

## Step 1) Selection

I chose Puzzle 23 from the Survey of NP-Complete puzzles. This puzzle is Sudoku.

(23) **Sudoku** (also known as Number Place) was first published in Dell Magazines in 1979<sup>10</sup>. It is currently attracting more attention than other puzzles, perhaps because it is featured regularly in daily news articles. It is

## Step 2) Rules of Sudoku

The goal of Sudoku is to completely fill a 9×9 grid with digits. When filled with valid digits, every column, every row, and every 3×3 subgrid that compose the game board contain each digit from 1 to 9. The Sudoku player is provided a grid which is partially complete (some cells are left blank), and the Sudoku player aims to fill in the grid with valid digits so that the puzzle can be completed. If enough blanks are present when the board is first presented, it is possible for multiple valid solutions to exist for a single board, but a good instance of a Sudoku puzzle has only one single valid solution.

## Step 3) Implementation Allowing User to Solve the Puzzle

I implemented Sudoku using Python on the command line. When the 'consoleSudoku.py' file is run using Python, the following 'Console Sudoku Main Menu' is displayed in the console.

```
Welcome to Console Sudoku!

This is the Console Sudoku Main Menu.
Please enter one of the following options:

1. Read a short description about Sudoku.
2. Load a random Sudoku puzzle from the Puzzle Library.

Or enter 'Q' to quit Console Sudoku.
```

Option 1 displays the same paragraph as rules in Step 2 (above).

Option 2 randomly loads one of three puzzles from a Puzzle library json file and opens the 'Puzzle Selection Menu'.

When a randomly selected puzzle is displayed, the puzzle name & source are displayed as well:

```
Enter your selection: 2
You have selected a Sudoku puzzle called 'Easy Sudoku'.
This source of this puzzle is https://dingo.sbs.arizona.edu/~sandiway/sudoku/examples.html.
```

```
Columns
  0  1  2  3  4  5  6  7  8
0  -  -  -  -  2  -  -  -  -
1  6  8  -  -  7  -  -  9
2  1  9  -  -  4  5  -  -
3  8  2  -  1  -  -  4  -
4  -  -  4  6  -  2  9  -
5  -  5  -  -  -  3  -  2
6  -  -  9  3  -  -  -  7
7  -  4  -  -  5  -  -  3
8  7  -  3  -  1  8  -  -

Rows
  0  1  2  3  4  5  6  7  8
0  -  -  -  -  2  -  -  -  -
1  6  8  -  -  7  -  -  9
2  1  9  -  -  4  5  -  -
3  8  2  -  1  -  -  4  -
4  -  -  4  6  -  2  9  -
5  -  5  -  -  -  3  -  2
6  -  -  9  3  -  -  -  7
7  -  4  -  -  5  -  -  3
8  7  -  3  -  1  8  -  -

This is the Puzzle Selection Menu.
Please enter one of the following options:

1. Attempt to solve this puzzle.
2. Generate and view a valid solution to this puzzle.
3. Return to the Main Menu.

Or enter 'Q' to quit Console Sudoku.
```

Within the 'Puzzle Selection Menu', a user can attempt to solve the randomly selected puzzle by selecting option 1.

```

Columns
  0  1  2  3  4  5  6  7  8
0  ---|---|---|---|---|---|---|---|
1  6  8  ---|---|7  ---|9  ---|
2  1  9  ---|---|4  5  ---|
3  8  2  ---|1  ---|---|4  ---|
4  ---|---|4  6  ---|2  9  ---|
5  ---|5  ---|---|3  ---|2  8  |
6  ---|---|9  3  ---|---|7  4  |
7  ---|4  ---|---|5  ---|---|3  6  |
8  7  ---|3  ---|1  8  ---|---|
Rows

You are currently solving this puzzle.
Please enter one of the following options:

1. Place a number in an empty cell of this Sudoku board.
2. Submit this Sudoku board and verify if this puzzle has been solved.
3. Reset this Sudoku board. Warning: All progress will be lost!
4. Close this Sudoku board, and return to the Puzzle Selection Menu for this puzzle.

Or enter 'Q' to quit Console Sudoku.
Enter your selection:

```

Once option 1 is selected, the console UI will state "You are currently solving this puzzle" and present the user with additional options to enter values into the cells, submit the puzzle, reset the puzzle, or close the puzzle and return to the Puzzle Selection Menu.

