The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.
A (normal) Magic Square is a square grid of numbers 1..n², such that the columns, rows and diagonals sum to the value given by \( \frac{1}{2}(n+n^3) \), where \( n \) is the length of an edge of the square grid.

The aim of this project was to develop programs to create (produce a single square of any size) and solve (find all possible solutions to a square of a given size) Magic Squares.

Creating a Magic Square was done by implementing existing methods that produces Magic Squares. The Siamese, Lozenge and Bachet de Mézeriac methods were implemented to produce squares of odd size. Agrippa Diagonal and Filling 9 Blocks methods were implemented to create squares of size \( n=4m \) (\( m>0 \)) and the Lux and Strachney methods were implemented for squares of size \( n=4m+2 \) (\( m>0 \)).

Solving a Magic Square via brute-force methods yielded inefficient code and a quicker method was adopted and implemented (the sets-of-sums method). The ability to make this code run in parallel shortened computation time further and allowed me to compute the total solutions to the 5x5 Magic Square.

I briefly finish off with my attempts at developing my own method to create a Magic Square.
ACKNOWLEDGEMENTS

I would like to thank Dr. Matthew Hubbard for his advice and guidance throughout this project. I would also like to thank Dr. Haiko Müller for his invaluable feedback on my mid-project report.

Many thanks go to Frances Robinson and David Toyne for sparing CPU time to calculate the number of solutions for the 5x5 Magic Square. Words cannot express how much their help is appreciated.

Lastly, thanks go to my brother, Richard, for not only his time and effort in dedicating CPU time too, but also for his support and patience throughout.
## CONTENTS

1 INTRODUCTION .................................................................................................................. 1
  1.1 The Problem .................................................................................................................. 1
  1.2 Aims and Objectives ..................................................................................................... 1
  1.3 Methodology ............................................................................................................... 2
  1.4 Schedule ..................................................................................................................... 2

2 BACKGROUND .................................................................................................................. 3
  2.1 What is a Magic Square? .............................................................................................. 3
  2.2 History .......................................................................................................................... 3
  2.3 Properties of Magic Squares ......................................................................................... 4
  2.4 Scope of Project .......................................................................................................... 6
  2.5 Number of Solutions and Transformations ................................................................. 7

3 EXISTING ALGORITHMS ................................................................................................. 8
  3.1 Odd Order Magic Squares ........................................................................................... 8
  3.2 Double-Even Order (n=4m) Magic Squares ................................................................. 9
  3.3 Single-Even Order (n=4m+2, m>0) Magic Squares ..................................................... 10

4 CREATING A MAGIC SQUARE ......................................................................................... 12
  4.1 Analysis ....................................................................................................................... 12
  4.2 Design ......................................................................................................................... 12
    4.2.1 Siamese Method (odd order) ................................................................................ 13
    4.2.2 Lozenge Method (odd order) ............................................................................. 13
    4.2.3 Bachet de Mézeriac Method (odd order) ............................................................ 14
    4.2.4 Agrippa Diagonal & Filling 9 Blocks Methods (double-even order) ................. 14
    4.2.5 Lux Method (single-even order) ....................................................................... 16
    4.2.6 Strachney Method (single-even order) ............................................................. 17
  4.3 Implementation & Testing ......................................................................................... 18
  4.4 Evaluation ................................................................................................................... 19

5 SOLVING A MAGIC SQUARE ......................................................................................... 20
  5.1 Brute-Force Method ................................................................................................. 20
    5.1.1 Analysis ............................................................................................................. 20
    5.1.2 Design .............................................................................................................. 21
    5.1.3 Implementation & Evaluation ....................................................................... 22
  5.2 Order 5 Square (sets-of-sums method) ........................................................................ 23
    5.2.1 Analysis ............................................................................................................. 24
    5.2.2 Design .............................................................................................................. 24
    5.2.3 Implementation & Evaluation ....................................................................... 28
  5.3 Order 5 Square (parallel sets-of-sums method) .......................................................... 30

6 CREATING A MAGIC SQUARE (GENERIC METHOD) .................................................. 33
  6.1 Analysis ....................................................................................................................... 33
  6.2 Design ......................................................................................................................... 34
    6.2.1 Generating All Tuples ..................................................................................... 34
    6.2.2 Finding Sets-of-Sums ..................................................................................... 35
    6.2.3 Filling the Square ............................................................................................ 36

7 PROJECT EVALUATION .................................................................................................. 37
  7.1 Extensions .................................................................................................................. 38
1 INTRODUCTION

1.1 The Problem

If I asked you to draw me a 3x3 Magic Square using the numbers 1 to 9 (with each number being used once), that’s not a problem. If I asked you to draw me all the 3x3 squares, that’s more of a challenge but is possible. Now suppose I asked you to draw me a 10x10 square or even a 1000x1000 square, it’ll be silly to try.

What about if I asked you to draw all known 4x4 Magic Squares? You’d give up after a few minutes when you’d realised the scale of the problem (there are 7040 solutions by the way!).

Which leads onto the aim of this project…

1.2 Aims and Objectives

As specified in the mid-project report: -

The aim of this project is to develop programs to create and solve Magic Squares.

The following objectives were set in order to fulfil this:

- Investigate Magic Squares: What are they? Why are they magic? What is the history behind Magic Squares? What types of Magic Squares are there? What is the problem? Why do we need algorithms to create/solve Magic Squares?
- Investigate existing algorithms: What algorithms exist for creating/solving Magic Squares and how efficient are they?
- Create Magic Squares: Define what we mean by ‘create’ a Magic Square, develop code to do this, and compare with existing algorithms.
- Solve Magic Squares: Define what we mean by ‘solve’ a Magic Square, develop code to do this, and compare with existing algorithms.

The minimum requirements are to:

- Investigate Magic Squares and decide which to deal with.
- Investigate existing algorithms to solve/create Magic Squares.
- Develop code to create Magic Squares: find a single solution to a Magic Square.
- Develop code to solve Magic Squares: find all possible solutions of a Magic Square.

The possible extensions are:

- Consider other types of Magic Squares.
- Consider finding all solutions of a Magic Square efficiently.
1.3 Methodology

Being a joint-honour student, I had not covered methodologies in the modules I had taken. However, I was familiar with the Waterfall method, as I had employed this for my computing project during A-Levels. The Waterfall method is a stage by stage process going through analysis, design, implementation and evaluation. Based on previous experience, an element of backtracking is useful to optimise the objectives set at each stage of the process.

Familiarity of the Waterfall method meant it was easier for me to adopt and as it is also a typical method employed in software development, it was an appropriate method to use for this project.

1.4 Schedule

<table>
<thead>
<tr>
<th>WEEK COMMENCING</th>
<th>MILESTONE</th>
</tr>
</thead>
<tbody>
<tr>
<td>27 Oct 03</td>
<td>literature review: general</td>
</tr>
<tr>
<td>17 Nov 03</td>
<td>literature review: existing methods</td>
</tr>
<tr>
<td>29 Dec 03</td>
<td>semester 1 revision and exams</td>
</tr>
<tr>
<td>26 Jan 04</td>
<td>create Magic Squares: analysis</td>
</tr>
<tr>
<td>02 Feb 04</td>
<td>create Magic Squares: design</td>
</tr>
<tr>
<td>16 Feb 04</td>
<td>create Magic Squares: implementation</td>
</tr>
<tr>
<td>01 Mar 04</td>
<td>create Magic Squares: evaluation</td>
</tr>
<tr>
<td>08 Mar 04</td>
<td>solve Magic Squares: analysis</td>
</tr>
<tr>
<td>15 Mar 04</td>
<td>solve Magic Squares: design</td>
</tr>
<tr>
<td>29 Mar 04</td>
<td>solve Magic Squares: implementation</td>
</tr>
<tr>
<td>12 Apr 04</td>
<td>solve Magic Squares: evaluation</td>
</tr>
<tr>
<td>19 Apr 04</td>
<td>project evaluation</td>
</tr>
<tr>
<td>26 Apr 04</td>
<td>submit project</td>
</tr>
</tbody>
</table>

The above schedule was devised at the beginning of December to meet the minimum requirements. The literature review was to fulfil the first two objectives set above. The lack of material available to me in the University of Leeds’ libraries meant that this information was primarily web-based. However, many sites conveyed the same information and thus I deemed the information to be correct. Journals referenced in web-sites were followed up, if only to check that the information was indeed correct and consistent. A busy first semester meant the literature review slipped a little into the revision period but it was back on track come the next semester.

Creating the Magic Squares was completed about three weeks early so work on solving Magic Squares started early and finished around mid-March. The time saved was then spent on exceeding the minimum requirements by making code even more efficient and finding my own method to create a Magic Square. Computing the total solutions for the order 5 square was done concurrently with writing up this report at the end of March / beginning of April.
2. Background

2.1 What is a Magic Square?

A normal Magic Square, is a square of size n (n being a positive integer), with the integers 1..n² positioned in such a way that the sums of the numbers in each row, column and the two main diagonals are the same. The value of n is called the order of the square and the value the row/column/diagonal sum to is called the magic sum [1], constant [2], or number [3], and is given by:

\[ M(n) = \frac{1}{n} (1 + 2 + 3 + \ldots + n^2) \]  

Since the sum of n consecutive numbers is given by this known summation:

\[ \sum_{k=1}^{n} k = \frac{1}{2} n(n + 1) \]  

the sum of the first n² positive integers would therefore be:

\[ \sum_{k=1}^{n^2} k = \frac{1}{2} n^2(n^2 + 1) \]  

So substitute (3) into (1), and we get:

\[ M(n) = \frac{1}{n} \cdot \frac{1}{2} n^2(n^2 + 1) \]

\[ M(n) = \frac{1}{2} (n^3 + n) \]

Thus the magic sum of M(3) = 15, M(4) = 34, M(5) = 65, M(6) = 111 etc…

2.2 History

This order 3 square is the first known Magic Square and is called Loh-Shu [1,2] (or Lo-Shu [4,5]) and was found on a scroll of the river Loh in China. The scroll is supposedly devised by Fuh-Hi, the mythical founder of the Chinese civilisation, who lived between 2858-2738 BC [1,6].

This is Dürer’s Magic Square, a famous historical order 4 square. Engraved into his Melancholia engraving, Albrecht Dürer was a German painter, engraver and woodcut designer. His square has the numbers 15 and 14 adjacent to each other, depicting the year of creation, 1514 [1,2].
2.3 Properties of Magic Squares

The type of square we have so far referred to is called a pure or normal Magic Square. There are many more properties of Magic Squares, which I’ll briefly explain (the list below is not conclusive).

All examples, unless stated, can be found on Holger Danielsson’s website [1].

A square is **magic** if all rows, columns and diagonals sum to the same value [1].

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

A square is **semi-magic** if just the rows and columns sum to the same value.

The Latin Square to the left is semi-magic [2,3].

A square is **normal/pure** if it is magic and it sums to the magic sum value $\frac{1}{2}(n^2+n)$ using the numbers 1..$n^2$ [1].

A square is **symmetrical/associative/regular** if it is magic and symmetrically opposite cells also sum to $(n^2+1)$ [1,2,3].

E.g.

\[
\begin{array}{cccc}
3 & 16 & 9 & 22 \\
20 & 8 & 21 & 14 \\
7 & 25 & 13 & 19 \\
24 & 12 & 5 & 18 \\
11 & 4 & 17 & 10 \\
\end{array}
\]

The order 5 square has symmetry about the number 13 cell: $8+18 = 26$ and $22+4=26$

The order 4 square has symmetry about the centre point between cells 10,11,6 and 7:

$10+7 = 17$ and $9+8 = 17$

A square is **pandiagonal** (also called **perfect**, **diabolical**, **nasik**, **continuous**, **indian**, or **jaina**) if it is magic and the broken diagonals also sum to the magic sum [1,2,3].

E.g.

\[
\begin{array}{cccc}
18 & 22 & 10 & 14 \\
9 & 11 & 3 & 17 \\
2 & 20 & 24 & 16 \\
21 & 8 & 12 & 5 \\
15 & 4 & 16 & 23 \\
\end{array}
\]

Shaded broken diagonal: $2+11+10+23+19 = 65$

Dashed broken diagonal: $22+3+6+19+15 = 65$

A square is **self-complement/self-similar** if it is formed by taking a normal square of order n, and subtracting each number away from $(n^2+1)$ [1,3].

E.g.

\[
\begin{array}{cccc}
1 & 2 & 15 & 16 \\
12 & 14 & 3 & 5 \\
13 & 7 & 10 & 4 \\
8 & 11 & 6 & 9 \\
\end{array}
\]

Harvey Heinz shows on his website, that all symmetrical Magic Squares are also self-complement, as the complement is just the original square rotated by 180° [6].

