An Insightful Empirical Comparison of Sorting Algorithms

Elena Machkasova
Computer Science Discipline
University of Minnesota Morris
Morris, MN 56267
elenam@umn.edu

Abstract

We present a hands-on assignment for students in a junior-level Algorithms class that explores performance of sorting algorithms in practice. The assignment is set up as a competition to develop the fastest program to sort a large data file on a combined criterion: by the length of words and, within each group of the same length, alphabetically. The criterion allowed for a variety of approaches which led students to explore details of various sorting algorithms and to develop data-specific modifications. The paper presents the testing setup and discusses the results and the value of the assignment.

1 Introduction

A classical textbook for a course on algorithms, such as [1], covers several comparison-based sorting algorithms, such as *merge sort*, *quicksort* and *heapsort*, as well as linear sorting algorithms, such as *counting sort* and *radix sort*. Students learn theoretical Big-O approximation, implement some of the algorithms, and discuss conditions under which each of the algorithms should or should not be used. The latter discussion gives students an idea that a choice of a good algorithm is data-dependent and often system-dependent (for instance, relative execution time between two algorithms might depend on the size of an integer implementation on a given system). However, in a typical course students never get to explore such data-dependent trade-offs in practice. In fact, a student finishing a typical algorithms course may easily get an impression that all $O(n \log_2 n)$ algorithms are equivalent in practice.

This is somewhat unfortunate since students' future encounters with sorting will most likely require developing an algorithm tailored to a particular type of data, to the system that the sorting is done on, and even to the programming language used. The author felt that students often do not get a chance to explore and appreciate the beauty and the intricacies of sorting. The assignment presented in this paper gives students an opportunity to design an algorithm for a somewhat more complex sorting criteria than just a numeric field and to explore its performance measured by the actual running time of a program, as opposed to theoretical complexity. The sorting criteria was chosen to allow for multiple approaches and in fact resulted in a variety of basic sorting algorithms (and their combinations) used. In order to increase students' motivation, the assignment was set up as a competition, with the testing data not known in advance. A portion of the grade depended on the placement in the "competition". The assignment also included an analysis phase (students evaluating their peers' algorithm) and a presentation. The assignment was given in the CSCI 3501 "Algorithms and Computability" class in the Fall semester 2010.

The paper discusses the assignment goals and setup in Section 2, describes the details of the sorting criteria and assignment requirements in Section 3, presents results in Section 4, and sums up the experience and outlines possible future improvements in Section 5

2 Class Setup and Goals of the Assignment

In this section we describe the course and the way the assignment is incorporated into it, and list the pedagogical goals. More information is available at the course URL given in Section 2.1.

2.1 The Course and the Structure of the Assignment

The assignment was given in CSCI 3501 "Algorithms and Computability" class in the Fall semester of 2010. This is a 5-credit core course with a lab which is a requirement for Computer Science major (and an elective for Computer Science minor) at University of Minnesota, Morris (UMM). The course web page is located at `http://cda.morris.umn.edu/~elenam/3501_fall10/index.html`. The course includes three 65 min. lec-

tures and a two-hour lab per week. By the beginning of the class students are assumed to be familiar with data structures and Java, including the Java collections library, through Data Structures class (an enforced prerequisite). There were 16 students in the class, ranging from sophomore to senior level, all computer science majors or minors. The assignment required that students work in pairs, although two students elected to work without a group partner for various organizational reasons.
The assignment (referred to as the *sorting competition*) consisted of several stages spread out through several labs (lab assignments 6, 7, and 9) and involved the following:

- The code for a preliminary competition which allowed students to get a sense for how their algorithm compares to others (lab assignment 6),
- A final version of the code for the actual competition (lab assignment 7),
- A correctness analysis of another group's code (before lab 9),
- A final write-up that describes the algorithms and its results and addresses the correctness analysis received from another group (for lab 9)
- A presentation (at lab 9).

The first installment of the assignment took place after the material on all sorting algorithms in [1] was covered. Spreading out the assignment over several weeks allowed students to adjust their algorithms based on the results of a preliminary competition, to analyze another group's solution, to respond to the analysis, and to prepare and present a brief summary of their work.