Submitting a board will check if the proposed solution is valid. If it is not valid, it will display the first error found during the validation algorithm.

```

Enter your selection: 2
You have submitted the current board. Checking if the current board is a valid solution...
The provided submission is not valid.
The digit in Row 0, Column 0 cannot be 0.

Columns
  0  1  2  3  4  5  6  7  8
0  ---|---|---|---|---|---|---|---|
1  6  8  ---|---|7  ---|9  ---|
2  1  9  ---|---|4  5  ---|
3  8  2  ---|1  ---|---|4  ---|
4  ---|---|4  6  ---|2  9  ---|
5  ---|5  ---|---|3  ---|2  8  |
6  ---|---|9  3  ---|---|7  4  |
7  ---|4  ---|---|5  ---|---|3  6  |
8  7  ---|3  ---|1  8  ---|---|
Rows

You are currently solving this puzzle.
Please enter one of the following options:

1. Place a number in an empty cell of this Sudoku board.
2. Submit this Sudoku board and verify if this puzzle has been solved.
3. Reset this Sudoku board. Warning: All progress will be lost!
4. Close this Sudoku board, and return to the Puzzle Selection Menu for this puzzle.

Or enter 'Q' to quit Console Sudoku.

```

Submitting a valid solution will verify the user's solution and confirm whether it is actually a correct solution. If it is correct, Console Sudoku will display "Solved! Nice work." to the user.

```

Enter your selection: 2
You have submitted the current board. Checking if the current board is a valid solution...
Solved! Nice work.

      Columns
      0  1  2  3  4  5  6  7  8
-----
0  | 4  3  5  2  6  9  7  8  1 |
1  | 6  8  2  5  7  1  4  9  3 |
2  | 1  9  7  8  3  4  5  6  2 |
3  | 8  2  6  1  9  5  3  4  7 |
ROWS 4  | 3  7  4  6  8  2  9  1  5 |
5  | 9  5  1  7  4  3  6  2  8 |
6  | 5  1  9  3  2  6  8  7  4 |
7  | 2  4  8  9  5  7  1  3  6 |
8  | 7  6  3  4  1  8  2  5  9 |
-----

You are currently solving this puzzle.
Please enter one of the following options:

1. Place a number in an empty cell of this Sudoku board.
2. Submit this Sudoku board and verify if this puzzle has been solved.
3. Reset this Sudoku board. Warning: All progress will be lost!
4. Close this Sudoku board, and return to the Puzzle Selection Menu for this puzzle.

Or enter 'Q' to quit Console Sudoku.

```

**Bonus A: Finding A Solution to the Puzzle**

If the user returns to the 'Puzzle Selection Menu', the user can specify that they would like to see a solved version of the selected puzzle.

```
Enter your selection: 2
You have selected a Sudoku puzzle called 'Very Hard Sudoku'.
This source of this puzzle is https://www.sudokudragon.com/unsolvable.htm.
```

		Columns									
		0	1	2	3	4	5	6	7	8	
Rows	0	---	---	---	2	6	---	7	---	1	
	1	6	8	---	---	7	---	---	9	---	
	2	1	9	---	---	---	4	5	---	---	
	3	8	2	---	1	---	---	---	4	---	
	4	---	---	4	6	---	2	9	---	---	
	5	---	5	---	---	---	3	---	2	8	
	6	---	---	9	3	---	---	---	7	4	
	7	---	4	---	---	5	---	---	3	6	
	8	7	---	3	---	1	8	---	---	---	

