بسم الله الرحمن الرحيم

Kingdom of Saudi Arabia
Ministry of Education
College of Computer
Department of Computer Science

المملكة العربية السعودية
وزارة التعليم
كلية الحاسب
قسم علوم الحاسب

Algorithm design & Analysis Project Report

CS315 – Group: 6670

Semester: 471

Magic Square

Group members:

Saif Alotaibi 441108631

Omar Alsuhibani 441108502

Mohammed Alrouji 441109006

Abdulrahman Alghanayam 441109766

Mohammed Alsuhaibani 441108783

Supervisor: Prof. Dr. Mohammed Al-Hagery

Table of Contents

1. Overview

1.1. Introduction:

In this report, we will show how a **non-trivial magic** square can be systematically constructed using 1 of 3 methods based on the magic square's order: **Siamese Method** for odd order, **Exchange Method** for evenly-even order and **Strachey Method** for oddly-even order. [1][2][3]

A non-trivial magic square is a square matrix of order **n** where **n** is a natural number other than 2. The sum of each row, each column and each diagonal equal the matrix's magic constant **M** (Figure 1.1). A non-trivial magic square must follow an additional rule: the matrix must contain elements **1 to n^2** with no duplicates where **n** is a natural number greater than 2 (Figure 1.2).

$$M = n \cdot \frac{n^2 + 1}{2}$$

Figure 1.1 (Formula for the magic constant M. The sums for all rows, columns and diagonals of the magic square must equal this constant)

Each of the methods mentioned have been made for creating magic squares using pen-and-paper and hand calculations before or during the 20th century. Magic squares are used mainly for recreation, meaning they are only meant for entertainment and do not serve any scientific purpose other than to inspire people to study science. A person who knows how to build a magic square could showcase their cleverness to others or simply create one to pass the time. [4]

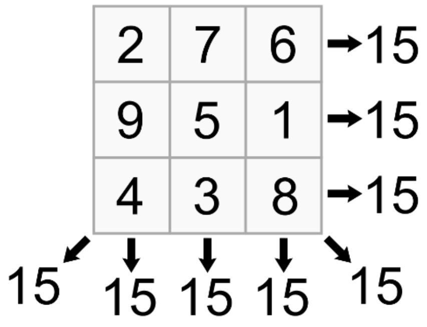


Figure 1.2 (Oldest Known non-trivial magic square found in ancient Chinese text in 190 BCE. A matrix of order 3 containing elements 1 to 3^2 with no duplicates. Each row, column and diagonal sum up to the magic constant 15)

The main goal of our project is to turn these pen-and-paper techniques into one algorithm to build a non-trivial magic square of any order and improve them.

1.2. Problem Description

The problem is to create a square matrix of order **n > 2** containing elements 1 to n^2 with no duplicates. the summation of each row, column and diagonal must equal the magic constant M. In order to do this, the algorithm will take **n** as input, create the matrix using one of three algorithms based on whether **n** is less than 3, odd, evenly-even, or oddly-even and return the matrix as output (Figure 1.3).

Input	Output
3	$\begin{matrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{matrix}$
4	$\begin{matrix} 16 & 2 & 3 & 13 \\ 5 & 11 & 10 & 8 \\ 9 & 7 & 6 & 12 \\ 4 & 14 & 15 & 1 \end{matrix}$
6	$\begin{matrix} 35 & 1 & 6 & 26 & 19 & 24 \\ 3 & 32 & 7 & 21 & 23 & 25 \\ 31 & 9 & 2 & 22 & 27 & 20 \\ 8 & 28 & 33 & 17 & 10 & 15 \\ 30 & 5 & 34 & 12 & 14 & 16 \\ 4 & 36 & 29 & 13 & 18 & 11 \end{matrix}$

Figure 1.3 (Input and output examples)

2. Theoretical Analysis

2.1. Naïve Algorithm

2.1.1. Description (Rationale)

The whole algorithm is made up of 3 lesser algorithms based on pen & paper methods: for odd order magic squares we will implement the Siamese Method, for evenly-even order squares we will implement the Exchange Method and for oddly-even order squares we will implement Strachey's Method. Before we start implementing we must first check if the order is lesser than 3, if so the algorithm immediatlly returns 0 since no non-trivial magic square is order lesser than 3 (Figure 2.1).

Not Magic Square ($n < 3$):

If the order of the square is less than 3, it cannot form a non-trivial magic square. The algorithm immediately returns 0 and the algorithm ends.

Siamese Method (odd n):

For odd-order squares, the Siamese method fills numbers sequentially by moving diagonally up and right. When a filled cell is encountered, it moves downward instead. Wrap-around is applied for edges, ensuring continuity across the matrix. This method guarantees a valid magic square for any odd n.

Exchange Method (evenly-even n, n divisible by 4):

For evenly-even order squares, the matrix is filled sequentially with numbers from 1 to n^2. Specific cells are then swapped symmetrically across the center according to an offset pattern. This swapping ensures that all rows, columns, and diagonals sum to the magic constant. The method efficiently produces a magic square without iterative adjustment.

Strachey Method (oddly-even n, n divisible by 2 but not 4):

For oddly-even squares, the algorithm divides the matrix into four equal quadrants. Each quadrant is filled using the Siamese Method, then selective swapping of columns ensures the magic property. Adjustments at the middle rows and columns resolve conflicts between quadrants. This approach allows construction of a magic square for orders that are even but not divisible by 4.

2.1.2. Pseudocode

```
Input: Integer n: order of the magic square
Output: matrix mat or 0: mat if the input is greater than 2, otherwise 0
FUNCTION magic_square(n)
   mat[n][n] ← 0
   num ← 0

   IF n < 3 THEN
      RETURN 0

   ELSE IF n MOD 2 ≠ 0 THEN
      row ← 0
      column ← n // 2
      cell ← 0

      FOR i ← 0 TO n*n − 1 DO
         temp_row ← row
         temp_column ← column
         cell ← cell + 1
         num ← num + 1
         mat[temp_row][temp_column] ← num

         IF cell = n THEN
            temp_row ← temp_row + 1
            cell ← 0
         ELSE
            temp_row ← temp_row − 1
            temp_column ← temp_column + 1
         END IF

         IF temp_row = −1 THEN
            temp_row ← n − 1
         END IF
         IF temp_column = n THEN
            temp_column ← 0
         END IF

         row ← temp_row
         column ← temp_column
      END FOR

      RETURN mat

   ELSE IF (n // 2) MOD 2 = 0 THEN
      offset ← n // 4

      FOR i ← 0 TO n − 1 DO
         FOR j ← 0 TO n − 1 DO
            num ← num + 1
            mat[i][j] ← num
```

```
        END FOR
    END FOR

    FOR row ← 0 TO (n // 2) − 1 DO
        FOR column ← 0 TO n − 1 DO
            IF row < offset THEN
                IF column < offset OR column > offset*3 − 1 THEN
                    SWAP mat[row][column] WITH mat[n−1−row][n−1−column]
                END IF
            ELSE
                IF column > offset − 1 AND column < offset*3 THEN
                    SWAP mat[row][column] WITH mat[n−1−row][n−1−column]
                END IF
            END IF
        END FOR
    END FOR

    RETURN mat

ELSE IF (n // 2) MOD 2 ≠ 0 THEN
    switch ← 1
    offset ← n // 2
    begColumn ← offset // 2

    FOR i ← 0 TO 3 DO
        switch ← NOT switch
        begRow ← offset * switch
        cell ← 0

        IF i < 2 THEN
            begColumn ← begColumn + begRow
        ELSE
            begColumn ← begColumn − begRow
        END IF

        row ← begRow
        column ← begColumn

        FOR j ← 0 TO offset*offset − 1 DO
            temp_row ← row
            temp_column ← column
            cell ← cell + 1
            num ← num + 1
            mat[temp_row][temp_column] ← num

            IF cell = offset THEN
                temp_row ← temp_row + 1
                cell ← 0
            ELSE
                temp_row ← temp_row − 1
                temp_column ← temp_column + 1
            END IF

            IF temp_row = begRow − 1 THEN
                temp_row ← begRow − 1 + offset
            END IF
            IF temp_column = offset // 2 + begColumn + 1 THEN
                temp_column ← (offset // 2 + begColumn + 1) − offset
            END IF

            row ← temp_row
            column ← temp_column
        END FOR
    END FOR

    FOR i ← 0 TO offset − 1 DO
        FOR j ← 0 TO n − 1 DO
            IF j < offset // 2 OR j > n − offset // 2 THEN
                IF i = offset // 2 AND j < offset // 2 THEN
                    SWAP mat[i][j+1] WITH mat[i+offset][j+1]
                ELSE
                    SWAP mat[i][j] WITH mat[i+offset][j]
                END IF
            END IF
        END FOR
    END FOR
  END IF
RETURN mat
```

