COM S 228, Fall 2020
Programming Project 2
Sorting and Searching
Due 11:59 pm Thursday October 1

Problem Overview

Sorting is of enormous importance in the practice of computing. Rare is the application that does not include sorting in its implementation. Everything from word processing (spell checking, indexing) and image processing (color histograms), to operating systems (page tables) and computer graphics (depth culling); indeed, the greater challenge is coming up with real-world computing problem that do not require sorting in an efficient solution. Even non-computing applications rely on sorted data; imagine a business with unsorted filing cabinets. If you can't, that's because that business probably wouldn't operate for very long.

Owing to its ubiquity, sorting is among the most studied and well-understood problems in all of computing. Despite this extensive analysis, we cannot definitively claim to have found a "best sort"! QUICKSORT is usually the go-to algorithm, but it is unstable—and sometimes we want stability—and it has highly undesirable $O(n^2)$ worst case behavior, which, while easy to avoid in the degenerate case (through randomization), is impossible to avoid in all cases. MERGESORT is the defacto second choice. It avoids QUICKSORT's problems, being $\Theta(n \lg n)$ (best- *and* worst-case asymptotic running times are proportional to $n \lg n$), and it is stable, but it is not *in-place* (a big problem when either resources are highly limited or data is very large), and it has larger constants, such that it will lose to QUICKSORT in the average case. Adding to this confusion is that both of these sorts tend to lose to well written $O(n^2)$ sorts like INSERTION SORT and SELECTION SORT on small data sets! This is because with small n the constants that big-O brushes aside tend to dominate.

The problem of writing a *general* sorting algorithm is further complicated by the fact that precisely *how* to compare two items cannot be known by the library developer. Java gets around the latter problem with the `Comparable and Comparator interfaces`. All other popular (and reasonable) languages have ways of dealing with this, too. And Java uses a hybrid approach in its choice of algorithm. With certain datatypes it uses a modified QUICKSORT, and with others a variation of MERGESORT[1].

You are an API developer at Oracle in charge of choosing the sorting algorithm(s) for Java 10. Your analysis has already eliminated many sorts from the competition. You're left with QUICKSORT, MERGESORT, and INSERTIONSORT. You decide to build a front-end to these algorithms which compares their performance head-to-head. The current testing is concerned only with sorting text, but even text can be sorted in many different ways: Is is case sensitive or insensitive? Do we want lexicographical order? Alphabetical? Something else entirely? Does it include characters from different alphabets? You decide to pass a configuration file listing character order to your program. You build a lookup table from the configuration alphabet. Your sorts will use a `Comparator` to sort according to the order in the configuration. Finding the relative positions of alphabet characters within your comparator will require BINARYSEARCH, which you must also implement.

[1]Rather unsatisfying explanations for these choices are available in the Java 7 API documentation for `public static void sort(byte[] a)` and `public static void sort(Object[] a)`.

Requirements

You will be implementing three sorts—QUICKSORT, MERGESORT, and INSERTIONSORT—and BINARYSEARCH, as well as a framework for testing and timing them, using the classes described below. You will time their performance using the supplied timer class and compare their actual performance over a range of input sizes compared with their expected performance as predicted by big-O analysis.

Required classes, interfaces, and methods

All classes, interfaces, and methods appearing in the template are required and described therein.

Input

`main()` takes two file names as arguments.

The first file, the *configuration file*, contains an alphabet—a list of characters, one character per line—defining sorted order. There are no invalid characters, except that newlines are used as the seperator and thus cannot be a character in the sorted order.

The second file, the `wordlist`, is a list of words to be sorted according to the order defined in the configuration file. The words are newline seperated and may contain only characters that appear in the configuration file. In particular, note that whitespace characters, like space and tab, are valid letters in our alphabets and do not delimit words!

`main()` will read both files, process the configuration file as needed, then proceed to sort the word list with each of the three sorts. Each list may be sorted multiple times, until such time as each sort algorithm has sorted at least 1 million words.

It is difficult to accurately time very short activities on a computer, because the operating system often does other things to the detriment of your performance analysis. Over long periods of time, these other activities get averaged out, but short programs that happen to run when OS activity spikes do not get the benefit of that implicit filter. Sorting inputs multiple times will reduce the noise in this processes by allowing you to measure an average over many runs.

Output

After all three sorts have finished, report the following statistics to the terminal for each sort:

- Length of the word list
- Total number of words sorted
- Total time spent sorting
- Total number of comparisons
- Average time required to sort the word list
- Words sorted per second

Times should be reported to at least millisecond resolution.

Additional Requirements

We will supply word lists of lengths $n = 10, 100, 1000, 10000, 100000,$ and 1000000[2]. You will run your sorts on all lengths and analyze their performance over these varied ns. Write a discussion analyzing the

[2]An $O(n^2)$ sort with $n = 1000000$ may not finish in a reasonable time for your analysis. If you have the time to let it finish, that's great, and if you don't, that's fine too. In the latter case, simply leave that datapoint out of you analysis.

measured performance compared with expected performance as per big-O. Your discussion should be no longer than 500 words and in plain text format.

Classes

Complete documentation of the classes and methods appear in the code templates. All classes and methods described in the templates are required; however, you may write additional classes and methods as you deem necessary.

Alphabet

The `Alphabet` class encodes the character order.

AlphabetComparator

The `AlphabetComparator` class provides an "alphabetical order" string comparator based on an `Alphabet` object.

InsertionSorter

This is a subclass of `Sorter` that implements INSERTIONSORT in the `sort()` method.

MergeSorter

This is a subclass of `Sorter` that implements MERGESORT.

QuickSorter

This is a subclass of `Sorter` that implements QUICKSORT.

Sorter

The `Sorter` class implements all the common elements of the sorts, provides an interface for an abstract `Sort()` method, and orchestrates statistic gathering of the various sorts.

SorterFramework

`SorterFramework` contains the `main()` method. It's responsible for initializing all of the classes and running the sorts.

WordList

The `WordList` class holds the words to be sorted. It implements the `Cloneable` interface so that we have a straightforward method to run sorts multiple times.

Submission

You are required to include, in your submission, the source code for each of the classes in the code template, as well as any additional classes or methods you may have written to complete the assignment (you shouldn't need any). You need to write proper documentation with JavaDoc for each method in each class. Write your class so that its package name is `edu.iastate.cs228.hw2`. Your source files (`.java` files) will be placed in the directory `edu/iastate/cs228/hw2` (Linux) or `edu\iastate\cs228\hw2` (Windows), as defined in the template code. Be sure to put down your name after the `@author` tag in each class source file. Your zip file should be named `Firstname_Lastname_HW2.zip`. Your discussion should be in a file named `analysis.txt` and stored in the `edu` directory.