CS 251 Summer 2020
Project 3: Sorting and Search Trees
Due date:

General Guidelines

The APIs given below describe the required methods in each class that will be tested. You may need additional methods or classes that will not be directly tested but may be necessary to complete the assignment. Keep in mind that anything that does not *need* to be public should generally be kept private (instance variables, helper methods, etc.).

Unless otherwise stated in this handout, you are welcome (and encouraged) to add to/alter the provided java files as well as create new java files as needed.

In general, you are not allowed to import any additional classes in your code without explicit permission from your instructor!

Note on academic dishonesty: Please note that it is considered academic dishonesty to read anyone else's solution code to this problem, whether it is another student's code, code from a textbook, or something you found online. You MUST do your own work! You are allowed to use resources to help you, but those resources should not be code, so be careful what you look up online!

Note on implementation details: Note that even if your solution passes all test cases, if a visual inspection of your code shows that you did not follow the guidelines in the project, you will receive a 0.

Note on grading and provided tests: The provided tests are to help you as you implement the project. The Vocareum tests will be similar but not exactly the same. Keep in mind that the points you see when you run the local tests are only an indicator of how you are doing. The final grade will be determined by your results on Vocareum.

Project Overview

This project has two major programming parts that are relatively unrelated to each other, each worth 50 points. In Part 1, you will implement solutions to two different sorting problems. In Part 2, you will implement parts of two different search trees.

Part 1. Sorting (50 points)

A. *SortMerge.java* (24 points)

In this part, you will implement a class called *SortMerge*, which uses a variation of MergeSort to sort an array. In the typical *MergeSort* algorithm, an array is recursively divided into halves, which are then recursively sorted and merged together. In this variation, we expect that many of our input arrays are made up of smaller, already sorted arrays, and we seek to take advantage of that by finding the sorted subarrays and merging them together.

Example:

Input: [2, 5, 8, 10, 1, 3, 6, 7, 9, 11, 2, 4, 4, 5, 6, 1, 2]
This array is made up of 4 sorted subarrays: {2, 5, 8, 10}, {1, 3, 6, 7, 9, 11}, {2, 4, 4, 5, 6}, and {1, 2}. With regular MergeSort, the array would be divided by half for each recursive call, regardless of the actual sorted subarrays. In our version, *SortMerge*, we will take advantage of these subarrays and merge them together, following these steps:

1. Find the subarrays (starting and ending indices for each one). This should take no more than $O(N)$ time where N is the input size.
2. Merge pairs of subarrays together until the whole array is sorted.

Using the example above, this would look like:

1. Scan the array to find the starting and ending indices for each subarray. These are: (0, 3), (4, 9), (10, 14), and (15, 16).
2. Merge the subarrays together in pairs until the whole thing is sorted.
    a. Merge the first two subarrays together creating a new sorted subarray from index 0 to index 9: {1, 2, 3, 5, 6, 7, 8, 9, 10, 11}
    b. Merge the third and fourth subarrays together creating a new sorted subarray from index 10 to index 16: {1, 2, 2, 4, 4, 5, 6}
    c. Merge the two new subarrays together, sorting the entire array: {1, 2, 2, 2, 3, 4, 4, 5, 5, 6, 6, 7, 8, 9, 10, 11}

Implementation Requirements

- The runtime for your entire implementation should be no worse than $O(NlogN)$.
- The space usage for your implementation should be no worse than $O(N)$. (Hint: You will probably want to use an extra array of size N.)
- You may find it useful to use an extra data structure for keeping track of the indices of the subarrays that are to be merged. One way of doing this is with a Queue, so a *Queue.java* class is provided for you. Note that you are not required to use this, but if you do not, your implementation still has to meet the overall requirements of the project (including not importing unapproved classes!)

API for *SortMerge.java*

Skeleton code for *SortMerge.java* is provided for you. You must implement the following method.

`static void sort(int[] array)`	`sorts array from smallest to largest in the method described above`

Testing

We provide *SortMergeTest.java* to help you test the accuracy and robustness of your solution. Keep in mind that this test may not be exhaustive, and you should consider adding more robust testing to ensure that you handle all possible inputs correctly and efficiently

Grading Rubric for Part 1.A

Item	**Description**	**Points**
Accuracy Tests	Test that the implementation produces correct results (array is sorted and contains the same elements that it started with).	15
Robustness Tests	Tests that the implementation is efficient by running it on large input sizes.	9

B. *SortGrid.java* (26 points)

In this part, you will implement *SortGrid.java*, which sorts the elements in an $N \times N$ grid, represented as a 2-dimensional array.

