Sudoku Solver Algorithm with Depth-First Search

Data Structures Final Report

Almira Farahiyah Shafiqa Rana (2702378956)

Abyan Ali Kartasasmita (2702355385)

Rayan Ferhat Al Jaidi (2702426444)

Lecturer: Nunung Nurul Qomariyah, S.Kom., M.T.I., Ph.D.

Binus International University

Jakarta

Table of Contents

1. Background

Sudoku is a popular number-placement puzzle with origins from the 18th century. It is appreciated among the multitudes of people throughout the world because of its logic-based difficulties (Mehranfar, 2024). In general, Sudoku consists of a 9x9 grid as seen in Figure 1, where each square must be filled with the numbers 1 to 9 without any repetition horizontally or vertically. The puzzle is solved by a well-defined method, reflecting the technique used for other issues in mathematics and computer science, where every problem has a solution.

Figure 1

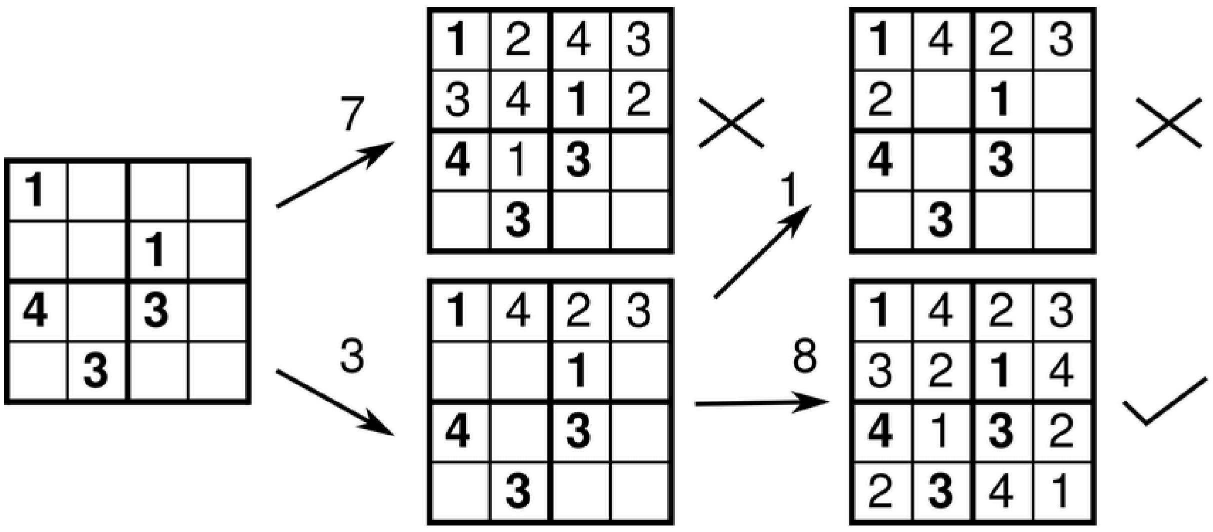
Basic Sudoku Board

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

According to Pauli (n.d.) an algorithm is a finite sequence of instructions for performing a task. Since Sudoku does not require any moves or steps, it will instead output a finished sudoku board. The data structure of a piece of software defines how data flows in relation to inputs, processes, and outputs. A data structure may arrange, process, retrieve, and store data, allowing users to easily access and interact with the data they require while also structuring the organization of information within a software so that both people and machines can better comprehend it (Brooks, 2023). Based on Berggren and Nilsson (2012), when choosing algorithms for solving Sudoku puzzles, one important factor to consider is how they investigate potential solutions. There are two basic approaches: deterministic techniques, which include backtracking and rule-based algorithms, and stochastic methods, which include genetic algorithms and Boltzmann machines. Deterministic approaches adhere to a specified set of principles and methodically investigate all potential solutions, resulting in a predictable path to the answer. In contrast, stochastic approaches produce candidate solutions and construct the search route using random processes, providing flexibility but no solid assurances about the time required to discover a solution. Backtrack, also known as depth-first search, is one of the most basic Sudoku solving strategies for computer algorithms seen in Figure 2. This algorithm is a brute-force approach that attempts several integers and, if unsuccessful, returns to try another number.

Figure 2

Sudoku Backtrack Algorithm



2. Problem Description

Sudoku will be analyzed to understand the solving algorithm with a depth-first search approach. Sudoku puzzles are difficult to solve because of the structure of reliance between these stages between steps. Furthermore, the interconnectivity of these processes, where the use of one method has a major influence on the use of succeeding approaches, adds to the puzzle's intricacy. This dependence can make it difficult to solve the puzzle efficiently.

Furthermore, human problem-solving tactics, like the use of redundancy, which allows for many solutions, can affect the difficulty of a Sudoku puzzle, making it more difficult for both people and machines to solve (Pelánek, 2014). Findings by Ekström and Pitkäjärv (2014) state that computers struggle to solve Sudoku puzzles effectively using a backtracking method for a variety of reasons. The algorithm's high temporal complexity, particularly for bigger issue cases, poses a significant restriction. Additionally, embedded devices with limited memory may face memory constraints while storing the states and all alternative solutions. These considerations make it difficult to create algorithms that can solve sudoku problems rapidly and efficiently.

This report aims to develop an algorithm with backtracking also known as depth-first searching, using several different types of data structures namely, 2D Arrays, HashMaps, LinkedLists, and Stacks to efficiently solve a Sudoku puzzle, by calculating the time complexity, space complexity, and benchmarking the overall performance. Practical implementations will be compared with one another through the programming language Java.

The initial hypothesis for the findings of this research is that the most inefficient data structure overall will be LinkedLists, as most of their functions that would possibly be used for the solver have high time complexities and space complexities compared to other data structures. The estimated most efficient data structure should be 2D arrays, as they are the most structurally sound and have constant complexities in most of their functions.

3. Solution

All the data structures were implemented using the same basic depth-first search solving logic. The cells themselves are also accessed in the way that is unique to each data structure, to obtain the most accurate time complexities, space complexities, and real-time taken to solve it.

3.1. Methodology

Static Variables

The program declares several static variables to store the Sudoku board, the GUI components, and the solver's state. These variables include:

- SIZE: The size of the Sudoku board.
- board: To store the Sudoku puzzle.
- operationCount: A counter for the number of operations performed during the solving process.
- inputSize: The size of the input.

- filledCellsCount: A counter for the number of filled cells in the Sudoku board.
- min_clues: The minimum number of clues required to solve the Sudoku puzzle.
- timer: A Timer object to measure the time taken to solve the puzzle.
- startTime: The start time of the timer.
- timerLabel: A JLabel object to display the elapsed time.

Main Method

The main method is the entry point of the program. It creates a GUI with a Sudoku board, a "Solve" button, and a "Clear" button.

1. The program prompts the user to select the size of the Sudoku board.
2. Based on the user's selection, the program sets the SIZE variable and initializes the board.
3. The program creates a JPanel and sets its layout to GridLayout with the same number of rows and columns as the Sudoku board.
4. The program creates a "Solve" button and adds an ActionListener to it. When the button is clicked, the program will parse the input from the GUI components, check for duplicates, and solve the Sudoku puzzle using the solve method.
5. The program creates a "Clear" button and adds an ActionListener to it. When the button is clicked, the program will clear the GUI and the board.

Solver Methods

The program provides several methods to solve the Sudoku puzzle:

- parseInput: Parses the input from the GUI components and stores it in the board data structure.
- updateBoard: Updates the GUI with the solution by setting the text of each GUI component to the corresponding value in the board data structure.
- clearBoard: Clears the GUI and the board by resetting the board data structure and the GUI components.
- solve: The recursive method that solves the Sudoku puzzle.
    - If the filledCellsCount is less than the minimum number of clues, the method returns false.
    - If the current cell is already filled, the method moves to the next cell.

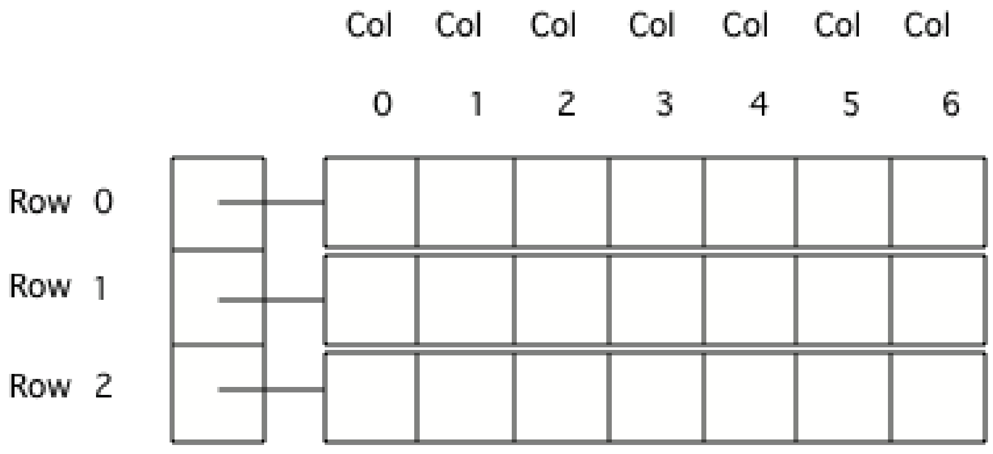
    - The method tries numbers from 1 to SIZE in the current cell and checks if the value is legal using the legal method.
    - If the value is legal, the method recursively calls itself with the next cell.
    - If no value can be placed in the current cell, the method returns false.
- legal: Checks if a value can be placed in a cell. It checks the row, column, and box for duplicates and returns false if a duplicate is found.
- hasDuplicates: Checks if there are any duplicates in the Sudoku board. It checks rows, columns, and boxes for duplicates and returns true if a duplicate is found.

3.2. Two-dimensional (2D) Arrays

2D Arrays in Java stores items of the same type, with rows and columns. A row has horizontal elements. A column has vertical elements. Java actually stores two-dimensional arrays as arrays of arrays. Each element of the outer array has a reference to each inner array. When accessing the element at *arr[first][second]*, the first index is used for rows, the second index is used for columns (Ericson, 2015).

Figure 3

Visualization of a 2D Array



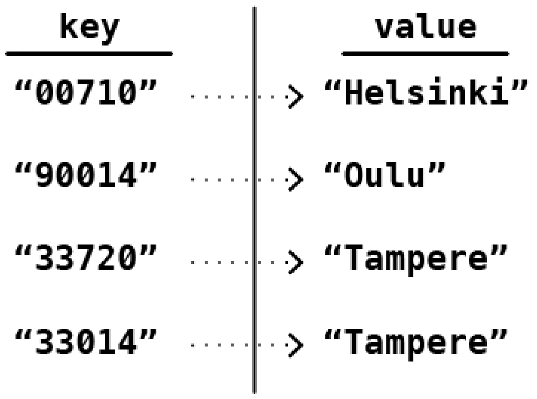
A 2D array makes it simple to check for valid solutions by iterating over the rows, columns, and subgrids. Storing the sudoku grid as a 2D array can be memory-intensive, since each cell requires memory allocation. These are static structures with fixed dimensions which makes it hard to implement some advanced algorithms that require dynamic resizing. The solution logic often begins in the top-left corner of the board and progresses row by row, testing each cell's legality against the Sudoku rules. If a cell is empty, attempt inserting legitimate numbers (1 to 9) and iteratively exploring each option until a solution is discovered.

