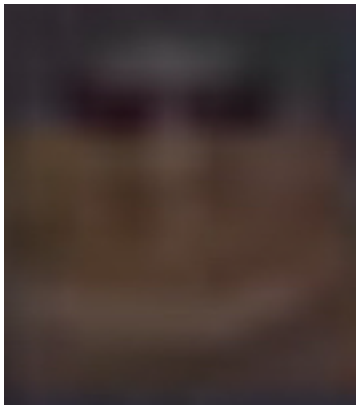
# Analysis of Sorting Algorithms

Ralph Miller

October 17th 2021



## Contents

# 1 Objectives

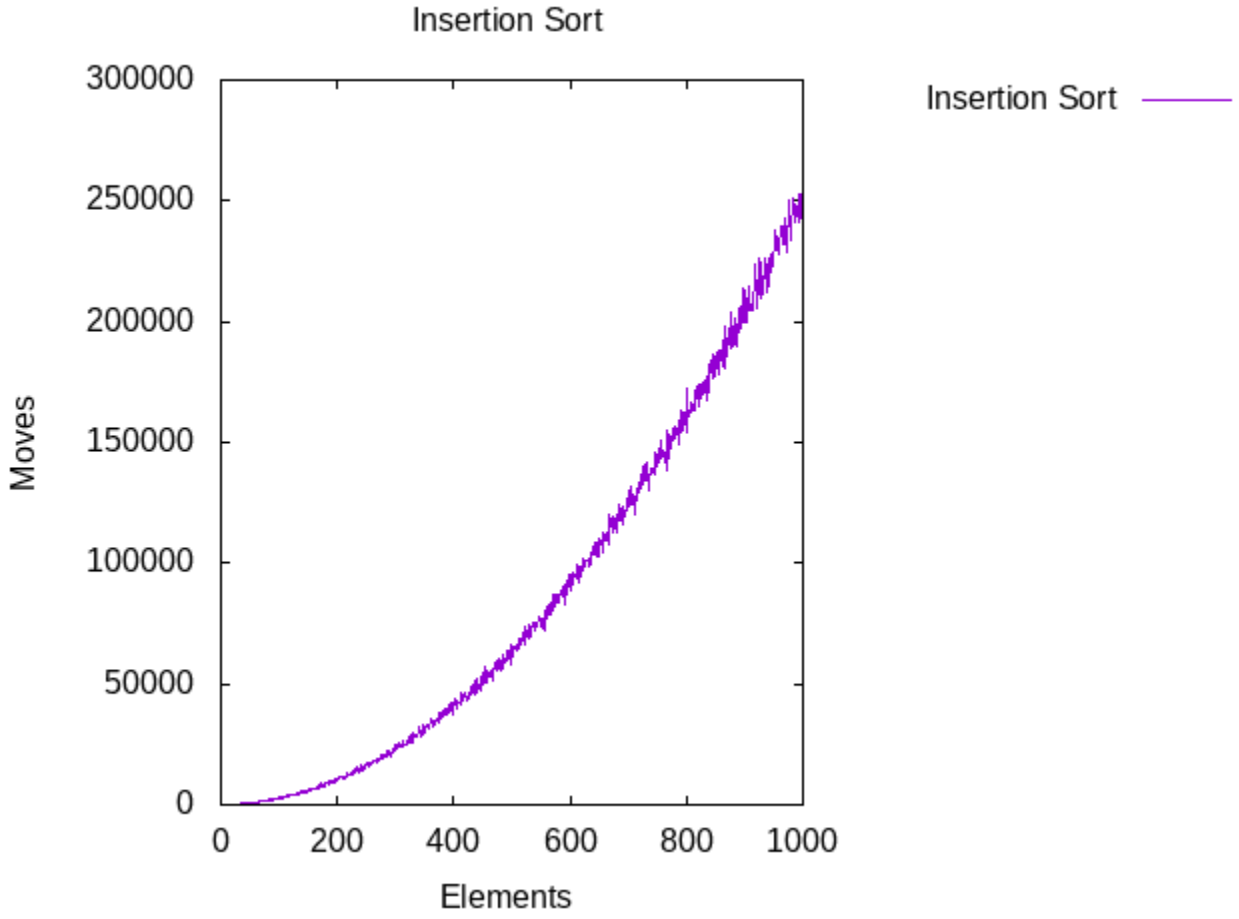
In this write up I will be analysing the four different sorting algorithms I have implemented in C. Insertion, Shell, Heap, and Quick Sort. I will be using the metrics collected within my program. Which count the amount of moves, comparisons, and swaps made to sort a randomly generated array of different sizes. First I will analyze each sorting algorithm, then I will compare the sorting algorithms to each other, concluding with the best use case for each sorting algorithm.

