Francesca Cambareri & Shyma Alshari

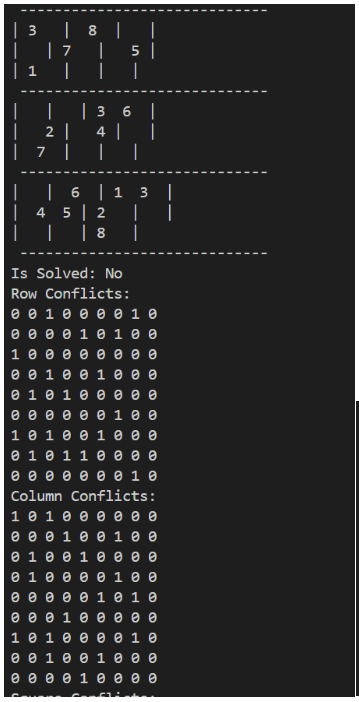
Professor Lili Su

EECE 2560

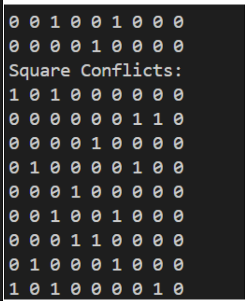
March 30th, 2024

Project 4 – Sudoku Puzzle Report

This purpose of this project was to use the basic code given and implement a recursion algorithm in order for the program to take in unsolved sudoku puzzles and be able to output the puzzle solved on a board, as well as show the boards conflicts, as well as the number of recursive calls to solve the board. The method to solve the board is within the 'solve' function and uses recursive attempts to fill each cell of the Sudoku board and backtracks when a conflict arises. Using this backtracking algorithm, the program is able to successfully find what number goes into each spot, as it systematically fills the cells of the sudoku board with possible values, and then checks those values for conflicts within the row, column and square every step of the way. When the program does discover a conflict, it backtracks to the previous decision point and tries to find an alternative value that will work better and goes through the same conflict checking process. Our program also includes functions to the board class that allow it to update the conflict vectors, print the board as well as the conflict vectors to the screen, add values to the cell for testing, and then go through and update the conflict vectors once again based off of this addition, as well as clear a cell if a value is found to have a conflict and update the conflict vectors based on that, and finally checks to see if the board has been solved or not, and lets the user know. Below we have included screenshots of some of the results from running the code with the provided 'sudoku.txt' file to show how all of these parts work together to produce a final product. Also included is the full working code for both part A and part B.

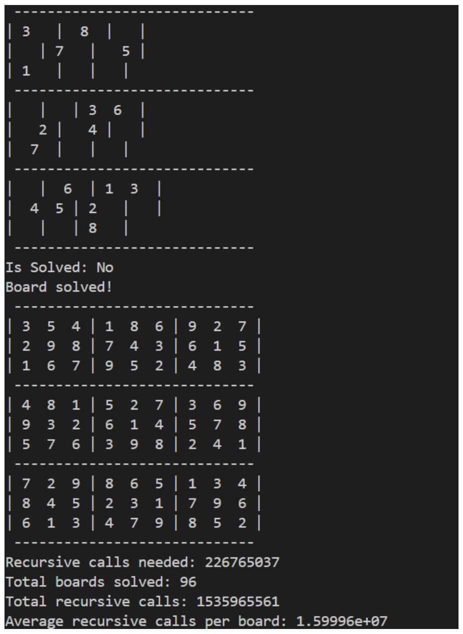


```
 ----------------------------
| 3   |  8 |   |
|   | 7   |   5 |
| 1   |   |   |
 ----------------------------
