IE523: Financial Computing
Fall, 2022
Programming Assignment 3: Dutch Miracle Sudoku Solver via Exhaustive-Search using Recursion
Due Date: 30 September 2022

©Prof. R.S. Sreenivas

A Sudoku Puzzle consists of a 9×9 grid, that is subdivided into nine 3×3 blocks. Each entry in the grid must be filled with a number from $\{1, 2, \ldots, 9\}$, where some entries are already filled in. The constraints are that every number should occur exactly once in each row, each column, and each 3×3 block.

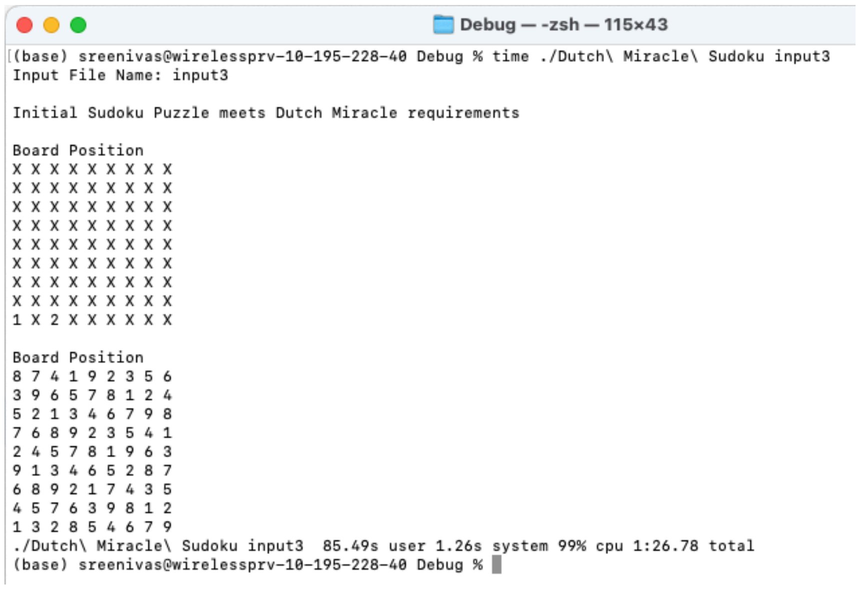
I want to do a little more than do a generic Sudoku Solver using recursive backtracking – it has been done before, and there is no charm in doing things that are not novel ☺.

Recently, there has been an enhanced-version of Sudoku puzzles called the Dutch Miracle Sudoku Puzzle. This has also been addressed by a few bloggers, here is one from a YouTube Blogger. In this version, the standard Sudoku rules apply. In addition, there are two more rules that have to do with the entries along the positive-diagonal of the puzzle. That is, these rules apply to fifteen-many South-West-to-North-East diagonals in the puzzle identified using the row- and column-indices as

- $\{(1,0),(0,1)\}$,
- $\{(2,0),(1,1),(0,2)\}$,
- $\{(3,0),(2,1),(1,2),(0,3)\}$,
- $\{(4,0),(3,1),(2,2),(1,3),(0,4)\}$,
- $\cdots$,
- $\{(8,0),(7,1),(6,2),(5,3),(4,4),(3,5),(2,6),(1,7),(0,8)\}$,
- $\{(8,1),(7,2),(6,3),(5,4),(4,5),(3,6),(2,7),(1,8)\}$,
- $\{(8,2),(7,3),(6,4),(5,5),(4,6),(3,7),(2,8)\}$,
- $\cdots$,
- $\{(8,7),(7,8)\}$

