



Algorithms and Analysis COSC 1285 / COSC 2123
Assignment 2: Solving Sudoku

	Assessment Type	Individual assignment. Submit online via Canvas → Assignments → Assignment Task 3: Assignment 2 → Assignment 2: Solving Sudoku . Marks awarded for meeting requirements as closely as possible. Clarifications/updates may be made via announcements/relevant discussion forums.
	Due Date	Week 13, Friday 5th June 2020, 11:59pm
	Marks	50

1 Objectives

There are three key objectives for this assignment:

- Apply transform-and-conquer strategies to to solve a real application.
- Study Sudoku and develop algorithms and data structures to solve Sudoku puzzles.
- Research and extend algorithmic solutions to Sudoku variants.

2 Learning Outcomes

This assignment assesses learning outcomes CLO 1, 2, 3 and 5 of the course. Please refer to the course guide for the relevant learning outcomes: <http://www1.rmit.edu.au/courses/004302>

3 Introduction and Background

Sudoku was a game first popularised in Japan in the 80s but dates back to the 18th century and the “Latin Square” game. The aim of Sudoku is to place numbers from 1 to 9 in cells of a 9 by 9 grid, such that in each row, column and 3 by 3 block/box all 9 digits are present. Typical Sudoku puzzle will have some of the cells initially filled in with digits and a well designed game will have one unique solution. In this assignment you will implement algorithms that can solve puzzles of Sudoku and its variants.

Sudoku

Sudoku puzzles are typically played on a 9 by 9 grid, where each cell can be filled in with discrete values from 1-9. Sudoku puzzles have some of these cells pre-filled with values, and the aim is to fill in all the remaining cells with values that would form a valid solution. A valid solution (for a 9 by 9 grid with values 1-9) needs to satisfy a number of constraints:

1. Every cell is assigned a value between 1 to 9.
2. Every row contains 9 unique values from 1 to 9.
3. Every column contains 9 unique values from 1 to 9.
4. Every 3 by 3 block (called a box) contains 9 unique values from 1 to 9.

As an example, consider Figure 1. Figure 1a shows the initial Sudoku grid, with some values pre-filled in. After filling in all the remaining cells with values that satisfy the constraints, we obtain the solution illustrated in Figure 1b. As an exercise, check that every row, column and 3 by 3 block/box (delimited by bold black lines) satisfy the respective constraints.

5	3			7					5	3	4	6	7	8	9	1	2
6			1	9	5				6	7	2	1	9	5	3	4	8
	9	8						6				1	9	8	3	4	2
8				6					3	8	5	9	7	6	1	4	2
4			8		3				1	4	2	6	8	5	3	7	9
7				2					6	7	1	3	9	2	4	8	5
	6						2	8		9	6	1	5	3	7	2	8
			4	1	9				5	2	8	7	4	1	9	6	3
				8			7	9		3	4	5	2	8	6	1	7

(a) Puzzle.

(b) Solved.

Figure 1: Example of a Sudoku puzzle from Wikipedia.

For further details about Sudoku, please visit <https://en.wikipedia.org/wiki/Sudoku>.

Killer Sudoku

Killer Sudoku puzzles are typically played on 9 by 9 grids also and have many elements of Sudoku puzzles, including all of its constraints. It additionally has cages, which are subset of cells that have a total assigned to them. A valid Killer Sudoku must also satisfy the constraint that the values assigned to a cage are unique and add up to the total.

Formally, a valid solution for a Killer Sudoku of 9 by 9 grid and 1-9 as values needs to satisfy all of the following constraints (the first 4 are the same as standard Sudoku):

1. Every cell is assigned a value between 1 to 9.
2. Every row contains 9 unique values from 1 to 9.
3. Every column contains 9 unique values from 1 to 9.
4. Every 3 by 3 block/box contains 9 unique values from 1 to 9.
5. The sum of values in the cells of each cage must be equal to the cage target total and all the values in a cage must be unique.

