Ascertaining the fastest algorithm for sorting in various conditions using experiments

Iliuță-Laurențiu Andoni
West University,
Timișoara, România,
Email: iliuta.andoni01@e-uvt.ro

April 2021

Abstract

We analyze the behaviour of multiple sorting algorithms such as Radix Sort, Quick Sort, Merge Sort, Bubble Sort and Insertion Sort in various experimental scenarios to see which will have the best average performance. We present several experiments on many self-generated data sets with different dimensions from small ones to extra-large ones to affirm the overall quickest but to also state the distinct scenarios which can be disastrous to the algorithm performance.

1 Introduction

We address the problem of the speed of various sorting algorithms for different experimental scenarios to affirm which are not suited for particular situations and which will have the quickest running time on the average (*e.g. considering all data sets result and the average performance*).
The outcome of these experiments is especially valuable because sorting plays an important purpose in our quotidian experiences and the impact of the experiments are relevant to the domain of sorting as we can determine the best performer algorithm.
A simplistic example to motivate the purpose of the paper is the daily usability of the sorting methods from a daily task like establish the best schedule to using sorting algorithms as an intermediate step for more elaborated algorithms such as Binary Search which is a searching method that can only be implemented on sorted inputs.
To accomplish the goal set by the paper we generated different data sets using python programming language then we attempt to sort them using the implementation of the mentioned sorting algorithm using the C programming language. To accomplish precise data we ran the experiments three times and tracked the data in tables to compare them afterwise intending to confirm that a better complexity algorithm will act appropriately.
In the next sections, we will take a look at the theoretical aspects of the problem as we will proceed to compare the theory with the experimental data we obtained, we will introduce the experimental data sets, algorithms that were used in the experiments, we will take a close look at the results of the experiments and compare it with related work made by other authors and eventually we will state the conclusion and present the future work for our problem.

2 Formal Presentation of the Theoretical Aspects of the Problem

A sorting algorithm intends to change the order of elements in a given list such that after a finite number of steps, the list will be whether in increasing or decreasing order.

The algorithm will take as the input data a sequence of n elements and will give as an output a permutation of the input sequence such that $a_1^{'} \leq a_2^{'} \leq ... \leq a_n^{'}$
[3] [4] . To have a better understanding of the efficiency of an algorithm we will introduce now the big-O notation. We will use this to describe functions that are **asymptotically bounded** above. For a given function $g(n)$, we denote by $O(g(n))$ the of functions [3]

$$O(g(n)) = \{f(n) : \textit{there exist positive constant } c \textit{ and } n_0 \textit{ such that} \\ 0 \leq f(n) \leq cg(n) \textit{ for all } n \geq n_0\}$$

Another way to distinguish the nature of the sorting algorithm is whether they are comparison sort based algorithm or not.[3][6] [5] On the first list, we can mention Quick Sort, Merge Sort, Bubble Sort and Insertion Sort as they need to compare the elements of the list to perform sorting. On the non-comparison sort based algorithm, we will mention Radix Sort as it uses the Counting Sort algorithm which takes advantages of the number of appearances of an element rather than the value of the element.
We can add a new branch to our analysis and we can categorise the algorithms by the property of stability [3], that is, the value of the elements appear in the same order in the output as in the input, this being a key concept for Radix Sort. The other algorithms mentioned are not stable.
Proof of Stability of Counting Sort: [3] [4]Rather than trying to determine the position of the element i by comparing all the elements before it, this algorithm will count the number of elements less than the element i and decide the place of the element i based on how many numbers are less than i.
In the code for counting sort, we assume that the input is an array $A[1...n]$. Then we need two other arrays: the array $B[1...n]$ holds the sorted output, and the array $C[0...k]$ provides temporary working storage.
Suppose positions i and j with $i < j$ both contain some element k. Since $j > i$, the loop will examine $A[j]$ before examining $A[i]$. When it does so, the algorithm correctly places $A[j]$ in position $m = C[k]$ of B. Since $C[k]$ is getting decremented, and is never again incremented, we are guaranteed that when the for loop examines $A[i]$ we will have $C[k] < m$. Therefore $A[i]$ will be placed in an earlier position of the output array, proving stability.

