CS 211: Computer Architecture, Summer 2021
Programming Assignment 2: Advanced C (100 points)

Instructor: David Pham

Due: July 21, 2021 at 11:55pm.

Introduction

The goal of this assignment is to help improve your C programming. You will write a total of 3 programs. Each part of the assignment is detailed below. The programs have to run on iLab machines.

We will not give you improperly formatted files. You can assume all your input files will be in proper format as described.

No cheating or copying will be tolerated in this class. Your assignments will be automatically checked with plagiarism detection tools that are pretty powerful. Hence, you should not look at your friend's code. See CS department's academic integrity policy at:
`https://www.cs.rutgers.edu/academic-integrity/introduction`

First: Binary Addition (10 Points)

You have to write a program that will compute the binary sum of two addends. You will receive as input, the amount of input bits, the amount of output bits, and two addends. The program will truncate both of the addends to the amount of bits specified by the amount of input bits. The program will then add them together and output the sum as output. The sum will be truncated to the amount of bits specified by the amount of output bits.

Input format: This program takes four numbers as arguments from the command line. The first number will be the number of input bits. The second number will be the number of output bits. The third number will be the first addend. The fourth number will be the foruth addend. All numbers willl only be positive values. Binary representations will thus only represent magnitude.

Output format: Your program should output the sum of the two truncated addends. The sum should be truncated to the amout of bits specified by the amount of ouput bits. In the case where the amount of output bits are not enough to represent the proper sum of the two truncated addends, then your program should output "OVERFLOW".

Example Execution:

Some sample runs and results are:

```
$./first 4 2 1 1
10

$./first 5 5 5 5
01010

$./first 4 1 1 1
OVERFLOW
```

Second: Sudoku (60 Points)

In Part 2 of this assignment, you will write a C program that implements a simple Sudoku solver (in the third part you will implement a complex Sudoku solver). Sudoku is a simple logic puzzle where the objective is to fill a 9x9 grid of cells with each cell containing a number from the set 1-9 while fulfilling the following constraints:

- Each number is unique in its row and column.
- Each number is unique in its subgrid where a subgrid is defined by cutting the 9x9 into 9 nonoverlapping 3x3 grids (top left, top, top right, left, middle, right, bottom left, bottom, bottom right).
- Each row, column, and subgrid have all numbers from 1-9 present.

A Sudoku is defined as solved when all of its cells in the 9x9 grid are filled with numbers with each cell satisfying the constraints above. A Sudoku is defined as unsolvable when there is no configuration where all the cells can be filled without breaking the constraints. A Sudoku will start partially completed with some numbers placed in the 9x9 grid. This will allow the solver to determine where to place new numbers in order to find the solution for the given Sudoku. For this part, we will only be looking at Sudoku grids with one unique solution.

In this part, we will look at Sudoku grids wehre there will always be a cell that you can fill with 100% certanity with a unique value according to the constraints described above.

Third: Complex Sudoku (30 points)

It is not always possible to fill at least one element with 100% certainity in each step for many Sudoku grids. Sometimes the next step to be taken cannot be known and instead, a guess must be taken in order to move forward, such as perhaps filling in a cell randomly. In this section, you will implement an algorithm that solves these more complex Sudoku grids. A possible algorithm to do so would be to solve as much of the Sudoku as can be done so confidently. Once no more cells can be filled with 100% certainty, continue by taking the first unsolved cell (let's call this cell A) and filling it with a number that satisfies the constraints and continue solving as usual from there. If a solution is found, then you are done. Otherwise, if no moves can be taken and the board has not yet been filled than you must backtrack to the state where you guessed the value of cell A and either guess a new value for cell A or perhaps choose a new cell altogether. This is a simple backtracking algorithm. There are many algorithms to solve these complex Sudoku and it is entirely up to you

to decide which to implement. No points will be given based on good efficiency but poor effiency that times out on the autograder will not be given points

Your submission should include a Readme detailing in 3-6 sentences what you did to solve the Complex Sudoku problem. This should detail your algorithm and what changes you had to make to your initial approach in Second. You must include this Readme in your submission in order to receive full credit for this portion of the assignment.

Input Format for Part 2 and Part 3

Your program will read in the data from a file given to the program as an argument from the command line. The file format will be a 9x9 grid with each number in the same row separated by a tab. Blank cells will be represented with a dash symbol: '-'. An example Sudoku grid test1.text is given below:

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

A sample Sudoku grid test2.txt is given below:

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

You can assume that all inputs will be valid meaning the spacing will be correct and only the '-' character and numbers 1-9 will be present in the test files. The preset numbers in the test may not satisfy the constraints of Sudoku as in the example test2.txt above resulting in an unsolvable Sudoku.