Figure 2.1 (Naïve pseudocode algorithm)

2.1.3. Analysis

```
                                                                   #operations
1. FUNCTION magic_square(n)
2. mat[n][n] ← 0                                                        $n^2$
3. num ← 0                                                              1
4. IF n < 3 THEN                                                        1
5. RETURN 0                                                             1
6. END IF
7. IF n MOD 2 ≠ 0 THEN                                                  2
8. row ← 0                                                              1
9. column ← n // 2                                                      2
0. cell ← 0                                                             1
11. FOR i ← 0 TO $n^2 - 1$ DO                                           $n^2 + 1$
12. temp_row ← row                                                      $n^2$
13. temp_column ← column                                                $n^2$
14. cell ← cell + 1                                                     $2n^2$
15. num ← num + 1                                                       $2n^2$
16. mat[temp_row][temp_column] ← num                                    $2n^2$
17. IF cell = n THEN                                                    $n^2$
18. temp_row ← temp_row + 1                                             $2n^2$
19. cell ← 0                                                            $n^2$
20. ELSE
21. temp_row ← temp_row – 1                                             $2n^2$
22. temp_column ← temp_column + 1                                       $2n^2$
23. END IF
24. IF temp_row = −1 THEN                                                $n^2$
25. temp_row ← n – 1                                                    $2n^2$
26. END IF
27. IF temp_column = n THEN                                             $n^2$
28. temp_column ← 0                                                     $n^2$
29. END IF
30. row ← temp_row                                                      $n^2$
31. column ← temp_column                                                $n^2$
32. END FOR
33. RETURN mat                                                          1
34. END IF
35. IF (n // 2) MOD 2 = 0 THEN                                           3
36. offset ← n // 4                                                     2
37. FOR i ← 0 TO n − 1 DO                                               n + 1
38. FOR j ← 0 TO n − 1 DO                                               n(n + 1)
39. num ← num + 1                                                       $2n^2$
40. mat[i][j] ← num                                                     $2n^2$
41. END FOR
42. END FOR
43. FOR row ← 0 TO (n // 2) − 1 DO                                 (n // 2) + 1
44. FOR column ← 0 TO n − 1 DO                                     (n // 2)(n + 1)
45. IF row < offset THEN                                                n(n // 2)
46. IF column < offset OR column > offset*3 − 1 THEN               4n(n // 2)
47. SWAP mat[row][column] WITH mat[n−1−row][n−1−column]            15n(n // 2)
48. END IF
```

49. ELSE	
50. IF column > offset − 1 AND column < offset*3 THEN	4n(n // 2)
51. SWAP mat[row][column] WITH mat[n−1−row][n−1−column]	15n(n // 2)
52. END IF	
53. END IF	
54. END FOR	
55. END FOR	
56. RETURN mat	1
57. END IF	
58. IF (n // 2) MOD 2 ≠ 0 THEN	3
59. switch ← 1	1
60. offset ← n // 2	2
61. begColumn ← offset // 2	2
62. FOR i ← 0 TO 3 DO	5
63. switch ← NOT switch	8
64. begRow ← offset * switch	8
65. cell ← 0	4
66. IF i < 2 THEN	4
67. begColumn ← begColumn + begRow	8
68. ELSE	
69. begColumn ← begColumn – begRow	8
70. END IF	
71. row ← begRow	4
72. column ← begColumn	4
73. FOR j ← 0 TO offset*offset − 1 DO	$4(offset^2 + 1)$
74. temp_row ← row	$4(offset^2)$
75. temp_column ← column	$4(offset^2)$
76. cell ← cell + 1	$8(offset^2)$
77. num ← num + 1	$8(offset^2)$
78. mat[temp_row][temp_column] ← num	$8(offset^2)$
79. IF cell = offset THEN	$4(offset^2)$
80. temp_row ← temp_row + 1	$8(offset^2)$
81. cell ← 0	$4(offset^2)$
82. ELSE	
83. temp_row ← temp_row – 1	$8(offset^2)$
84. temp_column ← temp_column + 1	$4(offset^2)$
85. END IF	
86. IF temp_row = begRow − 1 THEN	$8(offset^2)$
87. temp_row ← begRow − 1 + offset	$12(offset^2)$
88. END IF	
89. IF temp_column = offset // 2 + begColumn + 1 THEN	$16(offset^2)$
90. temp_column ← (offset // 2 + begColumn + 1) – offset	$20(offset^2)$
91. END IF	
92. row ← temp_row	$4(offset^2)$
93. column ← temp_column	$4(offset^2)$
94. END FOR	
95. END FOR	
96. FOR i ← 0 TO offset − 1 DO	offset + 1
97. FOR j ← 0 TO n − 1 DO	offset(n + 1)
98. IF j < offset // 2 OR j > n − offset // 2 THEN	5offset(n)
99. IF i = offset // 2 AND j < offset // 2 THEN	5offset(n)

```
100. SWAP mat[i][j+1] WITH mat[i+offset][j+1]                    13offset(n)
101. ELSE
102. SWAP mat[i][j] WITH mat[i+offset][j]                        9offset(n)
103. END IF
104. END IF
105. END FOR
106. END FOR
107. RETURN mat                                                  1
108. END IF
```
Figure 2.2 (Naïve pseudocode algorithm analysis)

2.1.4. Time Complexity Analysis of the Naïve Algorithm

The space complexity of the algorithm reflects the amount of memory used relative to the size of the input. Below, we provide a detailed breakdown of how memory is utilized throughout each of the lesser algorithms.

Not Magic Square (n < 3):

This case simply checks whether n is less than 3 and returns immediately after a few constant-time operations. The only n-dependent work is initializing the n×n matrix, which performs n^2 writes. Therefore, the time complexity is quadratic.

Total number of operations: $n^2 + 4$
Time complexity: $O(n^2)$

Siamese Method (odd n):

The Siamese method fills every cell of an n×n matrix exactly once, producing n^2 iterations of its main loop. Inside that loop, all operations are constant time, so the total grows proportionally to $21n^2 + 9$. The overall time complexity is quadratic.

Total number of operations: $21n^2 + 9$
Time complexity: $O(n^2)$

Exchange Method (evenly-even n, n divisible by 4):

This method first fills the n×n matrix sequentially, performing n^2 operations. It then applies swapping rules based on fixed patterns, which scan only half of the rows and all columns. Each cell is touched a constant number of times. As a result, the time complexity is quadratic.

Total number of operations: $\frac{33}{2}n^2 + 3n + 9$
Time complexity: $O(n^2)$

Strachey Method (oddly-even n, n divisible by 2 but not 4):

The Strachey method builds four smaller magic squares of size n/2 × n/2. After that, the method performs a series of structured swaps between the upper and lower halves of the matrix, touching only specific columns and rows. When all steps are combined and offset is replaced with n/2, the operation count simplifies to $42n^2 + n + 59$. Because every important step involves scanning or filling large portions of the n × n matrix. The Strachey method runs in **quadratic time**.

total number of operations: $42n^2 + n + 60$
Time complexity: $O(n^2)$

Overall Space Complexity:

The overall algorithm never executes more than one condition for a given n, so its runtime is bounded by the most expensive sequence of operations among the available conditions, which is the Strachey Method. Summing the required operations shows that the total number of operations grows as $42n^2 + n + 60$, which is based on the n^2 term. Since every method fills each cell of the n×n matrix only a constant number of times, the growth rate is quadratic. Consequently, the entire algorithm runs in **O(n²)** time.

