Module 11 Assignment: Divide & Conquer Sorts

Marcar esta página

Coding Assignment fecha límite Oct 15, 2023 11:43 -05

Divide & Conquer Sorts

For this assignment you will be coding 2 different sorts: **Merge Sort** and **LSD Radix Sort**. For merge sort, the autograder will be looking at the number of comparisons made between elements to test for efficiency.

For each sorting algorithm, you may assume that the array you are sorting will not contain null elements. You should also assume that the array may contain any number of duplicate elements.

Your implementations should match what was taught in this course.

IMPORTANT:

- **You will be given 5 attempts on this assignment, with a 30 minute cooldown between submissions.**
- **Please run your code before each submission to ensure that there are no formatting errors! If there are formatting errors in your code, your code will not be graded and a submission attempt will be logged. For more information, please review the Vocareum overview below.**

Comparator

Merge sort will take in a Comparator, which is used to compare two elements. You **must** use this Comparator, as the number of comparisons performed with it will be used when testing your assignment. The comparator is used as follows:

`comparator.compare(A, B)` will return:

- < 0 if A is less than B
- > 0 if A is greater than B
- 0 if A is equal to B

Generic Methods

Most of the assignments is this course so far have utilized generics by incorporating them into the class declaration. However, the rest of the assignments will have you implement various algorithms as static methods in a utility class. Thus, the generics from here on will use generic methods instead of generic classes (hence the **<T>** in each of the method headers and javadocs). This also means any helper methods you create will also need to be static with the same **<T>** in the method header.

Merge Sort

Merge sort should be out-of-place, stable, and not adaptive. It should have a worst case running time of $O(nlogn)$ and a best case running time of $O(nlogn)$. When splitting an odd size array, the extra data should go on the **right**.

LSD Radix Sort

LSD Radix sort should be out-of-place, stable, and not adaptive. It should have a worst case running time of $O(kn)$ and a best case running time of $O(kn)$, where k is the number of digits in the longest number. You will be implementing the least significant digit version of the sort. You will be sorting ints. Note that you CANNOT change the ints into Strings at any point in the sort. The sort **must** be done in base 10. You may use `Math.abs()` when finding the largest magnitude number. However, be wary of handling overflow if you use `Math.abs()`!

General Tips

- For LSD Radix sort, you cannot take the absolute value of Integer.MIN_VALUE, as it does not fit in a Java int. How should you account for this edge case when initially searching for the largest magnitude number?

- We highly recommend copying the starter code and working in your preferred IDE in order to have access to features such as code completion, auto-formatting, and much more!

Here are general assignment guidelines that should be followed.

- Do not include any package declarations in your classes.
- Do not change any existing class headers, constructors, instance/global variables, or method signatures. For example, do not add throws to the method headers since they are not necessary. Instead, exceptions should be thrown as follows: `throw new InsertExceptionHere("Error: some exception was thrown");`
- All helper methods you choose to write should be made private. Recall the idea of Encapsulation in Object-Oriented Programming!
- Do not use anything that would trivialize the assignment. (e.g. Don't import/use java.util.ArrayList for an ArrayList assignment.)
- Always be very conscious of efficiency. Even if your method is to be $O(n)$, traversing the structure multiple times is considered inefficient unless that is absolutely required (and that case is extremely rare).
- If applicable, use the generic type of the class; do not use the raw type of the class. For example, use `new LinkedList<Integer>()` instead of `new LinkedList()`.

Use of the following statements should be avoided at all times.

package	System.arraycopy()	clone()
assert()	Arrays class	Array class
Thread class	Collections class	Collection.toArray()
Reflection APIs	Inner or nested classes	Lambda Expressions

The Vocareum (code editor) interface has six main components:

- The **Drop-Down** in the top left. This lets you choose from multiple available files. Note that this drop-down will only be visible in assignments that require multiple files.
- The **Run** button. This will compile your code and run a file scan. Running your code will not count towards your total allowed submission attempts, therefore you are free to run as many times as needed.
- The **Submit** button. This will compile your code, run a file scan, grade your assignment, and output results to console. Note that for most assignments in this class, you will only be allowed a limited number of submissions. A submission is counted when the submit button is clicked, regardless of whether or not your code can compile or if there are any file issues. Therefore, we **highly recommend** that you run your code before submitting to ensure that there are no issues that will prevent your code from being graded and that every submission attempt will generate meaningful results.
- The **Reset** button. This will revert all your changes and reset your code to the default code template.
- The **Code Window**. This is where you will write your code. For large coding assignments, we highly recommend copying the starter code and working in your preferred IDE to have access to features such as code completion, auto-formatting, and much more!
- The **Output Window**. This window will appear whenever you run or submit your code and will display the output for you to view.

For additional help, please visit the Vocareum information page located in the course information module!

< Anterior   Siguiente >