BİL 102 – Computer Programming
HW 06

Last Submission Date: April 22, 2013 – 13:59

Part1. (50 Pts) In this part you will work on 9x9 standard Sudoku puzzles, in which normal entries are represented by numbers 1-9 and unknown entries are represented by 0. Elements of Sudoku will be stored as integers delaminated by a space character horizontally and a newline character vertically in text files. In arrays, first index will represent the row, and the second one will represent the column of the Sudoku. Because the size of the array holding a Sudoku is clear in this part, the function prototypes does not include this size (note that elementsCanBePlaced() returns the size of partially filled array validElements).
Implement the functions below:

- **int readSudoku(FILE* inFile, int sudoku[][9])**: reads the Sudoku from a text file; returns 0 normally and 1 in the case of an error (invalid number, wrong size).
- **int writeSudoku(FILE* outFile, const int sudoku[][9]):** writes the Sudoku to a text file. Unknown elements will be written as 0 (as they exist in the array). Returns the number of known elements written on success and -1 on failure (check what fprintf returns).
- **int printSudoku(const int sudoku[][9]):** prints the content of the Sudoku to the console. Unknown elements will be written as spaces without causing any shift. Returns the number of known elements written on success and -1 on failure (check what printf returns).
- **int checkCanBePlaced(const int sudoku[][9], int posY, int posX, int value)**: checks if the value can be placed in the given location in the Sudoku. (1: can be placed, 0: cannot be placed)
- **int checkIfConsistent(const int sudoku[][9])**: checks if the given Sudoku is consistent (if the known entries violate the rules of the game). Returns 1 in the case of consistency and 0 o/w. (for each element, use checkCanBePlaced() to understand if the element can really be placed in its position)
- **int elementsCanBePlaced(const int sudoku[][9], int validElements[], int posY, int posX)**: fills validElements array with all possible numbers that can be placed in the given position and returns the size of the array as the return value.
- **Int solveSudoku(int sudoku[][9])**: Solves the Sudoku by replacing unknown elements with some acceptable ones if possible and if the puzzle is not too complex. Returns 0 on success, -1 on error (impossible to solve or inconsistent), -2 if the puzzle is too complex. Implement the algorithm below:

while there exist unknown elements
*if there exists an unknown element which **cannot have any** acceptable value (use elementsCanBePlaced()), return **-1***
*if all unknown elements can have **more than one** acceptable value, return **-2** /* too complex to solve*/*
let "e" be an element having only one acceptable value
change the status of "e" as known by replacing its acceptable value with the unknown code 0
recalculate number of acceptable values for all unknown elements

return 0

In main(), test all the functions you implemented reading the Sudoku from a text file "**SudokuIn.txt**" and writing the solution to "**SudokuOut.txt**".

Part2. (60 Pts) In this part you will make a simulation of heat transfer on a metal plate. We will conceptually divide the plate into rectangular pieces and assume that:

- the temperature is the same everywhere inside a piece,
- the plate is heated up and cooled down from some parts of it and these parts will have some constant temperatures (does not change in time). The temperature of other parts will change in time according to the given update rule,
- simulation time is discrete,(i.e. time = 0, 1, 2, …, n; temperatures are calculated for only these integer time values)

  update rule: at a simulation time **t+1**, the temperature of a non-constant element a_{ij}, is the mean of the temperatures of the following maximum 9 elements **at time t**:
  - itself
  - all elements surrounding it (8 elements unless a_{ij} is at an edge)

The problem will be given using 2 text files representing 2 tables:

- InitialTemperatures.txt: holds initial temperatures of the plate
- Constants.txt: indicates constant and non-constant elements. 0 indicates a non-constant element, 1 indicates a constant element.

