# CS 1027A - Assignment 1 - Sudoku

● Graded

**Student**

Mohammed Ali Abdul-nabi

**Total Points**

20 / 20 pts

**Autograder Score**

15.0 / 15.0

**Passed Tests**

[--------------- TEST 01 (Sudoku) --------------] (1/1)
[--------------- TEST 02 (Sudoku) --------------] (1/1)
[--------------- TEST 03 (Sudoku) --------------] (1/1)
[--------------- TEST 04 (Sudoku) --------------] (1/1)
[--------------- TEST 05 (Sudoku) --------------] (1/1)
[--------------- TEST 06 (Sudoku) --------------] (1/1)
[--------------- TEST 07 (Sudoku) --------------] (1/1)
[--------------- TEST 08 (Sudoku) --------------] (1/1)
[--------------- TEST 09 (Sudoku) --------------] (1/1)
[--------------- TEST 10 (Sudoku) --------------] (1/1)

**Question 2**

## Code Logic

**1** / 1 pt

✓ **– 0 pts** Code logic is completely or mostly correct

**– 1 pt** Several errors in the code logic

**– 0.5 pts** Code logic is partially correct

[redacted]

**Question 3**

## Code Formatting/Readability

**2** / 2 pts

✓ **– 0 pts** Code is clean, indented properly, and variables have descriptive names

**– 1 pt** Some parts of code are not formatted well or the variables don't have descriptive names

**– 2 pts** No proper code formatting. Code not readable

Question 4

**Comments** 2 / 2 pts

✓ **– 0 pts** Comments throughout the code are proper and relevant

**– 1 pt** Some comments but not a sufficient amount or they're not completely relevant

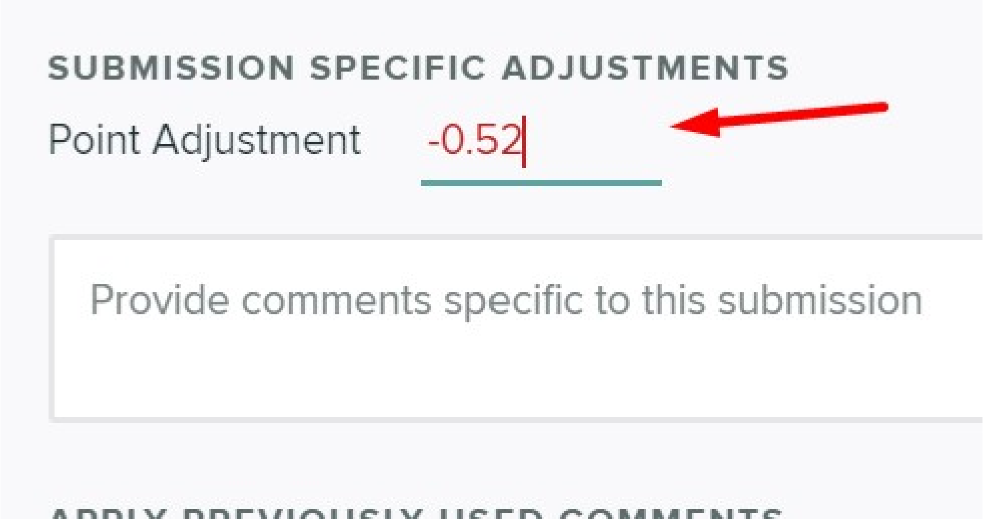
**– 2 pts** No or very few comments

Question 5

Penalties **0** / 0 pts

5.1 ***Late Submissions* -2/day** **0** / 0 pts

✓ **– 0 pts** **@TAs: DO NOT ADD YOUR OWN RUBRICS HERE** *Please enter the deduction in the* **Point Adjustment** *field below if late penalty applies.*



SUBMISSION SPECIFIC ADJUSTMENTS

Point Adjustment -0.52

Provide comments specific to this submission

APPLY PREVIOUSLY USED COMMENTS

5.2 **Incorrect submission (doesn't compile, package line, .class file, etc.) -5** **0** / 0 pts

✓ **– 0 pts** Code compiled and ran

**– 5 pts** Code did not compile

5.3 **Incorrect instance variables or methods -2** **0** / 0 pts

✓ **– 0 pts** No additional methods or instance variables and all modifiers are correct

**– 2 pts** Additional methods or instance variables OR incorrect modifiers (i.e. public instead of private)

## Autograder Results

[--------------- TEST 01 (Sudoku) --------------] (1/1)

[--------------- TEST 02 (Sudoku) --------------] (1/1)

