**Quicksort and Natural Merge Sort**

**Brandon L Morrow**

**Sort Analysis**

Sorting is a very useful and necessary tool that can be implemented in many ways. Each type of sort has its own benefits and drawbacks that make one sort more appealing than another depending on the structure of the data needing sorted. From analysis of runtime, comparisons, and swaps we can identify which sorting methods are optimal based on the ordering of the data needing sorted. To optimize sorts multiple sorting techniques can be implemented together to further decrease runtime.

First let us look at the worst-case scenario for most sorting algorithms: reverse ordered data. We see that the hybrid implementation of quicksort which uses insertion sort on partitions of 50 or less proves to be the most efficient algorithm both by time, and by the number of comparisons and swaps. It is slightly more efficient than the hybrid implementation where partitions of 100 are insertion sorted. This would suggest that insertion sort optimizes quicksort, but only when implemented for lower sized partitions. There seems to be a performance drop off when utilizing insertion sort on partitions under 100 vs under 50 with the smaller partition being faster.

Continuing with reverse ordered data, as we would expect we see that the median of three technique to choose a pivot is more efficient than picking the first element as a pivot for reverse ordered data. This is due to a middle value pivot being selected instead of the extreme first value being used as a pivot. While the number of comparisons remains the same for both pivot selections the median of three technique executes less swaps and is therefore a more efficient algorithm than simply picking the first element of an ordered list. Natural Merge sort shows that reverse ordered data is its worst-case scenario and had a significantly higher number of swaps than any other technique for reverse ordered data.

Now looking at ascending ordered data we see that the opposite with natural merge being much more efficient than quicksort. We see natural merge have a similar number of comparisons to other techniques, but substantially less swaps than any of the quicksort algorithms. We see that insertion quicksort is also more efficient here than standard quicksort or median of three quicksort. We see standard quicksort being the worst of the four which is to be expected with it selecting an extreme pivot at the first location.

Now looking at randomly ordered data we see that quicksort is far superior to natural merge sort with the quicksort hybrid implementation being the preferred method. There were substantially less comparisons done by the quicksort algorithms than there were for the natural merge algorithm. The fewest comparisons and swaps were by the median of three implementation which shows that pivot selection is crucial.

Selection of an appropriate algorithm is crucial depending on what type of data is being analyzed. We can see that if we wanted to add to an already sorted list and then sort that list again, we would want to use a natural merge sort. If we had a random set of data, then we would want to use the median of three technique along with an insertion sort to make sure the algorithm is optimized. Lastly, for reverse ordered data we would also want to use a hybrid quicksort that utilizes an insertion sort. While minimizing comparisons, swaps, and therefore runtime is important, memory implications must also be considered.

Merge sorts biggest downfall is that it requires O(n) additional memory to create the final merged list. Quicksort only uses O(logn) additional memory. The benefit of natural merge sort is that merge sort is a stable sorting algorithm that retains the order of the original data. Quicksort utilizes the space that the data is located in and as a result is an unstable algorithm since it does not retain the unsorted array in its base form. This is another consideration when selecting a sorting algorithm is if the original ordering needs to be maintained.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Reverse Integer Sorting** | | | | | | | | | | |
|  | **Natural Merge** | | **Quicksort** | | **Quicksort Insert-50** | | **Quicksort Insert-100** | | **Quicksort Median** | |
| **Sorted** | **Comparisons** | **Swaps** | **Comparisons** | **Swaps** | **Comparisons** | **Swaps** | **Comparisons** | **Swaps** | **Comparisons** | **Swaps** |
| **50** | 123 | 700 | 1249 | 49 | 1274 | 1225 | 1274 | 1225 | 1249 | 49 |
| **1000** | 2621 | 252200 | 501248 | 1048 | 334074 | 1858 | 335448 | 5649 | 49999 | 999 |
| **2000** | 7619 | 1255200 | 2499998 | 2998 | 1334473 | 2574 | 1335816 | 6416 | 1999999 | 1999 |
| **5000** | 20117 | 7512700 | 14499998 | 6998 | 8335473 | 4574 | 8336816 | 8416 | 12499999 | 4999 |
| **10000** | 45115 | 32527700 | 62499998 | 14998 | 33337074 | 7858 | 33338448 | 11649 | 49999999 | 9999 |
|  |  |  |  |  |  |  |  |  |  |  |
| **Ascending Integer Sorting** | | | | | | | | | | |
|  | **Natural Merge** | | **Quicksort** | | **Quicksort Insert-50** | | **Quicksort Insert-100** | | **Quicksort Median** | |
| **Sorted** | **Comparisons** | **Swaps** | **Comparisons** | **Swaps** | **Comparisons** | **Swaps** | **Comparisons** | **Swaps** | **Comparisons** | **Swaps** |
| **50** | 50 | 100 | 1274 | 49 | 49 | 0 | 49 | 0 | 1274 | 49 |
| **1000** | 1050 | 2100 | 501773 | 1048 | 249899 | 475 | 248049 | 450 | 500499 | 999 |
| **2000** | 3050 | 6100 | 2501498 | 2998 | 1000399 | 975 | 998549 | 950 | 2000999 | 1999 |
| **5000** | 8050 | 16100 | 14503498 | 6998 | 6251899 | 2475 | 6250049 | 2450 | 12502499 | 4999 |
| **10000** | 18050 | 36100 | 62507498 | 14998 | 25004399 | 4975 | 25002549 | 4950 | 50004999 | 9999 |
|  |  |  |  |  |  |  |  |  |  |  |
| **Random Integer Sorting** | | | | | | | | | | |
|  | **Natural Merge** | | **Quicksort** | | **Quicksort Insert-50** | | **Quicksort Insert-100** | | **Quicksort Median** | |
| **Sorted** | **Comparisons** | **Swaps** | **Comparisons** | **Swaps** | **Comparisons** | **Swaps** | **Comparisons** | **Swaps** | **Comparisons** | **Swaps** |
| **50** | 335 | 370 | 288 | 84 | 681 | 632 | 681 | 632 | 288 | 84 |
| **1000** | 98116 | 124722 | 11914 | 2727 | 15545 | 9945 | 22490 | 17786 | 11626 | 2643 |
| **2000** | 501540 | 631718 | 36848 | 8408 | 34622 | 18602 | 50876 | 36528 | 25222 | 5765 |
| **5000** | 2974762 | 3747000 | 98287 | 22063 | 91190 | 47500 | 125217 | 86405 | 73065 | 16298 |
| **10000** | 12833802 | 16147547 | 229680 | 50468 | 183609 | 98247 | 264117 | 189187 | 156615 | 34170 |

