Jared Keirns CS325 Portfolio Project

Game Description:

The program runs the 9x9 sudoku game with one fixed puzzle. The goal of the game is to fill out every row, column, and sub-square of the board with integers 1-9 with no integers repeating. The structure of the "board" is 9 lists of 9 elements, or a 9x9 matrix. The game begins as soon as the program is ran. The player will be displayed a board with some spaces already containing correct numbers that cannot be changed. The empty spaces are represented with a "0". The program will immediately ask the player for the row that they would like to select. At this point, the player can enter an integer 1-9 to designate the row (1 being the top and 9 being the bottom), or they can enter "ANSWER", which will replace the player's board with a completed, correct board and send it to the "game_end" function that checks to see if a completed board is correct (This was implemented for testing purposes, there is no prompt within the program, it was left in as a potential "ease-of-use" grading tool). The player will then be asked to enter the column containing the location that they wish to use with a 1-9 integer(1 being furthest left, 9 being furthest right). Lastly the program will ask for the value that the player would like to place. If the row and column identify a place that was already identified on the original board, the program will not change the value and start the inquiry over. The player can identify a place that they had previously set a value and change it. After every move, the program runs an n-size for loop within an n-size for loop, e.g.:

```
for columns in rows:
    for elements in columns:
```

To check each place for a "0". If a "0" is found, the program will print the player's current board and ask for another move. If a "0" is not found, meaning every position is filled with a value, the player's board is sent to the game_end function for validation.

Determination Algorithm:

The game_end function represents the determination problem of a sudoku puzzle. Given a completed board, how can its correctness be checked? The game_end algorithm has three distinct phases. The idea is to check each row, then column, and lastly each sub-square for the integers 1-9, with none repeating. Checking the rows is the first check completed. It is the easiest and most intuitive portion of the algorithm given its matrix structure. It is handled in much of the same way that the check to see if the board was complete or not:

```
for rows in player_grid:        # n size loop
```

```
test_list = [1, 2, 3, 4, 5, 6, 7,  8, 9}        # Used to check for 1-9 completion and repetition.
For elements in rows:                # n size loop
        If elements in test_list:
                test_list.remove(elements)
        else:
                return incorrect
```

There are n checks for n rows with smaller, constant-time, operations. So, the time to check the rows is O(n^2). The next phase of the algorithm is to check each column for integers 1-9 with none repeating. The algorithm becomes a little less intuitive, because instead of just checking each list from front to back, a constant index, let's say i, is required to check the same location in each list. So, the first position is of each list is checked, it loops back, the second of each list, third and so on:

```
i = 0
j = 0
while j <= # of rows:                # n size loop.
        test_list = [1, 2, 3, 4, 5, 6, 7,  8, 9}        # Used to check for 1-9 completion and repetition.
        for rows in player_grid:             # n size loop.
                if rows[i] in test_list:
                        temp = rows[i]
                        test_list.remove(temp)
                else:
                        return incorrect
        i ++
        j ++
```

Like checking the rows, checking the columns also requires an n size loop within an n size loop, keeping the runtime at O(n^2). The most complicated portion of the algorithm comes with checking each sub-square. A sub-square can be defined as an nth divisible portion of the larger board. For example, if you had an n x n sudoku board, you would not only have n rows of n elements, n columns of n elements, but also n sub-squares of n elements. The player's puzzle is a 9 x 9 board, so the sub-squares are neatly separated and identifiable as: top-left, top-mid, top-right, mid-left, mid-mid, mid-right, bottom-left,

bottom-mid, bottom-right. So, to reiterate the goal is to check each of these sub-squares (each containing 9 spaces), for integers 1-9 with none repeating. The below example is the algorithm for one specific sub-square (top-left). The variables j and k need to be set for every specific sub-square, variable g will be used to keep track of the sub-squares that are evaluated, so the following portion of code is repeated 9 times with minor alterations that will not significantly impact runtime:

```
j = 0
k = 0
g = 0
while g < 1:                # A termination loop that only runs once.
        test_list = [1, 2, 3, 4, 5, 6, 7,  8, 9}       # Used to check for 1-9 completion and repetition.
        while k <= 2:               # Loop that will have n/3 iterations
                while j <= 2:               # Loop that will have n/3 iterations.
                        if players_grid[k][j] in test_list:
                                temp = player_grid[j][k]
                                test_list.remove(temp)
                                j ++
                        else:
                                return incorrect
                j = 0
                k ++
        j = 3               # This is setup for the next sub-square evaluation.
        k = 0
        g = 1
```

The algorithm is starting in the very top left position of the puzzle and checks the first three numbers within that row. Then the algorithm looks at the next row down and checks the first three consecutive numbers within the row. Lastly, the algorithm checks the first three numbers in the next row. If every item is removed from the “test_list” and there are no repeats, then the top-left sub-square is good. Note that variable j is changed to 3 at the end of the example code because it will be starting at column 4 and working through column 6 during the evaluation of the next sub-square, which would be top-mid. Because of the smaller evaluation area, the above code contains two loops, each of which only having n/3 evaluations, making the runtime of the portion O(n). However, the above code only accounts for the

evaluation of one sub-square. Being that there are n sub-squares within an n x n sudoku board, the runtime for checking all sub-squares is O(n^2). So, for each portion of the determination function, checking rows, checking columns, and checking sub-squares for correctness, each portion took O(n^2), making the complexity of the entire algorithm itself O(n^2). Simply stated, given any correct sudoku grid, the algorithm can verify its correctness in polynomial time. Now, because we know that the determination problem can be solved in polynomial time, we know that the problem is in both P and in NP.

ReadMe for sudoku.py file

A sudoku game program with one board. The player can place valid inputs (integers 1-9), into valid locations within the board that were not occupied by an 'original' number (numbers that were already placed on the original board). When all the board locations are occupied, the program will run a validation function to see if the board is correct. The board is correct if each row contains numbers 1-9 with none repeating, each column contains numbers 1-9 with none repeating, and each sub-square contains numbers 1-9 with none repeating.

The program will begin by displaying the board and asking the player for the row they would like to select. The player can enter any integer 1-9 where 1 is the top row and 9 is the bottom row. Next, the program will ask the player to enter the column they would like to select. The player can enter any integer 1-9, where 1 is the leftmost column and 9 is the rightmost column. Lastly the player will be asked to enter a value to fill the spot identified by their coordinates. If the location identified by the player belongs to an 'original' number, the program will ask for new coordinates. The player can however identify a number that they themselves has placed and change it.

To skip to the end of the game and check the algorithm, the player can enter 'ANSWER' for the program's first request ("Please enter row number: "), and the program will send the 'answer' board to the validation algorithm to check it. This was implemented for easy testing, but was left in for grading. A copy of this ReadMe file was submitted alongside the python.py file named README.txt.