Comp151 Lab06

In this lab you will analyze the behavior of selected simple sorts and you will implement three sorting assignments.

I. Working with `SortArray.java,` investigate the effect of array size and initial element order on the number of comparisons and moves required by each of the sorting algorithms. Use arrays with 50 and 1000 integers. Use three initial orderings of each array:

   a) randomly generated
   b) copy of the above array sorted in ascending order
   c) copy of the array generated in step b) sorted in descending order.

   A. Use the `Random` class methods to generate a list of the requested number of integer values in an array. Implement methods:
      - `generateRandomArray()` and
      - `reverseOrder()` – this method must **<u>not</u>** create another array
   B. Run `SortArray.java` file (that is `selectionSort`, `insertionSort`, and `shellSort`) to sort the array.
      - Analyze how the code counts the number of comparisons and the number of moves performed by each sort
      - Test your code using the test driver implemented in `main` of `SortArray.java.`
   C. Implement exercise #14c on page 266 of the textbook (also see the lecture notes). The skeleton of the `improvedShellSort` is already defined in the `SortArray.java`
   D. Implement exercise #10 on page 266 of the textbook in method called `improvedBubbleSort` as defined in the `SortArray.java.` Note that the ***"regular" bubble sort*** is described in exercise #8.
   E. A ***shaker improved bubble sort*** further modifies the original bubble sort. Each pass has up to two iterations; the direction of bubbling changes in each iteration: in one iteration, the largest element is bubbled up; in the next the smallest element is bubbled down; in the next the second largest is bubbled up; in the next the second smallest is bubbled down; and so forth. Implement this new algorithm in `shakerImprovedBubbleSort` method that is also defined in `SortArray.java.` Make sure to apply improvements you've implemented in `improvedBubbleSort.`
   F. Add statistics to `improvedBubbleSort`, and `shakerImprovedBubbleSort` methods
   G. Report the results in the provided spreadsheet.

