CS367 Programming Assignment 5

Lecture 1, Spring 2018

Due by 11:59 pm on Thursday, May 3, 2018

In this page: Overview | Specifications | Handing in Related pages: Assignments

Overview

Why are we doing this program?

Description

In this assignment you will be implementing several sorting algorithms. In addition, each time you run a sorting algorithm, you will be collecting statistics about the sort. The statistics for each sort will include the number of comparisons, the number of data moves, and the time required by the sort. By generating these statistics, we can compare the performance of the different sorting algorithms in practice and see how our results match up with the complexities we derived for each algorithm.

For this assignment, you may use any code or algorithms given in the on-line reading on sorting, in lecture, or in the documentation for the `ComparisonSort` class. You may **not** use any other sources to develop or create the code for your sorting algorithms.

Goals

The goals of this assignment are to:

- Gain experience instrumenting code.
- Gain experience measuring elapsed time in Java.
- Implement several sorting algorithms.
- Gain experience writing and using `static` methods
- Analyze the results of your program to better understand how your program behaves.

Specifications

What are the program requirements?

The `ComparisonSort` Class

You will be implementing six sorting algorithms (selection, insertion, merge, quick, heap, and an improved variation on selection sort we'll call selection2) and one method that, given an input array, runs each of the algorithms, generates statistics, and prints out the results. All of these methods will be `static` (i.e., class) methods in the `ComparisonSort` class.

Each sort takes a `Comparable` array as a parameter. Note that nothing is returned. The array that is passed in is modified by the sort so that after the sort is finished the array is in ascending order.

The `runAllSorts` method is passed an array, runs each of the sort algorithms, and displays the following statistics for each sort:

- number of comparisons
- number of data moves
- time (in milliseconds)

The sample output shows what is required. All output will go to the console. Note that each sort method will modify the array passed as a parameter. However, in order to compare the behavior of the different sorting algorithms we will want to use the *same* input for each algorithm. Thus, it is important that the `runAllSorts` method pass **identical** information to each sort method. Note also that the `runAllSorts` method should **not** modify the original array it is passed.

The `SortObject` Class

The `SortObject` class is a class which implements the `Comparable` interface and is designed to help you count the number of comparisons your code does. It has a class data member that keeps track of the number of comparisons done. Each time the `compareTo` method is called, the class data member is incremented. The `resetCompares` method resets the class data member to 0 and the `getCompares` method returns the number of comparisons done on `SortObjects` since the last reset. See the `SortObject` documentation for additional details about this class.

Generating Statistics

The `SortObject` class takes care of counting the comparisons. You will just need to make sure that you reset the counter or get its value at appropriate times.

You will need to add all the code to keep track of data moves. For the purposes of this programming assignment, each assignment to a variable of type `SortObject` is considered one data move. For example, if `arr` is an array of `SortObject`, then the code

```
SortObject temp;
temp = arr[0];
arr[0] = new SortObject(8);
arr[1] = arr[0];
```

does three (3) data moves. You may wish to look at the code for the `SortObject` class to get ideas for how to keep track of the number of data moves your code does.

To keep track of the time, use the `currentTimeMillis` method of the `System` class. This method returns the current time (in milliseconds). To figure out how long a method takes, just get the time right before and right after the method is called and subtract the before time from the after time.

The sample output gives an example of the statistics generated. This output was formatted using a private `ComparisonSort.printStatistics` that we provide. You are encouraged (but not required) to use it to make your output easier to read.

Analyzing the results

In the file `Questions.txt` you will find some questions that you will need to answer. These questions are meant to have you think about the results you get from the sorting algorithms you are implementing.

Testing

The driver class `TestSort` has been provided to help you test some of the functionality of your sort algorithms and answer the questions. It takes two command-line arguments:

1. the number of items in the input array
2. the seed for the random number generator

The sample output was obtained by running `TestSort` with the command line arguments "5000 43210" You can change this code as you wish to test your entire `ComparisonSort` class.

Extra Credit

You can earn up to 15% extra credit by

- implementing the `insertion2` sort described in the documentation for the `ComparisonSort` class,
- modifying your `runAllSorts` method to include `insertion2` as the last sorting algorithm run, and
- including the `insertion2` results in your answers in `Questions.txt`

Summary of provided materials

- `ComparisonSort.java` - you will need to modify this class
- `SortObject.java` - do not modify this class
- `TestSort.java` - you can modify this class
- `Questions.txt` - you will need to modify this file

How to proceed

After you have read this program page and given thought to the problem we suggest the following steps:

1. Review these style and commenting standards that are used to evaluate your program's style.
2. You may use the Java programming environment of your choice in CS 367. However, all programs must compile and run (using the Java 8 SE) for grading. We recommend that you use Eclipse.
3. Look over the Javadoc documentation for the provided classes.
4. Download the provided files.
5. Incrementally implement `ComparisonSort` one sorting algorithm at a time by first writing the sort, then testing it, and finally adding code to `runAllSorts` to generate the statistics for that sort.

