Member	Last Name	First Name	Member	Last Name	First Name
1	Modukuru	Prady	4	Desiraju	Pujitha
2	Sharma	Piyush	5	Montenegro	Martin
3	Vaid	Pushkal	6		

Project Title: Sudoku Solver

Project Definition: Sudoku is a puzzle in which players insert the numbers one to nine into a grid consisting of nine squares subdivided into further nine smaller squares in such a way that every number appears once in each horizontal line, vertical line, and square.

For a 9x9 grid Sudoku problem:

Each row of cells contains digits 1 − 9 exactly once.
Each column of cells contains digits 1−9 exactly once.
Each 3 x 3 block contains digits 1 − 9 exactly once

Project Summary: To build a simulation suite for solving Sudoku puzzles using backtracking and layman's method to visually compare time complexities of efficient algorithms with traditional brute-force methods.

Project Modules:

Module 1: Sudoku Generator

Functions used: permute9, permute81, backtrack_opst, backtrack, random_backtrack, rand_full_sudoku, rand_sudoku

Definition:
Produces a matrix with numbers filled in and with as many blanks as the user chooses. This generator will produce 9x9 Sudoku puzzles in matrix form. This program starts from a blank Sudoku and fills in three 3x3 blocks along the main diagonal with random numbers. It then solves the Sudoku with the algorithms built in module 2 modified to place numbers in a random order. This fully solved Sudoku will then have a unique solution. At this point the program will attempt to create blank cells while keeping the solution of the puzzle unique. First, the program attempts to remove as many cells as possible such that in each removal leaves a blank cell with only one valid possibility. For the remaining removals, the program will run two solvers to check that each removal leaves a unique solution. One solver places numbers in ascending order (backtrack) and the other solver places numbers in descending order (backtrack_opst). If they

converge to the same solution, then the Sudoku has a unique solution. If any removal would result in a Sudoku with multiple solutions, the removal is skipped.

Module 2: Sudoku Solver

Functions used: backtrack, laymansolve, random_backtrack, isAvailable, add, delete, add_feasible,

Definition:
This module takes an input matrix, an unsolved Sudoku matrix generated by module 1, and solves for the unique solution using two different algorithms of varying time and space complexities.

Algorithms used:
Backtracking algorithm:
The crux of this algorithm is to try out possible values to fill in the empty cells of a Sudoku. These values can be possible solutions based on the current constraints placed on the row, column, and 3x3 blocks by the puzzle. As soon as it is unable to enter a valid value in an empty cell, it traces its steps back while clearing previously filled cells and restarting the filling process until the Sudoku is solved using recursion. The algorithm is characterized as being memory efficient and done through brute force. The steps to implement the algorithm are the following:

> Find the first empty cell and add a number that has not been previously filled. If it is impossible to enter a number, due to constraints of the puzzle, return that the given Sudoku has no solution. Otherwise, move on to the next cell.
>
> If the cell is filled, move on to the next, else fill the cell with a valid number. If it is not possible to fill the cell with a number due to constraints of solving, retrace steps and replace values until the cell can be filled. (If impossible there is no solution – Step 1)
>
> If steps one and two are repeatedly executed on a given Sudoku and the algorithm's index ends up moving beyond the given dimensions, then the Sudoku has been successfully solved.

Layman’s algorithm:
The Layman's algorithm is a logic based algorithm that emulates how a non-professional man would approach solving a Sudoku puzzle. The basic structure of the algorithm is as follows:

> The algorithm will visit every empty cell and assign it a priority value based on the number of legal moves it has. The legal moves are found by checking the row, columns, and 3x3 submatrix constraints.
>
> After assigning the priority value for the empty cells, the algorithm will assign the cell with the least priority value. Since the priority value of the empty cells are mostly one for easy puzzles, they can mostly be solved by inputting a value one-by-one.

When least priority number is larger than one, the algorithm will assign the empty cell with the first legal number and then assigns it a tracker variable. It then continues in the same way.

A dead end is encountered when an empty cell has no more legal moves. When this happens, the empty cell with the tracker variable is assigned with the next possible number and will remove all of the cells that were entered after the tracking variable was placed.

The tracker variables record the sequence in which empty cells are given values and are used to modify and delete cells with incorrect solutions.
The resulting Sudoku is solved according to the constraints of the problem.