A square is **ultramagic** if it is pandiagonal and self-complement [1].
A square is **most-perfect** if it contains the numbers 1..\(n^2\) such that:

1. all 2x2 sub-squares sum to \(2(n^2+1)\)
2. all pairs of distance \(1/2n\) along any diagonal sum to \((n^2+1)\)
3. it is a double-even Magic Square (i.e. order \(n=4m\) for integer \(m\))

It can be shown that all most-perfect squares are pandiagonal [1].

E.g. This double-even (\(m=1\)), order 4 square is most-perfect:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>15</th>
<th>4</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>10</td>
<td>5</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>3</td>
<td>16</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>6</td>
<td>9</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

1. all 2x2 sub-squares sum to 34 (including wrapping as shown here)
   
   \[1+15+8+10 = 34\quad\text{and}\quad13+12+2+7 = 34\]

2. all diagonal pairs of distance \(1/2n\) sum to 17
   
   \[8+9 = 17\quad\text{and}\quad3+14 = 17\]

An **inlaid** square is magic, and has magic sub-squares within [1].

E.g. This order 5 magic square has a magic sum of 65, and has an order 3 magic diamond (grey) of magic sum 39 within.

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>10</th>
<th>24</th>
<th>15</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>1</td>
<td>18</td>
<td>3</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>20</td>
<td>13</td>
<td>6</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>23</td>
<td>8</td>
<td>25</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>16</td>
<td>19</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A square is **concentric/bordered** if it is magic, and remains magic when its border is removed, provided the border numbers, \(x\), satisfies the following conditions…

\[1 \leq x \leq 2n-2\quad\text{and}\quad n^2-2n+3 \leq x \leq n^2\]

These two conditions ensure the numbers to be taken out are in the upper and lower bounds so the summations remain balanced [1].

E.g. Remove the (grey) border and the resulting square is still magic.

Values of the border lie between 1..10 and 27..36, so the upper and lower values are removed from this order 6 square.

<table>
<thead>
<tr>
<th></th>
<th>36</th>
<th>2</th>
<th>3</th>
<th>7</th>
<th>32</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>26</td>
<td>13</td>
<td>12</td>
<td>23</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>15</td>
<td>20</td>
<td>21</td>
<td>18</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>19</td>
<td>16</td>
<td>17</td>
<td>22</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>14</td>
<td>25</td>
<td>24</td>
<td>11</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>35</td>
<td>34</td>
<td>30</td>
<td>5</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
A square is *composite* if it is magic and is composed of other Magic Squares, which can’t be normalised (composed of consecutive integers starting from 1).

A composite square of order \( m \times n \) can be decomposed into \( m^2 \) subsquares, each of order \( n \). The minimal composite square must therefore be of order 9, with \( m=3 \) and \( n=3 \) [1].

E.g. A composite square made up of 9 subsquares, each of order 3.

<table>
<thead>
<tr>
<th>71</th>
<th>66</th>
<th>67</th>
<th>20</th>
<th>25</th>
<th>24</th>
<th>29</th>
<th>34</th>
<th>33</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>68</td>
<td>72</td>
<td>27</td>
<td>23</td>
<td>19</td>
<td>36</td>
<td>32</td>
<td>28</td>
</tr>
<tr>
<td>69</td>
<td>70</td>
<td>65</td>
<td>22</td>
<td>21</td>
<td>26</td>
<td>31</td>
<td>30</td>
<td>35</td>
</tr>
<tr>
<td>38</td>
<td>3</td>
<td>40</td>
<td>40</td>
<td>39</td>
<td>22</td>
<td>74</td>
<td>79</td>
<td>78</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>9</td>
<td>45</td>
<td>41</td>
<td>37</td>
<td>81</td>
<td>77</td>
<td>73</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>2</td>
<td>38</td>
<td>43</td>
<td>42</td>
<td>76</td>
<td>75</td>
<td>80</td>
</tr>
<tr>
<td>47</td>
<td>54</td>
<td>49</td>
<td>56</td>
<td>63</td>
<td>58</td>
<td>11</td>
<td>16</td>
<td>15</td>
</tr>
<tr>
<td>52</td>
<td>50</td>
<td>48</td>
<td>61</td>
<td>59</td>
<td>57</td>
<td>18</td>
<td>14</td>
<td>10</td>
</tr>
<tr>
<td>51</td>
<td>46</td>
<td>53</td>
<td>60</td>
<td>55</td>
<td>62</td>
<td>13</td>
<td>12</td>
<td>17</td>
</tr>
</tbody>
</table>

A square is *multimagic* if it belongs in the class of bimagic, trimagic, tetramagic, pentamagic etc… squares. Where a bimagic square is a Magic Square which stays magic if all elements are squared, a trimagic square is a Magic Square which stays magic if all elements are cubed, and so forth for tetramagic (raised to the power 4), pentamagic (raised to the power 5) etc… [1,2]

There is no magic 2x2 square, and a magic square can have multiple properties. Copious examples of Magic Squares can be found on Holger Danielsson and Harvey Heinz’s websites [1, 6].

### 2.4 Scope of Project

Due to all the various properties, there is no single algorithm to generate all types of Magic Squares [1,5,6]. Moreover, for each property, there is no single algorithm that generates a square for all orders [1]. For instance, algorithms to create normal Magic Squares create just odd or even order squares but not both. Algorithms for pandiagonal squares create either prime, odd or even orders and there are copious ways to create inlaid squares due to the fairly unrestrictive nature of them (the magic sub-square within doesn’t always have to be a magic diamond). Thus, there exists several algorithms to generate Magic Squares.

The scope of this project will not allow me to cover them all, and so this project is solely focused on normal Magic Squares. This is the type of square I am most familiar with and will allow for extensions if time permits, or for future projects. Also, Heinz [6] and Trump [7] have done extensive work on non-normal Magic Squares anyway.

NB: From now on, a normal Magic Square shall be referred to simply as a Magic Square.
2.5 Number of Solutions and Transformations

There are a total of eight solutions to the order 3 Magic Square, which are…

However, take a closer look and you’d see that each square is just a transformation of the first:

The second, third and fourth square are 90, 180 and 270 degrees, clockwise rotations about the number 5 respectively. All the other squares are reflections along the shaded axes shown.

So, not counting symmetric squares, the 3x3 normal Magic Square is unique. The Lo-Shu square is the only solution [1,7].

The following table lists the number of total and unique solutions for the normal Magic Square:-

<table>
<thead>
<tr>
<th>ORDER</th>
<th>UNIQUE</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>880</td>
<td>7040</td>
</tr>
<tr>
<td>5</td>
<td>275 305 224</td>
<td>2 202 441 792</td>
</tr>
<tr>
<td>6</td>
<td>1.775399x10⁷</td>
<td>???</td>
</tr>
<tr>
<td>7</td>
<td>3.79809x10⁹</td>
<td>???</td>
</tr>
<tr>
<td>8</td>
<td>5.2210x10²⁵</td>
<td>???</td>
</tr>
<tr>
<td>9</td>
<td>7.8448x10⁷⁵</td>
<td>???</td>
</tr>
<tr>
<td>10</td>
<td>2.4160x10¹⁰³</td>
<td>???</td>
</tr>
</tbody>
</table>

Solutions for the order 3 square were obtained from [1,2,7].

The unique solutions for the order 4 square were first calculated by Bernard Frénicle de Bessy in 1693 [8]. The total solutions were obtained from Holger Danielsson [1].

The unique solutions for the order 5 square were obtained from [1,2,7]. I have not found the total number of solutions listed anywhere, but since each unique solution can be transformed into a total of eight solutions, I predict the total solutions to be that shown. Thomas Rasch [9] also seems to state this number but how he gets this number is unclear. For the purpose of this project, I assumed that the total number of solutions for the order 5 square was unconfirmed.

The unique solutions for the higher order are estimated by Pinn & Wieczerkowski using Monte Carlo simulation [10] and are confirmed by Trump [7].
3. EXISTING ALGORITHMS

No single algorithm exists to create a Magic Square of general order, instead, existing algorithms are
classed for odd order squares, double-even order squares (n=4m, m>0) and single-even order squares
(n=4m+2, m>0).

Holger Danielsson’s website [1] gives you the chance to create Magic Squares using different
algorithms; unfortunately, he doesn’t give their algorithms (again, the list below is not conclusive).

3.1 Odd Order Magic Squares

Siamese (also called de la Loubères) method – place the 1 in the top centre cell, now incrementally
place the numbers diagonally top right of it. If you fall off the edge then you wrap round (left to right,
and top to bottom) and if you encounter an occupied cell then the number is entered into the cell
directly below the previously entered cell [2].

Bachet de Mézeriac method – fill the square with the numbers in sequence. Rotate the numbers in the
square by 45 degrees anticlockwise such that a pyramid shape is formed on each edge of the square.
Now wrap any number not in the square around into the empty cell [11].
3. Existing Algorithms

3.2 Double-Even Order (n=4m) Magic Squares

**Agrippa: Diagonal** method – fill the square with the numbers in sequence and draw crosses through each 4x4 sub-square. Now replace each crossed square, $x$, by its complement $(n^2+1)-x$ [1,2].

```
1  2  3  4  5  6  7  8  64  2  3  61  8  6  7  8
9  10 11 12 13 14 15 16  9  55 54 12 13 14 15 16
17 18 19 20 21 22 23 24 17 47 46 20 21 22 23 24
25 26 27 28 29 30 31 32 25 26 27 28 29 30 31 32
33 34 35 36 37 38 39 40 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 41 42 43 44 45 46 47 48
49 50 51 52 53 54 55 56 49 50 51 52 53 54 55 56
57 58 59 60 61 62 63 64 57 58 59 60 61 62 63 64
```

**Agrippa: Inverse Diagonal** method – as the diagonal method above, but keep the crossed squares and replace the non-crossed squares, $x$, by their complements $(n^2+1)-x$ [1,12].

**Filling 9 Blocks** method – like Agrippa’s inverse diagonal method, but instead of drawing crosses through each 4x4 sub-square, draw it through the whole square, in a 1:2:1 ratio (see example).

Keep the crossed out squares the same, and replace the non-crossed squares, $x$, by their complements $(n^2+1)-x$ [1,13].

```
1   2   3   4   5   6   7   8  1   2  62  61  60  59  7   8
9  10  11  12  13  14  15  16  9  10  54  53  52  51  15  16
17 18  19  20  21  22  23  24 17 18  47  46  20  21  22  42 41
25 26  27  28  29  30  31  32 25 26  40  27  28  29  30  34 33
33 34  35  36  37  38  39  40 33 34  32  35  36  37  38  26 25
41 42  43  44  45  46  47  48 41 42  24  43  44  45  46  18 17
49 50  51  52  53  54  55  56 49 50  49  51  13  12  11  55 56
57 58  59  60  61  62  63  64 57 58  57  58  6  5  4  3  63 64
```

**Inverse Filling 9 Blocks** method – although I have not found such a method, but the ‘inverse’ (replace the crossed squares instead) of the Filling 9 Blocks also produces a normal Magic Square. Applying the method by hand to the order 8, 12, 16 and 20 square gives normal Magic Squares.
3.3 Single-Even Order (n=4m+2, m>0) Magic Squares

Lux method – create a matrix of (m+1) rows of L’s, one row of U’s and (m-1) rows of X’s, each row of length half n. Interchange the middle U with the L above it. Now, starting from the top centre L, use the Siamese method to get to each letter, and fill the set of four squares around each letter according to its shape (shown below) [2].

```
4   1
  2   3
  1   4
  2   3
```

```
68  65  96  93  4   1  32  29  60  57
66  67  94  95  2   3  30  31  58  59
92  89  20  17  28  25  56  53  64  61
90  91  18  19  26  27  54  55  62  63
16  13  24  21  49  52  80  77  88  85
14  15  22  23  50  51  78  79  86  87
37  40  45  48  76  73  81  84  9   12
38  39  46  47  74  75  82  83  10  11
41  44  69  72  97 100  5   8  33  36
43  42  71  70  99  98  7   6  35  34
```

Figure 1: Example taken from Eric Weisstein’s MathWorld website [2].

Strachney method – divide the square into four smaller sub-squares, fill each sub-square with sequential numbers using the Siamese method, filling the top left square first, moving onto the bottom right sub-square, then filling the top right sub-square before finishing with the bottom left sub-square. Now perform the number swaps, which is best explained visually (see below) [1,13].

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```

```
8   1   6
3   5   7
4   9   2
```
3. Existing Algorithms

...then swap the dark shaded numbers with the light shaded numbers according to the order of the square. Here are the swapping masks for orders 6, 10 and 14.

