# Symmetry of Mutually Orthogonal Latin Squares

Anthony Morast

May 8, 2015

# Contents

## 1 Latin Squares

To begin, a **Latin square** of order $n$ is defined as *a square array which contains n different elements with each element occuring n times but with no element occurring twice in the same column or row* [7]. For example, the following is a Latin square of order four

$$\begin{array}{cccc} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \\ 3 & 4 & 1 & 2 \\ 2 & 1 & 4 & 3 \end{array}.$$

The earliest occurrences of Latin squares is in their use on amulets and in rites in Arab and Indian communities from about the year 1000. These amulets were believed to be worn to fight evil spirits, show reverence for gods, and celebrate the sun and planets. Later, in the 13th century, Spanish mystic and philosopher Ramon Lull used Latin squares in attempts to explain the world by combinatorial means. In the 18th century a card problem, which consisted of playing cards being arranged such that each row, column, and main diagonal contain an ace, a king, a queen, and a jack, all of different suits, could be solved with the use of Latin squares. The problem would form a Latin square of both value and suit.

Later in the 18th century, about 1779, Leonhard Euler began studying Latin squares. He began studying the squares when presented with the 36 Officer Problem, which later became know as the Euler Officer Problem. This problem consisted of 36 officers of 6 different ranks being selected from 6 regiments. Euler was charged with finding an arrangment for the officers so that no row or column contained an officer of the same rank or from the same regiment. Euler gave Latin squares their name, he was the first to define them using mathematical terminology, and he was also the first to investigate their properties. Euler made many conjectures about these properties, a few of which ended up being false [4].

Today Latin squares are used for experiments more similar to Euler's than to the ancient Arab and Indian cultures. The Latin square design is used when researchers desire to control variation in experiments that are related to rows and columns. For example, if a researcher were to study the effects of four diets, each to be given to four different cows in succession, he could use a Latin square to arrange his experiments. To ensure the effects are based solely on the four diets, each cow must have a different meal at any given time. Thus, an order-4 Latin square could be produced so that the researcher can keep track of which cow is to have which diet at any given time until each cow has had each diet [8].

## 2 Types of Latin Squares

This section covers a few useful structures that Latin squares can take. The types of Latin squares deal with particular orderings of the square's rows, columns, and/or symbols. Much of the following material is dependent on these structures and the properties they have. The names for these structures are not yet standardized within the combinatorics community.

Thus one mathematician may call a particular square normalized and another might call it reduced. We choose to define these terms as follows.

The first type is referred to as normalized. We define a **normalized Latin square** as a *Latin square with the first row and column given by 1,2,...,n.* For example

$$\begin{array}{cccc} 1 & 2 & 3 & 4 \\ 2 & 1 & 4 & 3 \\ 3 & 4 & 1 & 2 \\ 4 & 3 & 2 & 1 \end{array}$$

is a normalized Latin square of order 4. Normalized Latin squares are useful since they represent a subset of Latin squares such that, after applying a few operations, every other Latin square of a particular order can be generated.

The second Latin square type is reduced. We define a **reduced Latin square** as *a Latin square in which the first row is given by 1,2,...,n.* That is, the first row is in order but the first column need not take the form 1,2,...,n. For example,

$$\begin{array}{cccc} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \\ 3 & 4 & 1 & 2 \\ 2 & 1 & 4 & 3 \end{array}$$

represents a reduced Latin square of order 4, derived from the square above by swapping rows two and four. This type of square was used to determine if sets of Latin squares are mutually orthogonal, which is discussed later.

## 3 Isotopy Classes

Latin squares have many properties, but this paper will focus on two in particular. The first property studied is isotopy classes. Two Latin squares belong to the same **isotopy class** if *one square can be obtained by permuting the rows, columns, and symbols of the other square.* For example

$$\begin{array}{cccc} 2 & 1 & 4 & 3 \\ 3 & 4 & 1 & 2 \\ 4 & 3 & 2 & 1 \\ 1 & 2 & 3 & 4 \end{array} \quad \text{and} \quad \begin{array}{cccc} 4 & 3 & 2 & 1 \\ 1 & 2 & 3 & 4 \\ 3 & 4 & 1 & 2 \\ 2 & 1 & 4 & 3 \end{array}$$

are two order-4 Latin squares that belong to the same isotopy class. In this example, the second square can be obtained from the first by swapping columns one and two,

$$\begin{array}{cccc} 2 & 1 & 4 & 3 \\ 3 & 4 & 1 & 2 \\ 4 & 3 & 2 & 1 \\ 1 & 2 & 3 & 4 \end{array} \quad \rightarrow \quad \begin{array}{cccc} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \\ 3 & 4 & 1 & 2 \\ 2 & 1 & 4 & 3 \end{array}$$

and then by swapping the rows one and two,

$$\begin{array}{cccc} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \\ 3 & 4 & 1 & 2 \\ 2 & 1 & 4 & 3 \end{array} \quad \rightarrow \quad \begin{array}{cccc} 4 & 3 & 2 & 1 \\ 1 & 2 & 3 & 4 \\ 3 & 4 & 1 & 2 \\ 2 & 1 & 4 & 3 \end{array}.$$

As you can see, applying these two operations to the first square created the second; therefore they are **isotopic**, i.e. *they belong to the same isotopy class.*

Isotopy classes were used when it was required to generate other Latin squares. This can be done by taking a single Latin square from an isotopy class, referred to as an **isotopy class representative**, and permuting the rows, columns, and symbols of that Latin square. By permuting rows, columns, and symbols of isotopy class representatives from each of a particular order's isotopy classes one can produce every other Latin square of that order. The Latin squares,

$$\begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 5 & 1 & 4 \\ 3 & 5 & 4 & 2 & 1 \\ 4 & 1 & 2 & 5 & 3 \\ 5 & 4 & 1 & 3 & 2 \end{array} \quad \text{and} \quad \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 1 & 5 & 3 \\ 3 & 5 & 4 & 2 & 1 \\ 4 & 1 & 5 & 3 & 2 \\ 5 & 3 & 2 & 1 & 4 \end{array}$$

are two isotopy class representatives, one from each of the two isotopy classes of order-five Latin squares. To create a new square, first swap rows one and two of the first square,

$$\begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 5 & 1 & 4 \\ 3 & 5 & 4 & 2 & 1 \\ 4 & 1 & 2 & 5 & 3 \\ 5 & 4 & 1 & 3 & 2 \end{array} \quad \rightarrow \quad \begin{array}{ccccc} 2 & 3 & 5 & 1 & 4 \\ 1 & 2 & 3 & 4 & 5 \\ 3 & 5 & 4 & 2 & 1 \\ 4 & 1 & 2 & 5 & 3 \\ 5 & 4 & 1 & 3 & 2 \end{array}.$$

This operation will create a new square. Furthermore, more new squares can be created by swapping rows, columns, and/or symbols of either the original square or the newly-generated square, such as

$$\begin{array}{ccccc} 2 & 3 & 5 & 1 & 4 \\ 1 & 2 & 3 & 4 & 5 \\ 3 & 5 & 4 & 2 & 1 \\ 4 & 1 & 2 & 5 & 3 \\ 5 & 4 & 1 & 3 & 2 \end{array} \quad \rightarrow \quad \begin{array}{ccccc} 2 & 3 & 5 & 1 & 4 \\ 5 & 4 & 1 & 3 & 2 \\ 3 & 5 & 4 & 2 & 1 \\ 4 & 1 & 2 & 5 & 3 \\ 1 & 2 & 3 & 4 & 5 \end{array}.$$

Here the square generated above had its second and fifth rows swapped to create another new square. Also consider,

$$\begin{matrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 5 & 1 & 4 \\ 3 & 5 & 4 & 2 & 1 \\ 4 & 1 & 2 & 5 & 3 \\ 5 & 4 & 1 & 3 & 2 \end{matrix} \quad \rightarrow \quad \begin{matrix} 1 & 3 & 2 & 4 & 5 \\ 2 & 5 & 3 & 1 & 4 \\ 3 & 4 & 5 & 2 & 1 \\ 4 & 2 & 1 & 5 & 3 \\ 5 & 1 & 4 & 3 & 2 \end{matrix}.$$

In this set of squares the original isotopy class representative, from above, had its second and third columns swapped to create a fourth new square.

Permuting rows and columns consists of these swapping operations. The last operation required to generate all Latin squares is permuting symbols. This consists of swapping symbols within the square, e.g. permuting symbols 3 and 1 would mean changing all the 3's in the square to 1's and vice versa. For example,

$$\begin{matrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 5 & 1 & 4 \\ 3 & 5 & 4 & 2 & 1 \\ 4 & 1 & 2 & 5 & 3 \\ 5 & 4 & 1 & 3 & 2 \end{matrix} \quad \rightarrow \quad \begin{matrix} 3 & 2 & 1 & 4 & 5 \\ 2 & 1 & 5 & 3 & 4 \\ 1 & 5 & 4 & 2 & 3 \\ 4 & 3 & 2 & 5 & 1 \\ 5 & 4 & 3 & 1 & 2 \end{matrix}.$$

The second square is the first square, where the symbols 1 and 3 have been switched. This operation, like row and column permutations, can be applied recursively to produce all Latin squares of a particular order.

To generate all Latin squares of order $n$, begin by permuting the rows, columns, and symbols of the isotopy class representatives. Then the operations need to be applied to each generated square to ensure we get every possible square. The process of permuting rows, columns, and symbols is repeated until no new squares are created.

For much of this research only reduced Latin squares were required. Therefore, to find all reduced squares, the isoptopy class representatives were permuted to find all normalized Latin squares. Then each normalized Latin square had its last $n - 1$ rows permuted to produce every reduced Latin square of order $n$.

Latin squares also belong to another equivalence class called a main class. A **main class** of a Latin square is *the union of the isotopy classes of its conjugates.* A Latin square **conjugate** is *any permuation of its rows, columns, or symbols.* Two Latin squares $L$ and *L'* are said to be **main class equivalent** if *they belong to the same main class, i.e. if L is isotopic to a conjugate of $L'$.*

To generate isotopy classes row, column, and symbol permutations are used. In order to generate a complete main class another operation must be added, which is a permutation on all three. Letting the triple $(r, c, s)$ represent the row, column, and symbol of a particular entry in a Latin square, we can permute this triple to create a new square. For example, in

$$\begin{array}{ccccc}1&2&3&4&5\\2&3&5&1&4\\3&5&4&2&1\\4&1&2&5&3\\5&4&1&3&2\end{array},$$

the entry represented by (2,4,1) would be the entry at row two, column four, which contains symbol one.

We can create a $(r, c, s)$ triple for each of the entries in the Latin square. We can then permute by swapping all three values in the triple, e.g $(r, c, s) \rightarrow (s, r, c)$. In this operation the entry symbol now becomes its row, its row becomes its column, and its column becomes its symbol. Doing this for each entry in the Latin square will produce another square. The square

$$\begin{array}{ccccc}1&2&3&4&5\\2&3&5&1&4\\3&5&4&2&1\\4&1&2&5&3\\5&4&1&3&2\end{array}$$

will produce the triples

{(1,1,1), (1,2,2), (1,3,3), (1,4,4), (1,5,5), (2,1,2), (2,2,3), (2,3,5), (2,4,1), (2,5,4), (3,1,3), (3,2,5), (3,3,4), (3,4,2), (3,5,1), (4,1,4), (4,2,1), (4,3,2), (4,4,5), (4,5,3), (5,1,5), (5,2,4), (5,3,1), (5,4,3), (5,5,2)}.

Applying the permutation $(r, c, s) \rightarrow (s, c, r)$ will produce the triples

{(1,1,1), (2,2,1), (3,3,1), (4,4,1), (5,5,1), (2,1,2), (3,2,2), (5,3,2), (1,4,2), (4,5,2), (3,1,3), (5,2,3), (4,3,3), (2,4,3), (1,5,3), (4,1,4), (1,2,4), (2,3,4), (5,4,4), (3,5,4), (5,1,5), (4,2,5), (1,3,5), (3,4,5), (2,5,5)}

and the square

$$\begin{array}{ccccc}1&4&5&2&3\\2&1&4&3&5\\3&2&1&5&4\\4&5&3&1&2\\5&3&2&4&1\end{array},$$

which is a new square of order 5 that will belong to the same main class as the original square of order 5.

Table 1 shows the number of main classes and isotopy classes of the Latin squares of order 1 through 11 [5].

| $n$ | main classes | isotopy classes |
|---|---:|---:|
| 1 | 1 | 1 |
| 2 | 1 | 1 |
| 3 | 1 | 1 |
| 4 | 2 | 2 |
| 5 | 2 | 2 |
| 6 | 12 | 22 |
| 7 | 147 | 564 |
| 8 | 283657 | 1676267 |
| 9 | 19270853541 | 115618721533 |
| 10 | 34817397894749939 | 208904371354363006 |
| 11 | 208904371354363006 | 12216177315369229261482540 |

Table 1: Number of Main and Isotopy Classes

## 4 Mutually Orthogonal Latin Squares

The second property studied was orthogonality, more specifically mutual orthogonality. **Orthogonal Latin squares** of order $n$ over two sets $S$ and $T$, each consisting of $n$ symbols, *is an $n \times n$ arrangement of cells, each cell containing an ordered pair (s,t), where s is in S and t is in T, such that every row and every column contains each element of S and each element of T exactly once, and that no two cells contain the same ordered pair.* For example, the following squares are orthogonal,

$$\begin{array}{cccc} 1 & 2 & 3 & 4 \\ 2 & 1 & 4 & 3 \\ 3 & 4 & 1 & 2 \\ 4 & 3 & 2 & 1 \end{array} \qquad \begin{array}{cccc} 1 & 2 & 3 & 4 \\ 3 & 4 & 1 & 2 \\ 4 & 3 & 2 & 1 \\ 2 & 1 & 4 & 3 \end{array} \quad \rightarrow \quad \begin{array}{cccc} (1,1) & (2,2) & (3,3) & (4,4) \\ (2,3) & (1,4) & (4,1) & (3,2) \\ (3,4) & (4,3) & (1,2) & (2,1) \\ (4,2) & (3,1) & (2,4) & (1,3) \end{array}.$$

These squares are orthogonal because each of the sixteen ordered pairs occur exactly once in the array that was produced with the ordered pairs of the two squares left of the arrow.

The following Latin squares are not orthogonal,

$$\begin{array}{cccc} 1 & 2 & 3 & 4 \\ 2 & 1 & 4 & 3 \\ 3 & 4 & 1 & 2 \\ 4 & 3 & 2 & 1 \end{array} \qquad \begin{array}{cccc} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \\ 3 & 4 & 1 & 2 \\ 4 & 1 & 2 & 3 \end{array} \quad \rightarrow \quad \begin{array}{cccc} (1,1) & (2,2) & (3,3) & (4,4) \\ (2,2) & (1,3) & (4,4) & (3,1) \\ (3,3) & (4,4) & (1,1) & (2,2) \\ (4,4) & (3,1) & (2,2) & (1,3) \end{array},$$

because the order pairs (1,1), (2,2), (3,3), (4,4), (1,3), and (3,1) are all repeated. For example, the ordered pair (2,2) occurs in column one, row two; column two, row one; and column three, row four.