Output format

For a Sudoku grid that has a solution, you should print out the 9x9 grid with each number in the same row separated by a tab and with each row on a new line. There should be no trailing spaces or tabs on any line of output. A sample execution is given below:

```
./second test1.txt
3   1   4   5   8   2   6   7   9
7   8   5   6   9   3   2   4   1
9   2   6   1   7   4   3   8   5
5   6   8   3   2   1   7   9   4
1   3   7   4   6   9   5   2   8
2   4   9   7   5   8   1   6   3
4   7   2   9   3   5   8   1   6
6   5   1   8   4   7   9   3   2
8   9   3   2   1   6   4   5   7
```

For a Sudoku grid that has no solution, your program should print "no-solution" and nothing else. For part 2, Sudoku grids that have a solution but cannot be solved without guesses are considered unsolvable.

Structure of your submission folder

All files must be included in the `pa2` folder. The `pa2` directory in your tar file must contain 3 subdirectories, one for each each of the parts. The name of the directories should be named first through third (in lower case). Each directory should contain a c source file, a header file (if you use it) and a Makefile. For example, the subdirectory first will contain, first.c, first.h (if you create one) and Makefile (the names are case sensitive). Third should contain a Readme detailing your adaptations from second to third.

```
pa2
|- first
   |-- first.c
   |-- first.h (if used)
   |-- Makefile
|- second
   |-- second.c
   |-- second.h (if used)
   |-- Makefile
|- third
   |-- third.c
   |-- third.h (if used)
   |-- Makefile
   |--Readme
```

Submission

You have to e-submit the assignment using Sakai. Your submission should be a tar file named `pa2.tar`. To create this file, put everything that you are submitting into a directory (folder) named `pa2`. Then, `cd` into the directory containing `pa2` (that is, `pa2`'s parent directory) and run the following command:

```
tar cvf pa2.tar pa2
```

To check that you have correctly created the tar file, you should copy it (`pa2.tar`) into an empty directory and run the following command:

```
tar xvf pa2.tar
```

This should create a directory named `pa2` in the (previously) empty directory.

The `pa2` directory in your tar file must contain 3 subdirectories, one each for each of the parts. The name of the directories should be named first through third (in lower case). Each directory should contain a c source file, a header file (if necessary) and a make file. For example, the subdirectory first will contain, first.c, first.h and Makefile (the names are case sensitive).

AutoGrader

We provide the AutoGrader to test your assignment. AutoGrader is provided as autograder.tar. Executing the following command will create the autograder folder.

```
$tar xvf autograder.tar
```

There are two modes available for testing your assignment with the AutoGrader.

First mode

Testing when you are writing code with a `pa2` folder

(1) Lets say you have a `pa2` folder with the directory structure as described in the assignment.

(2) Copy the folder to the directory of the autograder

(3) Run the autograder with the following command

$python auto_grader.py

It will run your programs and print your scores.

Second mode

This mode is to test your final submission (i.e, pa2.tar)

(1) Copy pa2.tar to the auto_grader directory

(2) Run the auto_grader with pa2.tar as the argument.

The command line is

$python auto_grader.py pa2.tar

Scoring

The autograder will print out information about the compilation and the testing process. At the end, if your assignment is completely correct, the score will something similar to what is given below.

```
You scored
30.0  in  second
15.0  in  third
5.0  in  first
Your TOTAL SCORE =  50.0 /50
Your assignment will be graded for another 50 points with test cases not given to you
```

Grading Guidelines

This is a large class so that necessarily the most significant part of your grade will be based on programmatic checking of your program. That is, we will build the binary using the Makefile and source code that you submitted, and then test the binary for correct functionality against a set of inputs. Thus:

- **You should not see or use your friend's code either partially or fully. We will run state of the art plagiarism detectors. We will report everything caught by the tool to Office of Student Conduct**.
- You should make sure that we can build your program by just running `make`.
- You should test your code as thoroughly as you can. For example, programs should *not* crash with memory errors.
- Your program should produce the output following the example format shown in previous sections. Any variation in the output format can result **in up to 100% penalty**. Be especially careful to not add extra whitespace or newlines. That means you will probably not get any credit if you forgot to comment out some debugging message.

Be careful to follow all instructions. If something doesn't seem right, ask on the discussion forum.