Total number of operations: $42n^2 + n + 60$
Time complexity: $O(n^2)$

2.1.5. Space Complexity Analysis of the Naïve Algorithm

The space complexity of the algorithm reflects the amount of memory used relative to the size of the input. Below, we provide a detailed breakdown of how memory is utilized throughout the algorithm.

Not Magic Square (n < 3):

This allocates an n×n matrix even though it quickly returns for invalid sizes. No additional structures are created beyond a few scalar variables.

Space complexity $O(n^2)$.

Siamese Method (odd n):

The Siamese method stores all values directly into a single n×n matrix. Aside from a handful of temporary variables for row, column, and counting, no extra data structures are used.

Space complexity: $O(n^2)$.

Exchange Method (evenly-even n, n divisible by 4):

This method also operates entirely within one n×n matrix, filling and swapping values in place. It does not allocate extra arrays or any other storage, relying only on basic scalar variables.

Space complexity: $O(n^2)$.

Strachey Method (oddly-even n, n divisible by 2 but not 4):

Although it conceptually divides the matrix into quadrants, all work is performed within the same n×n array. The algorithm does not create sub-arrays or additional matrices; it only uses index counters and temporary variables for swaps.

Space complexity: $O(n^2)$.

Overall Time Complexity:

The algorithm always allocates a single n×n matrix to store the magic square, regardless of which method is used. All other variables are simple scalars or counters, which use negligible memory compared to the matrix. Since no additional large data structures are created, the memory usage is based on the matrix itself.

Space complexity: $O(n^2)$.

2.1.6. Algorithm Scenarios:

While the time complexity of the naive magic square algorithm is always quadratic, there are some variation based on if the order is odd, evenly-even or oddly-even:

Best Case:

The best case scenario is that if the order is odd, since the Siamese method requires the most minimal amount operations required for filling n^2 cells sequentially with simple wrap around conditions:

Total number of operations: $21n^2 + 9$
Time complexity: $O(n^2)$

Average Case:

For the average case, since n can be odd, evenly-even or oddly even, it is usually roughly in the middle of all the methods. With that being said, we decided to take the average between the total number of operations of the three methods.

Total number of operations: $\frac{53}{2}n^2 + \frac{4}{3}n + \frac{77}{3}$
Time complexity: $O(n^2)$

Worst Case:

The heaviest branch is Strachey (oddly-even n), which performs the most operations due to quadrant handling and extra swaps.

Total number of operations: $42n^2 + n + 60$
Time complexity: $O(n^2)$

Summary:

Best Case: $O(n^2)$
Average Case: $O(n^2)$
Worst Case: $O(n^2)$

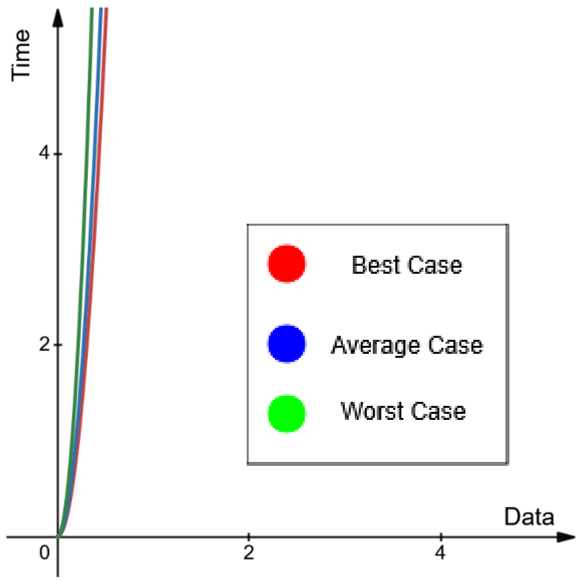


Figure 2.3 (Naïve time complexity graph)

2.1.7. Naïve Algorithm Examples

Siamese Method:

Input: n = 3

Step 1: Create an empty matrix of size n^2

0	0	0
0	0	0
0	0	0

Step 2: Go through the matrix diagonally and plot the numbers 1→9. wrap around when hitting an edge and move down once when hitting an occupied cell

8	1	6
3	5	7
4	9	2

Output: The output of step 2

Exchange Method:

Input: n = 4

Step 1: Create an empty matrix of size n^2

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

Step 2: Fill up the matrix in order

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Step 3: Swap the top corner cells and upper half of the four center cells with their complements over the center

16	2	3	13
5	11	10	8
9	7	6	12
4	14	15	1

Output: The output of step 3

Strachey Method:
Input: n = 10

Step 1: Create an empty matrix of size n^2

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

Step 2: Fill each of the four quadrants sequentially via the Siamese method in the following order:
First, the top left quadrant

17	24	1	8	15	0	0	0	0	0
23	5	7	14	16	0	0	0	0	0
4	6	13	20	22	0	0	0	0	0
10	12	19	21	3	0	0	0	0	0
11	18	25	2	9	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

Second, the bottom right quadrant

17	24	1	8	15	0	0	0	0	0
23	5	7	14	16	0	0	0	0	0
4	6	13	20	22	0	0	0	0	0
10	12	19	21	3	0	0	0	0	0
11	18	25	2	9	0	0	0	0	0
0	0	0	0	0	42	49	26	33	40
0	0	0	0	0	48	30	32	39	41
0	0	0	0	0	29	31	38	45	47
0	0	0	0	0	35	37	44	46	28
0	0	0	0	0	36	43	50	27	34

Third, the top right quadrant

17	24	1	8	15	67	74	51	58	65
23	5	7	14	16	73	55	57	64	66
4	6	13	20	22	54	56	63	70	72
10	12	19	21	3	60	62	69	71	53
11	18	25	2	9	61	68	75	52	59
0	0	0	0	0	42	49	26	33	40
0	0	0	0	0	48	30	32	39	41
0	0	0	0	0	29	31	38	45	47
0	0	0	0	0	35	37	44	46	28
0	0	0	0	0	36	43	50	27	34

Finally, the bottom left quadrant

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

Step 3: Go through the upper half of the matrix and swap the appropriate cells with the cells n/2 rows under them except for the cell located at the center row first column of the top left quadrant

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

Output: The output of step 3

2.2. Optimized Algorithm

2.1.1. Description (Rationale)

For the optimized code, we worked to minimize the number of operations for each lesser algorithm by reducing the number of unnecessery swaps and iterations.

<u>Siamese Method (odd n):</u>

Fills the matrix using the Siamese method with simplified row and column updates, eliminating unnecessary temporary variables and redundant checks.

<u>Exchange Method (evenly-even n, n divisible by 4):</u>

Fills the matrix while inverting specific cells using a direct formula, removing the extra loops and swaps of the naive method

<u>Strachey Method (oddly-even n, n divisible by 2 but not 4):</u>

Constructs the magic square by handling half-matrix blocks with systematic swaps and position updates, reducing redundant calculations and simplifying edge adjustments.

