CS 345 (Fall 18): Analysis of Discrete Structures

Project #1

Basic Sorting

due at 5pm, Thu 6 Sep 2018

1 Overview

In this project, you'll be implementing several of the sorting algorithms that we've discussed. You'll implement them all in Java, and the grading script will automatically test your program against a variety of inputs.

Each algorithm will be implemented as a Java class; each class will implement the `Proj01_Sort` interface, which I'll provide to you.

We'll test these algorithms using a couple of different classes. First, I've provided the `Proj01_TestMany` class. This will not be used by the grading script, but it's a great way for you to do a quick check on all of your algorithms. Second, I've provided the `Proj01_Main` class; this has **lots** of options, which you can use (and the grading script will use) to test different situations.

1.1 Required Algorithms

You must implement all of the following algorithms:

- Bubble Sort
- Insertion Sort
- Selection Sort
- Quick Sort
- Merge Sort

Details will be listed below.

1.2 Test Early, Test Often

Since you're implementing several Java classes, you won't be able to compile and run the `Main` class until you have created all of them. **Do not wait to test your code!**

Instead, you should (first thing!) create a stub for each class. That will allow you to compile and run the code. Then, implement the sort algorithms **one at a time** - and make sure to debug each one thoroughly before you move on to the next.

Trust me, it'll save you time in the long run.

1.3 Using the `Proj01_Main` class

While `Main` supports many command-line arguments, the most important are the first two. The first (which is required) is the name of the algorithm to test. The second (which can be omitted) is the seed used to initialize the random-number generator. When you run a test the first time, don't specify a seed, and the class will automatically choose one for you. (The seed that it chooses is printed on the first line of output.) But if you find a bug in your code, you can re-run the **exact same test** again, as often as you like, by telling it exactly what seed it should use.

The next most important argument is the `debug` option, which must always be the last option (if provided). If you turn on this flag, then the sort algorithm will be initialized with `debug=true` - and thus, your code should print out **lots** of debug information as it runs. (This is how you will debug problems in your own code.)

There are three other options, which can be used in any combination.

- `String`

  Normally, our code will sort integers. However, if you turn on this flag, then it will sort strings instead. If you implement your sort algorithms properly - using the `Comparable` interface - then this shouldn't require any change to your code to support!

- `million`

  Normally, our code will generate a few dozen items as input. But if you set this flag, then it will generate exactly 1 million items.

  A good Quicksort/Merge Sort algorithm should be able to sort this in just a couple of seconds. But **don't use this option with any of the $O(n^2)$ algorithms!**[1]

- `sorted`

  This option tells `Main` to pre-sort the data, before sending it to the sort algorithm.

  This can be used with any sort algorithm. However, if you choose Quicksort, this option will force the mode to 2 (median-of-3). See below for a discussion of the Quicksort mode.

(Sorry, my command-line arg parser is pretty simplistic. So while you can use any combination of these arguments, they must always be in the proper order. Otherwise, I won't recognize that they are there.)

[1] Exception: You'll notice that the grading script will use this option with Insertion Sort - but it only does this when it **also** sets the `sorted` option. When the input is sorted, Insertion Sort runs in $O(n)$ time, if it's implemented correctly.

1.4 Java Warnings

Normally, I think that it's very important to pay close attention to any compiler warnings. However, because we're using the Comparable interface, we're going to get warnings like this from the compiler:

```
Note: Some input files use unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

I'm hoping to cover Java generics in more detail this semester, so we should be able to discuss this warning later (and learn how to fix it). However, for now, feel free to ignore it.

1.5 Describe Your Debug

Your textbook already has rough code for most of the algorithms that you need to implement for this project. This makes it difficult to assign a code project! So, in addition to the code project, you will also turn in a PDF which describes the code that you used for debug.

The PDF that you turn in must be at least one page in length (not counting figures, code, or copy-pasted output), and describe the debug features that you added to your sort algorithms. All of these features must be available in the code that you turn in - but they will be turned on or off, based on the value of the 'debug' parameter, when we create the object for each sort algorithm.

You must implement reasonable debug features for each sort algorithm. The features should be detailed enough so that, when debug is turned on, you can demonstrate the proper operation of the algorithm. This might be text statements that describe what you are doing, printouts of the values in the array (which change over time), or whatever. You get to decide **what** you use for debug - but you must describe it in the PDF that you turn in to D2L along with your code.

1.5.1 Example

This is what the debug output looks like for my code, when running the Insertion Sort algorithm:

```
20 92 14 66 10 61 50 44 85 83 0 98 71
(20) . 92 14 66 10 61 50 44 85 83 0 98 71
20 (92) . 14 66 10 61 50 44 85 83 0 98 71
20 92 (14) . 66 10 61 50 44 85 83 0 98 71
20 (14) 92 . 66 10 61 50 44 85 83 0 98 71
(14) 20 92 . 66 10 61 50 44 85 83 0 98 71
14 20 92 (66) . 10 61 50 44 85 83 0 98 71
14 20 (66) 92 . 10 61 50 44 85 83 0 98 71
14 20 66 92 (10) . 61 50 44 85 83 0 98 71
14 20 66 (10) 92 . 61 50 44 85 83 0 98 71
14 20 (10) 66 92 . 61 50 44 85 83 0 98 71
```

As you can see, I've added decorators (the number with parens, and the dots) to make clear where we are working in the algorithm. You don't have to replicate this, but it would be wise to have a debug format that's easy to read!

2 How will Russ tell if you have have implemented the algorithms?

The unfornate thing about sorting - for a class project - is that **all** of the (correct) algorithms get the same result - even if some of them are very slow.

For this reason, I'll be asking the TAs to take a look at your code. While they will not be doing a point-by-point debug of your algorithm, they will take a quick look to make sure taht you are implementing each one in the correct fashion.

3 Some Requirements

Take note of the following rules and recommentations:

- **MUST** - In all of your sort algorithms, you must implement debugging code.
- **MUST** - In all of your sorts, your `debug` version of the sort, and the non-`debug` version, must be the same code.

  This means that you should scatter

  ```
  if (debug)  ...
  ```

  lines throughout your algorithm. It clutters up the code, but it ensures that the code you're debugging is the **real** code[2].
- **MUST NOT** - Your code must not duplicate the buffers, or any part of them - except for the temporary buffer required by Merge Sort. (It's fine to copy one or two temporary values into variables, but you cannot duplicate a **range** of values out of the array.)
- **MUST** - In Quicksort and Merge Sort, you must honor the `baseLen` argument that I send to your constructor. (See the description below.)
- **MUST** - In Quicksort, you must honor the `mode` argument that I send to your constructor. (See the description below.)
- **SHOULD**[3] - Your Merge Sort and Quicksort implementations should reuse the code in one of the other sort algorithms to sort the small blocks.

[2] Code reuse is good style, because it (usually) reduces the overall complexity of the code, and (almost always) prevents duplication of bugs.

[3] "Should" means that this is good advice, but not a strict requirement.

Of course, the `Proj01_Sort` interface doesn't allow you to specify a range of values to sort - it always sorts the entire array. How can you obey that interface, while also giving Quicksort the ability to sort only a small range out of an array?

NOTE: If you choose not to do this, you are still **required** to sort the small blocks with an $O(n^2)$ algorithm - the only choice you have is whether to actually **call** it (good style), or to cut-n-paste it into Quicksort/Merge Sort (bad style).

- **SHOULD** - Your Merge Sort implementation should have a single temporary buffer, which it re-uses for all of the merge steps.

  If you choose to allocate a new buffer for each merge step, that's permissible - but in that case, the size of the buffer you allocate **MUST** be the proper size for the merge, **not** the size of the entire array! (If you break this rule, you won't run in $O(n \lg n)$ time.)

- **SHOULD** - Your sort implementations should use helper functions as appropriate.

- **MUST NOT** - You must not use global variables anywhere in your code. **Instance variables** of classes are perfectly fine (and actually, are necessary), but global variables (that is, variables declared with `static`) are banned!

  Of course, local variables inside functions are just fine, as are parameters passed from one function to another.

4 Interfaces and Objects

This project defines a Java `interface` named `Proj01_Sort`, which contains a single method. Every one of your sort algorithms must be implemented as a `class` which implements this interface.

Thus, the way that one uses one of these sort algorithms is something like this:

```
Proj01_Sort curAlgo = new Proj01_SomeParticularAlgorithm(...params...);
Comparable[] dataset = { ...some values... };
curAlgo.sort(dataset);
```

5 Comparable

The input data to the sort algorithms is an array of `Comparable`. `Comparable` is a standard Java interface, which represents any class where two objects can be compared and put in order. `Comparable` includes a method `compareTo()`, which you've probably already seen when comparing `String` objects.

This is pretty cool - because it means that your sort algorithm can sort almost any type of input. But the downside is that you cannot simply use the `<` operator; instead, every comparison will have to use `compareTo()`.

(To see this in action, try running the project's `Main` class, with the `String` command-line argument.)

6 Arguments to the Constructor

Each of your classes must include a public constructor. The first argument to every constructor is a `boolean`, which indicates whether or not you should do debugging when you run the sort. For the $O(n^2)$ algorithms, that is the only argument.

6.1 `mode`

The `mode` argument is used only in Quicksort. (The order of the parameters in the constructor to this class is `(debug,mode,baseLen)`.) The mode indicates how you find the pivot. There are three possible values:

- 0 - Always select the first element in the range as the pivot
- 1 - Always select the middle[4] element in the range as the pivot
- 2 - Do the median-of-3 algorithm to choose the pivot, comparing the first, middle, and last elements

6.2 `baseLen`

Both Quicksort and Merge Sort have an `int baseLen` parameter, which tells you when to switch to an $O(n^2)$ algorithm. You **must** switch to the $O(n^2)$ algorithm when the length of the block that you are sorting is **equal to or less than** the `baseLen` that we've specified.

7 Merge Sort - Bottom Up, or Top Down?

When you are running Merge Sort, there are two variants about how you can organize your sort; I call them bottom-up and top-down.

The slides show the top-down strategy; you recurse into the left half of the tree, and **fully** sort it - and then you go into the right-hand side. You then merge the two sections together at the end.

In the top-down strategy, your block sizes are not nice powers of two; instead, you split the data in half, sort half of it, and so on. So the size of the blocks is determined by the size of the input data.

[4] I don't precisely specify what the "middle" element is; I don't care. Just split the range in half, more or less.

The bottom-up strategy instead iterates through the **entire** data in a first pass, doing **all** of the $O(n^2)$ sorts at once. All of the blocks are then the "perfect" size (exactly `baseLen`), except for the last one, which is usually smaller. Then, in a second pass, you combine pairs of the small blocks together; in the third pass, you combine those into yet-larger blocks, and so on.

The total amount of work is exactly the same; both of them work fine. You are allowed to use either strategy in your code. The top-down strategy will likely require a recursive implementation; the bottom-up strategy can be implemented as nested loops in a single function.

8 Base Code

Download all of the files from the project directory

`http://lecturer-russ.appspot.com/classes/cs345/fall18/projects/proj01/`.

If you want to access any of the files from Lectura, you can also find a mirror of the class website (on any department computer) at:

`/home/russelll/cs345f18_website/`

9 A Note About Grading

Your code will be tested automatically. Therefore, your code must:

- Use exactly the filenames that we specify (remember that names are case sensitive).
- **Not** use any other files (unless allowed by the project spec) - since our grading script won't know to use them.
- Follow the spec precisely (don't change any names, or edit the files I give you, unless the spec says to do so).
- (In projects that require output) match the required output **exactly!** Any extra spaces, blank lines misspelled words, etc. will cause the testcase to fail.

To make it easy to check, I have provided the grading script. I **strongly recommend** that you download the grading script and all of the testcases, and use them to test your code from the beginning. You want to detect any problems early on!

9.1 Automatic Testing

We have provided a testing script (in the same directory), named `grade_proj01`. Place this script, all of the testcase files, and your program files in the same directory. (I recommend that you do this on Lectura, or a similar department machine. It **might** also work on your Mac, but no promises!)

10 Turning in Your Solution

Turn in the following files:

```
Proj01_BubbleSort.java
Proj01_InsertionSort.java
Proj01_SelectionSort.java
Proj01_QuickSort.java
Proj01_MergeSort.java
<some PDF file, describing your debug strategy>
```

You must turn in your code using D2L, using the Assignment folder for this project.