DATE: GRADE:

COMPUTER PROGRAMMING 2 [PROJECT 1]

TECHNICAL DOCUMENTATION

TITLE: Sudoku solver

TEACHER: DANIEL KRASNOKUCKI

STUDENT: WOJCIECH DRZEWIECKI

1. TASK DESCRIPTION

Sudoku is a puzzle in which we have to fill a grid (usually 9x9) with numbers from 1 to 9 taking into account that each number can be written only once in each column, row and square. Sudoku solver is a program that does it algorithmically to any provided valid Sudoku diagram.

2. EXTERNAL SPECIFICATION

Program works like this: When you provide correct path to file, f.e. sudoku.txt after switcher '-i' it prints out solved Sudoku or proper error message in output file, which you provide like input file, but after switcher '-o'. Each character in input file should represent corresponding cell in diagram. To properly indicate value of cell put there a digit. If cell has no value put there zero or any character that is not a digit. Do not separate cells with anything. To move to next row of diagram use line feed.

Example of correct call of the program:

C:\sudokusolver\sudokusolver.exe –i C:\sudokusolver\sudoku.txt –o C:\sudokusolver\output.txt

If path has any spaces inside, place it between quotation marks.

3. INTERNAL SPECIFICATION

Functions

```
vector<vector<int>> TakeGridFromFile(ifstream &input);
```

Convert input file into grid of numbers. Throws exceptions from CheckGridCorrectness function in case when file doesn't meet requirements of file convertible to Sudoku diagram.

```
void CheckGridCorrectness(vector<vector<int>> grid);
```

Throws three possible exceptions:

- Size of sudoku isn't a perfect square.
- Grid isn't a square.
- Value exceeds size of grid.

```
void Transpose(vector<vector<int>>& grid);
```

Transpose grid. Works only with square matrix.

SudokuDiagram class

```
SudokuDiagram(vector<vector<int>> grid);
```

Constructor of diagram based on grid provided as a parameter.

```
void WriteToFile(ofstream& output);
```

Print out current state of sudoku diagram in output stream.

Solver class

```
Solver(SudokuGrid *&sudokugrid);
```

Constructor of solver class with sudokugrid to solve.

```
bool SolveSudoku();
```

Solve sudoku. Return true if sudoku is valid, false otherwise.

```
bool LookForNakedSubset(Cell * cell, vector<Cell*>& emptycells);
```

Check if cell has any naked subsets of cells inside emptycells container. Return true if candidate of any cell changes its value.

```
bool LookForPointingSubset(Cell * pcell);
```

Check if pcell has any pointing subset in the grid. Return true if candidate of any cell changes its value.

```
bool LookForUniqueCandidate(Cell * cell);
```

Check if cell is a unique candidate anywhere in the grid. Return true if candidate of any cell changes its value.

```
bool LookForXYWing(Cell * cell);
```

Check if cell is a XY to XY wing anywhere in the grid. Return true if candidate of any cell changes its value.

```
void SetCellsQueue();
```

Set queue of cells based on ones number of candidates.

Sudoku grid class

```
SudokuGrid(vector<vector<int>> grid);
```

Constructor of grid class based on grid.

```
bool FillCell(Cell * certaincell);
```

Set value to certaincell based on its candidates. Candidates count has to be 1 for this cell. Update candidates of cells related to this in the grid.

```
vector<tuple<Cell*, int, int>> EmptyCandidateLinkedCells(Cell* cell);
```

Returns vector of cells linked to cell that share only one candidate with cell. First int in tuple is common candidate for element and cell, second is different candidate.

General draft of how the program work

First, program reads input file character by character and transforms it into matrix of numbers. If character is a digit program pushes it into matrix, if anything else program pushes it as '0'. Then, it checks it's correctness – matrix has to be square, its size have to be a perfect square and values cannot exceed its size.

Once input file passes validation, program can start processing it. First it rewrites it as grid of cells – class representing little window in diagram. Then it can insert candidates into each cell. It takes values of column, row and square for each cell and if some value between one and size of Sudoku is not included in any of the block – voilà. It is now a candidate of this cell. Candidates for each cell are vector of bools – if it is a candidate, it is true. I chose it over vector of integers for sake of simplicity – it's much easier to process and compare it than it is with integers.

Once each cell has initial candidates set up, we can begin to solve the Sudoku. First thing we want to do is to create a priority queue of empty cells of a diagram based on it's number of candidates – if it's equal to one we can fill this cell without any doubt. Once we fill any cell we have to delete value that this cell has possessed from candidates of linked cells to this cell. Sudoku grid class allows us to easily iterate through cells of our interest – it contains many methods that specify what cells you are looking for.

Until we drain cells queue – we solve the Sudoku – or we find out that some cell has candidates count of 0 – Sudoku is invalid – we have to decrease number of candidates of cells. To do that we use four logical solving methods – Unique candidate, naked subsets, pointing subsets and XY wing. They allow us to decrease number of candidates of some cells by logic. When we get stuck – logical methods doesn't affect any cells we have to use brute force – recursive method of trying out every candidate of cell to see if diagram is valid if we put it in there.

4. TESTING

First, lets check if we can insert any invalid input file. Input1.txt is some made up Sudoku of size 6 – how do you want to divide it into squares? As we can see, output1.txt printed out appropriate error message. Now, let's try to skip one row of Sudoku in file input2.txt. As we can see, output2.txt printed out appropriate error message. Now, lets take valid 4x4 Sudoku and add 5 to every value in input3.txt, so values inside are greater than Sudoku size. Although diagram is "valid", program doesn't know that it should operate on values from 5 to 9. Why should it?

Second, we can insert invalid Sudoku. Input4.txt contains unsolvable Sudoku, and output4.txt contains proper error message.

Next, we can see if program can solve Sudoku with many solutions – it should print the first one it finds. Input5.txt contains empty 25x25 grid. After a while it prints solved Sudoku in output5.txt. It may not look nice, but it shows that once we insert proper Sudoku, this program works every time.

Finally, we can insert regular Sudoku and see what happens. Input6.txt contains "worlds hardest Sudoku". Output6.txt is exactly what we would have expected – it is solved.