2.2 Pedagogical Goals

We divide the assignment's goals into two categories: direct ones, i.e. those explicitly listed in the assignment requirements, and indirect ones, i.e. skills that students acquire or improve upon by following the assignment's structure. The distinction is not clear-cut, but even if approximate, it is still helpful for evaluating the learning outcomes.
We identify direct goals of the assignment as follows:

- To design an efficient criterion-specific sorting algorithm by examining the algorithms studied in class as potential "building blocks", and choosing and combining promising ones.
- to get experience with factors that affect program performance in practice, such as a choice of data structures or loop constructs ("for" vs. "foreach" loop in Java),
- to be able to analyze a program for correctness (implementing the sorting criteria and following the guidelines) and for algorithmic complexity,

In addition the assignment has several indirect educational goals:

- To get experience with following specific guidelines for reading, storing, and writing data, as well as integrating the sorting into a larger program. These guidelines were developed to ensure fair comparison (see Section 4), but turned out to be educational on their own. Solutions that failed to follow the guidelines were disqualified from the competition, and thus did not receive any competition points. This consequences were significant enough to motivate students to understand the details of the guidelines and to be careful about following them,
- To be able to write a clear report, both on the students' own program and on the program they analyzed,
- To be able to respond to feedback from another group,
- To be able to give a presentation about the algorithm and to answer questions.

Most of these skills would be helpful for students' future workplace experiences. In particular, this is a good preparation for code reviews that are performed in many companies, but are rarely present in computer science curricula. Technical writing is also highly important at a workplace or a graduate school, but is lacking in many CS courses .

3 The Specifics of the Assignment

3.1 The Sorting Criterion

The most important factor that determines the educational value and the success of this assignment is the sorting criterion. The criterion has to be simple enough as to not shift focus to unnecessary details and interesting enough to be open to multiple approaches rather than allowing only one reasonable choice. It should also allow for a way to easily obtain or generate large input files for convenient testing.

The criterion chosen for the assignment was *sorting a text file on a word length as the primary criterion and alphabetically as the secondary one*. "Alphabetical" is defined as determined by the `compareTo` method of the `String` class. Symbols other than English letters appear in the input text, but the text is guaranteed to be in English, i.e. the majority of characters are English letters. This condition guarantees certain frequency distributions in the input. The theoretical complexity lower bound for this problem is $O(n \log_2 n)$ because comparison is based on `compareTo`, and without knowing the character set one cannot assume any particular range for input characters.

The criterion gives rise to several ideas for constructing an algorithm. A natural approach is a two-pass algorithm that combines a comparison sort for alphabetical sorting, such as quicksort or mergesort, with a counting sort (or a linear-time radix sort) for sorting on word length. These approaches may vary in the order of the two sorting algorithms and in the comparison sort used.

Another approach would be to construct a one-pass algorithm that uses quicksort or another comparison-based sorting algorithm to sort on the combined criterion, i.e. on both the word length and the alphabetical order. Note that a potential issue with using quicksort for this

problem is a large number of identical elements (such as the word "the") which leads to quadratic performance of quicksort.
A yet another approach is to use a linear sorting algorithm (a counting sort or a radix sort) for both numeric and alphabetical sorting. If implemented directly, this approach fails on non-alphabetical characters. However, if a "fall-back" mechanism is used for handling the rest of the elements (for instance, for inserting them into the sorted array), then there is a hope of getting times close to linear. The rich spectrum of approaches to the problem makes it ideal for the assignment.

3.2 The Program Requirements

The initial assignment was phrased as follows[1] (see lab 6 at the URL in Section 2):

> Develop a sorting algorithm to sort words in a long text file (a few thousand words) by the following criteria: by word length as the primary criterion and alphabetically as a secondary one. This means that all shorter words come before all longer words and within each group of words of the same length the words are sorted alphabetically (more precisely, according to `compareTo` method of `String` class).
>
> Your goal is to develop an algorithm that sorts as fast as possible. "Fast" here means direct timing of the program, not just the Big-Theta approximation. You will get a text file to practice, but the final test will be done on a different file.

Additionally the following requirements were specified (some details omitted):

1. You must use Java. While this may not be the best language choice for efficiency purposes, this assignment is not about languages, it's about algorithms. Thinking about implementation details is important, however.
2. The input file may contain data that is not just letters, but most of it is English words. Alphabetical comparison must be done based on `compareTo` of `String` class. Note that punctuation marks attached to words are considered a part of the word (e.g. when? has the length of 5).
3. Your program must read data from a file, store it internally as an array, sort it, and then output it into a file.
4. The program must take three command-line arguments: the name of the input file, output file, and the number of loops. The last argument will allow repeating the sorting multiple times to get more accurate times if needed.
5. The program running time is determined using a predefined function `System.currentTimeMillis()`. The timer starts after you read the file and stops before you start writing out the result.

[1] Some grammatical errors have been corrected.

