Koc University
COMP202
Data Structures and Algorithms
Assignment 5

Instructor: Gözde Gül Şahin
Authors: Önder Akaçık, Burak Can Biner
Due Date: 24.05.23
Submission Through: BOTH Blackboard and Github Classroom

This programming assignment will test your knowledge and your implementation abilities of what you have learned in the Sorting parts of the class.

This homework must be completed individually. Discussion about algorithms and data structures are allowed but group work is not. Any academic dishonesty, which includes taking someone else's code or having another person doing the assignment for you will not be tolerated. **By submitting your assignment, you agree to abide by the Koc University codes of conduct.**

Description

This assignment requires you to implement various sorting algorithms. These algorithms include **insertion sort**, **heap-sort**, **quick-sort**, **counting sort**, and **tim sort** (as an application of insertion sort and merge sort).

The files provided to you have comments that will help you with implementation.Keep the slides handy as they include the pseudo-codes for the algorithms. The file descriptions are below. Bold ones are the ones you need to modify and they are placed under the *code* folder, with the rest under *given*.

- *AbstractArraySort.java*: This file describes the abstract sorting class which the other sorting algorithms extend. It defines the sort method which is the primary function you should overwrite. It also has the swap and compare methods that you need to call in the concrete classes whereever applicable. Hint; not all algorithms utilize swap. It also has other utility methods which might be of use in debugging.
- *JavaArraySort.java*: A wrapper class for the Java's default sorting method. You can take a look if interested.
- ***InsertionSort.java***: Implement the insertion sort algorithm in this file. Remember to use the swap and compare methods from the abstract parent class wherever applicable.
- ***HeapSort.java***: Implement the heap-sort algorithm in this file. Remember to use the swap and compare methods from the abstract parent class whereever applicable. Make sure to fill out the heapify method which will also be graded.
- ***QuickSort.java***: Implement the quick-sort algorithm in this file. Remember to use the swap and compare methods from the abstract parent class whereever applicable. Make sure to fill out the partition method which will also be graded. You have two options for this, either version which returns an indexPair object to return two indices or comment this out and uncomment the version which returns an integer.

- ***CountingSort.java***: Implement the counting sort algorithm in this file. Note that you need to do this only for integers. You do not have to use any function from its parent in this class.

- ***TimSort.java****:* Implement the tim-sort algorithm in this file.Remember to use the swap and compare methods from the abstract parent class whereever applicable. Make sure to fill out the merge method which will also be graded.

  Timsort is a hybrid sorting algorithm that combines elements of both insertion sort and merge sort. It was invented by Tim Peters in 2002 and is used in many popular programming languages and systems. It is the standard sorting algorithm for Python and Java Arrays.sort(). It is basically a combination of insertion sort and merge sort algorithms. To sort an array using Timsort, we first divide the array into smaller blocks called runs. Each run is sorted individually using an insertion sort algorithm. Once all the runs have been sorted, they are merged together using a modified merge sort algorithm. During the merging process, adjacent runs are compared and combined into larger sorted runs until the entire array is sorted. For further information, please refer to: Timsort.

  Timsort uses various optimization techniques. We do not expect you to implement a perfectly optimized version of this algorithm. We expect you to implement the given Timsort algorithm below.

  1. Divide the input array into smaller blocks. Block sizes should be 32.
  2. Sort each block using insertion sort separately.
  3. Merge two adjacent blocks into one by using a modified Merge Sort algorithm. Once you merge each block new block size will be doubled (e.g. for first time we merge adjacent blocks, block size will be 64) and number of blocks will decrease to half.
  4. Now repeat the previous step 3 until the entire array is sorted. (Step 3 and step 4 is also shown in the Figure 1).