For these positive-diagonal entries, we require that

1. all the digits are different, and
2. adjacent digits (i.e. those touching at a corner) must have a difference of at least 4.



```
Debug — -zsh — 115×43
[(base) sreenivas@wirelessprv-10-195-228-40 Debug % time ./Dutch\ Miracle\ Sudoku input3
Input File Name: input3

Initial Sudoku Puzzle meets Dutch Miracle requirements

Board Position
X X X X X X X X X
X X X X X X X X X
X X X X X X X X X
X X X X X X X X X
X X X X X X X X X
X X X X X X X X X
X X X X X X X X X
X X X X X X X X X
1 X 2 X X X X X X

Board Position
8 7 4 1 9 2 3 5 6
3 9 6 5 7 8 1 2 4
5 2 1 3 4 6 7 9 8
7 6 8 9 2 3 5 4 1
2 4 5 7 8 1 9 6 3
9 1 3 4 6 5 2 8 7
6 8 9 2 1 7 4 3 5
4 5 7 6 3 9 8 1 2
1 3 2 8 5 4 6 7 9
./Dutch\ Miracle\ Sudoku input3  85.49s user 1.26s system 99% cpu 1:26.78 total
(base) sreenivas@wirelessprv-10-195-228-40 Debug %
```

Figure 1: Sample output for the first part of Programming Assignment 3.

Write a C++ program that takes the initial board position as input and presents the solved puzzle as the output. Your program is to take the name of the input file as a command-line input, print out the initial- and final-board positions (cf. figure 1 for an illustration). The input file is formatted as follows – there are 9 rows and 9 numbers, where the number 0 represents the unfilled-value in the initial board. Figure 2 contains an illustrative sample.

The approach you will take for this assignment is to use recursion to implement an exhaustive-search procedure that solves a Sudoku puzzle. That is, you start with the first unfilled-value in the board and pick a valid assignment that satisfies the row-, column-, block-constraints along with the two additional constraints identified above. Following this, you move to the next unfilled position and do the same. If at some point in this process you reach a partially-filled board-position that cannot be filled further under the three constraints, you backtrack to the last assignment that was made before you got stuck and modify the assigned value accordingly. This can be implemented using recursion as follows:

First Part of the Assignment

Your implementation should use a class Sudoku with appropriately defined private and public functions that

1. check if the row-, column- and block-assignment constraints are met,

```
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
1 0 2 0 0 0 0 0 0
```

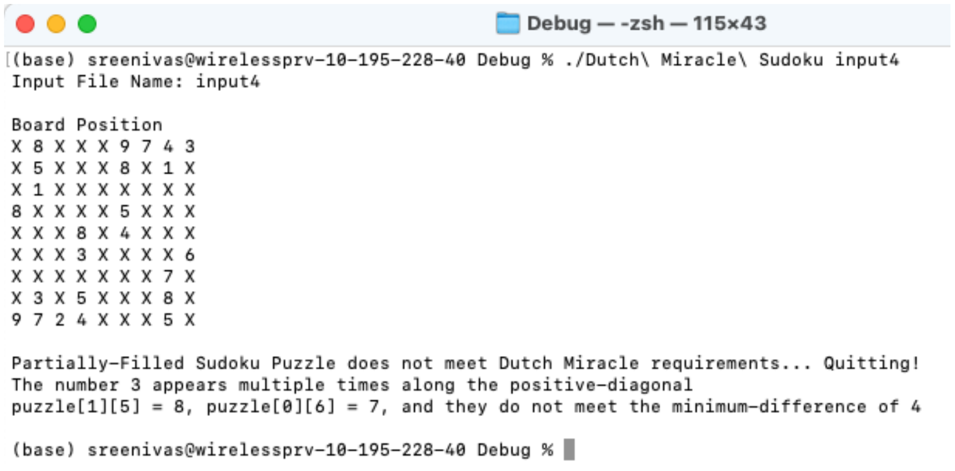
Figure 2: Sample input file called input3, which was used in the illustrative example of figure 1. The 0's represent the incomplete board positions/values.

```
*Boolean* **Solve**(row, column)
1: Find an $i \geq$ row and $j \geq$ column such that puzzle[i][j] $= 0$. If you cannot
   find such an $i$ and $j$, then you have solved the puzzle.
2: for $k \in \{1, 2, \ldots, 9\}$ do
3:     puzzle[i][j] $= k$.
4:     if Row-, Column-, Block-Assignment, and the two additional constraints
       are satisfied by the above assignment, and Solve($i$, $j$) returns true then
5:         Return true.
6:     end if
7: end for
   { /* If we got here then all assignments made to puzzle[i][j] are invalid. So,
   we reset its value and return false*/}.
8: puzzle[i][j] $= 0$
9: Return false.
```

Figure 3: Pseudo-code for the recursive implementation of the exhaustive-search algorithm for the Dutch Miracle Sudoku puzzle. You solve the puzzle by calling **Solve**(0,0), assuming the indices are in the range $\{0, 1, \ldots, 8\}$.