A set of **mutually orthogonal Latin squares**, or **MOLS**, is *a set of two or more Latin squares of the same order, all of which are orthogonal to one another* [3]. Note that

$$\begin{matrix}1&2&3&4\\2&1&4&3\\3&4&1&2\\4&3&2&1\end{matrix}, \quad \begin{matrix}1&2&3&4\\3&4&1&2\\4&3&2&1\\2&1&4&3\end{matrix}, \quad \text{and} \quad \begin{matrix}1&2&3&4\\4&3&2&1\\2&1&4&3\\3&4&1&2\end{matrix}$$

are mutually orthogonal since,

$$\begin{matrix}1&2&3&4\\2&1&4&3\\3&4&1&2\\4&3&2&1\end{matrix} \quad \begin{matrix}1&2&3&4\\3&4&1&2\\4&3&2&1\\2&1&4&3\end{matrix} \quad \rightarrow \quad \begin{matrix}(1,1)&(2,2)&(3,3)&(4,4)\\(2,3)&(1,4)&(4,1)&(3,2)\\(3,4)&(4,3)&(1,2)&(2,1)\\(4,2)&(3,1)&(2,4)&(1,3)\end{matrix},$$

$$\begin{matrix}1&2&3&4\\2&1&4&3\\3&4&1&2\\4&3&2&1\end{matrix} \quad \begin{matrix}1&2&3&4\\4&3&2&1\\2&1&4&3\\3&4&1&2\end{matrix} \quad \rightarrow \quad \begin{matrix}(1,1)&(2,2)&(3,3)&(4,4)\\(2,4)&(1,3)&(4,2)&(3,1)\\(3,2)&(4,1)&(1,4)&(2,3)\\(4,3)&(3,4)&(2,1)&(1,2)\end{matrix},$$

$$\begin{matrix}1&2&3&4\\4&3&2&1\\2&1&4&3\\3&4&1&2\end{matrix} \quad \begin{matrix}1&2&3&4\\3&4&1&2\\4&3&2&1\\2&1&4&3\end{matrix} \quad \rightarrow \quad \begin{matrix}(1,1)&(2,2)&(3,3)&(4,4)\\(4,3)&(3,4)&(2,1)&(1,2)\\(2,4)&(1,3)&(4,2)&(3,1)\\(3,2)&(4,1)&(1,4)&(2,3)\end{matrix}.$$

It can be seen in the example above that the first square is orthogonal with the second and third squares and the second and third square are orthogonal with each other. Thus the three sqaures are MOLS.

Mutually orthogonal Latin squares have many uses, such as experiment design. They are useful in experiment design for a slightly different reason than Latin squares. **Graeco-Latin sqaures** are *pairs of orthogonal Latin squares, where Graeco-Latin squares use Greek and Latin letters to replace the numbers used in our examples, though fundamentally they are the same thing.* For example, the Latin squares to the left of the arrow above would be Graeco-Latin squares if the numbers were changed to Greek and Latin symbols. The Graeco-Latin squares are used in experiment design. For example, if an experiment with treatments is conducted at two different times, the two squares that can be used to design the experiment will be orthogonal if there are the same number of treatments on the same subjects. Thus, the Graeco-Latin square could be used to display two distinct experiment designs with a single square. Mutually orthogonal Latin squares are required since they ensure each of the experiment's participants get a different treatment in each experiment [2].

MOLS are also used to determine the existence of other combinatorial objects. One such object is the finite projective plane [6]. A theorem devised by R. C. Bose states

**Theorem 1.** *A finite projective plane of order $n$ exists if and only if there exists a complete set of MOLS, i.e. a set of MOLS containing (n-1) Latin squares of order $n$.*

MOLS also play a role in partially balanced complete block designs, i.e. $(v, k, \lambda)$-designs. However, these applications are beyond the scope of this paper. The interested reader is referred to *Applied Combinatorics by Fred Roberts and Barry Tesman* [1].

# 5 Symmetry and Mutually Orthogonal Latin Squares

## 5.1 Algorithms

Initially when studying Latin squares, we tried to determine whether or not the conjecture that the largest set of MOLS of order $n$ is $n-1$. To do so, it seemed every pair of Latin square would need to be checked for orthogonality. However, it was determined that only reduced Latin squares need be compared. This is because orthogonality will hold no matter what permutation of the square is done. Consider two Latin squares that are not mutually orthogonal. Thus there exist distinct entries $(a_{ij}, b_{ij})$ and $(a_{mn}, b_{mn})$ such that $(a_{ij}, b_{ij}) = (a_{mn}, b_{mn})$. Let $\sigma$ and $\tau$ be row and column permutations, respectively. Applying these permutations to the entries produces, $(a_{\sigma(i)\tau(j)}, b_{\sigma(i)\tau(j)}) = (a_{\sigma(m)\tau(n)}, b_{\sigma(m)\tau(n)})$. Since $\sigma$ and $\tau$ are bijections, then it is not possible that $\sigma(i) = \sigma(m)$ or $\tau(j) = \tau(n)$ for $i \neq m$ and $j \neq m$. Thus the Latin square is still not orthogonal; the duplicate pairs have essentially just moved to a different location in the Latin square.

The general algorithm used to find the sets of MOLS for order $n$ is shown in Algorithm 1. This algorithm would determine not only the largest set of mutually orthogonal Latin

**Algorithm 1** Finding Sets of Mutually Orthogonal Latin Squares

```
 1: Generate all Latin squares of order $n$
 2: Find all the reduced squares
 3: Pair the entries of every reduced Latin square with the corresponding entries from every
    other Latin square
 4: Determine if there are repeating pairs
 5: if No Repeating Pairs then
 6:     Output squares that were orthogonal to a file
 7: else
 8:     Continue
 9: Recursively read in the file containing sets of Latin squares that were output and compare
    with other squares again
10: Repeat recursion until there are no new sets of MOLS
```

squares but also which Latin squares were part of any of the MOLS.

This data sparked our interest in the relationship between symmetry and orthogonality, and we began to analyze the data to determine how connected the individual isotopy classes were with each other based on orthogonality. That is, we wanted to determine the isotopy classes of each Latin square that made up the MOLS. The algorithm used for determining this is fairly straightforward and is outlined in Algorithm 2.

Once we determined from which isotopy classes the squares originated we once again analyzed the data. During this analysis we noticed two peculiar properties of these squares: isotopy classes and MOLS. First, we noticed all symmetric Latin squares of order $n$ belong to the same isotopy class. Here a **symmetric Latin square** is *a Latin square that is equal to its transpose.* Simply put, a symmetric Latin square's rows will mirror its columns, so

**Algorithm 2** Determining the isotopy classes of mutually orthogonal Latin squares

```
1: Read in square from the output file generated when finding the largest set of MOLS
2: for Each square read from the generated file do
3:     for Each isotopy class of the order under investigation do
4:         for Each Latin square in the isotopy class do
5:             if The Latin squares are equal then
6:                 Record in which isotopy class the square was found
7:                 Goto 2
8: Observe which isotopy classes contained squares that made up the MOLS
```

row one is in the same order as column one, row two is in the same order as column two, row three is in the same order as column three, etc. Second, we noticed that all of the MOLS were comprised of squares that came from an isotopy class containing the symmetric Latin squares. That is, we found a potential link between a Latin square being orthogonal to other Latin squares and the symmetry of the Latin square.

We've verified this for Latin squares of order 3, 4, and 5, as orders 1 and 2 are trivial. When attempting to produce this data for order 6 and above we encountered a "computational barrier", i.e. our current algorithms could not produce the results in a timely manner for the large number of Latin squares that make up order 6 and above. In an attempt to overcome this barrier the algorithms were re-written using C# and .NET's multithreading libraries following Algorithm 3. In this algorithm, the threads chosen to compare squares further along in the list of generated squares were given more work as each square is only compared with all squares after its index in the list. Doing so allowed us to find that order 6 had absolutely no orthogonal Latin squares. However, Latin squares of orders $4n + 2$ for $n = 1, 2, 3, ...$ are known as Euler spoilers and typically do not have the same properties as the other orders of Latin squares. Therefore it was determined that orders beyond 6 need to be studied.

Although using C# and .NET multithreading libraries allowed us to produce results for order 6 it wasn't powerful enough to produce these results for order 7 in a timely manner. Therefore, it was decided to once again rewrite the algorithms in yet another language. This time we chose to use C++, since we could use both MPI (Message Passing Interface) to easily distribute the computations to many different machines on a network and OMP (Open Multi-Processing) to make many other pieces of the algorithm parallel that are currently computed serially. The algorithm used in the C++ implementation is outlined in Algorithm 4.

The C++ algorithms must be developed and optimized since the number of Latin squares of order $n$ used in our research increase the number of normalized Latin squares times $(n-1)!$ is equivalent to . For example, this list contains the number of Latin squares and the number of reduced Latin squares for orders 3 through 8,

- Order 3: 12 total squares, 2 reduced squares
- Order 4: 576 total squares, 24 reduced squares

**Algorithm 3** C# Multithreaded Algorithm

```
1: for Each isotopy class representative do                         ▷ Generate all squares
2:     Permute rows, columns, and symbols recursively until no new squares are generated
3: for Each Latin square (the ones just generated) do                ▷ Find reduced
4:     if The square is in reduced form then
5:         Add to list of squares to be compared
6: Divide the work into $i$ parts, where $i$ is number of cores the computer has
7: When dividing the work the threads allotted work last are given more squares, since they
   will end up doing fewer comparisons
8: for Each square in allotted work do                               ▷ This is for each thread
9:     Compare each square with every other square having an index higher than the current
   square's index in the reduced square list.
10:    That is, square 1 is compared with 2, 3, 4, ... (all the squares), but square 2 is only
   compared with 3, 4, 5, 6, 7, ..., and square 3 with squares, 4, 5, 6, 7, 8,..., etc. This is
   because the operation is commutative
11:    if There is an orthogonal set then
12:        Write the squares to a file and whether or not they are symmetric
```

**Algorithm 4** C++ Multithreaded Algorithm

```
1: for Each isotopy class representative do                         ▷ parallelize via OMP
2:     Permute rows, columns, and symbols to generate all Latin squares
3: for Each generate Latin square do                                 ▷ serial
4:     if Square is in reduced form then
5:         Add to list of squares to be compared
6: Divide the work up fairly to run on all nodes/cores
7: for Each Latin square list of squares to be compared do           ▷ MPI parallel
8:     for Each square whose index is greater than the current square in the check list do
9:         if The squares are orthogonal then
10:            Write the squares and whether or not they are symmetric to a file
```

- Order 5: 161,280 total squares, 1,344 reduced squares
- Order 6: 812,851,200 total squares, 1,128,960 reduced squares
- Order 7: 61,479,419,904,000 total squares, 12,198,297,600 reduced squares
- Order 8: 10,877,6032,459,082,956,800 total squares, 2,697,818,265,354,240 reduced squares

As this table shows, the number of Latin squares in each order grows very quickly. To be able to obtain and analyse this data Algorithm 4 must be optimized.

## 5.2 Results

### 5.2.1 Conjecture 1 - Largest Set of MOLS

We found that this conjecture held for Latin squares of orders 3, 4, 5 but not for order 6. We did not expect this conjecture to be true for order 6 as Latin squares whose orders satisfy the equation $n = 4k+2$ are known as Euler spoilers, for which many properties of other orders of Latin squares do not hold. We stopped researching this at order 6 since our results produced conjectures that were more interesting to study. However, in combinatorics this conjecture has been proved and the sets are commonly referred to as orthogonal families. That is, an **orthogonal family** of Latin squares is *a set of mutually orthogonal Latin squares of order $n$ such that the size of the set is equal to $n-1$* [1].

### 5.2.2 Conjecture 2 - Orthogonal Latin Squares are Generated From Symmetric Normalized Squares

For order 4 the isotopy class representatives used were,

$$\begin{array}{cccc} 1 & 2 & 3 & 4 \\ 2 & 1 & 4 & 3 \\ 3 & 4 & 1 & 2 \\ 4 & 3 & 2 & 1 \end{array} \qquad \begin{array}{cccc} 1 & 2 & 3 & 4 \\ 2 & 4 & 1 & 3 \\ 3 & 1 & 4 & 2 \\ 4 & 3 & 2 & 1 \end{array}$$

to generate two isotopy classes containing two normalized Latin squares each. Order 4 is a trivial case because all of the normalized Latin squares are symmetric. Therefore all squares are generated from symmetric squares; thus all MOLS are comprised of squares that are generated from symmetric squares. However, when applying Algorithms 1 and 2 above we found that all mutually orthogonal Latin squares were generated from a single isotopy class rather than a combination of the two isotopy classes.

Using these isotopy class representatives of order 5,

$$\begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 5 & 1 & 4 \\ 3 & 5 & 4 & 2 & 1 \\ 4 & 1 & 2 & 5 & 3 \\ 5 & 4 & 1 & 3 & 2 \end{array} \qquad \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 1 & 5 & 3 \\ 3 & 5 & 4 & 2 & 1 \\ 4 & 1 & 5 & 3 & 2 \\ 5 & 3 & 2 & 1 & 4 \end{array},$$

we generated two isotopy classes, one comprised of symmetric normalized Latin squares and one of asymmetric squares, containing 10 and 46 squares, respectively. After executing Algorithm 1 and Algorithm 2 we discovered that all of the mutually orthogonal squares were derived from a Latin square that was symmetric. To be clear, that is not to say all orthogonal squares are symmetric but rather the squares that are orthogonal are generated from a normalized square that is symmetric.

As order 6 has 22 isotopy classes, the 22 isotopy class representatives will not be listed here. When applying the same algorithms as we did with order 5 we found that order 6 had no orthogonal squares. However, order-6 Latin squares are the first order of squares of what are known as Euler spoilers as they don't follow any of the 'rules' Leonhard Euler had devised for Latin squares. Since there were no mutually orthogonal Latin squares of order 6 our conjecture is vacuously true for order 6. Therefore, we find it necessary to proceed to order-7 Latin squares and above to determine if our conjecture that symmetry and orthogonality are connected.

## 5.3 Future Work

Since our second conjecture is left unanswered at this point we plan on continuing this research by studying higher orders of Latin squares. To do so, the conversion of the algorithms used for orders 4, 5, and 6 to C++ must be completed so that they can more easily be ran on cluster computers. After the algorithms are completed we will try to get some time on a clustered super computer so that the results can be produced in a timely manner. Furthermore, more algorithms will need be devised to analyze the results since the results produced by higher orders will much larger than orders 4, 5, and 6.

## 6 Appendix

The following code can be found at http://www.github.com/anthonyam12/MathLibrary, my git repository.