As an example, consider Figure 2. Figure 2a shows the initial puzzle. Note the cages are in different colours, and in the corner of each cage is the target total. Figure 2b is the solution. Note all rows, columns and 3 by 3 blocks/boxes satisfy the Sudoku constraints, as well as the values in each cage add up to the target totals.

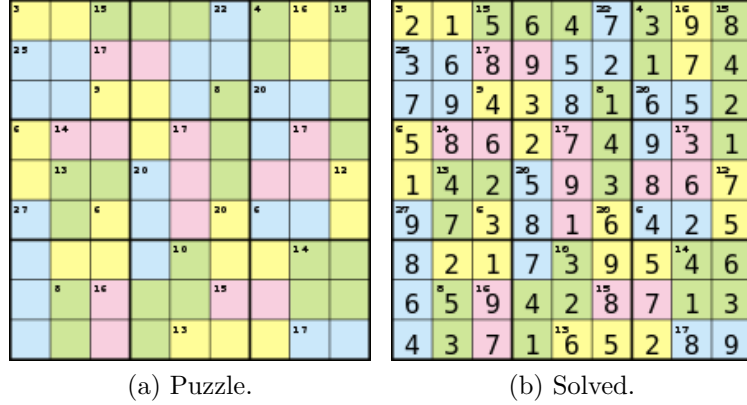


Figure 2: Example of a Killer Sudoku puzzle. Example comes from Wikipedia.

Sudoku Solvers

In this assignment, we will implement two families of algorithms to solve Sudoku, namely backtracking and exact cover approaches. We describe these algorithms here.

Backtracking

The backtracking algorithm is an improvement on blind brute force generation of solutions. It essentially makes a preliminary guess of the value of an empty cell, then try to assign values to other unassigned cell. If at any stage we find an empty cell where it is not possible to assign any values without breaking one or more of the constraints, we backtrack to the previous cell and try another value. This is similar to a DFS when it hits a deadend, if a certain branch of search tree results in an invalid (partial) Sudoku grid, then we backtrack and another value is tried.

Exact Cover

To describe this, we first explain what is the exact cover problem.

Given a universe of items (values) and a set of item subsets, an exact cover is to select some of the subsets such that the union of these subsets equals the universal of items (or they cover all the items) and the subsets cannot have any overlapping items.

For example, if we had a universe of items $\{i_1, i_2, i_3, i_4, i_5, i_6\}$, and the following subsets of items: $\{i_1, i_3\}$, $\{i_2, i_3, i_4\}$, $\{i_1, i_5, i_6\}$, $\{i_2, i_5\}$ and $\{i_6\}$, a possible set cover is to select $\{i_2, i_3, i_4\}$ and $\{i_1, i_5, i_6\}$, whose union includes all 6 possible items and they contain no overlapping items.

The exact cover can be represented as a binary matrix, where we have columns (representing the items) and rows, representing the subsets.

For example, using the example above, we can represent the exact cover problem as follows:

	i_1	i_2	i_3	i_4	i_5	i_6
$\{i_1, i_3\}$	1	0	1	0	0	0
$\{i_2, i_3, i_4\}$	0	1	1	1	0	0
$\{i_1, i_5, i_6\}$	1	0	0	0	1	1
$\{i_2, i_5\}$	0	1	0	0	1	0
$\{i_6\}$	0	0	0	0	0	1

Using the above matrix representation, an exact cover is a selected subset of rows, such that if we constructed a sub-matrix by taking all the selected rows and columns, each column must contain a 1 in exactly one selected row.

For example, if we selected $\{i_2, i_3, i_4\}$ and $\{i_1, i_5, i_6\}$, we have the resulting submatrix:

	i_1	i_2	i_3	i_4	i_5	i_6
$\{i_2, i_3, i_4\}$	0	1	1	1	0	0
$\{i_1, i_5, i_6\}$	1	0	0	0	1	1

Note each column in this sub-matrix have a single 1, which corresponds to the requirements of every item been covered and the subsets do not have overlapping items.

