A practical and theoretical analysis in the comparison of sorting algorithms

Șova Dumitru Ștefan Andrei
Department of Computer Science,
West Univesity,
Timișoara, Romania,
Email: dumitru.sova01@e-uvt.ro

June 2021

Abstract

One of the main points of interest regarding sorting algorithms stems from their respective time complexity. Be it old or new each algorithm has a specific complexity that can create a reference that can be used in a comparison with other such algorithms. Theoretically, the description of the complexity and its analysis, alongside experimental tests conducted at different input values can provide a starting point in contrasting sorting algorithms. In this paper, we compare several sorting algorithms, analyzing them both theoretically and experimentally in regard to their complexity. The algorithms used are Bubble Sort, Insertion Sort, Merge Sort, Quicksort, Heapsort, Radix Sort, and Counting Sort, which were implemented in the C programming language using Codeblocks.

1 Introduction

Sorting algorithms represent some of the building blocks of Computer Science, acquiring a keen interest in their analysis through large amounts of research. As such, these algorithms are well-established and documented design and performance-wise. With this, the purpose of the paper is to give a better understanding of the time complexity of several well-known sorting algorithms by comparing them with one another.

The data used for the experiments was generated in C, creating lists of randomly sorted integer numbers ranging from 10 000 to 1 million elements. The complexity of each algorithm was analyzed both theoretically, describing their behavior and properties, and practically, experiments being conducted in regards to the run times of the algorithms at said data inputs.

1.1 Motivation

The problem of comparing sorting algorithms arises from the multitude of already documented algorithms and the desire to find the most efficient one. We can become, as such, overwhelmed by the amount of currently studied sorting algorithms. For example, when sorting sizeable amounts of data inputs, the difficulty of choosing a preferred algorithm can become an inconvenience, some algorithms(Quicksort, Radix Sort) outperforming others(Bubble sort, Selection sort) by a wide margin when it comes to time efficiency. Such differences can be seen in Figure 1.1. As such, we become compelled to think that there exists a definitive best sorting algorithm that is yet to be encountered by us, but the fact of the matter is that a convenient enough algorithm is not so easily identifiable. Because of this, different sorting algorithms might be more desirable over others depending on the data type, how the data is represented, and the input distribution. Another example would be when trying to sort nearly sorted data, Insertion Sort proving to be a good choice for this type of scenario, whilst Quicksort, Merge Sort, and Heap Sort adapting at a slower rate to this type of distribution. With these facts in mind, the problem of comparing sorting algorithms might not have a definitive answer, but it might offer a more detailed look when choosing one.

Roughly the steps for the comparison of the sorting algorithms proceed as follows:

- The plan for the comparison is to analyze their respective complexity and to compare at different data inputs ranging from 10 000 to 1 million elements the time each algorithm takes to sort a list of elements.

- The first step is to implement the sorting algorithms that we will end up comparing, those being Bubble Sort, Insertion Sort, Merge Sort, Quicksort, Heapsort, Radix Sort, and Counting Sort. To do this, we will use the C language and Codeblocks to write the code.
- The data inputs will be generated in C, creating lists between 10 000 to 1 million elements that will have random integer values.
- With the data inputs created, each algorithm will run said inputs and offer results that are going to be used in the comparison.
- The complexity and other properties of each algorithm will be analyzed and will become a base of comparison between the algorithms in regards time.
- With both the sorting time and complexity analysis done, we can begin the proper comparison and produce results.
- The expected outcome is that some algorithms will fare better than others, namely Quicksort, Merge Sort, Radix Sort, and Heapsort, especially when looking at large data inputs that exceed 500 000 elements.