This is the Puzzle Selection Menu.  
Please enter one of the following options:

1. Attempt to solve this puzzle.
2. Generate and view a valid solution to this puzzle.
3. Return to the Main Menu.

Or enter 'Q' to quit Console Sudoku.

```
Enter your selection: 2
```

```
Checking for a solution...
```

```
One valid solution to this Sudoku puzzle is as follows:
```

		Columns									
		0	1	2	3	4	5	6	7	8	
Rows	0	4	3	5	2	6	9	7	8	1	
	1	6	8	2	5	7	1	4	9	3	
	2	1	9	7	8	3	4	5	6	2	
	3	8	2	6	1	9	5	3	4	7	
	4	3	7	4	6	8	2	9	1	5	
	5	9	5	1	7	4	3	6	2	8	
	6	5	1	9	3	2	6	8	7	4	
	7	2	4	8	9	5	7	1	3	6	
	8	7	6	3	4	1	8	2	5	9	

This is the Puzzle Selection Menu.  
Please enter one of the following options:

1. Attempt to solve this puzzle.
2. Generate and view a valid solution to this puzzle.
3. Return to the Main Menu.

Or enter 'Q' to quit Console Sudoku.

```
Enter your selection:
```

Note: The process of generating a solution does not compare the provided puzzle with any sort of 'expected' solution, and it only generates one possible solution using the solving algorithm. Other solutions may be possible depending on the puzzle selected.





## Step 5: Prove the Correctness of the Verification Algorithm

```
def is_valid_sudoku(board_to_test):
    for row in range(9):
        for col in range(9):
            if temp_board[row][col] == 0:
                return {'is_valid': False, 'invalid_row': row, 'invalid_column': col, 'invalid_digit': 0}
            temp = temp_board[row][col]
            temp_board[row][col] = 0
            if not is_placement_possible(y=row, x=col, n=temp, board=temp_board):
                return {'is_valid': False, 'invalid_row': row, 'invalid_column': col, 'invalid_digit': temp}
            temp_board[row][col] = temp
    return {'is_valid': True, 'invalid_row': None, 'invalid_column': None, 'invalid_digit': None}

def is_placement_possible(y, x, n, board):
    for i in range(0, 9):
        if copied_board[y][i] == n:
            return False
        if copied_board[i][x] == n:
            return False
    subsquare_x = (x//3)*3
    subsquare_y = (y//3)*3
    for i in range(3):
        for j in range(3):
            if copied_board[subsquare_y+i][subsquare_x+j] == n:
                return False
    return True
```

*Prove that for all inputs, the algorithm correctly determines whether the input is valid or not.*

*Prove the algorithm terminates*

At maximum, this algorithm will loop over all 81 cells of a provided Sudoku board and check each one to be valid. If it locates an invalid cell prior to completing all 81 cells, it will terminate and return that 'is\_valid' for the board is False. If the algorithm does not terminate at any of the prior 81 cells, then it will terminate at the end of the algorithm by returning that 'is\_valid' is True for this particular board. Therefore we have shown that the algorithm will always terminate in either a True or False value for 'is\_valid'.

- 1) If the value of any particular cell is 0, then the algorithm will terminate. This is due to the fact that no valid Sudoku board contains a zero. In the algorithm, it can be seen that if this is the case, the algorithm returns that the 'is\_valid' value is False.
- 2) If the value placed in any particular cell is not a possible input (the value is already found in the row of that cell, the column of that cell, or the subsquare of that cell), then the algorithm will terminate. This is due to the fact that no valid Sudoku board can contain a value that is repeated in the same row, column or subsquare. In the algorithm, it can be seen that if a repeated value is encountered, the algorithm returns that the 'is\_valid' value is False.
- 3) If the algorithm has investigated all possible cells on the board, and determined every cell to be valid, then the entire board has been determined to be valid. This can be seen since the end of the 'is\_valid\_sudoku' function returns an 'is\_valid' value of True, and this return statement is only accessible after all cells of a provided board have been examined and determined to be valid. Thus, the algorithm will always stop after completing x loops over the rows and y loops over the columns.