How does this relate to solving Sudoku puzzles? An example of transform and conquer, a Sudoku puzzle can be transformed into an exact cover problem and we can use two exact cover algorithms to generally solve Sudoku faster than the basic backtracking approach. We first describe the two algorithms to find exact cover, then explain how the transformation works.

Algorithm X Algorithm X is Donald Knuth’s basic solution to the exact cover problem. He devised Algorithm X to motivate the Dancing Links approach (we will discuss this next). Algorithm X works on the binary matrix representation introduced previously. Essentially it is a backtracking algorithm and works on the columns and rows of the binary matrix. Recall that each column represents an item, and each row represents a subset. What we want is to select some rows (subsets) such that across the selected rows, there is exactly a single ‘1’ in each of the columns – this condition means that all items are covered and covered exactly once by the selected rows/subsets. We try different columns and rows, and backtrack if there is an assignment that lead to an invalid (partial) grid. After backtracking, another column/row will be selected.

Keeping this in mind, the algorithm goes through a number of steps, but aims to essentially do what we have described above. See https://en.wikipedia.org/wiki/Knuth%27s_Algorithm_X for further details.

Dancing Links Approach One of the issues with Algorithm X is the need to scan through the (partial) matrices every time it seeks to select a column with smallest number of 1’s, which row intersects with a column, and which column intersects with a row. Also when backtracking it can be costly to reinsert rows and columns.

To address these challenges, Donald Knuth proposed a new approach, Dancing Links, which is both a data structure and set of operations to speed up above.

The binary matrix for any exact cover problem is typically sparse (i.e., most entries are 0). Recall our discussions about using linked list to represent graphs that are sparse, i.e., few edges? We can do the same thing here, but instead use 2D doubly linked lists. To best explain this, lets consider the structure from the exact cover example first:

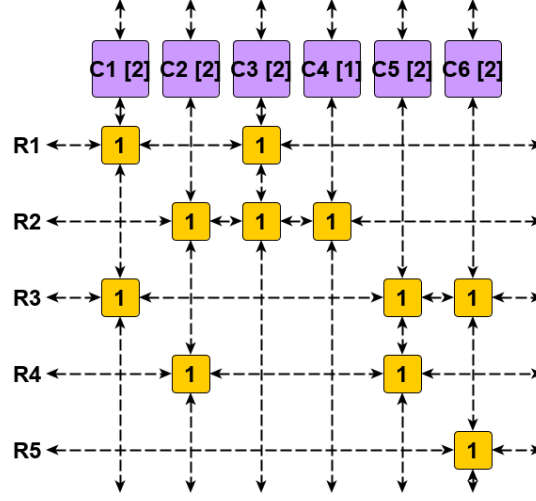


Figure 3: Example of dancing links data structure. Note columns header nodes have number of 1s in its column represented as [Y], where Y is the number of 1s. The structure looks back on itself, in both columns and rows.

As we can see, there is a node for each '1' entry in the binary matrix. Each column is a vertical (doubly) linked list, each row is a horizontal (doubly) linked list, and they wrap around in both directions. In addition, each column has a header node, that also lists the number of '1' entries, so we can quickly find the column with smallest number of '1's.

To solve the exact cover problem, we would use the same approach as Algorithm X, but now we can scan quickly and also backtrack more easily. The data structure only has entries for '1's, so we can quickly scan through the doubly linked data structure to analyse these. In addition, a linked list allows quick (re)insertion and deletion from backtracking, which is one issue with the standard Algorithm X formulation. See <https://arxiv.org/abs/cs/0011047> for further details.

Sudoku Transformation To represent Sudoku as an exact cover problem, we only need to construct a relevant binary matrix representation whose exact cover solution corresponds to a valid Sudoku assignment. At a high level, we want to represent the constraints of Sudoku as the columns, and possible value assignments (the 'subsets') as the rows. Let's discuss the construction of the binary matrix first before explaining why it works.