## 1.1 Defining Each Metric

- A move is defined as when one element of an array is moved to a new index or is stored in a variable.
- A swap is defined as when two addresses of array elements are swapped.
- A comparison is defined as when two elements of an array are compared with $(<, >, =)$ operators.
- Elements are defined as the number of elements in an array, represented as n. ie) an array[5] $= \{1, 2, 3, 4, 5\}$ contains five elements. n = 5.

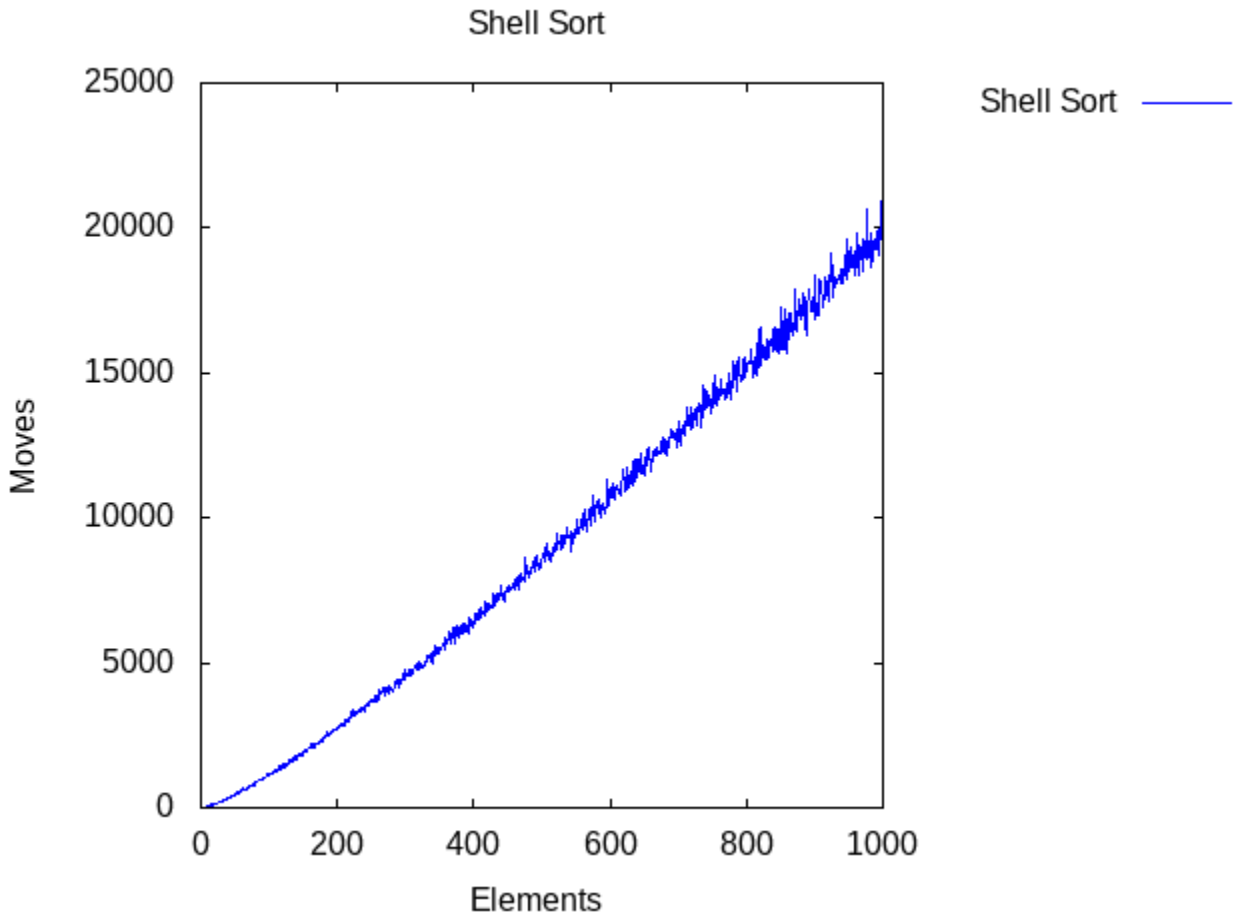
## 2 Insertion Sort

Insertion Sort is a fairly simple sorting algorithm, but as the number of elements grow it becomes very inefficient. With the amount of moves taken growing by its complexity $O(n^2)$ This is shown in the graph below.



