Sorting Algorithms

Hamza Kamal

March 7, 2024

Contents

1 Introduction

In this report I will be discussing how different sorting algorithms perform and analyze their time complexities and compare them to one another.

1.1 Bubble Sort

Here is the implementation of the bubble sort algorithm:

```
void BubbleSort(vector<int>& vec, int start, int end) {
    int length = vec.size();
    for (int i = end; i > start; i--) {
        for (int j = 0; j < i; j++) {
            if (vec[j] > vec[j+1]) {
                swap(vec, j, j+1);
            }
        }
    }
}
```

- Best Case: $O(n)$: This happens when the list is already sorted and the bubble sort just performs $O(n)$ comparisons.
- Average Case: $O(n^2)$
- Worst Case: $O(n^2)$: This usually happens when the largest element is placed in the start of the list which makes bubble sort compare and swap through the entire array to find the correct index for the largest element.

This is also shown by the graphs that I made by timing the function and collecting data.

Here is the Log-Log plot for the bubble sort sorting function:

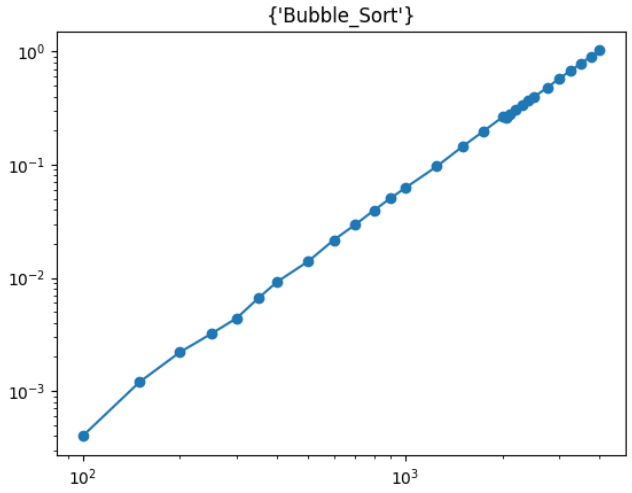

Figure 1: Bubble Sort Log-Log Plot

Here is the normal plot for the bubble sort function:

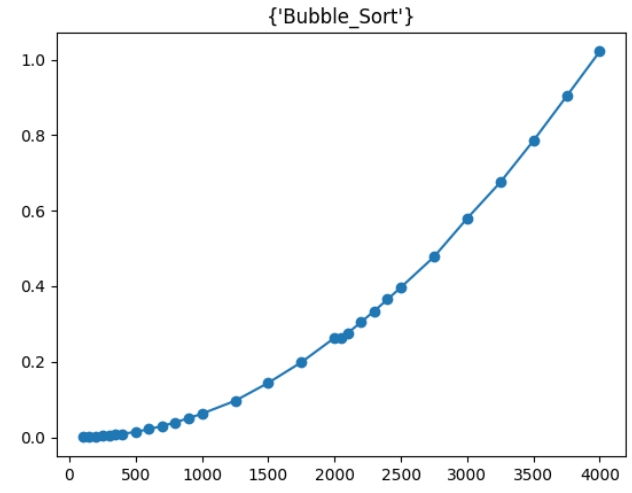

Figure 2: Bubble Sort Plot

As demonstrated by the graphs the time complexity of the bubble sort algorithm is $O(n^2)$.

Some ways that bubble sort can be improved is by implementing the adaptive bubble sort algorithm that takes advantage of the pre-sorted list by tracking if any swaps were made during a pass. With this the algorithm can stop as soon as the list is sorted removing the unnecessary comparisons.

1.2 Insertion Sort

Here is the implementation of the insertion sort function:

```
void InsertionSort(vector<int>& vec, int start, int end) {
    int length = vec.size();
    for (int i = start + 1; i < end; i++) {
        int val = vec[i];
        int j = i - 1;
        while ((j >= 0) && (vec[j] > val)) {
            vec[j+1] = vec[j];
            j -= 1;
        }
        vec[j+1] = val;
    }
}
```

- Best Case: $O(n)$: This happens when the list is already sorted just like bubble sort.
- Average Case: $O(n^2)$
- Worst Case: $O(n^2)$: This happens when the list is reversed.