- 4) Prove that the algorithm determines a valid solution to each posed board. This is proven using the loop invariant (the property of the loop that should always be true). In this case, the loop invariant is that after looping over  $x$  rows and  $y$  columns, the 'board' that contains cells with rows  $\leq x$  and column values  $\leq y$  will only contain valid inputs (if the algorithm has not terminated into one of the above-mentioned base cases). Put another way, the loop invariant is that if each cell that has been inspected up to the current cell have been valid, then all cells up the current cell are valid.

- a. Prove the base case that if the value of any particular cell is 0, then the algorithm will terminate. Given a provided board where the top left cell is 0 (row 0, column 0), we can walk-through the algorithm to determine the result.

```
def is_valid_sudoku(board_to_test):
    for row in range(9):
        for col in range(9):
            if temp_board[row][col] == 0:
                return {'is_valid': False, 'invalid_row': row, 'invalid_column': col, 'invalid_digit': 0}
```

In this part of the algorithm, we can see that if the board\_to\_test contains the topleft cell as zero, then a temporary duplicate of the provided board\_to\_test array is created. No differences have been incorporated into the duplicate. Then, the outer loop will enter at index 0 for the row range and 0 for the index of the column value. The algorithm will check to see if the value of the cell at the 0<sup>th</sup> row and 0<sup>th</sup> column is equal to 0. Since it is in this case, the algorithm will return that the function is not Valid via the "is\_valid" key in the result dictionary. Thus, we can see that this base case is addressed by the algorithm.

- b. Prove the 2<sup>nd</sup> base case. That is, prove that if the algorithm is provided a sudoku board with an impossible placement that the algorithm returns False. The board can be invalid based on a repeated value in the same row, the same column, or the same subsquare. Proving this base case correct requires a proof of each of these three cases. In each of these cases, the first base case (the value being 0) will not be returned. Thus the algorithm will enter the is\_placement\_possible subroutine to verify that each cell is valid.

```
def is_placement_possible(y, x, n, board):
    for i in range(0, 9):
        if copied_board[y][i] == n:
            return False
        if copied_board[i][x] == n:
            return False
    subsquare_x = (x//3)*3
    subsquare_y = (y//3)*3
    for i in range(3):
        for j in range(3):
            if copied_board[subsquare_y+i][subsquare_x+j] == n:
                return False
    return True
```

1. Provided an array of arrays with the following:

- Value of the cell in the 0<sup>th</sup> row and 0<sup>th</sup> column equals 1
- Value of the cell in the 0<sup>th</sup> row and 1<sup>st</sup> column equals 1

In this case, checking the value of the 0<sup>th</sup> row and 0<sup>th</sup> column will return False since 1 will be found within `copied_board[i][x]`.

2. Provided an array of arrays with the following:

- Value of the cell in the 0<sup>th</sup> row and 0<sup>th</sup> column equals 1
- Value of the cell in the 1<sup>st</sup> row and 0<sup>th</sup> column equals 1

In this case, checking the value of the 0<sup>th</sup> row and 0<sup>th</sup> column will return False since 1 will be found within `copied_board[y][i]`.

3) Provided an array of arrays with the following:

- Value of the cell in the 0<sup>th</sup> row and 0<sup>th</sup> column equals 1
- Value of the cell in the 1<sup>st</sup> row and 1<sup>st</sup> column equals 1

In this case, the subsquare values of  $x=0$  and  $y=0$  will be determined to be:

$$\text{subsquare\_x} = (0//3)*3 = 0$$

$$\text{subsquare\_y} = (0//3)*3 = 0$$

Then, for each  $i$  in 0, 1 and 2 and for each  $j$  in 0, 1, and 2, `copied_board[subsquare_y+i][subsquare_x+j] == n` will be run.

