**Sorting Algorithms Report**

**In this report, I will be discussing how each of the sorting algorithms worked, the best, average, and worst case time complexity of the each of them, as well as the challenges faced while doing the analysis. At the end, we will decide which algorithm is the most efficient and which one is the worst out of all of them.**

**List of sorting algorithms we are using:**

1. **Insertion Sort**
2. **Selection Sort**
3. **Quick Sort**
4. **Merge Sort**
5. **Heap Sort**
6. **Radix Sort**

**To do the experiments, we first need to know how each of the algorithms work and their time complexity based on how it works.**

1. **Insertion Sort: At the kth step put the kth element in the correct place among the first k elements**

* **Best-case O(N) Worst-case O(N2) “Average” case O(N2)**

1. **Selection sort: At the kth step, find the smallest element among the not-yet-sorted elements and put it at position k**

* **Best-case O(N2) Worst-case O(N2) “Average” case O(N2)**

1. **Quick Sort: Pick a “pivot” element. Divide elements into less-than pivot and greater-than pivot. Sort the two divisions (recursion twice)**

* **Best-case O(N\*log N) Worst-case O(N2) “Average” case O(N\*logN)**

1. **Merge Sort:**
   1. **To sort array from position lo to position hi:**
   2. **If range is 1 element long, it’s sorted! (Base case)**
   3. **Else:**
   4. **Sort from lo to (hi+lo)/2**
   5. **Sort from (hi+lo)/2 to hi**
   6. **Merge the two halves together**

* **Best-case O(N\*log N) Worst-case (N\*log N) “Average” case O(N\*logN)**

1. **Heap Sort: Add all items into a heap. Pop the largest item from the heap and insert it at the end (final position). Repeat for all items.**

* **Best-case O(N\*log N) Worst-case (N\*log N) “Average” case O(N\*logN)**

1. **Radix Sort: Get a series of numbers, and sort them one digit at a time (moving all the 1000’s ahead of the 2000’s, etc.). Repeat the sorting on each set of digits.**

* **Best-case O(N) Worst-case (N) “Average” case O(N)**

**From experiments with each of them, we chose to use an input size of 50000 to thoroughly test the algorithms and the data gathered are:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Experimental Results** | **ArraySize** | **50K Elements** |  |
| **List Property: InOrder** | **Comparisons** | **Movements** | **Run Time (ms)** |
| **Insertion Sort** | **49999** | **0** | **1** |
| **Selection Sort** | **1249975000** | **0** | **612** |
| **Quick Sort** | **99660** | **0** | **3** |
| **Merge Sort** | **382512** | **784464** | **10** |
| **Heap Sort** | **2662925** | **2360963** | **34** |
| **Radix Sort** | **0** | **45** | **6** |
|  | | | |
| **List Property: ReverseOrder** | **Comparisons** | **Movements** | **Run Time (ms)** |
| **Insertion Sort** | **1250024999** | **1249975000** | **976** |
| **Selection Sort** | **1249975000** | **625000000** | **1660** |
| **Quick Sort** | **107766** | **24999** | **4** |
| **Merge Sort** | **401952** | **784464** | **10** |
| **Heap Sort** | **1416046** | **1014963** | **47** |
| **Radix Sort** | **0** | **45** | **6** |
|  | | | |
| **List Property: AlmostOrder** | **Comparisons** | **Movements** | **Run Time (ms)** |
| **Insertion Sort** | **624640925** | **624590926** | **500** |
| **Selection Sort** | **1249975000** | **455779** | **1513** |
| **Quick Sort** | **265313** | **127333** | **13** |
| **Merge Sort** | **718239** | **784464** | **13** |
| **Heap Sort** | **1545297** | **1153931** | **39** |
| **Radix Sort** | **0** | **45** | **6** |
|  | | | |
| **List Property: RandomOrder** | **Comparisons** | **Movements** | **Run Time (ms)** |
| **Insertion Sort** | **625836784** | **625786785** | **510** |
| **Selection Sort** | **1249975000** | **459873** | **1499** |
| **Quick Sort** | **257920** | **115700** | **17** |
| **Merge Sort** | **718049** | **784464** | **13** |
| **Heap Sort** | **1543647** | **1152237** | **42** |
| **Radix Sort** | **0** | **45** | **6** |

**Based on the experimental results, we can conclude the best and worst algorithms for each of the data types:**

* **InOrder:**
  + **Best: Insertion Sort**
  + **Worst: Selection Sort**
* **ReverseOrder:**
  + **Best: Quick Sort**
  + **Worst: Selection Sort**
* **AlmostOrder:**
  + **Best: Radix Sort**
  + **Worst: Selection Sort**
* **RandomOrder:**
  + **Best: Radix Sort**
  + **Worst: Selection Sort**

**Overall, the most efficient algorithm is radix sort as the number of the comparisons, movements, and run time is the least out of all the algorithms.**

**During the project, challenges that were encountered were that fixing the errors that the Quick Sort algorithm had, which is not showing any results at all. Another challenge was that figuring out which part of the algorithm to decide as a comparison and movements of the elements.**