2.2.2. Optimized Algorithm Pseudocode

```
FUNCTION magic_square(n)

mat[n][n] ← 0

IF n < 3 THEN
   RETURN 0

ELSE IF n MOD 2 ≠ 0 THEN
   num ← 0
   row ← 0
   column ← n // 2
   cell ← 0

   FOR i ← 0 TO n*n − 1 DO
      cell ← cell + 1
      num ← num + 1
      mat[row][column] ← num

      IF cell = n THEN
         row ← row + 1
         cell ← 0
      ELSE
         row ← row − 1
         column ← column + 1
      END IF

      IF row = −1 THEN
         row ← n − 1
      ELSE IF column = n THEN
         column ← 0
      END IF
   END FOR

ELSE IF n MOD 4 = 0 THEN
   num ← 0
   half ← n // 2
   fourth ← n // 4

   FOR i ← 0 TO n − 1 DO
      FOR j ← 0 TO n − 1 DO
         num ← num + 1

         IF i < half THEN
            IF i < fourth AND (j < fourth OR j ≥ fourth*3) THEN
               mat[i][j] ← (n*n + 1) − num
            ELSE IF i ≥ fourth AND (j ≥ fourth AND j < fourth*3) THEN
               mat[i][j] ← (n*n + 1) − num
            ELSE
               mat[i][j] ← num
            END IF
         ELSE
            IF i ≥ fourth*3 AND (j < fourth OR j ≥ fourth*3) THEN
               mat[i][j] ← (n*n + 1) − num
            ELSE IF i < fourth*3 AND (j ≥ fourth AND j < fourth*3) THEN
               mat[i][j] ← (n*n + 1) − num
            ELSE
               mat[i][j] ← num
            END IF
         END IF
      END FOR
   END FOR

ELSE
   num ← 0
   switch ← 1
   half ← n // 2
   fourth ← n // 4
   begColumn ← fourth

   FOR i ← 0 TO 3 DO
      switch ← NOT switch
      begRow ← half * switch
      cell ← 0

      IF i < 2 THEN
         begColumn ← begColumn + begRow
      ELSE
         begColumn ← begColumn − begRow
      END IF

      row ← begRow
      column ← begColumn
```

```
        FOR j ← 0 TO half*half − 1 DO
            cell ← cell + 1
            num ← num + 1
            mat[row][column] ← num

            IF cell = half THEN
                row ← row + 1
                cell ← 0
            ELSE
                row ← row − 1
                column ← column + 1
            END IF

            IF row = begRow − 1 THEN
                row ← row + half
            ELSE IF column = fourth + begColumn + 1 THEN
                column ← column − half
            END IF
        END FOR
    END FOR

    FOR i ← 0 TO half − 1 DO
        FOR j ← 0 TO fourth − 1 DO
            SWAP mat[i][j] WITH mat[i+half][j]
        END FOR
    END FOR

    SWAP mat[fourth][0] WITH mat[fourth + half][0]
    SWAP mat[fourth][fourth] WITH mat[fourth + half][fourth]

    FOR i ← 0 TO half − 1 DO
        FOR j ← fourth + half + 2 TO n − 1 DO
            SWAP mat[i][j] WITH mat[i+half][j]
        END FOR
    END FOR

END IF

RETURN mat
```
Figure 2.4 (Optimized pseudocode algorithm)

2.2.3. Optimized Algorithm Analysis

```
                                                                    #operations
1. FUNCTION magic_square(n)
2.   mat[n][n] ← 0                                                  $n^2$
3.   IF n < 3 THEN                                                  1
4.     RETURN 0                                                     1
5.   ELSE IF n MOD 2 ≠ 0 THEN                                       2
6.     num ← 0                                                      1
7.     row ← 0                                                      1
8.     column ← n // 2                                              2
9.     cell ← 0                                                     1
10.    FOR i ← 0 TO n*n − 1 DO                                      $n^2 + 1$
11.      cell ← cell + 1                                            $2n^2$
12.      num ← num + 1                                              $2n^2$
13.      mat[row][column] ← num                                     $2n^2$
14.      IF cell = n THEN                                           $n^2$
15.        row ← row + 1                                            $2n^2$
16.        cell ← 0                                                 $n^2$
17.      ELSE
18.        row ← row – 1                                            $2n^2$
19.        column ← column + 1 2                                    $n^2$
20.      END IF
21.      IF row = −1 THEN                                           $n^2$
22.        row ← n – 1                                              $2n^2$
23.      ELSE IF column = n THEN                                    $n^2$
24.        column ← 0                                               $n^2$
25.      END IF
26.    END FOR
27.  ELSE IF n MOD 4 = 0 THEN                                       2
28.    num ← 0                                                      1
29.    half ← n // 2                                                2
30.    fourth ← n // 4                                              2
31.    FOR i ← 0 TO n − 1 DO                                        $n + 1$
32.      FOR j ← 0 TO n − 1 DO                                      $n(n + 1)$
33.        num ← num + 1                                            $2n^2$
34.        IF i < half THEN                                         $n^2$
35.          IF i < fourth AND (j < fourth OR j ≥ fourth*3) THEN    $4n^2$
36.            mat[i][j] ← (n*n + 1) – num                          $5n^2$
37.          ELSE IF i ≥ fourth AND (j ≥ fourth AND j < fourth*3) THEN  $4n^2$
38.            mat[i][j] ← (n*n + 1) – num                          $5n^2$
39.          ELSE
40.            mat[i][j] ← num                                      $2n^2$
41.          END IF
42.        ELSE
43.          IF i ≥ fourth*3 AND (j < fourth OR j ≥ fourth*3) THEN  $5n^2$
44.            mat[i][j] ← (n*n + 1) – num                          $5n^2$
45.          ELSE IF i < fourth*3 AND (j ≥ fourth AND j < fourth*3) THEN  $5n^2$
46.            mat[i][j] ← (n*n + 1) – num                          $5n^2$
47.          ELSE
48.            mat[i][j] ← num                                      $2n^2$
```

```
49. END IF
50. END IF
51. END FOR
52. END FOR
53. ELSE
54. num ← 0                                                    1
55. switch ← 1                                                 1
56. half ← n // 2                                              2
57. fourth ← n // 4                                            2
58. begColumn ← fourth                                         1
59. FOR i ← 0 TO 3 DO                                          5
60. switch ← NOT switch                                        8
61. begRow ← half * switch                                     8
62. cell ← 0                                                   4
63. IF i < 2 THEN                                              4
64. begColumn ← begColumn + begRow                             8
65. ELSE
66. begColumn ← begColumn – begRow                             8
67. END IF
68. row ← begRow                                               4
69. column ← begColumn                                         4
70. FOR j ← 0 TO half*half − 1 DO                  $4half^2 + 1$
71. cell ← cell + 1                                        $8half^2$
72. num ← num + 1                                          $8half^2$
73. mat[row][column] ← num                                 $8half^2$
74. IF cell = half THEN                                    $4half^2$
75. row ← row + 1                                          $8half^2$
76. cell ← 0                                               $4half^2$
77. ELSE
78. row ← row – 1                                          $8half^2$
79. column ← column + 1                                    $8half^2$
80. END IF
81. IF row = begRow − 1 THEN                               $8half^2$
82. row ← row + half                                       $8half^2$
83. ELSE IF column = fourth + begColumn + 1 THEN           $12half^2$
84. column ← column – half                                 $8half^2$
85. END IF
86. END FOR
87. END FOR
88. FOR i ← 0 TO half − 1 DO                                   half + 1
89. FOR j ← 0 TO fourth − 1 DO                         half(fourth + 1)
90. SWAP mat[i][j] WITH mat[i+half][j]                 9half(fourth)
91. END FOR
92. END FOR
93. SWAP mat[fourth][0] WITH mat[fourth + half][0]             9
94. SWAP mat[fourth][fourth] WITH mat[fourth + half][fourth]   9
95. FOR i ← 0 TO half − 1 DO                                   half + 1
96. FOR j ← fourth + half + 2 TO n − 1 DO      (n + 1) – (fourth + half + 2)
97. SWAP mat[i][j] WITH mat[i+half][j]             9(n – (fourth + half + 2))
98. END FOR
99. END FOR
```

```
100. END IF
101. RETURN mat                                                    1
```

Figure 2.5 (Optimized pseudocode algorithm analysis)

2.2.4. Time Complexity Analysis of the Optimized Algorithm

In order to make the algorithm more faster, we focused on minimizing the amount of swaps and iterations and extra variables for each of the lesser algorithms.

