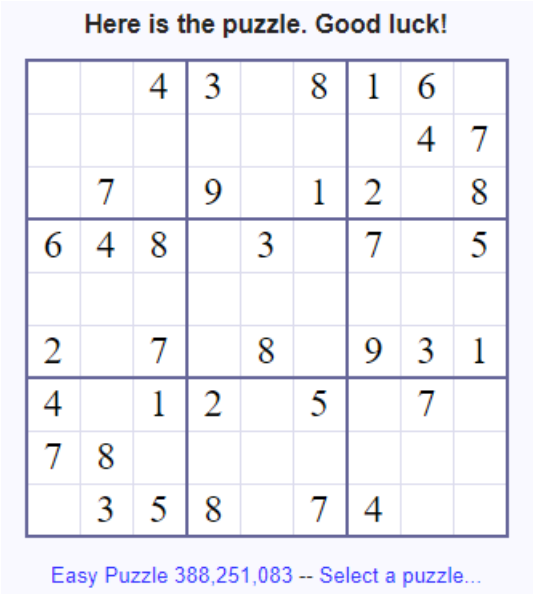
Theresa Bolaney
Portfolio Project – Sudoku
CS 325

## Sudoku Puzzle and Rules

I have chosen Sudoku as my puzzle. The rules for Sudoku are pretty straightforward. The player is given a 9x9 grid of squares that is partially filled with numbers 1-9. Their goal is to fill out the board such that each column, row, and 3x3 mini-grid have all numbers 1-9 represented. There can be no conflicts, such as repeat numbers or missing numbers, in any of the columns/rows/mini-grids.

Here is an example puzzle from websudoku.com:




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

For my sudoku puzzle program, the user must follow the rules stated above and "fill in" the puzzle. They will see a text representation of the puzzle approximating (though sadly with no GUI) a standard sudoku puzzle you could find online or in a newspaper. The user fills in the puzzle by entering each row at the command line in the following format: example 1,2,3,4,5,6,7,8,9. Only the numbers 1-9 may be entered. The entry format is very important as the program does not check for strings, values outside of 1-9, missing/extra values, etc. After entering the rows, the program will return whether the user has correctly solved the puzzle or failed the game.

The correct solution has been provided in the README for easy testing. Only one puzzle exists in the game and it is hardcoded. (The verification is not hard coded and is its own set of routines.)

## Correctness of Verification Algorithm

The verification algorithm consists of three parts: checking that all rows, all columns, and all mini-grids contain only the numbers 1-9. If any of these three checks fails at any point, the puzzle must be incorrect.

To prove that this algorithm works, let's start by discussing the rows. Here is my code for this portion of the algorithm:

Theresa Bolaney
Portfolio Project – Sudoku
CS 325

```
#This function will check that each row contains the values 1 through 9 and return True or False
def checkRows(puzzle):
    for i in range(9):
        row = puzzle[i]
        for counter in range(9):
            if (counter+1) not in row:
                return(False)
    return(True)
```

This function increments a counter so that it passes over the values 1-9. It compares this counter against the row that it is evaluating and uses the "not in" phrase to detect if the number exists or not on the line. If it does not exist, it exits the algorithm with "False". If the number does exist, the counter increments and the next number is searched. If all numbers are searched and found in each row, the function passes "True" on to the next phase.

By this logic, we know that any row that contains the values 1-9 in any order will pass. Note that this relies on the assumption, as stated in the rules, that the user is only entering a row with nine values composed of the numbers 1-9 which is congruent with standard Sudoku rules.

An interesting thing to consider is that any time we have a duplicate number, we must also have a missing number. This is due to our restriction that exactly nine values be entered for each row. For example, if we decided to duplicate the value 2, a possible row could look like this: 1,2,3,2,5,6,7,8,9 (any order is fine). The 4 was arbitrarily removed in this example to ensure the length was not exceeded. The algorithm does not explicitly care about duplicates but it absolutely cares about any missing values. So, since any time we have a duplicate we must also have a missing value, these cases will always be caught.

