CSE143X Section #19 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.).

For problems 7-10, assume that we are using the standard ListNode class:

```
        public class ListNode {
            public int data;       // data stored in this node
            public ListNode next;  // link to next node in the list

            <constructors>
        }
```

And that we are writing methods for a class called LinkedIntList that has a single data field of type ListNode called front:

```
        public class LinkedIntList {
            private ListNode front;

            <methods>
        }
```

In solving these problems, you may not call any other methods of the class, you may not construct new nodes and you may not use any auxiliary data structure to solve this problem (no array, ArrayList, stack, queue, String, etc). You also may not change any data fields of the nodes. You MUST solve these problems by rearranging the links of the lists involved.

7. Write a method evenSum that returns the sum of the values in even indexes in a list of integers. Assume we are using 0-based indexing where the first value in the list has index 0, the second value has index 1 and so on. The values we are interested in are the ones with even indexes (the value at index 0, the value at index 2, the value at index 4, and so on).

   For example, if a variable called list stores the following sequence of values:

   ```
   [1, 18, 2, 7, 39, 8, 40, 7]
   ```

   then the call:

   ```
   list.evenSum()
   ```

   should return the value 82 (1 + 2 + 39 + 40). Notice that what is important is the position of the numbers (index 0, index 2, index 4, etc), not whether the numbers themselves are even. If the list is empty, your method should return a sum of 0.

8. Write a method removeDuplicates that removes any duplicates from a list of integers. The resulting list should have the values in the same relative order as their first occurrence in the original list. In other words, a value i should appear before a value j in the final list if and only if the first occurrence of i appears before the first occurrence of j in the original list. For example, if a variable called list stores the following:

   ```
   [14, 8, 14, 12, 1, 14, 11, 8, 8, 10, 4, 9, 1, 2, 5, 2, 4, 12, 12]
   ```

   and the following call is made:

   ```
   list.removeDuplicates();
   ```

   then list should store these values after the call:

   ```
   [14, 8, 12, 1, 11, 10, 4, 9, 2, 5]
   ```

9. Write a method switchPairs that switches the order of elements in a linked list of integers in a pairwise fashion. Your method should switch the order of the first two values, then switch the order of the next two, switch the order of the next two, and so on. For example, if the list initially stores these values:

   ```
   [3, 7, 4, 9, 8, 12]
   ```

   your method should switch the first pair (3, 7), the second pair (4, 9) and the third pair (8, 12), to yield this list:

   ```
   [7, 3, 9, 4, 12, 8]
   ```

   If there are an odd number of values in the list, the final element is not moved. For example, if the original list had been:

   ```
   [3, 7, 4, 9, 8, 12, 2]
   ```

   It would again switch pairs of values, but the final value (2) would not be moved, yielding this list:

   ```
   [7, 3, 9, 4, 12, 8, 2]
   ```

10. Write a method takeSmallerFrom that compares two lists of integers, making sure that the first list has smaller values in corresponding positions. For example, suppose the the variables list1 and list2 refer to lists that contain the following values:

```
list1: [3, 16, 7, 23]
list2: [2, 12, 6, 54]
```

If the following call is made:

```
list1.takeSmallerFrom(list2);
```

the method will compare values in corresponding positions and move the smaller values to list1. It will find that among the first pair, 2 is smaller than 3, so it needs to move. In the second pair, 12 is smaller than 16, so it needs to move. In the third pair, 6 is smaller than 7, so it needs to move. In the fourth pair, 54 is not smaller than 23, so those values can stay where they are. Thus, after the call, the lists should store these values:

```
list1: [2, 12, 6, 23]
list2: [3, 16, 7, 54]
```

One list might be longer than the other, in which case those values should stay at the end of their list. For example, for these lists:

```
list1: [2, 4, 6, 8, 10, 12]
list2: [1, 3, 6, 9]
```

the call:

```
list1.takeSmallerFrom(list2);
```

should leave the lists with these values:

```
list1: [1, 3, 6, 8, 10, 12]
list2: [2, 4, 6, 9]
```