Lab 4 Analysis

Arta Seyedian

Program

This program reads all files found in a directory and, one by one, loads the data found therein and sorts it according to the 5 different sorting algorithms outlined in the program requirements. The program first gets and stores the file names in an array and then sorts each file after reading and storing all the values found in the file first as strings in an ArrayList, which are then transferred as integers into a regular array set to the size of the ArrayList. The program then creates two sort objects, one for quicksort and another for natural merge sort. The quicksort contains four different variations: a standard quicksort, a quicksort that is limited to partitions of 100 or greater, a quicksort limited to partitions of 50 or greater, and a Median-of-Three quicksort. The limited quicksorts (quicksort 100 and quicksort 50) are implemented with insertion sort to take care of all partitions smaller than the limit. Median-of-Three quicksort finds the median value between the first, middle, and last elements and uses that as a pivot. natural merge sort is a type of merge sort that finds previously-occurring sorted sublists in the array and incorporates them into the merge sort.

Note: this lab was complicated by the fact that I would routinely get stack overflow error 4/5 times when trying to run. Still, I would be able to get the program to run, but as a result, I had to temper many of the features I could have given this program with this fact.

Analysis

In the home directory, there should be an Excel spreadsheet with the name Time Values.xslx. These values represent the time, in nanoseconds, difference between when the sort began to when the sort ended. Using Excel's 'Data > Filter' tool, one can learn from these outputs.

Some of the results are unsurprising. For example, the fastest sorts were the ones done on file sizes below one thousand. However, something that does stick out to me is that, for example, like the median-of-three quicksort giving an order of magnitude faster sort time for the rev10k file than for the rev5k file. This could be due to some sort of technical hardware or code issue, but was hard to ignore, nonetheless.

The reverse files were apparently better serviced by quicksort than by natural merge sort. This is probably since none of the items are in order in a natural merge sort, which is to its disadvantage, because natural merge sort exploits pre-existing order found in the files to perform the merges of the sublists. This is also attributable to the fact that in general, quicksort is faster, performs more comparisons. The difference between quicksort's rev1k processing and natural merge sort's rev1k processing is at least one order of magnitude.

The slowest sorts, by far, were the natural merge sorts for the large files. Among the quicksorts, the standard quicksorts were the slowest. Between quicksort 100 and quicksort 50, quicksort 100 was slower. Insertion sort is a low efficiency sort on average compared to

quicksort, however, in best-case scenarios, insertion sort can be even more effective, which is why quicksort 50 was more efficient than quicksort 100 and quicksort standard in some cases.

For the duplicate files, again, natural merge sort pales in comparison to quicksort. Interesting here is that standard quicksort was reliably inefficient in sorting these duplicate files, and quicksort 100 had the highest efficiency for these duplicate files, demonstrating that duplicates function as being "already sorted," i.e. they are digits that do not need to be sorted yet contribute to the overall size of the dataset, cutting down on resource expenditure. Quicksort is not as efficient as insertion sort in near-sorted sets, and this further lends evidence to that.

For the ascending files, quicksort 100, again, led the pack. What was surprising for me is that natural merge sort was not as effective as quicksort 100 in dealing with the ascending files, which is where it should outperform quicksort. This suggests to me that my code for natural merge sort might be flawed, or the relationship between ascending data values and resource conservation in natural merge sort may not be quite that simple. Regardless, quicksort 100 outperformed the rest of the sorts once again.

Lastly, again, quicksort 100 has a better rate than the other sorts. Another thing I've noticed in calculating the rate of sorting is that smaller files tend to have a higher rate of calculation in nanoseconds than the larger files (total time of sort over number of entries). This could potentially be due to the resource cost of accessing the array and undergoing all the various allocations required for each sort.

Overall, I would say that quicksort 100 or 50 (depending on the size of the data file) is the best, most effective sort. An exception could possibly be made for natural merge sort and ascending files; however, my data does not suggest that and I'm only leaving that possibility open because there may be some flaw in my code. Median-of-three did not prevail in any of the analyses that I made but generally tended to be better than standard quicksort.

REFERENCES

Insertion Sort - GeeksforGeeks. (n.d.). Retrieved November 30, 2018, from https://www.geeksforgeeks.org/insertion-sort/

Merge Sort for Linked Lists. (2010, June 6). Retrieved November 30, 2018, from https://www.geeksforgeeks.org/merge-sort-for-linked-list/

Quick sort with median-of-three partitioning : Sort « Collections « Java Tutorial. (n.d.). Retrieved November 30, 2018, from http://www.java2s.com/Tutorial/Java/0140__Collections/Quicksortwithmedianofthreepartitioning.htm

quicksort - GeeksforGeeks. (n.d.). Retrieved November 30, 2018, from https://www.geeksforgeeks.org/quick-sort/