AAiT
Algorithm Analysis and Design Assignment #1

Prepared by: Yonas Y. (yonas.yehualaeshet@aait.edu.et)

Submission Deadline: 17:00 on January 04, 2019

Instructions:

- Implement the different sorting algorithms given in the question part using C++, Java or Python programming language.
- Clearly explain if you make any assumptions.
- Submit both the soft copy (sourse codes and report) and hard-copy (report) before the stated deadline.
- Submission after the stated deadline is not acceptable, even with any excuse.
- In the implementation make sure that the files are organized in a reasonable way, and that it does not require a wizard to make the implementation work outside your personal environment.

Questions :

Implement the following algorithms

1. Insertion Sort
2. Merge Sort
3. Heap Sort
4. Quick Sort
5. Selection Sort

The testing parameters for the algorithms that shall be used are:

- values $start$ and $stop$ that determine the minimum and maximum number of entries to be sorted,
- a value steps that determines into how many intervals the range from start to stop shall be divided (see Figure 1 for an example where $start = 0$, $stop = 2,000,000$, $presortedness = 0.25$ and $steps = 20$), and
- an input array to be sorted.
  - To create this array, use a **presortedness** parameter between 0 and 1.
    - **presortedness = 0** should yield very low order (i.e., elements ordered reversely),
    - **presortedness = 1** should correspond to total order (i.e., nothing gets mixed up at all).
    - **presortedness = 0.5** should correspond to a random disorder (i.e., elements get mixed up randomly).

	Sorter A	Sorter B	Sorter C	Sorter D	Sorder E
0	9	12	20	12	11
100000	12093	12748	11374	8597	7594
200000	26213	28404	23526	17994	16042
300000	41541	43116	36216	27766	24754
400000	57432	59706	49123	37654	33532
500000	73667	77051	62368	47608	42512
600000	90078	92728	75901	57851	51775
700000	107490	109480	89391	68291	61118
800000	125219	126169	101949	78758	70547
900000	143597	144369	115692	89242	80152
1000000	162833	159360	129119	99846	89457
1100000	183864	178579	143106	110704	99413
1200000	206267	198708	157776	121135	108884
1300000	229888	214230	171688	131899	118623
1400000	253556	233266	186779	142735	128394
1500000	277791	251697	200687	153933	138663
1600000	303894	269136	214126	164484	148061
1700000	329817	288343	229304	175913	158385
1800000	356927	306674	243766	186671	168111
1900000	383737	326219	257804	197568	177867
2000000	411521	343627	272232	208660	188016

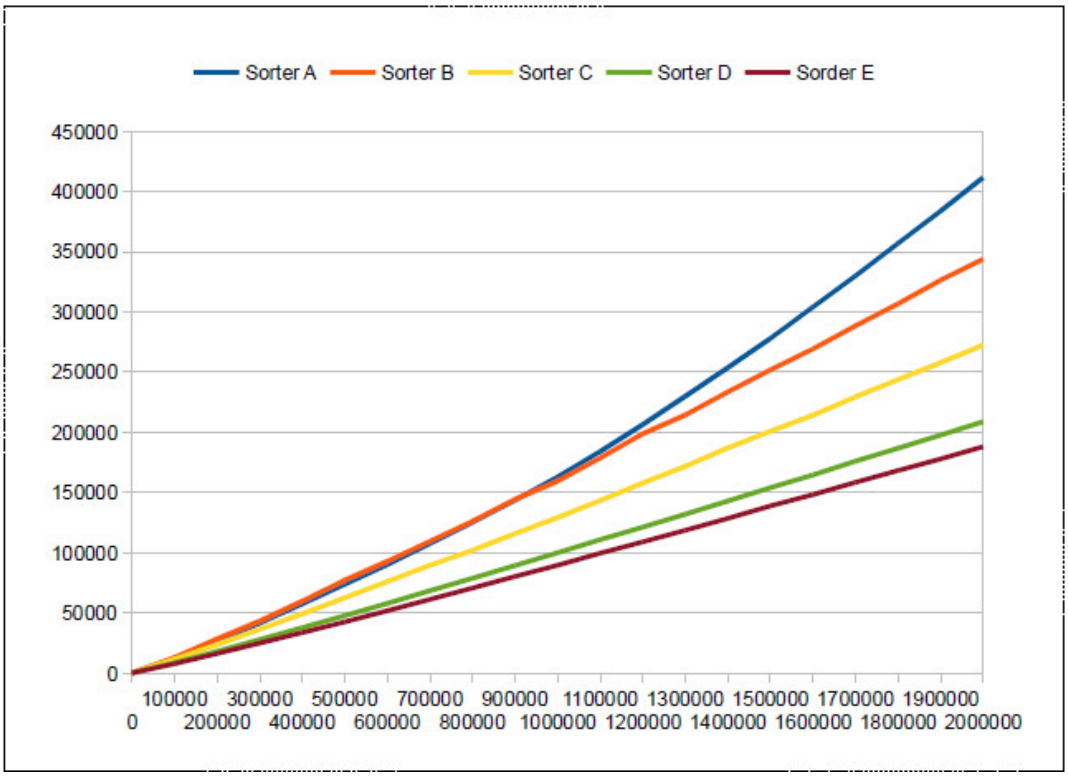

Figure 1: Sample test output.

- Use presortedness values of $\{0, 0.5, 1\}$.

In addition, there should be a constant (that can of course be altered) called rep.

- It determines how many times the sorters are run on arrays of a each size (with different array contents as input, of course). This number should not be too small, in order to make the tests less prone to random fluctuations caused by an unlucky choice of input data.

The implementation should be done in such a way that that the running times of the sorters are

measured as accurately as possible. In particular, pay attention to the following:

- Search for good ways to measure elapsed time in the programming language of your choice. Select the one which is most suitable and make a few tests to ensure that it works as expected.
- Avoid unnecessary memory allocation, especially repeated allocation of large input arrays. Instead, use and re-use sufficiently large arrays.
- For the sake of fairness, let all sorters run on the same input arrays, i.e., populate an array with random integers and apply all sorters to these data before turning to the next round. (Obviously, you need two arrays for this, one holding the original data and a copy to which the sorters are applied.)
- During your test runs, try to make sure that the workload caused by other processes in your computer is as low as possible.

In addition, take the following points into consideration when implementing the algorithms

- In the sorter implementations, avoid wasting time by unnecessary comparisons, assignments or parameter passing. Since we want to find out how efficient the algorithms are, they should not be implemented in a sub-optimal way, because otherwise comparisons are meaningless.
- Note that textbook psuedocode versions sometimes can be tuned a bit, as they sometimes trade efficiency for conciseness of the description.
- If the tests reveal a strange behavior of some or all sorters, study your implementation to find the reason and improve it.

Finally, prepare a report which discus's whether the results of the implementations confirm your expectations and coincide with the theoretical results. If you discover discrepancies, try to reason out why it occurred.