EECS 233 Programming Assignment #4

Due April 22, 2020 (before 11:59pm)

80 points

In this assignment, you will implement an improved mergeSort I mentioned in class. Our implementation of merge-sort in the lecture notes involved merging pairs of sub-arrays into a temporary array and then copying the merged array of double size back into the original place in the input array at the end of each merging phase. Please write your implementation in a way that avoids this back copying. You still need the temporary array, but instead of copying the output run from the temporary array back into the input array and always merging from the input array into the output array, you will alternate: one merging phase will merge runs from the input array into the temp array and the next merging phase will merge runs from temp array back into the input array, and so on.

The method takes as arguments an integer array, which then becomes sorted in the increasing order of elements:

- void mergeSort(int [] arr).

Please make this a static method of a Sorting class (as you would often with utility functions – we discussed this in class when we started our module on sorting).

After implementing the method, measure its execution time on random integer arrays of different size: 250,000, 500,000, 1,000,000, and 1,250,000 elements. (See tips below on how to generate random numbers.). While we know theoretical running time of this method from our algorithm analysis, experimental measurements are still useful to make sure your implementation did not introduce some inefficiencies that the algorithm analysis has not reflected.

Running the same program on a random input can take different time due to random input that happens to produce some edge cases, and even on the same input, the program execution can take different time due to random scheduling and parallel activities. Therefore, we usually obtain each measurement multiple times (using different randomly generated arrays each time) to characterize the performance of the method in question. Since many of you have yet to take statistics, in this assignment, please use a simplified approach: run your method three times for each input array size, each time with different randomly generated array and use the median value[1] as the result for the given array size. This is useful to show that your results are not a function of some lucky input data.

You need to write two programs to test your mergeSort.

The **"runtime"** program is to test the running time of your sorting method. This program takes no arguments and executes your sorting function 12 times on 12 random arrays respectively (three different arrays for each input size), records the execution time of these methods, prints out the results (you will have 4 numbers, one per array size, reflecting the median value of the three execution times for this size). **Important (and keep this in mind as a general programming consideration):** Make sure you don't create 12 different arrays and assign them to 12 different array variables. If you do it, you would "hoard" a large amount of memory by holding all these arrays in memory at the same time. Instead, reuse *the same* array variable and when creating different arrays, assign them to this one variable. This way, you create one array, populate it with random numbers, work with it, and once done, you create the next array and assign it to the same array vaiable – allowing the previous array to be garbage collected since there are no longer references in your program pointing to it.

The **"fnTest"** program is to test the functionality of your implementation of the sorting methods. We will be testing your work by typing

[1] The median of a set of values is the value such that half the set are less than or equal to this value and half the set is greater or equal to this value. I.e., if x1 <= x2 <=x3, then x2 is the median of the three values.

```
java fnTest <input_file_name>  <output_file_name>
```

on the command line, for example, "`java fnTest inFile.txt outFile.txt`". This program must accept the names of an input and output files as arguments. It will read the input file into an array then invoke your mergeSort method and write out the sorted array into the output file.

File format: both the input and output files are text files that have one integer number per line so you can just read it line by line and populate your array in memory. **Please make sure to use this format in your program.**

Tips:

1. Implement the merge-sort without recursion, taking care of the arrays whose size is not a power of two (I don't know how to implement this enhanced merge-sort recursively).

2. You can use `java.util.Random` library to generate random integers in Java. First create the random number generator object by calling a constructor, e.g., `Random randNums = new Random();` and then repeatedly call `randNums.nextInt()` to produce random integers.

3. To measure time at a fine granularity, use System.nanoTime() call:

   ```
   long startTime = System.nanoTime();
   // ... the code being measured ...
   long estimatedTime = System.nanoTime() - startTime;
   ```

4. For accurate measurements, make sure your starting and ending timestamps include as little extraneous processing as possible, and especially avoid expensive operations such as I/O. In particular, make sure you are not including reading any files from disk into the input array or printing any output or, in your case, creating random arrays.

Deliverables:

- All source codes, including the Sorting class with the sortMerge method, and the runtime and fnTest programs.
- Output of runtime program for the random arrays of four specified sizes.
- Result of running fnTest program on a small (e.g., 10-number) input file. Include both the input file and the output file used, with understandable file names (e.g., inTestFile and outTestFile).

Grading:

- The sorting method: 50 pts
  - including 20pts for avoiding back-copying on each iteration.
- The testing code: 20pts
  - `runtime` program (automatic execution of all required runs, economical use of memory, proper timestamping for performance measurement): 15 pts
  - `fnTest` program: 5 pts
- Programming style, comments, clarity: 10 pts