6. Your sorting starts by copying the given array into a new array (in a loop) and then sorting that array in place. This is so that you can repeat the sorting multiple times. Since the copying time is the same for everyone, it will not affect the competition.
7. Only the last copy of the sorted array should be copied to the resulting file.

The students were given a large file (a plain text version of Shakespeare's "The Tragedy of Hamlet, Prince of Denmark", which has about 20,000 words) to practice on.

3.3 Explanation of Requirements and Testing Setup

While it is interesting to compare program running times for different programming languages, this would not be a fair competition since not all students know the same languages. Besides, there are many real-life situations when software developers do not have a choice in the language they use. Thus we require that all group use Java (see item 1 in the requirements list in Section 3.2). The items 2, 3, and 4 specify the format of data, clarify the comparison criterion, and specify input and output requirements.

Items 5 and 6 specify the details of time measurement of the program. Taking accurate time measurements of the sorting portion of a program is a non-obvious task. First, one needs to exclude file reading and writing from the timing since these processes are time-consuming (thus obscuring the sorting time) and dependent on operating system events (such as file opening) that are difficult to control or measure. Thus file reading and writing are excluded from the time measurements: the timer starts after the input file is read and stops before the writing of the output file begins.

Excluding input and output, however, is not enough to get accurate and useful measurements. Sorting even a large data set just once might result in times that are too small and too close to each other to demonstrate differences between programs. To counteract this problem, the programs are set up to repeat sorting in a loop, where the loop is controlled by a parameter passed on the command line. A few issues arise at this point, discussed below. The first one is the fact that every sorting loop has to start with the same unsorted data. Thus the first task performed in the sorting loop is to create a new copy of the data array (or `ArrayList`, depending on the choice of storage). While this is not a part of sorting, it is an operation that might appear in a real-life program before sorting is done.

The second issue is the fact that starting and stopping a timer in a program results in system calls and may not happened exactly at the point specified by the programmer due to dynamic program optimization. In our experience multiple calls to the system time function within the same program might produce the times that do not add up to the total time of the program. To maximize accuracy, we require that there are only two calls to the time function: one before the first iteration of the loop and the other one after the loop is done. This way of timing includes the copying (in the beginning of each loop iteration) into the measured time. The overhead is just one pass through the array and should be approximately the same for all programs. One should be careful, however, to avoid inefficient copying operations, such repeated calls to `add` method of an `ArrayList` (an efficient way of copying an `ArrayList` is `clone` method).

Another important issue that needed clarification was the amount of *preprocessing* that a program is allowed do. Preprocessing is defined as any operations performed before the first time reading is done, typically while the input file is being read. After a series of questions from students about what is acceptable in the preprocessing stage we came up with the following requirement: operations performed during preprocessing must take no more than a linear time and generate a constant amount of information, where the constant is known before the file is being read. Thus allowed operations include getting information about the maximal word length in the input file or determining whether a certain character appears in the input file. Operations that are disallowed are creating an array of all characters appearing in the input file or pre-sorting the input data into "buckets". Additionally any data structures that store data in a specific order (such as sorted) are not allowed in the preprocessing phase. An example of such a structure is `HashMap` in Java.
Programs were tested by running a script in the `/tmp` directory on a LINUX machine. Unlike users' home directory, reading and writing a file in `/tmp` directory does not require an access to a file server, and thus has fewer unpredictable delays. The script runs each program five times and records the times. It will also check that the output file is sorted and has the right number of lines. If resulting file is incorrect then the results are not counted.

3.4 Correctness Analysis, Final Write-up, and Presentation

After the final competition each group was given program code of another group and asked to check correctness of that program. More specifically, the assignment was:

> Correctness analysis: for the program given to you please verify that all the rules have been followed, including the following:
>
> 1. The pre-processing phase is not performing tasks that should be done in sorting. I know this is informal, but all algorithms are different. Finding out the range of word lengths is OK, verifying the presence or the absence of certain characters is OK, doing any sort of counting is not OK. A non-sorted data structure (such as an array or `ArrayList`) must be used for storing the initial data.
> 2. The loop contains a copying phase and a sorting phase. The copying phase is performing a copy of the original array/`Arraylist`. The list to sort is unsorted after the copying phase.
> 3. The timing starts before the loop and ends after the loop. The loop runs the specified number of times.
> 4. The resulting array at the end of each loop contains all the original elements and is sorted according to the sorting criteria (i.e. the "alphabetical" ordering is consistent with `compareTo`) no matter what text input is given to the program.
>
> Additionally you may analyze a sorting program from any group other than the one that the one you are assigned. This will not give you extra credit, unless

you were **the first to discover a problem that the assigned reviewing group missed**. This part can be done individually or in groups.