3.3. HashMap

HashMaps store data as key-value pairs, where values can be added, retrieved, and deleted using keys, with a maximum of one value per key. If a new key-value pair is added to the hash map, but the key has already been associated with some other value stored in the hash map, the old value will vanish from the hash map.

Figure 4

Visualization of HashMap



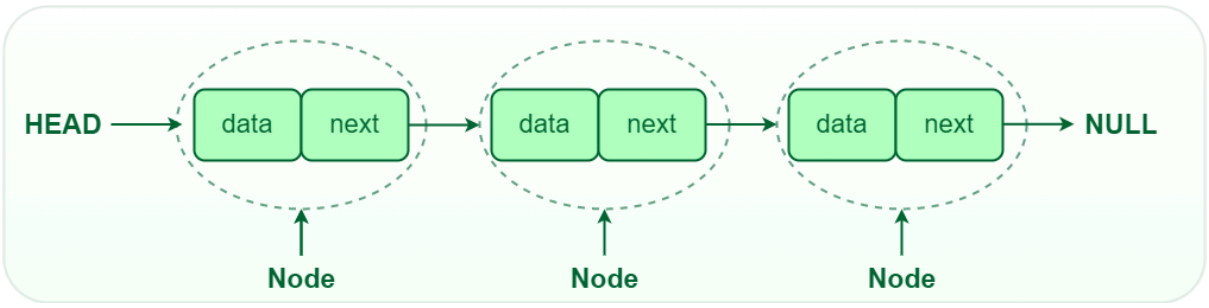
HashMaps are very valuable to the implementation of a Sudoku solver due to their insertion, deletion, and lookup operations yielding constant time complexity, which is significant (Agile Education Research, 2017). However, Hashmaps can be more complex compared to using arrays or matrices,slower lookup times compared to direct array access (noticeable in larger Sudoku grids). Hash collisions can occur when there are two different keys that are mapped to the same value and with that, a collision can easily occur. The HashMap is implemented in a similar way to the Array approach, though offers more flexibility for the board size and cell access.

3.4. LinkedList

A linked list is a linear data structure in which elements are not stored in contiguous locations but are linked together via pointers. A Linked List is a series of connected nodes, with each node storing data and the address of the next node (GeeksforGeeks, 2024).

Figure 5

Visualization of LinkedList



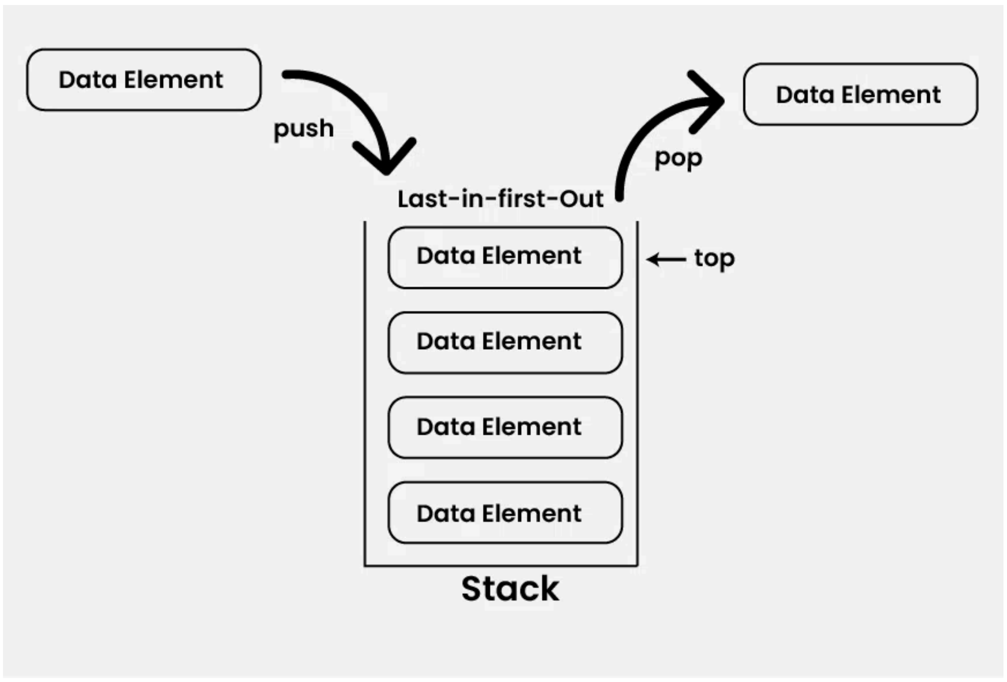
Sudoku solvers frequently employ linked lists to efficiently manage the puzzle's restrictions and candidates. The backtracking technique, for example, represents the restrictions and candidates as a linked list. This strategy enables quick exploration and modification of the limitations, making the puzzle easier to solve. For the linked list implementation, a custom linked list node would be built for each cell on the Sudoku board, with each node containing the cell's value and a reference to the next node in the list. The solution mechanism would go through these linked list nodes, verifying legality and backtracking as necessary. The expense of visiting linked list nodes makes this technique less efficient for Sudoku solution than direct array or hashmap access.

3.5. Stack

Stack is a linear data structure based on the LIFO (Last In First Out) principle, in which new elements are inserted and existing elements are removed at the same end of the stack, which is represented as the top (GeeksforGeeks, 2024b).

Figure 6

Visualization of Stacks



When completing Sudoku using a stack, a stack of states are kept track of that represent various board configurations and locations. The solution logic begins with the starting state and examines potential steps by adding new states to the stack. If the algorithm reaches a dead end (no viable movements), it returns by popping states from the stack and tries various pathways until a solution is discovered or all paths have been exhausted. This backtracking strategy is critical for fast solving Sudoku since it allows the algorithm to investigate alternative options and retrace when necessary, guaranteeing a valid solution is found without an exhaustive search.

4. Complexity Analysis

Algorithm complexity refers to the amount of resources (such as time or memory) needed to solve a problem or complete a job (GeeksforGeeks, 2023). This section of the report will analyze the time and space complexity of the overall program. Taken into account are the types of data structures used, size of input, and number of operations.

4.1. Time Complexity

The time complexity of a program refers to the amount of time it takes to complete its task, typically measured by the number of operations performed in relation to the size of the input. In the context of this project, we will analyze the worst-case scenario using the big-O notation and examine the overall computational complexity.

The input size of the program is influenced by both the dimensions of the Sudoku board and the number of empty squares within it. Given the constraints of a Sudoku puzzle, which require a minimum number of clues based on the board size, we must consider the smallest possible input. For instance, a 3x3 Sudoku board necessitates at least 4 clues, while a 9x9 board requires at least 17. This relationship between input size and the number of clues already provides insight into the potential time complexity, which can be represented as n^m, where n is the board size and m is the number of empty squares. As the input size increases, so does the number of operations.

The number of operations is primarily counted within the recursive function of the code, as this is where the worst-case scenario typically arises. To visualize the time complexity, we created a table and graph that illustrate the relationship between input size and the number of operations. There are 6 difficulty levels for each of the board sizes, which can be seen in the tables below. The difficulty rises depending on how many hints are available, whilst following the rule that the minimum number of clues for a 3x3 is 4 and the for a 9x9 is 17.

Table 1

Data for 3x3 Board

Number of Available Hints	Number of Empty Cells	Number of Operations			
		2D Array	HashMap	LinkedList	Stack
4	5	65	71	68	68
5	4	40	43	43	43
6	3	28	35	35	35
7	2	17	18	18	18
8	1	10	10	10	10
9	0	0	0	0	0

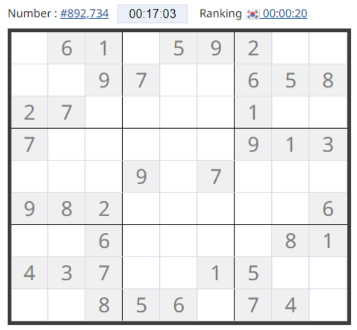
Table 2

Data for 9x9 Board

Number of Available Hints	Number of Empty Cells	Number of Operations			
		2D Array	HashMap	LinkedList	Stack
17	64	463228946	446527569	37286067	37286067
20	61	66207302	41661679	23293839	23293839
25	56	2705690	2978526	157051	157051
27	54	46322	252690	148144	148144
29	52	116566	120909	39122	39122
36	45	26657	25821	12015	12015

For each difficulty level in the 9x9 board, each puzzle was standardized as seen below.

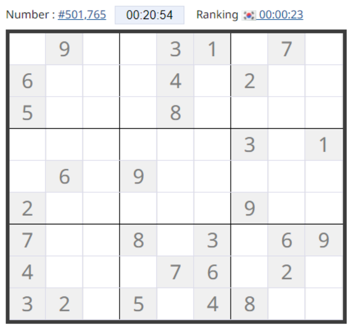
Level 1 (Easy : 36 hint)



Number : #892,734 00:17:03 Ranking 00:00:20

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

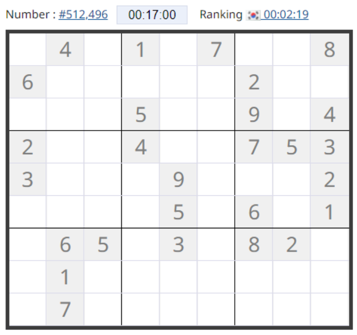
Level 8 (Medium : 29 hint)



Number : #501,765 00:20:54 Ranking 00:00:23

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

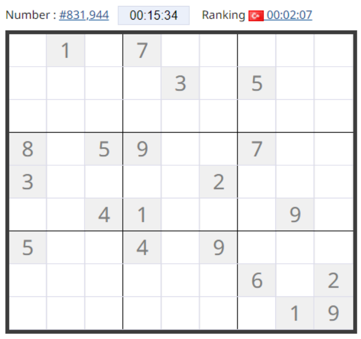
Level 10 (Hard : 27 hint)



Number : #512,496 00:17:00 Ranking 00:02:19

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

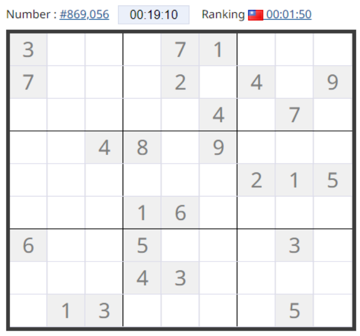
Level 17 (Expert - Silver : 20 hint)



Number : #831,944 00:15:34 Ranking 00:02:07

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

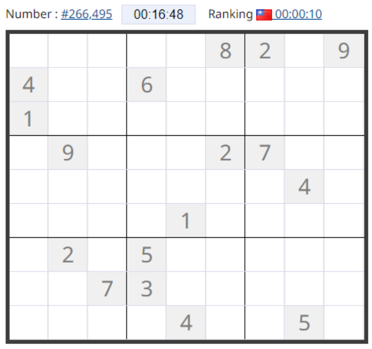
Level 12 (Expert : 25 hint)



Number : #869,056 00:19:10 Ranking 00:01:50

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

Level 20 (Expert - Gold : 17 hint)



Number : #266,495 00:16:48 Ranking 00:00:10

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

After standardizing the input, the data can be seen through linear graphs, where it becomes apparent which data structure is the most efficient and least efficient.