Example:

Input:

4	1	0	2	8
10	3	24	2	8
9	0	37	29	3
43	22	7	11	0
5	67	3	2	0

Output:

0	0	0	0	1
2	2	2	3	3

3	4	5	7	8
8	9	10	11	22
24	29	37	43	67

It is up to you to figure out a solution to this problem according to the requirements below. We will test your solution for accuracy (i.e. does it sort the grid?) and for robustness. Points will be awarded based on accuracy *and* efficiency, so you should design your solution with that in mind. Keep in mind that some algorithms may be efficient on small grids but blow up on larger grids, so be sure to test your solution on many different input sizes.

API for *SortGrid.java*

Skeleton code for *SortMerge.java* is provided for you. You must implement the following method.

`static int sort(int[][] grid)`	`sorts grid; returns the number of compares done by the algorithm`

Implementation Requirements

- For this part, you *must* use the starter code provided for you in *SortGrid.java.* We will grade the efficiency of your algorithm based on the *number of compares* it uses. To this end, every time you compare two elements in the grid, you must use either *lessThan* or *greaterThan*, which are provided for you. (Note that you can test $\leq$ with $!greaterThan$ and $\geq$ with $!lessThan$.)
- DO NOT make any changes to the *lessThan* or *greaterThan* code. If you do, this will be considered academic dishonesty, and you will receive a 0 for this part of the project. Additionally, if you attempt to trick the grader by not using the provided comparison functions, you will receive a 0 for this part of the project.
- The *sort* method should start by setting *compares* to 0 and should return *compares.* This is already done in the code. Any changes to this will be considered academic dishonesty and will result in a 0 for this part of the project.
- There is also a *swap* method provided for you.
- **You are allowed to use an extra copy of the grid if necessary for your solution. However, no other data structures are to be used (i.e. no extra arrays or lists, etc.)** Any use of extra data structures without express permission will be considered academic dishonesty and will result in a 0 on the part of the project.

Testing

We provide *SortGridTest.java*, which checks your solution for accuracy and efficiency. We also provide a few specific test cases that will give you an indication of how your solution holds up with regards to efficiency.

Grading Rubric for Part 1.B

Item	**Description**	**Points**
Accuracy Tests	Test that the implementation produces correct results (array is sorted and contains the same elements that it started with).	14
Robustness Tests	Tests that the implementation is efficient by running it on various input sizes. Points will be awarded based on the number of compares. For each test, there will be different ranges, and points will be awarded based on which range your solution falls into.	12

Part 2. Search Trees (50 points)

A. *BST.java*

In this part, you will implement the *insert* and *get* functions for a binary search tree. In this BST, we order the nodes so that the key of Node *n* is greater than all keys in its left subtree and less than all keys in its right subtree. You should use the skeleton code that is provided and your *insert* and *get* functions should be implemented so that all the other functions that are provided work correctly as well (specifically, the functions related to *height* and *size*.) **Do not alter any of the provided inner classes or methods.**

B. *RLRBT.java*

In this part, you will implement the *insert* and *blackHeight* functions of a right-leaning red-black tree. To get started, you should copy over the *get* function you wrote for BST.java, as it should work for both trees. It is also recommended that you copy over the *insert* function that you wrote for BST.java, as it should help as a basis for the right-leaning red-black tree implementation.

A right-leaning red-black tree is a binary search tree that maintains black balance by ensuring that the following properties are upheld after each insert (and delete).

- The root is black.
- Red nodes have black children.
- No node can have more than one red child.
- All red links lean to the right.

In other words, it's the opposite of a left-leaning red-black tree. It is up to you to study the cases for a left-leaning red-black tree and determine the corresponding cases for the right-leaning tree. The good news is that the transformations used are the same as for the left-leaning

tree--*rotateRight, rotateLeft,* and *colorFlip*. These are provided for you. The *Node* class is also updated with color information.

C. *TreeSim.java*

The code for *TreeSim.java* is provided for you. It runs a simulation to compare the worst case inputs for BST and RLRBT and the average cases. You do not have to implement any part of this, but your code needs to run with it. Also, you should run it several times to make sure that your output *makes sense*. It is up to you to determine this based on what you know about BSTs and RBTs. Take time to look through *TreeSim* to understand what is happening and determine if your output *makes sense*. This part will be graded by a manual inspection of your output.

Grading Rubric for Part 2

Item	**Description**	**Points**
Tests for BST	Test that the implementation produces correct results. Also tests robustness.	20
Tests for RLRBT	Tests that the implementation produces correct results. Also tests robustness.	20
Simulation	Builds large trees out of large amounts of random data and compares the resulting heights.	10

Submission

Submit ONLY the following files.

- *SortMerge.java*
- *SortGrid.java*
- *BST.java*
- *RLRBT.java*