Sample run with the seed set to 101:

```
What size arrays should be used?
   It should be an integer value greater than or equal to 1.
17

*** ANALYZING SELECTION SORT ***
---> ARRAY IN RANDOM ORDER
[ 40 90 93 54 30 65 79 33 82 6 14 12 99 6 18 97 88 ]
Number of comparisons --> 136
Number of moves --> 26
The sorted array is:
[ 6 6 12 14 18 30 33 40 54 65 79 82 88 90 93 97 99 ]
---> ARRAY SORTED IN ASCENDING ORDER
The array is:
[ 6 6 12 14 18 30 33 40 54 65 79 82 88 90 93 97 99 ]
Number of comparisons --> 136
Number of moves --> 0
The sorted array is:
[ 6 6 12 14 18 30 33 40 54 65 79 82 88 90 93 97 99 ]
---> ARRAY SORTED IN DESCENDING ORDER
```

```
The array is:
[ 99 97 93 90 88 82 79 65 54 40 33 30 18 14 12 6 6 ]
Number of comparisons --> 136
Number of moves --> 18
The sorted array is:
[ 6 6 12 14 18 30 33 40 54 65 79 82 88 90 93 97 99 ]

*** ANALYZING INSERTION SORT ***
---> ARRAY IN RANDOM ORDER
[ 40 90 93 54 30 65 79 33 82 6 14 12 99 6 18 97 88 ]
Number of comparisons --> 87
Number of moves --> 89
The sorted array is:
[ 6 6 12 14 18 30 33 40 54 65 79 82 88 90 93 97 99 ]
---> ARRAY SORTED IN ASCENDING ORDER
The array is:
[ 6 6 12 14 18 30 33 40 54 65 79 82 88 90 93 97 99 ]
Number of comparisons --> 16
Number of moves --> 16
The sorted array is:
[ 6 6 12 14 18 30 33 40 54 65 79 82 88 90 93 97 99 ]
---> ARRAY SORTED IN DESCENDING ORDER
The array is:
[ 99 97 93 90 88 82 79 65 54 40 33 30 18 14 12 6 6 ]
Number of comparisons --> 136
Number of moves --> 151
The sorted array is:
[ 6 6 12 14 18 30 33 40 54 65 79 82 88 90 93 97 99 ]

*** ANALYZING SHELL SORT ***
---> ARRAY IN RANDOM ORDER
[ 40 90 93 54 30 65 79 33 82 6 14 12 99 6 18 97 88 ]
Number of comparisons --> 70
Number of moves --> 78
The sorted array is:
[ 6 6 12 14 18 30 33 40 54 65 79 82 88 90 93 97 99 ]
---> ARRAY SORTED IN ASCENDING ORDER
The array is:
[ 6 6 12 14 18 30 33 40 54 65 79 82 88 90 93 97 99 ]
Number of comparisons --> 53
Number of moves --> 53
The sorted array is:
[ 6 6 12 14 18 30 33 40 54 65 79 82 88 90 93 97 99 ]
---> ARRAY SORTED IN DESCENDING ORDER
The array is:
[ 99 97 93 90 88 82 79 65 54 40 33 30 18 14 12 6 6 ]
Number of comparisons --> 60
Number of moves --> 73
The sorted array is:
[ 6 6 12 14 18 30 33 40 54 65 79 82 88 90 93 97 99 ]

*** ANALYZING IMPROVED SHELL SORT ***
---> ARRAY IN RANDOM ORDER
[ 40 90 93 54 30 65 79 33 82 6 14 12 99 6 18 97 88 ]
Number of comparisons --> 66
Number of moves --> 73
The sorted array is:
[ 6 6 12 14 18 30 33 40 54 65 79 82 88 90 93 97 99 ]
---> ARRAY SORTED IN ASCENDING ORDER
The array is:
[ 6 6 12 14 18 30 33 40 54 65 79 82 88 90 93 97 99 ]
Number of comparisons --> 50
Number of moves --> 50
The sorted array is:
[ 6 6 12 14 18 30 33 40 54 65 79 82 88 90 93 97 99 ]
---> ARRAY SORTED IN DESCENDING ORDER
The array is:
[ 99 97 93 90 88 82 79 65 54 40 33 30 18 14 12 6 6 ]
Number of comparisons --> 65
Number of moves --> 77
The sorted array is:
[ 6 6 12 14 18 30 33 40 54 65 79 82 88 90 93 97 99 ]

*** ANALYZING BUBBLE SORT ***
---> ARRAY IN RANDOM ORDER
[ 40 90 93 54 30 65 79 33 82 6 14 12 99 6 18 97 88 ]
Number of comparisons --> 136
Number of moves --> 146
The sorted array is:
[ 6 6 12 14 18 30 33 40 54 65 79 82 88 90 93 97 99 ]
---> ARRAY SORTED IN ASCENDING ORDER
The array is:
[ 6 6 12 14 18 30 33 40 54 65 79 82 88 90 93 97 99 ]
```

```
Number of comparisons --> 136
Number of moves --> 0
The sorted array is:
[ 6 6 12 14 18 30 33 40 54 65 79 82 88 90 93 97 99 ]
---> ARRAY SORTED IN DESCENDING ORDER
The array is:
[ 99 97 93 90 88 82 79 65 54 40 33 30 18 14 12 6 6 ]
Number of comparisons --> 136
Number of moves --> 270
The sorted array is:
[ 6 6 12 14 18 30 33 40 54 65 79 82 88 90 93 97 99 ]

*** ANALYZING IMPROVED BUBBLE SORT ***
---> ARRAY IN RANDOM ORDER
[ 40 90 93 54 30 65 79 33 82 6 14 12 99 6 18 97 88 ]
Number of comparisons --> 112
Number of moves --> 146
The sorted array is:
[ 6 6 12 14 18 30 33 40 54 65 79 82 88 90 93 97 99 ]
---> ARRAY SORTED IN ASCENDING ORDER
The array is:
[ 6 6 12 14 18 30 33 40 54 65 79 82 88 90 93 97 99 ]
Number of comparisons --> 16
Number of moves --> 0
The sorted array is:
[ 6 6 12 14 18 30 33 40 54 65 79 82 88 90 93 97 99 ]
---> ARRAY SORTED IN DESCENDING ORDER
The array is:
[ 99 97 93 90 88 82 79 65 54 40 33 30 18 14 12 6 6 ]
Number of comparisons --> 136
Number of moves --> 270
The sorted array is:
[ 6 6 12 14 18 30 33 40 54 65 79 82 88 90 93 97 99 ]

*** ANALYZING SHAKER IMPROVED BUBBLE SORT ***
---> ARRAY IN RANDOM ORDER
[ 40 90 93 54 30 65 79 33 82 6 14 12 99 6 18 97 88 ]
Number of comparisons --> 106
Number of moves --> 146
The sorted array is:
[ 6 6 12 14 18 30 33 40 54 65 79 82 88 90 93 97 99 ]
---> ARRAY SORTED IN ASCENDING ORDER
The array is:
[ 6 6 12 14 18 30 33 40 54 65 79 82 88 90 93 97 99 ]
Number of comparisons --> 16
Number of moves --> 0
The sorted array is:
[ 6 6 12 14 18 30 33 40 54 65 79 82 88 90 93 97 99 ]
---> ARRAY SORTED IN DESCENDING ORDER
The array is:
[ 99 97 93 90 88 82 79 65 54 40 33 30 18 14 12 6 6 ]
Number of comparisons --> 136
Number of moves --> 270
The sorted array is:
[ 6 6 12 14 18 30 33 40 54 65 79 82 88 90 93 97 99 ]
****************DONE********************
```

