|  |  |  |  |
| --- | --- | --- | --- |
| eXPERIMENTAL RESULTS: ARRAY SIZE 50K | | | |
| List property: in order | **COMPARISONS** | **MOVEMENTS** | **TIME (MS)** |
| insertion Sort | 49999 | 0 | 5 |
| Selection Sort | 1249975000 | 0 | 262 |
| Quick Sort | 1250024998 | 0 | 524 |
| Merge Sort | 382512 | 784464 | 10 |
| Heap sORT | 2687924 | 1318963 | 68 |
| rADIX SORT | 0 | 250000 | 13 |
|  |  |  |  |
| List property: reverse order | **COMPARISONS** | **MOVEMENTS** | **TIME (MS)** |
| insertion Sort | 1250024999 | 1249975000 | 433 |
| Selection Sort | 1249975000 | 25000 | 1505 |
| Quick Sort | 1250049997 | 25000 | 686 |
| Merge Sort | 401952 | 784464 | 11 |
| Heap sORT | 2065043 | 648893 | 94 |
| rADIX SORT | 0 | 250000 | 12 |
|  |  |  |  |
| List property: Partial order | **COMPARISONS** | **MOVEMENTS** | **TIME (MS)** |
| insertion Sort | 238399019 | 238349020 | 111 |
| Selection Sort | 1249975000 | 49983 | 818 |
| Quick Sort | 3183035 | 133016 | 23 |
| Merge Sort | 489707 | 784464 | 11 |
| Heap sORT | 2536897 | 1150088 | 90 |
| rADIX SORT | 0 | 250000 | 13 |
|  |  |  |  |
| List property: Random order | **COMPARISONS** | **MOVEMENTS** | **TIME (MS)** |
| insertion Sort | 626033016 | 625983017 | 264 |
| Selection Sort | 1249975000 | 49987 | 817 |
| Quick Sort | 1170021 | 167537 | 16 |
| Merge Sort | 718268 | 784464 | 13 |
| Heap sORT | 2130224 | 714493 | 65 |
| rADIX SORT | 0 | 250000 | 12 |
|  |  |  |  |
|  |  |  |  |

The chart on page 1 goes over the 24 outcomes using 50k elements. The program allows the user to choose between 5k, 15k or 50k elements.

**Overview of Algorithms**

The insertion algorithm works by sorting each element by comparing it to the element on it’s left. It starts from the left moves its way right through the array. As it continues to sort, the left side becomes sorted, and the right is the unsorted side. The best case for this algorithm is when the data is already sorted, giving us time complexity of O(n). The average and worse case is O(n^2).

Selection sort is O(n^2) for all cases. The selection sort works by looking for smallest number in the data and swaps it to the first element. Then we move on to the next index. Selection sort repeatedly scans for next smallest even if the array is sorted. So, it doesn’t really depend on how the data is given.

Quick sort is is O(nlogn) in best or average case. The best case occurs when pivot is median. It is O(n^2) for worst case which happens when pivot is largest or smallest element. The quick sort works by picking a pivot, which in our case was the first element. It works by portioning the elements greater than the pivot and less than the pivot. It then recursively sorts those two sub lists/partitions.

Merge sort is always O(nlogn). This algorithm works by splitting the list repeatedly until we have each element in their own list. Then, we continue to work our way up by joining adjacent list into bigger lists in a sorted fashion. It is a divide and conquer type of algorithm that works recursively making is O(nlogn) in all cases.

The heap sort is always O(nlogn). A heap sort works by essentially creating a heap from the array given. It makes a max heap and removed the root by swapping it with last node so that the end of our array is sorted and then deleting last node. We repeat this process until the array is fully sorted.

The radix sort doesn’t use comparisons to sort. It basically sorts by each place value and digit and by places them into “buckets”. It does this until the largest digit is also sorted. It is O(nk) for all cases.

**In Order Data**

For the in-order data, the best sorting algorithm is the insertion sort. Insertion sort ran in the shortest amount of time, did no movements, and had the least comparisons among the comparison-based sorts. Since, the array was ordered this is the best case for insertion sort as well.

