Assignment 2

DUE DATE: FRIDAY JUNE 12, 2020 AT 11:59 PM

Notes:

- Submit ONE .java file per question. Each .java file must contain ALL of the source code for a question. See the Programming Standards document for how to store several classes in the same file.
- Each filename must have the format `<your last name><your first name>A2Q2.java` (e.g. `SmithJohnA1Q1.java`). Please use your name as shown in UM Learn.
- Do not submit any output. Your code will be run during marking.
- Your program must compile, run, and end normally (not crash or get stuck in an infinite loop) to receive any marks. See the Assignment Information file for some tips on how to make sure your code runs for the markers.
- Assignments must follow the Programming Standards posted in UM Learn.
- Assignment submissions are only accepted via UM Learn. Submissions by email will not be accepted.
- You may submit your assignment multiple times, but only the most recent version will be marked.
- Assignments become late immediately after the posted due date and time. Late assignments will be accepted up to 49 hours after that time, at a penalty of 2% per hour or portion thereof. After 49 hours, an assignment is worth 0 marks and will no longer be accepted.
- The time of the last submission controls the late penalty for the entire assignment.
- These assignments are your chance to learn the material for the tests. *Code your assignments independently*. We use software to compare all submitted assignments to each other, and pursue academic dishonestly vigorously.

Question 1: Time to Sort [20 marks code + 5 marks report]

This question can be done after Week 3 of the course.

In this question you will complete a program that times the execution of seven sorting algorithms (insertion sort, bubble sort, selection sort, merge sort, quick sort, a hybrid quick sort (details below), and a shell sort) and reports on the run time and whether the algorithms actually sorted the list.

The file **`A2Q1SortingTemplate.java`** contains your starting point for this assignment. Do not change any of the provided code, except change the name of the application class to include your name. **Your job is to add the sorting methods, as described below, and any helper methods needed by the sorting methods.** You are permitted to use and modify code provided in the class notes and textbook. You are not permitted to use code from any other sources.

A (non-recursive) Insertion Sort

Take the insertion sort algorithm from class and modify it so that it sorts only positions `array[start]` to (and including) `array[end-1]` in the array, not touching any other position in array. The header should be:

```
private static void insertionSort( int[] array, int start, int end)
```

Also write a public driver method with the header

```
public static void insertionSort( int[] array )
```

whose task is simply to call the private method, passing the correct values to the private method's parameters so that it sorts the entire array.

These methods must not create arrays.

Note that the private method with also be used by the hybrid quick sort.

Hint: Test the private method's ability to sort only part of the array without touching any other part of the array. Many students get this wrong, and then can't figure out why their hybrid quick sort doesn't work.

A (non-recursive) Bubble Sort

Implement a bubble sort with the header

```
public static void bubbleSort( int[] array )
```

to sort the entire array, as reviewed in class and seen in COMP 1020.

A (non-recursive) Selection Sort

Implement a selection sort with header

```
public static void selectionSort( int[] array )
```

that sorts the entire array by locating the minimum item amongst the unsorted items at each step. This sort should make use of a helper method

```
private static int findMin( int[] array, int start, int end )
```

that will return the index of the minimum item stored in positions `start` to `end-1` (inclusive) in the array. This sort should also make use of the provided `swap` method.

A Recursive Merge Sort

Implement the algorithm as discussed in class, except that you should add a new base case: if there are just two items to be sorted, swap them if necessary. There are three methods that you need to write:

- The public driver `mergeSort` method: It simply calls the private recursive method with the array and the extra parameters (the `start` and `end` indices and the extra array `temp`) that the recursive method needs.
- The private recursive helper `mergeSort` method: It does the recursive merge sort. It receives the array, indices `start` and `end`, and the temporary array as parameters. Its task is to merge sort positions `start` to `end-1` (inclusive) in the array – and it must not touch any other positions in the array. This is the method that should have the new base case added to it.
- The non-recursive helper `merge` method: It merges two sorted sublists (defined by three indices `start`, `mid`, and `end`) into one sorted list, using the extra array `temp`. Make sure that you use the indices to define the sublists in a way that is consistent with how sublists are defined by indices in all other parts of the code: from some index up to, but not including, some other index.

Reminder: Only the public driver method should create an array. All the other methods used by merge sort must use the arrays and positions (indices) that they are passed in their parameters without creating any other arrays.

A Recursive Quick Sort

Implement the algorithm as discussed in class, except that you should add the base case: if there are just two items to be sorted, swap them if necessary. You need to write the following methods:

- The public driver `quickSort` method: It simply calls the private recursive method, passing it the extra parameters (the start and end indices) the recursive method needs.
- The private recursive helper `quickSort` method: It is the recursive quick sort method, which receives the array, and indices `start` and `end` as parameters. Its task is to quick sort positions `start` to `end-1` (inclusive) in the array – and it must not touch any other positions in the array. This is the method that should have a base case added for two items.
- The private non-recursive median-of-three method: It chooses a pivot from the items in positions `start` to `end-1` (inclusive) in the array using the median-of-three method, and swaps the chosen pivot into position `start` in the array.
- The private non-recursive partition method: It partitions the items in positions `start` to `end-1` (inclusive) in the array using the chosen pivot (which it assumes is already in position `start`), and returns the final position of the pivot after the partition is complete. It should use one simple `for`-loop.

None of these methods should create an array.

A Hybrid Recursive Quick Sort that uses a Breakpoint

Use the header

```
private static void hybridQuickSort( int[] array, int start, int end )
```

This method is similar to the recursive `quickSort` algorithm above, except it has a different base case:

- If `array[start]` to (and including) `array[end-1]` is fewer than BREAKPOINT items, then call the `private insertionSort` method to sort `array[start]` to (and including) `array[end-1]` (and not touch any other position in the array).