   Check out these Frequently Asked Questions (and their answers).

   You will be turning in one Java source code file: `ComparisonSort.java` This means that for heap sort, you should **not** implement a heap class (or use any Java API classes). Your heap sort code should use the algorithms for inserting into and removing from an array-based heap to accomplish the sort as described in the `ComparisonSort` documentation. (Note: you *are* free to make use of the algorithms for inserting and removing from a heap that you developed for the `ArrayHeap` class for Program 3; you will just have to figure out how to take them out of the instantiable `ArrayHeap` class and modify them for use inside your `ComparisonSort` class.)
6. Answer the questions in Questions.txt. This will involve running `TestSort` several times and analyzing your results.
7. If you are doing the extra credit, make sure to include the results of the insertion2 sort in your answers in the questions.
8. Submit your work for grading.

Handing in

What should be handed in?

Make sure your code follows the style and commenting standards used in CS 302 and CS 367.

Electronically submit the following files to the Program 5 tab on Canvas:

- "`ComparisonSort.java`" containing your modified `ComparisonSort` class
- "`Questions.txt`" containing your answers to these questions

Please turn in only the file named above. Extra files clutter up the "handin" directories.

CS367 Programming Assignment 5

Lecture 4, Fall 2016

FAQ

Related pages: Programming Assignment 5

Some frequently asked questions (and their answers)

1. What counts as a data move?
2. The numbers I'm getting are different from the numbers in the sample. Is that important?
3. The times I am getting are really different than the ones in the sample although the number of compares and data moves is about the same. Is something wrong?
4. How far should I take my quick sort and merge sort (i.e., down to subarrays of size what)?
5. My numbers for quick sort are a lot bigger than yours. Is that OK?
6. Am I allowed to modify the `SortObject` class?
7. How do create an array of generics, i.e., `E[]`?

Q1: What counts as a data move?

Any assignment to an item from a class that implements the `Comparable` interface (such as `SortObject`) counts as a data move. For example, given an array `A` containing items of type `E` where `E` implements `Comparable`, the following code does two data moves.:

```
E temp = A[0];
A[0] = A[3];
```

Q2: The numbers I'm getting are different from the numbers in the sample. Is that important?

We are not expecting everyone to get exactly the same numbers as in the sample output. We may or may not have done a small optimization in some places that you may or may not have done. As a general rule, if the numbers you are getting are within a factor of two of ours then you probably don't need to worry about it. So, if the number we got using an array of size 5000 was 60,000 and you got 65,000 or 55,000 then that's OK. But if we got 60,000 and you got 600,000 then that's a problem.

For selection sort, if you use the code from the on-line reading, you'll find that your code does a lot more data moves than our program (e.g., about three times as many on an input array of size 5000). That's ok - when we implemented selection sort, we used a variant of this algorithm which does a lot fewer data moves.

Q3: The times I am getting are really different than the ones in the sample although the number of compares and data moves is about the same. Is something wrong?

Not necessarily. Your times may vary depending on what machine you are running the program on, how many other people are also on that machine, what other programs are currently running, etc. If you find that you keep getting a time of 0 for one or two sorts, you may need to increase the size of the array (to say 7000 or 10000) to be able to measure the running time of the sort(s).

Q4: How far should I take my quick sort and merge sort (i.e., down to subarrays of size what)?

You should do all sorts **as far as possible**. For merge sort, that means you keep doing merge sort until you get to a subarray of size 1. For quick sort, you keep doing quick sort until you get to size 1 or 2. At that point the on-line reading says to do insertion sort on the subarray. That's really not necessary for us (since it requires you to implement a second version of insertion sort); it only takes a couple lines of code to sort a subarray of size 1 or 2.

Q5: My numbers for quick sort are a lot bigger than yours. Is that OK?

No. If you are using the code from the on-line reading, then when quick sort reaches a subarray of size 1 or 2 it switches to insertion sort. But this is **not** the same insertion sort that you implement elsewhere for this assignment. The call to insertion sort in the on-line notes passes the array and low and high indices. What this version of insertion sort is supposed to do is sort the array **only** from index low to index high. The insertion sort you implement for this assignment takes only the array as a parameter and it sorts the **entire** array. So, unless you write a second version of insertion sort, each quick sort on a subarray of size 1 or 2 will have insertion sort sort the whole array (not just the subarray). You can avoid writing a second version of insertion sort by just putting the code that sorts a subarray of size 1 or 2 directly into the appropriate place in the quick sort code.

Q6: Am I allowed to modify the `SortObject` class?

No. You must use the `SortObject` class as it was given.

Q7: How do create an array of generics, i.e., `E[]`?

The logical way to create an array containing items of type `E` (where `E` implements `Comparable`) would be to do something like:

```
E[] myArr = new E[INIT_SIZE];
```

However, Java does not allow you create a new array using generic type so what you'll need to do is create an array of `Comparable`s and cast it to an array of items of type `E`:

```
E[] myArr = (E[])(new Comparable[INIT_SIZE]);
```