CS 345 (Spring 18): Computer Organization

Project #1

Basic Sorting

due at 5pm, Thu 1 Feb 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 set of different inputs.

However, sorting is a notoriously hard problem to grade automatically, since (of course) **any** sorting algorithm - if it works correctly - will have exactly the same output as any other. Should I give you really huge inputs, to see if your algorithm is efficient? If I did that, how would I test the inefficient ones? I could have the TAs look at everybody's code, but they have limited time. How do I grade automatically?

My solution is this: 4/5 of the automated score will come from **correctness** - that is, simply sorting the data. The TAs will take a quick look at your code (so no cheating by using Bubble Sort for everything!), but we will automatically check your algorithms by **adding debug outputs.** The last 1/5 of your automated score (for each testcase I run) will be to see if you match my example output exactly - when we run the function `sort_trace()` instead of `sort()`.

(See the next page for an example of what the output from BubbleSort looks like, when tracing is turned on.)

1.1 Required Algorithms

You must implement all of the following algorithms:

- Bubble Sort
- Insertion Sort
- Quick Sort
- Merge Sort (recursive)
- Merge Sort (bottom-up)

Details will be listed below.

EXAMPLE OUTPUT: BubbleSort, tracing turned on

```
43
39
12
13
44
14
48
19

swap(0,1)
swap(1,2)
swap(2,3)
swap(4,5)
swap(6,7)
swap(0,1)
swap(1,2)
swap(3,4)
swap(5,6)
swap(2,3)
swap(4,5)
swap(3,4)

12
13
14
19
39
43
44
48
```

NOTES:

- Whether tracing is turned on or off, the testcase will always print the original array first, and the sorted array at the end.
- When tracing is turned on, the sorting algorithm will print out debug statements about what it is doing. Sometimes, it is as simple as documenting where swaps occurred. Sometimes, it has a lot more information - such as in the Partition and Merge operations.

2 Where's the Spec?

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! This is another reason why I added the requirement that you include a "trace" feature in your code.

However, I won't be giving you precise details about all of the corner cases of your algorithm, or about exactly what to print. Instead, I've provided "example" implementations of each algorithm (as `.class` files); the grading script will run the example code, and then also run yours, and compare their outputs.

So, you will have to explore the corner cases and figure out the details. For instance, how small does the block have to be before my QuickSort code switches to Insertion Sort? What element, exactly, do I consider the "middle" value when I'm doing the median-of-3 algorithm? Where does Merge Sort split the array, when there are an odd number of elements?

What I will provide is a list of all of the places where I print. Make sure to exercise all of the cases, see what my example code prints out, and then "reverse engineer" my code to duplicate it.

I print out (in trace mode):

- **All:**
  Any time that I swap two elements in the input array. (Note that the "merge" step in Merge Sort doesn't use swaps - but when we find small blocks, Merge Sort calls Insertion Sort, and will print out swap traces.)
- **Merge Sort:**
  At the beginning of each recursive call to the Merge Sort algorithm.
- **Quick Sort, Merge Sort, Merge Sort Bottom-Up:**
  At the beginning of any call into Insertion Sort (to sort a small block).
- **Merge Sort, Merge Sort Bottom-Up:**
  At the beginning of any merge operation.
- **Merge Sort, Merge Sort Bottom-Up:**
  During the merge operation, every time that you compare two values.
- **Merge Sort, Merge Sort Bottom-Up:**
  During the merge operation, after one side has been emptied, and we need to "flush" the rest of the data, from the other side, into the merge buffer.
- **Quick Sort:**
  At the beginning of the partitioning process.
- **Quick Sort:**
  During the positioning process, printing the indices (and then values) of the first, middle, and last values (potential pivots).

- **Quick Sort:**
  Information about which of the 3 values was chosen as the pivot.

- **Quick Sort:**
  At the end of a partition operation, the position of the pivot.

2.1 Can We Work Together?

While you of course cannot work together on writing the code, I encourage the class to collaborate to **write testcases.** Write testcases which explore various scenarios that the sorting algorithms might face - and share what you've written on Piazza!

3 Notes on the Algorithms

Although I won't give you every detail, I wanted to pass along a few hints and pointers.

