CSE143 Section #18 Problems

Two-dimensional arrays are covered in chapter 7. The first index represents the row and the second index represents the column, as in data[3][5] for the value in row 3 and column 5. In a rectangular array, the number of columns is the same for each row. It is typically constructed as follows (constructing an array of four rows and six columns):

```
        int[][] data = new int[4][6];
```

In a jagged array, the number of columns varies across rows. You construct one by first constructing the array of rows and then each individual row, as in:

```
        int[][] data = new int[3][];
        data[0] = new int[2];
        data[1] = new int[3];
        data[2] = new int[5];
```

This constructs an array of three rows where row 0 has 2 columns, row 1 has 3 columns, and row 2 has 5 columns.

For all problems involving a two-dimensional array, the contents should be indicated using the Arrays.deepToString format of nested bracketed lists. For example, given the following array:

```
        int[][] data = {{8, 12, 14}, {7, 19, 4}, {8, 3, 42}};
```

Its contents should be displayed as follows:

```
        [[8, 12, 14], [7, 19, 4], [8, 3, 42]]
```

1. Consider the following method:

```
        public static int[][] mystery1(int n) {
            int[][] result = new int[n][2 * n - 1];
            for (int i = 0; i < result.length; i++) {
                for (int j = 0; j < result[i].length; j++) {
                    result[i][j] = i + j;
                }
            }
            return result;
        }
```

   For each call below, indicate the contents of the two-dimensional array that is returned.

Method Call	Value Returned
mystery1(1)	______________________________
mystery1(2)	______________________________
mystery1(3)	______________________________
mystery1(4)	______________________________
	______________________________

2. Consider the following method:

```
public void mystery2(int[][] data, int pos, int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        int sum = 0;
        for (int j = 0; j < cols; j++) {
            sum = sum + data[pos + i][pos + j];
        }
        System.out.print(sum + " ");
    }
    System.out.println();
}
```

Suppose that a variable called grid has been declared as follows:

```
int[][] grid = {{4, 6, 8, 8, 2, 1}, {7, 4, 8, 8, 7, 7},
                {7, 8, 6, 6, 7, 2}, {1, 2, 2, 7, 5, 7},
                {8, 3, 6, 6, 1, 1}, {9, 7, 9, 6, 6, 1}};
```

which means it will store the following 6-by-6 grid of values:

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

For each call below, indicate what output is produced:

Method Call	Output Produced
mystery2(grid, 0, 2, 2);	_______________________
mystery2(grid, 2, 3, 2);	_______________________
mystery2(grid, 1, 4, 1);	_______________________

3. Consider the following method:

```
public Set<Integer> mystery3(int[][] data) {
    Set<Integer> result = new TreeSet<>();
    for (int i = 0; i < data.length; i++) {
        for (int j = 0; j < data[i].length; j++) {
            result.add(i * 10 + data[i][j]);
        }
    }
    return result;
}
```

In the left-hand column below are specific two-dimensional arrays. You are to indicate in the right-hand column what values would be stored in the set returned by method mystery3 if the array in the left-hand column is passed as a parameter to mystery3. Set elements should be listed in proper order as a comma-separated bracketed list, as in [3, 18, 25].

Two-Dimensional Array	Contents of Set Returned
[[1, 2], [3, 4]]	______________________________
[[7], [], [8, 8, 9, 10]]	______________________________
[[3, 14], [5, 13, 4], [4, 3, 1]]	______________________________

4. Consider the following method:

```
     public Set<Integer> mystery4(int[][] data, int pos, int n) {
         Set<Integer> result = new TreeSet<>();
         for (int i = 0; i < n; i++) {
             for (int j = 0; j < n; j++) {
                 result.add(data[i + pos][j + pos]);
             }
         }
         return result;
     }
```

Suppose that a variable called grid has been declared as follows:

```
    int[][] grid = {{8, 2, 7, 8, 2, 1}, {1, 5, 1, 7, 4, 7},
                    {5, 9, 6, 7, 3, 2}, {7, 8, 7, 7, 7, 9},
                    {4, 2, 6, 9, 2, 3}, {2, 2, 8, 1, 1, 3}};
```

which means it will store the following 6-by-6 grid of values:

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

For each call below, indicate what value is returned. If the method call results in an exception being thrown, write "exception" instead.

Method Call	Contents of Set Returned
mystery3(grid, 2, 2)	_________________________________
mystery3(grid, 0, 2)	_________________________________
mystery3(grid, 3, 3)	_________________________________

5. Write a method called recordGrade that takes as parameters a map of student grades, a student id, a numerical grade, and a course, and that records the grade in the map. The student id is used as a key for the overall map with each id associated with a map of student grades. The map of student grades uses the course as a key and associates each course with a numerical grade (stored in an object of type Double). For example, suppose that a variable called grades stores a map with information for two students each with grades in two courses:

```
     {1212121={cse142=3.0, engl111=2.5}, 4444444={cse143=3.6, engl131=3.8}}
```

The following calls indicate a new grade for student 4444444 (3.2 in phys121) and a grade for a new student (3.5 in math126 for student 1234567).

```
     recordGrade(grades, "4444444", 3.2, "phys121");
     recordGrade(grades, "1234567", 3.5, "math126");
```

After the call, the grades map would store:

```
     {1212121={cse142=3.0, engl111=2.5}, 1234567={math126=3.5},
      4444444={cse143=3.6, engl131=3.8, phys121=3.2}}
```

Notice that the map now includes grade information for student 1234567 and a third grade for student 4444444. A student may take a course more than once, in which case you should store the highest grade the student has gotten for that course. For example, the call:

```
     recordGrade(grades, "1212121", 2.8, "engl111");
```

would change the map to:

```
     {1212121={cse142=3.0, engl111=2.8}, 1234567={math126=3.5},
      4444444={cse143=3.6, engl131=3.8, phys121=3.2}}
```

If the call instead had been with a grade of 2.4 in engl111, the map would have been unchanged because that grade is lower than the grade currently in the map. The overall map has keys that are ordered by student ID and each student's map of grades should be ordered by the course number.

