CSCI 1933 Lab 6

Comparators, Sorting, and Big-O

Lab Rules

You may work individually or with *one* partner in lab. We suggest that you have a TA evaluate your progress on each milestone before you move onto the next. The labs are designed to be completable by the end of lab. If you are unable to complete all of the milestones by the end of lab, you have **until the last office hours on Monday** to get those checked off by **any** of the course TAs. We suggest you get your milestones checked off as soon as you complete them since Monday office hours tend to become crowded. If you worked with a partner, both of you must be present when getting checked off to receive credit. You will only receive credit for the milestones you have checked off by a TA. There is nothing to submit to Moodle for this lab.

Attendance

Per the syllabus, students with more than 3 unexcused lab absences over the course of the semester will automatically fail the course. The TAs will take attendance during lab; it is expected that students will be actively working on the lab material. *Your physical presence in lab is not sufficient to be marked as present for the purposes of attendance.*

Sorting Bookshelves

In a previous lab, we implemented a Bookshelf class. In this lab, we will explore comparators and sorting methods to sort the Books in our Bookshelf.

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`?".

Use this information about the `Comparable` interface to write a function for the Book class. This function should compare two books by a specified parameter (author, title, numPages, or genre) and should have signature `public Integer compareTo(Book other, String sortBy)`. `compareTo` should return null if the specified sorting parameter is invalid. (Hint: `compareTo` is already implemented for String objects in Java. You can use it to sort the books' authors, titles, and genres alphabetically. Comparing by numPages case can return the difference of the page numbers.)

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.

Milestone 1: Test your `compareTo` function to see if it works on different author, numPages, etc. Show the results to a TA

Selection Sort and Bubble Sort ★

The Selection Sort and Bubble Sort sorting methods were discussed in lecture. Bubble sort is already implemented in the Bookshelf class(`public Bookshelf bubbleSortBookshelf(String` `sortBy))`. Now implement Selection Sort `public Bookshelf selectionSortBookshelf`.

Note: The function `public static void printBookTitles(Book[] books)` given in the Bookshelf class will be useful for debugging.

Milestone 2: Show a TA that your Selection Sort implementation works

Merge Sort

While bubble-sort and selection-sort all operate in a worst-case time complexity of $O(n^2)$, 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 O(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 $O(n)$ 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.

Merging Two Bookshelves ★

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

Complete the function `public Bookshelf mergeBookshelves(Bookshelf bookshelf, String sortBy)`. This function merges the "self" Bookshelf and passed-in Bookshelf, returning a final Bookshelf in a sorted order. Assume both Bookshelves are sorted to begin with.

Merge Sort Recursive Calls ★

Once we can combine sorted arrays efficiently, the rest of merge-sort quickly falls in to place. Complete the `public Bookshelf mergeSortBookshelf(String sortBy)` in the Bookshelf class.

Note: The function `public static Book[] cloneBookArray(Book[] books, int startIndex , int endIndex)` given in the Bookshelf class will help you make your "subproblems" for the recursive calls.

Milestone 3: Demonstrate your Merge and Merge Sort functions to a TA.

Does it work?

Now you have a hypothetically more efficient Merge Sort algorithm – but does it actually work? Create a variable for the Bookshelf class `int numComparisions`. Increment `numComparisons` every time the `compareTo` function is called. Compare the time complexities of your Merge Sort and Selection Sort implementations.

What are the best, worst, and average runtimes of each sorting algorithm? Are your observations consistent with the Big-O constraints you expected to see?