3.1 Bubble Sort, Insertion Sort

No notes.

3.2 Quick Sort

You **must** implement the median-of-3 algorithm, or you will not match what my example code does. But how should you handle median-of-3 when two or three of the values you look at are duplicates?

You must use Insertion Sort to sort the small blocks - that is, Insertion Sort must be the algorithm you use at the base case of recursion. (Do **not** use Insertion Sort to sort the 3 candidate pivot values - just use a big `if/else` block to figure out which one is the median!)

3.3 Merge Sort

This version of Merge Sort (the normal, recursive version) splits the array in half each time, and recurses into each.

When you need to merge data together, you have two choices:

- Allocate a correct-sized array for **each** merge operation. This is simple (fewer parameters to pass around between helper functions). It is also $O(n)$, since the cost is proportional to the length of the array. However, the **total cost**, in CPU time, is poor.

- Allocate a **single** huge array, right at the beginning, and pass it around. This is **more efficient** - but it requires that you manage additional parameters and variables.

3.4 Merge Sort Bottom-Up

All of the same rules apply as do for Merge Sort - except that the order of sorting the tiny blocks (and the merging).

In the first pass, divide the input data into fixed-size blocks (there might be some leftovers at the end). Sort all of these blocks.

In the second pass, merge adjacent pairs of blocks together. So if, for example, the first pass used 33 numbers per block, then in the 2nd pass, you will be building large blocks which are 66 numbers in size.

Keep looping, until the entire array has been merged into a single block.

4 Misc Rules

(These rules apply to all of your algorithms!)

4.1 Global Variables

Global variables are forbidden in this project! Your functions should work without any need for any globals; instead, make sure to pass parameters, containing any information that other, deeper calls might need.

Of course, local variables are just fine.

4.2 Don't Duplicate the Array

Except for the **necessary** temporary buffer in the merge step of Merge Sort, you **must not** duplicate the data. For instance, in Merge Sort, if you "split the data in half," you **must not** actually allocate two new arrays, half as large, and copy everything into it.

Instead, include parameters to tell you what **part** of the array you are going to sort.

4.3 Helper Functions

You are welcome to use helper functions in your code. It is also legal to call functions in **other classes** - but please, don't create any **additional** classes, that I didn't plan for. If you do try this, it makes your code a lot harder to grade!

5 Base Code

Download all of the files from the project directory

`http://lecturer-russ.appspot.com/classes/cs345/spring18/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/cs345_website/`

I have provided a (nearly empty) Java file for each of the required classes; fill in the `sort()` and `sort_trace()` functions.

6 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!

6.1 Testcases

You can find a set of testcases for this project at
`http://lecturer-russ.appspot.com/classes/cs345/spring18/projects/proj01/`

In this project, all testing will be done using a single Java class, named `Proj01_TestDriver`. You should not change this file - since we'll be providing it for you when we do the grading.

For input, the grading script will look for any files named `*.dat` - each file needs to be a series of integers, separated by whitespace. (We use Java's `Scanner` class to read the integers into memory.)

The grading script will run each of the required algorithms against each of the `*.dat` files that it finds. For each combination, it will run the test twice: once with the "tracing" turned on, and once with it turned off. If you **exactly match** the output of the "example" class - with tracing turned off - then you will earn 4/5 of the credit for that testcase. However, the last 1/5 of the credit will require that you also match the example code **exactly** - with tracing turned on!

6.2 Other Testcases

For many projects, we will have "secret testcases," which are additional testcases that we do not publish until after the solutions have been posted. These may cover corner cases not covered by the basic testcase, or may simply provide additional testing. **You are encouraged to write testcaes of your own, in order to better test your code.** You are also encouraged to share your testcases on Piazza!

6.3 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!)

7 Turning in Your Solution

Turn in the following files:

```
Proj01_BubbleSort.java
Proj01_InsertionSort.java
Proj01_QuickSort.java
Proj01_MergeSort.java
Proj01_MergeSortBottomUp.java
```

You must turn in your code using D2L, using the Assignment folder for this project. Turn in only your program; do not turn in any testcases or other files.