II. Open `UniqueAnagrams.java` and implement the recursive method `permutations` that produces all the permutations of the characters in a string passed to the method as its argument. Take each character and follow it with all permutations of the remaining characters. For the character sequence `abcd`, begin with `a` and follow it with all permutations of the remaining characters, `b, c,` and `d`, to get: **a**`bcd`, **a**`bcd`, **a**`cbd`, **a**`cdb`, **a**`dbc`, **a**`dcb`. Then begin with the next character, `b`, in the original string and follow it with all permutations of the remaining characters, `a, c,` and `d`, to get: **b**`acd`, **b**`adc`, **b**`cad`, **b**`cda`, **b**`dac`, and **b**`dca`. Continue in this fashion, by beginning with `c` and following it with all permutations of the remaining characters, `a, b,` and `d`, and finally by beginning with `d` and following it with all permutations of the remaining characters, `a, b,` and `c`.

To give more insight into the recursive solution, let's consider the step when we take `b` and follow it with all permutations of the remaining characters, `a, c,` and `d`. Let's represent this step as

b `acd`

A recursive call on `acd` would give us

ba `cd`

The recursive call on `cd` would result in

bac `d`

and the recursive call on `d` would be the base case and give us the permutation

bacd

Let's now consider the following two examples:

1. The character sequence `abc` has the following permutations: `acb, bac, bca, cab, cba`. In this example, all the characters are unique, so the permutations do not have duplicates.
2. However, the character sequence `xbba` has the following permutations: `xbb, xbb, bxb, bbx, bxb, bbx.` In this example, the recursive method produces all possible permutations, and when the characters repeat in the original sequence, there will be duplicates in the list of permutations.

Next implement `sort` method (and its helper methods `getIndexOfSmallestAndRemoveDuplicates` and `swap`) that should utilize the ***iterative selection sort algorithm*** to sort the permutations, **and** while sorting, **remove duplicates**. You need to call `ArrayList` methods like `get` and `set` to rearrange the elements in `this.anagrams` collection as appropriate.