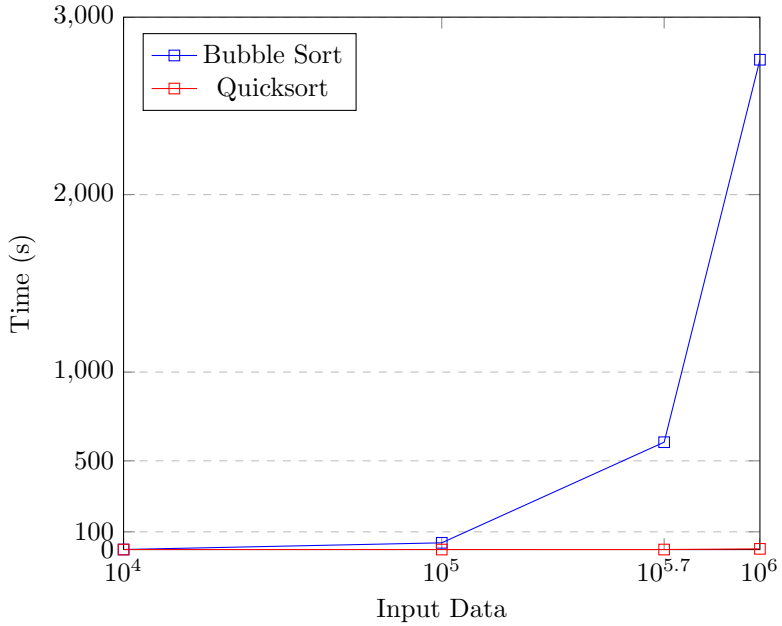

Figure 1.1: Comparison of Bubble Sort and Quicksort at data inputs between 10 thousand and 1 million elements

The rest of the paper is organized as follows: in Section 2, we present a formal description of the problem and its solution. In Section 3 the reader can find the implementations of the algorithms, along side details about their complexity. At Section 4 can be found the results of the experiments and the differences between the algorithms. Next, at Section 5 the reader can find related work, offering a look at different works connected to the papers topic. Finally at Section 6 is the conclusions, wrapping up everything discussed in the previous sections.

2 Theoretical Analysis of Sorting Algorithms

The problem of comparing sorting algorithms requires both a time-based comparison between results given at different data inputs and an analysis of the properties of the algorithms. The analysis of an algorithm can help to estimate the resources needed for the algorithm to solve a given problem, in our case, sorting a list of elements and the time results can provide a practical way to differentiate the algorithms. Because of this, for the purpose of finding a suitable algorithm for a specific problem we have to analyze several sorting algorithms. By doing so, we can discern more than one competent algorithm. A way to discern a good algorithm for a given problem is to implement a bunch of algorithms and find out the respective efficiency for each one, especially when looking at the running time of a program. After finding the algorithm that outperforms the other algorithms, we can get the best algorithm for a given problem.

A way to ensure the execution time of an algorithm is the anticipation of the best case, average case and worst case performance of the algorithm (see [1][2]). These help us to analyze the complexity of an algorithm. The worst-case analysis assumes the largest possible running time that an algorithm needs to solve a problem of size n. In contrast, the best-case analysis assumes the least amount of running time that an algorithm needs to solve a problem of size n and gives a lower bound on the computational complexity. The average case analysis anticipates the average amount of running time that an algorithm needed to solve a problem for any input of size n.

To describe the upper bound we use the Big-O notation, which states the maximum amount of resources needed by an algorithm. The Big-O notation is defined according to the definition at [1] as such:

$$O(g(n)) = \{f(n) : \text{"}\textit{there exist two positive constants c and } n_0 \textit{ such that}$$
$$0 \leq \text{f(n)} \leq \text{cg(n)} \quad \text{for all n} \geq n_0 \text{ "}\}.$$

To describe the tight lower bound we use the Big-Ω notation, which states the minimum amount of resources needed by an algorithm. The Big-Ω notation is defined according to the definition at [1] as such:

$$\Omega(g(n)) = \{f(n) : \text{"}\textit{there exist two positive constants c and } n_0 \textit{ such that}$$
$$0 \leq \text{cg(n)} \leq \text{f(n)} \quad \text{for all n} \geq n_0 \text{ "}\}.$$

Because sorting algorithms have both Big-O and Big-Ω notations we get the Big-θ notation which is used to describe a function that has both a tight upper bound and a tight lower bound. The three notations(O,Ω,θ) are all asymptotic. The Big-θ notation is defined according to the definition at [1] as such:

$$\theta(g(n)) = \{f(n) : \text{"}\textit{there exist positive constants } c_1, c_2 \textit{ and } n_0 \textit{ such that } 0 \leq c_1 \text{g(n)} \leq \text{f(n)} \leq c_2 \text{g(n) for all n} \geq n_0 \text{ "}\}.$$

