Christian Znidarsic
EN.605.202 Data Structures
5/3/22

Lab 4 Analysis

Data Structure Description

For this lab, one primary data structure was used.

Linked List

This project implements two linked lists, both used for the merge sort. One linked list is a list of integer values. This list contains the data of the input arrays and the output sorted arrays.

The second linked list is a list whose nodes are other linked list nodes. This list is used in the merge sort to hold header nodes of runs that were found in the unsorted list.

Design

My program works using 8 total classes: Main, MergeSort, QuickSort_LowPivot_Insertion_50, QuickSort_LowPivot_Insertion_100, QuickSort_LowPivot, QuickSort_MedianThree, linkedList and ListNode. The most significant classes are described in-depth below.

Main

The main class reads the input files and calls methods from other classes to sort them.

MergeSort

The MergeSort class performs a linked implementation of a natural merge sort. It leverages two recursive methods, sortedMerge() and sortRecursion() to perform the sort. It also uses an iterative method called findRuns() to check the entire input for runs. These runs are then put into nodes and sorted. This aspect of the class is what makes it a "natural" merge. It eliminates redundancy be leveraging pre-sorted areas of the input. In the best case, the input is already sorted, and the merge sort performs zero swaps.

QuickSort_LowPivot

The low pivot version of quicksort takes the lowest element in the unsorted input array and sets it as the pivot. This method is inefficient, especially for pre-ordered data. It works by identifying a pivot and placing all elements less than that pivot on the left side of it, and elements greater than the pivot on the right side. It then recursively partitions both halves repeatedly until the array is completely sorted.

QuickSort_LowPivot_Insertion_50 and QuickSort_LowPivot_Insertion_100

These versions of quicksort leverage insertion sort for partitions less than 50 and 100 elements long respectively.

QuickSort_MedianThree

This version of quicksort finds the pivot in a unique way. It takes the first, middle and last elements in the unsorted partition and sorts them. Then, it takes the middle element to be the pivot for that partition. This variation is effective because it helps offset the issues with pre-sorted data. It manages to get closer to finding the real median of the data for pivots, which increases the algorithm's efficiency.

Justification

To me this ordering made sense because main only performs user I/O and very general operations like calling other more complex methods.

Sorts are encapsulated within their own classes by their respective names. They then act upon external data structures. This allows attributes of the sorts to exist within sort classes, namely the "comparisons" and "swaps" fields. This organization also increases readability.

Algorithm Efficiency

QuickSort vs MergeSort

The most significant direct comparison in this lab is between QuickSort and MergeSort. Based on the data, it is clear that MergeSort is vastly superior to QuickSort when it comes to sorting pre-sorted data. This conclusion applies to both ascending and descending ordered data. When sorting pre-sorted data, QuickSort's number of comparisons sky-rockets up closer to O(N^2), whereas the natural MergeSort takes advantage of the pre-sorted data, running in just O(N) time. With regards to swaps, for the ascending-ordered data, MergeSort makes zero swaps, and QuickSort makes N swaps.

When it comes to the random data input, however, things get much more close. For randomly ordered input data, MergeSort only just barely beats the QuickSort variations with respect to comparisons. For swaps, the QuickSort variations insertion 100, insertion 50, and median of 3 perform better than MergeSort, but not by a lot. With these results in mind, a QuickSort may be the better option in scenarios where randomized data is prevelant, and pre-sorted data is rarely encountered.

Effect of Data Order

Of great interest is the effect that the order of the unsorted data has prior to being sorted. In this lab, we have tested three different orders of data: ascending order, descending order, and random order.

It is interesting how providing data in either ascending or descending order brings out behavior in some sorts that isn't otherwise seen. Most importantly, the ascending/descending inputs cripple the QuickSort variations, because they struggle to find pivots that are representative of the data. For example, in the pre-ordered data, the pivot for the "low pivot" QuickSort variation is always chosen to be the lowest number in a given partition. This pivot choice results in the most number of comparisons, because every value in the partition must be compared against the low pivot. The exception is the medain of three variation, which manages performance much better than that of the other QuickSort variations. In contrast, the median of three variation can find the median values for pivots in the pre-sorted data, minimizing the damage caused by the data. However, it still does not come close to MergeSort when sorting pre-ordered data.

