The Efficiency of Sorting Algorithms in the C language

Alex Asch

October 2021

1 Abstract

This paper outlines the resulting efficiency of implementing various sorting algorithms in the C language. These given sorting algorithms all take arrays, which are collections of homogeneous elements, and sorts them into ascending order. From the results we can see differences in computational complexity, and the increasing effect it has as terms get larger.

2 Introduction

To begin, as outlined in the abstract, the nature of this research is to ascertain the relative computational complexity of sorting algorithms as implemented in C. The four algorithms that are being examined are insertion sort, shell sort, heap sort, and quick sort. The differences of these methods will be discussed in the next section on materials and methods. The constraining property of all of the above algorithms, is that the computer only has the ability to compare two elements at the once. In addition, the computer can only move elements one at a time. Thus, our efficiency will be measured in the number of comparisons and moves made.

3 Materials and Methods

Here we will go into more depth on the methods that these sorting algorithms were implemented in. To begin, the first algorithm implemented is insertion sort. This method begins at the first element, and compares each element to the elements to the left of itself, sliding that element down, and thus the rest of the array up, until there are no smaller values to the left of the element. This is implemented using two nested loops, one to iterate through the elements, and one to check the chosen element and move it to the correct position. The next method is Shell sort, this method is very similar to insertion sort. This method uses a sequence of gaps, for which the elements will be compared across. In

essence, the gap value is a certain distance between indices, and an insertion sort will be implemented for all elements within gap*k distance of an index. This will repeat until the gap value is equal to 0. Next, we have the method of heap sort. This sorting algorithm utilizes a heap, which is a type of binary tree that follows a min or max constraint. In this case we are using a max heap, thus for each value at k, it is greater than it's children, which are at 2k, and 2k+1 respectively. This method first builds this binary tree, and then after building the binary tree, swap the first and last element, thus moving the largest value to the back. It then fixes the binary tree, which means it makes sure that the array of the elements minus the last element which was just removed follows the constraints of a binary tree. It repeats this until all elements are popped off there is a sorted array. Finally, the last method used it quick sort. This method differs from the other methods, as it is recursive. This method works by first choosing a pivot, and sorting the array such that everything to the left of the pivot is smaller than it, and everything to the right of the pivot is larger than it. It then repeats this process on the two sub arrays to the left and right of the pivot, until there is an array of size 1, which means it is sorted.

4 Results

The results are comparing the number of moves and compares done in each sorting algorithm. This is seen via the below graphs. The first and very obviously notable result is extraordinary inefficiency of insertion sort for large element arrays. It can be seen that when the axes are not scaled as logarithms, the insertion sort function has such a large number of moves and compares such that the rest of the arrays are near obscured. The next interesting result is the change in quick sort as it increases, the graph of quick sort does not follow a uniform pattern, and instead is jagged. Outside of that it can be that heap, shell, insert, and quick are all similar in efficiency.

