CSCI 1933 Lab 8

Mergesort and Comparators

Rules

You may work individually or with *one* other partner on this lab. All steps should be completed primarily in the lab time; however, we require that at least the starred steps (★) to be completed and checked off by a TA during lab time. Starred steps cannot be checked off during office hours. You have **until the last office hours on Monday** to have non-starred steps checked by ANY TA, but we suggest you have them checked before then to avoid crowded office hours on Monday. There is nothing to submit via Moodle for this lab. This policy applies to all labs.

IMPORTANT: As announced in the previous lab, you may now use an IDE to do all your programming for the remainder of the course. This lab will involve using JUnit tests to validate the code you write. JUnit is a powerful tool, but unfortunately it does not have a friendly command line interface. You should plan on using an IDE for this lab.

As previously discussed, Java supports taking in a *type* as a parameter. The use of generics provides several benefits, including reducing the amount of casting required by the programmer using the class or method. Since casting objects can carry risk (e.g. how do we know this `Object` is actually a `String`?), the use of generics can make your code safer by moving the type checks from run-time to compile-time.

Comparable

Although all classes automatically inherit an `equals` method from the `Object` base class, we often want to do more than check objects for equality. For example, we might want to know which of two words would come first in a dictionary. Here there are more than two options of 'equal' and 'not equal' – it is possible that the first comes before the second, after it, or even that the two words are the same!

Java provides a number of generic interfaces and classes as part of its standard libraries. Included in this list is exactly what we are looking for: the `Comparable` interface.

The `Comparable` interface defines a single method `public int compareTo(T other)`. Calling the method as `a.compareTo(b)` is equivalent to asking "Is `a` less than, equal to, or greater than `b`?".

Caution: A common misconception is that `compareTo` will return one of $-1, 0, 1$. All that you should assume is that `a.compareTo(b)` will return a negative number when `a` $<$ `b`, zero when `a` $=$ `b`, and a positive number when `a` $>$ `b`. Remember this as being equivalent to a - b in the case of integers.

Merge-sort

While bubble-sort, insertion-sort, and selection-sort all operate in a worst-case time complexity of $\Theta(n^2)$, comparison-based sorting algorithms exist which operate much faster. The lower bound for worst case performance using comparison-based sorting is $\Omega(nlogn)$. Merge-sort attains this performance in all cases, so merge-sort is $\Theta(nlogn)$.

Merge-sort operates on the premise that it is easy to combine (or merge) two sorted lists to get a larger sorted list. For example, to find the smallest element of two sorted lists, we need only examine the first element of each list. Using this knowledge, we can combine two sorted lists totaling n elements in $\Theta(n)$ comparisons and end up with a sorted list! Merge-sort operates by dividing the input data into smaller and smaller pieces, and using this efficient merging to quickly recombine the data into a sorted order.

1 Merging arrays ★

In order to accomplish our ultimate goal of sorting an array in less than $\Theta(n^2)$ time, we need to first merging two sorted arrays in $\Theta(n)$ time.

Create a `Mergesort` class and implement the following methods:

- `public static <T extends Comparable<T>> T[] merge(T[] arr1, T[] arr2)`. This should accept two sorted arrays as input and return a sorted array with their combined contents.
- A `main` method showing that `merge` works. You can use either `String`s or `Integer`s to test this easily.

Think about the edge cases you may run into when attempting to combine two arrays? Will they always be the same length? What happens when you have finished going through the elements of the first array, but not the second?

2 The rest of merge-sort ★

Once we can combine sorted arrays efficiently, the rest of merge-sort quickly falls in to place. Implement the following methods to complete the sorting algorithm:

- `public static <T extends Comparable<T>> T[] mergesort(T[] arr)`. This recursive method should accept an array and return a copy of the array that is sorted. Think about how we can use a divide-and-conquer approach to recursively sort each half of the array. After this, the arrays containing each half should be sorted. We can then use the `merge` method we wrote earlier to combine them. *Hint: What is the base case?*
- Add code to the `main` method you write earlier to test `mergesort`.

