Project: Solving Sudoku Via Satisfiability

CS 201; Summer 2012

Due: Monday July 16 by 11:29am

Sudoku Rules

The rules of the game are probably familar to most of you. If you do not know the rules, take a look at the Sudoku Wikipedia entry; you can also get many sample puzzles online.

In brief, given a 9×9 board with some squares pre-assigned values in the range 1..9, the player must assign numbers (again in the range 1..9) to each of the remaining squares while obeying the constraints that:

- each row contains every number in 1..9
- each column contains every number in 1..9 and
- each of 9 non-overlapping 3×3 squares also contain every number in 1..9.

Checklist

1. A checker program, `CheckSK` which reads a completed board and determines if it obeys the constraints. If it does, it simply says so; if it doesn't it reports the conflict (e.g., by pointing out that a particulr row does not have a particular value in it).

   `CheckSK` takes a single input file as a command-line parameter.

   This will account for 25% of your score.

2. The solver program `SolveSK` which reads a partially filled board and, by using a Satisfiability solver finds a legal assignment to the empty squares (if possible). If such an assignement is impossible, the program says so; otherwise, it prints the original board and a solution to the terminal. I recommend that you format your output to be more human readable than the input file format specified below (a "pretty" formatting of a board also appears below).

   `SolveSK` takes a single input file as a command line parameter.

   This will account for 75% of your score.

3. Bonus: by extending both of these programs to handle the `16 x 16` and `25 x 25` generalized versions of Sudoku, you can receive up to 25 bonus points.

   The programs will still be named the same; the input files wil simply indicate the board dimension on the first line. Details below.

Input Format

Input files will have the *"dimension"* of the board on the first line as an integer – since the number or rows/columns must be a perfect square, we express the size of the board using the square root of this number. Thus the traditional `9x9` board has dimension 3. Unless you are doing the bonus, you can simply skip this line and assume a `9x9` board follows.

The remaining lines in the file contain each row in sequence; we use a hyphen ("-") to indicate an empty square. Individual characters are separated by white space.

An example input file (an example board from the Wikipedia page):

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

Error handling: Your program should be able to detect the following error conditions:

- too many/too few symbols (columns) in a row (line)
- too many/too few rows
- invalid symbol

That's it! If there is for example a blank line between rows, you just say there aren't enough symbols on that row.

Output

As mentioned, it is recommended that you make the output to the terminal somewhat easier for a human to read than the input format above. For example, the output of `SolveSK` on the above input might be something like:

```
Input board:

 5 3 - | - 7 - | - - -
 6 - - | 1 9 5 | - - -
 - 9 8 | - - - | - 6 -
 ======+=======+======
 8 - - | - 6 - | - - 3
 4 - - | 8 - 3 | - - 1
 7 - - | - 2 - | - - 6
 ======+=======+======
 - 6 - | - - - | 2 8 -
 - - - | 4 1 9 | - - 5
 - - - | - 8 - | - 7 9
```

```
Solution:

  5 3 4 | 6 7 8 | 9 1 2
  6 7 2 | 1 9 5 | 3 4 8
  1 9 8 | 3 4 2 | 5 6 7
  ======+=======+======
  8 5 9 | 7 6 1 | 4 2 3
  4 2 6 | 8 5 3 | 7 9 1
  7 1 3 | 9 2 4 | 8 5 6
  ======+=======+======
  9 6 1 | 5 3 7 | 2 8 4
  2 8 7 | 4 1 9 | 6 3 5
  3 4 5 | 2 8 6 | 1 7 9
```

Resources

- Pages 32-33 of Rosen (7th Edition) present a way to encode a Sudoku puzzle as propositional formula in *Conjunctive Normal Form* (CNF). This formula is satisfiable if and only if the puzzle is solvable.

  (Scans of this section of Rosen have been included in the archive for this project).

  Read (and re-read) this section until you get what is going on. It is probably easier to understand the formula if you break it down; for example, the constraint that a particular row has a 1 in it is a single clause; for that same row you generate additional clauses for 2, 3, ..., 9.

  Note: The encoding in Rosen is not the *only* CNF encoding possible for Sudoku. If you come up with another encoding, feel free to run it by me.

  Scans of these pages will be posted to Blackboard for those with the 6th edition.

- Two short tutorials on getting `sat4j` working on your own machine and on the CS lab machines will be posted to Piazza.

  In addition, a simple program `TestSAT` will be posted which illustrates how to use `sat4j` to create a formula (essentially how to add clauses to a formula) and how to invoke the solver and ultimately extract a satisfying truth assignment (if the formula is indeed satisfiable).

  Pay particular attention to the conventions used for representing literals in CNF clauses.

  Also, the `sat4j` package is rather extensive, but for our purposes, only a few methods will really be needed.

Bonus

For the `16x16` and `25x25` puzzles, we will not assign integers to the table cells. Instead, in the `16x16` case we will use letters a-p:

```
              a b c d e f g h i j k l m n o p
```

In the `25x25` case we'll use the letters a-y:

```
a b c d e f g h i j k l m n o p q r s t u v w x y
```

This way, we can still use single characters for each cell. It might be handy to know that you can *cast* characters to integers and that consecutive characters in the alphabet cast to consecutive integers – in particular, lower case `a-z` correspond to integers 97-122.

Try running this program and you'll see what I mean.

```
public class Big {

  public static void main(String[] args) {
    int a = (int)'a';  // casting char to int
    int z = (int)'z';

    System.out.println("a: " + a);
    System.out.println("z: " + z);

    for(int i=97; i<123; i++)
        System.out.println((char)i);
  }
}
```

Submission

You will submit `CheckSK.java`, `SolveSK.java` and any other java files you write. In addition, submit a text file `readme` which states whether or not you did the bonus and if so, how complete it is.