Rows:

We specify a possible value assignment to each cell as the rows. For a 9 by 9 Sudoku puzzle, there are 9 by 9 cells, of which each can take one of the 9 values, giving us $9 * 9 * 9 = 729$ rows. E.g., $(r = 0, c = 2, v = 3)$ is a row in the matrix, means assign value 3 to the cell in row 0 column 2.

Columns: The columns represents the constraints. There are four kinds of constraints:

One value per cell constraint (Row-Column): Each cell must contain exactly one value. For a 9 by 9 Sudoku puzzle, we have $9 * 9 = 81$ cells, and therefore, we have 81 row-column constraints, one for each cell. If a cell contains a value (regardless of

what it is), we assign it a value of '1'. This means for rows $(r=0, c=0, v=1)$, $(r=0, c=0, v=2)$, ... $(r=0, c=0, v=9)$ in the matrix, they will all have '1' in the column corresponding to the row-column constraint $(r=0, c=0)$. This construction means only one of the above is selected for $(r=0, c=0)$, satisfying this constraint. Same applies for the other cells.

Row constraint (Row-Value): Each row must contain each number exactly once. For a 9 by 9 Sudoku puzzle, we have 9 rows and 9 possible values that can be assigned to each row, i.e., $9 \times 9 = 81$ row-value pairs. Therefore, we have 81 row-value constraints, one for each row-value pair. If a row contains a value (regardless in which column), we assign it a value of '1'. This means for rows $(r=0, c=0, v=1)$, $(r=0, c=1, v=1)$, ... $(r=0, c=8, v=1)$ in the matrix, they will all have '1' in the matrix column corresponding to the row-value constraint $(r=0, v=1)$. This construction means only one of the above matrix rows is selected in order to satisfy the row-value constraint $(r=0, v=1)$. Same applies for the other rows.

Column constraint (Column-Value): Each column must contain each number exactly once. For a 9 by 9 Sudoku puzzle, we have 9 columns and 9 possible values that can be assigned to each column, i.e., $9 \times 9 = 81$ column-value pairs. Therefore, we have 81 column-value constraints, one for each column-value pair. If a column contains a value (regardless in which row), we assign it a value of '1'. This means for rows $(r=0, c=0, v=1)$, $(r=1, c=0, v=1)$, ... $(r=8, c=0, v=1)$ in the matrix, they will all have '1' in the matrix column corresponding to the column-value constraint $(c=0, v=1)$. This construction means only one of the above rows is selected in order to satisfy the column-value constraint $(c=0, v=1)$. Same applies for the other columns.

Box Constraint (Box-Value): Each box must contain each value exactly once. For a 9 by 9 Sudoku puzzle, we have 9 boxes and 9 possible values that can be assigned to each box, i.e., $9 \times 9 = 81$ box-value pairs. Therefore, we have 81 box-value constraints, one for each box-value pair. If a box contains a value (regardless in which cell of the box), we assign it a value of '1'. This means for rows $(r=0, c=0, v=1)$, $(r=0, c=1, v=1)$, ... $(r=2, c=2, v=1)$ in the matrix, they will all have '1' in the matrix column corresponding to the box-value constraint $(b=0, v=1)$. This construction means only one of the above rows is selected in order to satisfy the box-value constraint $(b=0, v=1)$. Same applies for the other boxes.

Why this works? For exact cover, we select rows such that there is a single '1' in all subsequent columns. The way we constructed the constraints, this is equivalent to selecting value assignments (the rows) such that only value per cell, that each row and column cannot have duplicate values, and each box also cannot have duplicate values. If there are duplicates, then there will be more than a '1' in one of the column constraints. By forcing to select a '1' in each column, we also ensure a value is selected for every cell, and all rows, columns and boxes have all values present.

This concludes the background. In the following, we will describe the tasks of this assignment.