<table>
<thead>
<tr>
<th>Order 6 swapping mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 1 6 26 19 24</td>
</tr>
<tr>
<td>3 5 7 21 23 25</td>
</tr>
<tr>
<td>4 9 2 22 27 20</td>
</tr>
<tr>
<td>35 1 6 26 19 24</td>
</tr>
<tr>
<td>3 5 7 21 23 25</td>
</tr>
<tr>
<td>4 9 2 22 27 20</td>
</tr>
<tr>
<td>35 28 33 17 10 15</td>
</tr>
<tr>
<td>30 32 34 12 14 16</td>
</tr>
<tr>
<td>31 36 29 13 18 11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Order 10 mask to the left, and an order 14 mask below.</th>
</tr>
</thead>
<tbody>
<tr>
<td>17 24 1 8 15 67 74 51 58 65</td>
</tr>
<tr>
<td>23 5 7 14 16 73 55 57 64 66</td>
</tr>
<tr>
<td>4 6 13 20 22 54 56 63 70 72</td>
</tr>
<tr>
<td>10 12 19 21 3 60 62 69 71 53</td>
</tr>
<tr>
<td>11 18 25 2 9 61 68 75 52 59</td>
</tr>
<tr>
<td>32 99 76 83 90 42 49 26 33 40</td>
</tr>
<tr>
<td>98 80 82 89 91 48 30 32 39 41</td>
</tr>
<tr>
<td>79 81 88 95 97 20 31 38 45 47</td>
</tr>
<tr>
<td>85 87 94 96 78 35 37 44 46 28</td>
</tr>
<tr>
<td>86 93 100 77 84 36 43 50 27 34</td>
</tr>
</tbody>
</table>

Why all these methods construct Magic Squares is unknown to me. I can’t find the reasoning behind any of these methods, the Lux and Strachney methods in particular are very peculiar and it would be interesting to find out why they work. However, I have noticed that the ‘inverse’ Agrippa Diagonal and Filling 9 Blocks methods produce Magic Squares because these squares are the ‘normal’ created squares but rotated by 180 degrees. Symmetry seems to play an important role in creating double-even order squares and so knowing which cells to complement in order to balance the sums is perhaps why these methods create Magic Squares.
4. Creating a Magic Square

4 CREATING A MAGIC SQUARE

To meet the minimum requirement of creating a Magic Square, I aim to produce a single square of any given size that is magic. There are several problems we could deal with:

- Create a Magic Square of any order.
- Fill in a partially filled square such that it is a Magic Square.
- Come up with a single method to create a Magic Square of any order (existing methods deal with either odd, double-even or single-even squares).

I’ve decided to deal with the first problem by simply implementing the existing methods. Filling a partially filled square and coming up with a single method are discussed later where they are more appropriately explained.

4.1 Analysis

We have already encountered existing algorithms but doing these by hand, especially on a large square, gets tedious and perhaps cumbersome. Not only will a program aid us, but will create a square with speed and accuracy. Holger Danielsson’s [1] website lets you create Magic Squares but only to order 25, so it would also be nice to create bigger squares than that.

The programming languages that I am comfortable with are Pascal, C++ and Java. Considering it would be useful to print out squares into a file (a big square would not fit on the terminal), and the need of a suitable data structure to represent the square such as (2D) arrays, then any of the listed languages would be sufficient. I’m most familiar with C++ and therefore is my language of choice. In today’s society, the computing industry favours OOP based languages like C++ and Java, so it would also be sensible to get more experience in using such languages.

I plan to implement all the existing algorithms, this then gives the user the chance to select a particular method and to see different squares of the same order. User interaction will be implemented via menus. We shouldn’t have to worry about HCI issues as all they would have to do is enter the order of square and select a method, which can easily be done at the prompt.

4.2 Design

I have decided to use 2D arrays/matrix to represent the Magic Square, this seems logical to me and is the only reasonable way I can think of to represent the square. It’s also easily implemented and it is this 2D-array structure that the following designs are based around.
Note that in the following, wrapping around the square is automatically assumed (particularly in the pseudo code), and a broken diagonal always runs from bottom left to top right through the square (please refer back to the Lozenge method for an example).

### 4.2.1 Siamese Method (odd order)

Each square has order n (broken) diagonals running through the square. Starting at the top centre cell, numbers increment diagonally. At the \((n-1)\)th increment, you move one cell down and increment along this broken diagonal. Do this for each broken diagonal.

In pseudo code, the algorithm would take the form:

```
start at the top centre cell and a start value of 1;
for each broken diagonal do
    for each cell in diagonal do
        assign value to cell;
        increment value;
        move onto next diagonal cell (if not \((n-1)\)th increment);
    endfor;
    move one cell down onto next broken diagonal;
endfor;
```

### 4.2.2 Lozenge Method (odd order)

Starting from the first odd number in each broken diagonal, each diagonal is either made up of \(\frac{1}{2}(n+1)\) incrementing odd numbers followed by \(\frac{1}{2}(n-1)\) incrementing even numbers (situation 1) or \(\frac{1}{2}(n-1)\) incrementing odd numbers followed by \(\frac{1}{2}(n+1)\) incrementing even numbers (situation 2). These two situations alternate as we move through the broken diagonals. The differing situations mean that we would need flags to keep track of which situation we are on, and also counters to keep track of which odd or even number we are on.

Also note that if situation (1) arises, then the next diagonal starts at the cell to the right of where the current diagonal started, and if situation (2) arises then the next diagonal starts at the cell one below from where the current diagonal started.

So in pseudo code:

```
start at in centre cell of first column;
set odd=1, even=2, situation=(1);
for each broken diagonal do
    if situation is (1) then
        move through broken diagonal (via for loops);
        assign \(\frac{1}{2}(n+1)\) incrementing odd numbers (odd=odd+2);
        assign \(\frac{1}{2}(n-1)\) incrementing even numbers (even=even+2);
    end if
```
else if situation is (2) then
    move through broken diagonal (via for loops);
    assign \(\frac{1}{2}(n-1)\) incrementing odd numbers (odd=odd+2);
    assign \(\frac{1}{2}(n+1)\) incrementing even numbers (even=even+2);

    //move onto next broken diagonal
    if situation is (1) then
        move one cell right onto next broken diagonal;
    else if situation is (2) then
        move one cell down onto next broken diagonal;
    alternate the situation;
endfor;

4.2.3 Bachet de Mézeriac Method (odd order)

On initial inspection, performing the rotation could be very messy, until I noticed how the numbers increment along each broken diagonal in the resulting square. In fact, starting from the cell right of centre, you increment along the broken diagonal, then to move to the next diagonal you move two cells right (wrapping if necessary) and increment along this diagonal. Do this until each broken diagonal is filled. The design is similar to the Siamese method.

In pseudo code:

    start at the cell right of centre and a start value of 1;
    for each broken diagonal do
        for each cell in diagonal do
            assign value to cell;
            increment value;
            move onto next diagonal cell (if not order n-1 increment);
        endfor;
        move two cells right onto next broken diagonal;
    endfor;

4.2.4 Agrippa Diagonal & Filling 9 Blocks Methods (double-even order)

For all the double-even order methods, numbers are entered into the square sequentially (which is easily done via two nested for loops) but if the cell is a ‘complement’ cell, then the complement number is assigned instead. All that differs between all the double-even order methods is the condition it has to meet in order to be a ‘complement’ cell.

For the Agrippa Diagonal method, a cell is complement if the row AND column are both multiples of 4 OR (4+1). OR if row AND column are both NOT multiples of 4 OR (4+1), (I’ve used capitals here to stress these are logical expressions and will be implemented as such).
For clarification, below is a table and the cells in the square that each entry corresponds to. The shaded cells in the square are the ‘complement’ cells.

<table>
<thead>
<tr>
<th>row,col</th>
<th>row is 4x?</th>
<th>row is 4x+1?</th>
<th>col is 4x?</th>
<th>col is 4x+1?</th>
<th>complement?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1,1</td>
<td>no</td>
<td>yes (1=0)</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>2</td>
<td>1,2</td>
<td>no</td>
<td>yes (1=0)</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>3</td>
<td>1,4</td>
<td>no</td>
<td>yes (1=0)</td>
<td>yes (1=1)</td>
<td>no</td>
</tr>
<tr>
<td>4</td>
<td>1,5</td>
<td>no</td>
<td>yes (1=0)</td>
<td>yes (1=1)</td>
<td>yes</td>
</tr>
<tr>
<td>5</td>
<td>2,3</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>6</td>
<td>2,4</td>
<td>no</td>
<td>no</td>
<td>yes (1=1)</td>
<td>no</td>
</tr>
<tr>
<td>7</td>
<td>2,5</td>
<td>no</td>
<td>no</td>
<td>yes (1=1)</td>
<td>no</td>
</tr>
<tr>
<td>8</td>
<td>2,6</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

For the *Agrippa Inverse Diagonal* method, a cell is complement if the row is a multiple of 4 OR (4+1) AND column is NOT a multiple of 4 OR (4+1). OR if row is NOT a multiple of 4 OR (4+1) AND column is a multiple of 4 OR (4+1). I.e. it is the opposite of the Agrippa Diagonal method.

For the *Filling 9 Blocks* method, a cell is complement if the row does NOT lie between m AND (order-m), AND column does lie between m AND (order-m). OR if row does lie between m AND (order-m), AND column does NOT lie between m AND (order-m), where m= ¼order (as n=4m for double-even order) and all ‘between’s are exclusive. In the table below, order=8 so m=2.

<table>
<thead>
<tr>
<th>row,col</th>
<th>m &lt; row &lt; (order-m)?</th>
<th>m &lt; col &lt; (order-m)?</th>
<th>complement?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(2,2)</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>2</td>
<td>(2,3)</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>3</td>
<td>(2,4)</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>4</td>
<td>(2,5)</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>5</td>
<td>(2,6)</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>6</td>
<td>(2,7)</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>7</td>
<td>(2,8)</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>8</td>
<td>(2,9)</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

For the *Inverse Filling 9 Blocks* method, a cell is complement if both the row AND column do NOT lie between m AND (order-m). OR if both the row AND column does lie between m AND (order-m); where m= ¼order (as n=4m for double-even order) and all ‘between’s are exclusive. I.e. it is the opposite of the Filling 9 Blocks method.

The pseudo code for all these methods is:

```
start at top left corner cell and a start value of 1;
for each row do
    for each column do
        assign value to cell and increment value;
        if cell is complement then assign complement value to cell;
    endfor;
endfor;
```
4. Creating a Magic Square

4.2.5 Lux Method (single-even order)

Creating the Lux matrix and then running the Siamese method on it to fill in the four corresponding cells around it, means we traverse the Lux matrix twice. Instead, we run the Siamese method on the Lux matrix, work out what letter it would be and fill in the four corresponding squares accordingly, thus traversing the matrix just once.

Running the Siamese method on the Lux matrix is via two nested for loops just like in the Siamese method. In determining what letter the cell is: if row is between 1 AND (m+1) inclusive then it’s L, if row is the (m+2)th row then it’s U, and rows after the (m+2)th row are X; where \( m = \frac{1}{4}(n-2) \) (since \( n=4m+2 \) for single-even order squares). The exceptions to this are if the cell is the centre cell then it’s a U (and not L), and if the cell is one below centre then it’s an L (and not U).

Finally the four squares that correspond to each Lux cell are: if \((x,y)\) is the Lux cell then \((2x-1,2y-1)\), \((2x,2y-1)\), \((2x,2y)\) and \((2x-1,2y)\) (going top-left, top-right, bottom-right, bottom-left cell) are the corresponding cells in the square. We then assign each cell in order as dictated by its Lux letter.

In pseudo code then:

```pseudo
start at top centre cell of Lux matrix and a start value of 1;
m = \frac{1}{4}(n-2);
for each broken diagonal in Lux matrix do
  for each cell in broken diagonal do
    if centre cell then assign cells in U order
      (top-left, bottom-left, bottom-right, top-right);
    else if cell is one below centre cell then assign cells in L order
      (top-right, bottom-left, bottom-right, top-left);
    else if L row then assign square cells in L order
      (top-right, bottom-left, bottom-right, top-left);
    else if U row then assign square cells in U order
      (top-left, bottom-left, bottom-right, top-right);
    else if X row then assign cells in X order
      (top-left, bottom-right, bottom-left, top-right);
    move onto next diagonal cell (if not Lux order n-1 increment);
  endfor;
move down one row onto next broken diagonal;
endfor;
```
4. Creating a Magic Square

4.2.6 Strachney Method (single-even order)

Like the Lux method, if we perform the swaps whilst creating the square, we save a traversal.

Dealing with just the top-left sub-square throughout: applying the Siamese method lets us fill in the rest of the square too, as the top centre cell in the top-left sub-square is the same as the top centre cell in the bottom-right sub-square but its value differs by \((\frac{1}{2}n)^2\). Similarly, the top centre cell in the top-right sub-square has a value differing by \(2(\frac{1}{2}n)^2\) and the bottom-left sub-square differs by \(3(\frac{1}{2}n)^2\).