|   |   | 3  6  |
|   2 |   4 |   |
|  7  |   |   |
 ----------------------------
|   |  6  | 1  3  |
|  4  5 | 2   |   |
|   |   | 8   |
 ----------------------------
Is Solved: No
Row Conflicts:
0 0 1 0 0 0 0 1 0
0 0 0 0 1 0 1 0 0
1 0 0 0 0 0 0 0 0
0 0 1 0 0 1 0 0 0
0 1 0 1 0 0 0 0 0
0 0 0 0 0 0 1 0 0
1 0 1 0 0 1 0 0 0
0 1 0 1 1 0 0 0 0
0 0 0 0 0 0 0 1 0
Column Conflicts:
1 0 1 0 0 0 0 0 0
0 0 0 1 0 0 1 0 0
0 1 0 0 1 0 0 0 0
0 1 0 0 0 0 1 0 0
0 0 0 0 0 1 0 1 0
0 0 0 1 0 0 0 0 0
1 0 1 0 0 0 0 1 0
0 0 1 0 0 1 0 0 0
0 0 0 0 1 0 0 0 0
```



```
0 0 1 0 0 1 0 0 0
0 0 0 0 1 0 0 0 0
Square Conflicts:
1 0 1 0 0 0 0 0 0
0 0 0 0 0 0 1 1 0
0 0 0 0 1 0 0 0 0
0 1 0 0 0 0 1 0 0
0 0 0 1 0 0 0 0 0
0 0 1 0 0 1 0 0 0
0 0 0 1 1 0 0 0 0
0 1 0 0 0 1 0 0 0
1 0 1 0 0 0 0 1 0
```

Figure 1 and 2: Screenshots from part a outputs to show how the conflict vectors are used to help recursively solve the puzzle.



```
 ----------------------------
| 3   |  8 |   |
|   | 7   |   5 |
| 1   |   |   |
 ----------------------------
|   |   | 3  6  |
|   2 |   4 |   |
|  7  |   |   |
 ----------------------------
|   |  6  | 1  3  |
|  4  5 | 2   |   |
|   |   | 8   |
 ----------------------------
Is Solved: No
Board solved!
 ----------------------------
| 3  5  4 | 1  8  6 | 9  2  7 |
| 2  9  8 | 7  4  3 | 6  1  5 |
| 1  6  7 | 9  5  2 | 4  8  3 |
 ----------------------------
| 4  8  1 | 5  2  7 | 3  6  9 |
| 9  3  2 | 6  1  4 | 5  7  8 |
| 5  7  6 | 3  9  8 | 2  4  1 |
 ----------------------------
| 7  2  9 | 8  6  5 | 1  3  4 |
| 8  4  5 | 2  3  1 | 7  9  6 |
| 6  1  3 | 4  7  9 | 8  5  2 |
 ----------------------------
Recursive calls needed: 226765037
Total boards solved: 96
Total recursive calls: 1535965561
Average recursive calls per board: 1.59996e+07
```

Figure 3: Screenshot from Part B results showing the board before and after it is solved, as well as the final printout at the end of the boards that shows the total number of recursive calls needed to solve the 96 boards, as well as the average number of calls needed to solve the board.