Figure 7

Time Complexity Graph Comparison in 3x3 Board

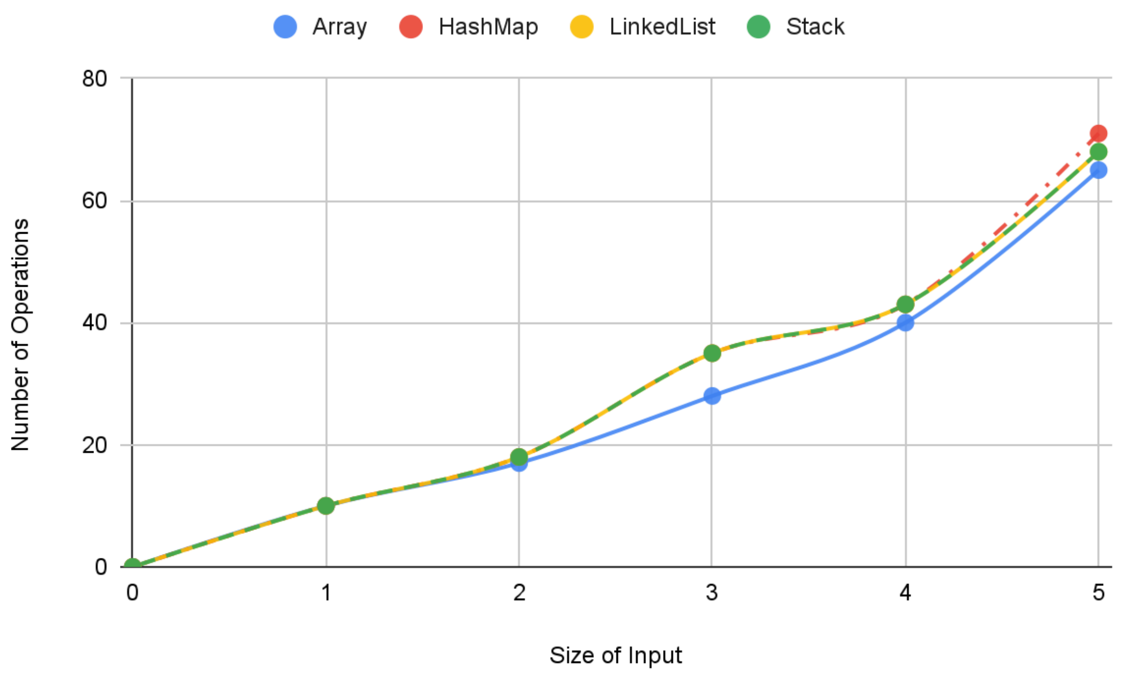


Figure 8

Time Complexity Graph Comparison in 9x9 Board

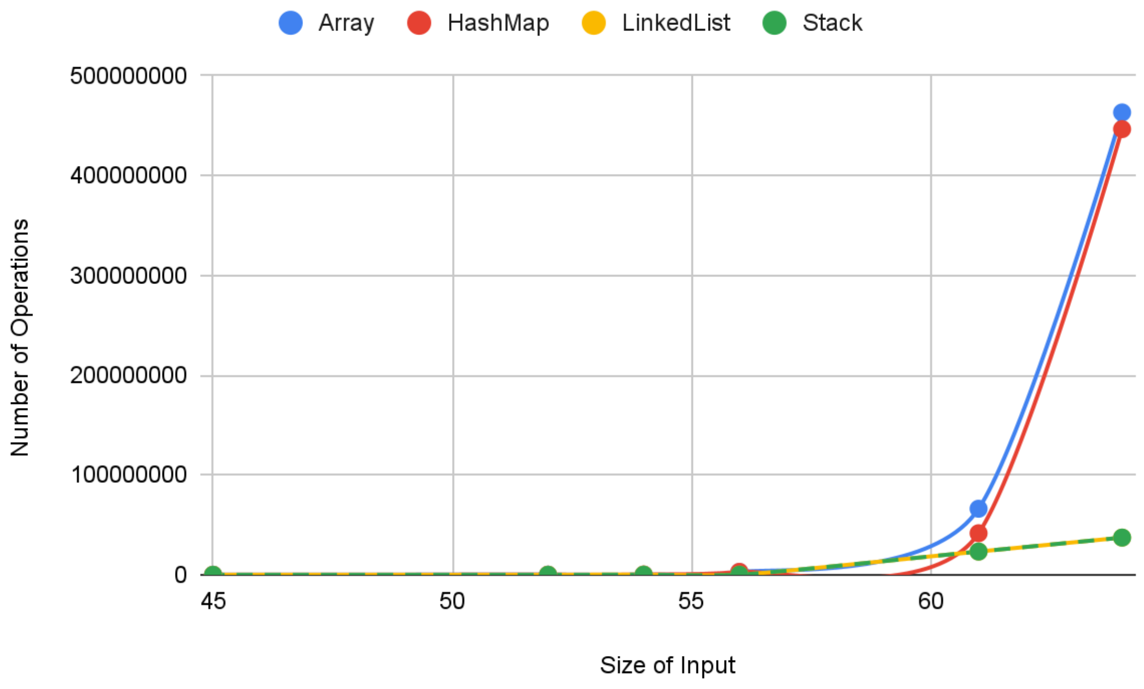


Figure 9

Comparison of Time Taken for Each Data Structure to Solve a 3x3 Sudoku

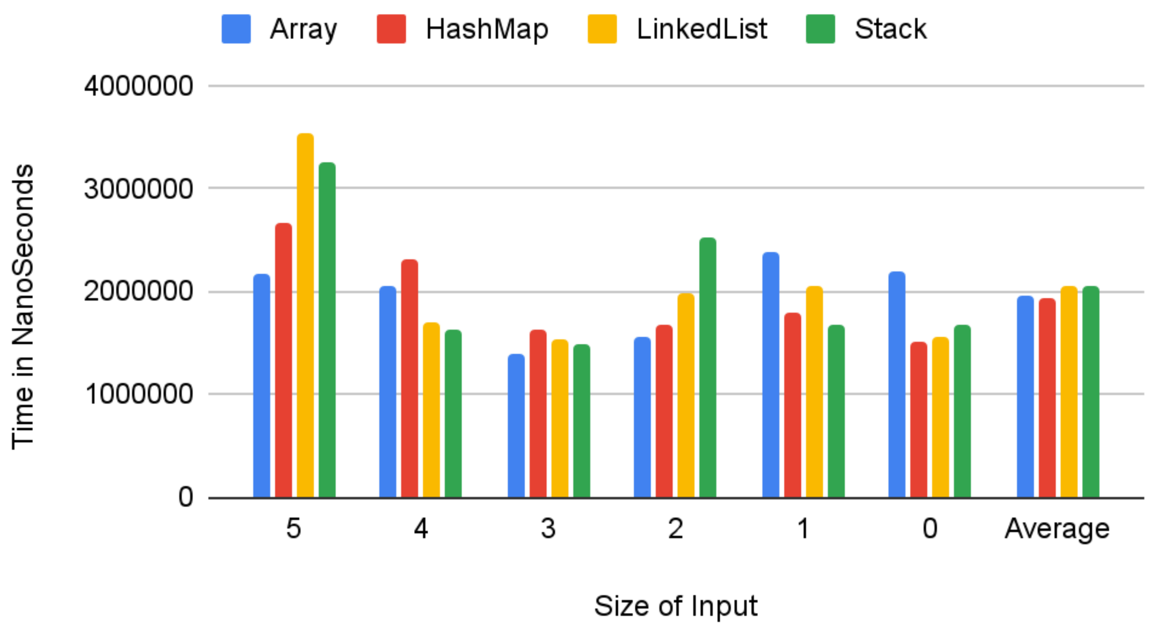
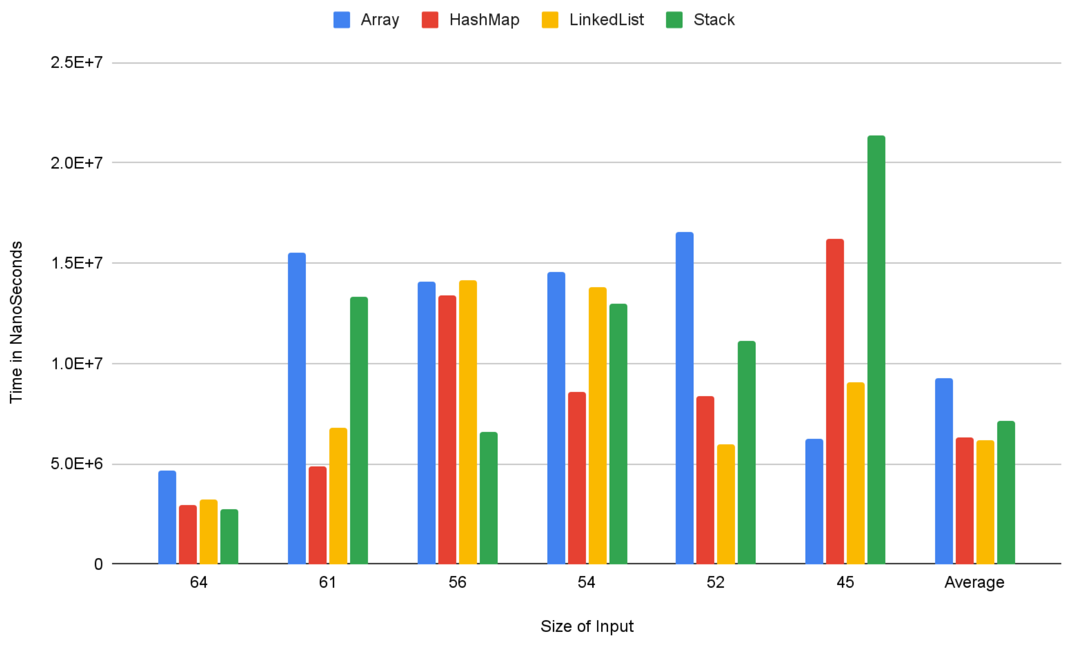


Figure 10

Comparison of Time Taken for Each Data Structure to Solve a 9x9 Sudoku



4.1.1. Time Complexity Analysis

Two types of analyses were made, the first counting the number of operations through the number of loops done by the *solve()* and *legal()* function, and the second counting through nanoseconds. Nanoseconds were used in this observation to find the most accurate differences between each data structure. With empirical data, it can be seen that the time complexity linear graphs show the same results, this is due to the structure of the recursive loops being the same throughout all 4 programs, even the overall time complexities are the same. Though, with theoretical data, the input size taken must be the size of the board, also known as the variable SIZE, this is due to how to program processes the loops, the analysis occurs through consideration of how each line of code within the recursive functions work. This yields the final equation:

$$T(n) = O(n^2) \times O(n) \times O(n^2) = O(n^5)$$

The final theoretical time complexity is polynomial, where the number of operations increases polynomially according to the size of the Sudoku board.