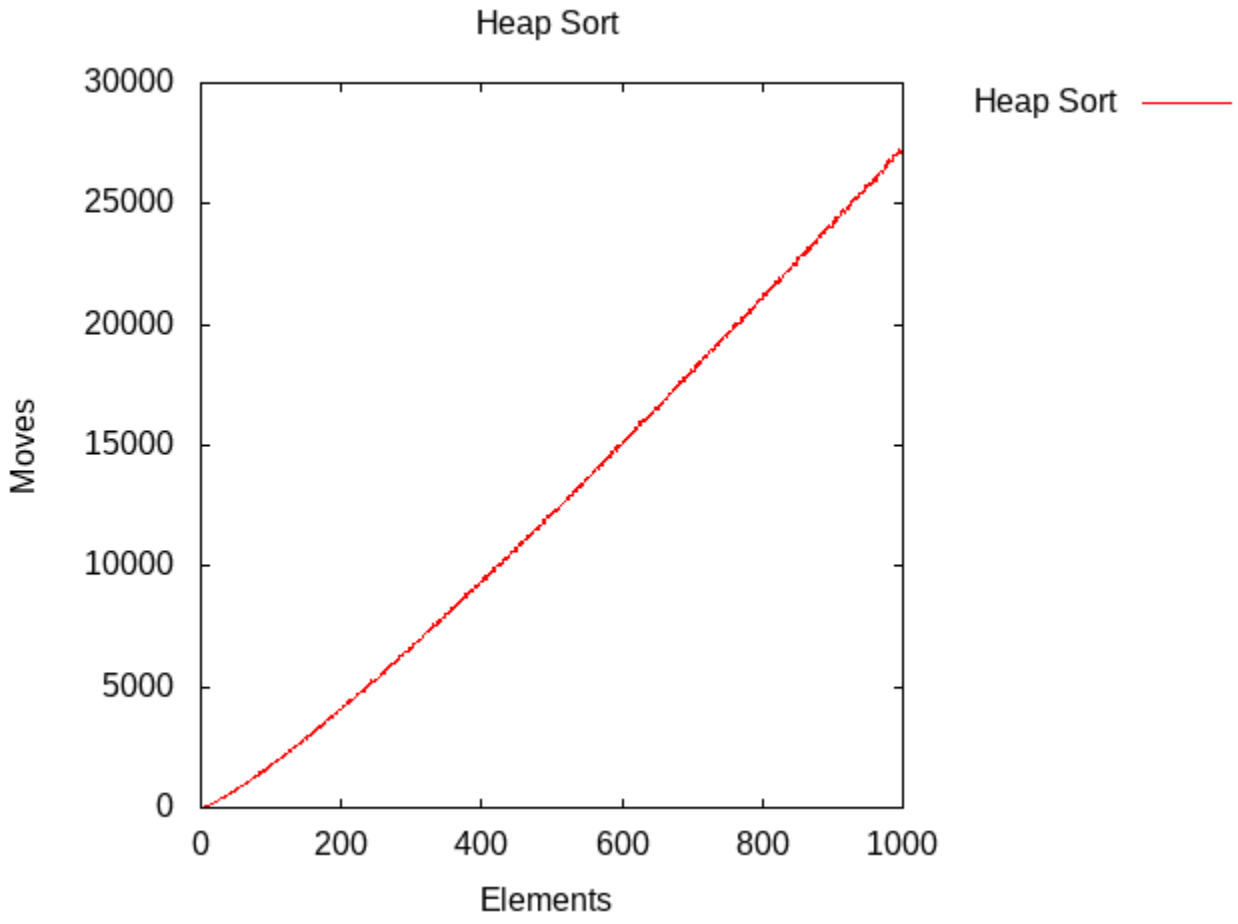
## 3 Shell Sort

Shell Sort is a sorting algorithm that is based off of Insertion Sort, that uses a gap sequence to sort elements that are a gap sequence away from each other. This changes the time complexity to the complexity of the gap sequence. In my implementation of Shell Sort I used Knuths gap sequence, $(3^k - 1)/2$ , which has a time complexity of $O(n^{3/2})$.