### 6.1 MATLAB Code

```
function BothOps (B, rep_number)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Author: Anthony Morast
% Advisor: Brent Deschamp
% Undergraduate Research - SDSMT 2015
%
% Permutes the symbols of a Latin square from an isotopy class
% to generate all normalized Latin squares.

% UPDATE AS OF 9-1-14:
% This function takes an isotopy class representative of order
    five,
% applies both operations (R,C,S permutations and Row and Column
    Permutations),
% and prints the result to a file. To make this order 6 a few
    tweaks will need to be made.
% The script eq_ls.m must be run to weed out repeats of Latin
    squares. The documentation in
% eq_ls.m is decent so it shouldn't be too hard to use.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%open file containing permutation of order 4
fid = fopen ( '5_perm.txt' );

%read in number of permutations
n = fscanf(fid, '%u',1);

%open file for writing
filename = strcat('isotopyclass',int2str(rep_number));
filename = strcat(filename,'.dat');

fid2 = fopen(filename, 'w');

m=n;
```

```
i = 0;

while ( i < factorial(n) )
    A = B;
    %read in permutation
    sigma = fscanf ( fid , '%u', n);
    for j=1:m
        for k=1:n
            x = A(j,k);
            if x == 1
                A(j,k) = sigma(1);
            elseif x == 2
                A(j,k) = sigma(2);
            elseif x == 3
                A(j,k) = sigma(3);
            elseif x == 4
                A(j,k) = sigma(4);
            else
                A(j,k) = sigma(5);
            end
        end
    end
    C = normalize_ls_5(A);

    %get squares created from permuting rows and columns of C
    D = getSquares(C);

    [z12,z13,z14] = size(D);


    for i_last=1:z14
        C = D(:,:,i_last);
        %print out Latin square
        fprintf(fid2, '%u %u %u %u %u ', C);
        fprintf(fid2, '\n');
    end

    i = i+1;
end

fclose(fid);
fclose(fid2);
```

```
function  x  = check_latin_square( A, B )
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Author: Anthony Morast
% Advisor: Brent Deschamp
% Undergraduate Research - SDSMT 2015
%
% This function checks if the ordered pairs of two Latin squares
   occurs
% exactly once. If every pair occurs once the function returns a
   1, if any
% ordered pair occurs more than once the function will return a
   zero. This is
% the method used for checking orthogonality.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[n,m] = size(A);

% declare n x n matrix to store a one if the pair is used and 0 if
    it is not
check = zeros (n,n);

flag = 0;

for i=1:n
    for j=1:m
        a = A(i,j);
        b = B(i,j);
        if ( check(a,b) == 0 )
            check ( a, b ) = 1;
        elseif ( check (a,b) == 1 )
            flag = 1;
            break;
        end
        if flag ==1
            break;
        end
    end
end

if flag == 1
    x = 0;
elseif flag == 0
```

```
    x = 1;
end

end



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Author: Anthony Morast
% Advisor: Brent Deschamp
% Undergraduate Research - SDSMT 2015
%
% This method was necessary since we used a lot of data from
   Brendan McKay's
% website (http://cs.anu.edu.au/~bdm/data/latin.html) The data on
   his site
% used 0-based numbering, this method converts the squares to 1-
   based numbering
% which was our standard.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% open input and output files
fid = fopen('mckay_nls5.dat');
fid1 = fopen('nlsO5.dat', 'wt');

% read in the squares, order 5 in this case and increment the
   values by 1
while (~feof(fid))
   A = fscanf(fid, '%u', [5 5]);

   for i=1:5
       for j=1:5
           A(i,j) = A(i,j) + 1;
       end
   end
   fprintf(fid1,'%u %u %u %u %u ', A);
   fprintf(fid1, '\n');
end



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Author: Anthony Morast
% Advisor: Brent Deschamp
% Undergraduate Research - SDSMT 2015
%
% Reads in a bunch of files and adds the counts together.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% base file name
filename = 'isotopyclass';
% keep track of number of squares
total = 0;
% write file
fid2 = fopen('normalized_ls_order6.dat', 'wt');
b = zeros(0);

for i=1:22
   %create file name
   this_file = strcat(filename,int2str(i));
   this_file = strcat(this_file, '-new');
   this_file = strcat(this_file, '.dat');

   fid = fopen(this_file);
   k=1;

   %count
   while (~feof(fid))
       A = fscanf (fid, '%u', [6 6]);
       if ( A == A')
          b(k) = i;
          k = k + 1;
       end
       total = total +1;
       fprintf(fid2, '%u %u %u %u %u %u ', A);
       fprintf(fid2,'\n');
   end

   fclose(fid);
end

disp(total);

i = size(b);
```

```
fclose('all');



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Author: Anthony Morast
% Advisor: Brent Deschamp
% Undergraduate Research - SDSMT 2015
%
% This script reads in multiple files that contain index
   references  to a file
% of reduced Latin squares. These indexes are references in the
   reduced square
% file , the square is compared to the isotopy classes to find
   which one it is
% from, and then the indexes and the isotopy class from which it
   originated are
% written to a file in a chart format.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% open files
fid_6 = fopen('setOf6.dat');
fid_10 = fopen('setOf10.dat');
fid_40 = fopen('setOf40.dat');
fid_out = fopen('chart.dat','wt');

% read in nls
squares6 = zeros(5,5,0);
squares10 = zeros(5,5,0);
squares40 = zeros(5,5,0);

while (~feof(fid_6))
   A = fscanf(fid_6 ,'%u',[5 5]);
   squares6 = cat(3, squares6 , A);
end

while (~feof(fid_10))
   A = fscanf(fid_10 ,'%u',[5 5]);
   squares10 = cat(3, squares10 , A);
end

while (~feof(fid_40))
    A = fscanf(fid_40 ,'%u',[5 5]);
    squares40 = cat(3, squares40, A);
end

% create filename base
filename = 'perm_class';

fprintf(fid_out, '%s      %s', 'Isotopy Class', 'Permute Class');
fprintf(fid_out,'\n');
for i=1:16
   currentfile = strcat(filename, int2str(i));
   currentfile = strcat(currentfile, '.dat');
   fid = fopen(currentfile);

   % read in 'permutation class' squares and compare with nls'
   while (~feof(fid))
       A = fscanf(fid, '%u', [5 5]);
       contains40 = 0;
       contains10 = 0;
       contains6 = 0;

       % see if it's one of the 6
       for j=1:6
           if (isequal(squares6(:,:,j),A))
               contains6 = 1;
               break;
           end
       end

       for j=1:10
           if (isequal(squares10(:,:,j),A))
                contains10 = 1;
                break;
           end
       end

       for j=1:40
           if (isequal(squares40(:,:,j),A))
               contains40 = 1;
               break;
           end
       end
```

```
        if contains6
            fprintf(fid_out, '%u␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣%u', 6, i);
            fprintf(fid_out, '\n');
        elseif contains10
            fprintf(fid_out, '%u␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣%u', 10, i);
            fprintf(fid_out, '\n');
        elseif contains40
            fprintf(fid_out, '%u␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣%u', 40, i);
            fprintf(fid_out, '\n');
        end
    end
end

% close files
fclose('all');



function eq_ls
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Author: Anthony Morast
% Advisor: Brent Deschamp
% Undergraduate Research - SDSMT 2015
%
% This function compares files of Latin squares to find out which
    ones
% appear more than once. If the ls has not appeared it is output
    to another
% file. This was used on many occasions but mainly for finding
    isotopy classes.
% This method was also modified multiple times to serve different
    purposes but
% conceptually perform the same task.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%open file to write normalized ls to
fid2 = fopen('all6s2.dat', 'w');

fid = fopen('all6s.dat');

%read in first ls
A = fscanf(fid, '%u', [5 5]);
```

```
B(1,:,:) = A;

while ( ~feof(fid) )
    %read in LS
    A = fscanf(fid, '%u', [5 5]);

    if size(A) ~= 0
        %get size of array
        [m, n, o] = size(B);

        %re-initialize flag to "not in array"
        flag = 0;

        %loop through array and compare with current LS
        for j=1:m
            for k=1:5
                C(k,:) = B(j,k,:);
            end

            %if they are equal,
            if isequal(A,C)==1
                %set flag not to put in array,
                flag = 1;
                %and break from the loop
                break;
            end

        end
        %if the LS was not found in the array already put it into
           the array
        if flag == 0;
            B (m+1,:,:) = A;
        end
    end
end

%get final size of B array
[m,n,o] = size(B);

%write all LS in array B into a file
for i=1:m
    fprintf(fid2, '%u %u %u %u %u ', B(i,:,:));
    fprintf(fid2,'\n');
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%dont read in all values into an array, read in value, check if
   its in an array, and put it into
%the array if its not



function eq_ls2 ( filename )
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Author: Anthony Morast
% Advisor: Brent Deschamp
% Undergraduate Research - SDSMT 2015
%
% This function compares files of Latin squares to find out which
   ones
% appear more than once. If the ls has not appeared it is output
   to another
% file. This was used on many occasions but mainly for finding
   isotopy classes.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

fid = fopen(filename);

% read in first ls
A = fscanf(fid, '%u', [5 5]);
A = A';
B(1,:,:) = A;

while ( ~feof(fid) )
    % read in LS
    A = fscanf(fid, '%u', [5 5]);
    A = A';
    if size(A) ~= 0
        %get size of array
        [m, n, o] = size(B);

        % re-initialize flag to "not in array"
        flag = 0;

        % loop through array and compare with current LS
        for j=1:m
            for k=1:5
                C(k,:) = B(j,k,:);
            end

            % if they are equal,
            if isequal(A,C)==1
                % set flag not to put in array,
                flag = 1;
                % and break from the loop
                break;
            end

        end
        % if the LS was not found in the array already put it into
            the array
        if flag == 0;
            B (m+1,:,:) = A;
        end
    end
end

fclose(fid);
% get final size of B array
[m,n,o] = size(B);

% open file to write normalized ls to
fid2 = fopen(filename, 'w');

% write all LS in array B into a file
for i=1:m
    fprintf(fid2,'%u %u %u %u %u ', B(i,:,:));
    fprintf(fid2,'\n');
end
fclose(fid2);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% don't read in all values into an array, read in value, check if
   its in an array, and put it into
% the array if its not



function b = exitCondition( b, c, n )
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Author: Anthony Morast
% Advisor: Brent Deschamp
% Undergraduate Research - SDSMT 2015
%
% EXITCONDITION
% Checks if we are ready to exit the main loop in move_to_normal
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    [h,i,l] = size(c);

    for i=1:l
        C = c(:,:,i);
        [h,i,p] = size(b);
        for j=1:p
            if isequal(b(:,:,j),C)
                b(:,:,j) = [];
                break;
            end
        end
    end
end



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Author: Anthony Morast
% Advisor: Brent Deschamp
% Undergraduate Research - SDSMT 2015
%
% This script reads in the Latin squares of order 5 and if they
   are reduced
% prints them to another file. This method can easily be altered
   to check
% more or less indices, allowing it to be used to find reduced
   Latin squares of
% any order.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

fid = fopen('latin_squares_order5.dat');
fid_write = fopen ('reduced_squares_o5.dat', 'a');

while ~feof(fid)
```

```
    A = fscanf (fid , '%u', [5  5]);
    if (~isequal(A, []))
        A = A';
                    % add or remove checks here for higher or lower
                        orders.
        if (A(1,1) == 1 && A(1,2) == 2 && A(1,3) == 3 && A(1,4) ==
            4 && A(1,5) == 5)
            fprintf (fid_write , '%u␣%u␣%u␣%u␣%u␣', A);
            fprintf (fid_write , '\n');
        end
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This would be the code altered to find the reduced Latin squares
    of order 4.
% It is the same code as above but the modifications have been
    made and
% different files are read and written.
%
% fid = fopen('4_latin.dat');
% fid_write = fopen('reduced_order4.dat', 'w');
%
% while ~feof(fid)
%     A = fscanf(fid, '%u', [4  4]);
%     A = A';
%     if isequal(A, '')
%         continue;
%     end
%
%     if (A(1,1) == 1 && A(1,2) == 2 && A(1,3) == 3 && A(1,4) == 4)
%         fprintf(fid_write, '%u %u %u %u ', A);
%         fprintf(fid_write, '\n');
%     end
% end
%
% fclose('all');
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



function gen_norm (B)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% generates all normalized Latin squares by permuting
%    1 2 3 4   then normalizing this matrix to see if
%A = 2 1 4 3   it generates any of the 3 other normalized
%    3 4 1 2   Latin squares of order 4.
%    4 3 2 1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% open file containing permutation of order 4
fid = fopen ( '4_perm.txt' );

% read in number of permutations
n = fscanf(fid, '%u',1);

% open file for writing
fid2 = fopen('normls4.dat', 'w');

m=n;
i = 0;
while ( i < factorial(n) )
    A = B;
    % read in permutation
    sigma = fscanf ( fid, '%u', n);
    for j=2:n
        for k=2:n
            x = A(j,k);
            if x == 1
                A(j,k) = sigma(1);
            elseif x == 2
                A(j,k) = sigma(2);
            elseif x == 3
                A(j,k) = sigma(3);
            else
                A(j,k) = sigma(4);
            end
        end
    end
    % print out Latin square
    fprintf(fid2, '%u %u %u %u ', A);
    fprintf(fid2, '\n');
    i = i+1;
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Author: Anthony Morast
% Advisor: Brent Deschamp
% Undergraduate Research - SDSMT 2015
%
%Generate Latin Squares Order 5
%
% Generates all the Latin squares of order 5 by passing in each
    normalized
% Latin square one by one and permuting the rows and columns.
    Should create
% a file with 161280 squares.

%open file containing normalized Latin squares of order 5
fid = fopen('normalized_ls_order5.dat');

while (~feof(fid))
    A = fscanf(fid, '%u', [5 5]);

    %pass square into generate_ls
    generate_ls(A);
end



function  generate_ls ( A )
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Author: Anthony Morast
% Advisor: Brent Deschamp
% Undergraduate Research - SDSMT 2015
%
% This function takes a normalized Latin square and permutes the
    rows and columns to generate
% Latin squares of a particular order. The permutations of n are
    generated via a c++ program.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% open file containing all permutations of 1,2,3, and 4
fid = fopen ( '4_perm.txt' );

% Get size of Latin square
[m,n] = size(A);
```

```
% create a second instance of the Latin square passed into the
    function
B = A;

% read in first value in file, # permutations
fscanf(fid,'%d', 1);

% initialize k
k = 0;

% while not end of file, i.e. factorial of Latin square order
while ( k < factorial(m) )

  %read permutation into sigma vector
  sigma = fscanf(fid,'%d', m);

  for i=1:n
    % permute columns
    % get value of column for D
    j = sigma(i);
    % move column i to column j
    D(:,j) = B(:,i);
    % D is the permuted matrix
  end

  % permute rows, D as input
  permute_rows(D);

  % increment k value
  k = k + 1;
end



function indexes = getIndexes(pairs)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Author: Anthony Morast
% Advisor: Brent Deschamp
% Undergraduate Research - SDSMT 2015
%
% This function is used to return a new list of squares that need
    be checked
```

```
% for mutual orthogonality. The squares used here will be the
   indexes used in
% the previous round of orthogonality checking. This function
   essentially culls
% the list of reduced squares so that less computations are done.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% returns a list of the new indexes to be checked
% pairs - N squares, checking for MOLS of size N+1
[h,k,l] = size(pairs);
indexes = zeros(0);
first = 1;

for i = 1:l
        % boolean values to determine if the square is already
           part of the MOL set
    containsOne = 0;
    containsTwo = 0;
    containsThree = 0;

        % get the N squares, in this case size = 3
    one = pairs(1,1,i);
    two = pairs(1,2,i);
    three = pairs (1,3,i);

        % put the first square in the indexed vector
    if (first)
       first = 0;
       indexes = cat(1, indexes, one);
       indexes = cat(1, indexes, two);
       indexes = cat(1, indexes, three);
       loop = 2;
    end
    if (~first)
                % check if square in vector already
        for j=1:loop
            if indexes(j) == one
                containsOne = 1;
            end
            if indexes(j) == two
                containsTwo = 1;
            end
```

```
                if indexes(j) == three
                    containsThree = 1;
                end
            end
        end

            % if not in vector, put in vector
        if (~containsOne)
            indexes = cat (1, indexes, one);
            loop = loop + 1;
        end
        if (~containsTwo)
            indexes = cat (1, indexes, two);
            loop = loop + 1;
        end
        if (~containsThree)
            indexes = cat (1, indexes, three);
            loop = loop + 1;
        end
end



function c = getSquares( A )
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Author: Anthony Morast
% Advisor: Brent Deschamp
% Undergraduate Research - SDSMT 2015
%
%
% This function finds the '1' in each row and moves the column
    that
% contains it to the first column. The square is then normalized
    and
% concatenated to the c vector which is returned.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[m,n] = size(A);
c = zeros(m,n,0);

for i=2:m
    for j=1:n
    if (A(i,j) == 1)
```

```
            swap = j;
            break;
        end
        end
        % swap column with '1' to first column
        D = A;
        temp = D(:,1);
        D(:,1) = D(:,swap);
        D(:,swap) = temp;

        % swap current row to row one
        temp = D(1,:);
        D(1,:) = D(i,:);
        D(i,:) = temp;
        D = normalize_ls_5(D);

        % add the square to c if not in c
        [h,j,k] = size(c);
        contains = 0;
        for j=1:k
            if isequal(D,c(:,:,j))
                contains = 1;
            end
        end

        if contains == 0
            c = cat(3,D,c);
        end
    end
end



function is_latin (order)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Author: Anthony Morast
% Advisor: Brent Deschamp
% Undergraduate Research - SDSMT 2015
%
% Determines whether or not a Latin square read in from
% a file is a Latin square or not.
%
% If it is print it, if not do nothing.
```

```
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% open file(s)
fid = fopen('5iso_permutation.dat');
fid2 = fopen('normalized_ls5.dat','w');


% check if this is a Latin square
i = 0;
count = 0;
while (i < factorial(order))
    % read in Latin square
    A = fscanf(fid,'%u', [order,order]);

    %check rows
    for j=1:order
        for k=1:order
            for n=k+1:order
                if A(j,k) == A(j,n);
                    count = count + 1;
                    break
                end
            end
        end
    end

    % check columns
    count = 0;
    for j=1:order
        for k=1:order
            for n=k+1:order
                if A(k,j) == A(n,j);
                    count = count + 1;
                    break
                end
            end
        end
    end

    % write to file if it is a ls
    if count == 0
```

```
        fprintf(fid2, '%u %u %u %u %u ', A);
        fprintf(fid2, '\n');
    end

    % increment i
    i = i+1;
end



function main_class_overlord ( n )
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Author: Anthony Morast
% Advisor: Brent Deschamp
% Undergraduate Research - SDSMT 2015
%
% This program reads in the main class representatives of order n
  ( the
% parameter to the function ). The representatives are stored as a
    3D
% array. The first index holds n*n values to represent the rows of
    a Latin
% Square, the second holds n*n values to represent columns, and
  the third
% holds the symbols ( numbers ) that populate the Latin square.
%
% The three indices are permuted to create new Latin squares.
  After each
% permutation, 6 per representative, the new Latin squares have
  their
% symbols permuted to create all other Latin squares. This handles
    the
% identity case as the first permutation in the list of
  permutations is
% always the identity permutation in our files.
%
%
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% initialize D matrix
D = zeros(3,n*n);
```

```
% open file(s)
fid = fopen('main_class1_weeded.dat');
k = 1;

% loop through all main class representatives in the file
while ( ~feof(fid) )
    % read in main class representative and transpose
    A = fscanf(fid, '%u', [n n]);
    A = A';
    % check if the file has not read "too far"
    if size(A) ~= 0
        % create D array
        for i=1:n
            for j=1:n
                D(1,(n*i) - (n-j)) = i;
                D(2,(n*i) - (n-j)) = j;
                D(3,(n*i) - (n-j)) = A(i,j);
            end
        end

        % permute rows/cols/symbols via D array in 3! ways
        perm = 1;
        fid2 = fopen ('3_perm.txt');
        a = fscanf(fid2,'%u',1);

        % create file name used in the isotopy permutation, one
           file for
        % each main class representative
        filename = 'main_class_';
        filename2 = filename;
        filename = strcat(filename ,int2str(k));
        filename = strcat(filename,'.dat');

        while ( perm <= factorial(a) )
            % create dummy of D to be permuted.
            B = D;
            % read in permutation
            sigma = fscanf(fid2, '%u', a);
            % permute row/col/symbol
            B = SRC_permute(B, sigma);
            % put D = B back into Latin square form  (L)
            for i=1:n
```

```
                    for j=1:n
                        L(B(1,(n*i)-(n-j)),B(2,(n*i)-(n-j))) = B(3,(n*
                            i)-(n-j));
                    end
                end
                % permute symbols of the new Latin Square and write
                    permutations
                % to file
                permu_iso_2(L, filename);
                perm = perm + 1;
                eq_ls(filename2,k);
            end
        end
        k = k + 1;
end

% close all files
fclose('all');



function b = move_to_normal (n)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Author: Anthony Morast
% Advisor: Brent Deschamp
% Undergraduate Research - SDSMT 2015
%
% This program finds the 1's in each row of a Latin square and
