Lab 4: Super Sudoku Solver
CSCI 2101 – Fall 2021

Due (Section A): Monday, October 25, 11:59 pm
Due (Section B): Tuesday, October 26, 11:59 pm
Collaboration Policy: Level 1 (review full policy for details)
Group Policy: Pair-optional (you may work in a group of 2 if you wish)

In this week's lab, you will write a program that can solve any instance of the popular number-placement puzzle *Sudoku*. Doing so will give you experience working with linear structures (*stacks* in particular), and show you how such structures can be applied to solve an interesting problem. You will also gain experience working with two-dimensional arrays in Java.

1 Sudoku

Sudoku is a puzzle that existed in some form since the 19th century, but has been popularized worldwide (starting in Japan) in the last few decades. A Sudoku puzzle consists of a 9x9 grid of cells, further subdivided into nine smaller 3x3 grids (or "boxes"). In the starting configuration of the puzzle, a subset of the cells are filled with numbers from 1 to 9, while the rest are left blank.

The objective of the puzzle is to fill in every blank cell with a number from 1 to 9 such that every digit from 1 to 9 is present in every *row*, every *column*, and every 3x3 *box*. Another way of thinking of this is that there are no repeated digits in any single row, column, or 3x3 box. An example of a Sudoku puzzle is shown below, along with its solution.

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

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

Figure 1: An example Sudoku puzzle (left) and its solution (right).

Sudoku's increasing popularity has resulted in many competitions between human players, who typically vie to complete series of multiple puzzles in the least total time. However, unlike many competitive games played by humans, Sudoku is a puzzle that can be solved fairly easily by a computer. In this lab, you will demonstrate this fact by writing a program that can solve any Sudoku puzzle nearly instantly. A natural way to approach this problem is using a type of linear structure known as a *stack*, which we discussed in class.

2 Solving Sudoku: Backtracking

The basic design of an automated Sudoku solver is fairly simple. We're essentially going to solve the puzzle by considering one empty cell at a time and filling in any valid digit for that cell that doesn't violate any of the constraints of the final puzzle (i.e., duplicating a digit in the row, column, or box). We then proceed to the next empty cell and do the same thing.

If we're able to choose a valid digit for every empty cell of the puzzle without violating any constraints, then we've solved the puzzle and are done. Of course, it's unlikely that we just happened to pick every digit correctly on the first try. If we happen to make a choice that is *not* the correct digit for that cell, the result will be that we'll eventually (possibly multiple choices later) encounter an empty cell where we *can't* place any digit without violating one of the puzzle constraints.

Once we encounter such an situation (i.e., placement is impossible), we know that at least one previous digit placement was wrong. To recover, we're going to **backtrack** – that is, we're going to walk backwards through the previously empty cells we filled in, undoing moves that we previously made (i.e., returning cells to empty). We continue backtracking until we reach a move where we can instead place a *different* (but still valid) digit into that cell. We then proceed forward filling in empty cells as usual, until the next time we cannot make a placement and have to backtrack again.

Here's a simple example: suppose we're considering a particular empty cell c_1, which can be filled with either 4 or 7 given the current puzzle configuration. We pick 4 (arbitrarily), then move onto the next empty cell c_2. This cell can only be filled with 1, so we fill with the value 1, then move to empty cell c_3. We then find that there is no possible valid digit for c_3, so we're forced to backtrack to c_2. However, c_2 has no valid digit other than the one we previously picked, so we undo that move (clearing c_2) and backtrack again to c_1. Now considering c_1 again, we know that picking 4 was a bad move, so we pick 7 instead, and then proceed forward again onto c_2.