3 JUnit testing ★

Up until this point we have tested code primarily using tediously written `main` methods. There are a number of problems with this approach, as you may have encountered. Most notably, if something is broken or not working, there is no easy way to alert the user – they have to scroll through all of the console output looking for an indication of failure.

JUnit solves this problem by providing an easy way to run multiple tests independently and report each of their failures. It integrates with a number of tools and can even be used to test automatically with every change you make.

Assuming you are using IntelliJ, you can easily create a test for any public method by clicking the method name and pressing Ctrl-Shift-T. It is important to change the testing library to JUnit 4, and select all of the methods you want to create tests for (although you can easily create additional ones later).

The general format for a test involves setting up some objects/state, calling a method or methods, and then asserting that either the return value is correct or some state has been modified appropriately. See Listing 1 for an example testing `String.contains()`.

Listing 1: Testing `String.contains()`

```
@Test
public void testMinnesotaContainsMinne() throws
     Exception {
    String minnesota = "minnesota";
    String minne = "minne";
    boolean result = minnesota.contains(minne);
    assertTrue(result);
}

@Test
public void testMinnesotaDoesNotContainIowa()
     throws Exception {
    String minnesota = "minnesota";
    String iowa = "iowa";
    boolean result = minnesota.contains(iowa);
    assertFalse(result);
}
```

Having different test cases for each type of input can help you narrow down exactly what is wrong. However, it's important not to bloat your tests by testing too many of the same type of input – having a few test cases covering the typical use cases of a method, and then at least one more test for each edge case is a good way to catch bugs without writing tests that no longer help you.

To get checked off for this step, do the following:

- Write 3 JUnit tests for the `merge` method.
- Write 3 JUnit tests for the `mergesort` method.
- Explain to a TA why your test inputs are helpful.

4 Sorting with comparators

Now that we've shown our merge-sort algorithm works, let's consider one of the issues with it (which the other sorting algorithms using Comparable also faced). In general, we may want to sort an array in several different ways – thinking back to the `Contact` class we made in Lab 5, it would be reasonable to sort people based on their name, or also based on their phone number! If we always sort using the `compareTo` method, how can we sort data based on different keys?

The answer to this problem is to use a `Comparator`. `Comparator<T>` is a generic interface (similar to `Comparable<T>`) which defines the method `public int compare(T o1, T o2)`. That is, a class implementing `Comparator<T>` provides a way to compare two other objects to each other. We can

define multiple `Comparator` classes, and use objects of each type to sort the data differently. This has the additional benefit of allowing you to compare types which do not implement `compareTo`, by defining your own comparison method without modifying the original class.

Do the following:

- Add `public static <T> T[] mergesort(T[] arr, Comparator<T> comp)` and any other necessary methods to the `Mergesort` class.
- Write a comparator class `ReverseStringComparator` which implements `Comparator<String>`. The `compareTo` method should compare strings based on their last letters first (do not worry about efficiency at this time, think about what built in operations can help you here).
- Write JUnit tests to show `ReverseStringComparator` and your new `mergesort` algorithm work. Be prepared to explain why your tests provide a reasonable assurance that things work in general.

5 A mostly in-place merge-sort (Honors)

This section is aimed at the Honors section, however you are advised to attempt this section *(it is not graded for students not in the Honors section)*

The merge-sort algorithm you have implemented above is slightly inefficient – it constantly creates new arrays and copies all of the data back and forth! While some copying is unavoidable with merge-sort, we can limit all of the extra storage space to the `merge` method!

Define the following methods:

- `public static <T extends Comparable<T>> void merge(T[] arr, int start, int length1, int length2)`. This method should merge two sub-arrays inside of `arr`, and replace these sub-arrays with the merged result. Assume that the first array starts at index `start` and is of length `length1`, and that the second array starts immediately after (at index `start + length1`) and is of length `length2`.
- `public static <T extends Comparable<T>> void inplaceMergesort(T[] arr)`. This method should call the new version of `merge`. Instead of returning the sorted array, it should be modified in place.

As usual, write JUnit tests to demonstrate this method works correctly.