Full Working Code for Part A:

```
// include necessary libraries
#include <iostream>
#include <fstream>
#include "d_matrix.h"
#include "d_except.h"
#include <vector>

using namespace std;

// The type of the value in a cell
typedef int ValueType;
// Indicates that a cell is blank
const int Blank = -1;
 // The number of cells in a small square
const int SquareSize = 3;
const int BoardSize = SquareSize * SquareSize;
const int MinValue = 1;
const int MaxValue = 9;
int numSolutions = 0;

class board
{
    // declare the public members of the board class
    public:
        board(int);
        void clear();
        void initialize(ifstream &fin);
        void print();
        bool isBlank(int, int);
        ValueType getCell(int, int);
        void setCell(int, int, ValueType);
        // the clearCell function is a function that clears the cell
        void clearCell(int, int);
        // the isSolved function returns a boolean which is the result of
checking if the board was solved
        bool isSolved();
        // the printConflicts function prints out the conflicts encountered
        void printConflicts() const;

    // declare the private members of the board class
    private:
        matrix<ValueType> value;
        matrix<bool> rows;
        matrix<bool> columns;
        matrix<bool> squares;
};

// constructor for the board class
board::board(int sqSize) : value(BoardSize + 1, BoardSize + 1), rows(BoardSize +
1, MaxValue + 1),
                            columns(BoardSize + 1, MaxValue + 1),
squares(BoardSize + 1, MaxValue + 1) {
    clear();
}

// calculates the square number of the sodoku board by dividing the row and
column indices
int squareNumber(int i, int j)
{
    return SquareSize * ((i - 1) / SquareSize) + ((j - 1) / SquareSize) + 1;
}

void board::clear()
{   // for every cell in the board, do the folloiwing
    for (int i = 1; i <= BoardSize; i++)
    {
        for (int j = 1; j <= BoardSize; j++)
        {
            // set the value of the square at every row and column to blank
            value[i][j] = Blank;
        }
    }
}


void board::initialize(ifstream &fin)
{
    // create a char called ch
    char ch;
    // call the clear function to clear the board before intitializing
    clear();

    // for each cell in the board, do the following
    for (int i = 1; i <= BoardSize; i++)
    {
        for (int j = 1; j <= BoardSize; j++)
        {
            // read a character from the input file
            fin >> ch;

            // if the read char is not Blank, do the following
            if (ch != '.')
            {
                try
                {
                    // convert the charcater to an integer and set the cell value
                    setCell(i, j, ch - '0');
                }
                // if there is an error reading the char or the char is out of
range, do the follwing:
                catch (rangeError &ex)
                {
                    // print error message
                    cout << "Error: " << ex.what() << endl;
                    // create a placeholder string
                    string placeholder;
                    getline(fin, placeholder);
                    // exit the function and go to the next board
                    return;
                }
            }
        }
    }
}

void board::print()
{
    // for each cell in the board, do the following
    for (int i = 1; i <= BoardSize; i++)
    {
        if ((i - 1) % SquareSize == 0)
        {
            // print a dash for each cell
            cout << " -";
            for (int j = 1; j <= BoardSize; j++)
                cout << "---";
            cout << "-";
            cout << endl;
        }
        for (int j = 1; j <= BoardSize; j++)
        {
            // create vertical division line between blocks
            if ((j - 1) % SquareSize == 0)
```

```
                cout << "|";
            // if not blank, print the cell value
            if (!isBlank(i, j))
                cout << " " << getCell(i, j) << " ";
            // if blank, print a dash
            else
                cout << " ";
        }
        // clsoe out the block using a vertical line
        cout << "|";
        cout << endl;
    }
    cout << " -";
    // create the boarder for the bottom of the board
    for (int j = 1; j <= BoardSize; j++)
        cout << "---";
