# Reflection Document
# Assignment 2 - Cis*3190 Legacy Systems
Brayden Cowell - 0844864 - bcowell@mail.uoguelph.ca

## Instructions
> gnatmake -Wall solver.adb
> ./solver
Run the program and give it an input and output file when prompted.
I provided some example puzzles so when prompted for an input file you could say easy1.txt and so forth. If either the input file is missing, or the output file is not formatted like 'out.txt', it will print and error and ask again for correct filenames.

## Algorithm
Source:
https://github.com/UlrikHjort/Sudoku-Solver/blob/6eee159f6203907e63af2bd7fd13d6d266d73162/Sudoku_Solver.adb#L137-L155 (Algorithm modeled after Lines 137-155).

I chose to implement a Brute-Force algorithm for finding the solution to any sudoku. I settled on this algorithm mostly because it guarantees a solution to any valid sudoku puzzle. The only downside is the resource cost has the chance to be somewhat slow compared to backtracking or narrowing down possibilities.

This brute force algorithm works by visiting every empty cell in the unsolved puzzle in order. Starting at the top left square and going through each point, row by row, testing every possible answer until it reaches a solution.

"For example, a brute force program would solve a puzzle by placing the digit "1" in the first cell and checking if it is allowed to be there. If there are no violations then the algorithm advances to the next cell, and places a "1" in that cell. When checking for violations, it is discovered that the "1" is not allowed, so the value is advanced to a "2". If a cell is discovered where none of the 9 digits is allowed, then the algorithm leaves that cell blank and moves back to the previous cell. The value in that cell is then increased by 1. The algorithm is repeated until a valid solution for all 81 cells is found."
From https://en.wikipedia.org/wiki/Sudoku_solving_algorithms

I went about checking violations in three ways. The first two were making sure that any number could not exist in the same row and column. And I also checked each individual three by three grid for any matching numbers that would invalidated the puzzle.

## Parsing the puzzle from a file

```
24          -- Read unsolved puzzle from file.
25          loop
26                  begin
27                          put_line("Enter an input filename:");
28                          get_line(filename_in,last1);
29                          put_line("And enter an output filename:");
30                          get_line(filename_out, last2);
31
32                          open(input_file, in_file, filename_in(1..last1));
33
34                          exit;
35                  exception
36                          when Name_Error => put_line("Invalid filename!");
37                  end;
38          end loop;
39          while not End_OF_File (Input_File) loop
40                  -- Read the puzzle in as a string.
41                  Get(Input_File, line);
42                  Put_line(line);
43
44                  x := 1;
45                  y := 9;
46                  -- Seperate the whole file into 9 character strings
47                  for i in 1..9 loop
48                          temp_line := line(x..y);
49                          -- Now seperate the 9-char strings into individual chars
50                          for j in 1..9 loop
51                                  -- And assign each number to its possition in the 9x9 puzzle.
52                                  puzzle(i,j) := character'Pos(temp_line(j)) - 48; -- It's ASCII so subtract the code for 0 (48).
53                          end loop;
54                          x := x + 9;
55                          y := y + 9;
56                  end loop;
57          end loop;
58          close(input_file);
```

This is an image of how I implemented the parsing of the sudoku puzzles in Ada. As you can see I prompt the user for an input and output file. I print an error message if the file names don't end with .txt or if the input file doesn't exist. Then I read the entire file's text in as one long string, and separate it into nine rows in order. After that I go through each row and place the integer conversion of the character into my array. I end by closing the file.

To output my solved puzzle to a file, I simply called Set_Output with my file. This redirects all text output to the named file. Then I call my print_board procedure, which uses put_lines in order to display the sudoku puzzle properly.

## Questions

- Would it have been easier to write the program in a language such as C?

I think the program would have been easier for me to write in C. Although I used many of the same principles with the array-based implementation, I think I could have done it faster in C.

- What were the greatest problems faced while designing the algorithm in Ada?

I think the hardest thing for me was figuring out how Strings work in Ada. I found a few comments in the Course Link discussion forums really helpful. Also once I got reading from a file working output, both to standard IO and to files, was a cinch.

- Is there a better way of writing the program?

If I had to use as little resources as possible, I would change the algorithm from brute-force to something involving backtracking or narrowing down the possibilities of each individual square. Although I found Ada pretty straightforward. I also think a Python implementation of a Sudoku solver would have been slightly faster.

- What particular structures made Ada a good choice?

I found Ada's error messages to be very concise and really helpful. There were a few times where I either misspelled variable names or forgot to include a package. Ada reported exactly what I did wrong, which was a nice change coming for a C-background.

- Given your knowledge of programming, was Ada easy to learn?

I found Ada actually very easy to learn. I had some previous experience with Oracle's Procedural Language SQL, and found the two very similar in practice.