Not Magic Square ($n < 3$):

This case simply checks whether n is less than 3 and returns immediately after a few constant-time operations. The only n-dependent work is initializing the n×n matrix, which performs n^2 writes. Therefore, the time complexity is quadratic.

Total number of operations: $n^2 + 2$
Time complexity: $O(n^2)$

Siamese Method (odd n):

The optimized odd-order algorithm simplifies the naive version by removing unnecessary temporary variables like `temp_row` and `temp_column`. It reduces redundant assignments by directly updating `row` and `column` within fewer conditional checks. The logic for wrapping around the matrix edges is merged into simple if-else statements, eliminating multiple separate checks, which make the algorithm a bit more faster.

Total number of operations: $15n^2 + 9$
Time complexity: $O(n^2)$

Exchange Method (evenly-even n, n divisible by 4):

In the optimized doubly even algorithm, we merged the naive version's nested loops and swapping steps into a single assignment using (n*n + 1) − num for inverted cells. This removes the need for extra loops over submatrices and complex swap conditions found in the naive approach. All conditional checks are organized to cover all cases in a more organized way, reducing repeated calculations. As a result, the algorithm is faster.

Total number of operations: $14n^2 + 2n + 9$
Time complexity: $O(n^2)$

Strachey Method (oddly-even n, n divisible by 2 but not 4):

The optimized oddly-even algorithm reduces the naive version's nested loops and excessive temporary variables by directly calculating row and column updates along with the swap pattern. This minimizes redundant swaps by handling specific half-matrix blocks instead of checking each index individually. overall, the algorithm becomes faster due to decrease in operations.

total number of operations: $\frac{73}{4}n^2 + 5n + 55$
Time complexity: $O(n^2)$

Overall Space Complexity:

Even though both algorithms are $O(n^2)$, the optimized version runs faster because it eliminates redundant calculations and unnecessary swaps and variables. It merges conditional checks and directly updates positions, reducing the number of instructions per cell. This makes it more efficient. The overall result is the same, but the optimized algorithm does it with fewer operations.

Total number of operations: $\frac{73}{4}n^2 + 5n + 55$
Time complexity: $O(n^2)$

2.2.5. Space Complexity Analysis of the Optimized Algorithm

By reducing the amount of operations we do, we ultimately also reduce the amount space we occupy during the creation of the magic square.

Siamese Method (odd n):

We use only the $n \times n$ matrix and a few variables for row, column, and counters, eliminating the extra temporary variables present in the naive version, meaning space usage is minimal.

Space complexity: $O(n^2)$.

Exchange Method (evenly-even n, n divisible by 4):

Stores the $n \times n$ matrix and a counter, replacing multiple extra swaps and temporary storage in the naive method, keeping space usage low.

Space complexity: $O(n^2)$.

Strachey Method (oddly-even n, n divisible by 2 but not 4):

Requires $n \times n$ matrix plus a small number of loop and swap variables, avoiding the extra temporary arrays or repeated variables from the naive approach, so space complexity remains $O(n^2)$.

Space complexity: $O(n^2)$.

Overall Time Complexity:

All algorithms are $O(n^2)$ due to the matrix itself, but the optimized versions reduce extra variable usage, lowering constant-space overhead compared to the naive version.

Space complexity: $O(n^2)$.

2.2.6. Algorithm Scenarios:

Best Case:

The best case scenario is still the Siamese method.:

Total number of operations: $15n^2 + 9$
Time complexity: $O(n^2)$

Average Case:

Just like the naïve algorithm albeit with lesser operations, the average case will still be somewhere between the time complexities of the three methods since the order can odd, evenly-even or oddly-even.

Total number of operations: $\frac{63}{4}n^2 + \frac{7}{3}n + 73$
Time complexity: $O(n^2)$

Worst Case:

The heaviest branch is yet again the Strachey (oddly-even n). However, optimizing the number of operations have lessened it and brought it closer to the best case scenario.

Total number of operations: $\frac{73}{4}n^2 + 5n + 55$
Time complexity: $O(n^2)$

Summary:

Best Case: $O(n^2)$
Average Case: $O(n^2)$
Worst Case: $O(n^2)$

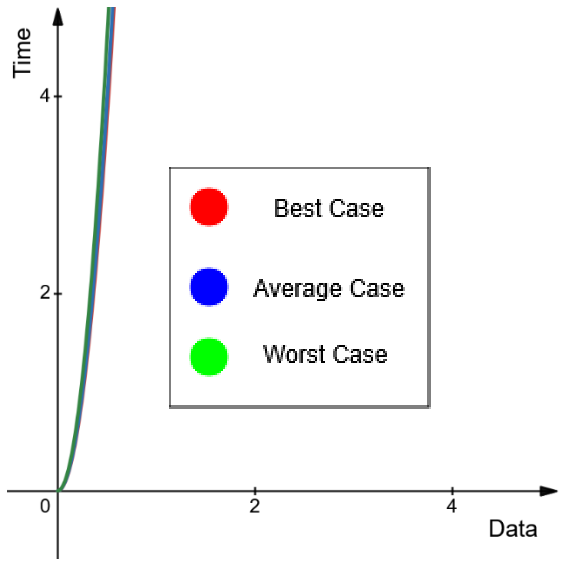


Figure 2.6 (Optimized time complexity graph)

2.2.7. Optimized Algorithm Examples

<u>Siamese Method:</u>

Input: n = 3

Step 1: Create an empty matrix of size n^2

0	0	0
0	0	0
0	0	0

Step 2: Go through the matrix diagonally and plot the numbers 1→9. wrap around when hitting an edge and move down once when hitting an occupied cell

8	1	6
3	5	7
4	9	2

Output: The output of step 2

<u>Exchange Method:</u>

Input: n = 4

Step 1: Create an empty matrix of size n^2

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

Step 2: Go through the matrix in order. we merge steps 2 and 3 of the naïve and enter the correct values automatically via merged conditions.

16	2	3	13
5	11	10	8
9	7	6	12
4	14	15	1

Output: The output of step 2

Strachey Method:
Input: n = 10

Step 1: Create an empty matrix of size n^2

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

Step 2: Fill each of the four quadrants sequentially via the Siamese method in the following order:
First, the top left quadrant

17	24	1	8	15	0	0	0	0	0
23	5	7	14	16	0	0	0	0	0
4	6	13	20	22	0	0	0	0	0
10	12	19	21	3	0	0	0	0	0
11	18	25	2	9	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

Second, the bottom right quadrant

17	24	1	8	15	0	0	0	0	0
23	5	7	14	16	0	0	0	0	0
4	6	13	20	22	0	0	0	0	0
10	12	19	21	3	0	0	0	0	0
11	18	25	2	9	0	0	0	0	0
0	0	0	0	0	42	49	26	33	40
0	0	0	0	0	48	30	32	39	41
0	0	0	0	0	29	31	38	45	47
0	0	0	0	0	35	37	44	46	28
0	0	0	0	0	36	43	50	27	34

Third, the top right quadrant

17	24	1	8	15	67	74	51	58	65
23	5	7	14	16	73	55	57	64	66
4	6	13	20	22	54	56	63	70	72
10	12	19	21	3	60	62	69	71	53
11	18	25	2	9	61	68	75	52	59
0	0	0	0	0	42	49	26	33	40
0	0	0	0	0	48	30	32	39	41
0	0	0	0	0	29	31	38	45	47
0	0	0	0	0	35	37	44	46	28
0	0	0	0	0	36	43	50	27	34

Finally, the bottom left quadrant

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

Step 3: Go through the first two columns of the top left quadrant and swap them with the cells n/2 rows under them

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

Step 4: Reverse swap the element located in the center row first column of the top left quadrant

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

Step 5: Swap the center element of the top left quadrant with the cell n/2 rows under it

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

Step 6: Swap the rightmost columns two columns after the center column of the top right quadrant with the cells n/2 rows under them

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

Output: the output of step 6

2.3. Theoretical Analysis: Comparison

Naïve Algorithm:

Total number of operations: $42n^2 + n + 60$

Time complexity: $O(n^2)$, where n is the order of the magic square

Space complexity: $O(n^2)$, where n is the order of the magic square

Steps:

- Create empty matrix
- Check the order
- If less than 3, return 0
- If odd, implement the Siamese method
  - Fill the matrix by placing numbers sequentially while moving up-right with wrap-around
  - If the next cell is occupied or the movement cycle completes, move down instead
  - Continue until all n^2 cells are filled
- If evenly-even, fill the matrix sequentially from 1 to n2n^2n2
  - Identify the patterned cells in the corner and center regions
  - Replace each patterned value with $(n^2 + 1) - value$
  - Leave all other cells unchanged
- If oddly-even, divide the matrix into four quadrants
  - Fill each quadrant using the odd-order method with the correct numerical offsets
  - Perform the required cross-swaps between quadrants to align rows and columns
  - Apply the special central-row swaps to complete the structure
- Return the magic square

Advantages: The naive algorithm is simple and works for all square types.