One challenge you need to deal with is the possibility of trying digits already attempted previously (e.g., maybe you already tried 4 and 9 for a given cell but haven't tried 2 yet – just trying 4 or 9 again isn't going to get you anywhere). The simplest way to approach this problem is to try digits starting with the smallest – e.g., if 2, 5, or 7 all work for empty cell c, initially choose 2. The first time you backtrack to c, choose 5 (i.e., a value larger than the one previously chosen). The next time you backtrack to c, choose 7. If you have to backtrack to c again, then you're out of new values to try, and therefore need to backtrack another step.

Note that backtracking is essentially a brute-force algorithm – we're effectively just trying different digit placements repeatedly (potentially every possible placement) until we find one that works. Since a regular Sudoku puzzle is fairly small, this algorithm works just fine on a typical modern computer. However, for larger grid sizes, backtracking would quickly become intractably slow. For larger puzzles, other Sudoku solving algorithms exist that are significantly faster (albeit more complex) than backtracking.

3 Program Overview

Implementing your Sudoku solver will involve a few distinct parts:

- Writing a class to represent a Sudoku puzzle (either an unsolved or solved puzzle).
- Writing a class to solve a Sudoku puzzle using backtracking.
- Writing test code to read a puzzle, solve it, and optionally verify it against a known solution.

You will be provided with a few sample Sudoku puzzles as well as their solutions in order to test your program. The operation of your final program will be quite simple. The program should first prompt the user to enter a file containing an unsolved Sudoku puzzle, as well as a file containing the solution to that puzzle (the latter may be omitted by just hitting enter). The program should then print out the unsolved puzzle, solve it, and print out the solved puzzle. If a solution file was entered, the program should finally verify that the found solution is correct by comparing it to the known-correct solution and printing out a message indicating whether verification succeeded. Here is an example where no puzzle solution is entered:

```
Enter filename of puzzle: puzzles/s2.txt
Enter filename of solution (optional):

Starting puzzle:
5 _ _ _ _ 9 7 6 _
_ _ 4 _ 8 _ _ 1 _
_ _ 2 6 _ _ _ 9 _
_ _ _ _ _ 8 _ _ _
6 _ 9 2 _ 5 4 _ 3
_ _ _ 4 _ _ _ _ _
_ 1 _ _ _ 2 6 _ _
_ 9 _ _ 4 _ 5 _ _
_ 5 6 8 _ _ _ _ 9

Solved puzzle:
5 3 8 1 2 9 7 6 4
9 6 4 5 8 7 3 1 2
1 7 2 6 3 4 8 9 5
3 4 5 7 9 8 1 2 6
6 8 9 2 1 5 4 7 3
7 2 1 4 6 3 9 5 8
8 1 3 9 5 2 6 4 7
2 9 7 3 4 6 5 8 1
4 5 6 8 7 1 2 3 9
```

And here is an example where a solution file is entered (the only difference in the output is that the solution is verified at the end and a message is printed to confirm):

```
Enter filename of puzzle: puzzles/s2.txt
Enter filename of solution (optional): puzzles/s2-solution.txt

Starting puzzle:
[same as above]

Solved puzzle:
[same as above]

Solution is correct!
```

Note that the final verification might fail due to a bug in the solver code, a bug in the verification code, or if the solution file entered didn't actually contain a solution to the given puzzle. In such cases, a failure message should be displayed instead, e.g., "`Solution is NOT correct!`". Your program can assume that input files are well-formed and that all given puzzles are solvable.

4 Class Overview

Your program will consist of four classes:

1. `SudokuPuzzle` – a particular configuration of a Sudoku puzzle (either a starting configuration, a solved puzzle, or an intermediate configuration).
2. `SudokuSolver` – an object that can solve a given `SudokuPuzzle`.
3. `SudokuMove` – an object representing a single digit placement while solving a Sudoku puzzle (e.g., the placement of the digit 8 in row 3, column 5 of the grid), to be used by the solver.
4. `SudokuTest` – a container for your `main` method that reads in the solution and/or puzzle file, uses a solver to solve the puzzle, and produces the program output.

5 Implementation Plan

As usual, you should build your program incrementally. A suggested plan of attack is given below.