In swapping, a cell needs to be swapped with the cell half n below if it’s the centre cell, or if column is between 1 and m inclusive, except the centre cell of column 1 (remember \(n=4m+2\)). If column is between \((\frac{1}{2}n-(m-2))\) and \(\frac{1}{2}n\) inclusive then the \((row, (column + \frac{1}{2}n))\) cell needs to be swapped with the \(((row+\frac{1}{2}n), (column + \frac{1}{2}n))\) cell.

Please refer to the table for clarification on the swapping: order=10 so \(m=2\), and the lightly shaded squares get swapped with the dark shaded squares below them.

<table>
<thead>
<tr>
<th>((row, col))</th>
<th>(1 \leq col \leq m)</th>
<th>((\frac{1}{2}n-(m-2)) \leq col \leq \frac{1}{2}n)</th>
<th>swap?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. (3,1)</td>
<td>yes</td>
<td>no</td>
<td>no, as centre cell of col 1</td>
</tr>
<tr>
<td>2. (3,2)</td>
<td>yes</td>
<td>no</td>
<td>((3,2)) with ((3,2+\frac{1}{2}n))</td>
</tr>
<tr>
<td>3. (3,3)</td>
<td>no</td>
<td>no</td>
<td>yes, as its centre cell</td>
</tr>
<tr>
<td>4. (3,4)</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>5. (3,5)</td>
<td>no</td>
<td>yes</td>
<td>((3,3+\frac{1}{2}n)) with ((3+\frac{1}{2}n, 3+\frac{1}{2}n))</td>
</tr>
</tbody>
</table>

The pseudo code for this is:

```
start in top centre cell of top-left sub-square and start value of 1;
\(m = \frac{1}{4}(n-2)\);
for each broken diagonal in sub-square do
    for each cell in broken diagonal do
        assign value to cell;
        assign \((value + (\frac{1}{2}n)^2)\) to bottom-right sub-square cell;
        assign \((value + 2(\frac{1}{2}n)^2)\) to top-right sub-square cell;
        assign \((value + 3(\frac{1}{2}n)^2)\) to bottom-left sub-square cell;
        increment value;
        if centre cell OR \(1 \leq col \leq m\) -(except column 1)) then
            swap cell with \(\frac{1}{2}n\) cell below it
        else if \((\frac{1}{2}n-(m-2)) \leq column \leq \frac{1}{2}n\) then
            swap \((row, (column + \frac{1}{2}n))\) with \((row+\frac{1}{2}n, col+\frac{1}{2}n)\) cell;
        move onto next diagonal cell (if not \(\frac{1}{2}n-1\) increment);
    endfor;
move one cell down onto next broken diagonal;
endfor;
```
4. Creating a Magic Square

4.3 Implementation & Testing

The designs were implemented in C++, remember that array indexing start at zero so (0,0) would index cell (1,1). MAX_ORDER can be changed to be the value of the largest square you can compute and that your hardware will allow.

The program asks the user to enter the order of the square, they then select the method of construction of which the choices available is determined by the order given. The user enters their choices via simple input at the prompt and the only validation is that order lies between 3 and MAX_ORDER. The square is created by the method specified and written to file, CREATE. Getting the order of the square from the user could have been done via command line arguments, of which the user could have also specified the file to write the square. These however just make the program ‘nicer’ to use and I was more concerned with implementing the algorithms correctly than worrying about user interaction (altering it to use command line arguments wont be difficult anyway should one want to do it).

Please see create.cc for implementation details and screenshots are given in appendix C.

A little function (isMagic) was written to test that the squares produced were indeed magic by checking that the rows, columns, and diagonals sum to the magic sum. This is run as a check whenever we created a square. Again, please see create.cc for details.

The created squares were also checked with Holger Danielsson’s examples [1] and with the examples supplied in the existing methods chapter; the squares were the same. Please see appendix D for the test data, which shows that the algorithms were implemented correctly and that any order square can be created.
4. Creating a Magic Square

4.4 Evaluation

The program was evaluated against the following criteria: -

- **Correctness of results**
  As testing has shown, I am confident that I have implemented the algorithms correctly and the program does indeed create a Magic Square of any size (up to MAX_ORDER, which is determined by hardware capabilities. The code does not compile if MAX_ORDER is too large).

- **Speed and efficiency**
  If every square needs to have order² numbers entered, then this will take $O(n^2)$ operations. Every method has two nested for loops of $n$ iterations and should therefore run in $O(n^2)$, which is indeed what we get. Please see appendix D for run-time data (this data shows how long it takes to compute the square and not write it to file as well), which was obtained using the time class of C++. I conclude that the algorithms are implemented efficiently as they take the required time to enter order² numbers into a square. This code easily produces a square of order 10,000 in a matter of seconds.

We have therefore met our aim of developing code to create a Magic Square of any order, and exceeded it by developing efficient code.
5 SOLVING A MAGIC SQUARE

To meet the minimum requirement of solving a Magic Square, I aim to find all solutions of a square of given size. Total solutions to the order 3 and 4 square are known and will be confirmed by me. The total solutions to squares bigger than 5x5 is unknown and is therefore worth investigating. As stated in the background (section 2.5), I predicted the total number of solutions for the order 5 square to be 2,202,441,792, which is indeed what my program will calculate. I plan to compute all the solutions (including transformations) because you have to compute all solutions in order to check for any similar squares anyway. Should you want to calculate the unique solutions, you can compute all solutions to a file, and then write a program to read the file and work them out later rather than working them out during computation. It could be quicker too because working them out during computation mean we would have to store the squares (making memory a problem) and search through them each time we find a square to check it’s unique. However, bear in mind that a file listing 2.2 billion solutions (in the order 5 case) is not going to be small and will be an issue if memory is a problem, in this situation it would be better to calculate the unique solutions during computation. Due to time, calculating unique solutions will not be explored.

5.1 Brute-Force Method

5.1.1 Analysis

Creating all possible squares and then checking if they are magic is a sure-fire (albeit slow) way of finding all the solutions of a square. We can do this by using nested for loops for each cell of the square and iterate through all the numbers. However, it can be speeded up in the following ways:

1. A number cannot appear twice in the square. If this happens, we can move onto the next iteration.
2. The last number of any sum (row/column/diagonal) must be the number that makes it sum to the magic sum. Therefore, knowing the penultimate number of any sum means you can calculate the last number in that sum, if that number already appears in the square or is not in the range 1..n² then we can move onto the next iteration.

A problem to resolve is the order in which the conditions are evaluated because it can make a difference to the overall computation time. This however, can be resolved on a ‘trial and error’ type approach during implementation and evaluation in order to find the shortest run time. Skipping iterations is the key idea here. Consider the order 3 square, there are 9! combination of squares but only 8 solutions – a very small percentage (0.002%) are valid squares. The order 4 square has a percentage of $3.4\times 10^8\%$. We need to cut down on the number of iterations and avoid computing invalid squares.
5. Solving a Magic Square

We could have generated all squares using recursive calls, which will give shorter code but decided not to as it’s clearer (to me) to keep track of which cell of the square we are at when using for loops (in conjunction with variables a to i – see below).

5.1.2 Design

A square can be generated using nested for loops for each cell in the square (bar the last cell in any sum as this is worked out from the previous cells). When we generate a square we can just output it to terminal or file, there’s no need for us to store the squares so we can save memory. To make output more neat (instead of printing lots of squares), we can display the numbers in cell order.

For example, the square to the right can be outputted all in a single line as: a b c d e f g h i.

Variables a to i can be used to represent the cells in the square and temporary store the numbers until they are outputted.

We also need to keep track of which numbers are in the square so that we can perform our conditions stated in analysis, which can be done by using an array of bools, where true means the number (index of array) is in the square.

The pseudo code for the order 3 brute-force method is:

```plaintext
initialise boolean (visit) array to be false for all numbers;
set sol = 0 (no solutions found yet);
set magicSum = \( \frac{1}{2}(n+n^3) \);

for (a = 1..n^2) do
    visit[a] = true;
    for (b = 1..n^2) do
        if visit[b] is true then continue onto next iteration of b;
        c = magicSum-a-b;
        if (visit[c] is true OR c>n^2 OR c<1 OR c equals b) then
            continue onto next iteration of b;
        visit[b] = true; visit[c] = true;

    for (d = 1..n^2) do
        if visit[d] is true then continue onto next iteration of d;
        g = magicSum-a-d;
        if (visit[g] is true OR g>n^2 OR g<1 OR g equals d) then
            continue onto next iteration of d;
        e = magicSum-g-c;
        if (visit[e] is true OR e>n^2 OR e<1 OR e equals g or d) then
            continue onto next iteration of d;
```

21
5. Solving a Magic Square

\[ f = \text{magicSum-d-e}; \]
\[ \text{if } (\text{visit}[f] \text{ is true OR } f > n^2 \text{ OR } f < 1 \text{ OR } f \text{ equals e, g or d}) \text{ then} \]
\[ \text{continue onto next iteration of } d; \]
\[ h = \text{magicSum-b-e}; \]
\[ \text{if } (\text{visit}[h] \text{ is true OR } h > n^2 \text{ OR } h < 1 \text{ OR } h \text{ equals f, e, g or d}) \text{ then} \]
\[ \text{continue onto next iteration of } d; \]
\[ i = \text{magicSum-c-f}; \]
\[ \text{if } (\text{visit}[i] \text{ is true OR } i > n^2 \text{ OR } i < 1 \text{ OR } i \text{ equals f, h, e, g or d} \]
\[ \text{OR } g + h + i \text{ or } a + e + i \text{ not equals 15}) \text{ then} \]
\[ \text{continue onto next iteration of } d; \]

otherwise, we have found a solution:
\[ \text{increment sol and print out a b c d e f g h i;} \]
\[ \text{visit}[d] = \text{false}; \]
\[ \text{endfor (d);} \]
\[ \text{visit}[c] = \text{false}; \]
\[ \text{visit}[b] = \text{false}; \]
\[ \text{endfor (b);} \]
\[ \text{visit}[a] = \text{false}; \]
\[ \text{endfor (a);} \]

The brute-force method for higher orders would be the same but just with more nested for loops. So, for the order 3 square, the cells get filled up in this order: a, b, c, d, g, e, f, h, i and should run in about \(O(n^6)\) - but quicker as we skip iterations.

5.1.3 Implementation & Evaluation

The brute-force method was implemented for orders 3, 4 and 5 in brute3.cc, brute4.cc and brute5.cc respectively. Solutions for the order 3 and 5 square were outputted to the terminal and all 7040 solutions to the order 4 square were written to file (BRUTE4).

The brute-force method was not implemented for higher orders as it was already too slow for the order 5. Screenshots for this program is given in appendix C.

The program was evaluated against the following criteria:

- Correctness of results

Results for the order 3 were checked by eye with the squares given in the background (section 2.5), the squares are the same.

Results for the order 4 were checked with the results computed by Harvey Heinz [6] via file comparison (and pre-computed ordering) and stated that there were no differences in the files – the squares are all the same.
5. Solving a Magic Square

(check4.cc was written to list all eight transformations of each square given by Heinz as only the unique solutions are given, and to order the solutions ready for file comparison. check4.cc also deals with a file generated by a sets-of-sums method which is explained later).

- **Speed and efficiency**

All 8 solutions of the order 3 square were calculated and outputted to the terminal in no time at all.

All 7040 solutions of the order 4 square (on my 1133MHz machine) were calculated and outputted to file in a total of 4.2 seconds. 3.6 seconds to compute without writing to file (speed to write to file is determined by hardware).

For the order 5 square, squares were calculated and outputted to the terminal but was very slow. It produces its first square after about two minutes and calculated the first 8000 solutions in half an hour. Assume there are 2.2 billion solutions, at this rate it would take 15.7 years to compute!

We have therefore met the minimum requirement of developing code to find all solutions of a square (for the order 3 and 4 square anyway), and it is reasonably efficient as it does this within a few seconds at most.

As predicted, the brute force method is a guaranteed way to find all solutions but is indeed very slow (even for order 5). We need to find a faster method for the order 5 square as the brute force method has fifteen nested for loops of $n^2$ iterations and so runs in about $O(n^{30})$.

Due to time constraints, the rest of this chapter will only deal with the order 5 square.

5.2 Order 5 Square (sets-of-sums method)

Let me describe to you a method that Juergen Koeller [3] uses to compute all the order 3 squares. The magic sum for order 3 is 15, the following three digits (between 1 to 9) can be summed to the magic sum…

1. $1+5+9$, 2. $1+6+8$, 3. $2+4+9$, 4. $2+5+8$, 5. $2+6+7$, 6. $3+4+8$, 7. $3+5+7$, and 8. $4+5+6$.