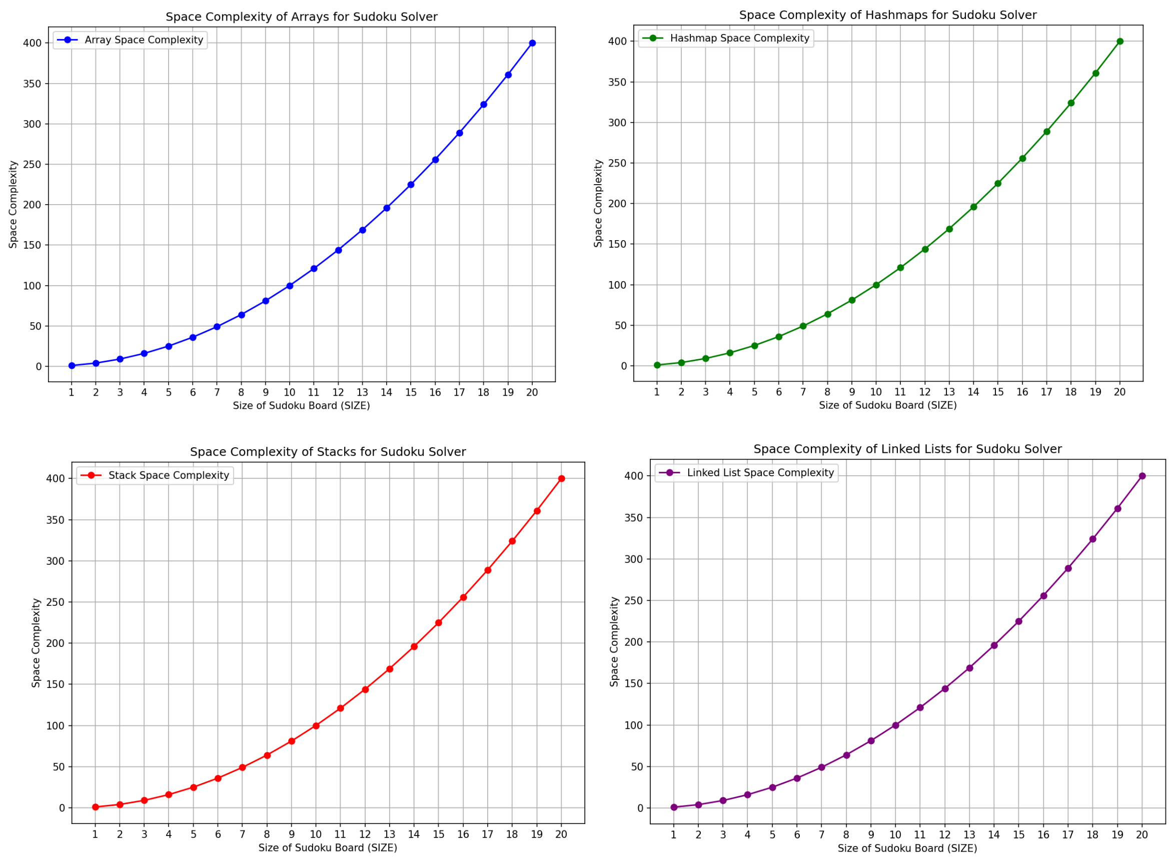
4.2. Space Complexity

Space complexity is the amount of memory that is required by the program to run and execute as a function of the size of the program's input. It measures how much additional memory is needed parallel to the input size to perform the task. In this report, we will use the Big-O notation to analyze the worst-case scenario of each program and analyze the overall complexity of each program (Thakrani, 2023).

This program's space complexity refers to the amount of memory needed by the program to run the functions of the size of the sudoku board (3x3 or 9x9). The space complexity relies on the size of the sudoku board instead of the user input in each cell.

Figure 11

Comparison of Space Taken for Each Data Structure to Solve a 9x9 and 3x3 Sudoku



Our group discovered that the space complexity remained constant regardless of what data structure we chose. For 'LinkedList<JTextField> cells', 'Stack<JTextField> cells', 'HashMap<Integer, JTextField> cells', as well as 'JTextField[][] cells', the space complexity for all of them is O(SIZE^2). Therefore, no matter the data structure used, all of the programs have the same space complexity, even for larger Sudoku boards such as the 9x9.

Linked list: The linked list program has a "LinkedList<Integer> board:" in the solve method which is used to store the input of the Sudoku board. It holds "SIZE * SIZE" INteger objects and each Integer value requires memory for the integer value and the object overhead which makes the space complexity O(SIZE^2).

Stack: The program with the Stack data structure has a "Stack<Integer, Integer>" in the solve method too which makes the space complexity O(SIZE^2).

HashMap: The Space complexity in the program with HashMap data structure associated with the ‘cells’ depends on the size of the grid, in this case(‘SIZE*SIZE’). Each grid is represented by an index which is ‘Integer’ making the space complexity O(SIZE^2)

2D Array: The array has a dimension which is Size by Size to represent the row and column of the sudoku grid therefore the space complexity is O(SIZE^2).

4.2.1. Space Complexity Analysis

LinkedList

1. ‘LinkedList<JTextField> cells’
   a. Size: ‘Size x Size’
   b. Each element referenced to a JtextField object
   c. Space Complexity: O(SIZE^2)
2. ‘LinkedList<Integer> board’:
   a. Size: ‘SIZE x SIZE’
   b. Each element stores an Integer value
   c. Space Complexity O(SIZE^2)

Primitive Variables

a. ‘int SIZE’ : 4 bytes
b. ‘Int operationCount’: 4 bytes
c. ‘int inputSize’: 4 bytes
d. ‘int filledCellsCount’: 4 bytes
e. ‘int min_clues’: 4 bytes
f. ‘Timer timer’: 4 bytes
g. ‘long startTime’: 8 bytes
h. ‘JLabel timerLabel’: 4 bytes

Total Space Complexity:

- LinkedList: O(SIZE^2)
- Primitive Variables: O(1)

Since the LinkedList space complexity is O(SIZE^2) and it is above the space complexity of the Primitive Variable O(1), the overall space complexity is determined by the Linked List, which is O(SIZE^2).

Stack

1. ‘Stack<JTextField> cells’
    a. Size: ‘SIZE x SIZE’
    b. Space Complexity: O(SIZE^2)
2. ‘Stack<Integer> board’
    a. Size: ‘SIZE x SIZE’
    b. Space Complexity: O(SIZE^2)

Primitive Variables

a. ‘int SIZE’ : 4 bytes
b. ‘Int operationCount’: 4 bytes
c. ‘int inputSize’: 4 bytes
d. ‘int filledCellsCount’: 4 bytes
e. ‘int min_clues’: 4 bytes
f. ‘Timer timer’: 4 bytes
g. ‘long startTime’: 8 bytes
h. ‘JLabel timerLabel’: 4 bytes

Total Space Complexity

- Stack: O(SIZE^2)
- Primitive Variables: O(1)

Since the Stack space complexity is O(SIZE^2) and it is above the space complexity of the Primitive Variable O(1), the overall space complexity is determined by the Stack, which is O(SIZE^2).

HashMaps

1. ‘HashMap<Integer, JTextField> cells’
    a. Size : ‘SIZE x SIZE’

    b. Space Complexity: O(SIZE^2)
2. ‘HashMap<String, Integer> board’
    a. Size: ‘SIZE x SIZE’
    b. Space Complexity: O(SIZE^2)

Primitive Variables

i. ‘int SIZE’ : 4 bytes
j. ‘Int operationCount’: 4 bytes
k. ‘int inputSize’: 4 bytes
l. ‘int filledCellsCount’: 4 bytes
m. ‘int min_clues’: 4 bytes
n. ‘Timer timer’: 4 bytes
o. ‘long startTime’: 8 bytes
p. ‘JLabel timerLabel’: 4 bytes

Total Space Complexity

- HashMap: O(SIZE^2)
- Primitive Variables: O(1)

Since the HashMap space complexity is O(SIZE^2) and it is above the space complexity of the Primitive Variable O(1), the overall space complexity is determined by the HashMap, which is O(SIZE^2).

2D Array

1. ‘JTextField[][] cells’
    a. Size : ‘SIZE x SIZE’
    b. Space Complexity: O(SIZE^2)
2. ‘Int[][] board’
    a. Size: ‘SIZE x SIZE’
    b. Each int occupies 4 bytes
    c. Space Complexity: O(SIZE^2)

Primitive Variable

a. ‘int SIZE’ : 4 bytes
b. ‘int operation Count’ : 4 bytes
c. ‘int inputSize’: 4 bytes
d. ‘int filledCellsCount’: 4 bytes
e. ‘int min_clues’: 4 bytes
f. ‘Timer timer’: 4 bytes
g. ‘long startTime’: 8 bytes
h. ‘JLabel timerLabel’: 4 bytes

Total Space Complexity

- Arrays: O(SIZE^2)
- Primitive Variable: O(1)

Since the 2D Array space complexity is O(SIZE^2) and it is above the space complexity of the Primitive Variable O(1), the overall space complexity is determined by the 2D Array, which is O(SIZE^2) (*Java Data Types*, n.d.).

5. Results and Implementation

The research for time complexity has revealed several key findings. The initial hypothesis has been proven partially incorrect. The program takes into account two different board sizes, 3x3 and 9x9. With a 3x3 board, there are 9 cells to be filled, and a minimum of 4 hints for the board to be solvable.

The data structure that could solve a 3x3 board with the least amount of operations, is 2D Array, this is possible because accessing elements in an array is faster as it uses direct indexing, especially for small to medium-sized arrays (Nikita et al., 2023). Continuing, the least efficient data structure for a 3x3 board is hashmap. Hashmaps have more memory cost than arrays because they hold key-value pairs and employ a hash table structure for quick search (Ilhamsyah, 2022). In a tiny data set, such as a 3x3 Sudoku board, this overhead might be rather large in comparison to the actual data size. While hashmaps give O(1) average-case lookup time, the benefit is more obvious in bigger datasets. In a small 3x3 board, direct access via array indices can be just as efficient, if not faster, than hash lookups.

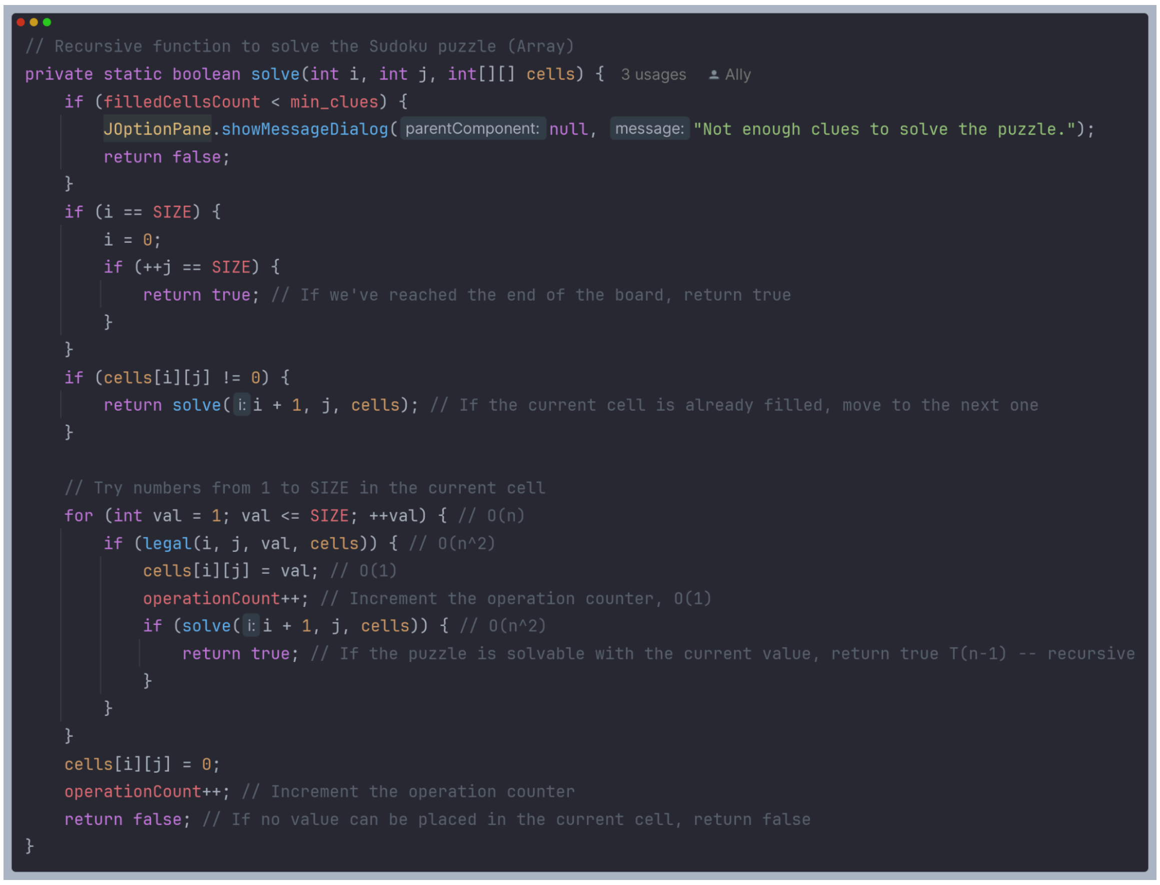
For the 9x9 board, the most efficient data structure was found to be LinkedList, although the initial hypothesis stated otherwise, this occurred due to the dynamic nature of a LinkedList. A linked list is particularly efficient for inserting and deleting elements at arbitrary positions because it does not require shifting elements, unlike arrays. This is crucial in applications like Sudoku, where cells can be filled or cleared dynamically. These findings are agreed upon by Lokeshwar et al. (2022), as it is stated that linked lists are more efficient than arrays, especially with large amounts of data (Naidu & Prasad, 2014). The least efficient data structure for this board size was HashMap, this is possibly due to the fact that hashing collisions may occur, if there are many hash collisions. it can degrade performance as these must be resolved (Ilhamsyah, 2022). In general, hash maps are the most inefficient data structure to solve a sudoku puzzle using a backtracking algorithm due to the possibilities of hashing collisions and the time complexities of the collision resolutions, whilst 2D arrays and linked lists are the most efficient.