    cout << "-";
    cout << endl;
}

ValueType board::getCell(int i, int j)
{
    // if the index is within a valid range of the board, do the folliwing
    if (i >= 1 && i <= BoardSize && j >= 1 && j <= BoardSize)
        // return the value of that cell at the (i,j)
        return value[i][j];
    // if the index is not within a valid range of the board, do the folliwing
    else
        // throw a rangeError
        throw rangeError("bad value in getCell");
}

void board::setCell(int i, int j, ValueType newValue)
{
    // if the index is within a valid range of the board, do the folliwing
    if (i >= 1 && i <= BoardSize && j >= 1 && j <= BoardSize)
    {
        // set the value of the cell (i,j) to newValue
        value[i][j] = newValue;
        rows[i][newValue] = true;
        columns[j][newValue] = true;
        squares[squareNumber(i, j)][newValue] = true;
    }
    // if the index is not within a valid range of the board, do the folliwing
    else
    {
        // throw a rangeError
        throw rangeError("bad value in setCell");
    }
}

void board::clearCell(int i, int j)
{
    // if the index is within a valid range of the board, do the folliwing
    if (i < 1 || i > BoardSize || j < 1 || j > BoardSize)
        // throw a rangeError
        throw rangeError("bad value in clearCell");
    // if the cell is not already blank
    if (!isBlank(i, j))
    {
        // get the value of the cell at (i,j)
        ValueType val = getCell(i, j);
        // set the cell (i,j) to blank
        value[i][j] = Blank;
        // the corresponding row is marked as not having the previous value
        rows[i][val] = false;
        // the corresponding column is marked as not having the previous value
        columns[j][val] = false;
        // the corresponding square is marked as not having the previous value
        squares[squareNumber(i, j)][val] = false;
    }
}

bool board::isBlank(int i, int j)
{
    // if the index is within a valid range of the board, do the folliwing
    if (i < 1 || i > BoardSize || j < 1 || j > BoardSize)
        // throw a rangeError
        throw rangeError("bad value in isBlank");
    // check if the value of the cell at (i,j) is blank and return
    return (value[i][j] == Blank);
}

bool board::isSolved()
{
    // for each cell of the board, do the following
    for (int i = 1; i <= BoardSize; ++i)
    {
        // if a cell is blank, the board is not solved
        for (int j = 1; j <= BoardSize; ++j) {
            if (isBlank(i, j))
            {
                // the board is not solved, return false
                return false;
            }
        }
    }
    // if every cell is filled (not blank), the board is solved, return true
    return true;
}


void board::printConflicts() const {
    cout << "Row Conflicts:" << endl;
    for (int i = 1; i <= BoardSize; ++i) {
        for (int val = 1; val <= MaxValue; ++val) {
            int count = 0;
            for (int j = 1; j <= BoardSize; ++j) {
                if (value[i][j] == val) {
                    ++count;
                }
            }
            cout << count << " ";
        }
        cout << endl;
    }

    cout << "Column Conflicts:" << endl;
    for (int j = 1; j <= BoardSize; ++j) {
        for (int val = 1; val <= MaxValue; ++val) {
            int count = 0;
            for (int i = 1; i <= BoardSize; ++i) {
                if (value[i][j] == val) {
                    ++count;
                }
            }
            cout << count << " ";
        }
        cout << endl;
    }

    cout << "Square Conflicts:" << endl;
    for (int i = 1; i <= BoardSize; i += SquareSize) {
        for (int j = 1; j <= BoardSize; j += SquareSize) {
            for (int val = 1; val <= MaxValue; ++val) {
                int count = 0;
                for (int r = i; r < i + SquareSize; ++r) {
                    for (int c = j; c < j + SquareSize; ++c) {
                        if (value[r][c] == val) {
                            ++count;
                        }
                    }
                }
                cout << count << " ";
            }
            cout << endl;
        }
    }
}


int main() {
    ifstream fin;
    string fileName = "sudoku.txt";
    fin.open(fileName.c_str());
    if (!fin) {
        cerr << "Cannot open " << fileName << endl;
        exit(1);
    }
    try {