When  $i=1$  and  $j=1$ , `copied_board[subsquare_y+i][subsquare_x+j]` will equal to `copied_board[0+1][0+1]`, which is equal to `copied_board[1][1]`, which will return the value of 1 which is in that cell. Since 1 is equal to  $n$  where  $n$  is the provided digit from the initial cell, this will return False.

iii) Finally we need to prove the final base case. This is that if the algorithm is provided a valid Sudoku board, it will return that the algorithm is valid. Given a provided board there all cells meet the conditions of the `is_placement_possible` subroutine, and no values are zero, we can walk through the algorithm to determine its correctness.

```
def is_valid_sudoku(board_to_test):
    for row in range(9):
        for col in range(9):
            if temp_board[row][col] == 0:
                return {'is_valid': False, 'invalid_row': row, 'invalid_column': col, 'invalid_digit': 0}
            temp = temp_board[row][col]
            temp_board[row][col] = 0
            if not is_placement_possible(y=row, x=col, n=temp, board=temp_board):
                return {'is_valid': False, 'invalid_row': row, 'invalid_column': col, 'invalid_digit': temp}
            temp_board[row][col] = temp
    return {'is_valid': True, 'invalid_row': None, 'invalid_column': None, 'invalid_digit': None}
```

In this case, for every cell that is visited, the value will not be zero so the first base case will not be returned. The value of the cell in the  $i$ th row and  $j$ th column will be saved as 'temp'. The value of the cell in the  $i$ th row and  $j$ th column will be assigned to be equal to 0. The result of `is_placement_possible()` placing temp in the  $i$ th row and  $j$ th column will return True, so the second base case will not be executed. The value of the cell will be returned to temp (the initial value). This process will proceed for all cells. At the end, the algorithm will return that the solution is valid.



*Inductive Step*

We will assume that the loop invariant (after looping over  $x$  rows and  $y$  columns, the 'board' that contains cells with rows  $\leq x$  and column values  $\leq y$  will only contain valid inputs) is true for some  $k$ th cell of an arbitrary Sudoku board. Thus, the value of the  $k_x^{\text{th}}$  row and the  $k_y^{\text{th}}$  column is possible. Using this assumption, we will prove that the loop invariant is also true for the  $k+1$ th cell in the Sudoku board (the cell immediately following the  $k$ th cell). After looping through  $k$  cells of the board, the board should check whether the  $k+1^{\text{st}}$  cell contains a valid digit. After  $k$  repetitions, `is_valid_sudoku` has not returned on any of the base cases. The inner code will execute on the  $k_{x+1}^{\text{th}}$  row and the  $k_{y+1}^{\text{th}}$  column (representing the  $k+1$ th cell) :

```
if temp_board[row][col] == 0:
    return {'is_valid': False, 'invalid_row': row, 'invalid_column': col, 'invalid_digit': 0}
temp = temp_board[row][col]
temp_board[row][col] = 0
if not is_placement_possible(y=row, x=col, n=temp, board=temp_board):
    return {'is_valid': False, 'invalid_row': row, 'invalid_column': col, 'invalid_digit': temp}
temp_board[row][col] = temp
```

If the value in the  $k+1$ th cell is not valid (either by containing a zero or containing a repeated value from the same row, column or subsquare, the algorithm will return that the board is not valid. If the value in the  $k+1$ th cell is valid, the loop will continue. If the  $k+1$ th cell was the last cell in the Sudoku board, then the loop is complete and the function will return that the board is valid. We can see that the loop invariant holds true (after looping over  $x+1$  rows and  $y+1$  columns, the 'board' that contains cells with rows  $\leq x+1$  and column values  $\leq y+1$  will only contain valid inputs, or will have returned False). We can confidently say that the loop invariant holds for all positive integers of  $k$ .