3 Implementation of Sorting Algorithms

We will now take a look at the sorting algorithm by the two categories presented earlier:

1. Comparison Based ,
2. Not Comparison Based.

3.1 Comparison Based

Considering the category presented earlier, we can group comparison based algorithms together.

3.1.1 Insertion Sort

[3] [4]The way insertion sort work is to iterate over every element of the list and put it in its right place by comparing it to all the items before the actual element. This is a simple and natural approach and yet very wasteful.

```
1  void insertionSort(int arr[], int n){
2      int i, key, j;
3      for (i = 1; i < n; i++){
4          key = arr[i];
5          j = i - 1;
6          while (j >= 0 && arr[j] > key) {
7              arr[j + 1] = arr[j];
8              j = j - 1;
9          }
10         arr[j + 1] = key;
11     }
12 }
```

As we are analyzing the complexity using the O notation, we will analyze the algorithm in the worst-case scenario. The worse case for *Insertion Sort* will be accomplished when the list is in reverse order. It is important to note that Insertion Sort performs best when the list is already sorted.
For each iteration of i there will be made i comparison so the total number of comparisons is:

$\sum_{i=1}^{n-1} i = 1 + 2 + 3 + ... + (n-1) = \frac{n(n-1)}{2} = O(n^2)$

3.1.2 Bubble Sort

[3] [4]This method is a naive approach to sorting done by comparisons. The algorithm switches two adjacent elements if they do not respect the order. This is done in a loop until the input is completely sorted (*i.e. no more switches can be made*)

```
1  void swap(int *xp, int *yp)
2  {
3      int temp = *xp;
4      *xp = *yp;
5      *yp = temp;
6  }
7  void bubbleSort(int arr[], int n)
8  {
9      int i, j;
10     for (i = 0; i < n-1; i++)
11         for (j = 0; j < n-i-1; j++)
12             if (arr[j] > arr[j+1])
13                 swap(&arr[j], &arr[j+1]);
14 }
```

This code will use a function called *swap* to swap two element. The complexity of this program will be given by the number of steps made by the for loop.
$\sum_{i=1}^{n-2}(n-1-i) = \frac{n(n-1)}{2} = O(n^2)$

3.1.3 Merge Sort

[3] [4]The strategy is a *divide-and-conquer* one. We will *divide* our list into a smaller list and keep doing that until we're left with some small lists. After that, we start creating a new list by joining the small lists in a sorted matter. Yet, this algorithm is still based on comparisons.

```
1  void merge(int arr[], int l, int m, int r)
2  {
3      int i, j, k;
4      int n1 = m - l + 1;
5      int n2 = r - m;
6      static int L[9999999], R[9999999];
7      for (i = 0; i < n1; i++)
8          L[i] = arr[l + i];
9      for (j = 0; j < n2; j++)
10         R[j] = arr[m + 1 + j];
11     i = 0;
12     j = 0;
13     k = l;
14     while (i < n1 && j < n2) {
15         if (L[i] <= R[j]) {
16             arr[k] = L[i];
17             i++;
18         }
19         else {
20             arr[k] = R[j];
21             j++;
22         }
23         k++;
24     }
25     while (i < n1) {
26         arr[k] = L[i];
27         i++;
28         k++;
29     }
30     while (j < n2) {
31         arr[k] = R[j];
32         j++;
33         k++;
34     }
35 }
36 void mergeSort(int arr[], int l, int r)
37 {
38     if (l < r) {
39         int m = l + (r - l) / 2;
40         mergeSort(arr, l, m);
41         mergeSort(arr, m + 1, r);
42         merge(arr, l, m, r);
43     }
44 }
```

Because in our case we are talking about subarrays that will result in a complexity of $O(log\ n)$. Adding the merging operation that will be performed in $O(n)$ since we have to traverse the whole list. So therefore the running time would become $O(nlog\ n)$