Here is the Log-Log plot for insertion sort:

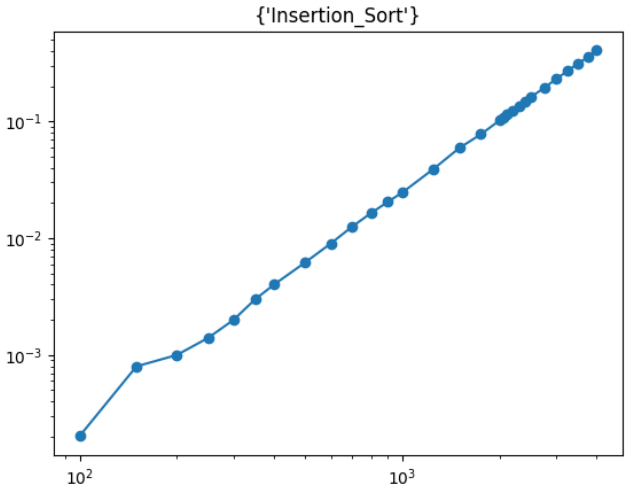

Figure 3: Insertion Sort Log-Log Plot

Here is the normal plot for insertion sort:

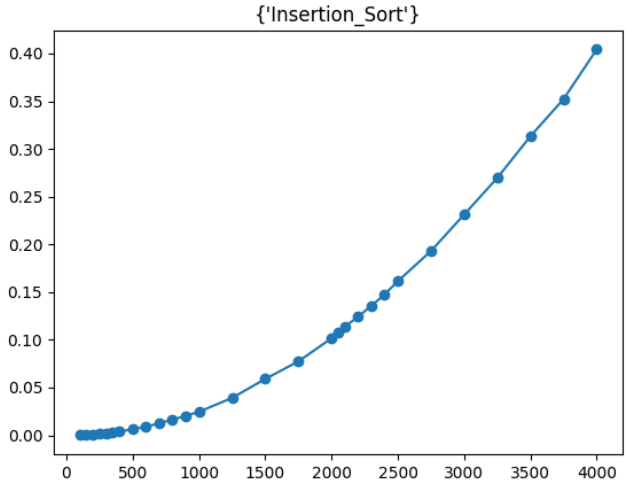

Figure 4: Insertion Sort Plot

As shown in the graphs the time complexity of insertion sort is $O(n^2)$.

We can improve insertion sort by using binary search instead of linear search

to find indexes for elements. This reduces our time complexity because we are able to find elements faster.

1.3 Merge Sort

Here is the implementation of the merge sort function:

```
void MergeSort(vector<int>& vec, int start, int end) {
    if (start < end) {
        int mid = start + (end - start) / 2;
        vector<int> left(vec.begin() + start, vec.begin() + mid + 1);
        vector<int> right(vec.begin() + mid + 1, vec.begin() + end + 1);

        MergeSort(left, 0, mid - start);
        MergeSort(right, 0, end - mid - 1);

        int i = 0, j = 0, k = start;
        while (i < left.size() && j < right.size()) {
            if (left[i] <= right[j]) {
                vec[k++] = left[i++];
            } else {
                vec[k++] = right[j++];
            }
        }

        while (i < left.size()) {
            vec[k++] = left[i++];
        }

        while (j < right.size()) {
            vec[k++] = right[j++];
        }
    }
}
```

- Best Case: $O(n \log n)$: This happens when all elements of the first array are less than the elements of the second array.
- Average Case: $O(n \log n)$
- Worst Case: $O(n \log n)$: When the list is sorted in descending order.

Here is the Log-Log plot for merge sort:

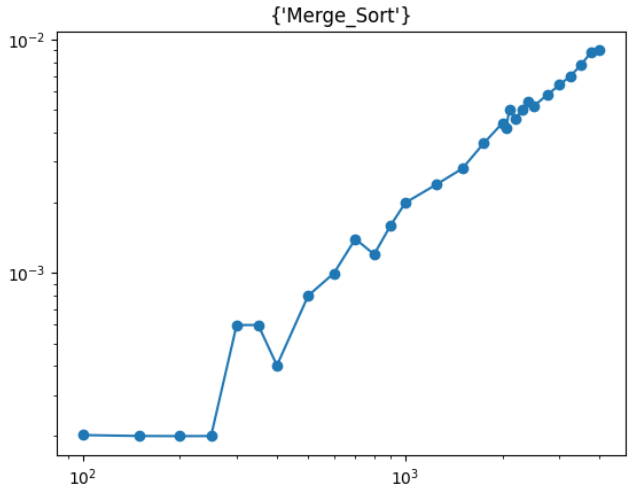

Figure 5: Merge Sort Log-Log Plot

Here is the normal plot for merge sort:

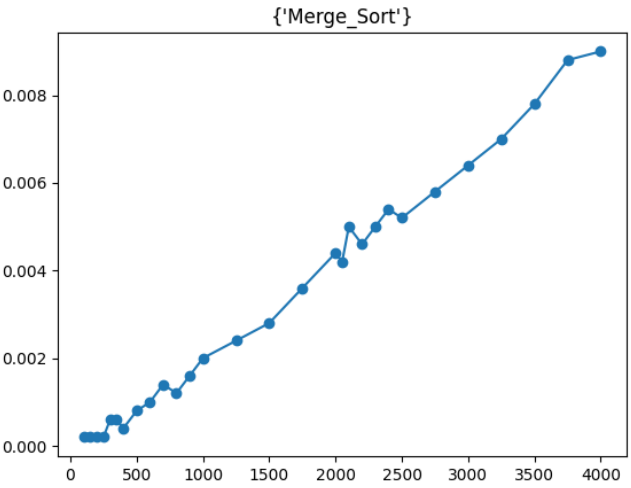

Figure 6: Merge Sort Plot

As demonstrated by the graphs the time complexity of the merge sort function is $O(n \log n)$.

The improved version of merge sort is the iterative merge sort which uses loops instead of recursion to implement the merge sort function.

1.4 Iterative Merge Sort

Here is the implementation of iterative merge sort:

```
void IterativeMergeSort(vector<int>& vec, int start, int end) {
    vector<int> temp(end, 0);
    int size = 1;
    while (size < end) {
        int left = start;
        while (left < end - size) {
            int mid = left + size;
            int right = min(left + 2*size, end);
            merge(vec, left, mid, right, temp);
            left += 2*size;
        }
        size *= 2;
    }
}
```

- Best Case: $O(n \log n)$
- Average Case: $O(n \log n)$
- Worst Case: $O(n \log n)$

Here is the Log-Log plot for iterative merge sort:

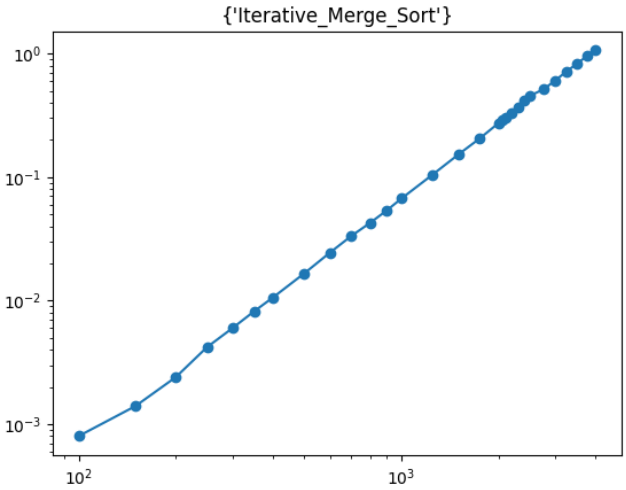

Figure 7: Iterative Merge Sort Log-Log Plot

Here is the normal plot for iterative merge sort:

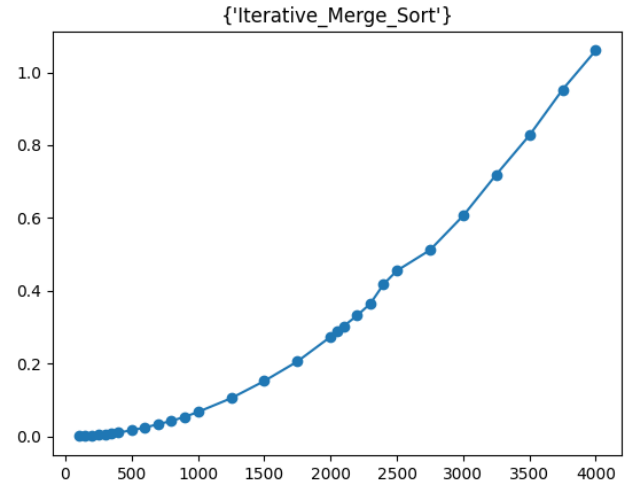

Figure 8: Iterative Merge Sort Plot

The Iterative merge sort function is slightly faster than the recursive version of merge sort. As demonstrated by the graphs.