    moves them
% into the first col. It then find and moves the remaining symbols
    to their
% "normalized" positions. That is 1 in first column, 2 in second
    col, etc.
%
% After the columns are in their proper order the rows are
    permuted into
% their proper order so that the Latin square is normalized.
%
% Then if the Latin square is different from the square that was
    read in,
% the new square and the old square are written out to check if
    the square
```

```
% was originally symmetric. Otherwise, nothing is done, or the
   square is
% written to s different file to ensure all squares are one or the
     other.
%
% NOTES:
% NLS/nls = normalized Latin square or normalized Latin squares
%
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


fid = fopen('nlsO5.dat');    % file to read nls from
count = 0;      % number of nls

b = zeros(n,n,0); % vector of arrays initialization

while (~feof(fid))  % get number of nls
    A = fscanf(fid, '%u', [n n]);
    if (isempty(A))
        break;
    end
    b = cat(3, b, A);  % put nls into vector
    count = count + 1;
end

[h,i,p] = size(b);
if (p > 0)
    bNotEmpty = 1;
else
    bNotEmpty = 0;
end

counter = 1;
filename = 'perm_class';
% loop through each nls
while (bNotEmpty)
    A = b(:,:,1);

    % call function that moves the '1' in each row to the (1,1)
        position and
    % then returns a vector of 4 matrices... remove from b
    c = getSquares(A);
```

```
% initialize cNotEmpty loop condition
[q,w,e] = size(c);
cNotEmpty = 1;
if (e == 0)
    cNotEmpty = 0;
end
used =zeros(n,n,0);

while (cNotEmpty)
    A = c(:,:,1);
    c(:,:,1) = [];
    used = cat (3, used, A);
    % for each matrix in the vector returned, store them in a
       master matrix
    % and recall that function on each one redoing these two
       steps
    d = getSquares(A);

    [q,w,e] = size(d);
    [r,t,y] = size(used);
    [u,o,p] = size(c);
    for i=1:e
        contains = 0;
        for j=1:y  % if not used
            if (isequal(used(:,:,j), d(:,:,i)))
               contains = 1;
               break;
            end
        end
        for j=1:p  % if not in c already
            if (isequal(c(:,:,j),d(:,:,i)))
                contains = 1;
                break;
            end
        end
        if ~contains
           c = cat(3,c, d(:,:,i));
        end
    end
    % once there are no new matrices found when that function
       is called exit
    % and write to a file; use next matrix in the b vector
```

```
            [q,w,e] = size(c);
            cNotEmpty = 1;
            if (e == 0)
                cNotEmpty = 0;
            end
        end

        % exit condition -- removes squares in c from b, this will
            eventually
         %make b empty
        b = exitCondition(b,used,n);

        [h,i,p] = size(b);
        if (p > 0)
            bNotEmpty = 1;
        else
            bNotEmpty = 0;
        end

        % get number of squares to write
        [z,x,v] = size(used);

        % create the write file name
        thisfilename = strcat(filename, int2str(counter));
        thisfilename = strcat(thisfilename,'.dat');

        % open file
        fid3 = fopen(thisfilename, 'w');

        % write to file
        for f=1:v
            fprintf (fid3, '%u␣%u␣%u␣%u␣%u␣',used(:,:,f));
            fprintf (fid3, '\n');
        end

        fclose(fid3);

        counter = counter + 1;
    end

    % close all files
```

```
fclose('all');



function B = normalize(A)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Author: Anthony Morast
% Advisor: Brent Deschamp
% Undergraduate Research - SDSMT 2015
%
% Dynamic Latin Square Normalize
%
% A program that takes a LS as input and returns the normalized
    square as
% output. Works for LS of any order.
%
% Bugs: This method may not work in particular cases.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[m,n] = size(A);

% permute rows into 1,2,3,...,n positions
for i=1:m
    % if the current row, first column isn't correct
    if A(i,1) ~= i
        % swap row into it's proper position
        temp = A(A(i,1),:);
        A(A(i,1),:) = A(i,:);
        A(i,:) = temp;
    end
end

% permute columns into 1,2,3,...,n positions
for i=1:n
    % if the current column, first row isn't correct
    if A(i,1) ~= i
        % swap row into it's proper position
        temp = A(:,A(1,i));
        A(:,A(1,i)) = A(:,i);
        A(:,i) = temp;
    end
end
```

```
% set the return value to the normalized square
B = A;



function C = normalize_ls(A)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Author: Anthony Morast
% Advisor: Brent Deschamp
% Undergraduate Research - SDSMT 2015
%
% Normalize Latin Square of Particular Order
%
% This function takes a Latin square of a particular order and
    rearranges the
% rows and columns until it is normalized. This function can be
    modified by
% adding more checks to more rows and columns until all N rows and
     columns of
% a Latin square of order N are checked. The function is currently
     set to
% normalize squares of order 3.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[m,n] = size(A);

% normalize the Latin square produced by the permutation
% swap columns
if( A(1,1) ~= 1)
    % find column starting with 1
    for i=1:n
        if A(1,i) == 1
            swap = i;
            break
        end
    end
    % swap columns
    D = A(:,1);
    A(:,1) = A(:,swap);
    A(:,swap) = D;
end

if (A(1,2) ~= 2)
```

```
    % find column starting with 2
    for i=1:m
        if A(1,i) == 2
            swap = i;
            break
        end
    end
    D = A(:,2);
    A(:,2) = A(:,swap);
    A(:,swap) = D;
end

if (A(1,3) ~= 3)
    % find column starting with 3
    for i=1:m
        if A(1,i) == 3
            swap = i;
            break
        end
    end
    D = A(:,3);
    A(:,3) = A(:,swap);
    A(:,swap) = D;
end

% swap rows
if (A(2,1) ~= 2)
    % find row starting with 2
    for i=1:m
        if A(i,1) == 2
            swap = i;
            break
        end
    end
    D = A(2,:);
    A(2,:) = A(swap,:);
    A(swap,:) = D;
end

if (A(3,1) ~= 3)
    % find row starting with 3
    for i=1:m
```

```
            if A(i,1) == 3
                swap = i;
                break
            end
        end
        D = A(3,:);
        A(3,:) = A(swap,:);
        A(swap,:) = D;
end

% return the square
C = A;



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Author: Anthony Morast
% Advisor: Brent Deschamp
% Undergraduate Research - SDSMT 2015
% Orthogonal Square Chart
%
% This function reads in the normalized Latin squares of order 5
    that have
% already been generated. The symbols of these Latin squares are
    permuted
% all possible ways and, if not already inside a file, are written
    to a
% file after they are normalized.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

fid3 = fopen ( 'missing_ls.txt' );

while ~feof(fid3)
    A = fscanf(fid3, '%u', [5 5]);
    if size(A) ~= 0
        permu_iso_2(A);
    end
end

% close all files
fclose('all');
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Author: Anthony Morast
% Advisor: Brent Deschamp
% Undergraduate Research - SDSMT 2015
% Orthogonal Square Chart
%
% Creates a file with the sets of orthogonal squares.. It finds
   the sets of
% two first then reads the file in checking each pair with the
   list of the
% squares in the vector.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% create vector of all reduced squares
fid = fopen('reduced_squares_o5.dat');

reduced_squares = zeros(5,5,0);
j  = 0;
while ~feof(fid)
   A = fscanf(fid, '%u', [5 5]);
    if ~isequal(A,'')
        reduced_squares = cat(3,reduced_squares,A);
        j = j + 1;
   end
end
[m,n,o] = size(reduced_squares);

fid_chart = fopen('mutually_orthogonal_squaresO5.dat','w');

% check each square in the vector with each other square in the
   vector
for i=1:o
   currSquare = reduced_squares(:,:,i);
   for j=i+1:o
      checkSquare = reduced_squares(:,:,j);
      % write the pairs to a file
      if (check_latin_square(currSquare,checkSquare))
         fprintf(fid_chart,'%u %u \n', i, j);
      end
   end
end
```

```
%close file containings sets of squares
fclose('all');



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Author: Anthony Morast
% Advisor: Brent Deschamp
% Undergraduate Research - SDSMT 2015
%
% This script finds sets of N+1 squares that are mutually
   orthogonal
% It will only use the squares that belonged to a set of N. That
   is,
% the method will loop through an input file which contain the
   positions of
% squares in another file. So the input file for this method is a
   list of
% positions that reference indexes in another file full of Latin
   squares.
% After being read in it is determined whether or not the set of N
    squares
% will form a set of N+1 orthogonal squares with the other squares
   .
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% open file containing square references of size N
fid = fopen('quadsO5.dat');

% array to hold the sets of size N
quads = zeros(1,4, 0);

% read in all sets of size N
while ~feof(fid)
   a = fscanf(fid, '%u %u',[1 4]);
   quads = cat(3, quads, a);
end

fclose(fid);

%get squares used in the sets of N
indexes = getIndexes(quads);
[q,w,e] = size(quads);
```

```
[r] = size(indexes);
quints = zeros (1,5,0);
first = 1;

% file containing all squares of the Latin square order in
   question
fid = fopen('reduced_squares_o5.dat');
reduced_squares = zeros(5,5,0);

% read in all Latin squares of the particular order
while ~feof(fid)
    A = fscanf(fid, '%u', [5 5]);
    reduced_squares = cat(3, reduced_squares, A);
end


for i=1:e
        % get squares at indexes corresponding locations in the
           master file
    currentSquareOne = reduced_squares (:, :, quads(1,1,i));
    currentSquareTwo = reduced_squares (:, :, quads(1,2,i));
    currentSquareThree = reduced_squares (:, :, quads(1,3,i));
    currentSquareFour = reduced_squares (:, :, quads(1,4,i));

    for j=1:r
                % get the square in the list of all squares to
                   check for mutual orthogonality
        checkSquare = reduced_squares(:, :, indexes(j));
        contains = 0;
        [p,o,u] = size(quints);

                % check the check square with each square in N-
                   sized vector
        if (check_latin_square(currentSquareOne, checkSquare) &&
           ...
            check_latin_square(currentSquareTwo, checkSquare) &&
               ...
            check_latin_square(currentSquareThree, checkSquare) &&
                ...
            check_latin_square(currentSquareFour, checkSquare))
```

```
                        % if mutually orthogonal add to list of
                          MOLS
                quint = [triples(1,1,i) triples(1,2,i) triples(1,3,i)
                   indexes(j)];

                if (first)
                     first = 0;
                     quints = cat (3, quints, quint);
                end
                for k=1:u
                   % if == 3 all same squares
                   [c,v] = size(intersect(quints(:,:,k), quint));
                   if (v == 5)
                        contains = 1;
                      break;
                   end
                end
                if (~contains)
                     quints = cat (3, quints, quint);
                end
            end
        end
end


fid3 = fopen ('quintsO4.dat','w');

% write all sets of size N+1 MOLS to a file
[p,o,u] = size(quints);
for i=1:u
     quints = quints(:,:,i);
     contains = 0;
     for j=i+1:u
          if (isequal(quint, quints(:,:,j)))
               contains = 1;
               break;
          end
     end
     if (~contains)
        fprintf(fid3, '%u␣%u␣%u␣%u\n', quint);
     end
end
```

```
% close all files
fclose('all');



function overlord
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Author: Anthony Morast
% Advisor: Brent Deschamp
% Undergraduate Research - SDSMT 2015
%
% This overlord function controls the logical flow required to
   find all
% orthogonal squares of an order, set up for order 4 at the time
   of writing.
% This is done by reading in each square, determining if they are
   orthogonal,
% if they are their positions within the file that is used for
   input is output
% to a different file which will hold the mutually orthogonal
   squares by
% holding their positions in the input file.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% open file to write Latin squares that 'work' together, i.e. are
   orthogonal
fid2 = fopen('latin_works_3.txt','w');

% loop through all Latin squares of order 4
for i=1:24
    % open file containing all Latin squares
    fid = fopen('reduced_order4.dat');

    %%%%%% Read in dummies
    for k=1:(i-1)
        dummy = fscanf(fid,'%u',[4 4]);
    end

    % Read in first LS
    C = fscanf(fid,'%u',[4 4]);

        % tranpose for Matlab read issues
        C =  C';

    for j=i+1:24
        % read in Latin square to be compared
        D = fscanf(fid,'%u',[4 4]);
        D=D';

        % if the Latin squares are orthogonal write their
           positions in the Latin square
                % file to another file
        if ( check_latin_square(C,D) == 1 )
            fprintf(fid2, '%u ', i);
            fprintf(fid2, '%u', j);
            fprintf(fid2,'\n');
        end
    end

    %close file containing order 4 Latin squares
    fclose(fid);
end

% close file
fclose(fid2);



function overlord_3iso ( root )
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Author: Anthony Morast
% Advisor: Brent Deschamp
% Undergraduate Research - SDSMT 2015
%
% Accepts input root, which is the root name of the file to be
   written. this
% program reads in one Latin square at a time, increments each
   index,
% permutes the symbols, and writes them each to an individual file
   . root is
% used to append the numbers 1-56 on these files.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% file containing Latin squares
fid = fopen ('mckay_nls5.txt');
```

```
for i=1:56

    % read in Latin square
    A = fscanf(fid, '%u',[5 5]);
    A = A';

        % convert to 1-based numbering
    for j=1:5
        for k=1:5
            A(j,k) = A(j,k) + 1;
        end
    end

        % create new file for each set of squares generated by a
           particular square
    filename = root;
    filename = strcat(filename,num2str(i),'.dat');
    filename2 = filename;
    permu_iso_2(A, filename);
    eq_ls2(filename2);
end

% close all files
fclose('all');



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Author: Anthony Morast
% Advisor: Brent Deschamp
% Undergraduate Research - SDSMT 2015
%
% Overlord Generate Latin Squares
%
% This script will read in all normalized Latin squares and
   permute the
% rows and columns to produce all Latin squares of order 6.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% file containing all normalized Latin squares
fid = fopen('normalized_ls_order6.dat');
```

```
i=1;
while ~feof(fid)

   % change this for other orders
   A = fscanf(fid, '%u', [6 6]);
   generate_ls(A);
   i=i+1;
   disp(i);
end



function overlord_red

%open file storing latin squares that work
fid = fopen('reduced_works.dat', 'w');

%loop through file of reduced latin squares
for i=1:24
    %open file containing reduced latin squares order 4
    fid2 = fopen('reduced_ls.dat');

    %read in dummies
    for k=1:(i-1)
        dummy = fscanf (fid2,'%u',[4 4]);
    end

    %read in first ls
    A = fscanf(fid2,'%u', [4 4]);
    %A = A';
    for j=i+1:24

        %read in ls to be compared
        B = fscanf(fid2,'%u',[4,4]);
        %B = B';
        %if the work output ls' that work
        if ( check_latin_square(A,B) == 1)
            fprintf(fid,'%u␣', i);
            fprintf(fid,'%u', j);
            fprintf(fid,'\n');

        end
    end
    fclose(fid2);
end

fclose(fid);



function permu_iso (B)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Author: Anthony Morast
% Advisor: Brent Deschamp
% Undergraduate Research - SDSMT 2015
%
% Permutes the symbols of a Latin square from an isotopy group
% to generate all normalized Latin squares.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%open file containing permutation of order 4
fid = fopen ( '5_perm.txt' );

%read in number of permutations
n = fscanf(fid, '%u',1);

%open file for writing
fid2 = fopen('missingMckayPerm.dat', 'w');

m=n;
i = 0;
while ( i < factorial(n) )
    A = B;

    %read in permutation
    sigma = fscanf ( fid, '%u', n);

        % permute the Latin square
    for j=1:m
        for k=1:n
            x = A(j,k);
            if x == 1
                A(j,k) = sigma(1);
            elseif x == 2
                A(j,k) = sigma(2);
            elseif x == 3
                A(j,k) = sigma(3);
            elseif x == 4
                A(j,k) = sigma(4);
            else
                A(j,k) = sigma(5);
            end
        end
    end

        % re-normalize the Latin square
    C = normalize_ls_5(A);

    %print out Latin square
    fprintf(fid2, '%u %u %u %u %u ', C);
    fprintf(fid2, '\n');
    i = i+1;
end

% close all files
fclose('all');



function D = permute_rows(A)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Author: Anthony Morast
% Advisor: Brent Deschamp
% Undergraduate Research - SDSMT 2015
%
% This function will read in a file of permutations which are to
    be applied to
% the Latin square passed into the method, 'A'. The permutations
    are written
% to another output file.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% open file containing permutations of 1, 2, 3
fid1 = fopen( '3_perm.txt' );

% open file for writing Latin squares of order 4
fid2 = fopen ('4_latin.txt(1)','a');
```

```
% get size of matrix passed in
[m,n] = size(A);

% read in first line of permutation file
fscanf ( fid1, '%u', 1 );

% create instance of passed in Latin square, initialize k value
D = A;
k = 0;

% loop until all permutations are found
while ( k < factorial(m-1) )
    % read values into sigma vector to determine new row positions
    sigma = fscanf ( fid1, '%u', n-1 );

    % permute rows
    for i=1:n-1
        j = sigma(i);
        % move row j+1 to row i+1
        D(j+1,:) = A(i+1,:);
    end

    % print Latin square to file
    fprintf(fid2, '%u␣%u␣%u␣%u␣', D);

    % increment k value
    k = k + 1;
    % print new line
    fprintf(fid2, '\n');
end

% close files
fclose('all');



function reduced_latin
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Author: Anthony Morast
% Advisor: Brent Deschamp
% Undergraduate Research - SDSMT 2015
%
```

```
% This method will currently only work with Latin squares of order
    4. This is
% because the loop is set to 576 iterations which is the number of
    order 4
% Latin squares.
%
% This function will read in a file of Latin squares and determine
    if they are
% reduced, that is if the first row has the order 1, 2, 3, ... If
   the Latin
% square is reduced it is printed to a different file. This method
    strictly
% weeds out the reduced Latin squares from a file containing all
   the squares.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% open list of order four Latin squares
fid = fopen('LS4_backup.dat');

% open file to store reduced Latin squares.
fid2 = fopen('reduced_ls.dat', 'a');

i = 0;
% while not all Latin squares are read, read and check if reduced
while ( i < 576 )
    % read in Latin square
    A = fscanf(fid, '%u', [1 16]);

        % check if the square is reduced.
    if ( A(1) == 1 && A(2) == 2 && A(3) == 3 && A(4) == 4 )
        %print to file if reduced.
        fprintf(fid2, '%u %u %u %u ', A);
        fprintf(fid2,'\n');
    end

    %increment counter
    i = i + 1;
end

%close files
fclose('all');
```

```
function relate (B)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Author: Anthony Morast
% Advisor: Brent Deschamp
% Undergraduate Research - SDSMT 2015

% Generates all normalized Latin squares by permuting
%     1 2 3 4   then normalizing this matrix to see if
% A = 2 1 4 3   it generates any of the 3 other normalized
%     3 4 1 2   Latin squares of order 4.
%     4 3 2 1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%open file containing permutation of order 4
fid = fopen ( '4_perm.txt' );

% read in number of permutations
n = fscanf(fid, '%u',1);

% open file for writing
fid2 = fopen('A_permutation.dat', 'w');

m=n;
i = 0;
while ( i < factorial(n) )
    A = B;
    % read in permutation
    sigma = fscanf ( fid, '%u', n);

        % permute the Latin squares rows
    for j=1:n
        for k=1:n
            x = A(j,k);
            if x == 1
                A(j,k) = sigma(1);
            elseif x == 2
                A(j,k) = sigma(2);
            elseif x == 3
                A(j,k) = sigma(3);
            else
                A(j,k) = sigma(4);
            end
        end
    end
        % re-normalize the Latin square
    C = normalize_ls(A);

    % print out latin square
    fprintf(fid2, '%u %u %u %u ', C);
    fprintf(fid2, '\n');
    i = i+1;
end

fclose('all');



function B = SRC_permute ( D, sigma )
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Author: Anthony Morast
% Advisor: Brent Deschamp
% Undergraduate Research - SDSMT 2015
%
% This function takes a D matrix of size 3 x n*n and a vector
    sigma which
% is the current permutation. The rows of the D matrix are
    permuted in
% accordance to the sigma vector. The newly permuted D matrix is
    passed
% back to the overlord function which handles execution control.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

B(1,:) = D(sigma(1),:);
B(2,:) = D(sigma(2),:);
B(3,:) = D(sigma(3),:);



function symmetric_squares ( filename, order )
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Author: Anthony Morast
% Advisor: Brent Deschamp
% Undergraduate Research - SDSMT 2015
%
```

```
% This program opens a file for input and a file for output. It
% reads in Latin squares from the read file and determines if the
   transpose
% is equivalent to the original. If it is they squares are
   symmetric, thus
% the rows and columns being permuted will not create a different
   normalized
% Latin Square.
%
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%open read file
fid = fopen ( filename );

%open output file - rename for different squares as this method is
     called many times
filename_write = 'symmetric_squares';
filename_write = strcat(filename_write,'_');
filename_write = strcat(filename_write, int2str(order));
filename_write = strcat(filename_write, '.dat');
fid2 = fopen ( filename_write, 'w' );

% read in all squares
while ( ~feof(fid) )
    % read in Latin square
    A = fscanf(fid, '%u', [order,order]);

        % convert the squares to 1-based numbering
    for i=1:order
        for j=1:order
            A(i,j) = A(i,j) + 1;
        end
    end

    % create format for writing the squares - for robustness
    format = '%u';
    for i=1:order - 1
        format = strcat(format, ' %u');
    end
    format = [format, ' '];
    %if not last square in file
    if size(A) ~= 0
        %determine if square is symmetric
        B = A';
        if isequal(A,B) == 1
            fprintf( fid2, format, A );
            fprintf ( fid2, '\n');
        end
    end

end

fclose ('all');
```