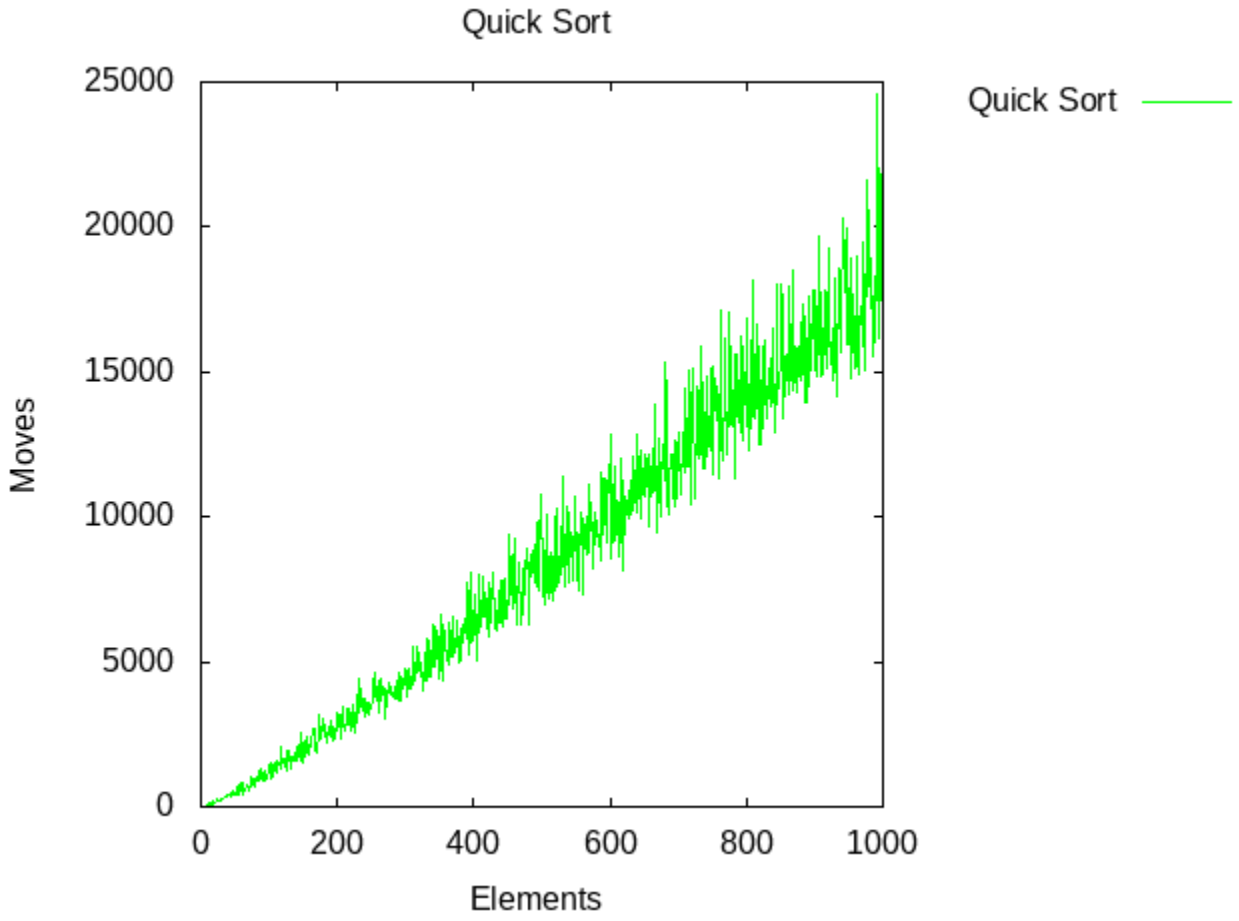
# 4 Heap Sort

Heap Sort is an interesting sorting algorithm that uses the heap data structure. My implementation of Heap Sort uses a Max Heap in which the parent node is greater than its two children. This sort is the most efficient when using arrays that store data in a heap structure. This would be the best use case for this sort. The time complexity of Heap Sort is $O(nlog(n))$.



## 5 Quick Sort

Quick Sort is an in place sorting algorithm, utilizing a pivot to partition two sub arrays around this pivot. Then recursively sorts the sub arrays. Its average time complexity is $O(nlog(n))$. With a worst case time complexity of $O(n^2)$. The worst case time complexity of $O(n^2)$ is seen in the spikes of the line graphed below.



# 6 Comparing Sorts

In this section I will be comparing each individual sorting algorithm to all of the other sorting algorithms. Concluding each sub section with the best use case for each individual sorting algorithm. Below the comparisons are the graphs will be referencing for these comparisons.

## 6.1 Comparing Insertion Sort

Best Time Complexity: $O(n^2)$
Use Case: $n < 50$

Looking at fig 1), it becomes apparent that Insertion Sorts relevant use case is for arrays of size $n < 50$. From $50 < n < 100$ Insertion Sort becomes the most inefficient sorting algorithm compared to Shell, Heap, and Quick sort. Looking at fig 2), this trend continues as Insertion Sort has dramatically more moves made than any other sorting algorithm.