        board b1(SquareSize);
        while (fin && fin.peek() != 'Z') {
            b1.initialize(fin);
            b1.print();
            cout << "Is Solved: " << (b1.isSolved() ? "Yes" : "No") << endl;
            b1.printConflicts();
        }
    } catch (indexRangeError &ex) {
        cout << ex.what() << endl;
        exit(1);
    }
    return 0;
}
```

Full Working Code for Part B:

```
// include necessary libraries and files
#include <iostream>
#include <fstream>
#include "d_matrix.h"
#include "d_except.h"
#include <vector>

// use standard nasespace std
using namespace std;

// declare global variables
// ValueType is the type of value in a sodoku cell
typedef int ValueType;
// a Blank is set to -1 to indicate that a cell has nothing in it (is blank)
const int Blank = -1;
// SquareSize is set to 3 which is the number of cells in a small square
const int SquareSize = 3;
// BOardSize is SquareSize multiplied by itself which in this case, is 9
const int BoardSize = SquareSize * SquareSize;
// the MinValue is initialized to 1
const int MinValue = 1;
// the MaxValue is initialized to 9
const int MaxValue = 9;
// the numSolutions is initialized to 0
int numSolutions = 0;

class board
{
    // declare the public members of the board class
    public:
        board(int);
        // the clear function returns sets all the cells of the board to be blank
        void clear();
        void initialize(ifstream &fin);
        void print();
        bool isBlank(int, int);
        ValueType getCell(int, int);
        void setCell(int, int, ValueType);
        // the clearCell function is a function that clears the cell
        void clearCell(int, int);
        // the isSolved function returns a boolean which is the result of
checking if the board was solved
        bool isSolved();
        // the printConflicts function prints out the conflicts encountered
        void printConflicts() const;
        // the solve function uses recursion to solve the sodoku boards
        bool solve(int, int);
        // the getRecursiveCalls fucntion returns the number of recursive calls
used to solve the sodoku board
        int getRecursiveCalls();

    // declare the private members of the board class
    private:
        matrix<ValueType> value;
        matrix<bool> rows;
        matrix<bool> columns;
        matrix<bool> squares;
        // recurciveCalls keeps track of the number of recuruve calls used to
solve the sodoku board
        int recursiveCalls;
};

// constructor for the board class
board::board(int sqSize) : value(BoardSize + 1, BoardSize + 1), rows(BoardSize +
1, MaxValue + 1),
                            columns(BoardSize + 1, MaxValue + 1),
squares(BoardSize + 1, MaxValue + 1) {
    clear();
    // initialize the number of recursive calls to 0
    recursiveCalls = 0;
}

// calculates the square number of the sodoku board by dividing the row and
column indices
int squareNumber(int i, int j)
{
    return SquareSize * ((i - 1) / SquareSize) + ((j - 1) / SquareSize) + 1;
}

// the getRecursiveCalls fucntion returns the number of recursive calls used to
solve the sodoku board
int board::getRecursiveCalls()
{
    // return the variable called recursiveCalls
    return recursiveCalls;
}

// the clear function returns sets all the cells of the board to be blank
```

```
void board::clear()
{
    // for every cell in the board, do the folloiwing
    for (int i = 1; i <= BoardSize; i++)
    {
        for (int j = 1; j <= BoardSize; j++)
        {
            // set the value of the square at every row and column to blank
            value[i][j] = Blank;
        }
    }

    // reset the row matrixs
    rows = matrix<bool>(BoardSize + 1, MaxValue + 1);
    // reset the columns matrixs
    columns = matrix<bool>(BoardSize + 1, MaxValue + 1);
    // reset the squares matrixs
    squares = matrix<bool>(BoardSize + 1, MaxValue + 1);
}

void board::initialize(ifstream &fin)
{
    // create a char called ch
    char ch;
    // call the clear function to clear the board before intitializing
    clear();

    // for each cell in the board, do the following
    for (int i = 1; i <= BoardSize; i++)
    {
        for (int j = 1; j <= BoardSize; j++)
        {
            // read a character from the input file
            fin >> ch;

            // if the read char is not Blank, do the following
            if (ch != '.')