Since we have shown that the loop will always terminate, and that at each termination the `is_valid_sudoku()` function will return whether the provided board is valid or not based on the current looped cell, we have shown that the algorithm both terminates and produces the correct answer when it terminates. Therefore we have proved the correctness of the verification algorithm.

*Note that this algorithm does not verify whether the value of any cell is outside the range of possible sudoku values (1 through 9, or 0 as a placeholder for empty). If somehow the board was provided some character other than 1 through 9 or 0, the algorithm may return false positives. As per piazza post @298, input validation is not required for this portfolio project.*

question @298

99 views

Input validation

If my game rules clearly state the expected input from a user, does my verification algorithm have to validate the input before checking if the solution is correct?

portfolio

edit

good question 0

Updated 7 days ago by Andrew Dorgan

the students' answer, where students collectively construct a single answer

Click to start off the wiki answer

the instructors' answer, where instructors collectively construct a single answer

It is up to you if you want to validate the input. But, I would say keep it to last. If you are done with everything else and have more time you could do it. But, it is not required.

thankst 0

Updated 7 days ago by Doshna

## Step 6) Discuss the Time Complexity of the Verification Algorithm

<pre>def is_valid_sudoku(board_to_test):     for row in range(9):         for col in range(9):             if temp_board[row][col] == 0:                 return {'is_valid': False, 'invalid_row': row, 'invalid_column': col, 'invalid_digit': 0}             temp = temp_board[row][col]             temp_board[row][col] = 0             if not is_placement_possible(y=row, x=col, n=temp, board=temp_board):                 return {'is_valid': False, 'invalid_row': row, 'invalid_column': col, 'invalid_digit': temp}             temp_board[row][col] = temp     return {'is_valid': True, 'invalid_row': None, 'invalid_column': None, 'invalid_digit': None}</pre>	<p>9 operations 3 operations 1 operation</p> <p>(~27 operations)</p>
--	--

This algorithm will always run on a board of size 9 rows by 9 columns. Since  $n$  (the input size) is not changing, it is not entirely appropriate to use Big O complexity to analyze the time complexity of the verification algorithm. Big O time complexity is meant to assist in determining the time complexity as the input size grows large, which is not relevant in this case.

However, we can discuss the actual complexity due to the number of operations taken by the algorithm while determining a valid solution.

At Best Case, the board is not valid with a zero in the 0<sup>th</sup> column and 0<sup>th</sup> row (the top left cell of the Sudoku board). In this case, the Sudoku `is_valid_sudoku` algorithm will perform the following actions:  
Construct a new list. This will take 81 operations as the list will be constructed using 81 values from the provided board.

Enter the 'row' loop at index 0. This will take 1 operation.

Enter the 'col' loop at index 0. This will take 1 operation.

Index the `temp_board` at the [0<sup>th</sup>] row. This will take 1 operation.

Index the `temp_board` row at the [0<sup>th</sup>] column value. This will take 1 operation.

Obtain the value at the indexed column at the indexed row of `temp_board`. 1 operation

Check to see if this value is equal to 0. This will take 1 operation.

In this case, the inner return will end the algorithm, returning `'is_valid' = False`  
(returning takes 1 operation)

Clearly, the process of the Best Case will take a fixed number of operations.

## Asa Holland - CS325 - Portfolio Project - March 8, 2021

In the Worst Case, the board is entirely valid and will need to check all cells to confirm they are valid. In this case, the Sudoku `is_valid_sudoku` algorithm will perform the following actions:

Construct a new list. This will take 81 operations as the list will be constructed using 81 values from the provided board.

Loop over 'row' loop 9 times. This will take 9 operations.

Loop over the 'col' loop 9 times. This will take 9 operations.