Disadvantages: It performs extra steps that make it slower and less efficient.

Optimised Algorithm:

Total number of operations: : $\frac{73}{4}n^2 + 5n + 55$, where n is the order of square.

Time complexity: $O(n^2)$, where n is the order of the magic square

Space complexity: $O(n^2)$, where n is the order of the magic square

Steps:

- Create an empty matrix
- Check the order
- If less than 3, return 0
- If odd, use the streamlined Siamese method
  - Place numbers while moving up-right with wrap-around using fewer checks
  - Move down only when a row segment completes, reducing condition handling
  - Continue until the matrix is filled
- If evenly-even, iterate from 1 to n^2 and decide each cell's value without swaps
  - Use direct pattern checks to assign either the number or its complement $(n^2 + 1 - num)$

    - Apply this logic in one pass without extra loops
- If singly even, divide the square into four quadrants
    - Fill each quadrant using the streamlined odd-order method with offsets
    - Apply the required swaps using fewer conditions and no nested pattern checks
    - Handle the special row swaps directly with fixed positions
- Return the optimized magic square

Advantages: The optimized algorithm reduces unnecessary checks and swaps, making each case (odd, evenly-even, oddly-even) run faster and more efficiently.

Disadvantages: It is more complex and harder to implement or understand due to its tighter conditions and specialized shortcuts

<u>Summary of Comparisons;</u>

The naive algorithm is simple and easy to follow but performs many unnecessery operations, making it slower. The optimized algorithm achieves the same results with fewer steps, merged checks, and direct calculations, making it faster and more efficient. Both have the same $O(n^2)$ complexity, but the optimized version runs better in practice.

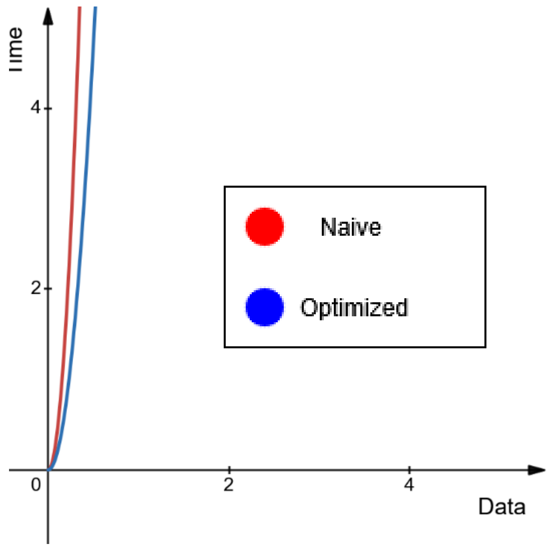


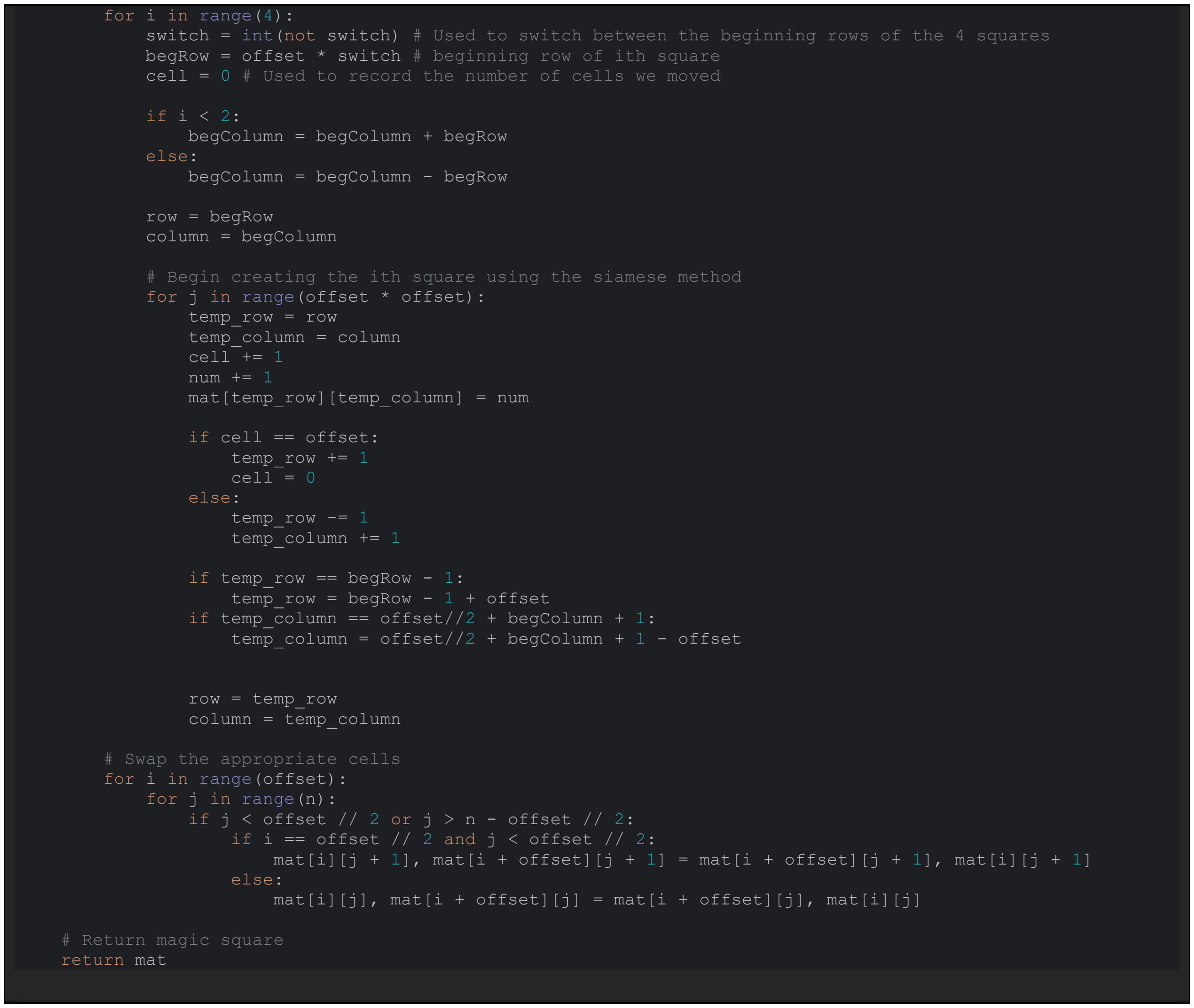
Figure 2.7 (Naive vs Optimized time complexity graph)