Note that pre-processing requirements can be easier phrased by allowing only constant-size output, with the constant known ahead of time. See Section 3.3 for a more detailed discussion. However, we came up with the shorter and clearer description only after this assignment was posted.

The correctness analysis pointed out a few problems in the solutions. Minor issues that did not affect the sorting or the times resulted in some point taken off, but did not disqualify the solution. One group did not include the copy phase into their timing and their results were disqualified from the competition since they were inaccurate. Another solution generated a substantial discussion about validity of their results. The copying phase rules have not been strictly followed, i.e. they were not copying the array into a new one. However, an equivalent copying phase was done at the end so the results were contained in a new array and did not overwrite the original one. While points were taken off for not following the requirements, the results were not disqualified since the complexity was equivalent.

The final part of the assignment was a final write-up and a presentation:

> You need to send me a write-up and 2-3 slides that you will be presenting on Thursday in the lab. Both group members will be presenting. Know who is saying what and practice your presentation at least once. Please keep it short (3-4 minutes), be prepared to answer questions.
>
> Guidelines for the final write-up:
>
> 1. Describe your algorithm. For each sub-algorithm that you use (such as counting sort or insertion sort) make it clear what the sorting keys are. If you use radix sort, explain what sub-algorithm is used for sorting. If you use a sorted data structure, please clearly explain what it is and how the data is ordered.
> 2. Specify what structure you are using for storage and how you copy it.
> 3. For each of your algorithms explain what its running time is in terms of Big-theta of the worst case. If you are making a claim that your time on the given data is in a different Big-theta class than the worst case, please explain why. Also, estimate constants by the number of times the array gets traversed.
> 4. Explain why the sorting is correct with respect to the given criteria.
> 5. List all of the additional non-constant memory used and what it is used for.
> 6. Address any correctness concerns that you get from the group that checked correctness of your algorithm. If you agree with their analysis then simply say so.
> 7. Please list the place that your algorithm got in the competition.
> 8. How well would you say your approach worked? Why?

9. If you would like, write down what you could have changed to make the program run faster.

Your slides should be a summary of your final write-up. You don't need to explain the correctness, unless there are issues. You should give the summary of reasons for your time analysis. Make sure to list the time that your program got in the final test and its place. How well would you say your approach worked? Why?

The correctness analysis phase and the ability to respond to the analysis prompted students to analyze the code (both their own and the other group's) very carefully. Several in-depth discussions were prompted, especially when the certain data structures (such as `HashMap` and `Stack` in the Java collections) were used. These discussions have a significant educational value since they prompt students to look into the details of predefined structures and algorithms.

Presentations were open to students not in class and faculty. Students got a chance to present technical material to their peers and to answer questions. It also gave students an opportunity to see many different approaches to the problem. One improvement that we would like to make in the future is to present a summary of all algorithms (similar to the table in Figure 1 in this paper) to give a better view of relative performance of algorithms.

4 Results and Lessons Learned

The table in Figure 1 presents summaries of all the algorithms and their resulting times at the final competition. The sorting was tested on a plain-text file containing the text of Herman Melville's "Moby-Dick" which is 215,137 words. Each program was ran 5 times. The "Times" column in the table gives the time ranges for each program (in milliseconds), excluding a couple of outliers that likely were due to an unrelated system event. The table also notes any correctness issues that were encountered. Two programs ran with overlapping times and share the 5th and 6th places. The last algorithm was disqualified due to an incorrect timing setup (the timing did not include the copying and was therefore not comparable to others).

The table demonstrates a variety of approaches to the problem. Almost all of the algorithms covered in the class appear as elements of solutions: quicksort, merge sort, radix sort, insertion sort, and counting sort. Most of the groups used a two-pass approach, sorting separately by length and in alphabetical order. However, the order in which these sortings were applied differed. Data structures used in the programs as intermediate storage for separating the data by word length included arrays, `ArrayLists`, and, in the case of one group, a `HashMap`.