3.1.4 Quick Sort

[3] [4]This algorithm belongs to the divide-and-conquer category as well. The list is portioned by a chosen element, which we will call "pivot", again and again until the lists become trivially small. When portioning the big list, the small ones either contains all the elements bigger or equal to the pivot or the elements smaller than the pivot. However, this is still a comparison based algorithm.

```
1  int partition(int arr[], int l, int h)
2  {
3      int x = arr[(l+h)/2];
4      int i = (l - 1);
5
6      for (int j = l; j <= h - 1; j++) {
7          if (arr[j] <= x) {
8              i++;
9              swap(&arr[i], &arr[j]);
10         }
11     }
12     swap(&arr[i + 1], &arr[h]);
13     return (i + 1);
14 }
15 void quicksort(int arr[], int l, int h)
16 {
17     static int stack[9999999];
18     int top = -1;
19     stack[++top] = l;
20     stack[++top] = h;
21     while (top >= 0) {
22         h = stack[top--];
23         l = stack[top--];
24         int p = partition(arr, l, h);
25         if (p - 1 > l) {
26             stack[++top] = l;
27             stack[++top] = p - 1;
28         }
29         if (p + 1 < h) {
30             stack[++top] = p + 1;
31             stack[++top] = h;
32         }
33     }
34 }
```

The swap function used in this code is the same as the one introduced earlier for the Bubble Sort. Although the running time of the Quick Sort is $O(n^2)$, the recursive approach would go so deep into recursions that the stack memory wouldn't suffice for some bigger inputs. That was the main issue until the iterative approach was implemented.

3.2 Non-Comparison Based

[3] [4]Here we will analyze both the intermediate algorithm, the *Counting sort*, and the main algorithm *Radix Sort*.

3.2.1 Counting Sort

The counting sort method computes the number of occurrences of a number between 0 and k and stores that value in an auxiliary list which will help to construct the new list. This algorithm is not based on comparison and it is a stable one. However, the memory usage is greater than memory usage by the other regular comparison based algorithms. We will not use this as an independent sorting method but rather as an intermediate algorithm for the Radix Sort.

```
1  void countSort(long arr[], long n, long exp)
2  {
3      static int output[9999999];
4      int i, count[10] = { 0 };
5      for (i = 0; i < n; i++)
6          count[(arr[i] / exp) % 10]++;
7      for (i = 1; i < 10; i++)
8          count[i] += count[i - 1];
9      for (i = n - 1; i >= 0; i--) {
10         output[count[(arr[i] / exp) % 10] - 1] = arr[i];
11         count[(arr[i] / exp) % 10]--;
12     }
13     for (i = 0; i < n; i++)
14         arr[i] = output[i];
15 }
```

3.2.2 Radix Sort

[3] [4]Taking advantage of the counting sort, Radix sort brings a more unique strategy to sorting as the concept used is to sort by the rightmost digit of a number. Taking use of the counting sort this still keeps up the bigger memory usage and the stability property. However, this sorting method does not work for negative numbers nor strings.

```
1  void radixsort(long arr[], long n)
2  {
3      long m = getMax(arr, n);
4      for (long exp = 1; m / exp > 0; exp *= 10)
5          countSort(arr, n, exp);
6  }
```

The *getMax* function would simply return the biggest value in a given array.
Even if the complexity of this algorithm is $O(n)$, there are some cases (*i.e. when the values are negative*) when the algorithm would fail to deliver any result.

4 Case Study

To decide the best performance algorithm, we would see how the running time would scale through variation of lengths starting with lists of 100 000 elements, denoted as *"small"* lists, then 500 000 elements denoted as *"medium"* lists and eventually 1 000 000 elements denoted as *"large"* lists.
Then we would analyse the behaviour in various situations: sorted lists (i.e. ascending), reverse sorted (i.e. descending), random positive integers, random negative integers only, random both positive and negative integers, almost sorted (the whole list is sorted except for five random elements), random similar elements (i.e. the range of the values is from 1 to 10) and eventually, random floating-point numbers.
The input data was generated using Python code, mainly the **random** library. The code similar, the only difference being the range values. The code and the data sets used for the experimental activity can be accessed `here` or at `https://github.com/laurentiuandoni/inputs-for-sorting-algorithms-MPI-2021`.