3. Empirical Analysis

3.1. Naïve Algorithm Implementation

Here is the implementation of the naïve algorithm in python. it checks the type of number the order is and performs the necessary algorithm based on it is odd, evenly-even or oddly-even.

```python
# Magic square naive algorithm
def magic_square(n):
    mat = [[0 for _ in range(n)] for _ in range(n)]
    num = 0

    # Check is order is less than 3. if so, return 0 as there are no non-trivial magic squares of an order
less than 3
    if n < 3:
        return 0
        # Siamese Method for magic squares of odd order
    elif n % 2 != 0:
        row = 0
        column = n // 2
        cell = 0

        # Assign numbers throughout the matrix via diagonal movement
        for i in range(n * n):
            temp_row = row
            temp_column = column
            cell += 1
            num += 1
            mat[temp_row][temp_column] = num

            # If we've moved n number of times, we need to go down. Otherwise, move diagonally
            if cell == n:
                temp_row += 1
                cell = 0
            else:
                temp_row -= 1
                temp_column += 1

            # If we hit the upper or right edge, we wrap around the matrix accordingly
            if temp_row == -1:
                temp_row = n - 1

            if temp_column == n:
                temp_column = 0


            row = temp_row
            column = temp_column
    # Exchange Method for magic squares of evenly even order
    elif (n//2)%2 == 0:
        offset = n//4 # Order of all partitioned normal squares. Used for moving to different normal squares

        # Fill the Magic Square with consecutive numbers from first element to last element
        for i in range(n):
            for j in range(n):
                num += 1
                mat[i][j] = num


        # Switch the elements in the two top corner odd squares and two top central odd squares with their
complementary squares
        for row in range(n//2):
            for column in range(n):
                if row < offset:
                    if (column < offset) or (column > offset*3 - 1):
                        mat[row][column], mat[n - 1 - row][n - 1 - column] = \
                            mat[n - 1 - row][n - 1 - column], mat[row][column]
                else:
                    if (column > offset - 1) and (column < offset*3):
                        mat[row][column], mat[n - 1 - row][n - 1 - column] = \
                            mat[n - 1 - row][n - 1 - column], mat[row][column]
    # Strachey Method for Oddly Even Magic Squares
    elif (n//2)%2 != 0:
        switch = 1 # Switch between the partitioned odd magic squares via "not" keyword
        offset = n//2 # Order of all the partitioned odd magic squares. Used for moving to different odd
ordered magic squares
        begColumn = offset//2 # Used to pinpoint the beginning column of ith magic square

        # Create 4 odd magic squares inside the oddly even magic square using the siamese method
```



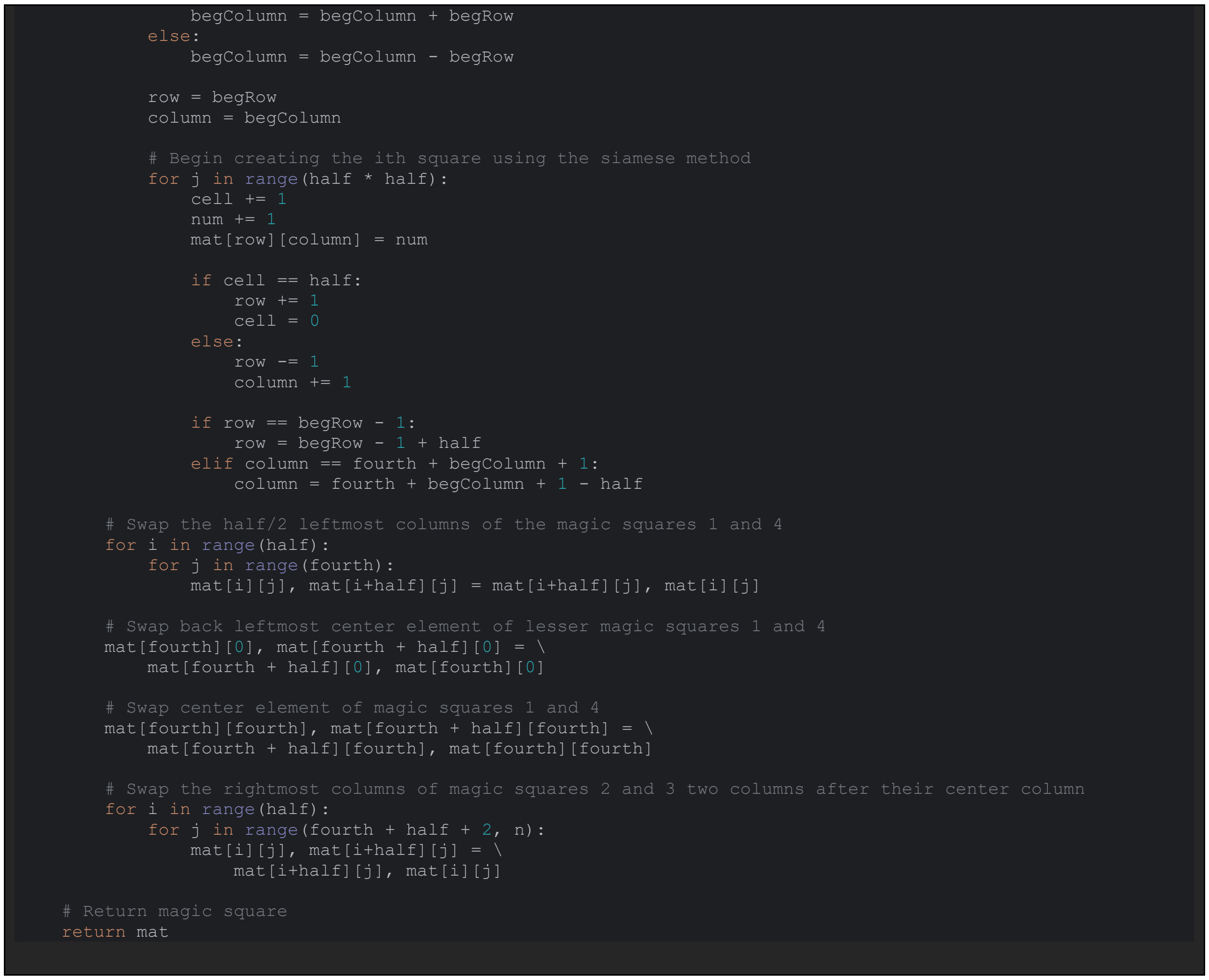
```
        for i in range(4):
            switch = int(not switch) # Used to switch between the beginning rows of the 4 squares
            begRow = offset * switch # beginning row of ith square
            cell = 0 # Used to record the number of cells we moved

            if i < 2:
                begColumn = begColumn + begRow
            else:
                begColumn = begColumn - begRow

            row = begRow
            column = begColumn

            # Begin creating the ith square using the siamese method
            for j in range(offset * offset):
                temp_row = row
                temp_column = column
                cell += 1
                num += 1
                mat[temp_row][temp_column] = num

                if cell == offset:
                    temp_row += 1
                    cell = 0
                else:
                    temp_row -= 1
                    temp_column += 1

                if temp_row == begRow - 1:
                    temp_row = begRow - 1 + offset
                if temp_column == offset//2 + begColumn + 1:
                    temp_column = offset//2 + begColumn + 1 - offset


                row = temp_row
                column = temp_column

        # Swap the appropriate cells
        for i in range(offset):
            for j in range(n):
                if j < offset // 2 or j > n - offset // 2:
                    if i == offset // 2 and j < offset // 2:
                        mat[i][j + 1], mat[i + offset][j + 1] = mat[i + offset][j + 1], mat[i][j + 1]
                    else:
                        mat[i][j], mat[i + offset][j] = mat[i + offset][j], mat[i][j]

    # Return magic square
    return mat
```

Figure 3.2 (Naïve algorithm python code)