The research on space complexity gave us some major findings. Our hypothesis is proven incorrect. Our hypothesis believed that the most and least space-efficient would be the same as the result in time complexity (2D array to be the most efficient and HashMap to be the least). Based on the research, we found that despite the different implementations of data structure, there is barely any difference in space efficiency and that it is largely comparable (Brodal et al., 2010). There are minor differences that do not affect the space complexity majorly such as the key-value pair of HashMap or the direct indexing of the array. Even as the size of the Sudoku board and the amount of space needed increase, the difference in space complexity between the four data structures will be relatively small.

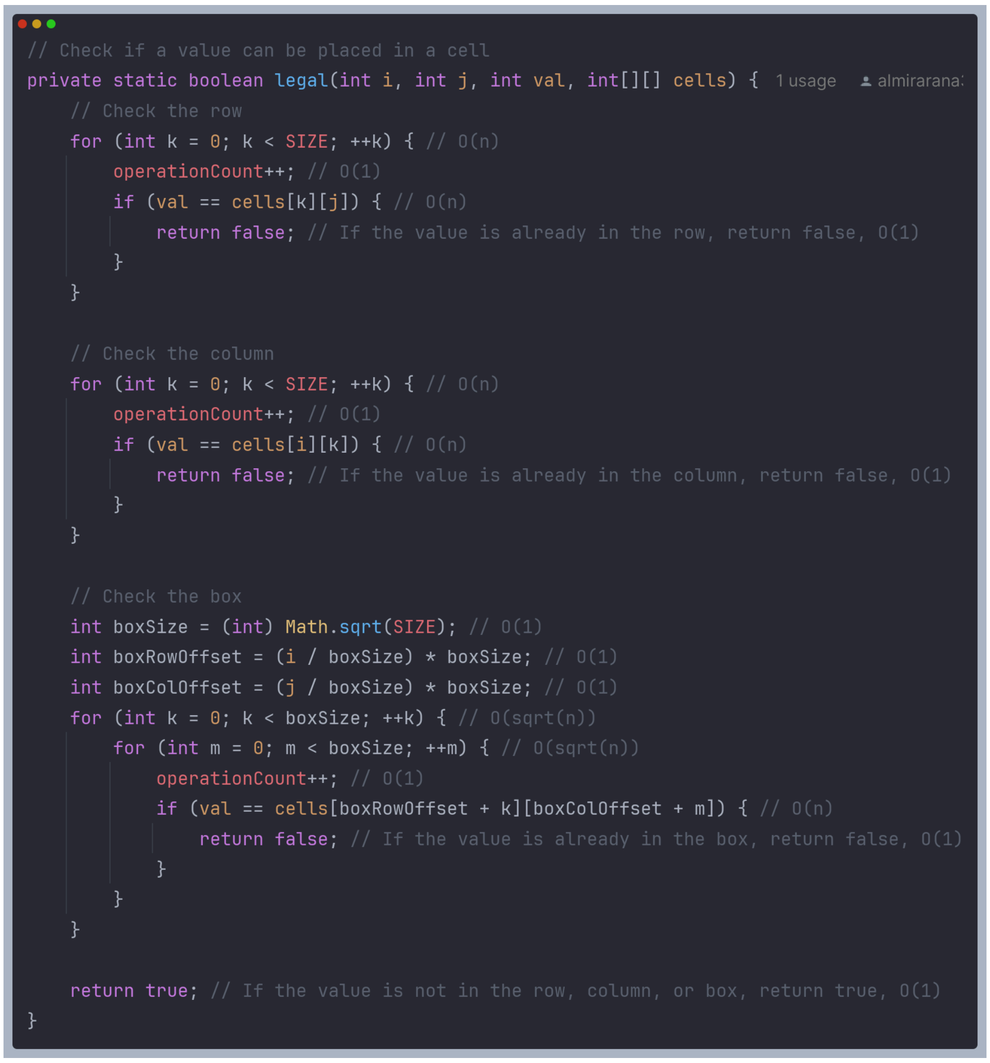
5.1. Screenshots of Solution

The provided screenshots highlight the *solve()* and *legal()* functions as those are where the solving algorithm occurs.

2D Array

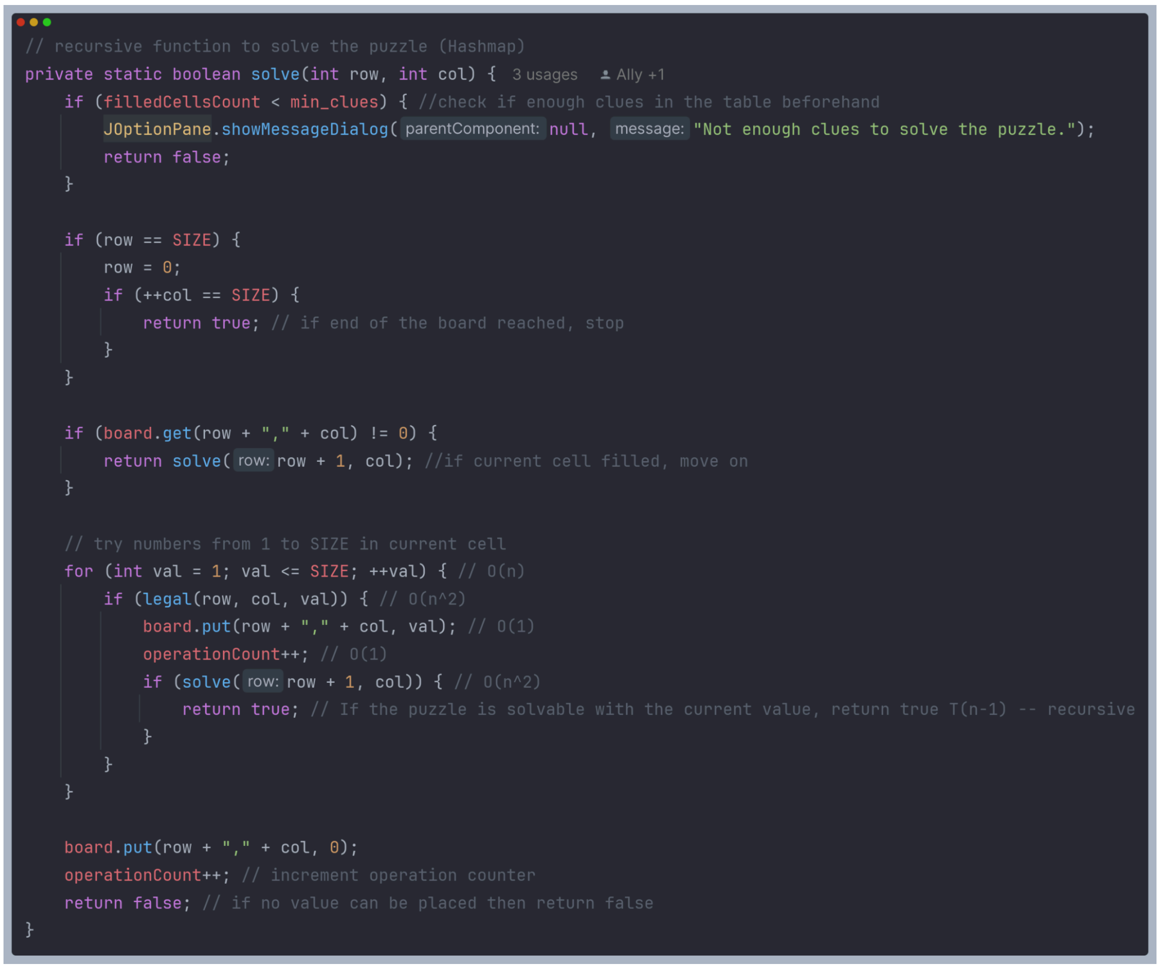


```
// Recursive function to solve the Sudoku puzzle (Array)
private static boolean solve(int i, int j, int[][] cells) {  3 usages  Ally
    if (filledCellsCount < min_clues) {
        JOptionPane.showMessageDialog( parentComponent: null,  message: "Not enough clues to solve the puzzle.");
        return false;
    }
    if (i == SIZE) {
        i = 0;
        if (++j == SIZE) {
            return true; // If we've reached the end of the board, return true
        }
    }
    if (cells[i][j] != 0) {
        return solve( i: i + 1, j, cells); // If the current cell is already filled, move to the next one
    }

    // Try numbers from 1 to SIZE in the current cell
    for (int val = 1; val <= SIZE; ++val) { // O(n)
        if (legal(i, j, val, cells)) { // O(n^2)
            cells[i][j] = val; // O(1)
            operationCount++; // Increment the operation counter, O(1)
            if (solve( i: i + 1, j, cells)) { // O(n^2)
                return true; // If the puzzle is solvable with the current value, return true T(n-1) -- recursive
            }
        }
    }
    cells[i][j] = 0;
    operationCount++; // Increment the operation counter
    return false; // If no value can be placed in the current cell, return false
}
```