See sample runs below:

Run #1:

```
Please enter a word
car
Generated anagrams = [car, cra, acr, arc, rca, rac]
Expected unique and sorted anagrams = [acr, arc, car, cra, rac, rca]

Computed unique and sorted anagrams = [acr, arc, car, cra, rac, rca]
```

Run #2:

```
Please enter a word
abc
Generated anagrams = [abc, acb, bac, bca, cab, cba]
Expected unique and sorted anagrams = [abc, acb, bac, bca, cab, cba]

Computed unique and sorted anagrams = [abc, acb, bac, bca, cab, cba]
```

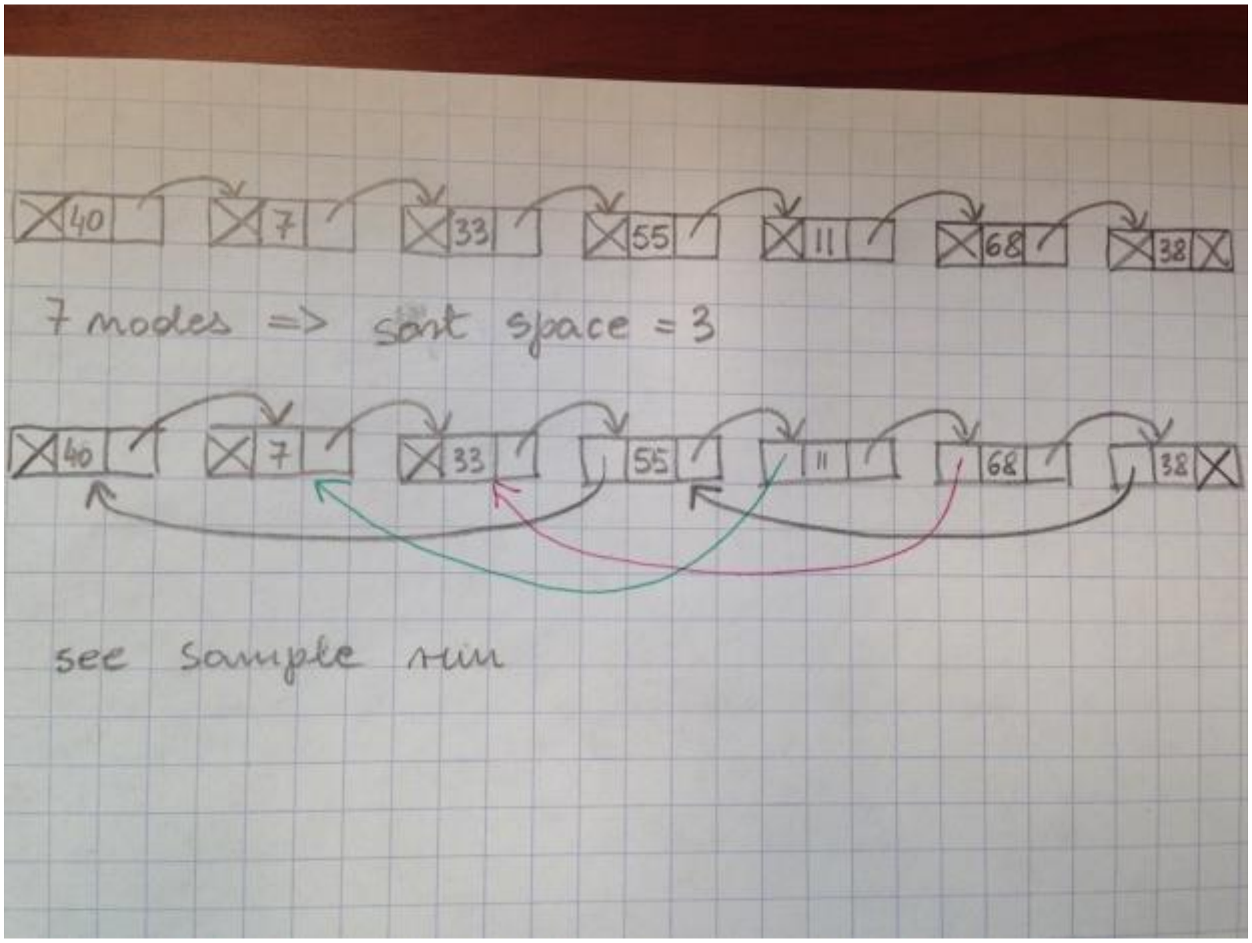
Run #3:

```
Please enter a word
abba
Generated anagrams = [abba, abab, abba, abab, aabb, aabb, baba, baab, bbaa, bbaa, baab, baba, baba, baab,
bbaa, bbaa, baab, baba, aabb, aabb, abab, abba, abab, abba]
Expected unique and sorted anagrams = [aabb, abab, abba, baab, baba, bbaa]

Computed unique and sorted anagrams = [aabb, abab, abba, baab, baba, bbaa]
```