Module 3: Sudoku Game User Interface

Definition:
This module comprises of the GUI for the Sudoku solver. This includes creating both the computer solving graphics as well as the UI for the player to solve their own Sudoku. A timer is present for each version of the algorithm as well as one for the player to keep track of their own progress. Time complexity data is available for viewing and plots are available to visualize the progress of the algorithm over time.

s

Programming Language, Data Structures/Algorithms used:

Programming Language:
C# (Very close to C++, supports better Visualizations/Graphics) for the user interface.
C/C++ for coding the main program modules (converted into C# for final use).
Algorithms/Data Structures:
Multidimensional Arrays
Backtracking Algorithm
Layman Algorithm

Performance Testing:

In order to test for correctness of the solutions the algorithms produce, we have a matrix generator that generates a completely filled 9x9 valid Solution Matrix and then removes at most 64 elements to generate a unique Sudoku Problem Matrix. Since this is the way we generate a viable Sudoku, we have a viable solution for the puzzle to immediately to check against. The matrix solver algorithm can be tested against this completely filled Sudoku in order to check for correctness. We will also have a timer so that we can show how quickly the algorithms will come up with the solution and can show the resources these solutions will require to run. Also, in order to test the performance of our algorithms and create data to analyze, we used Standard C functions to determine CPU runtime (more is explained in the graphs and figures section).

Time Complexity Analysis:

Backtracking:
The big-O time complexity of the backtracking algorithm is $O(n^m)$, where n is the number of possibilities for each square (i.e 9 in the case for classic Sudoku) and m is the number of spaces that are blank in the puzzle. The derivation for this can be found by working backwards from a single blank. If there is one blank in the entire Sudoku, then there are n (or 9) possible values that can be filled in. If there are five blanks, then you must work through n possibilities for the first blank, n for the second, and so forth through the fifth. This results in n^5 for a Sudoku with five blanks. This algorithm will perform a depth first traversal and search through the possible solutions where the levels represent the choices for a single box. For a Sudoku of depth m, the number of blanks required to be filled, and a n possible values to place in each blank, the worst case scenario will be $O(n^m)$.

Layman:
The solving time for n*n Sudoku puzzles using this algorithm is mainly dependent on the difficulty levels of the puzzles generated. The algorithm visits every blank cell within the matrix, represented by the variable *m*. Since at every blank cell, it checks the corresponding row, column and the square, it checks n*n* n (number of cells within each square). Thereby n^3 times the no of blank cells. After assigning each blank a priority value, we go through all the blanks recursively thereby making it n^{3m} . When taking big-O time complexity, we can equate the above calculated polynomial to $O(n^m)$.

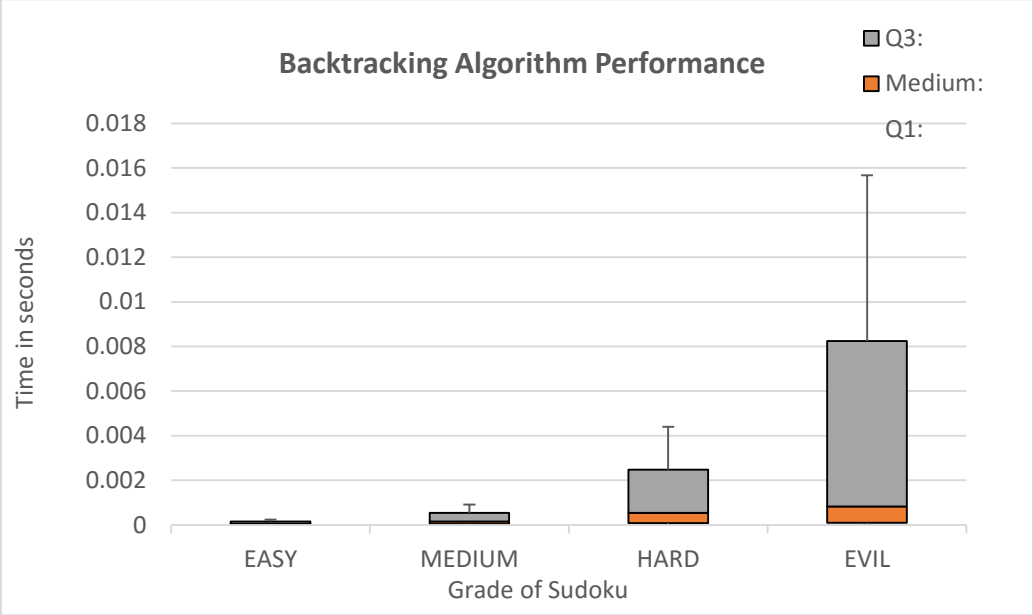
Sudoku Generate:
While going through the Sudoku and checking the add, delete and feasible functions during generate for each of the cells within a Sudoku, the time complexity would be O(1) for each of the functions and since you do so for each cell of the Sudoku , it is O(1)*O(n) where n is the number of cells within the Sudoku. In the next step, you remove cells and also make sure that the Sudoku still maintains a unique solution by solving the Sudoku in that state. Else you move on to the next cell. The next step would have a time complexity of O (n)*2O (n^m) since there are 2 solving algorithms. Therefore the total time complexity would be O (1)*O (n) + O (n)*2O (n^m).