**Measurements**

What I Learned

I have a much better understanding of what the benefits of each sorting algorithm are as well as when they should be implemented. I see that each sort is the most efficient sorting algorithm for a specific situation so it is important to see in what order the data is that will be sorted. I have also learned that implementing multiple sorts together is often the best way to optimize a sort. In the future I will always look for ways to make a sort more efficient such as the implementation of insertion sort on quicksort partitions. This was a great example to see how two sorts can be used together.

I feel I did a much better job of stopping to think about what it is I wanted to do before programming which made this a much smoother process than I had with programming assignments in the post

What I Might do differently Next Time

I need to take it slowly and need to step back to see where the best place to start will be. I ended up starting with a top-down approach and started with the sorts and then worked my way to reading the file. I realized that this was a mistake and then after a day or two went back and developed my read file and then my linked list. While this was not a huge deal it did cost me time. I should have known to read the file in first and start one section at a time instead of jumping right into programming the sorts. This is similar to the mistake I made last time. Luckily, I identified the mistake and worked slowly from reading the file in all the way to a working sort.

Justification for Design Decisions

This program executes five different sorting algorithms. The algorithms used are the natural merge sort and four variations of the quicksort algorithm. The four quicksorts are: quicksort with the first element as the pivot, standard with the median of three pivot, and two hybrid quicksorts that utilize insertion sorts on partitions. The two hybrids utilize insertions sort when partitions are less than 50 and 100.

There are Five classes in this program and they are Main, ReadFile, QuickSort, NaturalMerge, and LinkedList. This program is designed to be compartmentalized for easy adjustment to any required elements. The read file class is designed to take user input files as well as an output folder location. It will then sort using each sorting algorithm and write to an output file. Each input file writes five output files (one file per sorting algorithm).

The program reads each input file into a linked list and four arrays and then sorts the input integers using the five different algorithms. The linked list is sorted using a natural merge sort and the quicksorts are sorted using arrays. The natural merge class has a method to identify in order runs and the merge them and is a recursive sort. The algorithms in quicksort are also recursive and the different implementation methods are contained within the class.

Issues of Efficiency

In order to confirm my findings for O(n) I ensured that both the natural merge and quicksort algorithms were recursive so that the overhead of recursion vs iterative would not be a factor. The below table shows the averages across 5 runs of 10,000 lines of input for random files, reverse ordered files, and ascending ordered files.

We see that merge sort generally needs and performs less comparisons than quick sort and as a result merge sort will be more efficient should there be a lot of comparisons needed. Quicksort uses less memory with only O(logn) additional memory required so quicksort is more efficient in situations with higher memory usage.

The below chart contains an estimate of the big O notation for the different implementations.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Integer Sorting 10,000 Integers Big-O Runtime (Average of 5 Runs)** | | | | | |
|  | **Natural Merge** | **Quicksort** | **Quicksort Insert-50** | **Quicksort Insert-100** | **Quicksort Median** |
| **Best O(n)** | Ascending- O(n) | Random- O(nlogn) | Random- O(n) | Random- O(n) | Random- O(nlogn) |
| **Worst O(n)** | Reverse- O(n^2) | Reverse- O(n^2) | Reverse- O(nlogn) | Reverse- O(nlogn) | Reverse- O(nlogn) |
| **Reverse** | 177.5 | 85 | 16 | 16 | 26 |
| **Ascending** | 1 | 33.4 | 11.6 | 11.4 | 28.2 |
| **Random** | 119.6 | 0.955 | 0.7 | 0.7 | 0.89 |