2. reads the incomplete puzzle from an input file that is read at the command-line,
3. prints the puzzle at any point of the search.
4. it might be the case that certain partially-filled puzzles do not meet the Dutch Miracle Sudoku Rules. I want your code to check this condition. Figure 4 shows an illustration of this.

along with a recursive procedure that solves the puzzle that was introduced above.



```
[(base) sreenivas@wirelessprv-10-195-228-40 Debug % ./Dutch\ Miracle\ Sudoku input4
 Input File Name: input4

 Board Position
 X 8 X X X 9 7 4 3
 X 5 X X X 8 X 1 X
 X 1 X X X X X X X
 8 X X X X 5 X X X
 X X X 8 X 4 X X X
 X X X 3 X X X X 6
 X X X X X X X 7 X
 X 3 X 5 X X X 8 X
 9 7 2 4 X X X 5 X

 Partially-Filled Sudoku Puzzle does not meet Dutch Miracle requirements... Quitting!
 The number 3 appears multiple times along the positive-diagonal
 puzzle[1][5] = 8, puzzle[0][6] = 7, and they do not meet the minimum-difference of 4

 (base) sreenivas@wirelessprv-10-195-228-40 Debug %
```

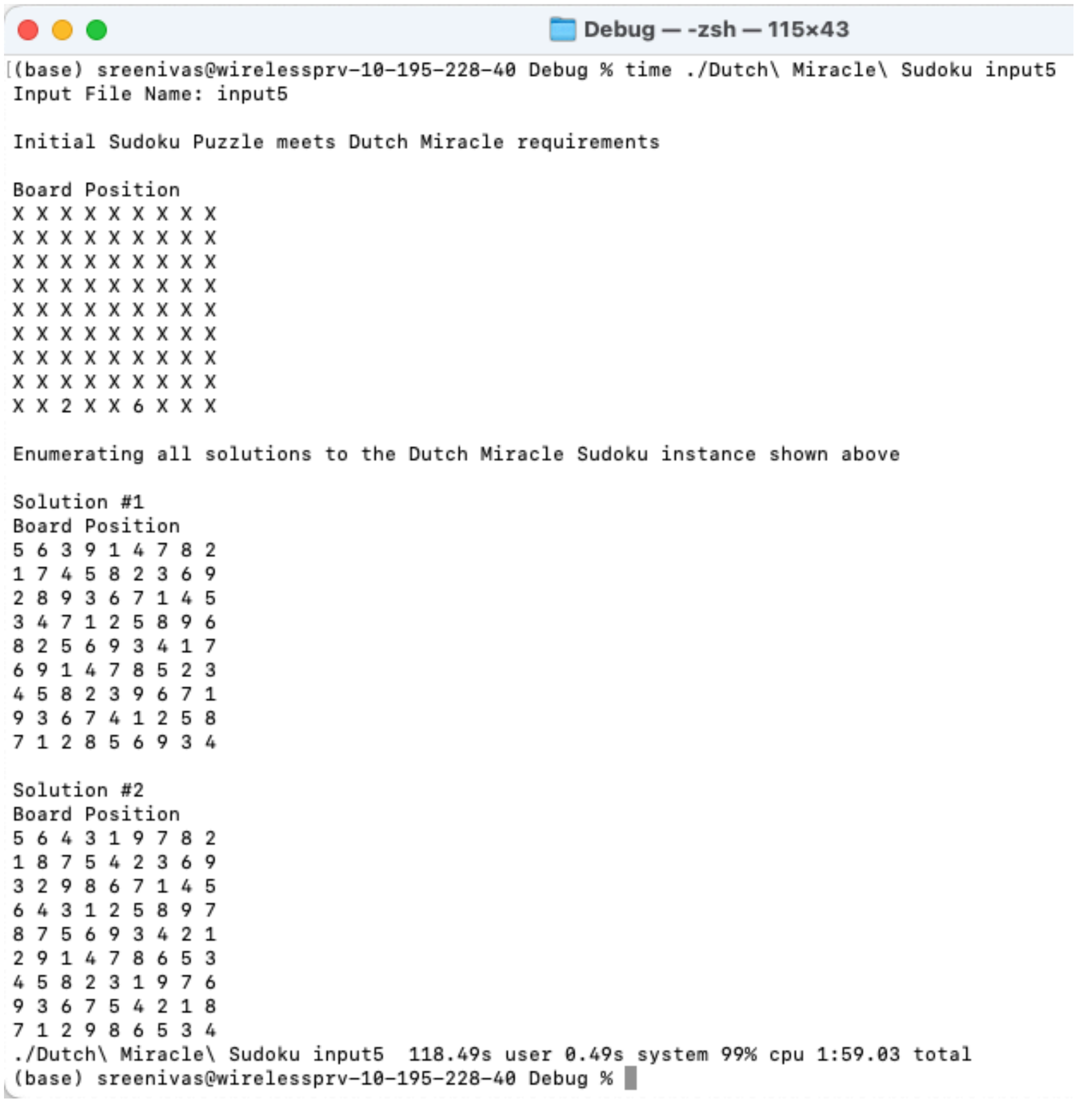
Figure 4: Sample output when there is no solution.

Second Part of the Assignment

The following blog mentions Sudoku puzzles that can have multiple solutions. It stands to reason that the same might be true for the Dutch Miracle version of the Sudoku puzzle. I want you to modify your code from the second programming assignment to find *all* solutions to a Dutch Miracle Sudoku puzzle.

You definitely want to be cautious with this – the empty Sudoku puzzle has theoretically 6,670,903,752,021,072,936,960 solutions. The last thing you want to do is to attempt to print them out! You will find five input files on Canvas that identifies five different (Dutch Miracle) Sudoku puzzles. Two of them have just one solution, while the other two have multiple solutions. I suggest you run your code on these input files.

Write a C++ program that takes the initial board position as input and presents **all** solutions the puzzle as the output. Your program is to take the name of the input file as a command-line input, print out the initial- and final-board positions (cf. figure 5 for an illustration).



```
Debug — -zsh — 115×43
[(base) sreenivas@wirelessprv-10-195-228-40 Debug % time ./Dutch\ Miracle\ Sudoku input5
Input File Name: input5

Initial Sudoku Puzzle meets Dutch Miracle requirements

Board Position
X X X X X X X X X
X X X X X X X X X
X X X X X X X X X
X X X X X X X X X
X X X X X X X X X
X X X X X X X X X
X X X X X X X X X
X X X X X X X X X
X X 2 X X 6 X X X

Enumerating all solutions to the Dutch Miracle Sudoku instance shown above

Solution #1
Board Position
5 6 3 9 1 4 7 8 2
1 7 4 5 8 2 3 6 9
2 8 9 3 6 7 1 4 5
3 4 7 1 2 5 8 9 6
8 2 5 6 9 3 4 1 7
6 9 1 4 7 8 5 2 3
4 5 8 2 3 9 6 7 1
9 3 6 7 4 1 2 5 8
7 1 2 8 5 6 9 3 4

Solution #2
Board Position
5 6 4 3 1 9 7 8 2
1 8 7 5 4 2 3 6 9
3 2 9 8 6 7 1 4 5
6 4 3 1 2 5 8 9 7
8 7 5 6 9 3 4 2 1
2 9 1 4 7 8 6 5 3
4 5 8 2 3 1 9 7 6
9 3 6 7 5 4 2 1 8
7 1 2 9 8 6 5 3 4
./Dutch\ Miracle\ Sudoku input5  118.49s user 0.49s system 99% cpu 1:59.03 total
(base) sreenivas@wirelessprv-10-195-228-40 Debug %
```

Figure 5: Sample output of all solutions.

The approach you will take for this assignment is to use recursion to modify the exhaustive-search procedure that solved the Sudoku puzzle in the second programming assignment. This is a relatively straightforward thing to do, if you have understood the recursion in the previous programming assignment.