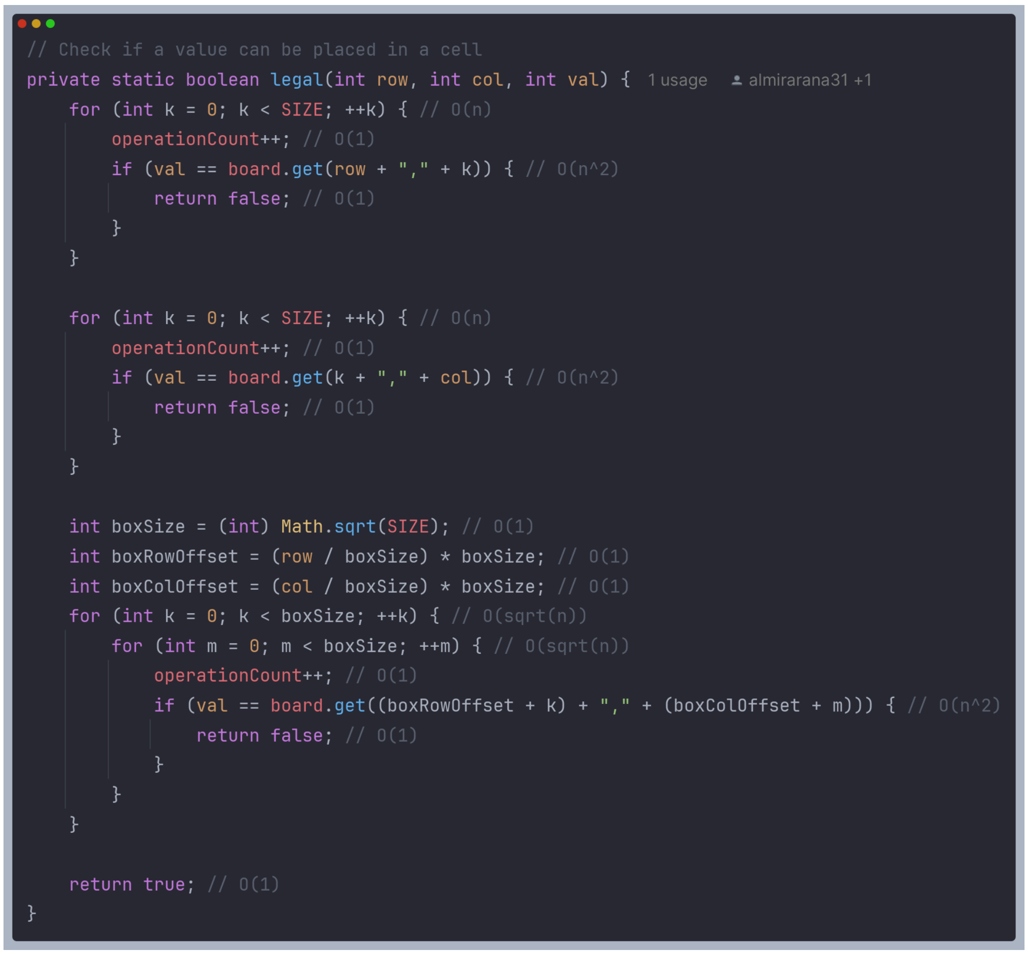


```
// Check if a value can be placed in a cell
private static boolean legal(int i, int j, int val, int[][] cells) {  1 usage  almirarana
    // Check the row
    for (int k = 0; k < SIZE; ++k) { // O(n)
        operationCount++; // O(1)
        if (val == cells[k][j]) { // O(n)
            return false; // If the value is already in the row, return false, O(1)
        }
    }

    // Check the column
    for (int k = 0; k < SIZE; ++k) { // O(n)
        operationCount++; // O(1)
        if (val == cells[i][k]) { // O(n)
            return false; // If the value is already in the column, return false, O(1)
        }
    }

    // Check the box
    int boxSize = (int) Math.sqrt(SIZE); // O(1)
    int boxRowOffset = (i / boxSize) * boxSize; // O(1)
    int boxColOffset = (j / boxSize) * boxSize; // O(1)
    for (int k = 0; k < boxSize; ++k) { // O(sqrt(n))
        for (int m = 0; m < boxSize; ++m) { // O(sqrt(n))
            operationCount++; // O(1)
            if (val == cells[boxRowOffset + k][boxColOffset + m]) { // O(n)
                return false; // If the value is already in the box, return false, O(1)
            }
        }
    }

    return true; // If the value is not in the row, column, or box, return true, O(1)
}
```

HashMap

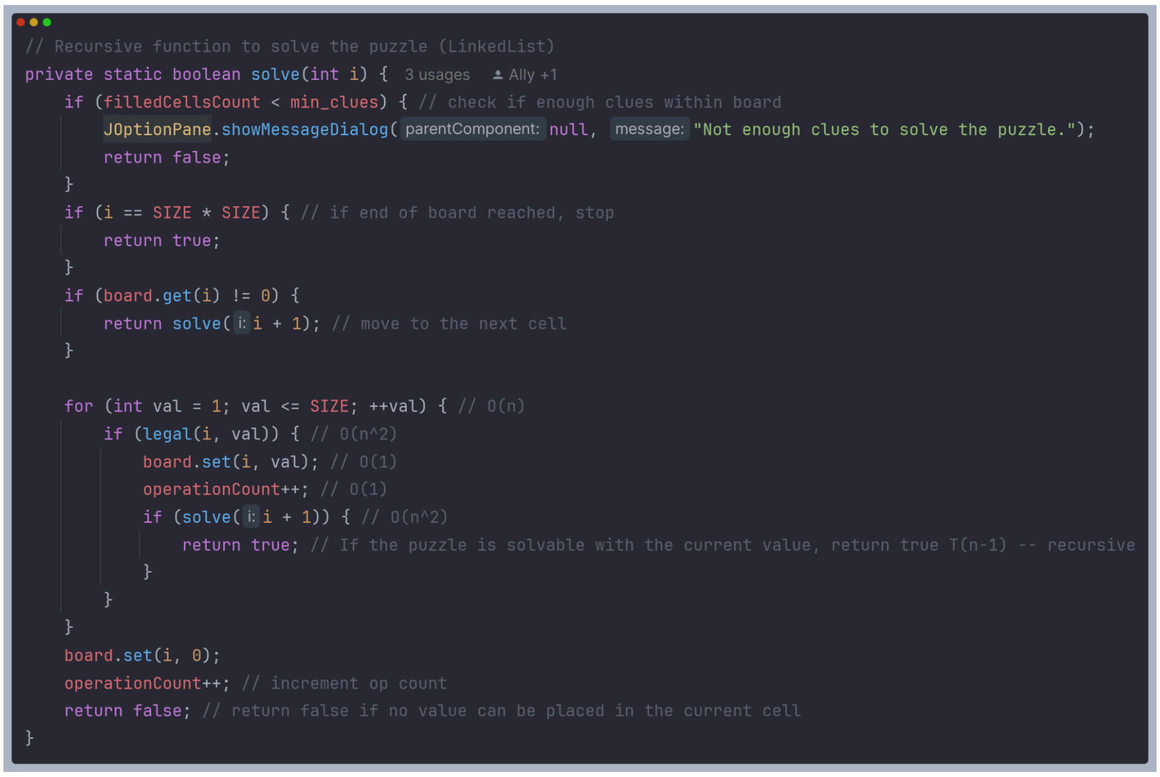


```
// recursive function to solve the puzzle (Hashmap)
private static boolean solve(int row, int col) {  3 usages   Ally +1
    if (filledCellsCount < min_clues) { //check if enough clues in the table beforehand
        JOptionPane.showMessageDialog( parentComponent: null,  message: "Not enough clues to solve the puzzle.");
        return false;
    }

    if (row == SIZE) {
        row = 0;
        if (++col == SIZE) {
            return true; // if end of the board reached, stop
        }
    }

    if (board.get(row + "," + col) != 0) {
        return solve( row: row + 1, col); //if current cell filled, move on
    }

    // try numbers from 1 to SIZE in current cell
    for (int val = 1; val <= SIZE; ++val) { // O(n)
        if (legal(row, col, val)) { // O(n^2)
            board.put(row + "," + col, val); // O(1)
            operationCount++; // O(1)
            if (solve( row: row + 1, col)) { // O(n^2)
                return true; // If the puzzle is solvable with the current value, return true T(n-1) -- recursive
            }
        }
    }

    board.put(row + "," + col, 0);
    operationCount++; // increment operation counter
    return false; // if no value can be placed then return false
}
```

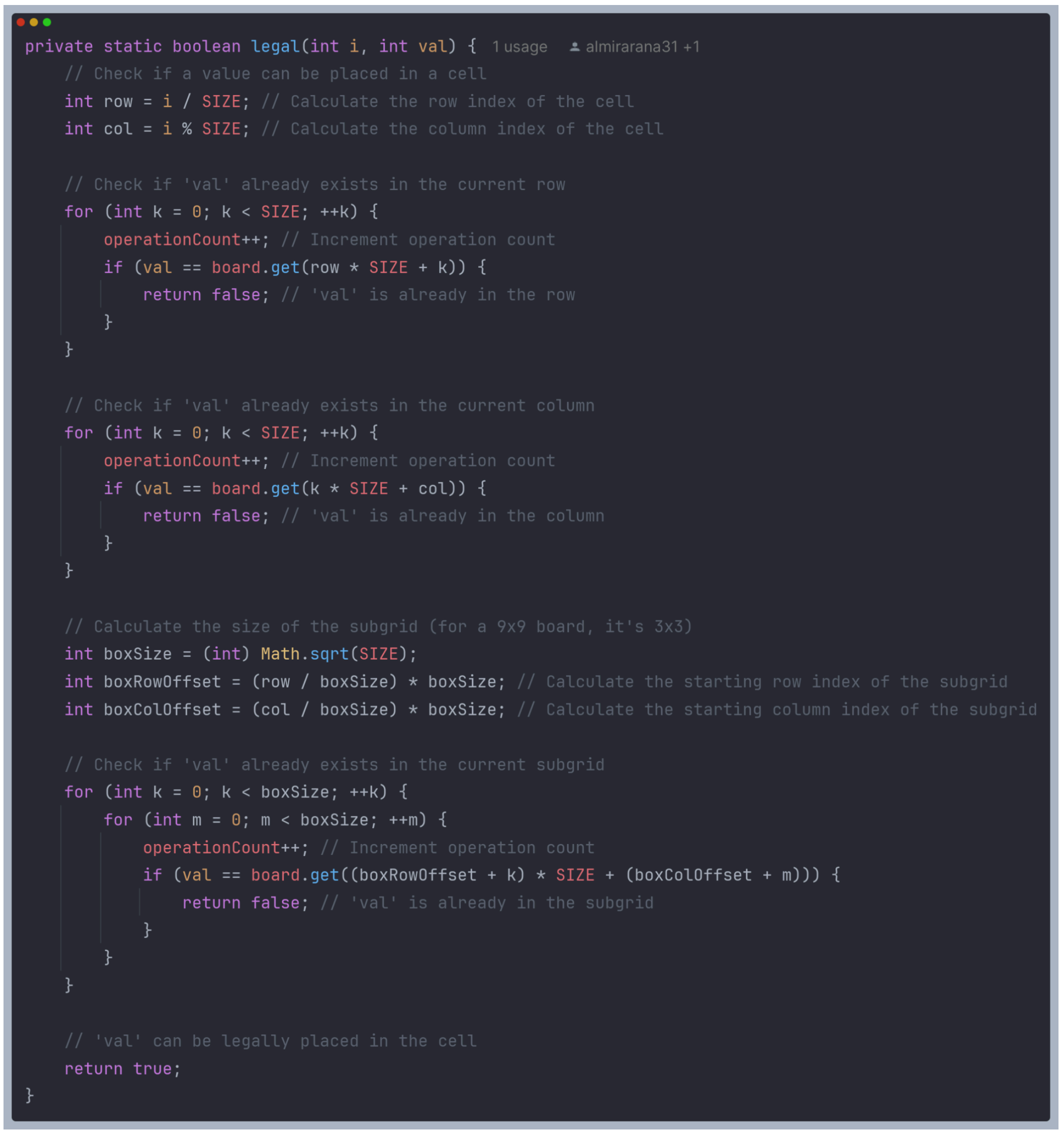


```
// Check if a value can be placed in a cell
private static boolean legal(int row, int col, int val) {  1 usage   almirarana31 +1
    for (int k = 0; k < SIZE; ++k) { // O(n)
        operationCount++; // O(1)
        if (val == board.get(row + "," + k)) { // O(n^2)
            return false; // O(1)
        }
    }

    for (int k = 0; k < SIZE; ++k) { // O(n)
        operationCount++; // O(1)
        if (val == board.get(k + "," + col)) { // O(n^2)
            return false; // O(1)
        }
    }

    int boxSize = (int) Math.sqrt(SIZE); // O(1)
    int boxRowOffset = (row / boxSize) * boxSize; // O(1)
    int boxColOffset = (col / boxSize) * boxSize; // O(1)
    for (int k = 0; k < boxSize; ++k) { // O(sqrt(n))
        for (int m = 0; m < boxSize; ++m) { // O(sqrt(n))
            operationCount++; // O(1)
            if (val == board.get((boxRowOffset + k) + "," + (boxColOffset + m))) { // O(n^2)
                return false; // O(1)
            }
        }
    }

    return true; // O(1)
}
```