Listing 1: Sorted List Generation

```
with open("small_sorted_list","w") as f:
    for i in range(0,1000):
        f.write(str(i))
        f.write(" ")
```

Listing 2: Reverse Sorted List Generation

```
with open("small_reversesorted_list","w") as f:
    for i in range(1000,0,-1):
        f.write(str(i))
        f.write(" ")
```

Listing 3: Random Positive Integers List Generation

```
import random
with open("small_positive_list","w") as f:
    for i in range(1000):
        f.write(str(random.randrange(0,1000)))
        f.write(" ")
```

Listing 4: Random Negative Integers List Generation

```
with open("small_negative_list","w") as f:
    for i in range(1000):
        f.write(str(random.randrange(-1000,-1)))
        f.write(" ")
```

Listing 5: Random both Positive and Negative Integers

```
with open("small_integers_list","w") as f:
    for i in range(1000):
        f.write(str(random.randrange(-1000,1000)))
        f.write(" ")
```

Listing 6: Almost Sorted

```
array=[]
with open("small_almostsorted_list","w") as f:
    for i in range(1000):
        array.append(i)
    array[random.randrange(1,1000)] =
    array[random.randrange(1,1000)]
    array[random.randrange(1,1000)] =
    array[random.randrange(1,1000)]
    array[random.randrange(1,1000)] =
    array[random.randrange(1,1000)]
    array[random.randrange(1,1000)] =
    array[random.randrange(1,1000)]
    array[random.randrange(1,1000)] =
    array[random.randrange(1,1000)]
    for i in range(1000):
        f.write(str(array[i]))
        f.write(" ")
```

Listing 7: Similar Elements

```
with open("small_similar_list","w") as f:
    for i in range(1000):
        f.write(str(random.randint(1,10))
        f.write(" ")
```

Listing 8: Floating-Point Numbers

```
with open("small_floating_list","w") as f:
    for i in range(1000):
        f.write(str(random.randint(1,1000))+
        "."
        +str(random.randint(1,1000)))
        f.write(" ")
```

We will now take a look at the three runs of the sorting algorithms grouped together in a table for a better visualization.

	Radix Sort	Quick Sort	Merge Sort	Bubble Sort	Insertion Sort
Small Sorted List	0.01	0.047	0.047	11.909	0.047
Small Reverse Sorted List	0.021	0.032	0.062	23.989	10.676
Small Positive List	0.037	0.039	0.046	30.986	5.364
Small Negative List	INVALID	0.047	0.047	31.618	5.357
Small Integer List	INVALID	0.047	0.047	31.243	5.406
Small Almost Sorted List	0.041	0.034	0.047	11.633	0.047
Small Similar List	0.059	2.265	0.047	30.38	5.256
Small Floating Point Numbers List	0.054	0	0.093	30.994	5.359
Medium Sorted List	0.128	0.379	0.531	1158.451	0.4539
Medium Reverse Sorted List	0.178	0.176	0.282	584.043	265.4329
Medium Positive List	0.257	0.237	0.309	777.456	131.9269
Medium Negative List	INVALID	0.243	0.315	775.024	132.5649
Medium Integer List	INVALID	0.235	0.317	77.552	132.6869
Medium Almost Sorted List	0.309	0.192	0.266	289.231	0.2349
Medium Similar List	0.357	56.165	0.187	759.826	119.0599
Medium Floating Point Numbers List	0.377	0	0.457	776.671	132.5579
Large Sorted List	0.132	0.465	0.652	1379.554	0.586
Large Reverse Sorted List	0.301	0.434	0.654	2630.884	1444.771
Large Positive List	0.437	0.554	0.779	3446.466	709.488
Large Negative List	INVALID	0.554	0.744	3455.477	712.805
Large Integer List	INVALID	0.563	0.762	3408.985	672.801
Large Almost Sorted List	0.636	0.404	0.637	1293.425	0.515
Large Similar List	0.789	243.986	0.446	3240.645	605.633
Large Floating Point Numbers List	0.709	0	1.054	3313.625	613.641