The worst case for this was the quick sort. It took the most comparisons and had the longest time out of all the sorts. This makes sense because for quick sort when pivot is the smallest value which it was in our case, it gives quick sort the worst time complexity.

Radix sort worked pretty well. It doesn’t do comparisons, but the movements and were nearly as good as the insertion sort. Merge sort did both comparison and movements but it terms of time it was pretty fast. Selection sort took a lot of comparisons and took a lot more time compared to the other sorts. Heap sort did both comparisons and movements and it’s time was in the middle compared to the other sorts.

Best: Insertion Sort Worst: Quick Sort

**Reverse Order Data**

Radix sort was the best for reverse data because it was one of the fastest while having no comparisons. The movements it did was also the least or equal to the others. The worst case for reverse order was insertion. Insertion had the most comparisons and movements. It was also taking a while. For insertion sort, the worst case is happening when the data is reversed which is seen with our tests,

Selection sort and quick sort were also not one of the best options for reverse data. They both took a lot longer than the other sorts and had very high comparison counts. Heap sort did both comparisons and movements, but they were not as many as the others and it falls somewhere in the middle in terms of time, so it was average. The merge sort did good. It didn’t have as many comparisons and movements as the others, and it was the fastest. It was second to radix because it did both comparisons and had more movements than radix.

Best: Radix Sort Worst: Insertion Sort

**Partial Order Data**

For partially sorted data, the best case was still radix sort. It was one of the fastest, did minimum movements and did not rely on comparisons. The worst was insertion sort again because it had the most movements. It also had one of the slowest times and a large number of comparisons.

Selection sort was also a poor choice for this data. It took the longest and had the largest number of comparisons. It was not the worst because the movements were not as large as some of the other sorts. Heap sort was doing an average job, as it’s time, comparison and movements were all in middle compared to the others. Merge sort was a good choice for this data. It was fast, did one of the fewest comparisons but its movements were somewhere in the middle compared to the other sorts.

Best: Radix Sort Worst: Insertion Sort

**Random Order Data**

For randomly ordered data, the radix sort was the best. It was the fastest and had one of the fewest movements. The worst was insertion sort. It had the most movements. It also had the second most comparisons and was the second slowest.

The selection sort also had a lot of comparisons, but the movements were on the lower side. However, it did take the longest, so it was a poor choice for this type of data. Heap sort falls in the middle again because it’s movements, comparisons and time all are average compared to others. The merge sort’s comparisons and movements were not too many or too few, but it was quite fast. The quick sort was a bit slower, but it did fewer movements than the merge sort. It had more comparisons though. This makes the merge and quick sort not the worst choice or the best, but rather average.

Best: Radix Sort Worst: Insertion Sort

**The Approach and Challenges**

The approach I used was to count the comparisons, movements, and time it took for each algorithm to sort the data. I used counters for called comp for comparison and swaps for movements. I typically made them static to ensure I could use them in recursive methods or in classes that used many methods. In the heap class, I defined the variables in the Heapsort class and called them in the heap class. Whenever, there was a movement, I incremented swap and the same for comparison.

I initially tested my sort counter using small arrays with 5 to 7 integers in them so that I could do the sorting myself and compare the programs values with my one. Afterwards, I generated the data and tested my program with the 4 different data to make sure they were running fine. Finally, I added all the user input code and added in the time calls.

Some challenges I faced were that initially my heapSort’s counter for comparisons wasn’t incrementing. I initially defined them in the heap class but then moved them to the HeapSort class. That change fixed my issue. I had many issues with the quicksort. Firstly, I was used to quick sort when the pivot was the middle value so it was a bit of a challenge to think of the pivot as the first element. The second issue I had was with stackoverflow. I did not use an ide to make or edit my code so I wasn’t quite sure where to put the -xss3m statement. I was able to figure it out by typing java xss3m Test into the terminal. However, it wouldn’t run so I had to change the statement to xss4m instead.