Index the value at this cell and check if it equal to 0 (3 operations)

Save the value of the indexed cell to temp (3 operations: index, index, assignment)

Assign the value at the indexed cell to 0 (3 operations: index, index, assignment)

Perform the `is_placement_possible` subroutine an compare to not

Assign the value at the cell to temp (3 operations: index, index, assignment)

Perform the outer return, which will end the algorithm and retuning 'is\_valid' = True (1 operation)

<pre>def is_valid_sudoku(board_to_test):     for row in range(9):         for col in range(9):             if temp_board[row][col] == 0:                 return {'is_valid': False, 'invalid_row': row, 'invalid_column': col, 'invalid_digit': 0}             temp = temp_board[row][col]             temp_board[row][col] = 0             if not is_placement_possible(y=row, x=col, n=temp, board=temp_board):                 return {'is_valid': False, 'invalid_row': row, 'invalid_column': col, 'invalid_digit': temp}             temp_board[row][col] = temp     return {'is_valid': True, 'invalid_row': None, 'invalid_column': None, 'invalid_digit': None}</pre>	<p>9 operations</p> <p>9 operations</p> <p>3 operations</p> <p>3 operations</p> <p>3 operations</p> <p>1 op +subroutine</p> <p>3 operations</p> <p>1 operation</p> <p>(~7,100 operations)</p>
<pre>def is_placement_possible(y, x, n, board):     for i in range(0, 9):         if copied_board[y][i] == n:             return False         if copied_board[i][x] == n:             return False     subsquare_x = (x//3)*3     subsquare_y = (y//3)*3     for i in range(3):         for j in range(3):             if copied_board[subsquare_y+i][subsquare_x+j] == n:                 return False     return True</pre>	<p>9 operations</p> <p>3 operations</p> <p>3 operations</p> <p>1 operation</p> <p>1 operation</p> <p>3 operations</p> <p>3 operations</p> <p>5 operations</p> <p>1 operation</p> <p>(~75 operations)</p>

In the worst case, the `is_placement_possible` subroutine copies the provide board (81 operations), iterates over through each index and checks if the value at each row and column is equal to the same provided number (are cell values duplicated within the same row and column?). Then the subroutine determined the subsquare location and iterates over all 9 cells of the subsquare, checking if those cells contain the provided digit. In the worst case, this subroutine will return True for all provided cells on the provided board. Clearly, in the worst case, the subroutine will perform a fixed number of operations.

Clearly, the process of running the algorithm through the Best Case will take a fixed number of operations.

This means that the actual complexity of the algorithm will vary based on whether the provided board is invalid (and how it is invalid) or if it is valid. The number of operations will range from roughly 100 operations in the Best Case to roughly 7,100 operations in the worst case.

**Bonus B) Discuss the Time Complexity of the Solving Algorithm**

<pre>def find_empty_location(board, coordinates):     for y_i in range(9):         for x_j in range(9):             if board[y_i][x_j] == 0:                 coordinates[0] = y_i                 coordinates[1] = x_j                 return True      return False</pre>	<p>9 operations  9 operations  3 operations  2 operations  2 operations  1 operation  1 operation</p> <p>Worst Case: ~240 operations  Best Case: ~10 operations</p>
<pre>def is_placement_possible(y, x, n, board):     for i in range(0, 9):         if copied_board[y][i] == n:             return False         if copied_board[i][x] == n:             return False      subsquare_x = (x//3)*3     subsquare_y = (y//3)*3     for i in range(3):         for j in range(3):             if copied_board[subsquare_y+i][subsquare_x+j] == n:                 return False      return True</pre>	<p>9 operations  3 operations  3 operations  1 operation  1 operation  3 operations  3 operations  5 operations  1 operation</p> <p>Worst Case: ~75 operations  Best Case: ~5 operations</p>
<pre>def solve_sudoku(board_to_solve):     copied_board = board_to_solve.copy()     current_coords = [0, 0]     if not find_empty_location(board=copied_board, coordinates=current_coords):         return True      [row, col] = current_coords     for digit in range(1, 10):         if is_placement_possible(y=row, x=col, n=digit, board=copied_board):             copied_board[row][col] = digit             if solve_sudoku(board_to_solve=copied_board):                 return True             copied_board[row][col] = 0      return False</pre>	<p>81 operations  1 operation  1 operation + find_empty_location subroutine  1 operation  1 operation  9 operations  1 operation + is_placement_possible subroutine  3 operations  1 operation + recursive solve_sudoku call  1 operation  3 operations  1 operation</p>

My implementation of the solving algorithm relies on three functions, a solving function as well as two supporting subroutines.