            {
                try
                {
                    // convert the charcater to an integer and set the cell value
                    setCell(i, j, ch - '0');
                }
                // if there is an error reading the char or the char is out of
range, do the follwing:
```

```
                catch (rangeError &ex)
                {
                    // print error message
                    cout << "Error: " << ex.what() << endl;
                    // create a placeholder string
                    string placeholder;
                    getline(fin, placeholder);
                    // exit the function and go to the next board
                    return;
                }
            }
        }
    }
}


void board::print()
{
    // for each cell in the board, do the following
    for (int i = 1; i <= BoardSize; i++)
    {
        if ((i - 1) % SquareSize == 0)
        {
            // print a dash for each cell
            cout << " -";
            for (int j = 1; j <= BoardSize; j++)
                cout << "---";
            cout << "-";
            cout << endl;
        }
        for (int j = 1; j <= BoardSize; j++)
        {
            // create vertical division line between blocks
            if ((j - 1) % SquareSize == 0)
                cout << "|";
            // if not blank, print the cell value
            if (!isBlank(i, j))
                cout << " " << getCell(i, j) << " ";
            // if blank, print a dash
            else
                cout << "   ";
        }
        // clsoe out the block using a vertical line
        cout << "|";
        cout << endl;
    }
    cout << " -";
    // create the boarder for the bottom of the board
    for (int j = 1; j <= BoardSize; j++)
        cout << "---";
    cout << "-";
    cout << endl;
}

ValueType board::getCell(int i, int j)
{
    // if the index is within a valid range of the board, do the folliwing
    if (i >= 1 && i <= BoardSize && j >= 1 && j <= BoardSize)
        // return the value of that cell at the (i,j)
        return value[i][j];
    // if the index is not within a valid range of the board, do the folliwing
    else
        // throw a rangeError
        throw rangeError("bad value in getCell");
}

void board::setCell(int i, int j, ValueType newValue)
{
    // if the index is within a valid range of the board, do the folliwing
    if (i >= 1 && i <= BoardSize && j >= 1 && j <= BoardSize)
    {
        // set the value of the cell (i,j) to newValue
        value[i][j] = newValue;
        rows[i][newValue] = true;
        columns[j][newValue] = true;
        squares[squareNumber(i, j)][newValue] = true;
    }
    // if the index is not within a valid range of the board, do the folliwing
    else
    {
        // throw a rangeError
        throw rangeError("bad value in setCell");
    }
}

void board::clearCell(int i, int j)
{
    // if the index is within a valid range of the board, do the folliwing
    if (i < 1 || i > BoardSize || j < 1 || j > BoardSize)
        // throw a rangeError
```

```cpp
            throw rangeError("bad value in clearCell");
        // if the cell is not already blank
        if (!isBlank(i, j))
        {
            // get the value of the cell at (i,j)
            ValueType val = getCell(i, j);
            // set the cell (i,j) to blank
            value[i][j] = Blank;
            // the corresponding row is marked as not having the previous value
            rows[i][val] = false;
            // the corresponding column is marked as not having the previous value
            columns[j][val] = false;
            // the corresponding square is marked as not having the previous value
            squares[squareNumber(i, j)][val] = false;
        }
    }

    bool board::isBlank(int i, int j)
    {
        // if the index is within a valid range of the board, do the folliwing
        if (i < 1 || i > BoardSize || j < 1 || j > BoardSize)
            // throw a rangeError
            throw rangeError("bad value in isBlank");
        // check if the value of the cell at (i,j) is blank and return
        return (value[i][j] == Blank);
    }

    bool board::isSolved()
    {
        // for each cell of the board, do the following
        for (int i = 1; i <= BoardSize; ++i)
        {
            // if a cell is blank, the board is not solved
            for (int j = 1; j <= BoardSize; ++j) {
                if (isBlank(i, j))
                {
                    // the board is not solved, return false
                    return false;
                }
            }
        }
        // if every cell is filled (not blank), the board is solved, return true
        return true;
    }
```

```
void board::printConflicts() const {
    cout << "Row Conflicts:" << endl;
    //iterates through each row in the board
    for (int i = 1; i <= BoardSize; ++i) {
        //iterates through the possible values (1-9)
        for (int val = 1; val <= MaxValue; ++val) {
            // initialized counter that is used to keep track of occurences of a
value in the row
            int count = 0;
            //iterates through the columns in the row
            for (int j = 1; j <= BoardSize; ++j) {