4 Tasks

The assignment is broken up into a number of tasks. Apart from Task A that should be completed initially, all other tasks can be completed in an order you are more comfortable with, but we have ordered them according to what we perceive to be their difficulty. Task E is considered a high distinction task and hence we suggest to tackle this after you have completed the other tasks.

Task A: Implement Sudoku Grid (4 marks)

Implement the grid representation, including reading in from file and outputting a solved grid to an output file. Note we will use the output file to evaluate the correctness of your implementations and algorithms.

A typically Sudoku puzzle is played on a 9 by 9 grid, but there are 4 by 4, 16 by 16, 25 by 25 and larger. In this task and subsequent tasks, your implementation should be able to represent and solve Sudoku and variants of any valid sizes, e.g., 4 by 4 and above. You won't get a grid size that isn't a perfect square, e.g., 7 by 7 is not a valid grid size, and all puzzles will be square in shape.

In addition, the values/symbols of the puzzles may not be sequential digits, e.g., 1-9 for a 9 by 9 grid, but could be any set of 9 *unique non-negative integer digits*. The same Sudoku rules and constraints still hold for non-standard set of values/symbols. Your implementation should be able to read this in and handle any set of valid integer values/symbols.

Task B: Implement Backtracking Solver for Sudoku (9 marks)

To help to understand the problem and the challenges involved, the first task is to develop a backtracking approach to solve Sudoku puzzles.

Task C: Exact Cover Solver - Algorithm X (7 marks)

In this task, you will implement the first approaches to solve Sudoku as an exact cover problem - Algorithm X.

Task D: Exact Cover Solver - Dancing Links (7 marks)

In this task, you will implement the second of two approaches to solve Sudoku as an exact cover problem - the Dancing Links algorithm. We suggest to attempt to understand and implement Algorithm X first, then the Dancing Links approach.

Task E: Killer Sudoku Solver (16 marks)

In this task, you will take what you have learnt from the first two tasks and devise and implement 2 solvers for Killer Sudoku puzzles. One will be based on backtracking and the other should be more efficient (in running time) than the backtracking one. Your implementation will be assessed for its ability to solve Killer Sudoku puzzles of various difficulties within reasonable time, as well as your proposed approach, which will be detailed in a short (1-2 pages) report. We are as interested in your approach and rationale behind it as much as the correctness and efficiency of your approach.

5 Details for all tasks

To help you get started and to provide a framework for testing, you are provided with skeleton code that implements some of the mechanics of the Sudoku program. The main class (RmitSudokuTester.java) implements functionality of Sudoku solving and parsing parameters. The list of main java files provided are listed in Table 1.

file	description
RmitSudokuTester.java	Class implementing basic IO and processing code. <i>Suggest to not modify.</i>
grid/SudokuGrid.java	Abstract class for Sudoku grids <i>Can add to, but don't modify existing method interfaces.</i>
grid/StdSudokuGrid.java	Class for standard Sudoku grids. <i>Please complete the implementation.</i>
grid/KillerSudokuGrid.java	Class for Killer Sudoku grids. <i>Please complete the implementation.</i>
solver/SudokuSolver.java	Abstract class for Sudoku solver algorithms. <i>Can add to, but don't modify existing method interfaces.</i>
solver/StdSudokuSolver.java	Abstract class for standard Sudoku solver algorithms, extends SudokuSolver class. This has empty implementation and added in case you wanted to add some common methods/attributes for solving standard Sudoku puzzles, but you don't have to touch this if you don't have these. <i>Can add to.</i>
solver/KillerSudokuSolver.java	Abstract class for Killer Sudoku solver algorithms, extends SudokuSolver class. This has empty implementation and added in case you wanted to add some common methods/attributes for solving Killer Sudoku puzzles, but you don't have to touch this if you don't have these. <i>Can add to.</i>
solver/BackTrackingSolver.java	Class for solving standard Sudoku with backtracking. <i>Please complete the implementation.</i>
solver/AlgorXSolver.java	Class for solving standard Sudoku with Algorithm X algorithm. <i>Please complete implementation.</i>
solver/DancingLinksSolver.java	Class for solving standard Sudoku with the Dancing Links approach. <i>Please complete the implementation.</i>
solver/KillerBackTrackingSolver.java	Class for solving Killer Sudoku with backtracking. <i>Please complete the implementation.</i>
solver/KillerAdvancedSolver.java	Class for solving Killer Sudoku with your advanced algorithm. <i>Please complete the implementation.</i>

Table 1: Table of supplied Java files.

