## 12 Just when you thought you were done with sorting

I know, I know. By now, you are probably sick and tired of sorting algorithms. You probably want to scream out "No more sorting algorithms!". Go ahead. Do it. It's therapeutic.

(Did you do it yet?)

Now, you are probably also asking yourself "What's the point of learning all these sorting algorithms if they all do the same thing!?!?" Don't worry. This is a common question. So common, in fact, that Randall Munroe, of XKCD fame, felt that it was necessary to add it to the FAQ page of his website:

(from `http://xkcd.com/about/`)

> **Which sorting algorithms should I use? They taught me so many.**
>
> *This is tricky. Most of what they teach you in school is just as an example of how to think about algorithms; 99% of the time you shouldn't worry about optimizing your sorts. Just learn to implement Quicksort (which is very good) and use that without fretting about it too much. People overfocus on efficiency over clarity and simplicity. And most of the time the environment you're coding in will have an efficient sort function built-in anyway.*
>
> *Note: If you're interviewing for a company for a position with a focus on algorithms, the above is not an excuse not to know your stuff.*

Randall has some good words of advice. Here are some reasons why knowing sorting algorithms is important:

1. A large part of computer science involves knowing how to understand and analyze algorithms. You'll see algorithms no matter what area of computer science you are interested in. Sorting algorithms are relatively easy to learn, which is why they are often used in introductory courses.

2. You are pretty much guaranteed to be asked a sorting question at some point if you ever interview for a job at a software company.

Finally, the main reason to learn sorting algorithms is that it impresses the ladies:


(Girls, it works doubly well on guys.)

So Randall is right in that people often overfocus on efficiency. Unfortunately for you, that is no excuse not to know about efficiency. Fortunately for you, in this lab, we will examine the running time of various sorting algorithms. We will use some sorting algorithms that you have already seen, like selection sort and insertion sort, and we will also implement some new sorting algorithms, like quicksort and tree sort. We will time how long it takes each algorithm to sort different types of data and then we will analyze the results to determine what works well and what doesn't work so well in different situations. Note that, in this lab, we will focus only on sorting `ArrayList`s using mutation.

## 12.1 Setup

Take the following steps to setup the lab:

1. Create a new Java Project in Eclipse called Lab12 and import `tester.jar`.
2. Download the provided source code from the course web site and import it into your new project. Here is a list of all the files that are

given, along with a brief description of what is contained in each of the files. For more info, refer to the comments in the code and/or compile the javadoc for the provided code.

- `Algorithms.java`
  This file contains implementations of sorting algorithms. This is where you will implement the additional sorting algorithms from this lab.
- `Examples.java`
  Your standard `Examples` class for the `Tester`. Your tests will go here.
- `IBST.java, Leaf.java, Node.java`
  An implementation of a binary search tree. It should be similar to the one you implemented in a previous homework. You will use these files later in the lab.
- `IntComp.java`
  A `Comparator` class for `Integer`s. You've written the code for this class several times by now so this should look very familiar. We will be sorting `Integer`s in this lab.
- `Sorter.java`
  This class contains wrapper classes for our different sorters. We use this class in order to better abstract our timing framework. Take a look at the wrapper classes for the given sorters. For each additional sorting algorithm that you implement, you will have to create another wrapper class that extends the `Sorter` class.
- `Timing.java`
  This class contains code to perform our timing analysis. You don't need to worry about the contents of this file.

3. Setup your run configuration as usual and run the code to make sure it works.

## 12.2 Verifying that the sorters work

Inspect the output generated by the given code. For each sorting algorithm, you should see the contents of an `ArrayList` before and after being sorted. Convince yourself that the "after" output contains sorted numbers. It would be nice if, instead of visually confirming that an `ArrayList` was sorted, we had a function that did the checking for us.