- First, implement the skeleton of the `SudokuPuzzle` class. You should first write the constructor, which should be given the name of a puzzle file, and should read the file to initialize the cell contents. Refer to the implementation tips section for advice on storing the puzzle data.
- Next, implement the `toString` method of `SudokuPuzzle` so you can print out puzzle objects. To test puzzle reading and printing, write a short `main` method in `SudokuTest`. For now, you can just hardcode a test puzzle filename and use that to construct a `SudokuPuzzle` object, then print it out. At this point, you should be able to produce a Sudoku puzzle like the starting puzzle shown in the example above.
- Ultimately you'll need to compare your solved puzzle to a known solution to verify that it's correct. In other words, what you'll need to do is to test whether two `SudokuPuzzle` objects are equivalent (one read from the initial puzzle file and then solved by your solver, and the other read directly from the solution file). The proper idiomatic way to test whether objects are equivalent is to define an `equals` method. Refer to the implementation tips section for more on the `equals` method. Once you have a functioning `equals` method, your `main` method (in `SudokuTest`) should be able to create two `SudokuPuzzle` objects (e.g., from the same file, or from two different files) and then compare them using `equals`. Two puzzle objects should be considered equal if they have the same cell contents.
- Plan out other methods that you will need in `SudokuPuzzle` – i.e., what are the capabilities that your puzzle will need to provide while solving it? As a simple example, you will want a method to check whether a particular move is valid in the current grid configuration. While

you do not need to implement every method of `SudokuPuzzle` now, you will be well served by spending a bit of time thinking about the methods you will need (remember that you can always write the method definitions now but defer their actual implementations until later).

Resist any temptation to write a getter method that just returns the internal grid instance variable. Your `SudokuPuzzle` class should provide any public methods needed to interact with the puzzle grid, and should not expose the grid instance variable to arbitrary modification outside the class by just returning it.

- Begin implementing the `SudokuSolver` class. The constructor for this class should just take the `SudokuPuzzle` that you want to solve and store it. The class should have only a single public method (e.g., `solve`) that actually runs the backtracking algorithm to solve the puzzle. However, you may want other private helper methods to assist with the solving method, such as a method that determines the next move that should be made.

  A good way to implement the initial solver is to ignore the possibility of backtracking – i.e., just assume that every time you consider an empty cell, you will be able to find a digit that satisfies it. The first puzzle example (`s1.txt`) can be solved without backtracking, so you can first get a non-backtracking solver working, then implement backtracking afterwards.

  As your `SudokuSolver` is essentially choosing a series of moves to make, at this point you should also implement the `SudokuMove` class. The `SudokuMove` class should be quite simple, and just represents one particular move (i.e., a particular digit placed at a particular row and column of the grid). Whenever your `SudokuSolver` determines the next move to make, it should create a `SudokuMove` object representing the move (which you can then return from the helper method that picks the next move).

  As you write the `SudokuSolver` class, you will need to use the public methods of the `SudokuPuzzle` class, so depending on what you wrote previously, you may need to define additional methods in that class (or you may just need to implement methods that you defined earlier but didn't implement).

- At this point, you should have a functioning (but non-backtracking) `SudokuSolver` class, as well as your complete `SudokuMove` class (used by the solver) and a mostly-complete `SudokuPuzzle` class. Now, change the `main` method in `SudokuTest` to actually solve the puzzle you read in, then re-print the puzzle after solving. Assuming the solver works, you should be able to produce the `s1-solution.txt` puzzle solution. You can verify your solved puzzle against the known solution using the `equals` method that you wrote earlier.

- Returning to your `SudokuSolver` class, now you should actually implement backtracking. Without backtracking, you didn't need to worry about storing moves as you made them, since you never needed to undo previous moves. Now, every mode you make must be saved, since you might have to backtrack and undo them later.

