The Universal Sudoku Puzzle in the newspaper provides directions for completing a Sudoku puzzle: “Complete the grid so that every row, column, and 3 x 3 box contains every digit from 1 to 9 inclusively.” Here is an example of a completed Sudoku:

258 || 137 || 649

146 || 985 || 327

793 || 246 || 851

=================

472 || 863 || 195

581 || 492 || 736

639 || 571 || 482

=================

315 || 728 || 964

824 || 619 || 573

967 || 354 || 218

Notice that each row and column contains the integers 1 to 9. This is also true of the nine 3 x 3 subsquares.

Mathematicians would say that a Sudoku puzzle is a special example of another mathematical object called a Latin square. Latin squares of size *n* x *n* satisfy the property that every row and every column contain the integers 1, 2, …, *n*. In fact we say that a row or column has the “Latin property” if it contains exactly the integers 1, 2, …, *n*. For Sudoku we can extend the notion to subsquares as well, so if a 3 x 3 subsquare contains 1, 2, …, 9, we say the subsquare satisfies the Latin property. With these definitions, a Sudoku square has the property that every row, column, and 3 x 3 subsquare is Latin.

Here is an invalid Sudoku arrangement:

258 || 137 || 649

146 || 985 || 227

793 || 246 || 851

=================

472 || 863 || 195

581 || 492 || 736

639 || 571 || 482

=================

315 || 728 || 964

824 || 619 || 573

967 || 354 || 218

Notice that in the square above, the second row, the seventh column, and the top-right subsquare do not satisfy the Latin property, while all other rows, columns and subsquares do satisfy the Latin property.

The Java code listed below has a main method that creates and prints three Sudoku puzzles. Each puzzle is created by calling the static method makeSudoku(String s), passing it an 81 character String that contains the contents of a single puzzle. The rows of the puzzle are listed in sequence in the string. The method converts the passed String to a two-dimension array of strings (each String containing a single digit) which is returned to the calling program.

The code below also contains a method called getPrintableSudoku(String[][] x). This method is passed the two-dimensional array representation of the puzzle, and returns a formatted String which can be printed to produce a two-dimensional representation on a page.

You will need to provide the body for the method isValidSudoku(String[][] x). Complete this method so it returns true or false based on whether the puzzle arrangement is a complete and valid Sudoku puzzle.

There are many approaches you can take in tackling this problem. Because the solution involves testing rows, columns, and subsquares, you will probably want to introduce a collection of helper methods to break the problem into simpler subtasks.

At the top level, we need to develop a method isValidSudoku so that it checks that 1) each row is Latin, 2) each column is Latin, and 3) each subsquare contains 1, 2, …, 9.

Let’s decompose isValidSudoku into three Boolean methods called rowsAreLatin, colsAreLatin, and goodSubsquares, which correspond to the tasks listed above. Each of these three methods can be further decomposed as well. For example, rowsAreLatin can be decomposed into a subordinate method rowIsLatin(int i) which examines a single row. (The other methods can be decomposed in a similar way.)

In designing rowIsLatin(String[][] x, int i), it is evident that we need a method of insuring that each symbol 1, 2, …, 9 appears in row i. One technique is to create a boolean array called found with the property that found[k] is true if symbol k occurs in row[i]. We can start by initializing the found array to all false values to indicate that none of the symbols 1, 2, …, 9 have yet be found in the current row.

For each String item in a Sudoku row, you change it to an int by invoking Integer.parseint(String) and use the resulting number as a subscript into the found array, setting the corresponding element true. As an example, if the first item in a row is 6, we need to set found[6] = true. (*Hint:* You will need to code something like found[k] = true where k takes on each integer value represented by the String values in the row.) When you have completed examination of a row, the entire found array should contain only true values if each symbol has occurred exactly once in the row.

To help you get started, we have provided a main method that creates and prints three Sudoku objects. It also contains the needed method headers that we discussed in our approach to decomposing the problem.

public class Sudoku

{

public static void main(String[] args)

{

// Row and column Latin but with invalid subsquares

String config1 = "1234567892345678913456789124567891235678912346"

+ "78912345789123456891234567912345678";

String[][] puzzle1 = makeSudoku(config1);

if (isValidSudoku(puzzle1))

{

System.out.println("This puzzle is valid.");

}

else

{

System.out.println("This puzzle is invalid.");

}

System.out.println(getPrintableSudoku(puzzle1));

System.out.println("--------------------------------------------------");

// Row Latin but column not Latin and with invalid subsquares

String config2 = "12345678912345678912345678912345678912345678"

+ "9123456789123456789123456789123456789";

String[][] puzzle2 = makeSudoku(config2);

if (isValidSudoku(puzzle2))

{

System.out.println("This puzzle is valid.");

}

else

{

System.out.println("This puzzle is invalid.");

}

System.out.println(getPrintableSudoku(puzzle2));

System.out.println("--------------------------------------------------");

// A valid sudoku

String config3 = "25813764914698532779324685147286319558149273663"

+ "9571482315728964824619573967354218";

String[][] puzzle3 = makeSudoku(config3);

if (isValidSudoku(puzzle3))

{

System.out.println("This puzzle is valid.");

}

else

{

System.out.println("This puzzle is invalid.");

}

System.out.println(getPrintableSudoku(puzzle3));

System.out.println("--------------------------------------------------");

}

public static String[][] makeSudoku(String s)

{

int SIZE = 9;

int k = 0;

String[][] x = new String[SIZE][SIZE];

for (int i = 0; i < SIZE; i++)

{

for (int j = 0; j < SIZE; j++)

{

x[i][j] = s.substring(k, k + 1);

k++;

}

}

return x;

}

public static String getPrintableSudoku(String[][] x)

{

int SIZE = 9;

String temp = "";

for (int i = 0; i < SIZE; i++)

{

if ((i == 3) || (i == 6))

{

temp = temp + "=================\n";

}

for (int j = 0; j < SIZE; j++)

{

if ((j == 3) || (j == 6))

{

temp = temp + " || ";

}

temp = temp + x[i][j];

}

temp = temp + "\n";

}

return temp;

}

public static boolean isValidSudoku(String[][] x)

{

return rowsAreLatin(x) && colsAreLatin(x) && goodSubsquares(x);

}

public static boolean rowsAreLatin(String[][] x)

{

// fill in your code here

}

public static boolean rowIsLatin(String[][] x, int i)

{

// fill in your code here

}

public static boolean colsAreLatin(String[][] x)

{

// fill in your code here

}

public static boolean colIsLatin(String[][] x, int j)

{

// fill in your code here

}

public static boolean goodSubsquares(String[][] x)

{

// fill in your code here

}

public static boolean goodSubsquare(String[][] x, int i, int j)

{

// fill in your code here

}

}