

**CS2020: Data Structures and Algorithms (Accelerated)**

**Problem Set 3**

*Due: Wednesday February 12, 11:59pm*

**Overview.** This problem set consists of two problems. The first asks you to be a detective: use your knowledge of sorting algorithms to determine which algorithm is which. The second problem asks you to solve a specific problem that comes up in deploying cellular phone towers: how do we calculate the size of cell-phone coverage zones?

**Collaboration Policy.** You are encouraged to work with other students on solving these problems. However, you **must** write up your solution **by yourself**. We may, randomly, ask you questions about your solution, and if you cannot answer them, we will assume you have cheated. In addition, when you write up your solution, you **must** list the names of every collaborator, that is, every other person that you talked to about the problem (even if you only discussed it briefly). Any deviation from this policy will be considered cheating, and will be punished severely, including referral to the NUS Board of Discipline.

## Problem 1. (The CS2020 Detectives)

We have six impostors on our hands. Each *claims* to be **Mr. QuickSort**, the most popular sorting algorithm around. Only one of these six is telling the truth. Four of the others are just harmless imitators, **Mr. BubbleSort**, **Ms. SelectionSort**, **Mr. InsertionSort**, and **Ms. MergeSort**. Beware, however, one of the impostors is *not a sorting algorithm*: **Dr. Evil** maliciously returns unsorted arrays! And he won't be easy to catch. He will try to trick you by sometimes returning correctly sorted arrays. Especially on easy instances, he's not going to slip up.

Your job is to investigate, and identify who is who and what is what. Attached to this problem set, you will find six sorting implementations: (i) `SorterA.class`, (ii) `SorterB.class`, (iii) `SorterC.class`, (iv) `SorterD.class`, (v) `SorterE.class`, and (vi) `SorterF.class`. Each of these class files contains a class that implements the `ISort` interface which supports the following method:

```
public <T extends Comparable<T>> void sort(T[] array)
```

You can test these sorting routines in the normal way: create an array, e.g., `Integer testArray[]`, and create a sorter object, e.g., `ISort sorter = new SorterA()`. Then, to sort, simply call: `sorter.sort(testArray)`. (You will have to set up Eclipse to include each of the `.class` files in your project.)

You can then use the `StopWatch` to measure how fast each of these sorting routines runs. Each sorting algorithm has some inputs on which it is fast, and some inputs on which it is slow. Some sorting algorithms are stable, and others are not. Using these properties, you can figure out who is who, and identify Dr. Evil.

Beware, however, that these characters can be deceptive. While they cannot hide their asymptotic running time, they may well choose to run consistently slower than you expect. (You should not assume that QuickSort is always the fastest for all sized inputs!)

For this problem, you should hand in a report giving:

- A correct identification of each of the six sorting routines.
- An explanation of how you performed the identification. Ideally, your identification should be based on how the algorithms execute on certain inputs. The key is to identify certain unique properties of each algorithm.
- Test code that verifies these properties.

Notice that decompiling the `.class` files and trying to identify the algorithms in that manner will not be considered an appropriate explanation.

As part of your test code, please submit the following routines:

```
boolean isStable(ISort sorter, int size)
boolean checkSorted(ISort sorter, int size)
```

This first method should test whether the given sort is stable by testing it on an input of the specified size. The second should test whether the sorter works correctly by sorting an input of the specified size and checking if it is sorted.

**Eclipse tips:** The first thing you will need to do is to import the class files into your project in Eclipse.

- The sorters are in the package `cs2020`. It is recommended that your classes also be in the package `cs2020`.
- Put the six `.class` files in a file named `cs2020`. Put the folder `cs2020` in a folder named `sorters`. Make sure the folder `sorters` is *not* under your Eclipse workspace. It is important that the folder hierarchy is set up in this way.
- For example, you can now find the file `SomeFolder/sorters/cs2020/SorterA.class`.
- Right click on your project in Eclipse and select: “Build Path -> Configure Build Path.”
- Click the “Libraries” tab.
- Click “Add External Class Folder.”
- Select the folder `sorters` that has the `cs2020` folder inside it. (Do not select the `cs2020` folder.)
- Click ok. The sorters should now appear under the “References Libraries” in the “cs2020” package.

## Problem 2. (Cellular System 2020)

After graduating from the prestigious School of Computing at the Notional University of Signalpros, you are hired by MobileTwo, a new cellular company that would like to take over the mobile phone industry.

Your job is to evaluate the cell phone coverage on a major national highway. Along the highway are several cell phone towers. Each tower provides coverage for a portion of the highway. We will assume that the signal strength determines exactly the coverage provided. Sometimes coverage zones overlap, i.e., two towers cover the location. In other places, a location may be covered by only one tower, or by no towers at all.

The MobileTwo team will provide you with the location of each tower, and the distance that it covers. Your program should calculate the percentage of the highway that is covered.

**Requirements.** Your program is required to have the following constructor:

```
CoverageCalculator(int highwayLength)
```

We will assume that the highway starts at location zero and proceeds for the specified number of kilometers. (Notice that a highway of length 2 will actually have three locations at which a tower can be placed: 0, 1, and 2.) Your program should also support the following method:

```
void addTower(int location, int range)
```

This method will add a tower to the highway at the specified location. You can assume that every location on the highway that is within the specified range of the tower is now covered. Finally, your program should implement the following method:

```
int getCoverage()
```

This method should return the total number of kilometers that are covered by at least one tower. (We can now calculate the percentage coverage by dividing the value returned by `getCoverage` by the `highwayLength`. However, to avoid problems of floating point arithmetic, we ask that you return the total coverage area, rather than the percentage.)

If there are  $n$  towers, the execution of `addTower` should run in  $O(1)$  time, and `getCoverage` should run in time  $O(n \log n)$ .

As part of your solution, you should implement a class `Tower` that implements the `Comparable` interface and encapsulates the state relevant to a tower. (*Hint:* you may want the implementation of `compareTo` to sort by something other than the location of the tower.)

## Example executions of the CoverageCalculator:

### Example 1.

```
CoverageCalculator calc = new CoverageCalculator(100);  
calc.addTower(20, 5);  
calc.addTower(10, 5);  
System.out.println(calc.getCoverage());
```

Output: 20

### Example 2.

```
CoverageCalculator calc = new CoverageCalculator(100);  
calc.addTower(20, 5);  
calc.addTower(10, 5);  
calc.addTower(30, 2);  
calc.addTower(16, 10);  
System.out.println(calc.getCoverage());
```

Output: 25

### Example 3.

```
CoverageCalculator calc = new CoverageCalculator(100);  
calc.addTower(20, 5);  
calc.addTower(10, 5);  
calc.addTower(30, 2);  
calc.addTower(16, 10);  
calc.addTower(5, 10);  
calc.addTower(18, 2);  
System.out.println(calc.getCoverage());
```

Output: 30

### Example 4.

```
CoverageCalculator calc = new CoverageCalculator(100);  
calc.addTower(20, 0);  
calc.addTower(40, 0);  
calc.addTower(60, 0);  
System.out.println(calc.getCoverage());
```

Output: 0

### Example 5.

```
CoverageCalculator calc = new CoverageCalculator(100);  
calc.addTower(110, 30);  
System.out.println(calc.getCoverage());
```

Output:

Error: invalid location.

0