Your method should construct a map for each student not already in the overall map and you may construct iterators, but you are not allowed to construct other structured objects (no string, set, list, etc.).

6. Write a method called removePoints that takes a map and an index as parameters and that removes particular points from the map returning them in a set. The map this method will manipulate uses integer indexes as keys and store as values a list of points. For example, a variable called data might store the following:

```
     {17=[[x=3,y=4], [x=12,y=6], [x=8,y=12], [x=3,y=6]],
      42=[[x=2,y=5], [x=3,y=3], [x=1,y=5], [x=4,y=2], [x=8,y=9]],
      308=[[x=1,y=2], [x=5,y=8], [x=4,y=4], [x=2,y=7], [x=3,y=9]]}
```

   This map has three entries. The first entry associates the key 17 with a list of four points. The second associates the key 42 with a list of five points. The third associates 308 with a list that also has five points.

   When the method is called, it will be passed the map and a key and it will return a set of points, as in:

```
     Set<Point> result = removePoints(data, 42);
```

   The method should manipulate the list of points for the given index, removing any points for which the x-value is less than the y-value and returning these points in a set. After the call above, result should be:

```
     [[x=2,y=5], [x=1,y=5], [x=8,y=9]]
```

   and data should store the following:

```
     {17=[[x=3,y=4], [x=12,y=6], [x=8,y=12], [x=3,y=6]],
      42=[[x=3,y=3], [x=4,y=2]],
      308=[[x=1,y=2], [x=5,y=8], [x=4,y=4], [x=2,y=7], [x=3,y=9]]}
```

   Notice that the index 42 is now associated with a list of just two points (the two that weren't removed). The method should return an empty set if there are no points to remove or if the index value has no corresponding entry in the map.

   Your method should construct a set to return and may construct iterators, but you are not allowed to construct other structured objects (no string, set, list, etc.).

The remaining problems involve writing code that involves two-dimensional arrays. The specific program we will consider is a Sudoku solver that uses recursive backtracking. As with the 8 queens problem, we will define a class that handles much of the low-level detail of the puzzle. In particular, we will have a class called Grid that keeps track of the current state of the 9x9 Sudoku grid. The basic class structure will be:

```
    public class Grid {
        public static final int SIZE = 9;
        private int[][] grid;

        // methods
    }
```

We are writing a fairly specific class with a size of 9, but it is best to use a named constant when possible to add to readability. The grid will store values between 1 and 9. We will assume that the value 0 means that the cell is empty (no value has been assigned to it).

Several of the methods involved use a cell number to refer to a particular location. The idea is that the cells are numbered starting at 0 in row/major order. In other words, we number all of the first row, then all of the second row, and so on. So the cell numbers are:

```
 0  1  2  3  4  5  6  7  8
 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26
...
72 73 74 75 76 77 78 79 80
```

Because we are starting at 0, the highest cell number is 80.

7. Write a method called print that prints the grid contents to System.out with each row printed on a separate line and with a space after each grid value. Empty cells (cells that store a value of 0) should be printed as a dash. For example, if you have a Grid variable g and you make this call:

   ```
   g.print();
   ```

   you should get output that looks something like this:

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

   Instead of using the SIZE constant, write this method so that it would work no matter what the dimensions of the grid are and even if the rows had different numbers of elements in them.

8. Write a constructor for the Grid class that takes a Scanner as a parameter. The Scanner will contain values like those produced by print. In other words, cells that are occupied will be listed with a number between 1 and 9 and cells that are empty will be listed with some other token like a dash. You may assume the Scanner is open and that it contains a legal sequence of tokens (81 of them).

9. Write a method called place that takes a cell number and a value n and that stores the value n in that cell. Your method will have to convert the cell number to a row and column in your grid.

10. Write a method called remove that takes a cell number and that removes the value in that cell (resetting it to 0). As with the place method, you will have to convert the cell number to a row and column in your grid.

11. Write a method called getUnassignedLocation. It should look through the grid in row/major order and return the cell number of the first unoccupied cell. If there are no unoccupied cells, it should return the value -1.

12. Write a method called noConflicts that takes a cell number and a value n and that returns true if it is possible to store n in that cell without generating any Sudoku conflicts. Remember that Sudoku does not allow repetition in any given row, column, or block.

13. Write the recursive backtracking code that will search all possible solutions to the Sudoku puzzle. You should write a method called explore that takes a Grid as a parameter and that attempts to fill the grid with a solution. It should return whether or not a solution is found. Below is a method that calls the explore method and reports the result:

```
    public static void solve(Grid g) {
        if (!explore(g))
            System.out.println("No solution.");
        else {
            System.out.println("One solution is as follows:");
            g.print();
        }
    }
```

The Grid object has the following public member functions available to you:

getUnassignedLocation()	returns the cell number of the first unoccupied cell (-1 if no such cell exists)
noConflicts(cellNumber, n)	returns whether it is legal to place the value n in the given cell without violating Sudoku constraints
place(cellNumber, n)	places the value n in the given cell
remove(cellNumber)	removes the value in the given cell, resetting it to be unoccupied