The table of results makes it clear that merge sort is not an optimal algorithm for this problem, regardless of whether it is done before or after sorting by length (or by itself, as it is the case for the group that placed 8th). This is somewhat surprising since the data has lots of repeated elements, such as numerous occurrences of the word "the". Since these elements are considered already sorted, one would expect that they will make merge sort a reasonable choice. However, it seems likely that the overhead of handling extra storage

Place	Time (ms)	Algorithm	Preprocessing	Notes
1	121 - 126	Pass 1: sort into “buckets” by word length Pass 2: alphabetical sorting by quicksort modified not to sort equal elements	Find the max word length	
2	144-152	Pass 1: sort into “buckets” by word length using `HashMap`. Pass 2: for each bucket (array): insertion sort for small arrays, radix sort on a range of 93 chars for large arrays; fallback merge sort when out-of-range chars are present	Find the max word length	returns an array (not in place)
3	212-222	Pass 1: counting sort by word length Pass 2: radix sort on English chars within each word length, then insertion sort for other chars	Find the max word length	
4	460-475	Pass 1: merge sort alphabetical Pass 2: counting sort by length	Find the max word length	
5-6	533-572	Pass 1: counting sort by length Pass 2: merge sort alphabetical	Find the max word length	
5-6	543-550	Pass 1: merge sort alphabetical Pass 2: push into stacks by length Finally: copy from the stacks	None	Slightly incorrect timing
7	680-716	Pass 1: merge sort alphabetical Pass 2: counting sort by length	Find the max word length	One line short
8	686 - 730	Merge sort, combined criterion	None	
–	–	Pass 1: sort into arrays by length Pass 2: merge sort alphabetical	Find the max word length	No place: wrong timing

Figure 1: Summary of algorithms and results

proportional to the length of the array may create inefficiency that outweighs the advantage of repeated elements.

It is quite interesting to examine the top three algorithms. The best solution uses quicksort. This is somewhat expected since quicksort is known as the most efficient algorithm in practice in most cases. However, a large number of identical elements lead to quadratic behavior of quicksort even with a randomized pivot selection. However, [1] proposes a modification of quicksort in one of the exercises. In this modification the array is partitioned into three groups: elements smaller than the pivot, elements larger than the pivot, and those equal to the pivot. The latter elements are not passed to the recursive calls and do not participate in any future comparisons. This version of quicksort turned out to be very well suited for this problem.

The next two algorithms are also quite efficient and can be characterized as data-specific. They attempt to get as close to linear time as possible on the majority of elements. Both algorithms use radix sort for the most common subset of characters (a range of 93 common ASCII characters in the 2nd place algorithm and letters of English alphabet in the 3rd place one). Both algorithms use insertion sort: one to sort small arrays and the other one to add words that have characters out of range. Since insertion sort is close to linear on almost sorted data, it does not create a substantial overhead. One of the groups also uses merge sort in cases when the data has unexpected characters, but this case has not occurred in the testing data. It is also interesting to observe that these algorithms were developed via a sequence of improvements which gave students an opportunity to try and fine-tune different approaches.

Earlier attempts included other sorting algorithms. One group used bubble sort at the preliminary competition which was drastically slower than the other sorting approaches even on a fairly short file (the file used for the preliminary competition was 24,240 words, the sorting loop was repeated 10 times). Bubble sort time was about 14,000ms, whereas the second slowest algorithm was about 650ms. Seeing the difference with their own eyes (in fact, waiting for the program to finish) gives students a much better appreciation of the difference between $O(n \log_2 n)$ algorithm and a quadratic one.

5 Conclusions and Future Work

The assignment gave students an opportunity to study sorting algorithms on a concrete data example and compare them based on actual program performance. The competitive nature of the assignment stimulated students' interest and caused them to try different approaches, prompting a fascinating hands-on study of sorting. The chosen sorting criteria worked perfectly for the goals of this assignment by allowing multiple potential approaches.

There are several aspects of the assignment that can be improved or built upon. This was the first time the assignment was given, and some requirements had to be fine-tuned as the work progressed. The resulting requirements are clear and useful which will make it easier to give this assignment in the future.

In the future it would be worthwhile to add another phase to the assignment: a discussion of the results and possibly a write-up on comparison of the algorithms. The correctness analysis was very useful, but it did not analyze why some algorithms performed better than

others. The final write-up asked students to explain the performance of their algorithm, but did not ask them to look into the differences between the faster and slower solutions. This question can been added to the final write-up or as a separate assignment.
Perhaps the most important future work item is to come up with other sorting criteria that allow for many approaches based on sorting algorithms typically covered in the course, which would prompt an exciting competition. Suggestions for such criteria would be appreciated.

6 Acknowledgments

The author thanks students in the Fall'10 offering of the CSCI 3501 course for making the class, including the sorting competition, a successful and enjoyable experience.

References

[1] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, USA, 2009.