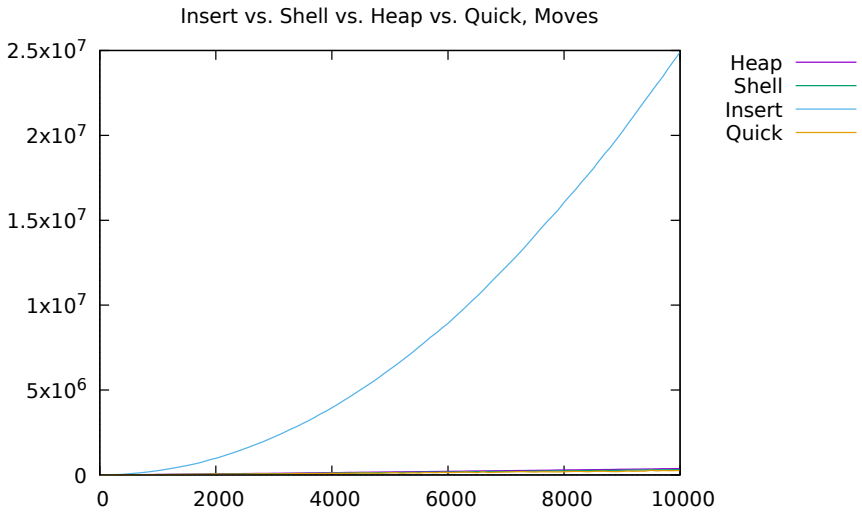

Figure 1: Number of Moves on a Large Domain

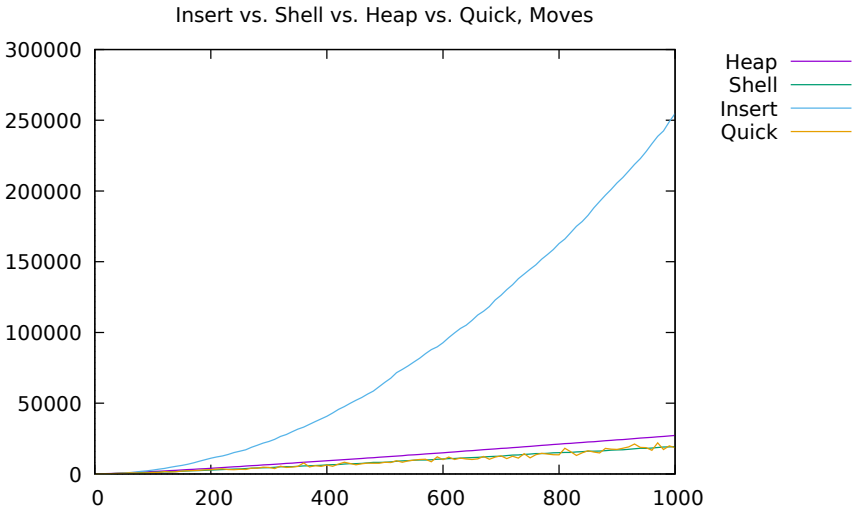

Figure 2: Number of Moves on a Smaller Domain

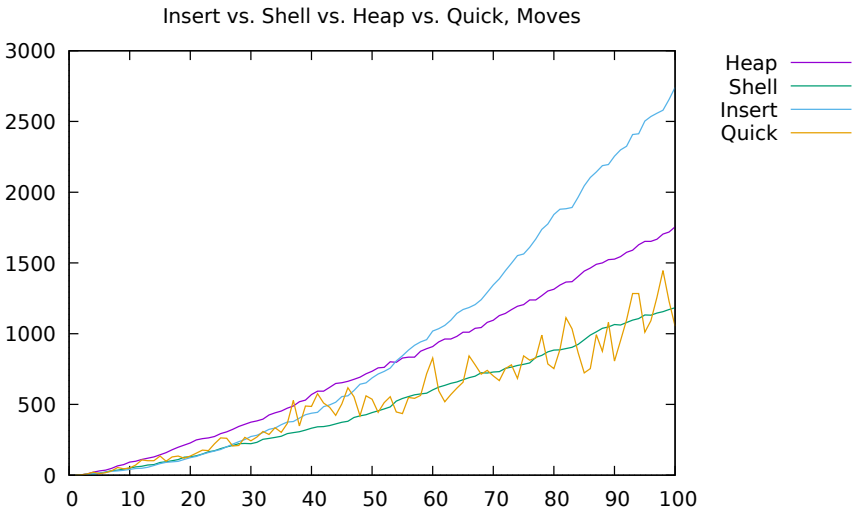

Figure 3: Number of Moves on a Smaller Domain

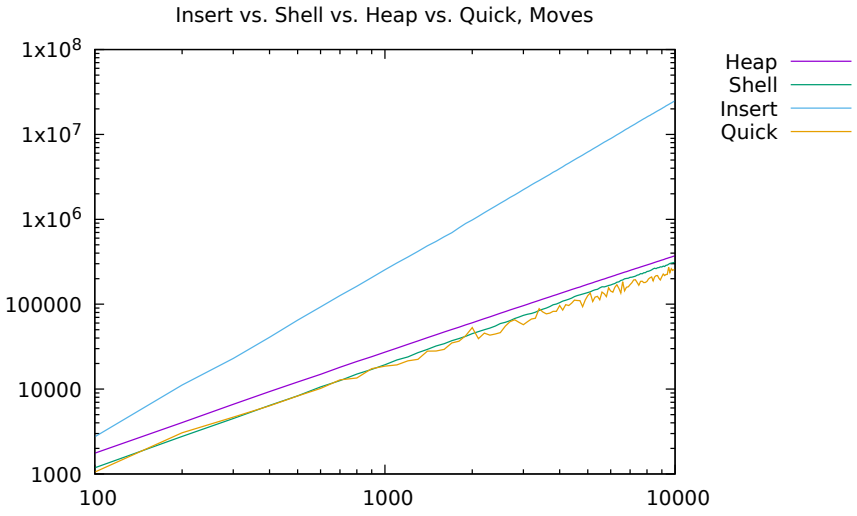

Figure 4: Number of Moves log scaled

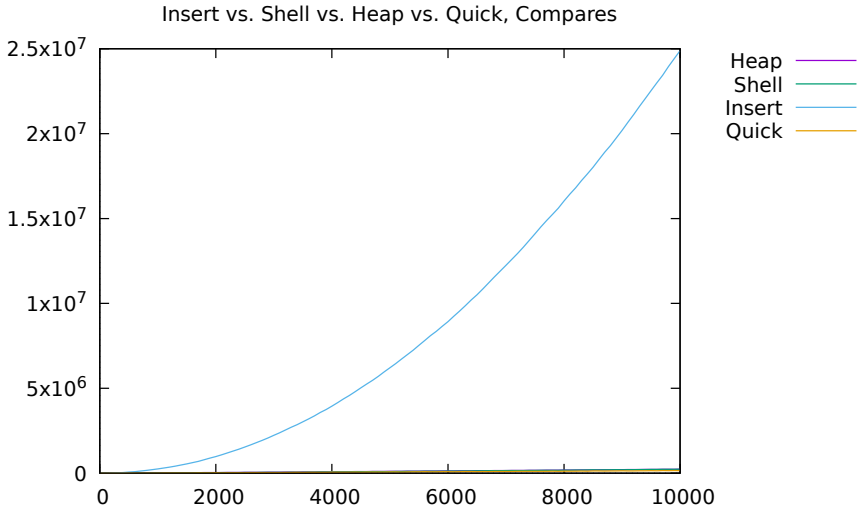

Figure 5: Number of Compares

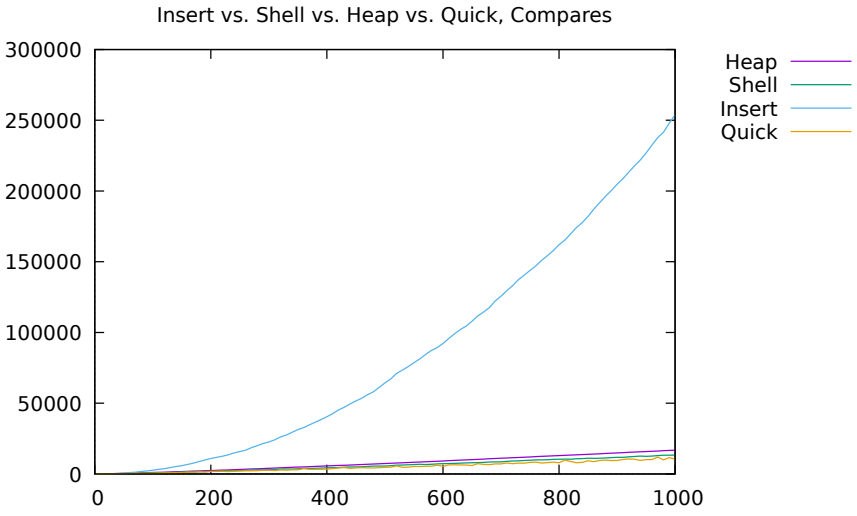

Figure 6: Number of Compares on a Smaller Domain

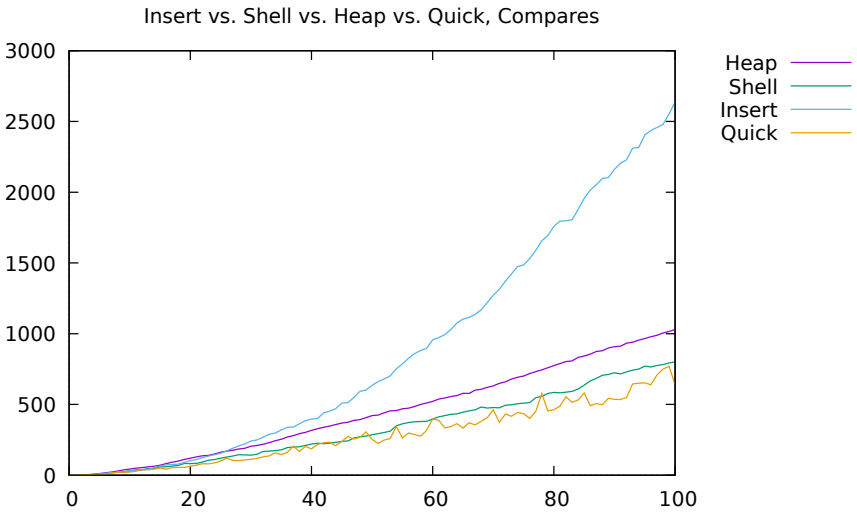

Figure 7: Number of Compares on a Smaller Domain

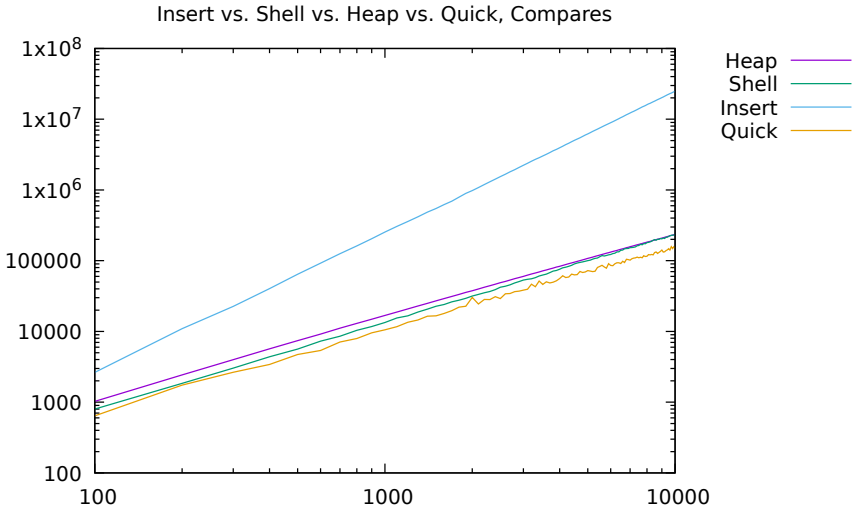

Figure 8: Number of Compares, log scaled

5 Discussion

We now will discuss why the results are as seen above. First to note is the two nested loops in insertion sort, which causes the amount of moves and compares to increase rapidly. In fact, since the nested loops both loop over the array, the time complexity of insertion sort is $O(n^2)$. The function of n^2 increases exponentially, thus leading to the rapid increase seen. Next we examine the jagged behavior of quick sort. This behavior is caused since the number of iterations change based on the given array. This is due to the method in which the pivot and sub arrays