Graphs and Figures:

The figures and charts below were each taken from a data sample of 4000 Sudoku puzzles. The run-time data is based off of actual CPU runtime and was created by utilizing the time functions in the Standard C "time.h" library. Of these 4000 Sudoku's, 1000 of each difficultly level (easy, medium, hard, and evil) were used in order to generate enough reliable data for runtime analysis. We used this data in order to test our complexity analysis and get an idea for the relative efficiencies for our algorithms. Also, the different ratings of Sudoku puzzles were defined as ~48 blanks for easy, ~51 blanks for medium, ~54 blanks for hard, and ~57 blanks for evil (close to the same grading scale used on Web Sudoku Online).

Backtracking
The first algorithm which we ran analysis for was for standard backtracking.

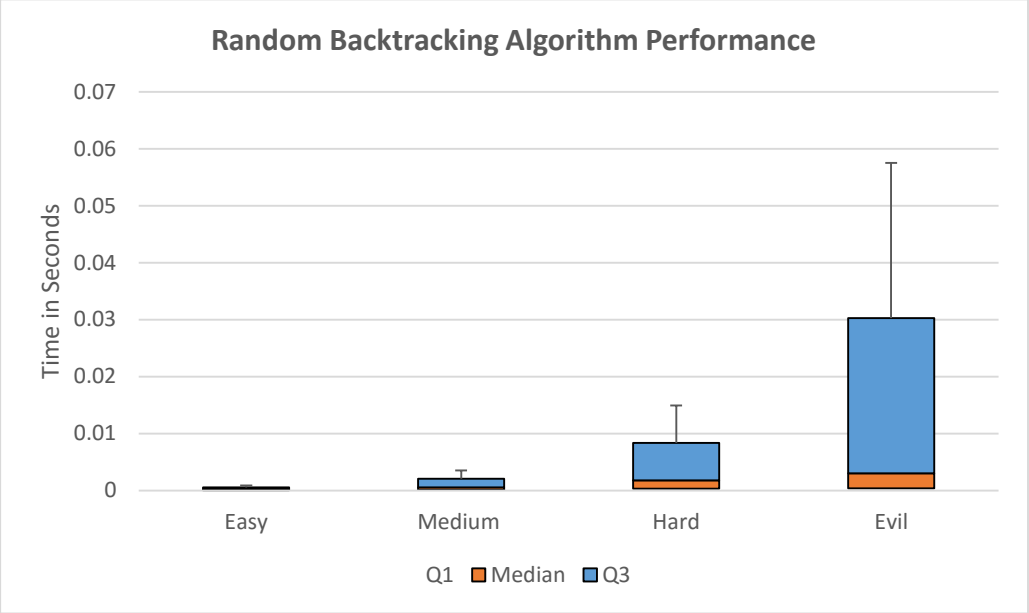
Figure 1:



Grading	Easy (sec)	Medium (sec)	Hard (sec)	Evil (sec)
Average:	0.000139	0.000438	0.003254	0.026428
Min:	0.000004	0.000005	0.000005	0.000005
Max:	0.030197	0.010288	0.213453	5.921399
Median	0.00004	0.000118	0.000453	0.000724

From the above data (in seconds), it is obvious that Evil level puzzles are more time consuming to solve than the rest, as they are on average 190 times more time and CPU intensive using the backtracking algorithm. Also from the graphs, you can see that outliers were relatively rare and, therefore, did not affect the mean and medians drastically. Also from this graph, you can see that the general shape of the graph is going to follow an exponential rise, based on the difficulty of the problem.