  Note that backtracking is needed when you're unable to find a valid move for some empty cell. If you have a helper method that determines the next move to make as suggested previously, a good way to flag that no move is possible is to return `null` from that method, which signals to the `solve` method that backtracking is needed.

  A stack is perfectly suited to backtracking, since all you're doing is either adding the most recent moves (i.e., *pushing* moves) or removing moves starting from the most recent (i.e.,

popping moves). In other words, your access pattern is exactly Last-In-First-Out (LIFO), which is precisely what a stack provides. See the implementation tips for more information on actually choosing a data structure to store your moves (remember that a stack is just an Abstract Data Type, not a data structure itself).

- Once your backtracking solver is working, you should be able to solve the `s2.txt` puzzle (or any other Sudoku puzzle). The only task remaining (if you didn't do it earlier) is providing the appropriate interface in `SudokuTest` to ask for the two filenames and print out the appropriate output. Remember that providing the solution file should be optional – you can also test on the `s3.txt` puzzle, for which no solution file is provided.

6 Implementation Tips

Below are specific implementation tips on various parts of the program.

6.1 Storing the puzzle grid

Your `SudokuPuzzle` class needs to store the puzzle grid in some way. While you could potentially use a regular array (or list) with 81 spaces to store every digit, it's much easier to think of the puzzle cells in terms of row and column numbers (e.g., the upper-left cell is row 0, column 0, and the lower-left cell is row 8, column 0). The best way to do this is using a *two-dimensional array* – or, equivalently, an array that stores other arrays.

Here's an example of creating a 2D array of integers:

```
int[][] nums = new int[numRows][numCols];
```

Accessing single elements is the same as with a regular (1-dimensional) array, except that both row and column numbers have to be specified:

```
int val = nums[0][5]; // get row 0, column 5
nums[3][2] = 8; // assign row 3, column 2
```

Note the particular types involved in a 2D array such as this. The base array `nums` is of type `int[][]` (i.e., a 2-dimensional array of ints). A specific row in the 2D array, such as `nums[3]`, is of type `int[]` (i.e., a 1-dimensional array of ints). A specific element in a specific row, such as `nums[3][5]`, is of type `int` (just a single number). Many methods working with a 2D array such as your grid will need to loop over the grid elements, which is best done using a nested loop (one loop for rows, one loop for columns).

6.2 Reading puzzle files

Puzzle files simply consist of a sequence of 81 digits separated by spaces and/or newlines. Blank spaces in the puzzles are indicated by zeroes, which is an approach you may wish to reuse in your program. Solution files are formatted in exactly the same way (except they won't contain any zeroes, since they're fully filled in).

Reading the puzzle files is simple using the `nextInt` method of the `Scanner` class, which will simply grab the next digit (skipping over spaces or newlines that come before it). Thus, reading a puzzle file just requires repeatedly calling `nextInt` (but write a loop for this!).

Refer to the handout for Lab 3 if you need a refresher on how to use a `Scanner` to read from a file (but you don't need to go line by line here – you just need `nextInt`).

6.3 Writing an `equals` method

You should write an `equals` method in `SudokuPuzzle` to allow you to check whether two puzzles match (e.g., your solved puzzle and a known solution to the puzzle). Remember that `equals` is another method like `toString` that is defined in the `Object` class and then overridden in a subclass (`SudokuPuzzle` in this case). Since you are overriding an existing method, the method declaration needs to *exactly* match that in the parent, which is as follows:

```
// check whether obj is equivalent to this object
@Override
public boolean equals(Object obj)
```

In particular, note that the `equals` method allows comparing again *any* other object, not just another object of the same type. Usually, we don't consider two objects equivalent if they aren't the same type (e.g., a `String` is never equivalent to a `SudokuPuzzle`), so the first thing the `equals` method normally does is test the type of the `obj` parameter. To do so, you can use the `instanceof` operator to check whether an object is of a given type:

```
if (obj instanceof T) {
    // obj is of type T
} else {
    // obj is not of type T
}
```

If the object is of the correct type, then we usually need to check some parts of their state (e.g., a `SudokuPuzzle` might or might not be equivalent to another `SudokuPuzzle`). Remember that you can cast an object to a desired type, which then allows you to access its methods and instance variables. Though casting is normally dangerous, doing so after first checking an object's type using `instanceof` is safe (since you won't cast anything that's not of that particular type).

