "\n\nDo you want to see another deBruijn section surrounding your necklace
input? ";
    cin >> answer;
    }

    cout << "\n\nGoodbye.." << endl;
    IOThesis::pauseBeforeExit(EXIT_NORMALLY);

    return 0;
} // end Thesismain.cpp
```

3.   IO Thesis CPP File

```cpp
//*********************************************************************
// IOThesis.cpp
//
// LT John Ortiz
//
// Thesis: Generating Necklaces
// Operating Environment: Windows XP Home
// Compiler: Visual Studio .NET
// Date: 08 December 2005
// Description: This program verifies that user inputs are valid
//*********************************************************************

#include <iostream>
#include <cstdlib>
#include <iomanip>
#include <cctype>
#include <string>
#include <sstream>
#include "IOThesis.h"
#include "FileOpeningException.h"

using namespace std;
```

```cpp
using std::exit;

//*****************************************
// Method        displayMessage
// Return value  none
// Parameters    none
// Purpose       prompts user for dimension input
//*****************************************

void IOThesis::displayMessage(string message)
{
  cout << message << endl;

} //end displayMessage



//*****************************************
// Method openOutputFile()
// Return value
// Parameters: const char *FILE_NAME the name of the file to open for writing
// Purpose: dynamically create a new ofstream and returns a pointer if successful
//*****************************************

ofstream* IOThesis::openOutputFile(const char *FILE_NAME)
{
  ofstream *outputFileptr = NULL;
  try
  {
    outputFileptr = new ofstream(FILE_NAME,ios::out);
    if (!(*outputFileptr)) // unable to open object outputFile of type ofstream
    {
      throw FileOpeningException();
    }
    else
    {
      return outputFileptr; //returns pointer to ofstream if file was opened successfully
    }
  }
  catch(FileOpeningException &)
  {
    throw;
  }
}// end openOutputFile
```

142

```
//****************************************
// Method openInputFile
// Return value ifstream*
// Parameters const char *FILE_NAME, the file to open for reading
// Purpose
//****************************************

ifstream* IOThesis::openInputFile(const char *FILE_NAME)
{
  ifstream *inputFileptr = NULL;
  try
  {
    inputFileptr = new ifstream (FILE_NAME,ios::in);
    if (!(*inputFileptr)) // unable to open object outputFile of type ofstream
    {
      throw FileOpeningException();
    }
    else
    {
      return inputFileptr; //returns pointer to ofstream if file was opened successfully
    }
  }
  catch(FileOpeningException &)
  {
    throw;
  }
} // end openInputFile


//****************************************
// Method        inputInt
// Return value   int
// Parameters     none, function will ask for input via I/O
// Purpose        take string input and determine if valid for an int (integer can be any value
less than MAXINT)
//              use inputFilter = IOThesis::inputInt(); vice cin >> inputFilter
//****************************************

int IOThesis::inputInt()
{
  const int SIZE = 12;
  char ch[SIZE]; // temporary buffer for input characters
  char ch2[SIZE]; // buffer for sstream
  string token;
  char *tokenPtr;
```

```cpp
    int size = 0;
    int integer = 0;
    int count = 0;
    bool state = false;
    stringstream ss;

    while(state == false) // This loop prompts a user to enter in only one integer that is less
than 12 characters long
        {
         int q = 0;
            int total=0;
            char element = cin.get();
            while(element != '\n')// reads elements into a character array
            {
            ch[q]= element;
            total++;
            if (total == SIZE) // checks if total exceeds 12 characters
                {
                    cout << "You have reached the maximum number of characters allowed" <<
endl;
                    cout << "Please enter another integer.";
                    state = false;
                    cin.clear();
                    break;
                }

            if (cin.eof()) // checks if ctrl-z in input as a character
                {
                    cout << "\nInvalid character.";
                    cin.clear(); //resets input stream so it can keep processing
                cin.putback('\n');
                    state = !IOThesis::clearInputBuffer();
                    break;
                }

            if (q == 0 && (ch[q] == '+'|| ch[q] == '-' )) // checks if first character is plus or
minus
                {
                    element = cin.get(); // gets the next element
                    q++;
                    continue;
                }

                else if(!isdigit(ch[q])) //checks if I have something other than a digit
                {
```

144