Table 1: First Run

	Radix Sort	Quick Sort	Merge Sort	Bubble Sort	Insertion Sort
Small Sorted List	0.01	0.033	0.049	11.76	0.046
Small Reverse Sorted List	0.022	0.039	0.064	24.863	10.773
Small Positive List	0.039	0.042	0.059	31.435	5.435
Small Negative List	INVALID	0.043	0.057	31.591	6.161
Small Integer List	INVALID	0.056	0.079	33.393	6.01
Small Almost Sorted List	0.045	0.034	0.049	13.025	0.071
Small Similar List	0.055	2.391	0.039	31.824	5.68
Small Floating Point Numbers List	0.057	0.001	0.1	31.67	6.331
Medium Sorted List	0.127	0.375	0.547	1157.877	0.457
Medium Reverse Sorted List	0.19	0.172	0.266	584.707	264.791
Medium Positive List	0.25	0.219	0.313	777.201	132.37
Medium Negative List	INVALID	0.234	0.313	776.641	132.469
Medium Integer List	INVALID	0.235	0.312	832.103	198.397
Medium Almost Sorted List	0.324	0.237	0.354	323.663	0.263
Medium Similar List	0.381	58.138	0.202	788.835	125.362
Medium Floating Point Numbers List	0.372	0	0.486	811.365	139.654
Large Sorted List	0.146	0.377	0.551	1646.082	0.471
Large Reverse Sorted List	0.295	0.386	0.555	2543.938	1118.025
Large Positive List	0.477	0.473	0.642	3165.502	656.237
Large Negative List	INVALID	0.498	0.655	3278.765	670.988
Large Integer List	INVALID	0.517	0.696	3286.828	677.006
Large Almost Sorted List	0.595	0.408	0.627	1302.143	0.539
Large Similar List	0.74	242.311	0.439	3226.96	608.932
Large Floating Point Numbers List	0.728	0	1.046	3304.513	675.779

Table 2: Second Run

	Radix Sort	Quick Sort	Merge Sort	Bubble Sort	Insertion Sort
Small Sorted List	0.01	0.034	0.05	12.64	0.041
Small Reverse Sorted List	0.025	0.047	0.062	32.921	6.472
Small Positive List	0.033	0.043	0.058	32.381	5.381
Small Negative List	INVALID	0.043	0.057	31.982	5.66
Small Integer List	INVALID	0.031	0.047	31.982	5.66
Small Almost Sorted List	0.044	0.031	0.047	13.176	0.041
Small Similar List	0.057	2.327	0.039	31.96	4.889
Small Floating Point Numbers List	0.052	0	0.08	31.54	5.468
Medium Sorted List	0.127	0.385	0.548	1442.024	0.596
Medium Reverse Sorted List	0.187	0.223	0.345	725.421	366.757
Medium Positive List	0.251	0.311	0.386	961.248	176.47
Medium Negative List	INVALID	0.276	0.431	969.173	176.48
Medium Integer List	INVALID	0.303	0.383	961.783	175.226
Medium Almost Sorted List	0.333	0.237	0.338	366.876	175.226
Medium Similar List	0.396	70.797	0.338	366.876	0.296
Medium Floating Point Numbers List	0.397	0	0.251	931.61	156.108
Large Sorted List	0.134	0.405	0.633	1295.806	0.516
Large Reverse Sorted List	0.296	0.405	0.618	2465.839	1395.423
Large Positive List	0.445	0.511	0.693	3348.885	696.118
Large Negative List	INVALID	0.529	0.716	3338.576	690.457
Large Integer List	INVALID	0.531	0.714	3351.679	693.685
Large Almost Sorted List	0.634	0.411	0.612	1315.794	0.519
Large Similar List	0.737	247.037	0.453	3190.94	486.615
Large Floating Point Numbers List	0.751	0	0.921	3096.137	528.495