- In the `Examples` class, implement the `isSorted` method. It should consume an `ArrayList` whose elements are of an arbitrary type `T`, as well as a `Comparator` that is parameterized by the same type. The method should return `true` if the `ArrayList` is sorted according to the given `Comparator` and `false` if not.
  Note: you have already been given a skeleton of the method so you only need to implement the body.

- Now uncomment the tests in the `testSorters` method, which uses your `isSorted` method, and run your project again. Make sure that these tests are now passing.

### 12.3 Timing the sorters

In the code provided, you are given a framework to time how long it takes each sorter to do its work. The results are outputted when the program is run. The numbers that are reported are in milliseconds. The code is set up to test each sorter on three types of data:

1. data that is randomly generated
2. data that is already sorted
3. data that is in reverse sorted order

- Uncomment the tests in the `testSorterTiming` method, run your project, and examine the generated output. Look at how each of the sorting algorithms performs on the different types of data. Make a mental note of the best and worst case scenarios for each sorter (if there is one).
  Note: When looking at the timing results, you should take into account that the resolution of Java's timing system is only about 10 milliseconds. Therefore, if a sorter is very fast, the result may get reported as 0.

### 12.4 QuickSort

Now it's time to add another sorter.

### 12.4.1 Implementation

- Before implementing the quicksort algorithm, setup the timing framework to include the quicksort algorithm. Follow these steps:

  1. Create a wrapper class in `Sorter.java`. You can pretty much follow the pattern of the other wrapper classes already there. Use `quickSort` as the method name.
  2. In the `Examples` class, add a test for quicksort in the `testSortersVisual` method. Again, you can follow what is already there. Just call `visualTest` with a new instance of the quicksort `Sorter` wrapper class.
  3. In the `Examples` class, add a test for quicksort in the `testSorters` method.
  4. In the `Examples` class, in the `testSorterTiming` method, create a new instance of the quicksort `Sorter` wrapper class and add it to the `sorters ArrayList`.
     Note: You may want to hold off on the last step until you are confident that your sorting algorithm is working. You should still do the other steps though, because they will help you debug if your implementation doesn't work on the first try.

- Now that the testing methods are set up, we can implement the actual algorithm. In the `Algorithms` class, implement the `quickSort` method. The `quickSort` method should consume an `ArrayList` of arbitrary type and a `Comparator` of the same type. Here is a description of the steps of the quicksort algorithm:

  1. Choose a random element of the list to be the *pivot* element.
  2. Rearrange the other elements of the `ArrayList` such that anything smaller than the pivot winds up to the left of the pivot in the `ArrayList` and anything larger winds up to the right.
  3. Repeat on each of the less-than-pivot and greater-than-pivot subsections of the `ArrayList`.

  Here are some hints:

  – Since, during each recursive call, we need to keep track of which subsection of the `ArrayList` we are working on, it seems like a

helper method would be useful here. In addition to the `ArrayList` itself and the `Comparator`, the helper method should probably take, as arguments, indices that represent the beginning and end of the current subsection that you are working on.

- Stop recurring when you have a subsection of one or zero elements.
- The hard part of this algorithm is rearranging the `ArrayList` such that the elements of the `ArrayList` that are smaller than the pivot wind up to the left and the elements that are larger wind up to the right. You will probably have to compare each element to the pivot and then swap that element into the appropriate place, depending on if it is larger or smaller than the pivot. Of course, a potential problem is that two elements are needed for every swap operation. An idea would be to scan elements starting from the beginning of the `ArrayList` until you find one that needs to be swapped, and then repeat, except starting at the end of the `ArrayList`. Now you have two items to swap.
  Note: Don't feel that you need to use these hints. There are many possible solutions and if you feel that you have an idea that will work, go for it.
- This algorithm is more difficult than previous sorting algorithms. You may find it useful to do some planning before starting to code. Drawing out some examples is always a good idea, as is exchanging ideas with your parter.

  (You ARE working with a partner, aren't you?)

  (No, that wasn't just a joke, actually go talk to someone else about the problem.)

  (Yes, this means you.)