## 6.2 Comparing Shell Sort

Best Time Complexity: $O(nlog(n))$
Best Use Case: As a more efficient replacement of Insertion Sort

Looking at fig 1), Shell Sort mirrors the amount if moves Insertion Sort makes until $n > 50$, we can see that Shell Sort steadily continues its trend while Insertion Sort decouples from Shell Sort. Shell Sorts trend stays in the middle of Quick Sorts volatile trend line, though the two take about the same amount of moves to sort an array on average as seen in fig 2), As well as being more efficient than Heap Sort.

## 6.3 Comparing Heap Sort

Best Time Complexity: $O(nlog(n))$
Use Case: For arrays already in the heap data structure.

Heap Sort is our third least-efficient sorting algorithm. But provides a stable trend line. As discussed previously, the best use case for Heap Sort would be when sorting an array that is already in a heap, or when stability is required.

## 6.4 Comparing Quick Sort

Best Time Complexity: $O(nlog(n))$
Use Case: When pre-existing conditions will provide a faster sort than Shell Sort.

Quick Sort is the most volatile sorting algorithm but can provide the fastest sorts out of all the algorithms given the order of the array being sorted. Looking at fig 1), Quick Sort is faster than Heap Sort and Insertion sort, as well as on average has the same amount of moves as Shell Sort.