Lastly, remember that if you want to compare two arrays, you shouldn't use `==`, which will check if they're the same object, but **not** if they simply have equivalent values. To check if two arrays have the same values, you need to loop over their values to verify each one.

6.4 Checking Sudoku boxes

One of the trickier parts of the `SudokuPuzzle` class is the method that checks whether a particular digit placement is allowed. Checking the relevant row or column to make sure that there are no duplicate digits is simple, but it's harder to check the 3x3 box that the grid position is part of.

A good way to approach this problem is to calculate a row index and column index of the relevant 3x3 box – e.g., the top-right 3x3 box is box row 0, box column 2, and the middle 3x3 box is box row 1, box column 1. Given a particular box row and box column, you can then check that particular 3x3 box to see if the desired digit already appears somewhere in the box.

Calculating the box row and box column from a regular row/column cell position is easy – just divide by 3, since integer division in Java throws away any remainder. For example, the cell position $(1, 8)$ is part of box row $1/3 = 0$ and box column $8/3 = 2$ (the top-right box).

6.5 Stack Classes

You will need a stack implementation to use when storing moves for backtracking. While Java does have a class named `Stack`, this is a historical class that exists for compatibility reasons and should not be used in any new code. Your program should not use the `Stack` class.

Instead, you should use one of the Java classes implementing the standard `Deque` interface (pronounced "deck"), which is short for "double-ended queue" and represents a linear collection that supports object access at either end (either the start or end of the collection). A deque is more general than a stack (since a stack only needs access to one end of the structure) but anything that a stack can do, a deque can equivalently do. Note that a `Deque` is different from a `List` in that a `List` supports object access via specific positions/indices, whereas a `Deque` *only* supports access from the ends (and should achieve O(1) access from either end, which a `List` may not).

The only `Deque` methods you need to use are `push` and `pop`. These methods are equivalent to `addFirst` and `removeFirst`, respectively, but it's a good idea to get used to stack terminology.

Since `Deque` is just an interface (like `List`), you will need an actual class that implements `Deque` to use in your program. The two most appropriate options are `LinkedList` (which implements both the `List` and `Deque` interfaces) and `ArrayDeque` (which implements the `Deque` interface but not the `List` interface). Note that an `ArrayList` implements `List` but not `Deque` (since adding or removing the front of an `ArrayList` is inefficient, which is inappropriate for a `Deque`). You can use either `LinkedList` or `ArrayDeque` as the stack implementation in your program; their asymptotic (Big-O) performance is equivalent for almost all operations, though `ArrayDeque` is empirically faster in most real-world scenarios due to constant factors that are ignored by asymptotic analysis.

Remember to import the appropriate classes in your program – the `Deque`, `LinkedList`, and `ArrayDeque` classes are all part of `java.util`.

7 Evaluation

As usual, your completed program will be graded on correctness, design, and style. Note that your solver will be tested on more puzzles than just the three provided to you. Make sure that your program is documented appropriately and is in compliance with the coding and style guide. Lastly, make sure that your name (and the name of your partner, if applicable) is included in all of your Java files. As usual, you must submit individual group evaluations if working in a pair.

8 Submitting Your Program

Submit your program on Blackboard in the usual way. Remember to create a zip file named with your username(s) and lab number, e.g., `sbowdoin-jbowdoin-lab4.zip`, and upload that file. Only one group member should upload the final submission to Blackboard (but both group members should submit group evaluations via Slack).