                //checks if the value in that cell is found and adds 1 to count
if so
                if (value[i][j] == val) {
                    ++count;
                }
            }
            cout << count << " ";
        }
        cout << endl;
    }

    cout << "Column Conflicts:" << endl;
    //iterates through each row in the board
    for (int j = 1; j <= BoardSize; ++j) {
        //iterates through the possible values (1-9)
        for (int val = 1; val <= MaxValue; ++val) {
            // initialized counter that is used to keep track of occurences of a
value in the row
            int count = 0;
            //iterates through the columns in the row
            for (int i = 1; i <= BoardSize; ++i) {
                //checks if the value in that cell is found and adds 1 to count
if so
                if (value[i][j] == val) {
                    ++count;
                }
            }
            cout << count << " ";
        }
        cout << endl;
    }

    cout << "Square Conflicts:" << endl;
    //iterates through each square row in the board
    for (int i = 1; i <= BoardSize; i += SquareSize) {
        //iterates through each square column
        for (int j = 1; j <= BoardSize; j += SquareSize) {
            //iterates through the possible values(1-9)
            for (int val = 1; val <= MaxValue; ++val) {
                // initialized counter that is used to keep track of occurences
of a value in the row
                int count = 0;
                //iterates through all the rows in the square
                for (int r = i; r < i + SquareSize; ++r) {
                    //iterates through all the columns in the square
                    for (int c = j; c < j + SquareSize; ++c) {
                        //checks if the value in that cell is found and adds 1 to
count if so
                        if (value[r][c] == val) {
                            ++count;
                        }
                    }
                }
                cout << count << " ";
            }
            cout << endl;
        }
    }
}

bool board::solve(int i, int j) {
    //increments the recursive calls count by one each time it is run
    ++recursiveCalls;
//checks if all rows have been traversed when iterating through
    if (i == BoardSize + 1) {
        //resets the row index to 1
        i = 1;
        //checks if all of the columns have been traversed when iterating through
        if (++j == BoardSize + 1)
            return true; // Entire board has been successfully filled without
conflict
    }
    //checks if the cell is not blank, if not blank moves to the next cell
    if (!isBlank(i, j))
        return solve(i + 1, j);
//iterates through possible values that could be placed in a blank cell
    for (ValueType val = MinValue; val <= MaxValue; ++val) {
        //checks if the value is not present in the row, column, or square that
it is solving
```

```
        if (!rows[i][val] && !columns[j][val] && !squares[squareNumber(i,
j)][val]) {
            //if the value is not present, sets the value in the cell
            setCell(i, j, val);
            //moves on to the next cell
            if (solve(i + 1, j))
                return true;
            clearCell(i, j); // Undo assignment if no solution found
        }
    }

    return false; // No value in [MinValue, MaxValue] worked, backtrack
}

int main() {
    ifstream fin;
    string fileName = "sudoku.txt";
    fin.open(fileName.c_str());
    if (!fin) {
        cerr << "Cannot open " << fileName << endl;
        exit(1);
    }
    try {
        board b1(SquareSize);
        int totalRecursiveCalls = 0;
        int boardsSolved = 0;
        while (fin && fin.peek() != 'Z') {
            b1.initialize(fin);
            b1.print();
            cout << "Is Solved: " << (b1.isSolved() ? "Yes" : "No") << endl;
            b1.solve(1, 1);
            //checks if board is solved and prints a statement saying it is if is
true
            if (b1.isSolved()) {
                cout << "Board solved!" << endl;
                b1.print();
                //prints the number of recursive calls needed for that particular
board
                cout << "Recursive calls needed: " << b1.getRecursiveCalls() <<
endl;
                //adds the number of recursive calls needed for current board to
total recursive calls count
                totalRecursiveCalls += b1.getRecursiveCalls();
                //increments boards solved count by 1
                ++boardsSolved;
            } else {
                //if not solved prints unable to solve
                cout << "Unable to solve board!" << endl;
            }
        }
        //prints the total number of boards solved
        cout << "Total boards solved: " << boardsSolved << endl;
        //prints the total number of recursive calls for the all the boards
        cout << "Total recursive calls: " << totalRecursiveCalls << endl;
        //if any boards were solved prints the average recursive calls needed
        if (boardsSolved > 0)
            cout << "Average recursive calls per board: " <<
static_cast<double>(totalRecursiveCalls) / boardsSolved << endl;
    } catch (indexRangeError &ex) {
        cout << ex.what() << endl;
        exit(1);
    }
    return 0;
}
```