LinkedList

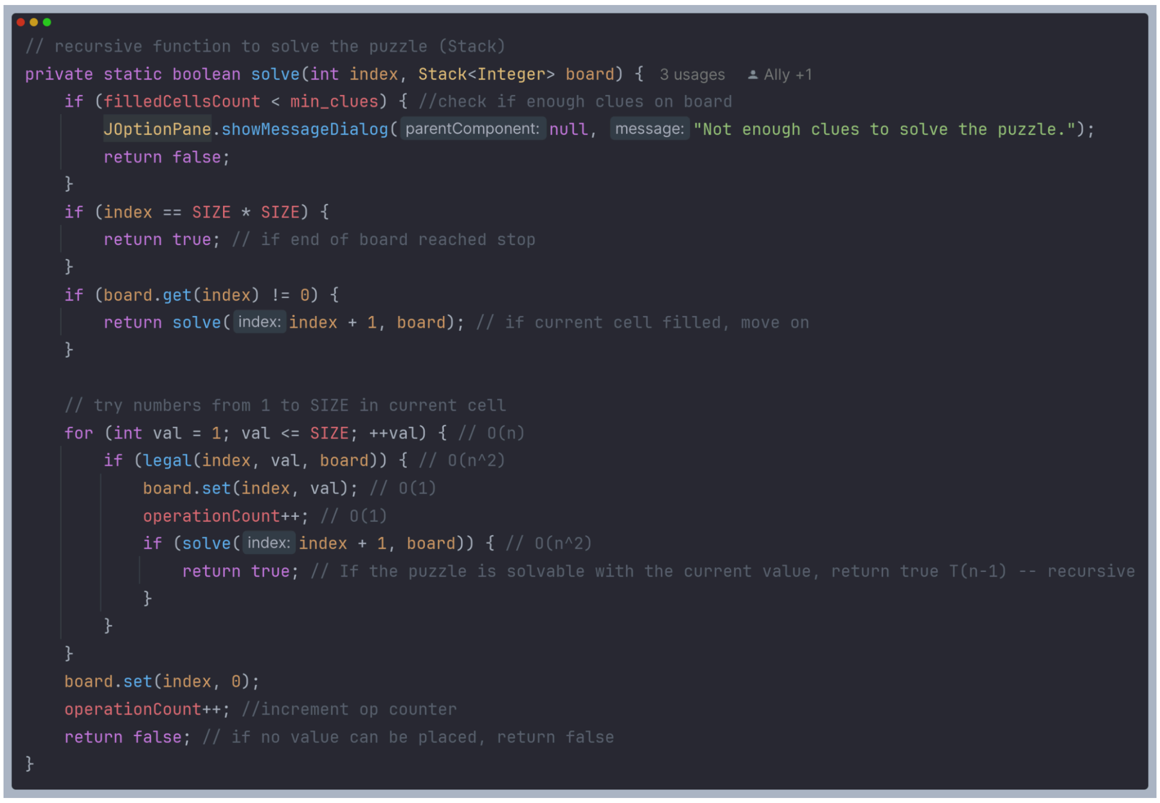


```java
// Recursive function to solve the puzzle (LinkedList)
private static boolean solve(int i) {  3 usages  Ally +1
    if (filledCellsCount < min_clues) { // check if enough clues within board
        JOptionPane.showMessageDialog( parentComponent: null,  message: "Not enough clues to solve the puzzle.");
        return false;
    }
    if (i == SIZE * SIZE) { // if end of board reached, stop
        return true;
    }
    if (board.get(i) != 0) {
        return solve( i: i + 1); // move to the next cell
    }

    for (int val = 1; val <= SIZE; ++val) { // O(n)
        if (legal(i, val)) { // O(n^2)
            board.set(i, val); // O(1)
            operationCount++; // O(1)
            if (solve( i: i + 1)) { // O(n^2)
                return true; // If the puzzle is solvable with the current value, return true T(n-1) -- recursive
            }
        }
    }
    board.set(i, 0);
    operationCount++; // increment op count
    return false; // return false if no value can be placed in the current cell
}
```

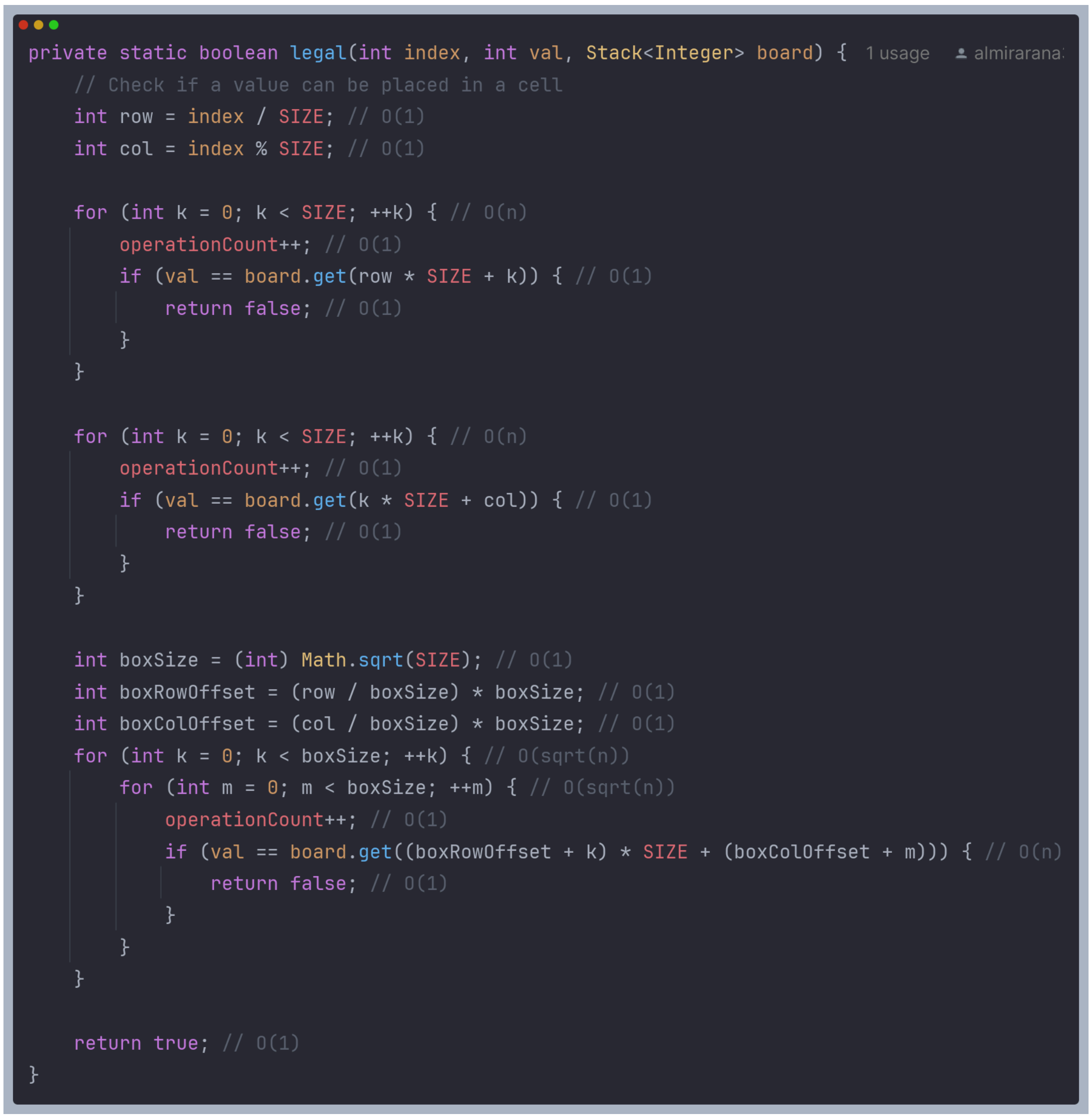


```java
private static boolean legal(int i, int val) {  1 usage  almirarana31 +1
    // Check if a value can be placed in a cell
    int row = i / SIZE; // Calculate the row index of the cell
    int col = i % SIZE; // Calculate the column index of the cell

    // Check if 'val' already exists in the current row
    for (int k = 0; k < SIZE; ++k) {
        operationCount++; // Increment operation count
        if (val == board.get(row * SIZE + k)) {
            return false; // 'val' is already in the row
        }
    }

    // Check if 'val' already exists in the current column
    for (int k = 0; k < SIZE; ++k) {
        operationCount++; // Increment operation count
        if (val == board.get(k * SIZE + col)) {
            return false; // 'val' is already in the column
        }
    }

    // Calculate the size of the subgrid (for a 9x9 board, it's 3x3)
    int boxSize = (int) Math.sqrt(SIZE);
    int boxRowOffset = (row / boxSize) * boxSize; // Calculate the starting row index of the subgrid
    int boxColOffset = (col / boxSize) * boxSize; // Calculate the starting column index of the subgrid

    // Check if 'val' already exists in the current subgrid
    for (int k = 0; k < boxSize; ++k) {
        for (int m = 0; m < boxSize; ++m) {
            operationCount++; // Increment operation count
            if (val == board.get((boxRowOffset + k) * SIZE + (boxColOffset + m))) {
                return false; // 'val' is already in the subgrid
            }
        }
    }

    // 'val' can be legally placed in the cell
    return true;
}
```

