7CCS2PEP- C++ Resit Task, August 2022/3

You are asked to implement a C++ program that solves a puzzle. The deadline for the submission (via Github) is 26th July 2023 at 16:00. There will be no automated tests for this resit, but the template and instructions include 2 test cases to aid you in the process of solving this puzzle.

⚠ IMPORTANT

- Please make sure you adhere to C++ Standard 11.
- No libraries are allowed to be used apart from <vector>, <iostream> and <Memory>.
- You are free to create additional .h files to help you create the data structure of your choosing. But this file cannot include any other libraries apart from the aforementioned.
- Also note that the running time for each test case will be restricted to a maximum of 4 minutes on my machine. Each test case should not take longer than a few seconds (normally under 3 or 4), so 4 minutes should allow for any differences in code efficiency or computer processing power.
- If you spot any mistakes or wish to clarify any of the tasks, please drop me a line.

⚠ Disclaimer

Please include the following paragraph at the top of your SetSolver.h file as a comment.

"The work I submitted represents my own effort and that I have implemented the code entirely on my own and I have not copied any line from anyone or anywhere. If I have, a comment was added to denote the source of that line or lines. "

Please, do not be tempted to ask Copilot/Chat GPT or any other Generative AI engine for that matter!

Background

Number Sets is a numbers puzzle, which is very similar to Sudoku. It is played on a 9x9 board, where the board is partially filled with numbers. Your task is to write a program that fills the board with the missing numbers in the white squares, taking into account all the rules described below. All tasks need to be implemented using 0-8 index to represent cell positions for both rows and columns.

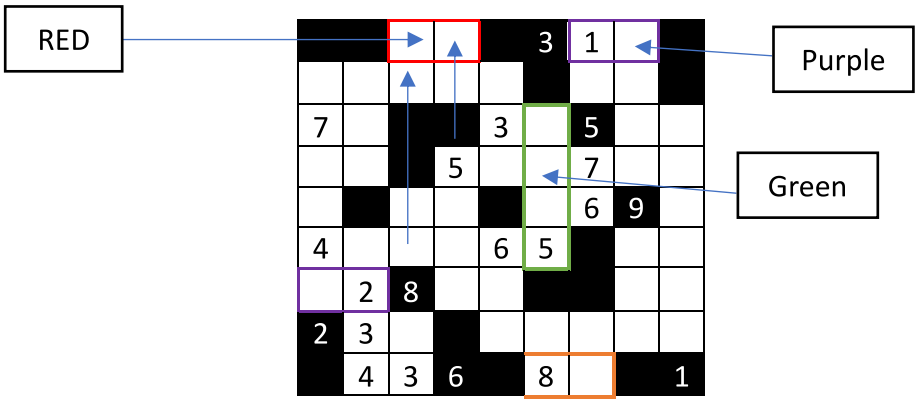
Table 1- Board 1

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

Rows and columns are divided into compartments of **white** squares. To solve the puzzle, use single numbers to complete the missing Set in each compartment. There are no negative numbers! A Set of numbers is said to be a group of numbers that contains no gaps in the middle (e.g., 1, 2, 3 or 5, 6, 7, 8, 9). Having said that, numbers in a given compartment can appear in any order (e.g., 1, 3, 2 and 6, 9, 8, 5 and 7). No single number can repeat in any row or column (irrespective of compartments).

A black square is a blocked square, which means the solver cannot input a number into it. If a black square contains a number, it helps the solver by providing a clue, effectively removing that number as an option from that row and column. In other words, a number in a black square cannot be part of any Set in that row or column.

As an example, the board shown above is repeated below showing different compartments in Red, Purple and Green. Both the Red and purple squares cannot contain the number 3 as it appears in the black square. The Red compartment cannot contain the numbers 1 as it already appears on that row or 8 and 5 (respectively from left) as 8 appears in the left column and 5 on the right column.



Using this logic, we can use the process of elimination to deduce which number fits in which white square. Looking above at the orange compartment (bottom row), it already contains the number 8. This means that the empty white square to its right, can only be a 7 or a 9 (a set contains no gaps but can appear in any order). The bottom row already contains the numbers 3, 4, 6, and 1 – so these numbers can be removed as possible options. However, the column (third from the right) contains 1, 5, 7 and 6, so these could also be removed as possibilities, leaving 9 as the only viable option.

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

Table 2- solution to board 1

As can be view in the above solution to the puzzle, the 'no gap sequence' needs to happen only within a compartment, not a row or a column. For example, the first row has two compartments marked in Red and Purple. Each of them have number running sequentially, but the compartments do **not** have to be a continuation of one another – there can be a gap between the compartments!

Task 1 [2 marks]:

To represent the board, a vector of strings will be provided in the .cpp test file. It represents an empty black square as Zero, an empty white square as asterisk and a black square with a number as a **negative** of that that number. We're only using negative numbers for black squares that have numbers. For example, the number 3 in top row (4^{th} column from the right in a black square) will be represented as -3 in the string. 3 to denote the fact that the square contains a number, and the minus denotes that it's a black square. For clarity, white squares can only contain the numbers 1 through to 9. (no negative numbers are used).

As an example, the above board will be represented in the following way:

```
vector<string> easyBoard =   {"00**0-31*0",
                              "*****0**0",
                               "7*003*-5**",
                              "**05**7**",
                              "*0**0*6-9*",
                              "4***650**",
                              "*2-8**00**",
                              "-23*0*****",
                               "043-608*0-1"};
```

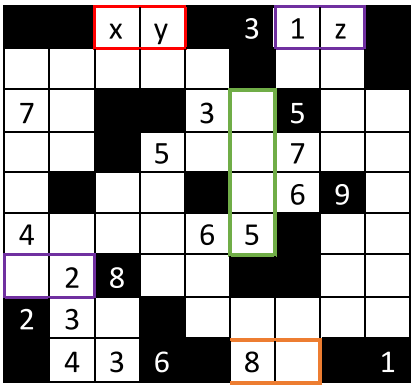
The first task is to write the function **PopulateBoard(vector<string>skeletonBoard)** in **SetSolver.h class.** The function takes a vector of strings and translates/converts it to a vector of integers based on the following rules.

1. A zero remains a zero (denotes an empty black square)
2. A string of "-x" (negative number) remains a negative number as an int (denotes a black square with a number)
3. An asterisk is converted and stored as the number 99. (denotes an empty white square, which may be any value from 1 through to 9).
4. **SetSolver.h** should also include a function **ReturnValue (int row, int col)**, which returns the value for a particular cell. It should return -x for a negative number (denoting a black square with a number), x for a white square with a number, a zero for an empty black square and 99 for an empty white square.

Task 2 [2 marks]:

For this task, you need to write the function **RemovePosibilities()** within the **SetSolver.h** file. This function removes numbers that appear in Black and white squares from other white squares in their rows and columns. In the example below, square x cannot contain 3, 1 or 8. Square y cannot contain 3, 1, 5 and 6 and square z cannot contain 3, 1 or 9 (for reasons explained above).

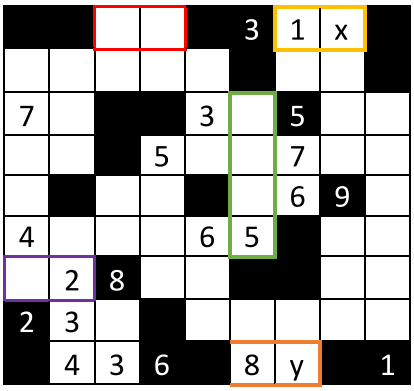
To help yourself and me check that **RemovePosibilities()** is working, you'll have to implement another function **vector<int> ReturnPosibilities(int row, int col)**. This function will return a vector of int/s that contains all possible number for a given square sorted in ascending order (smallest number at the beginning of the vector).



Task 3 [2 marks]:

For this task you'll have to implement the function **NearlyThere(int row, int column, string 'horizontal'/'vertical')**, which points to the beginning of a compartment. If **horizontal**, it will point to the left most square in the compartment and if **vertical** it will point to the top square in the compartment. Once called, the function will remove all **impossible** numbers from white squares in that compartment, based on existing numbers within that compartment. For example, the white square labelled 'x' (see below) in the yellow compartment (top row) should not contain any number apart from 2, as that's the only logical conclusion in that space (next to 1). On the other hand, the square marked 'y' in orange compartment (last row) may contain 9 or 7 as its next to 8 and there is only one space. That said, since 7 already appears in that column, 9 is the only conclusion. Another example is purple compartment in row 7. The first square (from left) can only contain the numbers 1 or 3 as it is next to a 2. Again, as in task 2, **vector<int> ReturnPosibilities(int row, int col)** will be used to check the values stored for each empty white square after **NearlyThere()** was called. The values returned by **ReturnPosibilities** should be sorted ascending order.

Before testing this function, I will first call **PopulateBoard()** and then **RemovePosibilities()** before finally calling **NearlyThere()** to ensure there is no knock on effect on any other compartment.



Task 4 [4 marks]:

For this task you'll have to implement the function Solve(), which will solve the board in its entirety. To aid this process the following additional example of a board is provided along with its solution.

Table 3- Board 2

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

Table 4 - Board 2 with Solutions

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