- *SortDebug.java*: This file has a smaller main function which you can use to debug individual sorting algorithms. We highly recommend that you test your implementations here first.
- *SortGrade.java*: This file grades your sorting implementations with different data distributions and data sizes. It checks aforementioned methods for certain algorithms and tests whether your implementations sort the data correctly. In addition to integers, doubles and strings are also utilizied. The code also checks for number of swaps and compares for three algorithms. Small implementation details might result in slightly different numbers. If you are getting sorted elements but swaps and compares do not match, make sure to finish the rest of the homework and come back to the issue.
- *DataGenerator.java*: This file has the code to generate data to test your implementations. Take a look at it if interested but you do not need to worry about it.

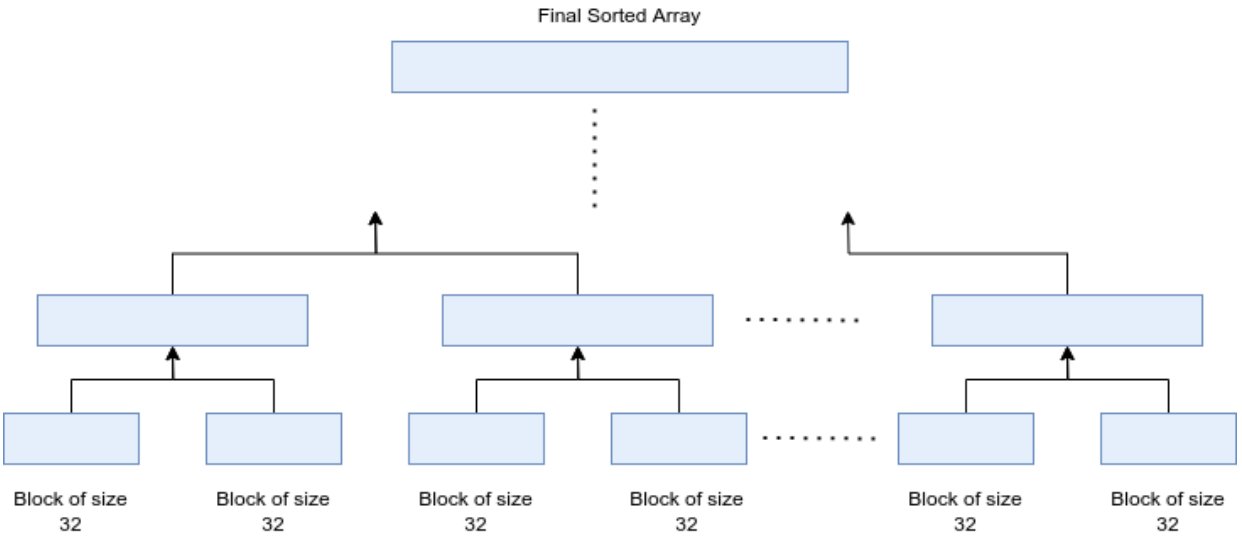

Figure 1: Timsort Algorthim Merging

Grading

Your assignment will be graded through an autograder. Make sure to implement the code as instructed, use the same variable and method names. A version of the autograder is released to you. Our version will more or less be similar, potentially including more tests.

Run the main program in the *SortGrade.java* to get the autograder output and your grade. In case the autograder fails or gives you 0 when you think you should get more credit, do not panic. Let us know. We can go over everything even after your submission. Make sure that your code compiles!

Submission

• Make sure to remove all the unused imports and that you have the original package names. What compiles on your machine may not compile on ours due to simple import errors. Code that does not compile will receive a grade of 0.

• You are going to submit a compressed archive through the blackboard site and to github classroom. The file you submit to blacboard can have zip, tar, rar, tar.gz or 7z format.

• The compressed file should have all the files that you need to modify. If anything is missing, we automatically copy over the default versions.

• The compressed file should not contain another compressed file.

• Once you are sure about your assignment and the compressed file, submit it through Blackboard.

• After you submit your code, download it and check if it is the one you intended to submit.

• Do not submit code that does not terminate or that blows up the memory.