Another way to analyze sorting algorithms is by classifying them with various parameters. These parameters include the computational complexity, stability and whether the sorting is comparison or non-comparison based(see [1][2][3]). Computational Complexity can predict the performance of the algorithm, taking into consideration the worst-case scenario at large data inputs and the best-case scenario at optimal conditions. Usually a good complexity for a sorting algorithm would be O(nlogn), whilst a bad complexity would be O(n^2). Stability for sorting algorithms can help in preserving the order of elements with equal values. For example, if a sorting algorithm is stable then whenever there are two elements x[0] and x[1] that have the same value and with x[0] is before x[1] in the initial list, x[0] will be shown before x[1] in the sorted list. Lastly, sorting algorithms can be classified as either being comparison based or non-comparison based. Comparison based algorithms, Quicksort for example, need to compare to perform the sorting, while non-comparison based algorithms, Counting Sort for example, focus on the number of appearances of an element rather than its value, not requiring the comparison of the elements.

With these resources in mind, we have several ways to examine the complexity of an algorithm. As an example(see [1][2][8]), we will look at the pseudocode of Insertion Sort and analyze its time complexity.

```
for i = 2 to length(A) do
        j = i
        while j > 1 and A[j - 1] > A[j] do
                swap A[j] and A[j - 1]
                j = j - 1
        end while
end for
```

We begin our analysis by looking at the outer loop (lines 1–5) that run for exactly n - 1 times (with n = length(A)). Keeping this in mind, we can determine the best-case and worst-case of the algorithm. In the best-case the inner loop is never executed, whilst for the worst-case the inner loop is executed exactly j - 1 times for every iteration of the outer loop so:

$$T(n) = \sum_{j=2}^{n} (j-1)$$

This results in T(n) being the arithmetic series $\Sigma_{k=1}^{n-1} k$, so:

$$\text{T(n)} = \frac{n(n-1)}{2}$$

$$\boxed{\mathrm{T(n)}=\Theta(n^2)}$$

The Beast case is:

$$\boxed{\mathrm{T(n)}=\Theta(\mathrm{n})}$$

The Average case is:

$$\boxed{\mathrm{T(n)}=\Theta(n^2)}$$

3 Implementations

In this section, we look at several sorting algorithms that are going to be used in our comparison tests. We will differentiate between two types of algorithms, the first being comparison based, sorting elements by comparing them with one another, and the other type called non-comparison based algorithms, which do not sort by comparing elements. The implementations for the algorithms were done in the C programming language using Codeblocks. The source code used for the implementations can be found at: `https://github.com/AndiSova/Source-Code-Sorting-Algorithms-MPI-2021`.

3.1 Comparison based algorithms

Firstly, we will take a look at the comparison based algorithms.

3.1.1 Bubble Sort

Bubble Sort(see [3]) is one of the oldest sorting algorithms and one of the most inefficient ones. It sorts by interchanging two adjacent elements if they are found to be out of order with respect to each other. First, items x[0] and x[1] are compared and swapped if they are out of order. Next, x[0] and x[2] are compared, and their order is changed if necessary, and so on up to x[n-1] and x[n-2]. In this way, the smallest element is placed up to the top of the array. It has a time complexity of O(n^2).

```
 1: void Bubblesort(int a[], int n)
 2: {
 3:     int i, j;
 4:     for (i = 0; i <= n-1; i++)
 5:         for (j = 0; j <= n-i-1; j++)
 6:             if (a[j] > a[j+1]) {
 7:                 int temp=a[j];
 8:                 a[j]=a[j+1];
 9:                 a[j+1]=temp;
10:             }
11: }
```