Next we consider the function that checks the columns:

```
#This function will check that each column contains the values 1 through 9 and return True or False
def checkColumns(puzzle):
    for i in range(9):
        column = []
        for j in range(9):
            column.append(puzzle[i][j])
        for counter in range(9):
            if (counter+1) not in column:
                return(False)
    return(True)
```

This function follows the same logic as above; we must have all of the number 1-9 in each column, once again verified by comparing against a counter. If this occurs, the function returns "True." But if at any point the function searches for a number and cannot find it, it will send "False" back as a response.

Having all the rows and all the columns come back clean is not enough – we must also verify that each 3x3 mini-grid only has the values 1-9 present. That is achieved with this function:

```python
#This function is a little trickier than the two above. It must check that each mini-grid (3x3 section) within
#the puzzle only contains the values 1-9.
def checkMiniGrids(puzzle):
    minigrid = []
    for i in range(3):
        for j in range(3):
            minigrid.append(puzzle[i][j])
    for counter in range(9):
        if (counter+1) not in minigrid:
            return(False)
    return(True)
```

The function loops through the puzzle and transforms each 3x3 section into a simple list. It then loops through a counter to (you guessed it) evaluate if all values are present. All of the logic used above applies here as well.

And so, if all three of these functions return "True" for our algorithm, the user wins.

```python
#If all conditions are met for the rows, columns, and minigrids, then the user wins! Win or lose, the status is printed
#to the console.
def verifyPuzzle(puzzle):
    if (checkRows(puzzle) and checkColumns(puzzle) and checkMiniGrids(puzzle)):
        print("Congratulations, you win!")
    else:
        print("Sorry, this is not the correct solution. You can play again by restarting the program.")
```

## Verification Algorithm Time Complexity

Let's consider the checkRows function again. The function will stop looping only when it either hits a fail condition (it cannot find the counter number in the row) or it has successfully evaluated all counter values 1-9 in each row. So, our successful case have the longest running time.

A search occurs each time "if (counter+1) not in row" is called. According to information from wiki.python.org/moin/TimeComplexity, the "x in s" operator used here has an average case run time of O(n) when used in a list. This operation occurs 81 times (nine times in each of the nine rows) giving this first part of the algorithm a run time of O(81n).

This logic holds true for the column and mini-grid portions as well, since we have shown that they work in a very similar manner in the section above. Overall, this means our running time is O(81n + 81n + 81n). But as we learned in an early module, we are able to drop coefficients. This gives us an overall run time of O(n) for the entire verification algorithm.

## Bonus B: Time Complexity of a Solver Algorithm

Though I was unable to implement my own solver algorithm, I would like to discuss a potential one using a simple algorithm:

1. Find the first blank cell in the puzzle
2. Consider the number 1
    a. Verify 1 is not already in the row
    b. Verify 1 is not already in the column
    c. Verify 1 is not already in the mini-grid

3. If true for all conditions, place 1 in the blank cell and move on to the next blank cell; if false for any condition, increment the number and repeat step 2
    a. If the increment attempts to pass 9, go back to the previous cell that was filled out and increment it by 1, then continue step 2
4. Stop when the last blank cell is correctly filled in

The time complexity is pretty horrible. Breaking it down, the first step is negligible. The next step should run in polynomial time if it uses the "x set in" operator to check the row/column/minigrid. Assuming things are fine, we move on to step 3 and occasionally need to increment. All of these increments and checks are polynomial at most. But imagine if you fill out multiple squares and for the latest one the algorithm attempts to exceed 9. This is when we need to go back, because it means something was incorrect about one of the previously filled values. After we backtrack and increment, we resume with our previously problematic cell and once again start from the ground up evaluating if any given number can work. So, we are now *repeating* our previously polynomial searches, possibly another 9 more times before we have to once again backtrack and increment. But, we don't immediately know where things went wrong. Hypothetically, this could happen in such a way that the algorithm now tries to exceed 9 for the previous square, then for the square behind that, and on.

If we have just one square, we would do our polynomial checks against the rows/columns/mini-grids up to 9 times. If we have two squares, we could do 9 searches in the first cell and then 9 *for each of the values in the first cell.* So, that would be 81 (9^2) times. And so on for the 3rd cell (9^3), etc. One row alone could force 9^9 operations.

So, how many spaces are blank? That is going to depend on the puzzle. But overall, the worst case scenario is that we need to do all possible backtracks for each open space, which we could write as O(9^n).