For representing the data in a file as 2D, we need to know the number of columns. Also, for calculating the total amount of data, we need to know the numbers of rows. Therefore, in each text files the first and the second values are integers indicating the **number of rows**, and the **number of columns**, respectively.
Because we do not want to specify a constant max column number and limit our implementation, we will represent both tables in 1D arrays and access the data on them using a function to convert 2D index to 1D index.
Implement the following functions:

- **int readTemperature(FILE* inFile, double temp[], int capacity, int* rowCount, int* columnCount)**: reads the temperature data into a 1D array "temp". Returns the number of temperatures it reads normally and -1 if the capacity of the input array is insufficient as the return value. This function should avoid all whitespace characters between temperature values (as fscanf does when reading as double) and therefore should be able to read a file writeTemperature produces correctly.
- **int readConstants(FILE* inFile, int consts[], int capacity, int* rowCount, int* columnCount)**: reads the constants data into a 1D array "consts". Returns the number of constants it reads normally and -1 if the capacity of the input array is insufficient as the return value. This function should avoid all whitespace characters between integer values.
- **int writeTemperature(FILE* outFile, const double temp[], int rowCount, int colCount)**: writes the temperature values to a file such that for each value there will be space for 2 digits before the decimal point and 2 digits after the decimal point and each row of the table will be in a separate line in the file.
- **int printTemperature(const double temp[], int rowCount, int colCount)**: prints the temperature values to the console such that for each value there will be space for 2 digits before the decimal point and 2 digits after the decimal point and each row of the table will be in a separate line. Use the following fact to avoid code replication: "stdout" is a file pointer showing standard output, therefore

"fprintf(stdout, "10\n");" will printf "10" to console unless standard output is redirected.

- **int getIndex(int colCount, int posY, int posX)**: returns the index of 1D array representing a table
- **int getNextTimeData(const double currentTemp[], double nextTemp[], const int constants[], int rowCount, int colCount)**: takes current temperature data and returns one-step-ahead temperature data
- **double getMSEDifference(const double table1[], const double table2[], int rowCount, int colCount)**: returns the difference between the **non-constant** elements of 2 tables by means of mse.
- **int simulate(const double initialTemps[], const int constants[], double resultTemps[], int rowCount, int colCount, double maxMse, int maxIteration)**: simulates the given system until:
  - the difference between the temperature tables belonging to the last 2 simulation times by means of mse is less than maxMse (a steady state is reached, returns number of iterations as return value)
  - maxIteration is reached (returns -1 as return value).

Example:
Assume that the content of "InitialTemperatures.txt" is:
3 2
50.00 10.00
50.00 10.00
10.00 0.00

and the content of "Constants.txt" is:
3 2
1 0
1 0
0 1

These files represent the following temperature distribution at simulation time = 0, where the shaded area represents the constant temperature area.

50	10
50	10
10	0

In this example: "readTemperature(fPtr, temp, 1000, &rowC, &colC);" should make the following assignments:

temp: {50, 10, 50, 10, 10, 0}
rowC: 3, colC: 2

after this line "a=temp[getIndex(colC, 2, 1)]" should assign a the value "0.0"

"temp2[getIndex(colC, 2, 0)] = 5;" should change the value at the index (2,0) of a table temp2, which has 2 columns as 5.

Then, calling getNextTimeData(temp, tempNew, constants, rowC, colC) should result tempNew to represent the following table:

50	30
50	21.67
17.5	0

Then, “getMSEDifference(temp, tempNew, rowC, colC)” should return $\frac{1}{3}(400 + 100 + 56.25)$

PS:

We sometimes use scripts we write to test your implementations. These scripts should be able to add our test code into your main function and the test code should be able to call the functions you implemented. Therefore, starting from this homework, you should obey the following rules strictly:

- You should put the 3 tags as defined here in your main function.
- You should use the function prototypes and file names **exactly** (case sensitively) the same as they are defined in the homework text (copy & paste is recommended).

General:

1. Obey honor code principles.
2. Obey coding convention.
3. Your submission should include the following file only
   HW06_<student_name>_<studentSirname>_<student number>.c
4. Deliver the printout of your code **until the last submission date**.
5. Do not use non-English characters in any part of your homework (in body, **file name**, etc.).