## 6.2 C# Code

```
  using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Numerics;

namespace Combanatorics
{
    /// <summary>
    /// This class will implement many common operations used in
       combinatorics. These will be
    /// similar to combinations and permutations, producing all
       combinations of a certain number,
    /// etc.
    /// </summary>
    public class Combinatorics
    {
        /// <summary>
        /// This method takes in an integer and produces all the
           possible permutations of the integer.
        /// For example, passing in '4' will produce a List of
           Arrays of the format enty one = 1,2,3,4
        /// entry two = 1,2,4,3; etc.
        /// </summary>
        /// <param name="order">The of which to find permutations
           </param>
        public static List<long[]> ProducePermutations(long order)
        {
            throw new NotImplementedException("Implementation␣
               saved␣for␣future␣release.");

            return new List<long[]>();
        }

        /// <summary>
        /// This method takes in an integer and produces all the
           possible permutations of the integer.
        /// For example, passing in '4' will produce a List of
           Arrays of the format enty one = 1,2,3,4
        /// entry two = 1,2,4,3; etc. It will also write these
           values out to a file.
        /// </summary>
        /// <param name="order"> The of which to find permutations
           </param>
        /// <param name="filename"> THe file path with filename to
            output permutations.</param>
        /// <returns></returns>
        public static List<long[]> ProducePermutations (long order
           , string filename)
        {
            throw new NotImplementedException("Implementation␣
               saved␣for␣future␣release.");

            return new List<long[]>();
        }

        public static List<long[]> ProduceCombinations (long order
           )
        {
            throw new NotImplementedException("Implementation␣
               saved␣for␣future␣release.");

            return new List<long[]>();
        }

        /// <summary>
        /// Returns the number of permutations of a given set of
           numbers, n Pick k.
        /// For values larger than approximately P(50, 40) use the
            BigInteger in the
        /// Systems.Numerics and the BigInterger implementation of
            this algorithm in this
        /// library, otherwise an overflow will occur.
        /// </summary>
        /// <param name="order"></param>
        /// <returns></returns>
        public static long Permutations(long n, long k)
        {
            if (k > n)
                throw new ApplicationException(string.Format(
                        "Cannot␣find␣number␣of␣permutations␣for␣
                           {0}␣pick␣{1},␣k␣>␣n.", n, k));

            if (k == 0)
                return 1;
            else if (k == 1)
                return n;

            long result = 1;
            long stop = (n - k);

            for (long i = n; i > stop; i--)
                result *= i;
            Console.WriteLine("n␣=␣{0}␣k␣=␣{1}␣i␣=␣{2}", n, k,
               result);

            return result;
        }



        /// <summary>
        /// The BigInteger implementation to find all permutations
            of a given set of numbers, n pick k.
        /// This implementation should be used if very large
           values, values larger than 1.9e+19, are
        /// expected, as this will overflow a 64 bit integer.
        /// </summary>
        /// <param name="n"></param>
        /// <param name="k"></param>
        /// <returns></returns>
        //public static BigInteger Permutations(int n, int k)
        //{
        //    throw new NotImplementedException("This method will
           be added in a future release.");
        //    if (k > n)
        //        throw new ApplicationException(string.Format(
        //                "Cannot find number of permutations for
           {0} pick {1}, k > n.", n, k));

        //    if (k == 0)
        //        return 1;
        //    else if (k == 1)
        //        return n;

        //    BigInteger result = 0;

        //    return result;
        //}

        /// <summary>
        /// Returns the total number of combinations of a given
           set of numbers, n Choose k.
        /// For values larger than approximately 50 C 40 use the
           BigInteger in the
        /// Systems.Numerics and the BigInterger implementation of
            this algorithm in this
        /// library, otherwise an overflow will occur.
        /// </summary>
        /// <param name="order"></param>
        /// <returns></returns>
        public static long Combinations(long n, long k)
        {
            if (k > n)
                throw new ApplicationException(string.Format(
                        "Cannot find number of combinations for
                           {0} choose {1}, k > n.", n, k));
            if (k == 0)
                return 1;
            else if (k == 1)
```

```
                return n;

            long result = 1;

            for (long d = 1; d <= k; d++)
            {
                result *= n--;
                result /= d;
            }

            return result;
        }

        /// <summary>
        /// The BigInteger implementation to find all combinations
            of a given set of numbers, n choose k.
        /// This implementation should be used if very large
           values, values larger than 1.9e+19, are
        /// expected, as this will overflow a 64 bit integer.
        /// </summary>
        /// <param name="n"></param>
        /// <param name="k"></param>
        /// <returns></returns>
        //public static BigInteger Combinations(int n, int k)
        //{
        //     throw new NotImplementedException("This␣method␣will␣
           be␣added␣in␣a␣future␣release.");
        //     if (k > n)
        //         throw new ApplicationException(string.Format(
        //                 "Cannot␣find␣number␣of␣combinations␣for␣
           {0}␣pick␣{1},␣k␣>␣n.", n, k));

        //     if (k == 0)
        //         return 1;
        //     else if (k == 1)
        //         return n;

        //     BigInteger result = 0;

        //     return result;
        //}
    }
```

```
}


  using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Combanatorics;

namespace CombinatoricsTests
{
    [TestClass]
    public class CombinatoricsTests
    {
        [TestMethod]
        public void CombinationsTests()
        {
            int correctCount = 0;

            if (Combinatorics.Combinations(100, 3) == 161700)     //
                correct
                correctCount++;
            if (Combinatorics.Combinations(100, 4) == 3921225)   //
                correct
                correctCount++;
            if (Combinatorics.Combinations(100, 4) == 12)         //
                incorrect
                correctCount++;
            if (Combinatorics.Combinations(100, 1) == 100)        //
                correct
                correctCount++;
            if (Combinatorics.Combinations(100, 0) == 1)          //
                correct
                correctCount++;

            Assert.AreEqual(correctCount, 4);
        }

        [TestMethod]
        [ExpectedException(typeof(ApplicationException), "Invalid␣
           parameters␣for␣Combinations")]
        public void CombinationsInvalidArdsTest()
        {
            Combinatorics.Combinations(1, 10);
        }

        [TestMethod]
        public void PermutationsTests()
        {
            int correctCount = 0;

            if (Combinatorics.Permutations(100, 3) == 970200)
               // correct
                correctCount++;
            if (Combinatorics.Permutations(100, 5) == 9034502400)
               // correct
                correctCount++;

            Assert.AreEqual(correctCount, 2);
        }

        [TestMethod]
        [ExpectedException(typeof(ApplicationException), "Invalid␣
           parameters␣for␣Permutations")]
        public void PremutationsInvalidArgsTest()
        {
            Combinatorics.Permutations(100, 1000);
        }
    }
}
```

```
  using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Combanatorics;
using System.IO;

namespace LatinSquaresOrthogonal
{
    class GenerateReducedOrder6
    {
        public static List<LatinSquare> GenerateReduced()
        {
```

```
        List<LatinSquare> squares = ReadInSquares("order6norm.
           txt");
        List<List<int>> permutations = ReadInPermutations("5
           _perm.txt");

        List<LatinSquare> permutedSquares = new List<
           LatinSquare>();
        // 012345 103254 234501 325410 450123 541032
        //LatinSquare ls = new LatinSquare(6, new List<int>{1,
            2, 3, 4, 5, 6, 2, 1, 4, 3, 6, 5, 3, 4, 5, 6, 1, 2,
        //                                        4, 3, 6, 5,
           2, 1, 5, 6, 1, 2, 3, 4, 6, 5, 2, 1, 4, 3});

        for (int i = 0; i < squares.Count; i++)
        {
            LatinSquare currentSquare = squares[i];
            for (int j = 0; j < permutations.Count; j++)
                permutedSquares.Add(currentSquare.PermuteRows(
                   permutations[j]));
        }

        return permutedSquares;
    }

    private static List<LatinSquare> ReadInSquares (string
       filename)
    {
        List<LatinSquare> returnValue = new List<LatinSquare
           >();
        using (StreamReader sr = new StreamReader(filename))
        {
            List<int> current = new List<int>();
            string line = "";

            while ((line = sr.ReadLine()) != null)
            {
                current = new List<int>();
                string[] ne = line.Split(new char[] { ' ' });
                for (int i = 0; i < ne.Count(); i++)
                {
                    long curr = long.Parse(ne[i]);
                    if ( i == 0 )
                    {
                        current.Add(1);
                        current.Add((int)((curr / 10000)) + 1)
                            ;
                        current.Add((int)((curr % 10000) /
                            1000) + 1);
                        current.Add((int)((curr % 1000) / 100)
                             + 1);
                        current.Add((int)((curr % 100) / 10) +
                             1);
                        current.Add((int)((curr % 10)) + 1);
                    }
                    else
                    {
                        current.Add((int)(curr / 100000) + 1);
                        current.Add((int)((curr % 100000) /
                            10000) + 1);
                        current.Add((int)((curr % 10000) /
                            1000) + 1);
                        current.Add((int)((curr % 1000) / 100)
                             + 1);
                        current.Add((int)((curr % 100) / 10) +
                             1);
                        current.Add((int)((curr % 10)) + 1);
                    }
                }
                returnValue.Add(new LatinSquare(6, current));
            }
        }

        return returnValue;
    }

    private static List<List<int>> ReadInPermutations(string
        filename)
    {
        List<List<int>> returnVal = new List<List<int>>();
        List<int> current = new List<int>{ 1 };

        using (StreamReader sr = new StreamReader(filename))
        {
            string line;
```

```
                while ((line = sr.ReadLine()) != null)
                {
                    current = new List<int> { 1 };
                    string[] ne = line.Split(new char[] { ' ' });
                    ne = ne.Where(x => !string.IsNullOrEmpty(x) &&
                        !string.IsNullOrWhiteSpace(x)).ToArray();
                    foreach (var item in ne)
                        current.Add(int.Parse(item)+1);
                    returnVal.Add(current);
                }
            }

            return returnVal;
        }
    }
}
```

```
  using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Combanatorics
{
    public class LatinSquare
    {
        /// <summary>
        /// The size of the Latin Square (square will be size x
           size)
        /// </summary>
        private int squareOrder = 0;

        /// <summary>
        /// List of all the values within the Latin Square
        /// </summary>
        private List<int> values = new List<int>();

        /// <summary>
        /// Create an empty Latin square of size n x n
        /// </summary>
/// <param name="n"></param>
public LatinSquare(int order)
{
    squareOrder = order;
}

/// <summary>
/// Create a Latin square of size n x n and fill it with
   the values. The list of values should be
/// "row_major". That is, the first n (where n is the
   order of the square) elements should be the
/// entries for row one in the square. The second n should
    be row two, etc.
/// </summary>
/// <param name="n"></param>
/// <param name="listOfValues"></param>
public LatinSquare(int order, List<int> listOfValues)
{
    squareOrder = order;

    // check if the list of values contains n squared
       entries (a square)
    SetValues(listOfValues);
}

/// <summary>
/// Returns the order of the Latin square.
/// </summary>
/// <returns></returns>
internal int GetOrder()
{
    return this.squareOrder;
}

/// <summary>
/// Set the values of the Latin square.
/// </summary>
/// <param name="valueList"></param>
/// <returns></returns>
internal void SetValues(List<int> valueList)
{
    string errorMessage = "";
    if (valueList.Count != (squareOrder * squareOrder))
    {
        throw new Exception("Incorrect number of values to
           fill Latin Square");
    }

    if (CheckValues(valueList, out errorMessage))
        this.values = valueList;
    else
    {
        throw new Exception(errorMessage);
    }
}

/// <summary>
/// Checks to ensure the values will create a Latin square
/// </summary>
/// <param name="valueList"></param>
/// <returns></returns>
internal bool CheckValues(List<int> valueList, out string
   error)
{
    // create rows and column lists
    List<int[]> rows = new List<int[]>();
    List<int[]> cols = new List<int[]>();
    for (int i = 0; i < squareOrder; i++)
    {
        rows.Add(new int[squareOrder]);
        cols.Add(new int[squareOrder]);
    }

    // fill the row and columns lists
    for (int i = 0; i < squareOrder; i++)
    {
        for (int j = 0; j < squareOrder; j++)
        {
            int currentElemnet = valueList[(i *
               squareOrder) + j];
            if ( currentElemnet > squareOrder ||
               currentElemnet <= 0)
            {
                    error = string.Format(
                        @"Element␣'{0}'␣in␣row␣{1}␣column␣{2}␣
                           is␣outside␣of␣valid␣range␣values.␣A
                           ␣Latin␣square␣should␣contain␣
                           elements␣from␣1␣to␣N,␣where␣N␣is␣
                           the␣order␣of␣the␣square",
                        currentElemnet, i + 1, j + 1);
                    return false;
                }

                rows[i][j] = currentElemnet;
                cols[j][i] = currentElemnet;
            }
        }

        // check for same elements in rows or columns
        for (int i = 0; i < squareOrder; i++)
        {
            if (rows[i].Distinct().Count() < squareOrder)
            {
                error = "Row␣" + (i+1).ToString() + "␣does␣not
                   ␣contain␣distinct␣elemnts.";
                return false;
            }

            if (cols[i].Distinct().Count() < squareOrder)
            {
                error = "Column␣" + (i+1).ToString() + "␣does␣
                   not␣contain␣distinct␣elemnts.";
                return false;
            }
        }

        error = "";
        return true;
    }

    /// <summary>
    /// Returns a specific element in the square by
       calculating the offset within
    /// the values list.
    /// </summary>
```

```
/// <param name="row"></param>
/// <param name="col"></param>
/// <returns></returns>
internal int GetElement(int row, int col)
{
    if (row > squareOrder || col > squareOrder)
        throw new ApplicationException(string.Format("
           Index␣{0},{1}␣not␣in␣square␣of␣size␣{2}x{2}.",
           row, col, squareOrder));

    row--;
    col--;
    return values[(row * squareOrder) + col];
}

/// <summary>
/// <para> Override the ToString() method to either print
   the Latin square to
/// the screen or to display that the Latin square is
   empty
/// </para>
/// </summary>
/// <returns></returns>
public override string ToString()
{
    string ret_string = "";

    if (values.Count == 0)
        return "Empty␣Latin␣square.\n";

    for (var i = 0; i < values.Count; i++)
    {
        if (i % squareOrder == 0 && i != 0)
        {
            ret_string += "\n";
        }

        ret_string += Convert.ToString(values[i]) + "␣";
    }

    return ret_string;
}
```

```
    public bool IsEqual(LatinSquare ls)
    {
        if (this.values.Intersect(ls.values).Count() ==
           squareOrder * squareOrder)
        {
            return true;
        }
        return false;
    }
}

public static class LatinSquareExtensions
{
    /// <summary>
    /// Fills the Latin square with the values in listOfValues
       . The list of values should be
    /// "row major". That is, the first n (where n is the
       order of the square) elements should be the
    /// entries for row one in the square. The second n should
        be row two, etc.
    /// </summary>
    /// <param name="ls"></param>
    /// <param name="listOfValues"></param>
    public static void Fill(this LatinSquare ls, List<int>
       listOfValues)
    {
        ls.SetValues(listOfValues);
    }

    /// <summary>
    /// Returns the order of this Latin square.
    /// </summary>
    /// <param name="ls"></param>
    /// <returns></returns>
    public static int GetOrder(this LatinSquare ls)
    {
        return ls.GetOrder();
    }

    /// <summary>
/// Determins if this square is mutually orhogonal with a
   given square.
/// </summary>
/// <param name="ls"></param>
/// <param name="checkSquare"></param>
public static bool IsOrthogonal(this LatinSquare ls,
   LatinSquare checkSquare)
{
    // squares not the same size
    if (ls.GetOrder() != checkSquare.GetOrder())
        return false;

    int rows = ls.GetOrder();
    int cols = rows;
    List<Tuple<int, int>> pairs = new List<Tuple<int, int
       >>();

    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
        {
            var currentPair = new Tuple<int, int>(ls.
               GetElement(i+1, j+1), checkSquare.
               GetElement(i+1, j+1));
            if (pairs.Contains(currentPair))
                return false;

            pairs.Add(currentPair);
        }
    }

    return true;
}

/// <summary>
/// Returns true if the square belongs to the same isotopy
    class as the given square.
/// </summary>
/// <param name="ls"></param>
/// <param name="checkSquare"></param>
/// <returns></returns>
```

```
public static bool IsSameIsotopyClass(this LatinSquare ls,
    LatinSquare checkSquare)
{
    throw new NotImplementedException("Implementation␣
       saved␣for␣future␣release.");

    // squares not the same size, can't be some isotopy
       class
    if (ls.GetOrder() != checkSquare.GetOrder())
        return false;

    return false;
}

/// <summary>
/// Returns true if the square belongs to the same main
   class as the given square.
/// </summary>
/// <param name="ls"></param>
/// <param name="checkSquare"></param>
/// <returns></returns>