A 3x3 square has eight sums in total, so which of the above sums fit where? The number 5 appears four times in the sums (1,4,7 and 8) and must therefore be in the centre cell as this cell is in four sums (middle row, middle column and both diagonals). Each even number appear three times in the sums so they must lie on the corners (the sums being both edges and the diagonal). This leaves the odd numbers, which appear twice in the sums to sit in the middle of the edges.

By inspection we can see that sums 4 and 8 would therefore be the diagonals and sums 1 and 7 would be the centre row and column. Swapping the sums round for each sum gives us a total of eight valid solutions – the eight solutions of the order 3 square as given previously in background (section 2.5).
5. Solving a Magic Square

5.2.1 Analysis
The brute-force method is slow because for each cell in the square, it has to iterate through all the numbers. If we know for example, that corners have to be even, then that already cuts down half the iterations, and if we know what number a cell is (like number 5 in order 3) then you miss out an entire for loop. So, to speed up the brute-force for order 5, I aim to cut down on the number of iterations we make.

If I can find four sums with unique numbers and one common number, then these sums can be the four sums going through the centre with the common number being the centre cell. Then each cell along the diagonals, centre row and column has to be a number in the sum, which means that these cells only has to iterate through four different numbers (as we know the centre cell has to be the common number). All other cells would have to iterate through all the possible numbers as there are no restrictions I can place on them (in fact, it’s just through all the numbers that are not in the four sums, which is only eight numbers). The final way to speed up the process is that the last cell of any row/column/diagonal has to be the number to make it sum to the magic sum and no number can appear twice in the square (these were employed previously anyway).

The set of four sums is the key to all this and so I call this method the sets-of-sums method.

5.2.2 Design
The main problem is to find the sums. Finding all five-tuples (via nested for loops) and then checking if it sums to the magic sum is yet another brute-force method problem running in \(O(n^{10})\) (five nested for loops doing \(n^2\) iterations) and must be speeded up.

Like before, the last number must make it sum to the magic sum and that no number can appear twice can be adopted here. If we say that each number in the tuple must be strictly decreasing then this avoids repetition (i.e. \(5,4,3,2,1\) and \(5,4,3,1,2\) are the same sums but we accept the first as this is in order).

Why have it in decreasing order (opposed to increasing)? Starting with the biggest number and working down (calculating the last number from the magic sum), we generate the following:

\[
\{25,24,23,22,-29\}, \{25,24,23,21,-28\}, \{25,24,23,20,-27\} \quad \ldots \quad \{25,24,23,1,-8\}, \{25,24,22,21,-27\}, \\
\{25,24,22,20,-26\} \quad \ldots \quad \{25,24,22,1,-7\}, \{25,24,21,20,-25\} \quad \ldots \quad \{25,24,14,13,-11\} \quad \ldots \quad \{25,24,14,2,0\}, \\
\{25,24,14,1,1\}, \{25,24,13,12,-9\} \quad \ldots \quad \{25,24,13,3,0\}, \{25,24,13,2,1\}, \{25,24,12,11,-7\} \quad \ldots \quad \{25,24,12,5,-1\}, \{25,24,12,4,0\}, \{25,24,12,3,1\}, \{25,24,12,2,2\}, \{25,24,11,10,-5\} \quad \ldots \quad \{25,24,11,6,-1\}, \{25,24,11,5,0\}, \{25,24,11,4,1\}, \{25,24,11,3,2\}, \{25,24,11,2,3\}, \{25,24,10,9,-3\} \quad \ldots \quad \{25,24,7,6,3\}, \{25,24,7,5,4\}, \{25,24,7,4,5\}, \{25,24,6,5,5\}, \{25,24,5,4,9\}, \{25,23,22,21,-26\} \
\]

The underlined tuples are the valid ones and the bold one is the first valid one we find; we calculate a lot of tuples and only a very small amount are valid. In fact, whenever the last number is negative, the next valid tuple is the one where the negative number (minus 1) is added to the fourth number. I.e.
5. Solving a Magic Square

{a,b,c,d,-e} is negative, so the next valid tuple would be {a,b,c,d-e-1,1} (take the shaded tuples for example). The decreasing order means we generate those negative numbers and so we can speed up the process further now. The tuples we generate now would be:

{25,24,23,22,-29}, {25,24,23,-8,1}, {25,24,22,21,-27}, [25,24,22,-7,1], {25,24,21,20,-25} …
{25,24,14,13,-11}, {25,24,14,1,1}, {25,24,13,12,-9}, [25,24,13,2,1], {25,24,13,1,2}, {25,24,12,11,-7}, [25,24,12,3,1], {25,24,12,2,2}, {25,24,11,10,-5}, [25,24,11,4,1], [25,24,11,3,2], {25,24,11,2,3},
{25,24,10,9,-3} … [25,24,7,6,3], [25,24,7,5,4]. {25,24,7,5,5}, {25,24,5,4,9},
{25,23,22,21,-26} etc…

We’ve generated far fewer tuples, but there is scope to speed this up further, we can eliminate the ‘…’ by adding the negative (7) number (minus 1) to the number before it. Thus, we can move from the first shaded tuple onto the second shaded tuple bypassing the ‘…’.

We can also see that when we have a valid tuple, the next tuple is one where the last value is increased by one, and the penultimate value is decreased by one. I.e. the last value does not need to be re-calculated from the magic sum thus saving a little computation.

The last thing to address is when we know to decrement the numbers and which one. If {a,b,c,d,e} is our tuple, then when e>d, we decrease c; when e>c, we decrease b and when e>b, we decrease a.

So, the pseudo code to finding the sums is:

```plaintext
a=26; b=0; c=0; d=0; e=0;
while (e<a)
    decrement a; b=a;
while (e<b)
    decrement b; c=b;
while (e<c)
    decrement c; d=c-1; e=magicSum-a-b-c-d;
    if (e<1) then d=d+e-1; e=1; //e is negative
    if (d<1) then c=c+d; //d is negative
    while (e<d)
        if (e<d AND a+b+c+d+e equals magicSum) then
            found solution so store {a,b,c,d,e};
            decrement d; increment e; //generate next tuple
        endwhile (d);
    endwhile (c);
endwhile (b);
endwhile (a);
```
We should now have a list of all tuples, and now need to find four of them each with different numbers bar one, which we can do with for loops and check that the numbers in the tuple have just one number in common. We don’t know which tuple (of the four) corresponds to which sum through the centre cell (diagonal, centre row or centre column), nor do we know which number in each tuple corresponds to which cell along the sum; however, these can all be done via nested for loops iterating through all combinations.

So, we generate the square one cell at a time again. If the cell is a sums cell (a cell that is represented by a tuple), then iterate through the numbers in the tuple, else iterate through the numbers not visited yet in the tuples, always checking that the last number to be filled from any sum sums it to the magic sum.

It seems useful to create a class to represent the tuple. It should make things neater as I wouldn’t have to represent the tuples as say an array of size 5. Since I plan to store the tuples in an array then we’ll get an array of sized 5 arrays, which could get a bit confusing in indexing a specific number in the tuple. The class would have a,b,c,d and e as its attributes (containing the numbers in the tuple) and assessors to return its attributes.

A method to compare two tuples and sees if they have a common number between them will be useful in finding those four sums with one common number between them (highlighted in bold in the pseudo code below). All numbers in the two tuples are checked against each other and if only one number matches then return the common number, else return zero (an invalid number).

```java
oneCommon( {aa, bb, cc, dd, ee} ) {
    int match=0; int common=0;
    if (aa equals a,b,c,d or e) then increment match, common=aa;
    if (bb equals a,b,c,d or e) then increment match, common=bb;
    if (cc equals a,b,c,d or e) then increment match, common=cc;
    if (dd equals a,b,c,d or e) then increment match, common=dd;
    if (ee equals a,b,c,d or e) then increment match, common=ee;
    if  match equals 1 then return common
    else return 0;
}
```
Another useful method will be one that returns the number not used in the sum and will be useful for the shaded lines in the pseudo code shown below. We can do this by comparing the numbers in the tuple with the numbers used in the square for that sum and return the number that isn’t used or zero otherwise.

```java
last(aa bb cc dd) {
  if (a not equals aa,bb,cc and dd ) then return a;
  if (b not equals aa,bb,cc and dd ) then return b;
  if (c not equals aa,bb,cc and dd ) then return c;
  if (d not equals aa,bb,cc and dd ) then return d;
  if (e not equals aa,bb,cc and dd ) then return e;
  else return 0;
}
```

The pseudo code of the whole thing to hopefully make things clearer then:

create list of tuples (as given above);

for every set of four tuples with a common number between them where...

1\textsuperscript{st} tuple is the bold dark shaded diagonal sum (a,g,m,s,y)
2\textsuperscript{nd} tuple is the light shaded centre column sum (c,h,m,r,w)
3\textsuperscript{rd} tuple is the dark shaded diagonal sum (e,i,m,q,u)
4\textsuperscript{th} tuple is the bold light shaded centre row (k,l,m,n,o)

set numbers in tuples visited;

$m =$ common number;

for (a=num in 1\textsuperscript{st} tuple, c=num in 2\textsuperscript{nd} tuple, e=num in 3\textsuperscript{rd} tuple)
  for (b=unvisited numbers)
    d=magicSum-a-b-c-e;  visit[b]=true;   visit[d]=true;

for (g=num in 1\textsuperscript{st} tuple, l=num in 4\textsuperscript{th} tuple, q=num in 3\textsuperscript{rd} tuple)
  v=magicSum-b-g-l-q;   visit[v]=true;

for (i=num in 3\textsuperscript{rd} tuple, n=num in 4\textsuperscript{th} tuple, s=num in 1\textsuperscript{st} tuple)
  u=number left in 3\textsuperscript{rd} tuple;   y=number left in 1\textsuperscript{st} tuple;
  x=magicSum-d-i-n-s;   visit[x]=true;
  w=magicSum-u-v-x-y AND is number in 2\textsuperscript{nd} tuple;

for (h=numbers in 2\textsuperscript{nd} tuple)  \quad r=number left in 2\textsuperscript{nd} tuple;
  for (k=numbers in 4\textsuperscript{th} tuple)  \quad o=number left in 4\textsuperscript{th} tuple;
5. Solving a Magic Square

for (f=unvisited numbers)
j=magicSum-f-g-h-i;
p=magicSum-a-f-k-u;
t=magicSum-e-j-o-y;
found solution so output solution;
endfor;
endfor; endfor; endfor;
visit[x] = false;
endfor; endfor; endfor;
visit[v]=false;
endfor; endfor; endfor;
visit[b]=false; visit[d]=false;
endfor;
endfor; endfor; endfor;
set numbers in tuples unvisited;
endfor;

NOTE: Checking that a number is valid happens every time a cell is assigned but is left out for clarity: if the number is out of range, is already in the square, or doesn’t sum to the magicSum then we continue onto the next iteration of the for loop it’s in.

5.2.3 Implementation & Evaluation

A bit of backtracking was employed here: the initial design and implementation of the class to represent the tuples did not have the ‘last’ method to help with those shaded lines in the pseudo code. Instead, it was implemented via a series of conditions that checked the number was in range, summed to the magic sum, was a number in the tuple and that the number had not been used yet.

The ‘last’ method was designed and implemented to reduce the number of conditions to evaluate and made computation time around three minutes quicker.

Dealing with ‘sums’ cells first as these have less iterations and completing sums as soon as possible as this leaves less numbers to use at the end should mean we move through iterations as quickly as possible because we find invalid squares as soon as we can. The cells therefore, got filled in this order: a,c,e,b,d, g,l,q,v, i,u,n,s,x,y,w, h,r,k,o, f,j,p,t.
5. Solving a Magic Square

The class to represent the tuples was implemented as `Sum`, its prototype is:

```cpp
class Sum {
    private:
        int a, b, c, d, e;
    public:
        Sum(int, int, int, int, int);  //constructor
        int get(int) const;             //assessor
        //returns number if sum contains one number only in common
        //with given sum, else it returns 0
        int oneCommon(Sum);
        //returns the last number in the sum
        int last(int, int, int, int);
    }
}
```

The full implementation can be found in appendix E, the full source code is implemented in `sums5.cc`. Solutions calculated were outputted to the terminal, please see appendix C for screenshots.

Generating the tuples took no time at all, the time class of C++ gives zero seconds to produce 1394 tuples (Juergen Koeller [3] also finds this many sums for the order 5 square). Finding sets-of-sums was via four nested for loops and so runs in \(O(m^4)\) (where \(m\) is the number of tuples to iterate through). Filling the square was via thirteen nested for loops (mostly) performing five iterations, thus run time for this is around \(O(n^{13})\). Therefore the overall run time should be \(O(m^4 \cdot n^{13})\), compared to \(O(n^{30})\) for the brute-force method. As \(m\gg n\), then this dominates and for \(m=1394\) and \(n=5\), the brute-force method should actually run faster. However, this does not account for the iterations skipped in finding sets-of-sums so it should run much faster than \(O(m^4)\), and therefore faster than \(O(n^{30})\).