We also strongly suggest to avoid modifying RmitSudokuTester.java, as they form the IO code, and any of the interfaces for the abstract classes. If you wish, you may add java classes/files and methods, but it should be within the structure of the skeleton code, i.e., keep the same directory structure. Similar to assignment 1, this is to minimise compiling and running issues. Please ensure there are no compilation errors because of any modifications. You should implement all the missing functionality in *.java files.

Ensure your structure compiles and runs on the **core teaching servers**. *Note that the onus is on you to ensure correct compilation and behaviour on the core teaching servers before submission, please heed this warning.*

As a friendly reminder, remember how packages work and IDE like Eclipse will automatically add the package qualifiers to files created in their environments. This is a large source of compile errors on the core teaching servers, so remove these package qualifiers when testing on the core teaching servers.

Compiling and Executing

To compile the files, run the following command from the root directory (the directory that RmitSudokuTester.java is in):

```
javac *.java grid/*.java solver/*.java
```

Note that for Windows machine, remember to replace ‘/’ with ‘\’.

To run the framework:

```
java RmitSudokuTester [puzzle fileName] [game type] [solver type]
                        [visualisation] <output fileName>
```

where

- puzzle fileName: name of file containing the input puzzle/grid to solve.
- game type: type of sudoku game, one of {sudoku, killer}.
- solver type: Type of solver to use, depends on the game type.
 - If (standard) Sudoku is specified (sudoku), then solver should be one of {backtracking, algorx, dancing}, where *backtracking* is the backtracking algorithm for standard Sudoku, *algorx* and *dancing* are the exact cover approaches for standard Sudoku.
 - If Killer Sudoku is specified (killer), then solver should be one of {backtracking, advanced} where *backtracking* is the backtracking algorithm for Killer Sudoku and *advanced* is the most efficient algorithm you can devise for solving Killer Sudoku.
- visualisation: whether to output grid before and another after solving, one of {n , y}.
- output fileName: (optional) If specified, the solved grid will be outputted to this file. Ensure your implementation implements this as it will be used for testing (see the outputBoard() methods for the classes in grid directory).

5.1 Details of Files

In this section we describe the format of the input and output files.

Puzzle file (input)

This specifies the puzzle and includes information:

- size of puzzle
- list of symbols used
- location of the cells with initial values
- (for Killer Sudoku, location of cages and their totals).

Standard Sudoku (input) The exact format for standard Sudoku is as follows:

```
[size/dimensions of puzzle]
[list of valid symbols]
[tuples of row,column value, one tuple per line]
```

For instance, for the tuple

0,0 1

means there is a value 1 in cell ($r = 0$, $c = 0$).

Using the example from Figure 1a, the first few lines of the input file corresponding to this example would be:

```
9
1 2 3 4 5 6 7 8 9
0,0 5
0,1 3
0,4 7
1,0 6
1,3 1
...
```

Killer Sudoku (input) The exact format for Killer Sudoku is as follows:

```
[size/dimensions of puzzle]
[list of valid symbols]
[number of cages]
[Total of cage, list of row,column for each cage, one per line]
```

Using the example from Figure 2a, the first few lines of file corresponding to this example would be:

```
9
1 2 3 4 5 6 7 8 9
29
3 0,0 0,1
15 0,2 0,3 0,4
...
```

Solved/filled in grid output file (output)