Stack



```java
// recursive function to solve the puzzle (Stack)
private static boolean solve(int index, Stack<Integer> board) {  3 usages  Ally +1
    if (filledCellsCount < min_clues) { //check if enough clues on board
        JOptionPane.showMessageDialog( parentComponent: null,  message: "Not enough clues to solve the puzzle.");
        return false;
    }
    if (index == SIZE * SIZE) {
        return true; // if end of board reached stop
    }
    if (board.get(index) != 0) {
        return solve( index: index + 1, board); // if current cell filled, move on
    }

    // try numbers from 1 to SIZE in current cell
    for (int val = 1; val <= SIZE; ++val) { // O(n)
        if (legal(index, val, board)) { // O(n^2)
            board.set(index, val); // O(1)
            operationCount++; // O(1)
            if (solve( index: index + 1, board)) { // O(n^2)
                return true; // If the puzzle is solvable with the current value, return true T(n-1) -- recursive
            }
        }
    }
    board.set(index, 0);
    operationCount++; //increment op counter
    return false; // if no value can be placed, return false
}
```



```java
private static boolean legal(int index, int val, Stack<Integer> board) {  1 usage  almirarana
    // Check if a value can be placed in a cell
    int row = index / SIZE; // O(1)
    int col = index % SIZE; // O(1)

    for (int k = 0; k < SIZE; ++k) { // O(n)
        operationCount++; // O(1)
        if (val == board.get(row * SIZE + k)) { // O(1)
            return false; // O(1)
        }
    }

    for (int k = 0; k < SIZE; ++k) { // O(n)
        operationCount++; // O(1)
        if (val == board.get(k * SIZE + col)) { // O(1)
            return false; // O(1)
        }
    }

    int boxSize = (int) Math.sqrt(SIZE); // O(1)
    int boxRowOffset = (row / boxSize) * boxSize; // O(1)
    int boxColOffset = (col / boxSize) * boxSize; // O(1)
    for (int k = 0; k < boxSize; ++k) { // O(sqrt(n))
        for (int m = 0; m < boxSize; ++m) { // O(sqrt(n))
            operationCount++; // O(1)
            if (val == board.get((boxRowOffset + k) * SIZE + (boxColOffset + m))) { // O(n)
                return false; // O(1)
            }
        }
    }

    return true; // O(1)
}
```

References

Agile Education Research. (2017). *Hash Map*. Java Programming. https://java-programming.mooc.fi/part-8/2-hash-map

Berggren, P., & Nilsson, D. (2012). *A study of Sudoku Solving Algorithms* [Bachelor of Science Thesis, KTH Royal Institute of Technology]. https://www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2012/rapport/berggren_patrik_OCH_nilsson_david_K12011.pdf

Brodal, G. S., Davoodi, P., & Rao, S. S. (2011). On space efficient two dimensional range minimum data structures. *Algorithmica*, *63*(4), 815–830. https://doi.org/10.1007/s00453-011-9499-0

Brooks, R. (2023, August 15). *What are data structures? - North Wales Management School - Wrexham University*. North Wales Management School - Wrexham University. https://online.wrexham.ac.uk/what-are-data-structures/

Ekström, J., & Pitkäjärv, K. (2014). *The backtracking algorithm and different representations for solving Sudoku Puzzles* [Bachelor's Thesis, KTH Royal Institute of Technology]. https://www.diva-portal.org/smash/get/diva2:721641/FULLTEXT01.pdf

Ericson, B. (2015). Two-dimensional (2D) Arrays. In B. Hoffman, L. Seiter, & D. Palmer (Eds.), *Introduction to Programming with Java* (2nd ed.). Runestone. https://runestone.academy/ns/books/published/csjava/Unit9-2DArray/topic-9-1-2D-arrays.html

GeeksforGeeks. (2023, October 16). *Definition, types, complexity and examples of algorithm*. GeeksforGeeks. https://www.geeksforgeeks.org/what-is-an-algorithm-definition-types-complexity-examples/

GeeksforGeeks. (2024a, February 16). *Understanding the basics of Linked List*. GeeksforGeeks. https://www.geeksforgeeks.org/what-is-linked-list/

GeeksforGeeks. (2024b, June 6). *What is Stack Data Structure? A Complete Tutorial*. GeeksforGeeks. https://www.geeksforgeeks.org/introduction-to-stack-data-structure-and-algorithm-tutorials/

Ilhamsyah, F. N. R. (2022). *A Case Study in Optimization of Brute-Force Algorithm : Finding Longest Substring With Unique Characters* (No. 13521060). Bandung Institute of Technology. Retrieved June 18, 2024, from https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2022-2023/Makalah/Makalah-Stima-2023-%2816%29.pdf

Java Data Types. (n.d.). W3Schools. https://www.w3schools.com/java/java_data_types.asp

Lokeshwar, B., Zaid, M. M., Naveen, S., Venkatesh, J., & Sravya, L. (2022). Analysis of Time and Space Complexity of Array, Linked List and Linked Array(hybrid) in Linear Search Operation. In *International Conference on Data Science*. https://doi.org/10.1109/icdsaai55433.2022.10028872

Mehranfar, K. (2024, January 24). The History of Sudoku - 18th Century - Komeil Mehranfar - Medium. *Medium*. https://medium.com/@komeil.mehranfar/complete-history-of-sudoku-1ee6ab3aeafc#3c51

Naidu, D., & Prasad, A., Jr. (2014). Implementation of Enhanced Singly Linked List Equipped with DLL Operations: An Approach towards Enormous Memory Saving. *International Journal of Future Computer and Communication*, 98–101. https://doi.org/10.7763/ijfcc.2014.v3.276

Nikita, Singh, A., & Tiwari, S. (2023). Sudoku solving algorithm and grid based models for digit recognition. *International Journal for Research in Applied Science and Engineering Technology*, *11*(5), 6251–6258. https://doi.org/10.22214/ijraset.2023.52828

Pauli, S. (n.d.). Definition of an Algorithm. In *MAT 112 Integers and Modern Applications for the Uninitiated*. Runestone Academy. https://mathstats.uncg.edu/sites/pauli/112/HTML/secalgdef.html

Pelánek, R. (2014). Difficulty rating of Sudoku Puzzles: An Overview and Evaluation. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.1403.7373

Thakrani, S. (2023, March 10). *Space complexity and Time Complexity*. Medium. https://medium.com/@suhailthakrani12/space-complexity-and-time-complexity-53c505e1dfb8.

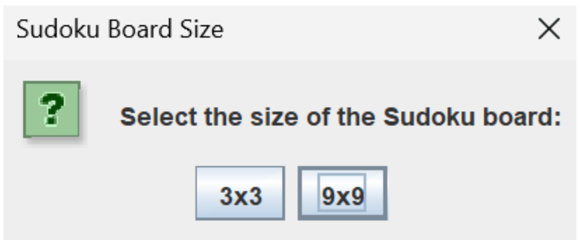
Appendix

Introduction

Sudoku Solver is a software that is made with Java language that is designed to help users solve the 3 x 3 and 9 x 9 sudoku. Users can input numbers to the sudoku puzzle, clear the board, and find the solution from what they have input using the built-in solver algorithm.

Getting Started

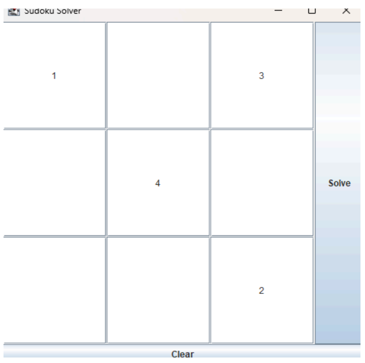
1. To run the program, install the Java file and run it on an IDE that supports the Java program. Then, run the program.
2. Once the program starts, Choose either a 3x3 or 9x9 sudoku box from the GUI button.


Sudoku Board Size

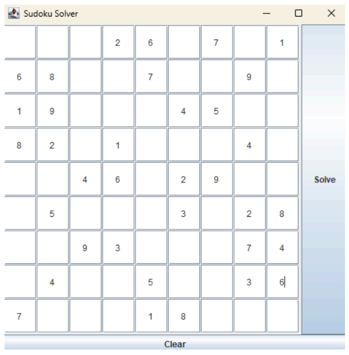
Select the size of the Sudoku board:

The Solving Features

3. Input Sudoku Puzzle: choose any cells and enter the number between 1 and the number of the board size(3 or 9). Make sure the puzzle has the minimum number of clues (4 numbers of input for 3x3 and 17 for 9x9) required for a valid solution.



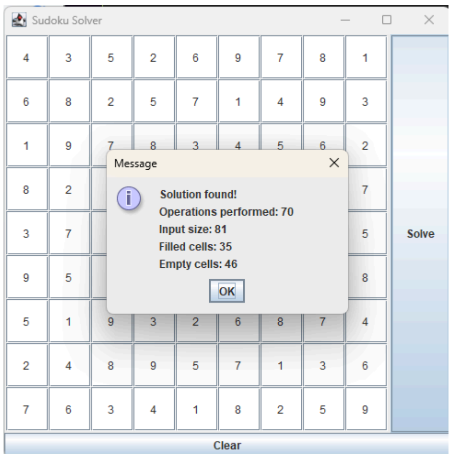
1		3
	4	
		2



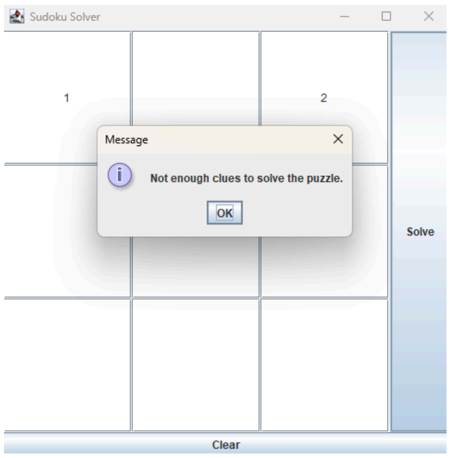
			2	6		7		1
6	8			7			9	
1	9				4	5		
8	2		1				4	
		4	6		2	9		
	5				3		2	8
		9	3				7	4
	4			5			3	6
7				1	8			

4. Solving the puzzle: Click the" solve" button to find the solution. Then the program will check to see if the input is valid or not, it will notify the users if

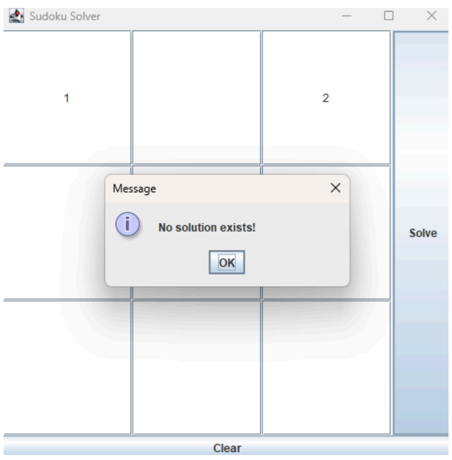
no solution exists because of the lack of clues or if there is a duplicate in the row, column, or subgrids.



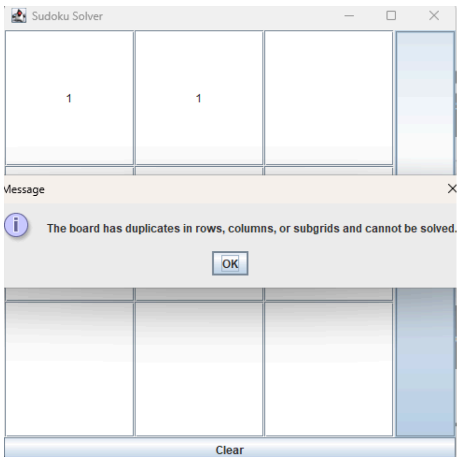
Solution found!
Operations performed: 70
Input size: 81
Filled cells: 35
Empty cells: 46



Not enough clues to solve the puzzle.



No solution exists!



The board has duplicates in rows, columns, or subgrids and cannot be solved.

5. Clear Sudoku Board: Click the “Clear” button to reset the board to make it empty.

Common Problem

- Problem: Puzzle is not solved
    - Solution: Make sure the puzzle has at least the number of clues for the program to solve it (4 for 3x3 and 17 for 9x9) and no duplicate numbers in the rows, columns, or subgrids.

Link to GitHub

https://github.com/almirarana31/DS_FinalProject.git

Link to Presentation

https://www.canva.com/design/DAGHwdvQ9zw/FVe5vibEKjvpGXTTKxcr8g/edit?utm_content=DAGHwdvQ9zw&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton