COMP3506 Homework 2 – Sorting Algorithms, Linear Data Structures

Weighting: 15%

Due date: 7th September 2020, 11:55 pm

Overview

This purpose of this assignment is for you to become familiar with understanding sorting algorithms and linear data structures, and to investigate using experimental analysis to measure the practical performance of different sorting algorithms.

Marks

This assignment is worth 15% of your total grade. COMP3506 students will be marked on questions 1 to 3 out of **55 marks**. COMP7505 are required to additionally do question 4 and will be marked out of **70 marks**.

Submission Instructions

- Your solutions to Q1 will be submitted via Gradescope to the `Homework 2 - Question 1` submission. You should only submit your completed `SortingAlgorithms.java` file.
- Your solutions to Q2 will be submitted via Gradescope to the `Homework 2 - Question 2` submission. You should submit your answer as a `pdf` file.
- Your solutions to Q3 will be submitted via Gradescope to the `Homework 2 - Question 3` submission. You should only submit your completed `SimpleArrayDeque.java`, `SimpleLinkedDeque.java`, `ReversibleDeque.java` files.
- (COMP7505) Your solutions to Q4 will be submitted via Gradescope to the `Homework 2 - Question 4` submission. You should submit your answer as a `pdf` file.
- No marks will be awarded for non-compiling submissions, or submissions which import non-supported 3rd party libraries. You should follow all constraints laid out in the relevant questions.
- **Hand-written answers for questions 2 and 4 will not be marked.**

Late Submissions and Extensions

Late submissions will not be accepted. It is your responsibility to ensure you have submitted your work well in advance of the deadline (taking into account the possibility of internet or Gradescope issues). Only the latest submission before the deadline will be marked. See the ECP for information about extensions.

Academic Misconduct

This assignment is an individual assignment. Posting questions or copying answers from the internet is considered cheating, as is sharing your answers with classmates. All your work (including code) will be analysed by sophisticated plagiarism detection software. Students are reminded of the University's policy on student misconduct, including plagiarism. See the course profile and the School web page: `http://www.itee.uq.edu.au/itee-student-misconduct-including-plagiarism`.

Support Files Provided

The following support files are provided to you:

- `SortingAlgorithms.java` - you need to implement your question 1 sorting algorithms in this file.
- `SimpleDeque.java` - an interface used for question 3.
- `SimpleArrayDeque.java` - you need to implement question 3(a) in this file.
- `SimpleLinkedDeque.java` - you need to implement question 3(b) in this file.
- `ReversibleDeque.java` - you need to implement question 3(c) in this file.

Questions

1. (20 marks) Implement the following sorting algorithms in `SortingAlgorithms.java` as they have been described in lectures and tutorials.
   - Selection Sort (in the `selectionSort` method)
   - Insertion Sort (in the `insertionSort` method)
   - Merge Sort (in the `mergeSort` method)
   - Quicksort (in the `quickSort` method), using the median element at $\lfloor n/2 \rfloor$ as your pivot.

   A stub has been provided in `SortingAlgorithms.java`. Each algorithm needs to be able to sort arbitrary `Comparable`[1] objects and should be able to sort in descending order with the *reversed* flag.

   Notes and constraints:
   - You may use code (or implement the pseudocode) provided in lecture slides and tutorials.
   - Your solution will be automatically marked using Java 11.
   - You will be marked by a human on your code style (e.g. for following the style guide, and general readability). You should follow the CSSE2002 style guide as given on Blackboard. Importantly, you should comment your code where necessary.
   - You may not use anything from the Java standard library (e.g. any built in sorting algorithms).
   - Do not modify any of the provided method signatures. This may result in you receiving a mark of 0 for this question as it won't work with our automated tests.
   - When submitting, only submit the `SortingAlgorithms.java` file to the Gradescope submission.
   - You may add private helper methods, but do not use any static or class variables.
   - All your algorithms should modify the resulting array and not return anything. Note that some of the sorting algorithms you need to implement are not in-place, so you might need to do a little extra work to make it eventually modify the input arrays.

[1] See `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Comparable.html`.

2. (10 marks) In this question, you will be empirically testing the performance of your sorting algorithms (as implemented in Question 1) when sorting various types of arrays of various different sizes.

   Write a Java program to time (in milliseconds) the algorithms you wrote in question 1 for arrays of:

   - n random unsorted numbers
   - n random sorted numbers in ascending (increasing) order
   - n random sorted numbers in descending (decreasing) order.

   For each category, you should test with arrays of size $n = 5, 10, 50, 100, 500, 1000, 10000$.

   Based on the results of your program, fill out the table below for each of the three categories.

Algorithm	$n = 5$	$n = 10$	$n = 50$	$n = 100$	$n = 500$	$n = 1000$	$n = 10000$
Selection Sort							
Insertion Sort							
Merge Sort							
Quicksort							

   Additionally, for each category of arrays, make a graph plotting the runtime of all four sorting algorithms with n on the x-axis (you can use Excel, `pyplot`, or something else). Clearly include a legend and appropriate titles/labels. **You should have 3 graphs and 3 tables.**

   Include (i.e. copy/paste) the Java code you used to time your sorting algorithms at the end of your written answer for this question.

   Discuss your observed results and how they compare with what you expected. You may want to reference the asymptotic complexities of the sorting algorithms. Make sure to discuss the effects of n and the sortedness of the input array. **This should be no more than 300 words.**

   Hints:

   - Your program should automatically generate the arrays used for testing.
   - The `Random` class and the `Arrays.sort` method may be useful here.
   - The time taken to generate these arrays should **not** be included when measuring a sorting algorithm's runtime.
   - You must generate the test arrays and time the algorithms using Java. You may use a different program/programming language to plot the graphs, if so desired (your plotting code does not need to be included).