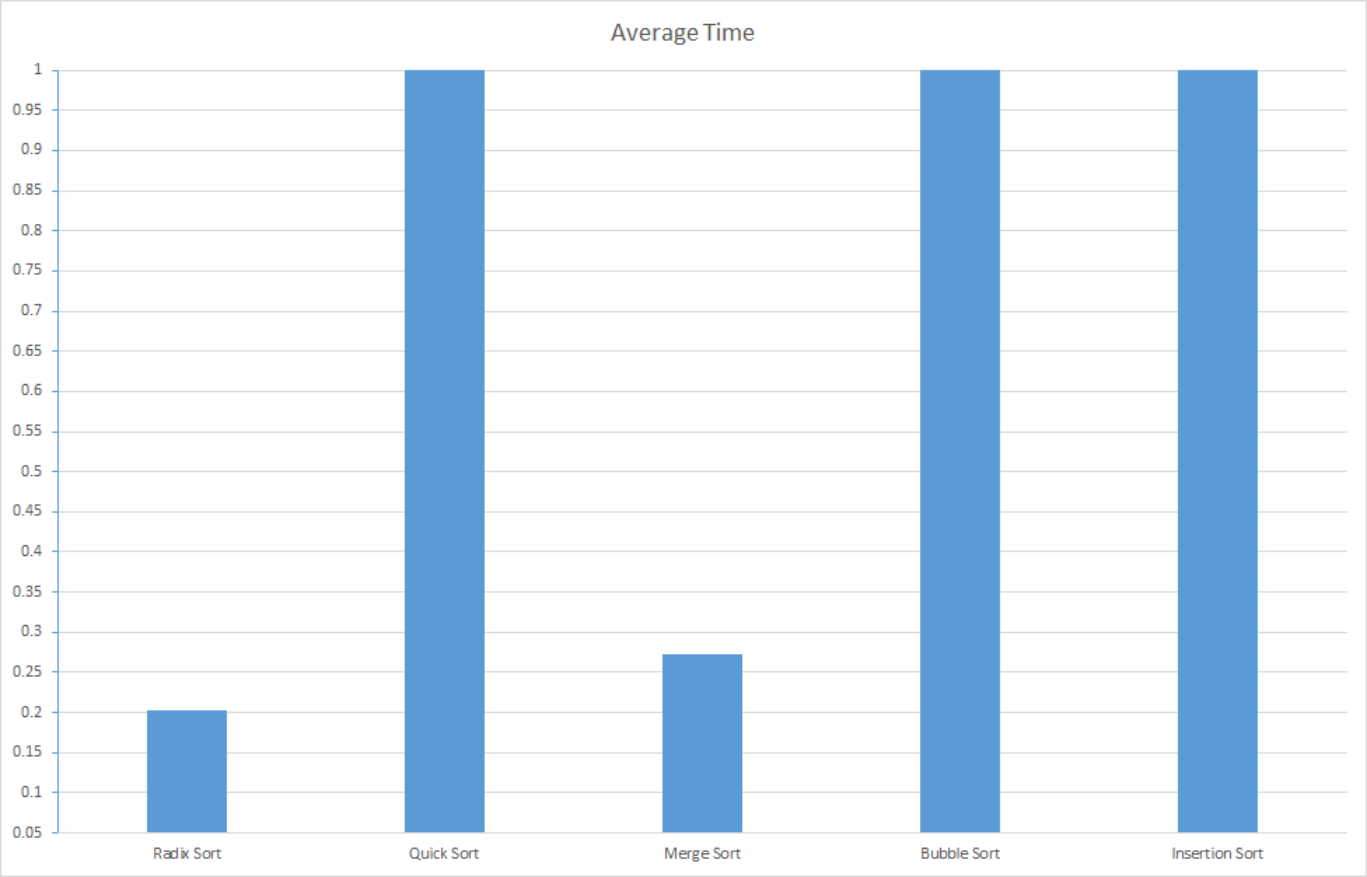
Table 3: Third Run

Now, to have a better overview over the situation we will take a look at the mean of all those values and the we will compute the total average for each sorting method:

	Radix Sort	Quick Sort	Merge Sort	Bubble Sort	Insertion Sort
Small Sorted List	0.01	0.038	0.048	12.103	0.044
Small Reverse Sorted List	0.022	0.039	0.062	27.257	9.307
Small Positive List	0.036	0.041	0.054	31.600	5.393
Small Negative List	INVALID	0.044	0.053	31.730	5.726
Small Integer List	INVALID	0.044	0.057	32.206	5.692
Small Almost Sorted List	0.043	0.033	0.047	12.611	0.053
Small Similar List	0.057	2.327	0.041	31.388	5.275
Small Floating Point Numbers List	0.054	0	0.091	31.401	5.719
Medium Sorted List	0.127	0.379	0.542	1252.784	0.502
Medium Reverse Sorted List	0.185	0.190	0.297	631.390	298.993
Medium Positive List	0.252	0.255	0.336	838.635	146.922
Medium Negative List	INVALID	0.251	0.353	840.279	147.171
Medium Integer List	INVALID	0.257	0.337	623.812	168.769
Medium Almost Sorted List	0.322	0.222	0.319	326.59	58.574
Medium Similar List	0.378	38.18	0.242	638.512	81.572
Medium Floating Point Numbers List	0.382	0	0.398	839.882	142.773
Large Sorted List	0.137	0.415	0.612	1440.480	0.524
Large Reverse Sorted List	0.297	0.408	0.609	2546.887	1319.406
Large Positive List	0.453	0.512	0.704	3320.284	687.281
Large Negative List	INVALID	0.527	0.705	3357.606	691.416
Large Integer List	INVALID	0.537	0.724	3349.164	681.164
Large Almost Sorted List	0.621	0.407	0.625	1303.787	0.524
Large Similar List	0.755	244.444	0.446	3219.515	567.06
Large Floating Point Numbers List	0.729	0	1.007	3238.091	605.971
	Total	**Total**	**Total**	**Total**	**Total**
	0.202	9.048	0.272	874.313	176.120