3.1.2 Quicksort

Quicksort(see [1][2][5]) is a divide-and-conquer algorithm. The idea of Quicksort is to partitioning the input elements into two sequences that represent non-overlapping ranges of key values. Then, we sort the shorter sequences recursively and concatenate the results. Quicksort has an execution time of O(nlogn), with a worst case scenario of O(n^2).

```
 1: void swap(int* x, int* y)
 2: {
 3:     int t = *x;
 4:     *x = *y;
 5:     *y = t;
 6: }
 7: int Partition(int a[], int p, int r)
 8: {
 9:     int x = a[r];
10:     int i = (p - 1);
11:     for (int j = p; j <= r- 1; j++)
12:     {
13:         if (a[j] <= x)
14:         {
15:             i++;
16:             swap(&a[i], &a[j]);
17:         }
18:     }
19:     swap(&a[i + 1], &a[r]);
20:     return (i + 1);
21: }
22: void Quicksort(int a[], int p, int r)
23: {
24:     if(p < r)
25:     {
26:         int q;
27:         q = Partition(a, p, r);
28:         Quicksort(a, p, q-1);
29:         Quicksort(a, q+1, r);
30:     }
31: }
```

The *swap* function used in partition is used to swap two elements.

3.1.3 Insertion Sort

Insertion Sort(see [1]) is an efficient sorting algorithm when looking at small lists of numbers. It works similarly to sorting a hand of playing cards. We start with an empty hand with the cards on the table, face down. We remove a card at a time and insert it in the correct position on the left hand, comparing it with the cards already in the hand, from right to left. As we discussed in section 2, Insertion Sort has an average running time of O(n^2).

```
 1: void InsertionSort(int n,int a[])
 2: {
 3:     int i, j;
 4:     for (i = 1; i <= n; i++) {
 5:         j = i;
 6:         while (j > 0 && a[j-1] > a[j]) {
 7:             int temp=a[j-1];
 8:             a[j-1]=a[j];
 9:             a[j]=temp;
10:             j = j - 1;
11:         }
12:     }
13: }
```

3.1.4 Merge Sort

Merge Sort(see [2]) uses the divide-and-conquer principle. The unsorted list of elements is split into two parts of about equal size. Both parts are sorted recursively and are merged into a single sorted list. The globally smallest element is either the first element of x or the first element of y. We end up moving the smaller element to the output and finding the second smallest element using the same approach, iterating until all elements have been moved to the output. Each iteration of the inner loop of merge performs one element comparison and moves one element to the output. Because each iteration takes constant time, the running time of merging is linear. As such, Merge Sort runs in time O(nlogn) and performs no more than [nlog] element comparisons.

```
 1: void Merge(int a[],int p,
 2: int m,int r)
 3: {
 4:     int *t=malloc((r-p+1)
 5:     *sizeof*t);
 6:     int i=p,j=m+1,x=0;
 7:     while(i<=m && j<=r){
 8:         if(a[i]<=a[j]){
 9:             t[x]=a[i];
10:             x+= 1;
11:             i+=1;
12:         }
13:         else{
14:             t[x]=a[j];
15:             x+=1;
16:             j+=1;
17:         }
18:     }
19:     while(i<=m){
20:         t[x]=a[i];
21:         x+=1;
22:         i+=1;
23:     }
24:     while(j<=r){
25:         t[x]=a[j];
26:         x+=1;
27:         j+=1;
28:     }
29:     for(i=p;i<=r;i+=1)
30:         a[i]=t[i-p];
31:     free(t);
32: }
33: void Mergesort(int a[],int p,
34: int r)
35: {
36:    if(p<r) {
37:         int m=(p+r)/2;
38:         Mergesort(a,p,m);
39:         Mergesort(a,m+1,r);
40:         Merge(a,p,m,r);
41:     }
42: }
```