The program was evaluated against the following criteria:

- **Correctness of results**

  The number of solutions for the order 5 square is very large (and I can’t find a listing of them) and so it is difficult to check the results. Even if it does find the solutions, how do we know those are all the solutions (and are the right ones)? We don’t, and until someone waits years for the brute-force method to compute then we never will.

  However, the ‘sets-of-sums’ technique was adapted and applied to the order 3 square (`sums3.cc`) and computed all the solutions (checked by eye with ones in the background: section 2.5). It was also applied to the order 4 square, but with the common number cell being a (top-left) corner cell and using three tuples (top row, first column and diagonal sums). This also managed to compute all the solutions.
5. Solving a Magic Square

(to SUMS4) and comparing it with the file (again with file comparison and ordering the solutions) from the brute-force method (BRUTE4) stated there were no differences between the files. As the order 4 brute-force results are correct then these must be too.
(check4.cc was written to order the solutions in SUMS4 ready for file comparison).
The fact that this method seems to produce all the squares for order 3 and 4, reassures us that the solutions computed for the order 5 will also be correct.
Incidentally, the brute4.cc was slightly quicker (virtually negligible) than the sums4.cc at computing all solutions of the order 4 square. The order 4 brute-force method takes $O(n^{16})$ and the sets-of-sums takes $O(m^4n^7)$. So in this case, the $m^4$ term dominates – computing the sets-of-sums in order to fill the square faster costs more than is saved.

- **Speed and efficiency**
sums5.cc computed the first half a million solutions in 562 seconds (on my 1133MHz machine). Assume there are 2.2 billion solutions, at this rate it would take 28.6 days to compute – which is a vast improvement on the brute force method. So the $m^4$ term is not so dominating here and time is saved in computing the sets-of-sums to make filling the square faster.

The sets-of-sums method meets the objective of finding solutions to the order 5 square, and sums3.cc and sums4.cc suggest that all solutions will be computed and will be correct.
This program is certainly more efficient than the brute-force method and so overall, this program meets the objective of solving a Magic Square (for the order 5).

5.3 Order 5 Square (parallel sets-of-sums method)
The nice thing about the sums5.cc code is that it is able to run on many machines (in parallel), thus computing the number of total solutions quicker, and this is because it is a series of for loops.
The first set of for loops finds the set of sums with a common number by iterating through all 1394 tuples. However, the very first for loop of these (for (aa=0; aa<sums.size(); aa++)) can be split to iterate through the first half of the tuples, whilst another processor has the for loop iterate through the second half of the list. In doing this, we should halve the computation time. We can do this because the first for loop is not nested, each tuple it iterates is iterated through once only.
So sums5.cc is modified to iterate through a specified tuple, the program can then be run on another machine and iterate through a different tuple. Outputting the solutions was commented out to save more time and memory as initially writing the solutions to file generated 40MB files. Checking that these solutions are correct is not feasible since there is no existing list of solutions available, which doesn’t surprise me because 1394 lots of 40MB files takes up about 56GBs. So all I shall be interested in will be the number of solutions it computes.
The modified sums5.cc (parallel5.cc) allows the user to enter the tuple via command line arguments; it computes the number of solutions for that tuple, outputs the number of solutions and how long it took (via the time class of C++). Results are outputted to the screen and written to file. A little verbose was outputted to the screen to indicate how much was computed. See appendix C for a screenshot.

parallel5.cc was initially run for several tuples in order to obtain more data and make a better estimate on the overall computation time than that made before. This data (given in appendix F) showed that parallel5.cc computes 1804 solutions per second, at this rate, total computation time would be 14 days (on my 1133MHz machine).

I had access to six machines, which meant that computation time could be cut to around 2.5 days. Considering the total number of solutions for the order 5 is ‘unconfirmed’ in this project, I decided that it was justified and feasible to attempt the computation.

The program ran on six different machines (each putting in different amounts of CPU time), appendix F gives a brief summary of the data obtained, the full data set is given in RESULTS5.

The predicted number of solutions was obtained over 11 days, taking up a total of 15.87 days of CPU time.

The program was evaluated against the following criteria: -

- **Correctness of results**
  
  As mentioned, we cannot check if our squares are correct, but at least it managed to get the predicted number of solutions as given in the background (section 2.5). With sums3.cc and sums4.cc being correct, there is a strong possibility that the sets-of-sums method does compute all the solutions correctly, and that our results for the order 5 are correct.

- **Speed and efficiency**
  
  Results took longer to obtain than predicted because no machine ran for a full 24hrs a day (which is what the predicted value assumes), instead they usually ran for a few hours at a time in order to reduce unnecessary strain on the computer. Also, the six machines didn’t have an equal share. The two machines I personally had access to computed about three-quarters of the total computation with the last quarter being computed on the other four machines (by friends). My 1133MHz machine did the most computation (6.67 days).

  Undoubtedly, this was more efficient than the ordinary set-of-sums method as this has no parallelism. Even though the whole process took 11 days, I still feel the program was quite efficient. Considering the number of solutions is not confirmed, it was perhaps worth the effort in finding out the solution. However, if the program was just to check an already confirmed value then it isn’t so justified to have computers running for days. It also backs-up that each unique solution translates to eight total
solutions, so any future projects should just concentrate on calculating unique solutions and then the total solutions will be the eight transformations of it.

The aim of this ‘solving’ chapter was to develop programs to find all solutions to a square. The evaluation from the brute-force method showed that this objective was fulfilled for the order 3 and 4 square. It was deemed inefficient for order 5 so efficiency was improved firstly by developing the sets-of-sums method and then by making it run ‘in parallel’ we therefore also fulfil the aim of this chapter for the order 5 square too.

The programs meet the minimum requirement of finding all solutions to a square, and exceeds it by being more efficient (at least in the order 5 case).
6 CREATING A MAGIC SQUARE (GENERIC METHOD)

There were two problems that weren’t dealt with in chapter ‘Creating a Magic Square’, they were:

- Fill in a partially filled square such that it is a Magic Square.
- Come up with a single method to create a Magic Square of any order (existing methods deal with either odd, double-even or single-even squares).

Filling in a partially filled square can be done by modifying the brute-force method of solving a magic square. We can tweak the existing code by adding a condition in each for loop that checks if a cell is a filled cell then set cell value to be the filled cell value and move onto the next for loop, else we iterate as per usual. More conditions will also have to be added to check that filled cells which are also the last cell in a sum, will still sum to the magic sum. Basically, add a few if statements and the objective is fulfilled, I therefore will not delve any further.

What I’m more interested in is developing a generic method to create a Magic Square because this does not currently exist - perhaps this is because such algorithms are slower?

6.1 Analysis

A recursive brute-force method (has to be recursive here as we won’t know how many nested for loops to use) would be a way to do this, but we have seen that this is slow. Instead, I plan to adapt the faster sets-of-sums method for any order square and then terminate the program as soon as it’s found the first solution.

The sets-of-sums method comprised of three main components:

- Generating the list of tuples
- Finding sets of tuples with a common number
- Filling the square: iterate through the tuples if it’s a tuple cell, else iterate through unvisited numbers

The sets-of-sums method had been adapted previously to the order 3 and 4 by removing nested loops, but we cannot do this for the general case. We will have to find a general method to perform each of the components above. We can see that this will take time and effort, which unfortunately, due to time constraints mean I will not be able to complete the work for this chapter. I did however, make a start on the design, which can be completed and implemented as part of a future project.
6. Creating a Magic Square (generic method)

6.2 Design

The pseudo codes given are perhaps quite vague, but the element of backtracking in the Waterfall method means we can flesh out the details during implementation.

6.2.1 Generating All Tuples

The existing way involves while loops and (first/second/third etc... number < last number) conditions. We therefore need a counter that keeps track of which number (first/second/third etc...) the condition evaluates against (major in the code). Another counter is needed to keep track of which column we decrement (minor in the code). Variables to temporarily store the tuple values cannot be used, so instead, we can use arrays (tuple in the code). Remember that whenever we decrement a value (that’s not the penultimate value in the tuple) then the rest of the tuple decreases sequentially by one, such as decreasing the second number in \{25,24,5,4,9\} generates \{25,23,22,21,-26\} as the next tuple.

The pseudo code (bear in mind that array indexing starts at zero in C++):

```
last = order n-1; tuple = array of size order n;
major = last-2; minor = last-2; magicSum = \frac{1}{2}(n^3+n);
initialise tuple: tuple[0] = n^2;
//create initial sequentially decreasing tuple
for (i=0..last-2) do  tuple[i+1] = tuple[i]-1;
calc tuple[last] from magicSum;

While (tuple[0] > tuple[last])
  While (major not equals -1 AND tuple[major] > tuple[last])
    While (minor not equals -1 AND tuple[minor] > tuple[last])
      //tuple is negative so recalculate last and penultimate values
      if (tuple[last]<1) then
        tuple[last-1] = tuple[last-1] + tuple[last] −1;
tuple[last]=1;
      //found valid tuple
      if (tuple[last]<tuple[last-1]) then
        found solution: store tuple in an array;
decrement tuple[last-1]; increment tuple[last];

      //last>penultimate, decrement and sequentially decrease tuple
      if (tuple[last] ≥ tuple[last-1] then
        decrement tuple[minor] and sequentially decrease rest of tuple;
calc tuple[last] by taking tuple values away from magicSum;
endwhile;
```
6. Creating a Magic Square (generic method)

//last>minor, decrement and sequentially decrease tuple
decrement minor;
if minor not equals -1 then
decrement tuple[minor] and sequentially decrease rest of tuple;
calc tuple[last] by taking tuple values away from magicSum;
endwhile;

//last>major, decrement and sequentially decrease tuple
decrement major;
if major not equals -1 then
decrement tuple[major] and sequentially decrease rest of tuple;
calc tuple[last] by taking tuple values away from magicSum;
minor=last-2; //reset minor
endwhile;

6.2.2 Finding Sets-of-Sums

In odd order cases we use four sums because we used the centre cell as the common number and there are four sums through the centre cell. The order 4 square does not have this so three sums were used because the corner cell was used as the common number and it has three sums through it. In the general case, we have to use the order 4 sets-of-sums way as not every square has a centre cell, but they do all have a corner cell.
The code for this then can be exactly the same as that already implemented in sums4.cc, which in a more pseudo-code looking way is:

   for ( aa=0..tuples.size() ) do
      for (bb=0..tuples.size() ) do
         if (bb equals aa OR bb has no common number with aa) then
            continue to next iteration of bb;
         common = common number between aa and bb;
         for (cc=0..sums.size() ) do
            if (cc equals aa or bb OR cc not have the common number) then
               continue to next iteration of cc;
            ...
               // code to fill in square
         endfor;
      endfor; endfor; endfor;
6. Creating a Magic Square (generic method)

6.2.3 Filling the Square

We have to apply recursion here as we don’t know the number of nested for loops to use. It’s based on the principle for the brute-force method with an element of the sets-of-sums method. Cells in the square get filled sequentially as in brute-force (remember the sets-of-sums filled in the order: a,c,e,b,d, g,l,q,v, i,u,n,s,x,y,w, h,r,k,o, f,j,p,t), but the numbers each cell can be is determined by the sets-of-sums method. If a cell is in a tuple cell then you iterate through the numbers in the tuple, otherwise you iterate through the unvisited numbers. Again, knowing the penultimate cell of a sum lets you calculate the last cell in the sum.

Below is the pseudo-code, where cell represents which cell of the square we are iterating through for. This recursive function can be started by calling iterate(1) and run to iterate(n²);

iterate(cell) {
    for each value cell can take do
        if visit[i] is false then
            if cell is penultimate cell in a sum then {
                x = calculate last cell in sum from magicSum;
                if x is valid then
                    value[cell] = i; visit[cell] = true;
                    value[last cell] = x; visit[last cell] = true;
                    if cell is last cell to be filled then
                        found solution: print value[cells] and exit program;
                        if cell is penultimate cell in row then
                            iterate(cell+2); //recurse
                        else iterate(cell+1); //recurse
                    }
            else if cell is not penultimate cell in a sum then
                value[cell] = i; visit[cell] = true;
                iterate(cell+1); //recurse
        //back from recursion
        if cell is penultimate cell in a sum then visit[last cell] = false;
        visit[cell] = false;
    endfor;
} //iterate

Due to time, this is all I managed to do but the completion of the work for this chapter can be done as a future project.
The focus of this project has been to create and solve Magic Squares using a program, because doing these by hand is a tiresome and cumbersome task and a program would also be able to do this with speed and accuracy.