3.2. Optimized Algorithm Implementation

Below is the implementation of the Optimized algorithm. By merging conditions to reduce unnecessary swaps and iterations, we improve the overall speed and efficiency of the algorithm.

```
# Magic square optimized algorithm
def magic_square(n):
    mat = [[0 for _ in range(n)] for _ in range(n)]

    # Check is order is less than 3. if so, return 0 as there are no non-trivial magic squares of an order
less than 3
    if n < 3:
        return 0
    # Siamese Method for magic squares of odd order
    elif n%2 != 0:
        num = 0
        row = 0
        column = n//2
        cell = 0

        # Assign numbers throughout the matrix via diagonal movement
        for i in range(n*n):
            cell += 1
            num += 1
            mat[row][column] = num

            # If we've moved n number of times, we need to go down. Otherwise, move diagonally
            if cell == n:
                row += 1
                cell = 0
            else:
                row -= 1
                column += 1

            # If we hit the upper or right edge, we wrap around the matrix accordingly
            if row == -1:
                row = n - 1
            elif column == n:
                column = 0
    # Exchange Method for magic squares of evenly even order
    elif n%4 == 0:
        num = 0
        half = n//2
        fourth = n//4

        # Fill each cell a number according to its coordinates within the matrix
        for i in range(n):
            for j in range(n):
                num += 1

                if i < half:
                    if i < fourth and (j < fourth or j >= fourth * 3):
                        mat[i][j] = (n * n + 1) - num
                    elif i >= fourth and (fourth <= j < fourth * 3):
                        mat[i][j] = (n * n + 1) - num
                    else:
                        mat[i][j] = num
                else:
                    if i >= fourth * 3 and (j < fourth or j >= fourth * 3):
                        mat[i][j] = (n * n + 1) - num
                    elif i < fourth * 3 and (fourth <= j < fourth * 3):
                        mat[i][j] = (n * n + 1) - num
                    else:
                        mat[i][j] = num
    # Strachey Method for Oddly Even Magic Squares
    else:
        num = 0
        switch = 1 # Switch between the partitioned odd magic squares via "not" keyword
        half = n//2 # Order of all the partitioned odd magic squares. Used for moving to different odd
ordered magic squares
        fourth = n//4
        begColumn = fourth # Used to pinpoint the beginning column of ith magic square

        # Create 4 odd magic squares inside the oddly even magic square using the siamese method
        for i in range(4):
            switch = int(not switch) # Used to switch between the beginning rows of the 4 squares
            begRow = half * switch # beginning row ith square
            cell = 0 # Used to record the number of cells we moved

            if i < 2:
```



```python
                begColumn = begColumn + begRow
            else:
                begColumn = begColumn - begRow

            row = begRow
            column = begColumn

            # Begin creating the ith square using the siamese method
            for j in range(half * half):
                cell += 1
                num += 1
                mat[row][column] = num

                if cell == half:
                    row += 1
                    cell = 0
                else:
                    row -= 1
                    column += 1

                if row == begRow - 1:
                    row = begRow - 1 + half
                elif column == fourth + begColumn + 1:
                    column = fourth + begColumn + 1 - half

        # Swap the half/2 leftmost columns of the magic squares 1 and 4
        for i in range(half):
            for j in range(fourth):
                mat[i][j], mat[i+half][j] = mat[i+half][j], mat[i][j]

        # Swap back leftmost center element of lesser magic squares 1 and 4
        mat[fourth][0], mat[fourth + half][0] = \
            mat[fourth + half][0], mat[fourth][0]

        # Swap center element of magic squares 1 and 4
        mat[fourth][fourth], mat[fourth + half][fourth] = \
            mat[fourth + half][fourth], mat[fourth][fourth]

        # Swap the rightmost columns of magic squares 2 and 3 two columns after their center column
        for i in range(half):
            for j in range(fourth + half + 2, n):
                mat[i][j], mat[i+half][j] = \
                    mat[i+half][j], mat[i][j]

    # Return magic square
    return mat
```

Figure 3.2 (Optimized algorithm python code)

3.3. Performance Analysis

To evaluate the performance of both the naive and optimized algorithms, we ran a series of tests using different input sizes. since the input is the order of the magic square, the size of the input is squared, and we recorded the time taken by each algorithm to create a magic square of the same order:

Order Size (Squared)	Naïve (ms)	Optimized (ms)
Small – 55	0.508	0.348
Medium – 306	28.96	17.77
Large - 4792	3593.40	2640.82

Note: The orders are squared in the algorithm (Example: $55^2 = 3025$)

The Results clearly demonstrate that the optimized algorithm is faster than the naïve thanks to the merging of conditions. The optimized algorithm, which leverages the consolidations of rules and splitting of some rules, scales pretty efficiently compared to the naïve algorithm.

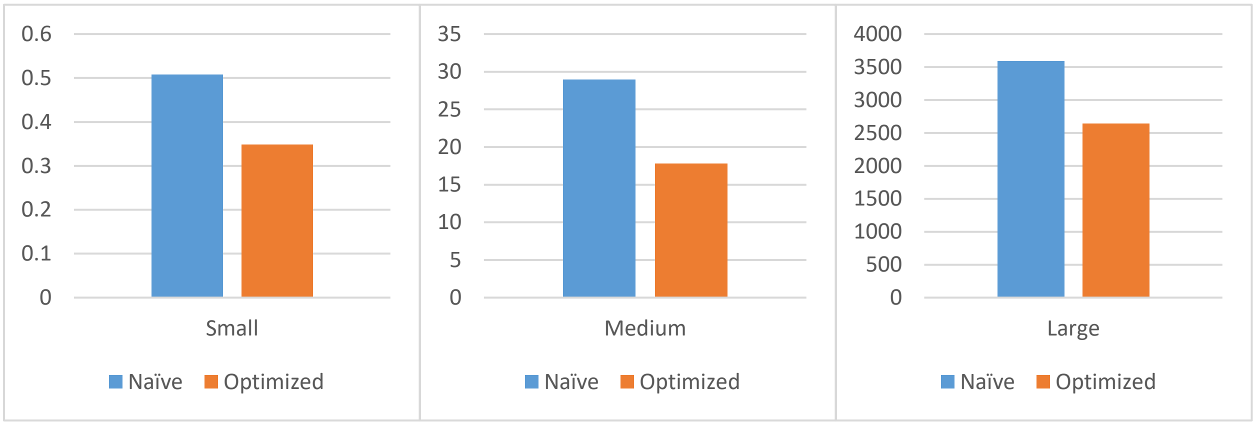


Figure 3.3 (Execution time charts)

4. Conclusion

4.1. Comparison

Theoretical Analysis suggests that the naïve and optimized algorithms have the exact same time complexity of $O(n\char`^2)$ and yet, the naïve is slower than the optimized. This is due to another thing that we discover during the theoretical analysis, which is that the total number of operations of the optimized algorithm is less than the number of operations of the naïve algorithm.

For example:

At an order of 55, the naïve took .508 ms to create the magic square while the optimized took .348 ms. At an order of 306, the naïve took 28.96 ms while the optimized took 17.77 ms. At an order of 4792, the naïve took 3593.40 ms while the optimized took 2640.82 ms.

Keep in mind that the data being handled during the creation of the magic square is the order squared. so when the order is 55, the data being handled is 55^2 = 3025, and an order of 306 is 306^2 = 93636!

While there might be some discrepancies between the theoretical and empirical analysis, this is mainly due to hardware. Overall, both the empirical and the theoretical analyses show that the optimized algorithm is faster than the naïve.

4.2. Final Thoughts

In this project, we implemented, analyzed and improved three different algorithms for creating different order magic squares: The Siamese Method, The Exchange Method and The Strachey Method. Our project shows that even though both the naïve algorithm and optimized algorithm have the same time complexity of $O(n^2)$, the optimized algorithm ran faster because it has fewer operations than the naïve.

Our Empirical analysis proves this even further by showing that the optimized algorithm created a magic square faster than the naïve algorithm. Overall, this project shows that even if you can't optimize the time complexity of an algorithm, you can still optimize the number of operations and expect the algorithm to run more efficiently.

Sources

1. Tutorial of building an odd ordered magic square using Siamese Method.

2. Tutorial of building an evenly even ordered magic square using Exchange Method.

3. Tutorial of building an oddly even ordered magic square using Strachey Method.

4. Miller, Jeff: Like many in his time, Benjamin Franklin built magic squares to alleviate his boredom.

5. Python documentation for writing the python code.

6. Course materials.