3.1.5 Heapsort

Heapsort(see [1]) similarly to Merge Sort has complexity O(nlogn). It sorts in place a constant number of array of elements, which are sorted outside the input array at any time. It introduces an algorithm design technique, namely a data structure called "heap", which manages information. Heapsort does not preserve the order of elements with equal keys, resulting in it being not a stable sorting algorithm.

```
1: void Heapify(int a[], int n, int i)
2: {
3:     int x = i;
4:     int l = 2 * i + 1;
5:     int r = 2 * i + 2;
6:     if (l < n && a[l] > a[x])
7:         x = l;
8:     if (r < n && a[r] > a[x])
9:         x = r;
10:    if (x != i) {
11:        swap(&a[i], &a[x]);
12:        Heapify(a, n, x);
13:    }
14: }
15: void BuildHeap(int a[],int n)
16: {
17:    for (int i = n / 2 - 1; i >= 0; i--)
18:        Heapify(a, n, i);
19: }
20: void Heapsort(int a[], int n) {
21:    BuildHeap(a,n);
22:    for (int i = n ; i >= 0; i--) {
23:        swap(&a[0], &a[i]);
24:        Heapify(a, i, 0);
25:    }
26: }
```

The function *swap* is used for the same purpose as the one found in Quicksort.

3.2 Non-comparison based algorithms

Next up, we will look at an intermediate algorithm, Counting sort, and the proper algorithm, Radix Sort.

3.2.1 Counting Sort

Counting Sort(see [1]) assumes that each of the n input elements is an integer in the range 0 to k, for some integer k. When k = O(n) the sort runs in Θ(n) time. Counting sort determines, for each input element x, the number of elements less than x. It uses this information to place element x directly into its position in

the output array. The worst case and average case performance of counting sort is O(n+k).

```
 1: void countingSort(int a[], int n, int x)
 2: {
 3:     static int r[9999999];
 4:     int i, k[10] = { 0 };
 5:     for (int i = 0; i <= n; i++) {
 6:         k[(a[i] / x) % 10]++;
 7:     }
 8:     for (int i = 1; i <= 10; i++)
 9:         k[i] += k[i - 1];
10:     for (int i = n -1; i >= 0; i--) {
11:         r[k[(a[i] / x) % 10] - 1] = a[i];
12:         k[(a[i] / x) % 10]--;
13:     }
14:     for (int i = 0; i <= n; i++)
15:         a[i] = r[i];
16: }
```

3.2.2 Radix Sort

Radix sort(see [1]) is a linear sorting algorithm, working without comparing any elements. It works by sorting data with keys, sorting each digit on the input element and for each of the digits in that element. Usually, it might start with the least significant digit and then follows with the next least significant digit till the most significant digit The run time complexity of radix sort is O(d·n).

```
 1: int getMax(int a[], int n)
 2: {
 3:     int max = a[0];
 4:     for (int i = 1; i < n; i++)
 5:         if (a[i] > max)
 6:             max = a[i];
 7:     return max;
 8: }
 9: void RadixSort(int a[], int n)
10: {
11:     int max = getMax(a, n);
12:     for (int x = 1; max / x > 0; x *= 10)
13:         countingSort(a, n, x);
14: }
```

The *getMax* function returns the largest element of a list.

4 Comparison of sorting algorithms

Considering the previous sections, we have looked at both theoretical ways to analyze an algorithm in terms of its complexity, and practical implementations of

the algorithms used in this paper. In this section we are going to look at several experiments with different test cases and analyze their respective results.

4.1 Practical Comparisons

The practical comparisons consist of a series of experiments at different input values that will give us different time scales for each individual algorithm, allowing for a more in depth view of how the algorithms fare with one another.

4.1.1 Case Study

The data inputs that we are going to use were randomly generated and start from 10 000 elements, going to 100 000 elements, 500 000 elements and finally 1 million elements. The lists generated consist of negative only, positive only and both positive and negative randomly sorted integer numbers. The data was generated using the C language and stored in files. The system which ran the algorithms consists of a Ryzen 5 4600H processor, an 500 GB SSD and 8GB of RAM DDR4. The code and the data inputs used for the practical experiments can be accessed at:`https://github.com/AndiSova/Data-inputs-for-MPI-2021`.

```
unsigned long x;
int i;
FILE *fp;
fp=fopen("Output.txt","w");
for(i=-1; i<1000000; i++){
    x = rand();
    x <<= 15;
    x ^= rand();
    x %= 1000001;
    fprintf(fp, "%d\n", x);
}
fclose(fp);
```