Figure 2:

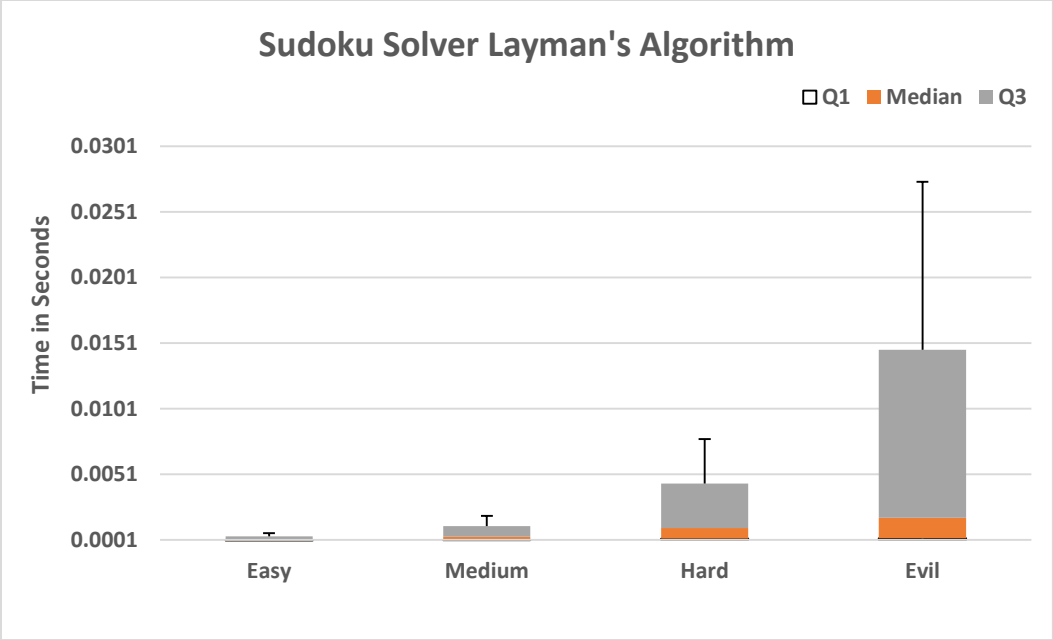


Difficulty	Easy (sec)	Medium (sec)	Hard (sec)	Evil (sec)
Median	0.0001385	0.000437	0.001457	0.002672
Average	0.00042511	0.002221	0.009785	0.083648
Max	0.022723	0.135314	0.975939	5.86858
Min	0.000019	0.000022	0.000026	0.000026

In an attempt to improve performance and see the effects of having a randomized backtracking instead of a linear choosing algorithm, we generated the above data. The idea to randomize the function came from the thought that using random values would actually help narrow down the number of attempts the algorithm would have to successfully fill a cell. We found that using a randomized permutation function that would choose a value 1-9 actually marginally improved performance, but only to a point where the statistical significance might be mute.

Layman

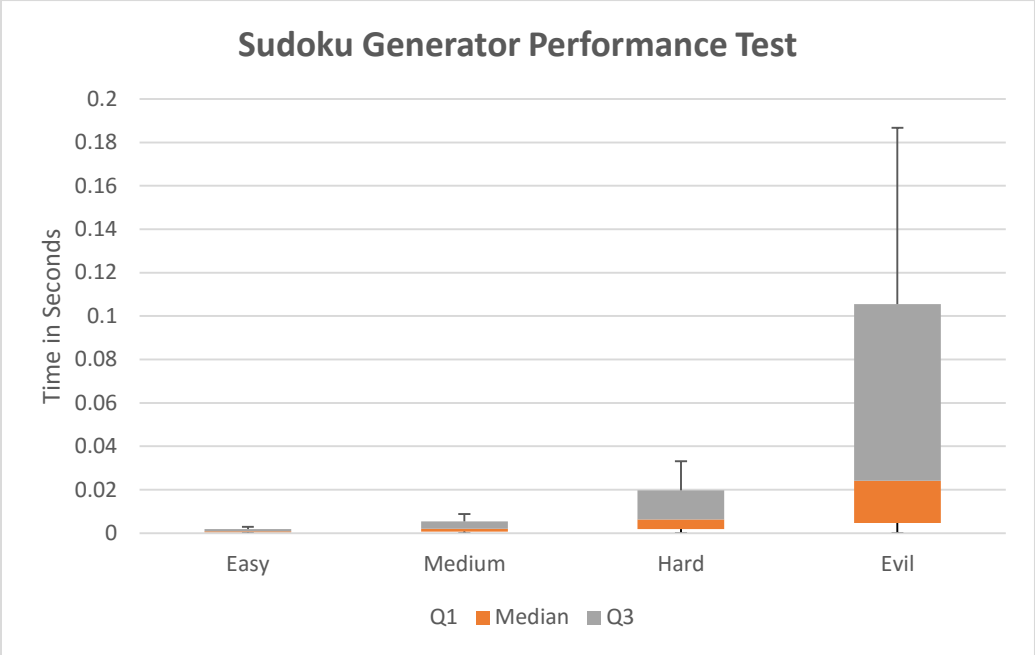
Figure 3:



Difficulty	Easy (sec)	Medium (sec)	Hard (sec)	Evil (sec)
Median	0.000094	0.00026	0.000812	0.001566
Average	0.000243	0.0011378	0.006345	0.054865
Max	0.008675	0.071934	0.386319	8.624124
Min	0.000012	0.000011	0.000014	0.000016

From the data above, Layman's algorithm actually provided better CPU runtimes than the Backtracking and Randomized Backtracking algorithms implemented earlier. From the data for the Evil puzzles, on average the backtracking took 1.7 times longer than Layman. Since the difference between the hard puzzles also ended up giving about a 1.7 ratio, we can conclude that the algorithms have similar time complexities, but run at different linear coefficients.

Figure 4:



Difficulty	Easy (sec)	Medium (sec)	Hard (sec)	Evil (sec)
Median	0.000513	0.001397	0.004548	0.019373
Average	0.001437	0.003378	0.015155	0.140701
Max	0.045231	0.170343	0.492869	9.852388
Min	0.000096	0.000179	0.000153	0.000294

For the above data, we ran the CPU runtime analysis on the generate function and found that the Evil Sudoku puzzles are not just harder to solve, but they are also exponentially harder to create. Also, the amount of time the generator takes to actually produce a new puzzle is more randomized the higher the required difficulty.

Work Distribution:

Martin Montenegro: Generate Sudoku Algorithm, Backtracking Algorithm, Document Preparation

Piyush Sharma: GUI functionality, translation of C code to C#, Document Preparation

Pushkal Vaid: Layman Algorithm, Backtracking Algorithm, Data Collection and testing, Document Preparation

Prady Modukuru: Layman Algorithm, Backtracking Algorithm, Data Collection and testing, Document Preparation

Pujitha Desiraju: Layman Algorithm, Backtracking Algorithm, Data Collection and testing, Document Preparation

Conclusion:

After analyzing the data produced, Layman's algorithm has been proven to be faster, although only on a linear scale. In future studies on Sudoku solving algorithms, we plan on using more advanced techniques in order to future drive down our CPU runtime, as well as improve our generation methods. Also, another aspect of future projects could be to analyze the memory requirements for all type of Sudoku solving algorithms, as well as introduce more complex solutions such as Genetic Algorithm, Constraint programming, and Stochastic Analysis. Some of the algorithms we were interested in but didn’t have the technical know-how to produce were Dancing links, Artificial Bee Colony, and Stochastic Hill Climbing. These advanced projects would be helpful to analyze and determine their pros and cons.

Citations:

Gunduz, A. F., Nugroho, L., & Saranli, A. (n.d.). Sudoku Problem Solving using Backtracking, Constraint Propagation, Stochastic Hill Climbing and Artificial Bee Colony Algorithms-METU 2013. Retrieved April 20, 2016, from https://www.academia.edu/6207354/Sudoku_Problem_Solving_using_Backtracking_Constraint_Propagation_Stochastic_Hill_Climbing_and_Artificial_Bee_Colony_Algorithms-METU_2013

Layman's Algorithm to Solve Sudoku Puzzles. (n.d.). Retrieved April 14, 2016, from http://raydioaktiv.blogspot.in/2013/10/how-laymans-algorithm-work.html

J. (n.d.). Http://www.diva-portal.org/smash/get/diva2:721641/FULLTEXT01.pdf. Retrieved April 10, 2016, from http://www.diva-portal.org/smash/get/diva2:721641/FULLTEXT01.pdf

Zendoku puzzle generation. (n.d.). Retrieved April 26, 2016, from http://garethrees.org/2007/06/10/zendoku-generation/

Solving Sudoku by Backtracking. (2014). Retrieved April 12, 2016, from https://codemyroad.wordpress.com/2014/05/01/solving-sudoku-by-backtracking/