**[--------------- TEST 03 (Sudoku) --------------] (1/1)**

**[--------------- TEST 04 (Sudoku) --------------] (1/1)**

**[--------------- TEST 05 (Sudoku) --------------] (1/1)**

**[--------------- TEST 06 (Sudoku) --------------] (1/1)**

**[--------------- TEST 07 (Sudoku) --------------] (1/1)**

**[--------------- TEST 08 (Sudoku) --------------] (1/1)**

**[--------------- TEST 09 (Sudoku) --------------] (1/1)**

**[--------------- TEST 10 (Sudoku) --------------] (1/1)**

### Submitted Files

▾ Sudoku.java Download

```java
public class Sudoku {
    private int size;
    private int[][] grid;

    public Sudoku(int[][] numbers) {
        this.grid = numbers;                              // Initalized the grid with numbers.
        size = grid.length;                               // Find the size of the grid using the length.
    }

    public int getSize() {                                // Returns the size variable (getter).
        return size;
    }

    public int[][] getGrid() {                            // Returns the grid variable (getter).
        return grid;
    }

    public int getDigitAt(int row, int col) {
        if (row < 0 || col < 0 && row >= size || col >= size) {  //Checks if either the row or col are out of range.
            return -1;
        }
        return grid[row][col];                            // Returns the digit if the row and col are in range.
    }

    public boolean isValidRow(int row) {
        boolean[] dupStorRow = new boolean[size];                  // Makes array to keep track of the int when going through the puzzle
        for (int i = 0; i < size; i++) {                 // For loop going through each slot in the row.
            int digitAt = grid[row][i];                   // Gets the digit at current location and places it in digitAt.
            if ((digitAt < 1) || (digitAt > size) || dupStorRow[digitAt - 1]) {
                return false;
            }
            dupStorRow[digitAt - 1] = true;                    //Marks as seen.
        }
        return true;
    }

    public boolean isValidCol(int col) {
        boolean[] dupStorCol = new boolean[size];                  // Makes array to keep track of the int when going through the puzzle
        for (int i = 0; i < size; i++) {                 // For loop going through each slot in the column.
            int digitAt = grid[i][col];                   // Gets the digit at current location and places it in digitAt.
            if ((digitAt < 1) || (digitAt > size) || dupStorCol[digitAt - 1]) {
                return false;
            }
            dupStorCol[digitAt - 1] = true;                     //Marks as seen.
        }
        return true;
    }

    public boolean isValidBox(int row, int col) {
        if (row < 0 || col < 0 || row >= size - 2 || col >= size - 2) {    // Checks if the row and col are within range on both sides
            return false;
        }
        boolean[] dupStorGrd = new boolean[size];
        for (int i = row; i <= row + 2; i++) {
            for (int j = col; j <= col + 2; j++) {
                int digitAt = grid[i][j];
                if (digitAt < 1 || digitAt > size || dupStorGrd[digitAt-1]) {  //Checks if digit is within range or seen
                    return false;
                }
                dupStorGrd[digitAt - 1] = true;              //Marks as seen.
            }
        }
        return true;
    }

    public boolean isValidSolution() {                       //Validates if the whole Sudoku is correct.
        for (int i = 0; i < size; i++) {                     //Checking for all values of i less than the total size.
            if (!(isValidCol(i) && isValidRow(i))) {         //Checks for any instances where they are not true
                return false;
            }
        }
        if (size == 9) {                                     //Checks 3x3 if the size is 9.
            for (int i = 0; i < size; i += 3) {
                for (int j = 0; j < size; j += 3) {
                    if (!(isValidBox(i, j))) {
                        return false;
                    }
                }
            }
        }
        return true;                                         //The Sudoku is completely Valid.
    }

    public boolean equals(Sudoku other) {
        if (this.size != other.size) {                       //Checks if the sizes are not equal.
            return false;
        }
        int[][] otherGrid = other.getGrid();
        for (int i = 0; i < size; i++) {                    //Compares all the gird slots in both grids.
            for (int j = 0; j < size; j++) {
                if (this.grid[i][j] != otherGrid[i][j]) {
                    return false;
                }
            }
        }
        return true;                                        //The two Sudokus are the same.
    }

    public String toString() {
        StringBuilder sb = new StringBuilder();

        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                sb.append(grid[i][j]);                      //Add i and j to string
                sb.append(' ');                             //Add space after each digit
            }
            sb.append('\n');                                //Add new Line.
        }
        return sb.toString();
    }
}
```

UniqueDiagonalSudoku.java

Download

```java
public class UniqueDiagonalSudoku extends Sudoku {

    public UniqueDiagonalSudoku(int[][] numbers) {
        super(numbers);
    }
    @Override
    public boolean isValidSolution() {
        if (!super.isValidSolution()) {                        // Check if base rules of Sudoku are met.
            return false;
        }
        boolean[] diaStorTB = new boolean[getSize()];           //Store the digits we have seen
        boolean[] diaStorBT = new boolean[getSize()];
        boolean isTBValid = true;                                //Keeps track of if the diagonals are valid or not
        boolean isBTValid = true;
        boolean areDiagValid = true;                             //Stores the end result of the diagonals


        for (int i = 0; i < getSize(); i++) {                  // Check Diagonal from top-left to bottom-right
             int digitAt = getGrid()[i][i];
             if ((digitAt < 1) || (digitAt > getSize()) || diaStorTB[digitAt - 1]) {
               isTBValid = false;
               break;
                }
             diaStorTB[digitAt - 1] = true;                      //Mark the digit as seen.
            }

        for (int i = 0; i < getSize(); i++) {                  // Check Diagonal from bottom-left to top-right
             int digitAt = getGrid()[getSize() - 1 - i][i];
             if ((digitAt < 1) || (digitAt > getSize()) || diaStorBT[digitAt - 1]) {
                 isBTValid = false;
                 break;
                }
             diaStorBT[digitAt - 1] = true;                      //Mark the digit as seen.
            }

        if (isTBValid == false && isBTValid == false) {          //Checks if both diagonals are invalid.
            areDiagValid = false;
        }

        return areDiagValid;
    }
}

```