  (Do it now.)

- When you are are satisfied that your algorithm works, run your project and look at the timing output. The times for quicksort should be faster than the previous two sorting algorithms, especially for larger

`ArrayList`s. If the times for your quicksort algorithm differ depending on the type of data being sorted (the times should always be the same, no matter if the data is random or already sorted), it may have something to do with how you chose your pivot. Did you choose a truly random pivot or did you always choose the first element or the middle element?

### 12.4.2 Analysis

In Big Oh notation, selection and insertion sort are $O(n^2)$ algorithms, where $n$ is the number of elements in the list. This makes sense because in each of those two algorithms, we are approximately comparing each element in the list to every other element in the list. Hence we do approximately $n * n = n^2$ comparisons.

In selection sort, finding the minimum element takes about $n$ comparisons because we have to check every item in the list (yes, we don't always check EVERY item, but we still do enough to qualify as $n$ comparisons). And we look for the minimum element $n$ times, giving us, again, $n^2$ comparisons.

In insertion sort, each time we add an element into the sorted list, we have to compare it to every other element to know where it goes (again, we don't always do the full $n$ comparisons, but on average, we do enough to count as $n$ comparisons). And since we have to insert $n$ elements, we again get $n^2$ comparisons.

However, for insertion sort, note that if the given list is in perfect reverse-sorted order, then we do get to do less than $n$ comparisons on each insert. Actually, we only have to do one compare on each insert, which means that our total run time would be $n * 1 = n$. So for reverse-sorted lists, the run time of insertion sort is only $O(n)$. Does this match up with our timing results? (If you were paying attention, right now you should be saying "wait, wasn't the sorted data set the one that was really fast?" Good observation! There must be something fishy with the insertion sort implementation that you were given. Check out the code if you are interested in finding out.)

As for quicksort, rearranging the list each time so that everything smaller than the pivot is to the left and everything larger is to the right takes about $n$ compares. Now we just have to figure out how many times we repeat this process. When we recur, we approximately cut the list into halves and recur on each half. The number of times we recur is the number of times that

we can cut the list in half. This is approximately $\log_2 n$, which is where the $\log n$ part of $O(n \log n)$ comes from (whenever you see a log in a computer science class, it always means $\log_2$, also known as log with base 2, or "how many times can you cut this number in half?"). Therefore, for quicksort, we have a total of $n \log n$ comparisons!

### 12.5 Tree Sort

You may have noticed that there is also an implementation for a tree sort algorithm in the `Algorithms` class (`treeSort213`). It uses the binary search tree that you implemented previously in this class to do sorting.

- Create a wrapper class for the tree sort algorithm and setup the testing methods to include tree sort, just like we did for the other sorters (the methods in `Examples` that you need to change are: `testSortersVisual`, `testSorters`, and `testSorterTiming`).

A tree sort sorting algorithm works as follows:

1. Add all elements to be sorted to a binary search tree.
2. Find the smallest element in the tree.
3. Add the smallest element back to the list to be sorted.
4. Remove the smallest element.
5. Repeat until there are no more elements in the tree.

Since adding, finding, and removing elements from a binary search tree are $O(\log n)$ operations (bonus points on the lab if you can explain why), in total, treesort takes $O(n \log n)$ time. (Big-oh notation allows us to drop constant factors. So something that runs in $3n \log n$ time still counts as $O(n \log n)$.)

The tree sort algorithm that is given relies on the method `getContentsInOrder` in the `IBST` interface.

- Implement `getContentsInOrder` in the `Node` class. This method should:

    1. Create an empty `ArrayList`

2. Do steps 2 through 5 of the tree sort algorithm above, each time adding the smallest element to the created `ArrayList`.