After a puzzle is solved, the output format of a filled in grid should be a comma separate file. For a n by n grid, with the cells referenced by (row, column) and the top left corner is (0,0) (row =0, column = 0), should have the following output (we included the first row and column for indexing purposes but they shouldn't be in your output file):

	$c = 0$	$c = 1$	$c = 2$	\dots	$c = n - 1$
$r = 0$	$v_{0,0},$	$v_{0,1},$	$v_{0,2},$	$\dots,$	$v_{0,n-1}$
$r = 1$	$v_{1,0},$	$v_{1,1},$	$v_{1,2},$	$\dots,$	$v_{1,n-1}$
\vdots	\vdots		\ddots		\vdots
$r = n - 1$	$v_{n-1,0},$	$v_{n-1,1},$	$v_{n-1,2},$	$\dots,$	$v_{n-1,n-1}$

where $v_{r,c}$ is the value of cell (r,c). More concretely, for a 4 by 4 puzzle using 1-4 values/symbols, a sample valid filled grid could be:

2,1,4,3
4,3,2,1
3,2,1,4
1,4,3,2

5.2 Clarification to Specifications

Please periodically check the assignment FAQ for further clarifications about specifications. In addition, the lecturer and course coordinator will go through different aspects of the assignment each week, so be sure to check the course material page on Canvas to see if there are additional notes posted.

6 Submission

The final submission will consist of:

- The Java source code of your implementations, including the ones we provided. Keep the same folder structure as provided in skeleton (otherwise the packages won't work). Maintaining the folder structure, ensure all the java source files are within the folder tree structure. Rename the root folder as **Assign2-<your student number>**. Specifically, if your student number is s12345, then all the source code files should be within the root folder Assign2-s12345 and its children folders.
- All folder (and files within) should be zipped up and named as **Assign2-<your student number>.zip**. E.g., if your student number is s12345, then your submission file should be called **Assign2-s12345.zip**, and when we unzip that zip file, then all the submission files should be in the folder Assign2-s12345.
- Your report of your approach, called "assign2Report.pdf". Place this pdf within the Java source file root directory/folder, e.g., Assign2-s12345.

Note: submission of the zip file will be done via Canvas.

Late Submission Penalty Late submissions will incur a 10% penalty on the total marks of the corresponding assessment task per day or part of day late. Submissions that are late by 5 days or more are not accepted and will be awarded zero, unless special consideration has been granted. Granted Special Considerations with new due date set after the results have been released (typically 2 weeks after the deadline) will automatically result in an equivalent assessment in the form of a practical test, assessing the same knowledge and skills of the assignment (location and time to be arranged by the instructor). Please ensure your submission is correct (all files are there, compiles etc), re-submissions after the due date and time will be considered as late submissions. The core teaching servers and Canvas can be slow, so please ensure you have your assignments are done and submitted a little before the submission deadline to avoid submitting late.

7 Academic integrity and plagiarism (standard warning)

Academic integrity is about honest presentation of your academic work. It means acknowledging the work of others while developing your own insights, knowledge and ideas. You should take extreme care that you have:

- Acknowledged words, data, diagrams, models, frameworks and/or ideas of others you have quoted (i.e. directly copied), summarised, paraphrased, discussed or mentioned in your assessment through the appropriate referencing methods
- Provided a reference list of the publication details so your reader can locate the source if necessary. This includes material taken from Internet sites. If you do not acknowledge the sources of your material, you may be accused of plagiarism because you have passed off the work and ideas of another person without appropriate referencing, as if they were your own.

RMIT University treats plagiarism as a very serious offence constituting misconduct. Misconduct and plagiarism covers a variety of inappropriate behaviours, including:

- Failure to properly document a source
- Copyright material from the internet or databases
- Collusion between students
- Assignment buying
- Submitting assignments of other students from previous semesters

For further information on our policies and procedures, please refer to the following: <https://www.rmit.edu.au/students/student-essentials/rights-and-responsibilities/academic-integrity>.

