UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science
Faculty of Engineering, Built Environment & IT
University of Pretoria

COS212 - Data structures and algorithms

Practical 8 Specifications: Efficient Sort

Release Date: 29-05-2023 at 06:30

Due Date: 02-06-2023 at 23:59

Total Marks: 255

Contents

1 General instructions:

- This assignment should be completed individually, no group effort is allowed.
- Be ready to upload your assignment well before the deadline as no extension will be granted.
- You may **not** import any provided Java library. Importing any library or data structure will result in a mark of zero. You may only make use of 1-dimensional native arrays where applicable. If you require additional data structures, you will have to implement them yourself.
- If your code does not compile, you will be awarded a mark of zero. Only the output of your program will be considered for marks, but your code may be inspected for the presence or absence of certain prescribed features.
- All submissions will be checked for plagiarism.
- Read the entire specification before you start coding.
- You will be afforded three upload opportunities.
- Submissions that result in a compilation failure or a run time error will also receive a mark of zero.

2 Plagiarism

The Department of Computer Science considers plagiarism as a serious offence. Disciplinary action will be taken against students who commit plagiarism. Plagiarism includes copying someone else's work without consent, copying a friend's work (even with consent) and copying material (such as text or program code) from the Internet. Copying will not be tolerated in this course. For a formal definition of plagiarism, the student is referred to http://www.library.up.ac.za/plagiarism/index.htm (from the main page

of the University of Pretoria site, follow the Library quick link, and then choose the Plagiarism option under the Services menu). **If you have any form of question regarding this, please ask one of the lecturers, to avoid any misunderstanding.** Also note that the OOP principle of code re-use does not mean that you should copy and adapt code to suit your solution.

3 Outcomes

The primary goal of this assignment is to implement a set of efficient sorting algorithms namely:

- Count sort
- Merge sort
- Quick sort

4 Introduction

Complete the task below. Certain classes have been provided for you alongside this specification in the Student Files folder. A very basic main has been provided. **Please note this main is not extensive and you will need to expand on it**. Remember to test boundary cases. Submission instructions are given at the end of this document.

5 Task:

Your task for this practical will be to implement the following class diagram as described in later sections.

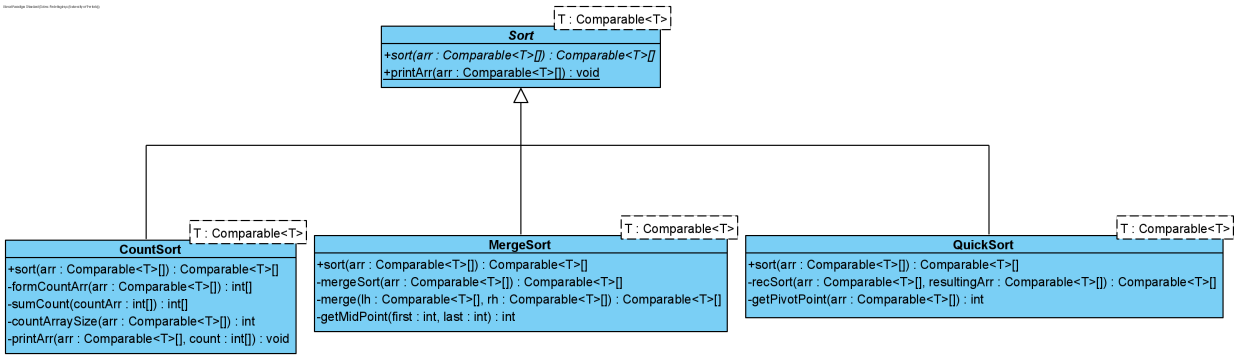

Figure 1: Class diagram

5.1 Sort

This class is the base class for the Sort Hierarchy.

- sort(arr: Comparable<T>[]): Comparable<T>[]
  - This is an abstract function.
  - This function takes in an array of values and returns a sorted version of the array.
  - Note the passed-in array should never be changed.
- printArr(arr: Comparable<T>[]): void
  - This is a static function.
  - This is provided and should not be altered.
  - This function will be used to print out arrays.

5.2 QuickSort

This class inherits from Sort. **Please create a new array when splitting the main unsorted array. Each recursive call should have a new smaller array passed to it.** This class should utilize the algorithm where the pivot point is moved to the front and the largest element to the back of the arrays [1].

[1]Lecture slides: Chapter 9: Part 3 Slide 5 and 6

- sort(arr: Comparable<T>[]): Comparable<T>[]
  - This function will sort the passed-in array and return a sorted copy of the array.
  - Use the Quicksort method as described in the lecture notes to sort the passed in array.
  - You will notice a printArray instruction at the top of the provided skeleton. Do not change this code nor its position. Add all your own code **below** this statement.
  - Also add a similar statement to any recursive functions that you may use such that the output in the provided main is mimicked.
- recSort(arr: Comparable<T>[], resultingArr: Comparable<T>[]): Comparable<T>[]
  - This is a **recommended** structure for a recursive function.
  - You will notice the print statement has already been added for you.
  - You do not need to use this function.
  - You can change the structure of this function if you wish. Do **not** alter the position of the print statement though.
- getPivotPoint(arr: Comparable<T>[]): int
  - This function is provided.
  - You will need to use this function to find the index of the pivot point for the algorithm.
  - The function returns an index.