III. Working with `ChainSort.java` apply `incrementalInsertionSort` algorithm to work with a linked chain instead of an array and implement a Shell sort for a linked chain. The skeleton of the application is provided.

1. Before calling `incrementalInsertionSort` method, set the `previous` pointers to create the appropriate sub-chains as shown in the picture below:



2. During the sorting process, **replace appropriate data without changing the `next` pointers**.
3. Sample run with 7 elements and seed 11:

```
What size chain should be used?
   It should be an integer value greater than or equal to 1.
7
What seed value should be used?
   It should be an integer value greater than or equal to 1.
11

Original Chain Content: 40 7 33 55 11 68 38

----->Before partial sort with space 3 :
40 7 33 55 11 68 38
-> Comparing 55 with 40
-> Comparing 11 with 7
-> Comparing 68 with 33
-> Comparing 38 with 55
--->  38 is smaller than 55:
-> Comparing 38 with 40
--->  38 is smaller than 40:
38 7 33 40 11 68 55
----->After partial sort done with space 3 :
38 7 33 40 11 68 55
```

```
----->Before partial sort with space 1 :
38 7 33 40 11 68 55
-> Comparing 7 with 38
--->  7 is smaller than 38:
7 38 33 40 11 68 55
-> Comparing 33 with 38
--->  33 is smaller than 38:
-> Comparing 33 with 7
7 33 38 40 11 68 55
-> Comparing 40 with 38
-> Comparing 11 with 40
--->  11 is smaller than 40:
-> Comparing 11 with 38
--->  11 is smaller than 38:
-> Comparing 11 with 33
--->  11 is smaller than 33:
-> Comparing 11 with 7
7 11 33 38 40 68 55
-> Comparing 68 with 40
-> Comparing 55 with 68
--->  55 is smaller than 68:
-> Comparing 55 with 40
7 11 33 38 40 55 68
----->After partial sort done with space 1 :
7 11 33 38 40 55 68

Sorted Chain Content: 7 11 33 38 40 55 68
```