The first subroutine `find_empty_location` iterates over the provided Sudoku board and searches for empty cells. If it finds an empty cell, it updates a pair of coordinates with the position of that cell and returns True (an empty cell has been found). In the worst case, this will iterate over all cells and determine that none are empty, returning False. In either case, there is standard range of operations that could be performed by this subroutine, depending on how many cells need to be searched to locate an empty cell.

The second subroutine is the same `is_placement_possible` subroutine used in my verification algorithm. This has the same range of operations as previously discussed, and is dependent on whether or not the value at a particular cell is valid or not, and if not, in which way it is invalid.

The main `solve_sudoku` function is a recursive, backtracking algorithm which relies on the other two subroutines.

The first base case of `solve_sudoku` is if no empty locations are found by `find_empty_location`. Since `solve_sudoku` is only provided unfinished puzzles, if there are no empty cells, that means that the only way this base case has been reached is if the entire puzzle has been solved with valid input. Therefore we can return True. This will (in the worst case) take approximately 320 operations (~80 operations in `solve_sudoku` + ~240 operations in `find_empty_location`).

The second base case of `solve_sudoku` is if an empty cell is located using 1 instance of `find_empty_location` (which will be from a fixed number of operations based on how many cells needed to be examined) and all digits from 1 to 9 are determined to NOT be valid options for that cell by `is_placement_possible` (the worst case, approximately 75 operations each time). In this case, we know that there either the puzzle is impossible or a mistake was made earlier when assigning digits to cells. Since there will be 9 loops where `is_placement_possible` is called, this will roughly  $(9 * 75) + x$  operations + ~85 where  $x$  is between 10 and 235 based on the operations performed by `find_empty_location`. Therefore in the worst case (where the last empty location is the last cell in the sudoku board, taking ~235 operations to determine) this base case will perform ~1,000 operations.

The recursive call in `solve_sudoku` occurs when an empty cell has been identified by `find_empty_location` and a viable digit option from 1 to 9 has been identified using `is_placement_possible`. In this case, we will place the value in the cell and then call `solve_sudoku` to see if the board can still be solved using the placement. If it is valid, we will solve the puzzle. If it is not valid, then we will return False, and backtrack by removing the placed value and placing a new digit in its place. Although not realistic, an upper bound for this recursion would be the hypothetical case in which we would need to recurse on every cell, for every possible input. Therefore, we can generalize the problem to say that a recurrence relation defines the complexity of this recursive call. The recurrence relation is as follows:

$$T(n) = 9T(n - 1) + O(1)$$

Each Sudoku cell has 9 possible values that must be checked. We can generalize to say there can be  $n$  unknown empty cells on a provided Sudoku board for which we are attempting to place a value. Each time we place a value in a cell, the number of unknown values decreases by 1 and we recurse onto that decision. There is a standard amount of operational time complexity associated with each step of the recursion.

We can simplify the recurrence relation to show that it is upper bounded by  $9^n$  possible operations where  $n$  is the number of cells that are empty and need assignment. Although this is exponential growth, this is deceptive because it is not a realistic bound based on the types of provided Sudoku boards. Realistically the number of operations performed would be much less than that. A fully blank board with the standard size of a 9x9 grid would have  $9^{81}$  operations plus some constant amount of operations. But that wouldn't make for a particularly fun puzzle.