public static bool IsSameMainClass(this LatinSquare ls,
   LatinSquare checkSquare)
{
    throw new NotImplementedException("Implementation␣
       saved␣for␣future␣release.");

    // squares not the same size, can't be some isotopy
       class
    if (ls.GetOrder() != checkSquare.GetOrder())
        return false;

    return false;
}

/// <summary>
/// Returns the element at a a specific row and column
   within the Latin square.
/// </summary>
/// <param name="ls"></param>
/// <param name="row"></param>
/// <param name="column"></param>
/// <returns></returns>
public static int GetElementAtPosition(this LatinSquare ls
   , int row, int column)
{
    return ls.GetElement(row, column);
}

/// <summary>
/// Swaps the rows in the Latin square in accordance to
   the newIndices list of integers.
/// This list must be the same size as the order of the
   Latin square. E.g. if the list contains
/// { 1, 3, 4, 5, 6, 2} (assuming the order of the square
   is 6) then row 1 doesn't change,
/// row 2 swaps with row 6, row 3 with row 2, row 4 with
   row 3, row 5 with row 4, and row 6
/// with row 5.
/// </summary>
/// <param name="ls"></param>
/// <param name="newIndices"></param>
public static LatinSquare PermuteRows (this LatinSquare ls
   , List<int> newIndices)
{
    if (newIndices.Count != ls.GetOrder())
        throw new ApplicationException("Not␣enough␣
           indicies␣to␣swap␣in␣PermuteRows␣method.");
    if (newIndices.Select(x => x > ls.GetOrder()).Count()
       == 0)
        throw new ApplicationException("Invalid␣index␣in␣
           new␣indices␣list␣in␣PermuteRows␣method");

    int squareOrder = ls.GetOrder();

    List<int[]> oldRows = new List<int[]>();
    List<int[]> newRows = new List<int[]>();
    for (int i = 0; i < squareOrder; i++)
    {
        oldRows.Add(new int[squareOrder]);
        newRows.Add(new int[squareOrder]);
    }

    // fill the row and columns lists
```

```
            for (int i = 0; i < squareOrder; i++)
                for (int j = 0; j < squareOrder; j++)
                    oldRows[i][j] = ls.GetElementAtPosition(i + 1,
                        j + 1);

            List<int> newVals = new List<int>();
            for (int i = 0; i < squareOrder; i++)
            {
                newRows[i] = oldRows[newIndices[i] - 1];
                newVals = newVals.Concat(newRows[i]).ToList();
            }

            return new LatinSquare(ls.GetOrder(), newVals);
        }

        public static bool IsNormal(this LatinSquare ls)
        {
            for (int i = 0; i < ls.GetOrder(); i++)
            {
                if (ls.GetElement(i+1, 1) != i+1 || ls.GetElement
                   (1, i+1) != i+1)
                    return false;
            }
            return true;
        }
    }
}
```

```
  using System;
using Combanatorics;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.Collections.Generic;

namespace CombinatoricsTests
{
    [TestClass]
    public class LatinSquareTests
    {
        [TestMethod]
        public void CreateSquaresTest()
        {
```

```
    LatinSquare square1 = new LatinSquare(10);
    LatinSquare square2 = new LatinSquare(3, new List<int
       >{ 1, 2, 3, 3, 1, 2, 2, 3, 1 });
    LatinSquare square3 = new LatinSquare(4, new List<int>
        { 1, 2, 3, 4, 2, 1, 4, 3, 3, 4, 2, 1, 4, 3, 1, 2
       });

    //Assert.AreEqual(square3.GetOrder(), 4);
}

[TestMethod]
[ExpectedException(typeof(Exception), "Bad␣row␣in␣creating
   ␣the␣square.")]
public void BadRowSquareTest()
{
    LatinSquare square3 = new LatinSquare(4, new List<int>
        { 1, 2, 3, 4, 2, 1, 4, 3, 3, 4, 2, 1, 4, 4, 1, 2
       });
}

[TestMethod]
[ExpectedException(typeof(Exception), "Bad␣column␣in␣
   creating␣the␣square.")]
public void BadColSquareTest()
{
    LatinSquare square3 = new LatinSquare(4, new List<int>
        { 1, 2, 3, 4, 2, 1, 4, 3, 3, 4, 2, 1, 4, 3, 2, 1
       });
}

[TestMethod]
[ExpectedException(typeof(Exception), "Not␣enough␣values␣
   in␣square.")]
public void IncorrectNumberOfValuesTest ()
{
    LatinSquare square3 = new LatinSquare(3, new List<int>
        { 1, 2, 3, 4, 2, 1, 4, 3, 3, 4, 2, 1, 4, 3, 2, 1
       });
}

[TestMethod]
public void FillSquaresTest()
```

```
{
    LatinSquare square1 = new LatinSquare(4);
    square1.Fill(new List<int>{ 1, 2, 3, 4, 2, 1, 4, 3, 3,
        4, 2, 1, 4, 3, 1, 2 });

    Assert.AreEqual(square1.GetOrder(), 4);
}

[TestMethod]
public void GetOrderTest()
{
    LatinSquare square = new LatinSquare(4);
    LatinSquare sqaure1 = new LatinSquare(4, new List<int>
        { 1, 2, 3, 4, 2, 1, 4, 3, 3, 4, 2, 1, 4, 3, 1, 2
       });

    Assert.AreEqual(square.GetOrder(), square.GetOrder());
}

[TestMethod]
public void SetValuesTest()
{
    LatinSquare square1 = new LatinSquare(3);
    square1.SetValues(new List<int> { 1, 2, 3, 2, 3, 1, 3,
        1, 2 });

    Assert.AreEqual(square1.GetOrder(), 3);
}

[TestMethod]
public void GetElementAtPositionTest()
{
    int correctElementCount = 0;
    LatinSquare square1 = new LatinSquare(4);
    square1.Fill(new List<int> { 1, 2, 3, 4, 2, 1, 4, 3,
       3, 4, 2, 1, 4, 3, 1, 2 });

    if (square1.GetElementAtPosition(1, 1) == 1)     //
       correct - count = 1
        correctElementCount++;
    if (square1.GetElementAtPosition(1, 2) == 2)     //
       correct - count = 2
```

```
            correctElementCount++;
        if (square1.GetElementAtPosition(1, 3) == 3)      //
           correct - count = 3
            correctElementCount++;
        if (square1.GetElementAtPosition(1, 4) == 4)      //
           correct - count = 4
            correctElementCount++;
        if (square1.GetElementAtPosition(4, 4) == 2)      //
           correct - count = 5
            correctElementCount++;
        if (square1.GetElementAtPosition(3, 3) == 2)      //
           correct - count = 6
            correctElementCount++;
        if (square1.GetElementAtPosition(2, 4) == 4)      //
           incorrect - count = 6
            correctElementCount++;
        if (square1.GetElementAtPosition(2, 1) == 2)      //
           correct - count = 7
            correctElementCount++;
        if (square1.GetElementAtPosition(3, 4) == 1)      //
           correct - count = 8
            correctElementCount++;
        if (square1.GetElementAtPosition(4, 3) == 2)      //
           incorrect - count = 8
            correctElementCount++;
        if (square1.GetElementAtPosition(3, 1) == 3)      //
           correct - count = 9
            correctElementCount++;

        Assert.AreEqual(correctElementCount, 9);
    }

    [TestMethod]
    public void PermuteRowsTest()
    {
        LatinSquare square1 = new LatinSquare(3);
        square1.SetValues(new List<int> { 1, 2, 3, 2, 3, 1, 3,
            1, 2 });

        Console.WriteLine(square1.ToString());
        Console.WriteLine();
        square1 = square1.PermuteRows(new List<int> { 1, 3, 2
           });
        Console.WriteLine(square1.ToString());
        Console.WriteLine();
        square1 = square1.PermuteRows(new List<int> { 2, 3, 1
           });
        Console.WriteLine(square1.ToString());
        Console.WriteLine();

        LatinSquare square2 = new LatinSquare(4);
        square2.Fill(new List<int> { 1, 2, 3, 4, 2, 1, 4, 3,
           3, 4, 2, 1, 4, 3, 1, 2 });

        Console.WriteLine(square2.ToString());
        Console.WriteLine();
        square2 = square2.PermuteRows(new List<int> { 4, 1, 2,
            3 });
        Console.WriteLine(square2.ToString());
        Console.WriteLine();
        square2 = square2.PermuteRows(new List<int> { 2, 3, 4,
            1 });
        Console.WriteLine(square2.ToString());
        Console.WriteLine();
    }


    [TestMethod]
    public void CheckGoodValuesTest()
    {

    }

    [TestMethod]
    public void IsMutuallyOrthogonalTest()
    {

    }

    [TestMethod]
    public void IsSameIsotopyTest()
    {
        }

        [TestMethod]
        public void IsSameMainClassTest()
        {

        }
    }
}



  using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Combanatorics;
using System.IO;

namespace LatinSquaresOrthogonal
{
    class OrderFiveOrthogonal
    {
        public static void FindOrthogonalOrder5 (String[] args)
        {
            LatinSquare square1 = new LatinSquare(int.Parse(args
               [0]));
            LatinSquare square2 = new LatinSquare(int.Parse(args
               [0]));
            LatinSquare square3 = new LatinSquare(int.Parse(args
               [0]));

            List<LatinSquare> squares = ReadInSquares("
               reduced_squares_o5.dat", int.Parse(args[0]));

            List<Tuple<int, int>> orthogonalPairs = new List<Tuple
               <int, int>>();
            List<Tuple<int, int, int>> orthogonalTriplets = new
               List<Tuple<int, int, int>>();

            // Get pairs of MOLS
            for (int i = 0; i < squares.Count; i++)
```

```
{
    square1 = squares[i];
    for (int j = i; j < squares.Count; j++)
    {
        square2 = squares[j];

        if (square1.IsOrthogonal(square2))
            orthogonalPairs.Add(new Tuple<int, int>(i,
                j));
    }
}

// Get element indicies that will be used for further
   testing
List<int> distinct = new List<int>();

foreach (var pair in orthogonalPairs)
{
    if (!distinct.Contains(pair.Item1))
        distinct.Add(pair.Item1);
    if (!distinct.Contains(pair.Item2))
        distinct.Add(pair.Item2);
}

// Get triplets of MOLS
for (int i = 0; i < orthogonalPairs.Count; i++)
{
    square1 = squares[orthogonalPairs[i].Item1];
    square2 = squares[orthogonalPairs[i].Item2];

    for (int j = 0; j < distinct.Count; j++)
    {
        // square1 == square3 or square2 == square3
        if (distinct[j] == (orthogonalPairs[i].Item2)
           || distinct[j] == (orthogonalPairs[i].Item1
           ))
            continue;

        square3 = squares[distinct[j]];
        var currentTrip = new Tuple<int, int, int>(
           orthogonalPairs[i].Item1, orthogonalPairs[i
           ].Item2, distinct[j]);
```

```
                if (square3.IsOrthogonal(square1) && square3.
                    IsOrthogonal(square2)
                     && !Contains(currentTrip,
                         orthogonalTriplets))
                      orthogonalTriplets.Add(currentTrip);
            }
        }

        Console.WriteLine(orthogonalPairs.Count + "␣" +
            orthogonalTriplets.Count);
        Console.Read();
    }

    // Read the squares in from the file
    private static List<LatinSquare> ReadInSquares(string
        filename, int order)
    {
        StreamReader reader = new StreamReader(filename);
        List<LatinSquare> squares = new List<LatinSquare>();

        string line;
        List<int> current = new List<int>();

        while ((line = reader.ReadLine()) != null)
        {
            string[] ne = line.Split(new char[] { ' ' });
            ne = ne.Where(x => !string.IsNullOrEmpty(x) && !
                string.IsNullOrWhiteSpace(x)).ToArray();
            current = ne.Select(x => int.Parse(x)).ToList();
            squares.Add(new LatinSquare(order, current));
        }

        return squares;
    }

    private static bool Contains(Tuple<int,int,int> current,
        List<Tuple<int,int,int>> orthogonalSets)
    {
        int count = 0;
        foreach (var item in orthogonalSets)
        {
            count = 0;
            if (item.Item1 == current.Item1 || item.Item1 ==
               current.Item2 || item.Item1 == current.Item3)
                count++;
            if (item.Item2 == current.Item1 || item.Item2 ==
               current.Item2 || item.Item2 == current.Item3)
                count++;
            if (item.Item3 == current.Item1 || item.Item3 ==
               current.Item2 || item.Item3 == current.Item3)
                count++;

            if (count == 3) break;
        }

        return count == 3 ? true : false;
    }

    private static bool Contains(Tuple<int, int, int, int>
       current, List<Tuple<int, int, int,int>> orthogonalSets)
    {
        int count = 0;
        foreach (var item in orthogonalSets)
        {
            count = 0;

            if (item.Item1 == current.Item1 || item.Item1 ==
               current.Item2 || item.Item1 == current.Item3 ||
                item.Item1 == current.Item4)
                count++;
            if (item.Item2 == current.Item1 || item.Item2 ==
               current.Item2 || item.Item2 == current.Item3 ||
                item.Item2 == current.Item4)
                count++;
            if (item.Item3 == current.Item1 || item.Item3 ==
               current.Item2 || item.Item3 == current.Item3 ||
                item.Item3 == current.Item4)
                count++;
            if (item.Item4 == current.Item1 || item.Item4 ==
               current.Item2 || item.Item4 == current.Item3 ||
                item.Item4 == current.Item4)
                count++;
                if (count == 4) break;
            }

            return count == 4 ? true : false;
        }
    }
}



  using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Combanatorics;
using System.IO;
using System.Threading;

namespace LatinSquaresOrthogonal
{
    class OrthogonalOrder6
    {
        public static void FindOrthogonalOrder6 (List<LatinSquare>
            squares)
        {
            Console.WriteLine("Reduced␣found,␣count␣=␣{0}",
               squares.Count);

            // create threads
            List<Tuple<int,int>> thread1Pairs = new List<Tuple<int
               ,int>>();
            List<LatinSquare> thread1squares = new List<
               LatinSquare>(); // squares 0 - 51,960
            int thread1Start = 0;
            Thread thread1 = new Thread(() => { thread1Pairs =
               GetLocalPairs(squares, thread1squares, thread1Start
               ); });

            List<Tuple<int, int>> thread2Pairs = new List<Tuple<
               int, int>>();
            List<LatinSquare> thread2squares = new List<
               LatinSquare>();  // squares 51,961 - 110,961
```

```
int thread2Start = 51961;
Thread thread2 = new Thread(() => { thread2Pairs =
   GetLocalPairs(squares, thread2squares, thread2Start
   ); });

List<Tuple<int, int>> thread3Pairs = new List<Tuple<
   int, int>>();
List<LatinSquare> thread3squares = new List<
   LatinSquare>(); // squares 110,962 - 179,961
int thread3tart = 110962;
Thread thread3 = new Thread(() => { thread3Pairs =
   GetLocalPairs(squares, thread3squares, thread3tart)
   ; });

List<Tuple<int, int>> thread4Pairs = new List<Tuple<
   int, int>>();
List<LatinSquare> thread4squares = new List<
   LatinSquare>(); // squares 179,962 - 262,962
int thread4Start = 179962;
Thread thread4 = new Thread(() => { thread4Pairs =
   GetLocalPairs(squares, thread4squares, thread4Start
   ); });

List<Tuple<int, int>> thread5Pairs = new List<Tuple<
   int, int>>();
List<LatinSquare> thread5squares = new List<
   LatinSquare>();  // squares 262,963 - 366,963
int thread5Start = 262963;
Thread thread5 = new Thread(() => { thread5Pairs =
   GetLocalPairs(squares, thread5squares, thread5Start
   ); });

List<Tuple<int, int>> thread6Pairs = new List<Tuple<
   int, int>>();
List<LatinSquare> thread6squares = new List<
   LatinSquare>();  // squares 366,964 - 504,964
int thread6Start = 366964;
Thread thread6 = new Thread(() => { thread6Pairs =
   GetLocalPairs(squares, thread6squares, thread6Start
   ); });
```

```
List<Tuple<int, int>> thread7Pairs = new List<Tuple<
   int, int>>();
List<LatinSquare> thread7squares = new List<
   LatinSquare>();  // squares 504,965 - 712,965
int thread7Start = 504965;
Thread thread7 = new Thread(() => { thread7Pairs =
   GetLocalPairs(squares, thread7squares, thread7Start
   ); });

List<Tuple<int, int>> thread8Pairs = new List<Tuple<
   int, int>>();
List<LatinSquare> thread8squares = new List<
   LatinSquare>();  // squares 712,966 - 1128960
int thread8Start = 712966;
Thread thread8 = new Thread(() => { thread8Pairs =
   GetLocalPairs(squares, thread8squares, thread8Start
   ); });

// fill thread square lists
for (int i = 0; i < squares.Count; i++)
{
    LatinSquare curr = squares[i];
    if (i <= 51960)
        thread1squares.Add(curr);
    else if (i <= 110961)
        thread2squares.Add(curr);
    else if (i <= 179961)
        thread3squares.Add(curr);
    else if (i <= 262962)
        thread4squares.Add(curr);
    else if (i <= 366963)
        thread5squares.Add(curr);
    else if (i <= 504964)
        thread6squares.Add(curr);
    else if (i <= 712965)
        thread7squares.Add(curr);
    else
        thread8squares.Add(curr);
}

// start and join threads
```

```
List<Thread> threads = new List<Thread> { thread1,
   thread2, thread3, thread4, thread5, thread6,
   thread7, thread8 };
foreach (var thread in threads)
    thread.Start();

foreach (var thread in threads)
    thread.Join();

List<Tuple<int, int>> pairs = new List<Tuple<int, int
   >>();
//for (int i = 0; i < squares.Count; i++)
//{
//    LatinSquare currentSquare = squares[i];
//    Console.WriteLine(i);
//    for (int j = i; j < squares.Count; j++)
//    {
//        if (currentSquare.IsOrthogonal(squares[j]))
//            pairs.Add(new Tuple<int,int>(i,j));
//    }
//}

Console.WriteLine("{0}␣{1}␣{2}␣{3}␣{4}␣{5}␣{6}␣{7}",
   thread1Pairs.Count, thread2Pairs.Count,
   thread3Pairs.Count, thread4Pairs.Count,
   thread5Pairs.Count,
    thread6Pairs.Count, thread7Pairs.Count,
       thread8Pairs.Count);

pairs.Concat(thread1Pairs);
pairs.Concat(thread2Pairs);
pairs.Concat(thread3Pairs);
pairs.Concat(thread4Pairs);
pairs.Concat(thread5Pairs);
pairs.Concat(thread6Pairs);
pairs.Concat(thread7Pairs);
pairs.Concat(thread8Pairs);

Console.WriteLine(pairs.Count);

List<int> distinct = new List<int>();
foreach (var pair in pairs)
```

```
        {
            if (!distinct.Contains(pair.Item1))
                distinct.Add(pair.Item1);
            if (!distinct.Contains(pair.Item2))
                distinct.Add(pair.Item2);
        }

        using (StreamWriter sw = new StreamWriter("
           orthogonalPairs6.dat"))
        {
            foreach (var pair in pairs)
            {
                sw.WriteLine("{0}␣{1}", pair.Item1 + 1, pair.
                   Item2 + 1);
            }
        }

        for (int i = 0; i < distinct.Count; i++)
        {
            if (squares[i].IsNormal())
                Console.WriteLine("{0}\n", squares[i]);
        }
    }

    public static List<Tuple<int,int>> GetLocalPairs (List<
       LatinSquare> squares, List<LatinSquare> mySquares, int
       startIndex)
    {
        // go through mysquares list and check orthogonailty
           with all squares in squares, can start at beginning
            index of mysquares
        // that is, if my squares are squares 222,000 thru
           444,000 then start comparing and squares[222000]
        List<Tuple<int, int>> localPairs = new List<Tuple<int,
            int>>();

        for (int i = 0; i < mySquares.Count; i++)
        {
            Console.WriteLine(i);
            for (int j = i + startIndex; j < squares.Count; j
               ++)
            {
                    if (mySquares[i].IsOrthogonal(squares[j]))
                        localPairs.Add(new Tuple<int, int>(i, j));
                }
            }
                return localPairs;
        }
    }
}
```

```
  using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IO;
using Combanatorics;

namespace LatinSquaresOrthogonal
{
    class Program
    {
        static void Main(string[] args)
        {
            //OrderFiveOrthogonal.FindOrthogonalOrder5(args);
            List<LatinSquare> squares = GenerateReducedOrder6.
                GenerateReduced();
            OrthogonalOrder6.FindOrthogonalOrder6(squares);
        }
    }
}
```