The ascending ordered data helps MergeSort incredibly, requireing a mere N-1 comparisons to sort and zero swaps. This is due to the "natural" implementation of the MergeSort, which takes advantage of the pre-ordered data.

Effect of File Size

As expected, larger files take longer to sort. Before starting this lab, I expected to see O(Nlog(N)) performance from both MergeSort and QuickSort on average, and that is exactly what I found.

Looking at the graphs provided in this submission, for the random data inputs, the Nlog(N) line serves nearly perfectly as a lower bound of these algorithms, with only MergeSort just barely beating it out. The rest are above only by small margins. The same applies for the number of swaps.

Effect of Using a Natural Merge Sort

Compared to a regular MergeSort, the Natural MergeSort implemented in this project takes advantage of pre-ordered data to greatly boost efficieny in the case of data that is either partially or completely sorted. It does this by first taking an iterative pass across the entire input data, which takes O(N) time. As it iterates through, it finds runs of the data that are already sorted, and then stores those runs as nodes. The resulting nodes are then used in the merge sort. This technique can greatly reduce the number of starting nodes in the sort, which as a result greatly reduces the sorting time in data that is already partially sorted.

Justification of Recursion Over Iteration

For this lab, I chose to write all sorting methods recursively. I did this for a few reasons. For one, I knew that writing the methods recursively would be more simple, and easier to debug, as opposed to the iterative solution which would require more counters and be vulnerable to more off-by-one errors. Another reason was that these algorithms were taught to us recursively in our readings, so I am more familiar with the recursive implementations of sorting algorithms. A third reason is that I wanted to gain more experience with recursive programming. The downside, or course, is efficiency. In a real-world application, recursion may not be the best way to write these sorting algorithms. An iterative solution may yield faster results. However, the point of this lab was to compare and contrast, so as long as all of the sorting algorithms were written recursively, they could be compared against one-another fairly.

Which Factor Has the Greatest Effect on Efficiency?

It is clear that the order of the data has the biggest effect on the sorting efficiency. This is due to the drastic effects pre-ordered data has on some of the QuickSort variations. Thus, when choosing a sorting algorithm, perhaps the most significant and important question the designer must ask is whether that data they will be working with may already be sorted regularly, or if it is almost always unsorted.

Space Complexity

The expected space complexity of QuickSort is O(log(N)), with the worst-case being O(N). In comparison, the average expected space complexity of MergeSort is O(N). Thus, QuickSort may be a better choice over MergeSort in applications where memory and space are critical to the system. This could be the case for small embedded systems that lack large stores of memory.

An interesting issue that I came across during this lab was stack overflow errors. These errors were caused several of the QuickSort variations, and only happened when large ascending or descending ordered data were input (random data was fine). It turned out that, because these are recursive functions, the input was broken down into so many sub-arrays that the stack overflowed. To fix this error I had to increase the stack size in Eclipse so that the program could run. This example goes to show the large space requirements for both MergeSort, and recursive functions.

Reflection

Biggest Strength

I am proud of the work that I did on this lab. I did a good job of creating sorting algorithms as well as analyzing the differences in performance between the variations. I took my time creating an Excel spreadsheet to accurately visualize the data and to allow myself to fully understand the performances of the algorithms. The spreadsheet I made proved invaluable when writing my comparison, because I had all the data visualized in front of me.

Biggest Difficulty

I thought that writing the sorting functions would be easier because we were permitted to use algorithms from online, but it proved to be quite difficult. I started with algorithms from geeksforgeeks.com for QuickSort and made adjustments to fit the lab criteria. Due to the various implementations, this proved to be a difficult task. The merge sort was most difficult because it required a linked implementation, as well as a natural aspect. I am pleased with the algorithms that I wrote and how they performed.

Conclusion

I am very proud of my work on this lab. I spent a good amount of time on this lab and I am happy with the result. I have learned a great deal about the efficiencies of various sorting algorithms, particularly the effects that the algorithm *and* the data have on the sort. I was surprised to see just how drastically the order of the input data effects the number of comparisons and swaps of a given algorithm. I have certainly gained a better understanding of how there isn’t a single sorting algorithm that is best. The choice of an algorithm should be made primarily based on how the algorithm works with the data it is meant to sort, not purely on the algorithm.