

Assignment 2 – Sorting Algorithms

Deadline: Tuesday, February 14, 2023, 11:59 pm

Objectives:

- Familiarize yourself with the basic sorting algorithms (i.e., implementation and performance)
- To implement several sorting algorithms and compare their relative performance. The sorting algorithms you will consider are bubble sort, insertion sort, selection sort, shellsort, quicksort, and mergesort

Instructions:

For problems marked as **(code)** – You should write a Java program that will be run by the instructor

For problems marked as **(text)** – You should write an answer. The answer can be either prose, an algorithm (pseudo-code or code), a proof, or mathematical equations. Whichever form of text answer is appropriate to solve the problem. In case the answer is an algorithm (code or pseudo code), it will be assessed mostly on logic rather than on whether it compiles.

Problems:

1 (text) [2 points] Show the step-by-step process of merging two sorted lists in using the mergesort algorithm. The sorted lists are: [2, 4, 6, 10] and [1, 7, 16, 32]

2 (text) [2 points] Show the step-by-step process of sorting a list using the insertion sort algorithm. The unsorted list is: [-2, 5, 6, -10, 1, 98, 4, 1]

3 (text) [2 points] Show the step-by-step process of sorting a list using the quicksort sort algorithm. The unsorted list is: [5, 14, 26, 10, 1, 5, 16, 0]

4 (text) [2 points] Show the step-by-step process of sorting a list using the shell sort algorithm. The unsorted list is: [5, 14, 26, 10, 1, 5, 16, 0]

5 (text) [4 points] Rank the six sorting algorithms in order of how you expect their runtimes to compare (fastest to slowest). Your ranking should be based on the asymptotic analysis of the algorithms. You may predict a tie if you think that multiple algorithms have the same runtime. Provide a brief justification for your ranking

6 (code) [13 points] Sorting algorithms. Implement bubble sort, insertion sort, selection sort, shellsort, quicksort, and mergesort to sort an array of integers. All your sorting algorithms must be their own class. All of them implementing a `SortingAlgorithm` interface that has the following method:

- `int[] sorty(int[] input);`

7 (code) [20 points] Performance testing framework. Write a `Tester` class to do a performance comparison of your sorting algorithms. Your `Tester` class should have at least the following helper functions:

- `Tester(SortingAlgorithm sa):` A constructor for the `Tester` that takes as parameter the sorting algorithm to be tested

- `double singleTest(int size)`: Should create an array of integers of the given size. Fill it with random numbers and run the sorting algorithm's `sorty` method. It should record the time it takes to sort the array
- `void test(int iterations, int size)`: Should run the `singleTest` method as many times as the number of iterations provided and print to the console the average time the algorithm takes to sort an array of the given size

8 (code) [20 points] Performance comparison. Write a `Performance` class to do a performance comparison of your sorting algorithms. Your `Performance` class should have at least the following function:

- `void main()`: A main function that runs the `Tester.test()` method with 20 iterations for each algorithm for each of the following array sizes: 100, 500, 1000, 2000, 5000, 10000, 20000, 75000, 150000 and produces a `.txt` file report with the results. Be patient, but not too patient. There should be no need to run your code for hours

Sample report

Sorting algorithm – Bubble sort

Sorted 100 elements in x ms (avg)

Sorted 500 elements in x ms (avg)

Sorted 1000 elements in x ms (avg)

Sorted 2000 elements in x ms (avg)

Sorted 5000 elements in x ms (avg)

Sorted 10000 elements in x ms (avg)

Sorted 20000 elements in x ms (avg)

Sorted 75000 elements in x ms (avg)

Sorted 150000 elements in x ms (avg)

Sorting algorithm – Quick sort

Sorted 100 elements in x ms (avg)

Sorted 500 elements in x ms (avg)

Sorted 1000 elements in x ms (avg)

Sorted 2000 elements in x ms (avg)

Sorted 5000 elements in x ms (avg)

Sorted 10000 elements in x ms (avg)

Sorted 20000 elements in x ms (avg)

Sorted 75000 elements in x ms (avg)

Sorted 150000 elements in x ms (avg)

9 (text) [7 points] Make a graph comparing the runtimes of the five algorithms. Your x-axis should be `N` (the number of elements you are sorting) and your y-axis should be the average time to sort an array of size `N`, in milliseconds. You may create this graph using Excel or any other tool of your choice. However, making the chart directly from your java code will award you extra credit.

10 (text) [7 points] Write a short summary of what you observed in your experiments. What did you notice about the relative performance of the different algorithms? Did the actual performance always match up to the ranking you predicted in part (1) based on the asymptotic analysis? Why or why not?

11 (code) [15 points] k-sorting. We say that an array `A` is `k`-sorted if it has the property that for every element `i` in array `A`, `i`'s position in array `A` is at most `k` away from `i`'s position in a sorted array of the same data. For example, the array: `[1, 4, 2, 3, 5, 7, 6, 8]` is 2-sorted because every element is within 2 positions of its "correct" position in a sorted array. Any array that is `k`-sorted is also `j`-sorted for all `j > k`.

In this part of the assignment, you will study the question of whether it is faster to sort `k`-sorted data than fully random data. You will need to write a method `generateKSorted(my_array)` that fills the array that is

passed as an argument with 10-sorted data (each element is within 10 positions of its correct position in the array).

Repeat the performance experiments from problem 5, this time with 10-sorted data instead of random data.

12 (text) [5 points] Make a graph comparing the runtimes of the six algorithms.

Write a short summary of what you observed in your experiments. Did the algorithms have the same ranking on 10-sorted data as on random data? Which algorithm(s) performed significantly differently? Why?

Hint on timing:

Inside your single test method, you will generate an array of random data of length N and time how long it takes to sort the data using the sorting algorithm. You should not include in the timing the time it takes to generate the random data. To do this record the system time before calling the sort() method, run the method to sort the data, and record the system time after calling the method.

```
long start_time = System.nanoTime();
```

```
sorting.sorty(my_array);
```

```
long end_time = System.nanoTime();
```

These values are in nanoseconds. You will need to convert it to milliseconds.

Extra credit (10 points): Make the graphs comparing the runtimes of the six algorithms on java. You can use JFreeChart is <https://www.jfree.org/jfreechart/samples.html>

Grading Rubric:

Item	Points
1. (text) mergesort [2 points] 2. (text) insertionsort [2 points] 3. (text) quicksort [2 points] 4. (text) shellsort [2 points] 5. (text) predictions [5 points]	13
6. Sorting algorithms: <ul style="list-style-type: none">• Sorting interface [1 points]• Bubblesort [2 points]• Insertion sort [2 points]• Selection sort [2 points]• Shellsort [2 points]• Quicksort [2 points]• Mergesort [2 points]	13
7. Performance testing framework Constructor [5 points] singleTest [7.5 points] test [7.5 points]	20
8. Performance comparison code [20 points] 9. Comparison graph [7 points] 10. Comparison observations [7 points]	34
11. k-sorting code [15 points] 12. k-sorting code [5 points]	20
Extra credit: Graphs done in Java [10 points]	
Total Points	100

Deliverables:

A zip file containing (1) the code in an IntelliJIDEA project with the implementation of the (code) problems and (2) a pdf with the answers to the (text) problems submitted on Blackboard. **Make sure you double-check that the file was uploaded correctly on Blackboard!** I will not be accepting excuses after the deadline that the file was not uploaded correctly.

You may submit a physical (by hand) solutions to (text) problems. Please be sure your work is written clearly and readable.