are determined. Since the pivot point cuts the array into two pieces, one less than, one greater than, the length of the sub arrays are not uniform. This means that depending on the original array, the length of the sub arrays, and thus total number of iterations can differ. Since in the worst case, the list only shrinks by one every time, and the list is iterated over each time, this results in quadratic time complexity, and thus a worst case of $O(n^2)$. Conversely, the best case scenario has a much smaller complexity, if the pivot bisects the list consistently. To begin, the amount of times something can be bisected is logarithm base 2 of the length. Thus this part of the ideal case is $O(log_n)$ complexity. In addition to this, each time the pivot must iterate over the list to find the values greater than or less than itself, which is an $O(n)$ complexity calculation. With these together we find the best case of quick sort is $O(nlog_n)$. The next algorithm to examine is heap sort. This algorithm first creates a binary tree. For a binary tree, the depth of the tree doubles every time, thus we again look at the amount of times we can bisect an array, which is $O(log_n)$ this means that making the array a heap or fixing the heap has the

above time complexity. From here, we examine how many times this is done. Since we pop off one element every time, it is done n times. Thus the complexity of heap sort is $O(nlog_n)$. Finally, we have the quick sort algorithm. Knuth's shell sorting algorithm is stated to have time complexity $n^{3/2}$. This is due to how the gaps are calculated, in Knuth's algorithm, the gaps are calculated as $\frac{3^k-1}{2}$, where the largest k is $\frac{log(2n+3}{log(3)}$. Due to this gap generation method, the number of iterations is dictated by the number of gaps, which is $n^{3/2}$. Clearly, the findings in the above graphs are mirrored by the relative complexities of these methods.

6 Acknowledgment

In this section, I would first like to acknowledge Professor Long for firstly, the necessary skills in coding in the C language to implement the above functions, for the assignment document, which provided python code to assist with implementation of the algorithms in, and for information on the interpretation of time complexity.

In addition, I would like to acknowledge Eugene Chou for the guide to usage of gnuplot in section, thus allowing me to present my findings in graphs, and the more broken down explanation of the sorting algorithms, to help my understanding.