   Hint: You may find it useful to use `getFirst` and `getRest` to get the smallest element and to remove the smallest element from the tree. (However, you are not required to use these methods. There are other solutions that would work – and in fact, may be slightly more efficient.)
3. Return the filled `ArrayList`

- When you are done, run your project and look at the timing output for `treeSort213`. The time for sorting random data should be fast – comparable to quicksort. But you may notice that the times for the sorted and reverse sorted data are quite slow. Try to come up with reasons why. Think about how elements are added to a binary search tree.

- *Optional* The reason why the tree sort was so slow on sorted data is because the $O(\log n)$ time for add and find (and remove) operations assumes that the tree is perfectly balanced (ie - has equal number of elements on the left and right side). On average this is true, but in certain special cases, this may not be true. For example, if items are added in order, then everything will wind up on one side, so searching, etc. would take $O(n)$ time instead, giving us an $O(n^2)$ sorting algorithm.

  To fix this, we can use a self-balancing binary tree. This data structure is like a regular binary search tree, except that when an element is added, the tree rearranges its elements if necessary to remain balanced. Therefore, each find and search operation is guaranteed to take only $O(\log n)$ time.

  Unfortunately, Java doesn't quite have what we need to implement tree sort with a balanced tree. You can use `TreeSet` to get an approximation, but `TreeSet` removes duplicate elements, so it wouldn't be quite right (if you do want to try this, make sure to use `Integer.MAX_VALUE` as the `max` argument when calling `genRandArrayList` in the `Examples` class, to minimize the chance of generating duplicate elements).

  Fortunately, we can get an implementation of what we need from Google:

1. Download the Google collections `.jar` files from the from this web site:

   `http://code.google.com/p/google-collections/`

   Add the `.jar` file to your project in the usual way (download the .zip file, extract it, and add the `google-collect-snapshot-xxxxxxxx.jar` file to your project).

2. Create a `treeSortBalanced` method in the `Algorithms` class. You should be able to copy `treeSort213` and just replace the `IBST` declaration with a `TreeMultiset` one. You will need to add this import to the top of your file:

   ```
   import com.google.common.collect.TreeMultiset;
   ```

   Creating a `TreeMultiset` is slightly tricky if you want to give it a `Comparator`; you need to use the static `create` method in this case. Check the javadoc provided by Google for more information.

3. Modify the appropriate test methods and run your project. The times for the new tree sort algorithm should be $O(n \log n)$ on all the different data sets.

4. Even though the new tree sort algorithm may seem like it is faster than even quicksort, can you think of a reason why you may not want to use it all the time? Think about what other resources besides cpu cycles are in a computer.

So in conclusion, there is no right answer to the question asked at the beginning of the lab. Different sorting algorithms are better for different situations. Sure an $O(n \log n)$ sorting algorithm is almost always better than a $O(n^2)$ one, but even then that might not always be the case, as we've seen in this lab. (If you don't choose the pivot randomly in quicksort, the worst case is also $O(n^2)$). So in general, Randall may be right that just using a sorting algorithm like quicksort that is $O(n \log n)$ on average should be ok. But hopefully this lab has convinced you that it is still useful to study the different types of algorithms.

As a final note, I'll mention that heap sort and merge sort are two other commonly used $O(n \log n)$ sorting algorithms. In fact, Java's `Collections.sort` method uses a merge sort (more bonus points, if you can add it to our timing framework and see how it compares). Merge sort has one advantage over quicksort in that it is a *stable* sort. A sorting algorithm is stable if it preserves the order of items that the `Comparator` says are equal. Why would you care about the order of items that are equal? Well you wouldn't if you were just sorting Integers, but what if there was more than one way to sort a particular type of item? Remember the balloon example from previous labs? Say I had a list of balloons and I wanted to sort by size, and then for balloons of the same size, I wanted to sort by height. Well if I had a stable sorting algorithm, I could first sort the list by height, and then by size, and I would get the result that I want. If my sorting algorithm were not stable, then the second sort would mess up the results of the first sort, and I wouldn't be able to get what I want. See how interesting sorting can be? If you are interested in learning more about sorting, take an algorithms course!