The creating a Magic Square program was deemed a success as it produced correct squares ‘quickly’, even for very large orders, and efficiently ($O(n^3)$).

The brute-force program to solve Magic Squares completed its objective of finding all solutions to a square. My results confirming those already calculated for orders 3 and 4. As the total number of solutions for the order 5 square was unconfirmed, such was the reason why I had to make the code faster in order to calculate it. Making the code more efficient by using the sets-of-sums method and then by running it in parallel allowed me to compute the expected number of solutions and so considered it a huge success. It is difficult to say if the solutions are correct (but evidence from sets-of-sums on order 3 and 4 suggest they are), at least the code is efficient (compared to the brute-force method anyway). Whether taking 11 days on a standard 1.1GHz PC to calculate the solution is ‘efficient’, is a matter of opinion, but I think as it was ‘unknown’, then it was worth computing.

As no existing implementations to create/solve squares were found, evaluating against them was not viable and so programs were only evaluated against correctness of results and the speed and efficiency. In terms of speed and efficiency, programs were only compared with each other or with my previous implementations (brute5.cc against sums5.cc for example).

The minimum requirements of this project were exceeded further with my attempts to develop my own generic algorithm to create a Magic Square. Due to time, this chapter did not get finished and so I cannot evaluate it.

Overall, I consider the project to be a success, it fulfils its aims and objectives, it surpassed its minimum requirements and I managed to calculate the (unconfirmed) total solutions to the order 5 square.
7. Project Evaluation

7.1 Extensions

There are many ways this project could be extended. In the order we encounter them in the project:

1. **Consider create/solving different types of Magic Squares.**
   
   This project only deals with normal Magic Squares.

2. **Finding another method that is more efficient than the sets-of-sums method.**
   
   The sets-of-sums method combined with parallelism is ‘quick’ and more efficient than the brute-force but it is not really justifiable to use it if it was to compute a value already known.

3. **Improve the efficiency of the sets-of-sums method further.**
   
   Re-ordering conditions in if statements can make a difference and I think I managed to get it running at its fastest. There is certainly scope in reducing the conditions when checking ‘w’ is valid as the existing implementation checks that the value is in range, is in the sum, is not a number used yet and sums it to the magic sum. Perhaps we can design a method to help us with this.

4. **Writing the solutions of the order 5 square to file.**
   
   There is no listing of such data available and producing it means future projects can be checked against them. This is however, purely hardware limited.

5. **Apply the sets-of-sums method to the order 6 (or higher) square and estimate the total solutions and its computation time.**
   
   Estimation of the order 5 square gave us an idea on the total computation time. Had this been longer, I would not have attempted to compute all solutions. Would the estimates on higher orders back up those already calculated? And how long would it take to compute?

6. **Work out how to iterate through all the numbers / fill in the square so that we just generate the unique solutions.**
   
   We have shown that sets-of-sums finds all the solutions and that this value is always eight times the unique solutions. We would therefore save time if we can calculate the unique solutions only (total solutions can be obtained via transformations of them). An interesting thing to note is that all the solutions in RESULTS5 can be divisible exactly by eight, which might suggest that all eight transformations are generated for each iteration. Therefore, we need to work out how to limit the iterations or how the square is being filled so that only unique solutions are produced.

7. **Finish designing and implementing my generic method to create a Magic Square.**
   
   I only managed to get part way through the chapter.

---

Number 5 can be easily done and would be interesting if it gives estimates similar to current values. Numbers 2, 3 and 6 improve on computation time and are more demanding. For numbers 2 and 6 in particular, it’s difficult (on initial inspection) to see how to go about it. Number 1 is like doing another project so requires a lot of work, and number 4 is silly to do due to the vast amounts of memory it would need. Number 7 is perhaps the best extension as this is half-completed anyway.
REFERENCES


Dare I say, that this project has been, well, fun.

I think the main thing learnt is definitely time management and to constantly dedicate some time per week in order to complete this project. After all, six months really isn’t that long considering you have to make sure you don’t let your other modules slide.

Another thing to learn is though computers are getting faster and faster, efficiency is key. Why I spent days waiting to compute the solutions for the order 5 square I do not know. I think that with friends willing to dedicate CPU time for me made it all the more feasible and motivated me to carry on. However, next time, I think I’ll just call up NASA and ask if I could borrow their supercomputers for a couple of hours! I was so relieved when the program finally came up with the predicted result.

I would like to think that I did not encounter any major problems and I would like to suggest that this is probably due to how well the background research is done. As the background paved the way I tackled this project, I knew what I was doing and had ideas on how I was going to do it. Approaching the project was not a problem, it was just a matter of finding time and getting it done. With having a busy first semester, it was difficult to do this sometimes but I did not stress over it as I was doing a 20-credits project and knew I could dedicate the time during the second semester.

So do not underestimate how important background research is: not only does it tell you what’s out there, it shows what you can and can’t do, and it gives you ideas, answers and directions to complete the project. The background sets a plan for your project and a good plan will let you foresee any possible problems, which you can overcome by doing more research.

Maybe the fact that the project was on Magic Squares made it more enjoyable as I was researching ‘recreational puzzles’, which adds an element of fun into the process.

If I were to do this project again, I would start obtaining results from parallel5.cc earlier if only to lighten the load come Easter. Either that or beg around more for any spare CPU time. My poor laptop must have been melting in doing seven days worth of computation within eleven days.

I would love to pursue this project further, especially to complete the last chapter on developing a generic method to create a Magic Square, and also to apply the sets-of-sums method to higher orders of squares and make estimations on their total number of solutions.
## APPENDIX B: FILE LISTING & COMPILING

- **File listing in alphabetical order**

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>brute3.cc</td>
<td>source code for program to compute the total solutions to the order 3 square via brute-force method</td>
</tr>
<tr>
<td>BRUTE4</td>
<td>total solutions to the order 4 square computed by brute4.cc</td>
</tr>
<tr>
<td>brute4.cc</td>
<td>source code for program to compute the total solutions to the order 4 square via brute-force method</td>
</tr>
<tr>
<td>BRUTE4SORT</td>
<td>sorted total solutions to the order 4 square (generated by check4.cc)</td>
</tr>
<tr>
<td>brute5.cc</td>
<td>source code for program to compute the total solutions to the order 5 square via brute-force method</td>
</tr>
<tr>
<td>check4.cc</td>
<td>source code for program to read in BRUTE4 and SUMS4 (with headers removed) and outputs ordered solutions to BRUTE4SORT and SUMS4SORT respectively. It also reads in ORDER4, translates each unique solution into eight total solutions and outputs the ordered solutions to ORDER4SORT</td>
</tr>
<tr>
<td>CREATE</td>
<td>sample square created by create.cc</td>
</tr>
<tr>
<td>create.cc</td>
<td>source code for program to create a normal Magic Square</td>
</tr>
<tr>
<td>ORDER4</td>
<td>unique solutions to the order 4 square taken from Harvey Heinz’s web-site [6]</td>
</tr>
<tr>
<td>ORDER4SORT</td>
<td>sorted total solutions to the order 4 square (generated by check4.cc)</td>
</tr>
<tr>
<td>Parallel5.cc</td>
<td>source code for program to compute the total solutions to the order 5 square via parallel sets-of-sums method. Usage: <code>cslin###% foo tuple outfile</code></td>
</tr>
<tr>
<td>RESULTS5</td>
<td>data obtained from running parallel5.cc on all 1394 tuples</td>
</tr>
<tr>
<td>sums3.cc</td>
<td>source code for program to compute the total solutions to the order 3 square via sets-of-sums method</td>
</tr>
<tr>
<td>SUMS4</td>
<td>total solutions to the order 4 square computed by sums4.cc</td>
</tr>
<tr>
<td>sums4.cc</td>
<td>source code for program to compute the total solutions to the order 4 square via sets-of-sums method</td>
</tr>
<tr>
<td>SUM4SORT</td>
<td>sorted total solutions to the order 4 square (generated by check4.cc)</td>
</tr>
<tr>
<td>sums5.cc</td>
<td>source code for program to compute the total solutions to the order 5 square via sets-of-sums method</td>
</tr>
</tbody>
</table>

All files can be found on the diskette accompanying this project except BRUTE4SORT and SUMS4SORT, which are identical to ORDER4SORT anyway.
• **Compiling the source code using g++ under linux...**

Compile: `cslin###% g++ foo.cc`
Execute: `cslin###% ./a.out`

or...

Compile: `cslin###% g++ foo.cc -o foo`
Execute: `cslin###% ./foo`

Except parallel5.cc which executes using command line arguments:-

`cslin###% foo tuple outfile`
APPENDIX C: SCREENSHOTS

- Creating a Magic Square (section 4)
  Valid input on the left, and invalid input on the right

![Magic Square: Creating a Magic Square VI.0 by Lemoning Tang](image1)

Please enter order of square: 3

Create an odd order square using the following methods...
1. Siamese / de la Loubère
2. Lecreux
3. de la Hire
Please choose a method: 4

Square has been created. Checking square is magic... square is magic.
Outputting square to file...
Please see file MAGIC to see the square.

- Solving a Magic Square: brute-force method (section 5.1)
  brute3.cc and brute4.cc top left and right respectively, brute5.cc bottom left.

![Magic Square: Solving a Magic Square VI.0 by Lemoning Tang](image2)

Order 3 square: brute-force method
1. 2 7 6
2. 1 5 9
3. 8 3 4

Order 5 square: brute-force method
1. 2 5 3 2 1 4 6 9 7 5 3 2 1 4 6 9 7 5 3 2 1 4 6 9 7 5 3 2 1 4 6 9 7 5 3 2 1 4 6 9 7 5 3 2 1 4

Order 9 square: brute-force method
1. 2 5 3 2 1 4 6 9 7 8 5 2 1 4 6 9 7 8 5 2 1 4 6 9 7 8 5 2 1 4 6 9 7 8 5 2 1 4 6 9 7 8 5 2 1 4
Solving a Magic Square: sets-of-sums method (section 5.2)

Non parallel and parallel sets-of-sums method for the order five square below.

sums3.cc (left) and sums4.cc (right) to prove sets-of-sums method calculates all the squares.
APPENDIX D: TEST & RUN-TIME DATA FOR CREATE.CC

- Test data for creating a Magic Square

Test methods are implemented correctly

<table>
<thead>
<tr>
<th>METHOD</th>
<th>EXAMPLE</th>
<th>IS MAGIC?</th>
<th>MATCH WITH DANIELSSON’S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Siamese</td>
<td>3, 5, 7, 9, 11</td>
<td>all true</td>
<td>all match</td>
</tr>
<tr>
<td>Lozenge</td>
<td>3, 5, 7, 9, 11</td>
<td>all true</td>
<td>all match</td>
</tr>
<tr>
<td>Mezeriac</td>
<td>3, 5, 7, 9, 11</td>
<td>all true</td>
<td>all match</td>
</tr>
<tr>
<td>Agrippa</td>
<td>4, 8, 12, 16, 20</td>
<td>all true</td>
<td>all match</td>
</tr>
<tr>
<td>Inverse-Agrippa</td>
<td>4, 8, 12, 16, 20</td>
<td>all true</td>
<td>all match</td>
</tr>
<tr>
<td>Blocks</td>
<td>4, 8, 12, 16, 20</td>
<td>all true</td>
<td>all match</td>
</tr>
<tr>
<td>Inverse-Blocks</td>
<td>4, 8, 12, 16, 20</td>
<td>all true</td>
<td>all match</td>
</tr>
<tr>
<td>Lux</td>
<td>6, 10, 14, 18, 22</td>
<td>all true</td>
<td>all match</td>
</tr>
<tr>
<td>Strachney</td>
<td>6, 10, 14, 18, 22</td>
<td>all true</td>
<td>all match</td>
</tr>
</tbody>
</table>

Test any order of square is created

<table>
<thead>
<tr>
<th>ORDER</th>
<th>PREDICTED RESULT</th>
<th>EXPECTED RESULT?</th>
</tr>
</thead>
<tbody>
<tr>
<td>33, 157, 4351</td>
<td>create odd square</td>
<td>yes</td>
</tr>
<tr>
<td>44, 160, 828</td>
<td>create double-even square</td>
<td>yes</td>
</tr>
<tr>
<td>66, 346, 422</td>
<td>create single-even square</td>
<td>yes</td>
</tr>
<tr>
<td>1, 2, 99999</td>
<td>order out of range</td>
<td>yes</td>
</tr>
</tbody>
</table>
- **Run-time data for creating a Magic Square**

Three times (in seconds via time class of C++) were taken for each order listed for each method. Data obtained from each method fits an $x^2$ curve with a very small error ($R^2$ is approximately 0.99 for all methods); each method runs in $O(n^2)$.