8 Getting Help

There are multiple venues to get help. There are weekly consultation hours (see Canvas for time and location details). In addition, you are encouraged to discuss any issues you have with your Tutor or Lab Demonstrator. We will also be posting common questions on the assignment 2 FAQ section on Canvas and we encourage you to check and participate in the discussion forum on Canvas. However, please **refrain from posting solutions**, particularly as this assignment is focused on algorithmic and data structure design.

9 Marking guidelines

The assignment will be marked out of 50 and a bonus of up to 3 marks.

The assessment in this assignment will be broken down into a number of components. The following criteria will be considered when allocating marks. All evaluation will be done on the core teaching servers.

Task A (4/50)

For this task, we will evaluate whether you are able to read in puzzle input files, represent and construct a grid and whether you can output a solved grid to output files.

Task B (9/50):

For this task, we will evaluate your implementation and algorithm on whether:

1. Implementation and Approach: It implements the backtracking algorithm to solve Sudoku puzzles.
2. Correctness: Whether it correctly solves Sudoku puzzles.
3. Efficiency: As part of correctness, your implementation should not take excessively long to solve a puzzle. We will benchmark the running time against our non-optimised solution and add a margin on top, and solutions taking longer than this will be considered as inefficient.

Task C (7/50):

For this task, we will evaluate your implementation and algorithm on whether:

1. Implementation and Approach: It implements AlgorithmX approach to solve Sudoku puzzles.
2. Correctness: Whether it correctly solves Sudoku puzzles.
3. Efficiency: As part of correctness, your implementation should not take excessively long to solve a puzzle. We will benchmark the running time against our non-optimised solution of the same algorithm and add a margin on top, and solutions taking longer than this will be considered as inefficient.

Task D (7/50):

For this task, we will evaluate your implementation and algorithm on whether:

1. Implementation and Approach: It implements the Dancing Link approach to solve Sudoku puzzles.
2. Correctness: Whether it correctly solves Sudoku puzzles.
3. Efficiency: As part of correctness, your implementation should not take excessively long to solve a puzzle. We will benchmark the running time against our non-optimised solution of the same algorithm and add a margin on top, and solutions taking longer than this will be considered as inefficient.

Task E (16/50):

For this task, we will evaluate your two implementations and algorithms on whether:

1. Implementation and Approach: It takes reasonable approaches and can solve Killer Sudoku puzzles.
2. Correctness: Whether they correctly solves Killer Sudoku puzzles.
3. Description of Approach: In addition to the code, you will be assessed on a description of your proposed *advanced* approach, which will help us to understand your approach. Include how you represented the Killer Sudoku grid, how you approached solving Killer Sudoku puzzles with your advanced approach and your rationale behind it. You may cite other references you have researched upon and utilised the information within. Include a comparison of the backtracking and advanced algorithms and include empirical evidence (similar to Assignment 1) to show your advanced approach is more efficient than your backtracking approach. This report should be no longer than *two* pages and submitted as a pdf as part of your submission. You will be assessed on the approach, its clarity and whether it reflects your code.

Interview (5/50) We will conduct a short interview with you about the tasks above. This is compulsory and if not completed, will lead to the assignment not being assessed and a mark of 0 given. A schedule will be released and the time can be further negotiated.

Coding style and Commenting (2/50):

You will be evaluated on your level of commenting, readability and modularity. This should be at least at the level expected of a first year masters student who has done some programming courses.

Bonus (up to 3 marks extra) To encourage exploration of novel solvers to Killer Sudoku, we will award up to 3 bonus marks to individuals who has the fastest empirical (average) running times for the *advanced* solver. The student with the fastest algorithm will receive 3 bonus marks, 2nd fastest 2 marks, and 3rd fastest will be awarded 1 bonus marks.

Note the bonus mark will be added to your overall course total and if after adding the bonus your overall course total is > 100 , your total will be capped at 100.