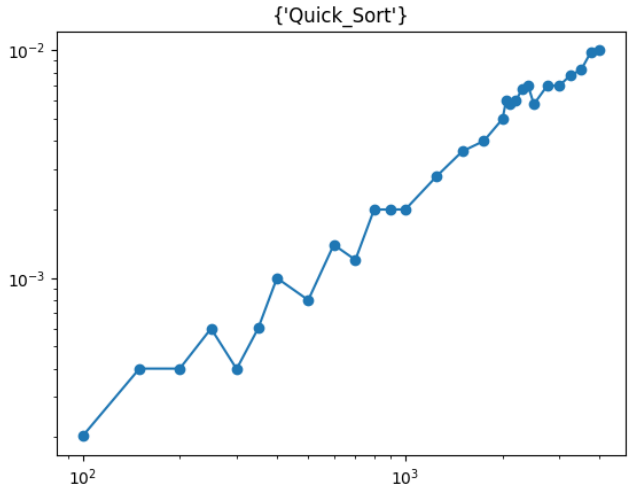

Figure 9: Quick Sort Log-Log Plot

1.5 Quick Sort

Here is the implementation of the quick sort function:

```
void QuickSort(vector<int>& vec, int left, int right) {
    if (left >= right) {
        return;
    }
    else if (left < right) {
        medianPivot(vec, left, right);
        quickSortHelper(vec, left, right);
    }
}
```

- Best Case: $O(n \log n)$: Depends on how the partition is chosen.
- Average Case: $O(n \log n)$
- Worst Case: $O(n \log n)$: Depends on how the partition is chosen.

Here is the Log-Log plot for quick sort:
Here is the normal plot for quick sort:
The time complexity of quick sort is $O(n \log n)$ as shown in the graphs.

The Quick Sort function has a better space complexity than the merge sort function even though the time complexities are the same. We can improve the

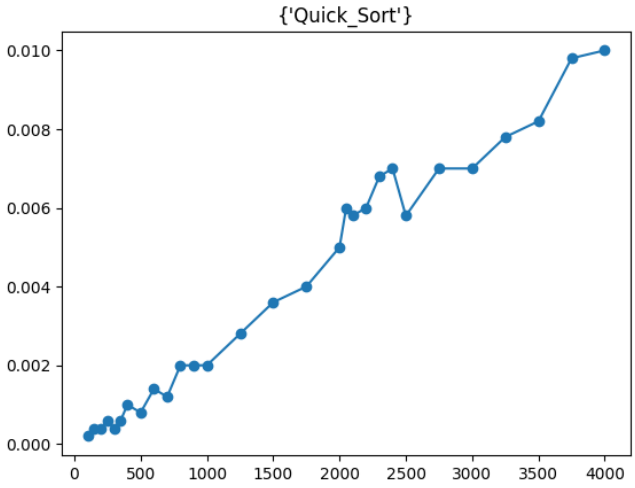

Figure 10: Quick Sort Plot

time complexity of quick sort by carefully choosing the partitions, so they result in lists that are favorable and easy to sort.

1.6 Shell Sort

Here is the implementation of shell sort:

```
void ShellSort(vector<int>& vec, int start, int end) {
    int gap = vec.size() / 2;
    while (gap >= 1) {
        for (int j = gap; j < end; j++) {
            int i = j - gap;
            while (i >= start) {
                if (vec[i+gap] > vec[i]) {
                    break;
                }
                else {
                    swap(vec, i+gap, i);
                }
                i -= gap;
            }
        }
        gap = gap / 2;
    }
```

```
}
```

- Best Case: $O(n)$: When the list is already sorted.
- Average Case: $O(n \log n)$
- Worst Case: $O(n^2)$: This happens when using the original gap sequence.

Here is the Log-Log plot of the shell sort function:

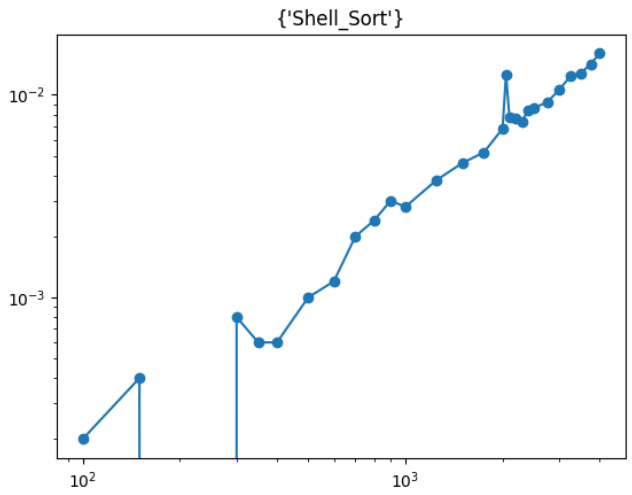

Figure 11: Shell Sort Log-Log Plot

Here is the normal plot for shell sort:

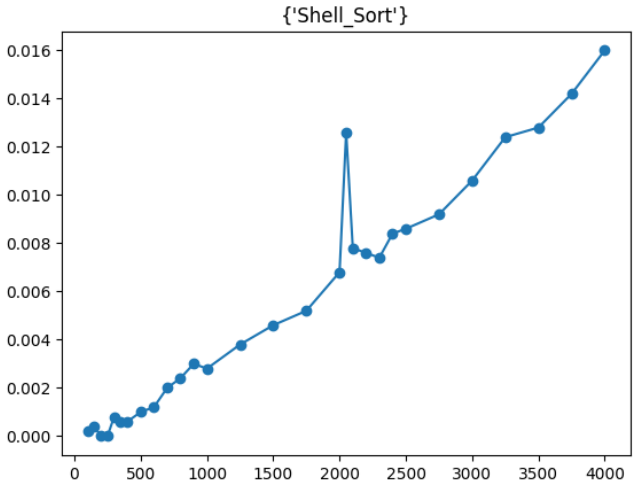

Figure 12: Shell Sort Plot

The graphs demonstrate that the time complexity of shell sort is $O(n \log n)$.

We can improve shell sort by using better gap sequences that have proven time complexities which are available on the shell sort wiki.

1.7 Comparison

Here I am going to compare all the different sorting functions.
Here is a Log-Log plot of all the sorting functions together:

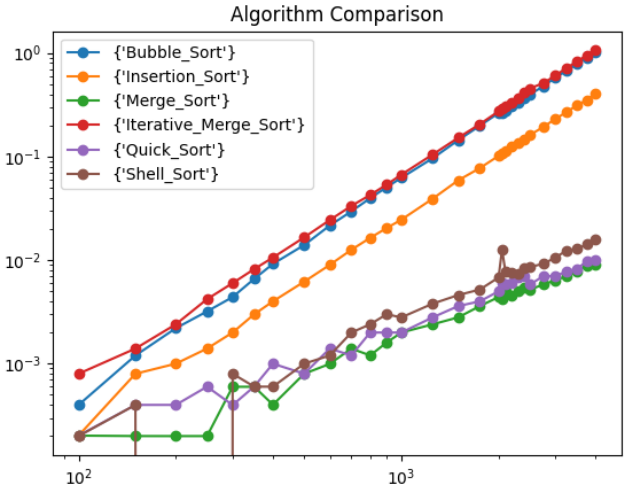

Figure 13: Sorting Log-Log Plot

Here is a normal plot of all the sorting functions:

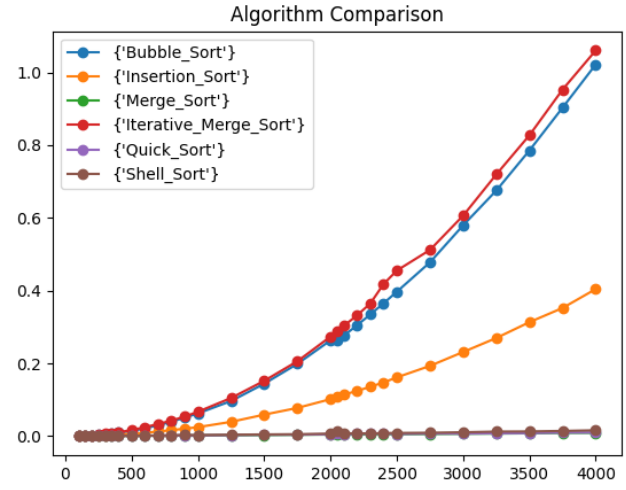

Figure 14: Sorting Plot

These graphs demonstrate that some sorting functions are better than others. For example:

- Quick Sort and Shell Sort are basically constant.

- The worst algorithm is Iterative Merge Sort
- Bubble Sort performs worse than Insertion Sort

1.8 Conclusion

In summary, this report looked at different ways to organize lists of numbers. We studied Bubble Sort, Insertion Sort, Merge Sort, Iterative Merge Sort, Quick Sort, and Shell Sort. We checked how fast they are and how well they work with big lists.

Bubble Sort and Insertion Sort are easy to understand but can be slow with big lists. Merge Sort and its iterative version are quicker and work well with big lists. Quick Sort is fast too, especially if we pick the right starting points. Shell Sort is also fast, especially when we use the right patterns.

When we compared them, Quick Sort and Shell Sort were consistently fast, even with big lists. But Bubble Sort and Iterative Merge Sort were slower, especially with lots of numbers.

We should research each sorting function as they have different use cases. For example shell sort is good at sorting arrays that have been almost sorted and just have a few outliers.