If `array[start]` to (and including) `array[end-1]` consists of at least BREAKPOINT items, then do the usual quick sort steps (choose a pivot using the median-of-three technique, partition the items using the chosen pivot, and finally recursively call `hybridQuickSort` twice to sort each of the smalls and the bigs).

Make sure the recursive calls in `hybridQuickSort` are to `hybridQuickSort`, not to `quickSort`.

Also write a public driver method with the header

```
public static void hybridQuickSort( int[] array )
```

Its task is to simply call the above private `hybridQuickSort` method, passing the correct values to the private method's parameters so that it sorts the entire array.

None of the hybrid quick sort methods should create an array.

A (non-recursive) Shell Sort

Implement a shell sort with the header

```
public static void shellSort( int[] array )
```

that will perform an insertion sort on widely-spaced items, then less widely-spaced items, then even less widely-spaced items, etc., until it ends with a regular insertion sort that includes all items. The shell sort should use Knuth's sequence for the gap sequence, starting with the largest possible gap ***h*** that is smaller than the size of the array, as described in the week 3 lecture slides.

Report

In the comments at the end of your program, paste the output from one run of your program, and answer the following questions:

1. Was insertion sort faster than selection sort? Why or why not?
2. Was quick sort faster than insertion sort? Why or why not?
3. Was hybrid quick sort faster than quick sort? Why or why not?
4. Which sort(s) would you recommend to others, and why?
5. Which sort(s) would you warn others against using, and why?

Question 2: Modelling a Train with a Linked List [25 marks]

This question requires material from Weeks 4 & 5.

In this question you will model a train with a linked list, where each node in the linked list will represent one car on the train.

- Model the train with a **doubly-linked list**, where each node has a reference (pointer) to both the node ahead of it and the node behind it.
- Your program should ask the user to enter the name of an input file.
- Your program will process the input file, performing the requested operations as it goes. The input file will consist of lines of commands, with a command possibly followed by lines of data. See the input description below. Echo each command to the console, perform the operation, and print a summary of the actions performed. Please see the sample output below.
- You must create at least four classes: a TrainCar class (which stores the information about one train car), a Node class (which stores links to neighbouring nodes and one TrainCar object), a Train class (the linked list class), and an application/main class (to process the input file).
- The TrainCar class should store the type of cargo (a String), and the value of the cargo.
- The Train class constructor will create a train consisting of one car (an engine, where the type of cargo is set to "engine" and the value is set to $0).

Program Input

Possible commands in the input file are:

- PICKUP [num]
  num (an integer) indicates the number of train cars to be added to the train. Each car is listed separately on a following line (type of cargo followed by value, separated by a space; engines do not have a value). Engines are always added at the front of the train. Cars containing cargo are added at the end of the train.
- PRINT
  Print the entire train, including engines(s). Print one line displaying the number of cars in the train and the total value of the cargo. On a second line, print the train from front to back, listing the type of cargo in each car (the list of cars may wrap to multiple lines).
- DROPLAST [num]
  Remove the last num (an integer) cargo cars from the train. Never drop engines. If the number given is larger than the number of cargo cars on the train, print the number actually dropped.
- DROPFIRST [num]
  Remove the first num (an integer) cargo cars from the train. That is, drop the first num cars after the engine(s). If the number given is larger than the number of cargo cars on the train, print the number actually dropped.

- DROP [type] [num]
  Remove the first num (an integer) cars that contain the specified type (a String) of cargo. If fewer than num cars of the specified type exist, print the number actually dropped.

Sample program input:

```
PICKUP 6
oil 40000
wheat 20000
lumber 30000
engine
oil 45000
oil 60000
PRINT
DROPLAST 1
PRINT
PICKUP 8
oil 50000
lumber 25000
wheat 25000
oil 30000
oil 45000
wheat 20000
lumber 20000
oil 60000
PRINT
DROP oil 4
PRINT
DROPFIRST 3
PRINT
```

Sample program output:

```
Processing command: PICKUP 6
1 engines and 5 cars added to train
Processing command: PRINT
Total number of engines: 2, Total number of cargo cars: 5, Total value of cargo:
$195000
The cars on the train are: engine - engine - oil - wheat - lumber - oil - oil
Processing command: DROPLAST 1
1 cars dropped from train
Processing command: PRINT
Total number of engines: 2, Total number of cargo cars: 4, Total value of cargo:
$135000
The cars on the train are: engine - engine - oil - wheat - lumber - oil
Processing command: PICKUP 8
0 engines and 8 cars added to train
Processing command: PRINT
Total number of engines: 2, Total number of cargo cars: 12, Total value of cargo:
$410000
The cars on the train are: engine - engine - oil - wheat - lumber - oil - oil -
lumber - wheat - oil - oil - wheat - lumber - oil
```

```
Processing command: DROP oil 4
4 cars dropped from train
Processing command: PRINT
Total number of engines: 2, Total number of cargo cars: 8, Total value of cargo:
$245000
The cars on the train are: engine - engine - wheat - lumber - lumber - wheat - oil -
wheat – lumber - oil
Processing command: DROPFIRST 3
3 cars dropped from train
Processing command: PRINT
Total number of engines: 2, Total number of cargo cars: 5, Total value of cargo:
$170000
The cars on the train are: engine - engine - wheat - oil - wheat - lumber - oil
End of processing.
```

Additional Notes

- You may assume that the input file does not contain any errors. All commands will be valid commands, and the format of each line in the file will be as shown above.
- Use the above input as a starting point. The file “trainInput.txt” contains additional test input.
- Place all classes for question 1 in the same file.

[Programming Standards are worth 8 marks]