1.
    a. Code is attached as a separate Python file. To run, please use a command of the form `python3.2 Sudoku.py <filename> [bf]
    b. Josiah looked into background research for Sudoku solving, and implemented the recursive solve functions for each of the methods. Josiah also handled file IO and the main method, prompting the user with detailed information. Jason looked into helper methods for the solve methods, such as square validation, the storing of constrained values for forward checking, and consistency checking. Both members, not extremely familiar with Python, looked into standard Python libraries, basic function calls and conventions, and other basic Python elements.
2.
    a.
        i. The board is represented as a two dimensional array of integers. A square on the board can be accessed with a row and column index, such as (4, 5) for a 9-by-9 board.
        ii. Variables are the integers 0-9, but are not explicitly stored anywhere. The integers 1 through BoardSize are iterated when trying to find a valid value, and 0 is used as the "empty square" value.
        iii. The constraints are the numbers in the current row, column and sub-square. These can be accessed via helper methods that iterate over length BoardSize, looking for matches to a passed-in number.
        iv. The constraint graph is represented through the combination of the board and the constraint-checking functions. In this way, filled-in squares and empty-squares are "nodes", and "edges" are loosely defined as a false return value from the constraint checking functions.
        v. The board, its current values, and the recursive stack function represent the state. The recursive stack function keeps track of each square alteration, and will fail back to the last valid point by a cascade of "false" return values, overwriting invalid values with 0.
    b. Forward checking is implemented by storing a list of the empty squares on the board, and a dictionary mapping the possible valid values that each of these squares can hold to the square. When an empty square is updated, it is assigned the first valid value available. Then, each of the neighbors of this square that are also empty are retrieved. The neighbors in this case are defined as the empty squares in the same sub-square, column, or row as the last assigned square. For each of these neighbors, the value assigned to the current square is removed from their valid set. If any of these empty neighbors have no more valid values, the solving function fails, and resets the value of the last assigned square to zero. The solve method then tries to solve the board with the next available value for last assigned square. If there are no valid values for this square, the function fails back recursively to an earlier square. This process is repeated until the last square in the board is assigned, at which point the puzzle is solved.
    c.

Problem	Backtracking	Forward Checking
4x4	125	97
9x9	208149	67339
16x16	Does not finish	Does not finish
25x25	Does not finish	Does not finish

d. Both backtracking and forward checking, in general, take many more steps to solve a 25x25 puzzle than a 16x16, and many more steps to solve a 16x16 than a 9x9, etc. In addition, forward checking performs significantly less checks for each puzzle than backtracking (even with the overhead for the dictionary setup). However, both methods don't necessarily take longer to solve a 16x16 puzzle than a 9x9. These results are expected. Forward checking is much more efficient than backtracking, which explains the discrepancy in checks, and certain larger puzzles might have more ordered results ("easier") than others, reducing the number of failures, and thus the number of recursive calls and number of consistency checks.