List1: Positive Numbers

```
unsigned long x;
int i;
srand(time(NULL));
FILE *fp;
fp=fopen("Output.txt","w");
for(i=-1; i<10000; i++)
{
    x=rand()%10000+(-10000);
    frintf(fp,"%d\n",x);
}
fclose(fp);
```

List2: Negative Numbers

```
unsigned long x;
int i;
srand(time(NULL));
FILE *fp;
fp = fopen("Output.txt", "w");
for(i=-1; i<1000000; i++){
    x = rand() %1000000+ (-10000);
    fprintf(fp, "%d\n", x);
}
fclose(fp);
```

List3: Integer Numbers

4.1.2 Analysis of the Case Studies

We will now look at the running times, measured in seconds, the three lists give. The results down below consist of the average value given by the run times of each list, which were run 3 times by each algorithm. We expect that either Quicksort or Radix Sort will be the fastest algorithms when looking at positive lists, and for the negative and integer lists Quicksort would perform the fastest. With that being said the other two sorting algorithm with O(nlogn) complexity should have very close run times with the ones from Quicksort. For the O(n^2) algorithms we expect to run well at small input data, but struggle at larger ones. The time of the experimental tests was measured both by looking at the time given by the console when running the algorithms, and by using a chronometer to testify the correctness of the console timer.

We are going to start with the positive only list:

Number of elements	Bubble Sort	Quick Sort	Insertion Sort	Merge Sort	Heap Sort	Radix Sort
10 000 elements	0.359	0.078	0.098	0.065	0.08	0.46
100 000 elements	34.12	0.138	0.898	0.098	0.109	0.071
500 000 elements	605	0.219	191.49	0.275	0.306	0.138
1 million elements	2760.5	4.384	540.32	0.926	0.828	0.177

Next up we are going to take a look at the negative only lists:

Number of elements	Bubble Sort	Quick Sort	Insertion Sort	Merge Sort	Heap Sort	Radix Sort
10 000 elements	0.401	0.075	0.086	0.084	0.051	∅
100 000 elements	35.57	0.106	1.676	0.0951	0.085	∅
500 000 elements	612.66	0.187	151.44	0.251	0.293	∅
1 million elements	3128.9	4.561	641.35	1.05	0.796	∅

Finally, we are going to look at the results from the integer lists:

Number of elements	Bubble Sort	Quick Sort	Insertion Sort	Merge Sort	Heap Sort	Radix Sort
10 000 elements	0.362	0.067	0.119	0.064	0.044	∅
100 000 elements	40.54	0.183	1.149	0.085	0.093	∅
500 000 elements	713.4	0.244	178.34	0.34	0.334	∅
1 million elements	2987.3	3.102	601.65	0.973	0.905	∅

By observing the tables presented above, we can observe quite a difference between algorithms with complexity O(nlogn) and O(n+d) when put aside algorithms with O(n^2) time complexity. During the test regarding positive only numbers, the fastest algorithm was Radix Sot at any of the input values given, followed by Heapsort, Merge Sort and Quicksort. This result is not surprising

considering its time complexity, but what is surprising is the much better performance given by one of the two O(n^2) algorithms, that being Insertion Sort, for the fact that the difference between it and Bubble Sort is quite large, even though both had pretty high run times at large input values. It is also surprising to see Quicksort being outperformed by the other O(nlogn) algorithms when looking at large input values, considering the fact that for the other values it outperformed the rest, with the exception of Radix Sort. When looking at the negative only lists, the results seem to be pretty similar to the ones at positive values, some algorithms, Bubble Sort for example, performing worse, while algorithms like Heapsort performing slightly better than at positive only lists. Unsurprisingly, for both negative and integer lists, Radix Sort is unable to sort them. Lastly, the integer lists gave similar results to the negative lists, some algorithms performing slightly better than when sorting negative only lists, with Quicksort performing faster with one second than the other two lists.

4.2 Theoretical Comparisons

Looking at the theoretical study(see [1][2][3]), we will compare the algorithms in terms of their time complexity in the three known cases: average, best and worst, along with their stability and methods of working. The following table looks at the differences between the implemented algorithms, n defining the number of elements that are going to be sorted, and d defining the range of numbers in the lists.