   Notes:

   - While the timing program's code will **not** be style-marked, please use suitable variable names, spacing, and comments as appropriate.
   - To include code in your document, you can use the `listings` package if you are using LaTeX. This which will format and highlight the code for you.
   - If you are using another word processor/typesetting system, please ensure that the code uses a monospaced font (such as Consolas or JetBrains Mono) and is appropriately formatted and indented (e.g. each level of indentation is four spaces).
   - Code which is not formatted appropriately may be penalized.

3. (25 marks) A deque data structure is a double-ended queue which supports $O(1)$ insertion and deletion from either end. This can be thought of as both a stack and a queue at the same time.

   You are given a generic interface for a deque in `SimpleDeque.java` (note this is *not* the `Deque` from Java's standard library). Your classes (described below) must implement all the methods from this interface.

   For this question, your implementation of all classes and methods should be as efficient as possible, in terms of both runtime and memory. Your Javadoc comments for methods should describe the runtime and memory complexity (using Big-O notation) of that method. Similarly, a class's Javadoc comment should describe the memory complexity of the class as a whole. Define all variables used in your bounds.

   (a) (10 marks) Implement the `SimpleArrayDeque` class. This implements a deque with a circular array (a basic Java array, not an `ArrayList`).

   (b) (10 marks) Implement the `SimpleLinkedDeque` class. This implements a deque with a doubly linked list. You cannot use Java's `LinkedList` and must implement the linked list yourself.

   (c) (5 marks) Implement the `ReversibleDeque` class which enables reversing a deque. This has exactly one constructor which takes a `SimpleDeque<T> data` argument. `ReversibleDeque` should implement the usual `SimpleDeque` methods by modifying this internal data deque.

   Additionally, `ReversibleDeque` defines a `void reverse()` method which, when called, reverses the deque so items at the left are now at the right and vice-versa.

   Notes:

   - `SimpleArrayDeque` has two constructors: one taking a single `int capacity` argument and one taking `int capacity` and `SimpleDeque<? extends T> otherDeque`. These constructors create deques with a limited capacity (see the interface's JavaDoc for more details).
   - `SimpleLinkedDeque` has four constructors: two which are the same as in `SimpleArrayDeque`, a default constructor taking no arguments, and one with a single `SimpleDeque<? extends T> otherDeque` argument. These constructors without a `capacity` argument have unlimited capacity.
   - The constructors taking an `otherDeque` argument are *copy constructors*. These should initialise the new deque as a copy of the other deque, without modifying the other deque.
   - For `ReversibleDeque`, you may assume that the given `data` deque will not be modified externally once it is passed to the `ReversibleDeque` constructor.

   Failure to adhere to the below constraints may result in you receiving a grade of 0 for the question.

   - You will be marked by a human on your code style (e.g. for following the style guide, and general readability). You should follow the CSSE2002 style guide as given on Blackboard.
   - You will submit only `SimpleArrayDeque.java`, `SimpleLinkedDeque.java`, and `ReversibleDeque.java` to the relevant Gradescope submission. You should not create or submit any additional files as these will not be marked.
   - You should not modify any of the provided method, constructor, or class signatures.
   - All of your files will be marked and compiled independently. None of your classes should rely implementation details of other classes (for example, `ReversibleDeque` should not depend on or even know about `SimpleArrayDeque`).
   - Your implementation will be automatically marked using Java 11.
   - You should not use any imports from the Java standard library, except for those already imported in `SimpleDeque.java` (e.g. `Iterator` and the exceptions). In particular, you may not use `LinkedList` or `ArrayList`. Ask on Piazza if you are unsure.
   - Any member variables, helper methods, or inner classes in your solutions should be made private (except for the constructors given in the assignment files, and the interface methods you have to implement).

4. **(COMP7505 only)** (15 marks) Sorting algorithms often have significantly different performance depending on the length of the array being sorted. Unfortunately, this is not conveyed in the asymptotic bounds. This means a $\Theta(n^2)$ algorithm might be faster in practice than a $\Theta(n \log n)$ algorithm for particularly small input sizes.

   Research (or experiment yourself) how the the performance of common sorting algorithms is affected by the length of the array. With this in mind, devise a sorting algorithm which is efficient for all lists (including random, sorted ascending, sorted descending) as well as a wide range of input lengths. Make sure to explain where the traditional algorithms fall short and how your algorithm improves on them. **Insert the code into your PDF writeup** for this question (you should follow the same guidelines from question 2). If you call the other sorting methods in `SortingAlgorithms.java`, you do **not** need to paste the code for those methods as well.

   Do an experimental analysis of your new sorting algorithm exactly like in question 2 (i.e. provide tables for your recorded times on the different inputs). Then, **plot your results against the experimental analyses you did in question 2**. (Note: you do **not** need to provide the code used for experimental analysis in this question).

   Discuss (in 400 words or less) how your sorting algorithm works, and how you came up with it (e.g. what did you experiment with to come up with it). Also discuss the performance of your algorithm in comparison to the other sorting algorithms (e.g. by referencing the plot).

   Some things you could try experimenting with are:

   - Making your sorting algorithm adaptive (i.e. runs in $O(n)$ time for sorted input).
   - Different quicksort pivot choices.
   - A hybrid sorting approach—that is, making use of two or more sorting algorithms.

   If you make use of external research in your solution, it must be referenced.