1 Assignment 2

Submit source code and running instructions to EAS[1]. Do it in Java. Submit any textual/diagram components in a single word/pdf file, providing the assignment numbering to help your marker. **Do not** use java's search or sort methods! **Do not** use Java's Collections or Subclasses, code your own. **Do** use Java's parseInt, Random and InputStream/Reader implementations!

Please put all files in the default package. This isn't good design, but it *is* easier to mark.

This assignment should use Command-Line. Not Scanner, not anything else. Command-Line input. Learn what String[] args are for in the main method! Similar, but a bit smarter, to what you may have used in C++.

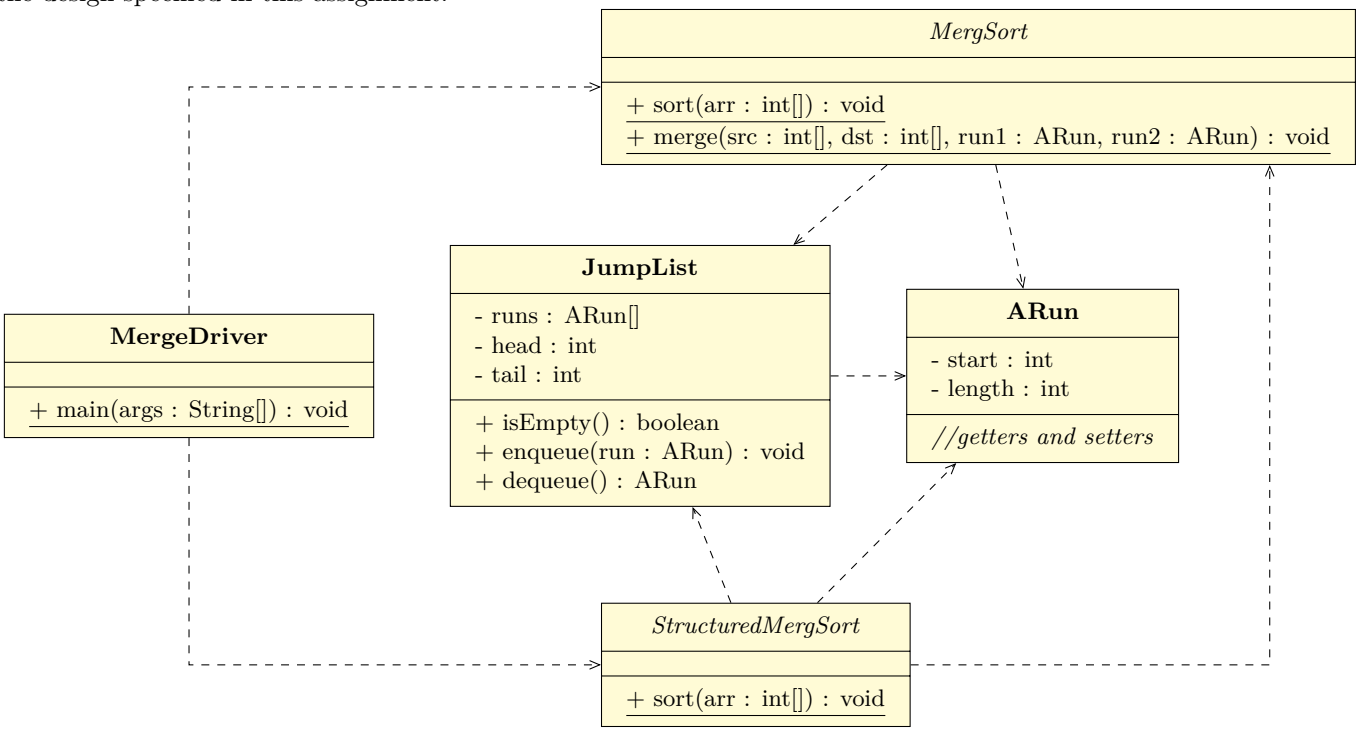
Code this yourself. I am surprised that I have to say this, but do not start with code from the book or code from the Internet, and most importantly, do not start with code from your friends. Ask theory questions. Ask your tutors (and myself) technical questions once you've explored the problem a bit. This will teach you to implement solutions that fit arbitrary problems.

Posted: Monday, July 9^{th}
Due: Sunday, July 22^{nd} **by 11:59pm**
Grade: 5%

1.1 Design

A structuring merge sort applies simple structuring to input data in order to merge it more quickly. By structuring, we mean the detection and organization of runs so that all runs are in sorted order. By comparison, a non-structuring merge sort will not take advantage of existing structure, and will order a piece at a time (considering a trivial array-based implementation).