Algorithm	Stable	Method	Best Case	Worst Case	Average Case
BubbleSort	Yes	Exchange	O(n^2)	O(n^2)	O(n^2)
Quicksort	No	Partition	O(nlogn)	O(n^2)	O(nlogn)
InsertionSort	Yes	Insertion	O(n)	O(n^2)	O(n^2)
MergeSort	Yes	Merging	O(nlogn)	O(nlogn)	O(nlogn)
Heapsort	No	Selection	O(nlogn)	O(nlogn)	O(nlogn)

Table 4.1: Comparison of comparison based sorting algorithms

Algorithm	Stable	Best Case	Worst Case	Average Case
CountingSort	Yes	O($n \cdot d$)	O($n \cdot d$)	O($n \cdot d$)
RadixSort	No	O(n+d)	O(n+d)	O(n+d)

Table 4.2: Comparison of non-comparison based sorting algorithms

Looking at the tables we can clearly see that the algorithms with the best cases are Merge Sort and Heapsort, both having for each case complexity O(nlogn), resulting in a better overall complexity when looking even at Quicksort which has a worst case scenario of O(n^2). The best case scenario is achieved, however,

by Insertion Sort, having complexity O(n) when looking at sorted lists. Both Counting Sort and Radix Sort depend on the range of numbers in the list.

5 Related Work

In this section, we are going to look at similar results that are relevant for the problem of comparing sorting algorithms. In their article(see [7]), Ashutosh Bharadwaj and Shailendra Mishra present several known sorting algorithms, namely Bubble Sort, Insertion Sort, and Merge Sort, analyzing their complexity and run times. Even tough the purpose of their article is to present a new sorting algorithm called Index Sort, the article clearly shows the superiority of Merge Sort over the other algorithms and the outdated nature of Bubble Sort, which is the slowest one among the algorithms. In the article (see [6]) of Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith and Marco Zagha concludes in a positive view over Radix Sort and Quicksort, both algorithms being easy to implement and giving fast run times, with both having performances comparable to one another.

6 Conclusion and Future Work

This paper presented a comparison of several sorting algorithms, along with an outlook at their respective complexities and the results of their run time performance. Several key results comprise of Quicksort not being the fastest sorting algorithm at large data inputs, being outperformed by Merge Sort, Radix Sort, and Heapsort, and the large difference between algorithms with complexity $O(n^2)$, namely Insertion Sort and Bubble Sort as they perform very differently at large data inputs. Radix Sort, as expected, had the fastest running time at positive data inputs, and was unable to sort the negative and integer lists.

For all the algorithms that we analyzed in this paper, there exist ways to improve them and drastically change their respective run times. For example (more information at [4]), Bubble Sort can be enhanced by sorting the elements in the same array by finding the minimum and the maximum of its array, exchanging the minimum with the first element and the maximum with the last element, decreasing as such the size of the array by two for next call.

Besides enhancing current algorithms, for a more detailed look at different run times, the comparison experiments can be done when looking at lists that are sorted, almost sorted, reverse sorted, and lists that have similar numerical values, or the comparison can be done by looking at more sorting algorithms then the ones in this paper(more information at [1][2][3]).

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein *Introduction to Algorithms*. The MIT Press, 2009.

[2] Kurt Mehlhorn and Peter Sanders *Algorithms and Data Structures The Basic Toolbox*.

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

[4] Jehad Alnihoud and Rami Mansi *An Enhancement of Major Sorting Algorithms*. Department of Computer Science, Al al-Bayt University, Jordan

[5] C. A. R. Hoare *Quicksort*. The Computer Journal, 1962

[6] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha *A Comparison of Sorting Algorithms for the Connection Machine CM-2*.

[7] Ashutosh Bharadwaj and Shailendra Mishra *Comparison of Sorting Algorithms based on Input Sequences*. International Journal of Computer Applications (0975 – 8887)Volume 78 – No.14, September 2013

[8] Gabriel Istrate *Course 3: Basics of Complexity Analysis(continued)Sorting: Insertion Sort*.