## 6.3 C++ Code

```
#include <iostream>
#include <string>
#include <stdio.h>
#include <stdlib.h>
#include <sstream>
#include <algorithm>
```

```
#include <vector>

using namespace std;

class LatinSquare
{
    public:
        LatinSquare (int order);
        LatinSquare (int order, vector<int> values);
        void Fill  (vector<int> values);
        int GetOrder ();
        bool IsOrthogonal (LatinSquare checkSq);
        bool IsSameIsotopyClass (LatinSquare checkSq);
        bool IsSameMainClass (LatinSquare checkSq);
        int GetElementAtPosition (int row, int column);
        LatinSquare PermuteRows (vector<int> newIndices);
        bool IsNormal ();
        void Print();

    protected:
        int GetElement (int row, int col);
        bool CheckValues (vector<int> valueList, string &error);
        void SetValues (vector<int> valueList);

    private:
        string to_string(int value);
        bool Distinct (int rowOrCol[]);
        int squareOrder;
        vector<int> values;
};

LatinSquare::LatinSquare (int order)
{
    squareOrder = order;
}

LatinSquare::LatinSquare (int order, vector<int> valueList)
{
    squareOrder = order;
    SetValues(valueList);
}
```

```
void LatinSquare::Fill (vector<int> valueList)
{
    SetValues(valueList);
}

int LatinSquare::GetOrder()
{
    return squareOrder;
}

bool LatinSquare::IsOrthogonal (LatinSquare checksq)
{
    vector<int> currentPair;
    vector< vector<int> > pairs;

    currentPair.push_back(0);
    currentPair.push_back(0);

    for (int i = 0; i < squareOrder; i++)
    {
        for (int j = 0; j < squareOrder; j++)
        {
            currentPair[0] = GetElementAtPosition(i+1, j+1);
            currentPair[1] = checksq.GetElementAtPosition(i+1, j
               +1);

            if (!pairs.empty())
            {
                if (find(pairs.begin(), pairs.end(), currentPair)
                   != pairs.end())
                    return false;
                else
                    pairs.push_back(currentPair);
            }
            else
                pairs.push_back(currentPair);
        }
    }

    return true;
}
```

```
bool LatinSquare::IsSameIsotopyClass (LatinSquare checkSq)
{
    printf("IsSameIsotopyClass␣not␣yet␣implemented.");
    throw new exception;

    return false;
}

bool LatinSquare::IsSameMainClass (LatinSquare checkSq)
{
    printf ("IsSameIsotopyClass␣not␣yet␣implemented.");
    throw new exception;

    return false;
}

int LatinSquare::GetElementAtPosition (int row, int column)
{
    return GetElement(row, column);
}

LatinSquare LatinSquare::PermuteRows (vector<int> newIndices)
{
    if (newIndices.size() != squareOrder)
    {
        cout << "Not␣enough␣indices␣to␣swap␣in␣PermuteRows␣method.
            " << endl;
        throw new exception;
    }

    vector< vector<int> > oldRows;
    vector< vector<int> > newRows;

    for (int i = 0; i < squareOrder; i++)
    {
        vector<int> row;
        for (int j = 0; j < squareOrder; j++)
            row.push_back(GetElementAtPosition(i+1, j+1));

        oldRows.push_back(row);
    }
    vector<int> newVals;
    for (int i = 0; i < squareOrder; i++)
    {
        if (newIndices[i] > squareOrder || newIndices[i] < 1)
        {
            cout << "Invalid index in new indices list in 
               PermuteRows method." << endl;
            throw new exception;
        }

        newRows.push_back(oldRows[newIndices[i] - 1]);
        vector<int> current = newRows[i];
        for (int j = 0; j < newRows[i].size(); j++)
            newVals.push_back(current[j]);
    }

    LatinSquare newSq (squareOrder, newVals);

    return newSq;
}

bool LatinSquare::IsNormal()
{
    return false;
}

int LatinSquare::GetElement (int row, int col)
{
    if (row > squareOrder || col > squareOrder)
    {
        printf("Index %d,%d not in square of size %d,%d", row, col
            , squareOrder, squareOrder);
        throw new exception;
    }

    row--;
    col--;
    return values[(row*squareOrder) + col];
}

bool LatinSquare::CheckValues (vector<int> valueList, string &
   error)
```

```
{
    int **rows = new int *[squareOrder];
    int **cols = new int *[squareOrder];

    for (int i = 0; i < squareOrder; i++)
    {
        rows[i] = new int [squareOrder];
        cols[i] = new int [squareOrder];
    }

    for (int i = 0; i < squareOrder; i++)
    {
        for (int j = 0; j < squareOrder; j++)
        {
            int currentElement = valueList[(i*squareOrder) + j];
            if (currentElement > squareOrder || currentElement <
               1)
            {
                error = "Element " + to_string(currentElement) + "
                   in row " +
                        to_string(i+1) + " and column " +
                           to_string(j+1) +
                        " is outside the valid range of values. 
                           Latin squares should contain " +
                        " elements between 1 and the order of the 
                           square.";
                return false;
            }

            rows[i][j] = currentElement;
            cols[j][i] = currentElement;
        }
    }

    for (int i = 0; i < squareOrder; i++)
    {
        if (!Distinct(rows[i]))
        {
            error = "Row " + to_string(i+1) + " does not contain 
               distinct elements.";
            return false;
        }
        if (!Distinct(cols[i]))
        {
            error = "Col " + to_string(i+1) + " does not contain 
               distinct elements.";
            return false;
        }
    }

    error = "";
    return true;
}

void LatinSquare::SetValues (vector<int> valueList)
{
    int size = valueList.size();
    string error = "";

    if (size != (squareOrder*squareOrder))
    {
        cout << "Incorrect number of values to fill Latin Square. 
           Size: " << size << " Order: " << squareOrder << "." <<
            endl;
        throw new exception;
    }
    else if (CheckValues(valueList, error))
    {
        values = valueList;
    }
    else
    {
        cout << error << endl;
        throw new exception;
    }
}

string LatinSquare::to_string (int value)
{
    ostringstream oss;
    oss << value;
    return oss.str();
}
```

```
bool LatinSquare::Distinct (int rowOrCol[])
{
    int size = (sizeof(rowOrCol)/sizeof(*rowOrCol));
    sort(rowOrCol, rowOrCol + size);

    for (int i = 0; i < size; i++)
    {
        if (rowOrCol[i] == rowOrCol[i+1])
            return false;
    }

    return true;
}

void LatinSquare::Print()
{
    string printString = "";

    for (int i = 0; i < squareOrder*squareOrder; i++)
    {
        if (i % squareOrder == 0 && i != 0)
            printString += "\n";

        printString += to_string(values[i]) + " ";
    }

    cout << printString << endl;
}



#include "LatinSquare.h"
#include <fstream>

vector<LatinSquare> ReadInSquares (string filename);
vector< vector<int> > ReadInPermutations (string filename);
vector<LatinSquare> GenerateReduced ();

void FindOrthogonalOrder6 (vector<LatinSquare> reducedSquares);


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