5.3 MergeSort

This class inherits from Sort.

- sort(arr: Comparable<T>[]): Comparable<T>[]

  - This function will sort the passed in array and return a sorted clone of the array
  - Use the MergeSort method described in the lecture notes to sort the passed in array.
  - You will notice a printArray instruction at the top of the provided skeleton. Do not change this code nor its position. Add all your own code **below** this statement.
  - Also add a similar statement to any recursive functions that you may use such that the output that is in the provided main is mimicked.

- mergeSort(arr: Comparable<T>[]): Comparable<T>[]
  - This is a **recommended** structure for a recursive function.
  - You will notice the print statement has already been added for you.
  - You do not need to use this function.
  - You can change the structure of this function if you wish. Do not alter the position of the print statement though.

- merge(lh: Comparable<T>[], rh: Comparable<T>[]): Comparable<T>[]
  - This is a **recommended** structure for a recursive function.
  - You will notice **no** print statement has been added.
  - Do not add one unless needed to achieve the desired output.
  - You do not need to use this function.
  - You can change the structure of this function if you wish.

- getMidPoint(first: int, last: int):int
  - This function is provided.
  - You will need to use this function to find the index of the pivot point for the algorithm.
  - The function returns an index.

5.4 CountSort

This class inherits from Sort.

- sort(arr: Comparable<T>[]): Comparable<T>[]
  - This function will sort the passed-in array and return a sorted clone of the array
  - Use the CountSort method described in the lecture notes to sort the passed in an array.
  - You will notice a set of printArray instructions at the top of the provided skeleton. Do not change this code nor its position. Add all your own code **below** this statement.
- formCountArr(arr: Comparable<T>[]): int[]
  - This function will form the count array as used in the lecture notes.
  - The resulting array should contain the frequencies of each element, where the element itself is the index of the array.
  - To determine the size of the countArray use the countArraySize function.
  - Use the .hashCode() of each element in the passed-in parameter to determine the index of the element.
  - Note with this implementation the count array may become easily bloated due to the hashCode() values.
- sumCount(countArr: int[]): int[]
  - This function should form a cumulative frequency array of the passed in countArr as described in the lecture notes.
  - Use the following formula to calculate the value for each index in the resulting array (named result) :

$$\forall i \in \big[1, countArr.length()\big)$$

$$result[i] = result[i-1] + countArr[i]$$

  - The resulting array should be sized to the same size as the passed in array.

- countArraySize(arr: Comparable<T>[]): int
  - This function should calculate the size of the countArray that is used as described above.
  - To accomplish this you will need to find the element with the biggest .hashCode() value
  - Return the value found in the point above.
- printArr(arr: Comparable<T>[], count: int[]): void
  - This is a provided function and should not be altered.
  - This function will be used to print out arrays in a specific predefined way.

6 Helper functions and variables

You are allowed to add helper functions to any of the classes.

7 Example output

In the provided main you will see each sort method sorting an array. You will also see the expected output at the bottom of the main, but it will also be listed in the tables below. Assume the intial unsorted array looks as follows:

$$[5, 4, 6, 3, 7, 2, 8, 1]$$

MergeSort	CountSort	QuickSort
[5;4;6;3;7;2;8;1]		
[5;4;6;3]		
[5;4]		[5;4;6;3;7;2;8;1]
[5]		[2;1]
[4]		[1]
[6;3]		[]
[6]	[5{1},4{1},6{1},3{1},7{1},2{1},8{1},1{1}]	[5;7;6;4;8]
[3]	[5{5},4{4},6{6},3{3},7{7},2{2},8{8},1{1}]	[5;4]
[7;2;8;1]	[1{0},2{1},3{2},4{3},5{4},6{5},7{6},8{7}]	[4]
[7;2]		[]
[7]		[7;8]
[2]		[]
[8;1]		[8]
[8]		
[1]		

Note: This output is to be formed when sort function is called

For quick sort the following table indicates the pivot value that was used to split the array.

QuickSort	Pivot value used to split the array
[5;4;6;3;7;2;8;1]	3
[2;1]	2
[1]	1
[]	N/A
[5;7;6;4;8]	6
[5;4]	5
[4]	4
[]	N/A
[7;8]	7
[]	N/A
[8]	8

8 Submission

You need to submit your source files on the Fitch Fork website (https://ff.cs.up.ac.za/). All methods need to be implemented (or at least stubbed) before submission. The following java files should at least be in a zip archive named uXXXXXXXX.zip where XXXXXXXX is your student number:

- Sort.java
- CountSort.java
- MergeSort.java
- QuickSort.java

You may add any other custom classes that you created. Your code should be able to be compiled with the following command:

```
make *.java
```

and run with the following command:

```
java App
```

You have 5 submissions and your best mark will be your final mark. Upload your archive to the Practical 8 slot on the Fitch Fork website. Submit your work before the deadline. **No late submissions will be accepted!**