4. Sample run with 15 elements and seed 101:

```
What size chain should be used?
   It should be an integer value greater than or equal to 1.
15
What seed value should be used?
   It should be an integer value greater than or equal to 1.
101

Original Chain Content: 18 6 99 12 14 6 82 33 79 65 30 54 93 90 40

----->Before partial sort with space 7 :
18 6 99 12 14 6 82 33 79 65 30 54 93 90 40
-> Comparing 33 with 18
-> Comparing 79 with 6
-> Comparing 65 with 99
--->  65 is smaller than 99:
18 6 65 12 14 6 82 33 79 99 30 54 93 90 40
-> Comparing 30 with 12
-> Comparing 54 with 14
-> Comparing 93 with 6
-> Comparing 90 with 82
-> Comparing 40 with 33
----->After partial sort done with space 7 :
18 6 65 12 14 6 82 33 79 99 30 54 93 90 40

----->Before partial sort with space 3 :
18 6 65 12 14 6 82 33 79 99 30 54 93 90 40
-> Comparing 12 with 18
--->  12 is smaller than 18:
12 6 65 18 14 6 82 33 79 99 30 54 93 90 40
-> Comparing 14 with 6
-> Comparing 6 with 65
--->  6 is smaller than 65:
12 6 6 18 14 65 82 33 79 99 30 54 93 90 40
-> Comparing 82 with 18
-> Comparing 33 with 14
-> Comparing 79 with 65
-> Comparing 99 with 82
-> Comparing 30 with 33
--->  30 is smaller than 33:
-> Comparing 30 with 14
12 6 6 18 14 65 82 30 79 99 33 54 93 90 40
-> Comparing 54 with 79
--->  54 is smaller than 79:
-> Comparing 54 with 65
--->  54 is smaller than 65:
-> Comparing 54 with 6
12 6 6 18 14 54 82 30 65 99 33 79 93 90 40
```

```
-> Comparing 93 with 99
--->  93 is smaller than 99:
-> Comparing 93 with 82
12 6 6 18 14 54 82 30 65 93 33 79 99 90 40
-> Comparing 90 with 33
-> Comparing 40 with 79
--->  40 is smaller than 79:
-> Comparing 40 with 65
--->  40 is smaller than 65:
-> Comparing 40 with 54
--->  40 is smaller than 54:
-> Comparing 40 with 6
12 6 6 18 14 40 82 30 54 93 33 65 99 90 79
----->After partial sort done with space 3 :
12 6 6 18 14 40 82 30 54 93 33 65 99 90 79

----->Before partial sort with space 1 :
12 6 6 18 14 40 82 30 54 93 33 65 99 90 79
-> Comparing 6 with 12
--->  6 is smaller than 12:
6 12 6 18 14 40 82 30 54 93 33 65 99 90 79
-> Comparing 6 with 12
--->  6 is smaller than 12:
-> Comparing 6 with 6
6 6 12 18 14 40 82 30 54 93 33 65 99 90 79
-> Comparing 18 with 12
-> Comparing 14 with 18
--->  14 is smaller than 18:
-> Comparing 14 with 12
6 6 12 14 18 40 82 30 54 93 33 65 99 90 79
-> Comparing 40 with 18
-> Comparing 82 with 40
-> Comparing 30 with 82
--->  30 is smaller than 82:
-> Comparing 30 with 40
--->  30 is smaller than 40:
-> Comparing 30 with 18
6 6 12 14 18 30 40 82 54 93 33 65 99 90 79
-> Comparing 54 with 82
--->  54 is smaller than 82:
-> Comparing 54 with 40
6 6 12 14 18 30 40 54 82 93 33 65 99 90 79
-> Comparing 93 with 82
-> Comparing 33 with 93
--->  33 is smaller than 93:
-> Comparing 33 with 82
--->  33 is smaller than 82:
-> Comparing 33 with 54
--->  33 is smaller than 54:
-> Comparing 33 with 40
--->  33 is smaller than 40:
-> Comparing 33 with 30
6 6 12 14 18 30 33 40 54 82 93 65 99 90 79
-> Comparing 65 with 93
--->  65 is smaller than 93:
-> Comparing 65 with 82
--->  65 is smaller than 82:
-> Comparing 65 with 54
6 6 12 14 18 30 33 40 54 65 82 93 99 90 79
-> Comparing 99 with 93
-> Comparing 90 with 99
--->  90 is smaller than 99:
-> Comparing 90 with 93
--->  90 is smaller than 93:
-> Comparing 90 with 82
6 6 12 14 18 30 33 40 54 65 82 90 93 99 79
-> Comparing 79 with 99
--->  79 is smaller than 99:
-> Comparing 79 with 93
--->  79 is smaller than 93:
-> Comparing 79 with 90
--->  79 is smaller than 90:
-> Comparing 79 with 82
--->  79 is smaller than 82:
-> Comparing 79 with 65
6 6 12 14 18 30 33 40 54 65 79 82 90 93 99
----->After partial sort done with space 1 :
6 6 12 14 18 30 33 40 54 65 79 82 90 93 99

Sorted Chain Content: 6 6 12 14 18 30 33 40 54 65 79 82 90 93 99
```