Table 4: Average of the Three Runs

We will take a step even further in the visualization process and we will develop a chart showing the total average performance of the described sorting algorithms. Note that the graph is scaled because some sorting



method performed such numbers that could not be easily observed in a real-time model.
We expected Quick Sort to be fast, there are some cases where it will not perform as good as the other $O(nlog\ n)$ sorting method.

5 Related Work

In this section, we will point out a similar approach to the problem of sorting, approaches that have already presented in the past but that are relevant to the domain of sorting algorithms and analyzing such algorithms. [1]In their article, D.R. Aremu, O.O. Adesina, O.E. Makinde, O. Ajibola and O.O. Agbo-Ajala have presented the performance of three algorithms with a computational complexity of $O(nlog\ n)$ and $O(n^2)$ which also include the analysis of Quick Sort and even though the numbers aren't related the behaviour described is similar, concluding once more the overall great performance of Quick Sort. [2]The work of Htwe Htwe Aung is related to this experimental analysis of sorting algorithms. The description of the behaviour of Bubble Sort and Insertions Sort is pointing to the same results obtained meaning that

even if both algorithms have a computational complexity of $O(n^2)$ one will have an overall performance greater than the other.
[6]The complex work presented by Ramesh Chand Pandey also point out that the results presented are accurate and the experiment can be declared as an absolute success as the descriptions of the results are similar.

6 Conclusion

The result of the experiment is conclusive as the behaviour of the algorithms mostly respected the theory that was presented in the first part. Unexpected results also show up as we can easily observe that even if Bubble Sort and Insertion Sort are said to be the same complexity (i.e. $O(n^2)$) they do not perform the same, sometimes Insertion Sort performing almost as good as Quick Sort or better than Merge Sort, taking into consideration restrictive cases (i.e. sorted list and almost sorted list). Bad performance was registered in the case of the Quick Sort when similar list scenario has been tested. The overall performance was drastically affected making it not reliable anymore compared to the Radix Sort or Merge Sort.
The performance of Radix Sort is as expected but we shall not ignore the exclusive cases where it couldn't perform, mainly when negative numbers are also involved. Of course, the algorithm can be modified such that it will work for all the inputs and here are some indications especially for that: the main list of numbers can be divided into two, one to contain negative numbers and another to contain just the positive ones. Sorting the list of negative as if they are positive would not affect the algorithm. Merging the two list after sorting, as a matter of complexity, would still be linear thus the overall complexity can be ranked as $O(n)$. As a matter of the space complexity in this case it could be even better than the original space complexity as the same auxiliary array used in the Counting Sort method can now be used gradually, in the first place for the positive and then for the negative if the range of the numbers is similar.[3][4][1][6]

References

[1] D.R Aremu et al. “A Comparative Study of Sorting Algorithms”. In: *African Journal of Computing and ICT* 6.5 (2013).

[2] Htwe Htwe Aung. “Analysis and Comparative of Sorting Algorithms”. In: *IJTSRD* 3.5 (2019).

[3] Thomas H. Cormen et al. *”Introduction to Algorithms, Third Edition”*. MIT Press, 1989.

[4] Adam Drozdek. *Data structures and algorithms in C++, Fourth Edition*. Cengage Learning, 1995.

[5] Ashok Kumar Karunanithi. *A Survey, Discussion and Comparison of Sorting Algorithms*.

[6] Ramesh Chand Pandey. *Study and Comparison of Various Sorting Algorithms*.