Figure 1)

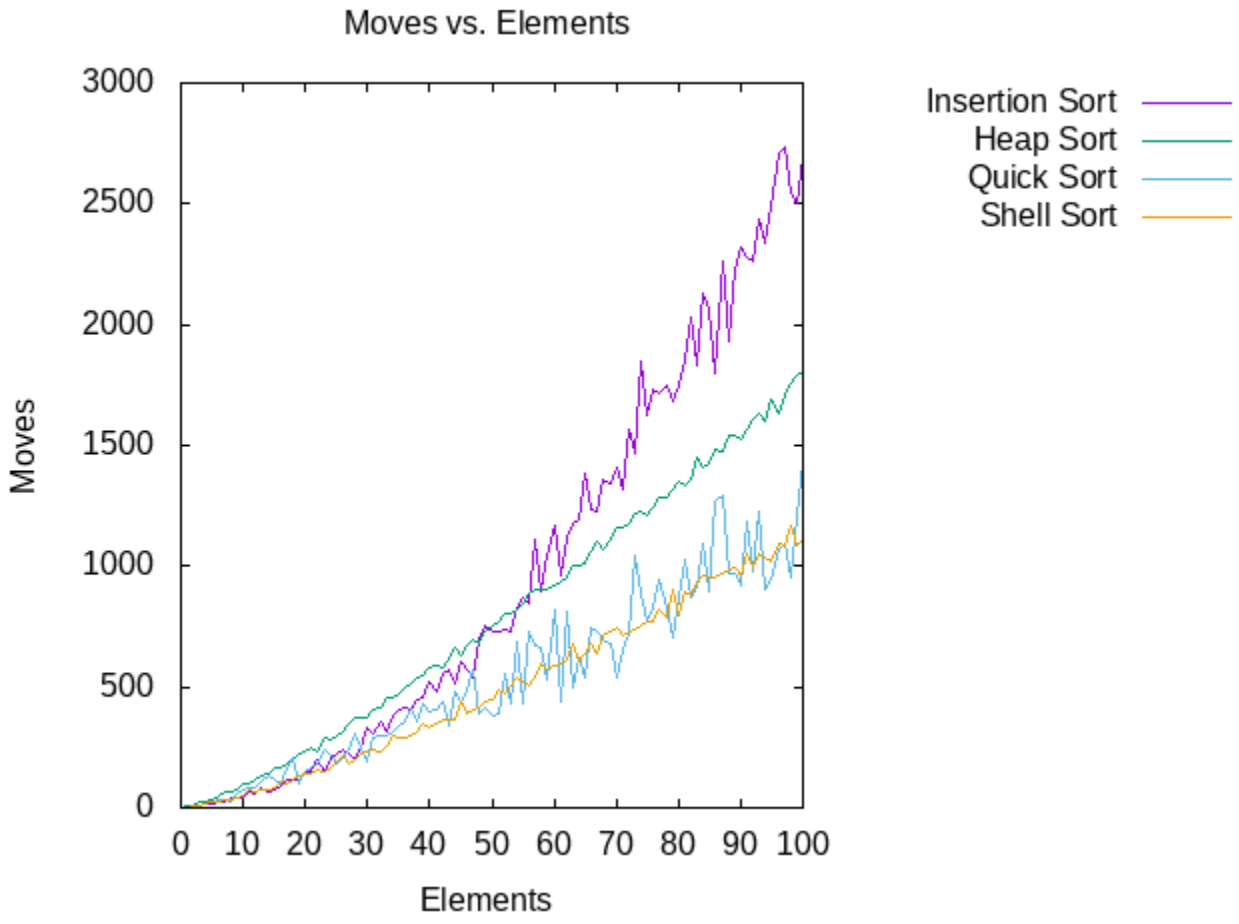
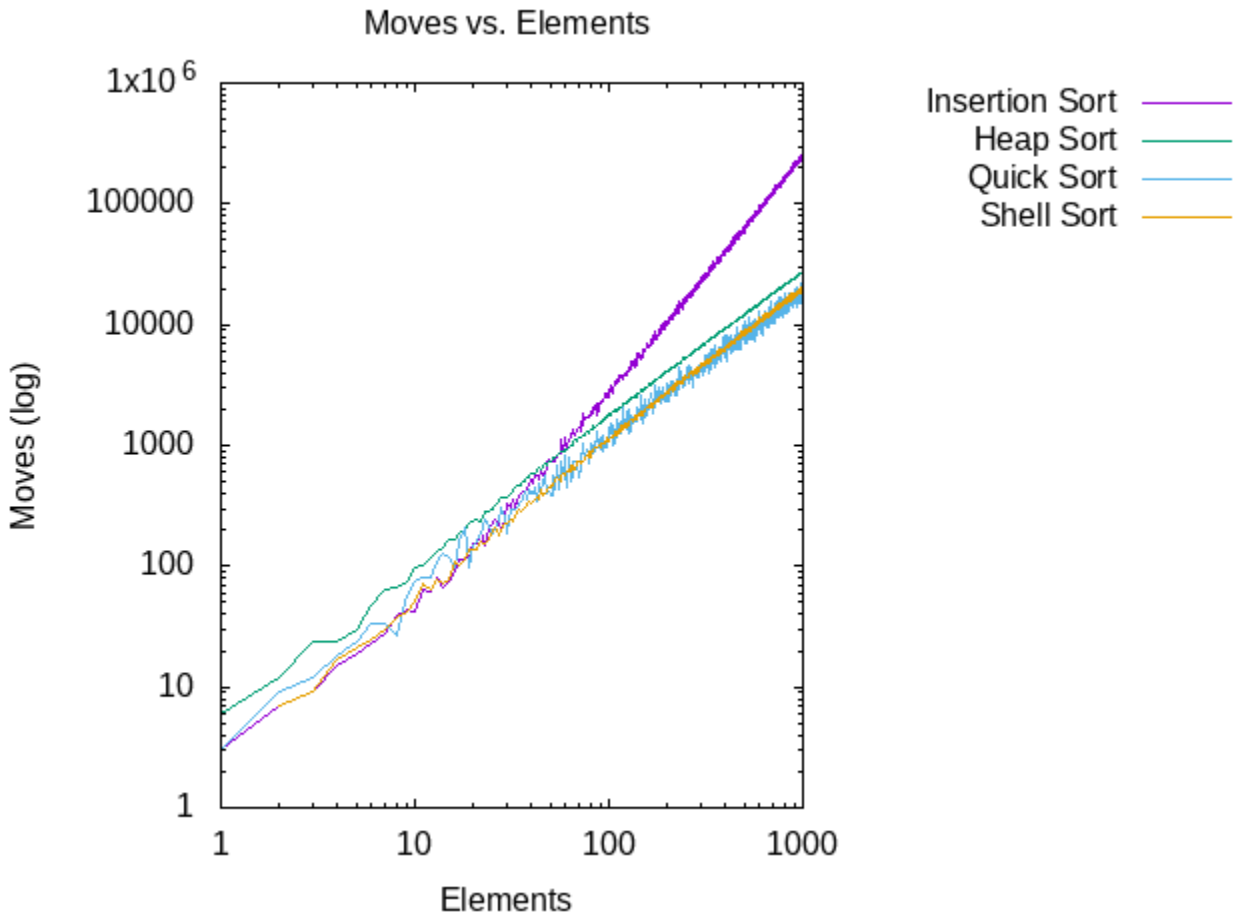

Moves vs. Elements
Moves
Elements
0
500
1000
1500
2000
2500
3000
0
10
20
30
40
50
60
70
80
90
100
Insertion Sort
Heap Sort
Quick Sort
Shell Sort

Figure 2)


Moves vs. Elements
1x10 6
100000
10000
1000
100
10
1
Moves (log)
1
10
100
1000
Elements
Insertion Sort
Heap Sort
Quick Sort
Shell Sort