```
{
    vector<LatinSquare> reducedSquares = GenerateReduced();
    cout << reducedSquares.size() << endl;

    FindOrthogonalOrder6 (reducedSquares);

    return 0;
}

vector<LatinSquare> GenerateReduced()
{
    vector<LatinSquare> reduced;

    vector<LatinSquare> squares = ReadInSquares("order6norm.txt");
    vector< vector<int> > permutations = ReadInPermutations("5
       _perm.txt");

    cout << squares.size() << "␣"  << permutations.size() << endl;

    for (int i = 0; i < squares.size(); i++)
    {
        LatinSquare current = squares[i];
        for (int j = 0; j < permutations.size(); j++)
            reduced.push_back(current.PermuteRows(permutations[j])
               );
    }

    return reduced;
}

vector<LatinSquare> ReadInSquares (string filename)
{
    vector<LatinSquare> squares;

    ifstream fin;
    fin.open(filename.c_str());
    if (!fin)
    {
        cout << "Failed␣to␣open␣file␣" << filename << endl;
        throw new exception;
    }
```

```
long read;
int count = 0;
vector<int> current;
while (fin >> read)
{

    if (count == 6)
    {
        count = 0;
        LatinSquare sq(6, current);
        squares.push_back(sq);
        current.clear();
    }

    if (count == 0)
    {
        current.push_back(1);
        current.push_back((int)((read / 10000)) + 1);
        current.push_back((int)((read % 10000) / 1000) + 1);
        current.push_back((int)((read % 1000) / 100) + 1);
        current.push_back((int)((read % 100) / 10) + 1);
        current.push_back((int)((read % 10)) + 1);
    }
    else
    {
        current.push_back((int)(read / 100000) + 1);
        current.push_back((int)((read % 100000) / 10000) + 1);
        current.push_back((int)((read % 10000) / 1000) + 1);
        current.push_back((int)((read % 1000) / 100) + 1);
        current.push_back((int)((read % 100) / 10) + 1);
        current.push_back((int)((read % 10)) + 1);
    }
    count++;
}

if (current.size() > 0)
{
    LatinSquare sq (6, current);
    squares.push_back(sq);
}

fin.close();
```

```
    return squares;
}

vector< vector<int> > ReadInPermutations (string filename)
{
    vector< vector<int> > permutations;

    ifstream fin (filename.c_str());
    if (!fin)
    {
        cout << "Failed to open file " << filename << endl;
        throw new exception;
    }

    int read;
    int count = 0;
    vector<int> permutation;
    // since we only permute the last 5 rows of the square we need
        to add 1 at the beginning of each permutation to keep the
       first row
    // as the first row. Otherwise, only 5 values will be in the
       vector, which isnt enough to permute the square.
    permutation.push_back(1);
    fin >> read;          // the top of the 5_perm.txt has '5' which
        is the number of items being permuted, dont wann this in
       our vector.
    while (fin >> read)
    {
        if (count == 5)
        {
            count = 0;
            permutations.push_back(permutation);
            permutation.clear();
            permutation.push_back(1);
        }

        permutation.push_back(read + 1);
        count++;
    }
    if (permutation.size() > 0)
        permutations.push_back(permutation);
    fin.close();
    return permutations;
}

void FindOrthogonalOrder6 (vector<LatinSquare> reducedSquares)
{

}
```

# 7 Bibliography

## References

[1] *Applied Combinatorics.* Chapman and Hall, 2008.

[2] More about latin squares, March 2015.

[3] Mutually orthogonal latin squares, March 2015.

[4] Lars Dovling Andersen. Chapter on the history of latin squares, March 2015.

[5] Brendan McKay. Latin squares, March 2015.

[6] Dhananjay P. Mehendale. Finite projective plains, March 2015.

[7] Merrian-Webster. Latin square, April 2015.

[8] A. Nova. Uses of latin squares, March 2015.