Sorting Algorithm Report

Insertion Sort – Insertion sort works by first sorting the first two elements then it sorts the next element by looping from the back and moving it toward the front of the list to where it belongs. The best case for this sort is O(n) when the list is sorted, the worst case is O(n^2) when the list is reverse sorted. The average time is O(n^2). In my data insertion sort was very quick with a sorted list but did take much longer with reverse sort, almost exactly matching selection sort in comparisons.

Selection sort – Selection sort works by finding the smallest element in the list and putting it at index 0. Then it finds the next smallest and puts it at index 1, and so on through the entire list until it is sorted. The next smallest element is “selected” and placed where it belongs. The best case, worst case, and average scenario are all O(n^2) because the algorithm always runs the same. This is very consistent. In my data the number of comparisons stayed the exact same through all tests (with 50k elements) as expected with the movements varying on the element’s initial positions. The time for completion was fairly consistent, but there may be variance here due to background app processes on the computer at any given moment.

Quick sort – In this sort you choose a “pivot” which is an element in the list (generally toward the middle) after this you sort the list so that all values less than the pivot are now before it. After that is done a new pivot is chose down the list and the process repeats. The best case/average time for quick sort is O(nlogn) and the worst case is O(n^2). The least comparisons for this was InOrder and the most was AlmostOrder oddly enough, which I suspect to be chance.

Merge Sort – Merge sort takes a divide and conquer approach. It divides the list into many sublists and sorts them, then merges them back together in order using recursion. Merge sort is O(nlogn). In my data merge sort was quicker the more ordered the list was, whether that was ascending or descending. But it was consistent when it was random and 20% ordered.

Heap Sort – Here we use a heap and find a minimum value and place it at the beginning of the list. You find the next minimum and place it where it belongs like selection sort, but this uses binary trees instead. Running time O(nlogn). In my data the Heap sort was very consistent in its results for comparisons and movements but performed worst in the ordered list. Otherwise, it had almost identical performance each time.

Radix Sort – A radix sort is a bucket sort on one digit at a time starting with the least significant digit. This uses less buckets but more phases. Radix sort is O(n). In my data radix sort made N comparisons every time and had almost the same exact amount of movements. The time was extremely quick as well. This sort was very efficient.

Data:

(Spreadsheet included in zip)

Table

Description automatically generated

The difference in the sorting of lists allowed each algorithm to show how they perform in various situations. As I detailed in each description, some performed consistently across the board (selection and radix sort) while other algorithms vastly changed the amount of comparisons/movements they made (insertion sort). This experiment allowed me to go deep into each sort and see what situations are best for them. The spreadsheet and program will be useful when deciding what sorts are best for which situations in the future.