Note that the higher the $x^2$ coefficient, the slower the method – odd order methods are the slowest with double-even methods being the quickest.
### Agrippa

<table>
<thead>
<tr>
<th>ORDER</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>AVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>3000</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>4000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0.33</td>
</tr>
<tr>
<td>5000</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1.87</td>
</tr>
<tr>
<td>6000</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2.00</td>
</tr>
<tr>
<td>7000</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2.33</td>
</tr>
<tr>
<td>8000</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>3.67</td>
</tr>
<tr>
<td>9000</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4.33</td>
</tr>
<tr>
<td>10000</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>6.33</td>
</tr>
<tr>
<td>11000</td>
<td>7</td>
<td>6</td>
<td>7</td>
<td>6.87</td>
</tr>
<tr>
<td>12000</td>
<td>8</td>
<td>7</td>
<td>7.87</td>
<td></td>
</tr>
<tr>
<td>13000</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9.00</td>
</tr>
<tr>
<td>14000</td>
<td>10</td>
<td>11</td>
<td>11.33</td>
<td></td>
</tr>
<tr>
<td>15000</td>
<td>12</td>
<td>12</td>
<td>12.67</td>
<td></td>
</tr>
</tbody>
</table>

#### Run-time for Agrippa method

- \( y = 0.0506x^2 + 0.2291x + 0.2644 \)
- \( R^2 = 0.9948 \)

### Inverse-Agrippa

<table>
<thead>
<tr>
<th>ORDER</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>AVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>3000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0.67</td>
</tr>
<tr>
<td>4000</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>5000</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1.33</td>
</tr>
<tr>
<td>6000</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1.87</td>
</tr>
<tr>
<td>7000</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2.33</td>
</tr>
<tr>
<td>8000</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3.00</td>
</tr>
<tr>
<td>9000</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4.00</td>
</tr>
<tr>
<td>10000</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5.00</td>
</tr>
<tr>
<td>11000</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6.00</td>
</tr>
<tr>
<td>12000</td>
<td>7</td>
<td>0</td>
<td>8</td>
<td>7.67</td>
</tr>
<tr>
<td>13000</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9.33</td>
</tr>
<tr>
<td>14000</td>
<td>19</td>
<td>13</td>
<td>19.00</td>
<td></td>
</tr>
<tr>
<td>15000</td>
<td>11</td>
<td>11</td>
<td>11.00</td>
<td></td>
</tr>
</tbody>
</table>

#### Run-time for Inverse-Agrippa method

- \( y = 0.0539x^2 + 0.1437x + 0.3963 \)
- \( R^2 = 0.9969 \)

### Blocks

<table>
<thead>
<tr>
<th>ORDER</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>AVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>3000</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0.67</td>
</tr>
<tr>
<td>4000</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0.33</td>
</tr>
<tr>
<td>5000</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>6000</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1.87</td>
</tr>
<tr>
<td>7000</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2.00</td>
</tr>
<tr>
<td>8000</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2.00</td>
</tr>
<tr>
<td>9000</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>3.33</td>
</tr>
<tr>
<td>10000</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3.87</td>
</tr>
<tr>
<td>11000</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>4.33</td>
</tr>
<tr>
<td>12000</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>5.67</td>
</tr>
<tr>
<td>13000</td>
<td>5</td>
<td>7</td>
<td>6</td>
<td>6.33</td>
</tr>
<tr>
<td>14000</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7.00</td>
</tr>
<tr>
<td>15000</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>8.33</td>
</tr>
</tbody>
</table>

#### Run-time for Blocks method

- \( y = 0.0365x^2 + 0.1494x + 0.2654 \)
- \( R^2 = 0.991 \)

### Inverse-Blocks

<table>
<thead>
<tr>
<th>ORDER</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>AVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>3000</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0.33</td>
</tr>
<tr>
<td>4000</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>5000</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>6000</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>7000</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2.00</td>
</tr>
<tr>
<td>8000</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2.00</td>
</tr>
<tr>
<td>9000</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3.00</td>
</tr>
<tr>
<td>10000</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3.00</td>
</tr>
<tr>
<td>11000</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4.00</td>
</tr>
<tr>
<td>12000</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4.87</td>
</tr>
<tr>
<td>13000</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5.00</td>
</tr>
<tr>
<td>14000</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7.00</td>
</tr>
<tr>
<td>15000</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7.00</td>
</tr>
<tr>
<td>16000</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8.00</td>
</tr>
</tbody>
</table>

#### Run-time for Inverse-Blocks method

- \( y = 0.0313x^2 + 0.199x + 0.1469 \)
- \( R^2 = 0.9953 \)
<table>
<thead>
<tr>
<th>ORDER</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>AVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>3002</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>4002</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>5002</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2.00</td>
</tr>
<tr>
<td>6002</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3.00</td>
</tr>
<tr>
<td>7002</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4.00</td>
</tr>
<tr>
<td>8002</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5.00</td>
</tr>
<tr>
<td>9002</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6.33</td>
</tr>
<tr>
<td>10002</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7.67</td>
</tr>
<tr>
<td>11002</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>9.67</td>
</tr>
<tr>
<td>12002</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11.00</td>
</tr>
<tr>
<td>13002</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13.00</td>
</tr>
<tr>
<td>14002</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15.00</td>
</tr>
<tr>
<td>15002</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17.00</td>
</tr>
</tbody>
</table>

**Run-time for Lux method**

\[
y = 0.1275x^2 + 0.38x + 0.31
\]

\[R^2 = 0.999\]

<table>
<thead>
<tr>
<th>ORDER</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>AVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>3002</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1.33</td>
</tr>
<tr>
<td>4002</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2.00</td>
</tr>
<tr>
<td>5002</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4.00</td>
</tr>
<tr>
<td>6002</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6.00</td>
</tr>
<tr>
<td>7002</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7.00</td>
</tr>
<tr>
<td>8002</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9.67</td>
</tr>
<tr>
<td>9002</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12.00</td>
</tr>
<tr>
<td>10002</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15.00</td>
</tr>
<tr>
<td>11002</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18.33</td>
</tr>
<tr>
<td>12002</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21.53</td>
</tr>
<tr>
<td>13002</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25.00</td>
</tr>
<tr>
<td>14002</td>
<td>29</td>
<td>29</td>
<td>29</td>
<td>28.67</td>
</tr>
<tr>
<td>15002</td>
<td>33</td>
<td>33</td>
<td>33</td>
<td>33.33</td>
</tr>
</tbody>
</table>

**Run-time for Strachney method**

\[
y = 0.1395x^2 + 0.7023x + 0.4219
\]

\[R^2 = 0.9963\]
APPENDIX E: SUM CLASS USED IN SETS-OF-SUMS METHOD

//Define Sum class, represents a sum in col/row/diagonal
//
#ifndef _SUM_
#define _SUM_

class Sum {  
private:
    int a,b,c,d,e;

public:
    //constructor for order5 
    Sum(int first, int second, int third, int fourth, int fifth) { 
        a = first; b = second; c = third; d = fourth; e = fifth; 
    }

    //accessor
    int get(int x) const { 
        if (x==0) return a;
        else if (x==1) return b;
        else if (x==2) return c;
        else if (x==3) return d;
        else if (x==4) return e;
    }
}
//returns number if sum contains one number only in common with
//given sum, else it returns 0
int oneCommon(Sum s) {
    int match=0;
    int common=0;

    if (s.get(0)==a || s.get(1)==a || s.get(2)==a || s.get(3)==a || s.get(4)==a)
        {match++; common=a;}
    if (s.get(0)==b || s.get(1)==b || s.get(2)==b || s.get(3)==b || s.get(4)==b)
        {match++; common=b;}
    if (s.get(0)==c || s.get(1)==c || s.get(2)==c || s.get(3)==c || s.get(4)==c)
        {match++; common=c;}
    if (s.get(0)==d || s.get(1)==d || s.get(2)==d || s.get(3)==d || s.get(4)==d)
        {match++; common=d;}
    if (s.get(0)==e || s.get(1)==e || s.get(2)==e || s.get(3)==e || s.get(4)==e)
        {match++; common=e;}

    if (match==1) return common;
    else return 0;
}

//returns the last number in the sum
int last(int first, int second, int third, int fourth) {
    if (a!=first && a!=second && a!=third && a!=fourth) return a;
    if (b!=first && b!=second && b!=third && b!=fourth) return b;
    if (c!=first && c!=second && c!=third && c!=fourth) return c;
    if (d!=first && d!=second && d!=third && d!=fourth) return d;
    if (e!=first && e!=second && e!=third && e!=fourth) return e;
    return 0;
}

);
APPENDIX F: TOTAL SOLUTIONS TO THE ORDER 5 SQUARE
(PARALLEL5.CC)

- Data used to estimate the total computation time of parallel5.cc

Results were calculated on my 1133MHz laptop.

<table>
<thead>
<tr>
<th>ITERATE</th>
<th>SOLUTIONS</th>
<th>TIME (s)</th>
<th>MACHINE</th>
<th>TUPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1112192</td>
<td>676</td>
<td>mine</td>
<td>100</td>
</tr>
<tr>
<td>300</td>
<td>1211232</td>
<td>724</td>
<td>mine</td>
<td>300</td>
</tr>
<tr>
<td>500</td>
<td>1164672</td>
<td>706</td>
<td>mine</td>
<td>500</td>
</tr>
<tr>
<td>700</td>
<td>1876992</td>
<td>962</td>
<td>mine</td>
<td>700</td>
</tr>
<tr>
<td>900</td>
<td>1630888</td>
<td>911</td>
<td>mine</td>
<td>900</td>
</tr>
<tr>
<td>1100</td>
<td>1679280</td>
<td>944</td>
<td>mine</td>
<td>1100</td>
</tr>
<tr>
<td>1300</td>
<td>1988144</td>
<td>980</td>
<td>mine</td>
<td>1300</td>
</tr>
</tbody>
</table>

TOTAL: 10649600 5903

Solutions per second = 1804.1
Estimated computation time = 1220798 seconds
(2202441792 / 1804.1) 14.13 days

- Brief summary of the data obtained from parallel5.cc

<table>
<thead>
<tr>
<th>ITERATE</th>
<th>SOLUTIONS</th>
<th>TIME (s)</th>
<th>CPU</th>
<th>No. OF TUPLES</th>
<th>SOLUTIONS</th>
<th>TIME (s)</th>
<th>TIME (days)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-99</td>
<td>97247344</td>
<td>102437</td>
<td>mine</td>
<td>643</td>
<td>1052430280</td>
<td>575345</td>
<td>6.67</td>
</tr>
<tr>
<td>100-199</td>
<td>14295794</td>
<td>126106</td>
<td>cslin</td>
<td>293</td>
<td>395313056</td>
<td>411326</td>
<td>4.76</td>
</tr>
<tr>
<td>200-299</td>
<td>133615792</td>
<td>88896</td>
<td>fr</td>
<td>140</td>
<td>240218624</td>
<td>86171</td>
<td>0.99</td>
</tr>
<tr>
<td>300-399</td>
<td>142633664</td>
<td>140345</td>
<td>rich</td>
<td>120</td>
<td>214694880</td>
<td>191733</td>
<td>1.52</td>
</tr>
<tr>
<td>400-499</td>
<td>158036640</td>
<td>96702</td>
<td>rta</td>
<td>103</td>
<td>173242000</td>
<td>103990</td>
<td>1.20</td>
</tr>
<tr>
<td>500-599</td>
<td>141506896</td>
<td>94224</td>
<td>dt</td>
<td>95</td>
<td>125673152</td>
<td>63043</td>
<td>0.73</td>
</tr>
<tr>
<td>600-699</td>
<td>17544228</td>
<td>10741</td>
<td>TOTAL</td>
<td>1944</td>
<td>2202441792</td>
<td>1370812</td>
<td>15.87</td>
</tr>
<tr>
<td>700-799</td>
<td>142273040</td>
<td>81886</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>800-899</td>
<td>169097056</td>
<td>62692</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>900-999</td>
<td>162735976</td>
<td>65406</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000-1099</td>
<td>17269562</td>
<td>7950</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100-1199</td>
<td>17269562</td>
<td>96010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1200-1299</td>
<td>176686112</td>
<td>101622</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1300-1394</td>
<td>198713840</td>
<td>108707</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TOTAL: 2202441792 1370812

Six machines were used in total:
- mine – Computed by Leeming Tang with a 1133MHz Intel Celeron processor
- cslin – Computed by Leeming Tang via remote login into cslin.leeds.ac.uk
- fr – Computed by Frances Robinson with a 2.00GHz Pentium 4 processor
- rich – Computed by Richard Tang with a 3.06GHz Pentium 4 processor
- rta – Computed by Richard Tang with a 1.00GHz Pentium 3 Mobile processor
- dt – Computed by David Toyne with a 1303MHz AMD Athlon processor