In this assignment, you will implement both a trivial array-based merge sort, and then a structuring merge sort according to the design specified in this assignment.



MergeDriver

MergeDriver will act as the driver for your application. The main method will take string arguments starting with either "MS" or "SMS", calling the sort method from either MergeSort or StructuredMergeSort respectively. If more than one additional argument is detected, it will assume a sequence of integers and generate an array to be sorted from that list. A single additional argument will be interpreted as the size of an array of integers to randomly generate (between 0 and MAX_VALUE, e.g. java.util.Random.nextInt(Integer.MAX_VALUE)).

e.g.

java MergeDriver <SortType> <length>

or

java MergeDriver <SortType> <value1> <value2> <value3> <...>

[1] https://fis.encs.concordia.ca/eas/

MergeSort

MergeSort has two important static methods, sort and merge. The sort method takes the array to be sorted and must prepare an initial buffer array to work with (it is not in-place), prepare runs and then call merge as needed. You want to be wary of the use of space and extra operations to maintain the expected complexity ($\Theta(n)$ space complexity and $\Theta(n \log n)$ average comparison complexity).

The merge method will be shared by both MergeSort and StructuredMergeSort. merge will take the source array, the buffer array and and the two run indices as well as their lengths. The two runs specified will be merged from the source array into the corresponding locations in the destination array.

There are many ways to perform the generic merge sort expected here. This implementation should merge every two values, then every four values, then every 8 values and so forth until it is all merged. That means there will likely be merging of big to small runs when not close to a power of 2 elements. That's ok, I just want you to explore doing it this way in the assignment, even if the other way arguably makes more sense.

You can implement other methods as needed.

StructuredMergeSort

StructuredMergeSort will have its own static sort implementation that differs from MergeSort, where it will run a structuring pass and prepare runs as appropriate before calling MergeSort's merge method (as many times as needed). Note that because we're using static methods, there is no inheritance (and in fact you should not make instances of either sort).

There are many ways to perform the merge sort expected here. This implementation should merge adjacent runs until all runs have been merged into the buffer, left to right, before repeating the process back again, continuing until everything is merged (and back in the source array). That means there will likely be some runs without a run to merge into, so just copy it over. Does that seem wasteful? There's interesting stuff that you could do, but leave that out of the assignment for simplicity's sake.

You can implement other methods as needed.

JumpList

Both implementations will make use of a "Jump List", an array-based circular queue that will hold runs and their lengths. How the "Jump List" is filled and when runs are merged may differ between MergeSort and StructuredMergeSort.

ARun

This is just a structure. It does not contain any useful behavior, but is a convenient representation of the runs that we want to work with in our Merge Sorts. Would you ever implement something like this? Perhaps...

1.2 Questions

1. MergeSort and StructuredMergeSort

   (a) Follow the class diagram above for class naming conventions.

      i. Your driver file should be called MergeDriver.java

      ii. This file should output the run-time in either *ns*.

      iii. it should accept command-line as follows:
      java MergeDriver <SortType> <length>
      or
      java MergeDriver <SortType> <value1> <value2> <value3> <...>

         A. <SortType> is either MS or SMS

         B. <length> is the number of ints to be randomly generated and then sorted in the input array

         C. <valueN> is just an arbitrary number if inputs (which should represent integers) that can be passed on the command-line to represent the input array to be sorted

   (b) Record performance times of runs for interesting input sizes for both implementations.

MergeSort Sort Example

```
%java MergeDriver MS 12 6 2 72 62 1 43
sorting: 12 6 2 72 62 1 43
comparisons: 14
result: 1 2 6 12 43 62 72
completed in 121555ns
```

```
%java MergeDriver SMS 12 6 2 72 62 1 43
sorting: 12 6 2 72 62 1 43
comparisons: 15
result: 1 2 6 12 43 62 72
completed in 341212ns
```

2. In clear, natural language, describe the performance differences between the two sorts. Try to correlate this with the underlying mechanism.

   (a) This textual response should be no more than 7 lines / 70 words.

3. In clear, natural language, describe how the best-case varies between the two sorts, indicating when one might work better.

   (a) This textual response should be no more than 7 lines / 70 words.

4. In clear, natural language, describe the use of JumpList in either implementation, focusing on asymptotic complexity considerations and how useful it was for either approach? I don't want to know what a better approach is, I want to know how best you were able to apply it in either approach and what that meant.

   (a) This textual response should be no more than 10 lines / 100 words.