```cpp
                cout << ch[q] << " is not an integer."<< endl;
                state = IOThesis::clearInputBuffer();
             state = false;
             cin.clear();
                break;
              }
          else
          {
           state = true;
           element = cin.get(); // gets the next element
           q++;
          }
        } // end while

     tokenPtr = strtok (ch," "); //begin tokenization of string
       int numtok = 0; // need to initialize through every pass of outer while loop
       while (tokenPtr != NULL) // loop exits if there is one or more token detected
          {
             tokenPtr = strtok (NULL, " "); //get next token
             numtok ++;
          } //     end while

       if (numtok != 1) // tests if there is more than or no inputs in the string
          {
             cout << endl;
             cout << "You have too many inputs." << endl;
             cin.putback('\n');
             state = IOThesis::clearInputBuffer();
             state = false;
             cin.putback('\n');
          } // end if
          else
          {
             state = true;
             count = q;
          }
      } // ends outer while loop

  if (state == true)
   {
     for (int m=0;m < count; m++)
        {
           ch2[m] = ch[m]; // only assigns current "good" input to sstream
        }
        ss << ch2; //takes input string and sends it to sstream
```

```cpp
      ss >> integer;
    } // end if
  return integer;

} // end inputInt



//****************************************
// Method        inputIntWithLimits
// Return value   int
// Parameters    (string message, int lower, int upper)
// Purpose        calls input int for a valid int, then checks limits
//              used similiarly to inputInt
//****************************************

int IOThesis::inputIntWithLimits(string message, int lower, int upper)
{
  cout << message << endl;
  int condition = IOThesis::inputInt();
  while ( condition < lower || condition > upper)
   {
    cout << message << endl;
    condition = IOThesis::inputInt();
   }// end while

   return condition;

}// end inputIntWithLimits



//****************************************
// Method        clearInputBuffer
// Return value   bool TRUE if cin buffer has no additional stuff in it
// Parameters    none
// Purpose        utility function for processing inputs
//****************************************

bool IOThesis::clearInputBuffer()
{
  bool boolstate = false;
  char somechar = cin.get();
  while(somechar!= '\n') //loop enables the input buffer to be flushed
  {
   if (isspace(somechar))
    {
```

```cpp
            boolstate = true;
        }
        else
        {
            boolstate = false;
        }
        somechar = cin.get(); // keeps writing over somechar's memory space with current
character
    } // end while

    return boolstate;

} // end clearInputBuffer


//****************************************
// Method        pauseBeforeExit
// Return value   none
// Parameters     int condition (EXIT_WITH_ERROR = 1, global defined in IOThesis.h)
//                      (EXIT_NORMALLY = 0, global defined in IOThesis.h)
// Purpose       keeps display window open long enough to see results
//****************************************

void IOThesis::pauseBeforeExit(const int condition)
{
    cout << "\n\n" << endl;
    char ch;
    ch = cin.get(); //need to flush the input buffer or it may still hold the last
                // carriage return and blow us out of the program early
    cout << "\n\tPress <enter> or <return> to exit...\n";
    ch = cin.get();
    exit(condition);

} // end pauseBeforeExit

// end of file IOThesis.cpp
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[Fr] H. Fredricksen, I. Kessler, "An algorithm for generating necklaces of beads in two colors", *Discrete Mathematics*, vol. 61, pp 181-188, 1986.

[Go] S. Golomb, *Shift Register Sequences*. San Francisco: Holden-Day, 1967.

[Ma] F. MacWilliams, N. Sloane, "Pseudo-random sequences and arrays", *Proceedings of the IEEE*, vol. 64, pp.1715-1729, 1976.

[Mar] M. Martin, "A problem in arrangement", *Bull. Amer. Math. Soc.* vol. 40, pp. 859-864, 1934.

[Mat] D. Matty, "Analysis of the necklace algorithm and its applications", *Naval Postgraduate School Thesis*, 1999.

[Mi] C. Mitchell, T. Etzion, K. Paterson, "A method for constructing decodable DeBruijn sequences", *IEEE Trans. Inform. Theory*, vol. 42, pp. 1472-1478, 1996.

[Pe1] E. Petriu, J. Basran, F. Groen, "Automated guided vehicle position recovery", *IEEE Trans. Instrum. and Meas.*, vol. 39, pp.254-258, 1990.

[Pe2] E. Petriu, "New pseudo-random / natural code conversion method", *Electron. Lett.*, vol. 24, pp.1358-1359, 1988.

[Pe3] E. Petriu, J. Basran, F. Groen, "Absolute position measurement using pseudo-random binary encoding", *IEEE Instrum. and Meas. Magazine*, Sept., pp.19-23, 1998.

[Sl] N. Sloane, On-Line Encyclopedia of Integer Sequences, http://www.research.att.com/~njas/sequences/Seis.html, accessed 20 January 2006.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1.   Defense Technical Information Center
        Ft. Belvoir, VA

2.   Dudley Knox Library
        Naval Postgraduate School
        Monterey, CA

3.   Harold M. Fredricksen
        Department of Applied Mathematics
        Naval Postgraduate School Code Fs / Ma
        Monterey, CA

4.   Jon T. Butler
        Department of Electrical Engineering
        Naval Postgraduate School Code EC / Bu
        Monterey, CA