Sorting: A Closer Look

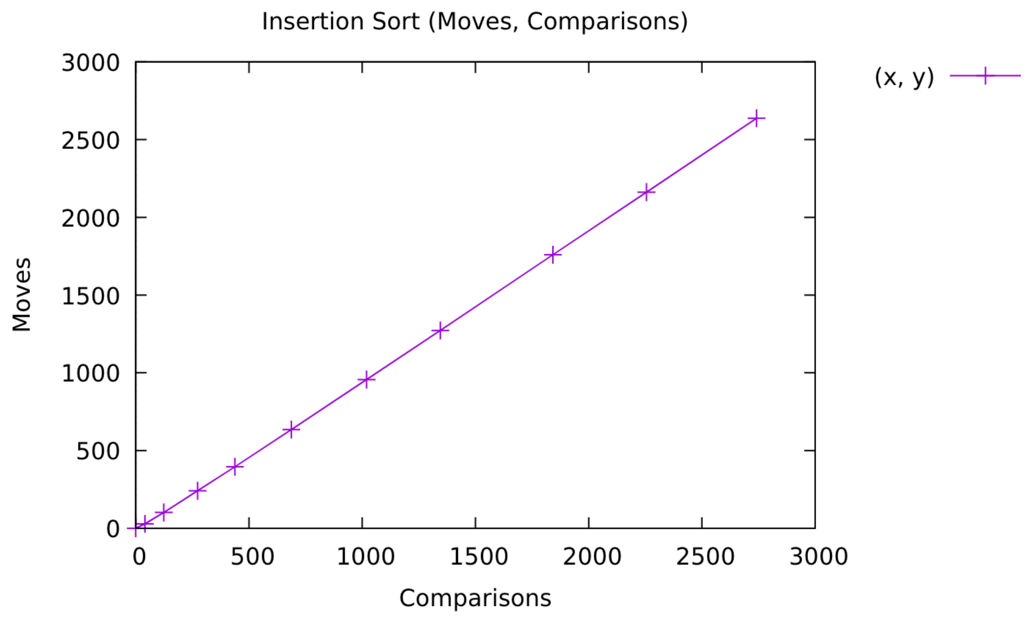
Analysis and Discussion

Christian Fong

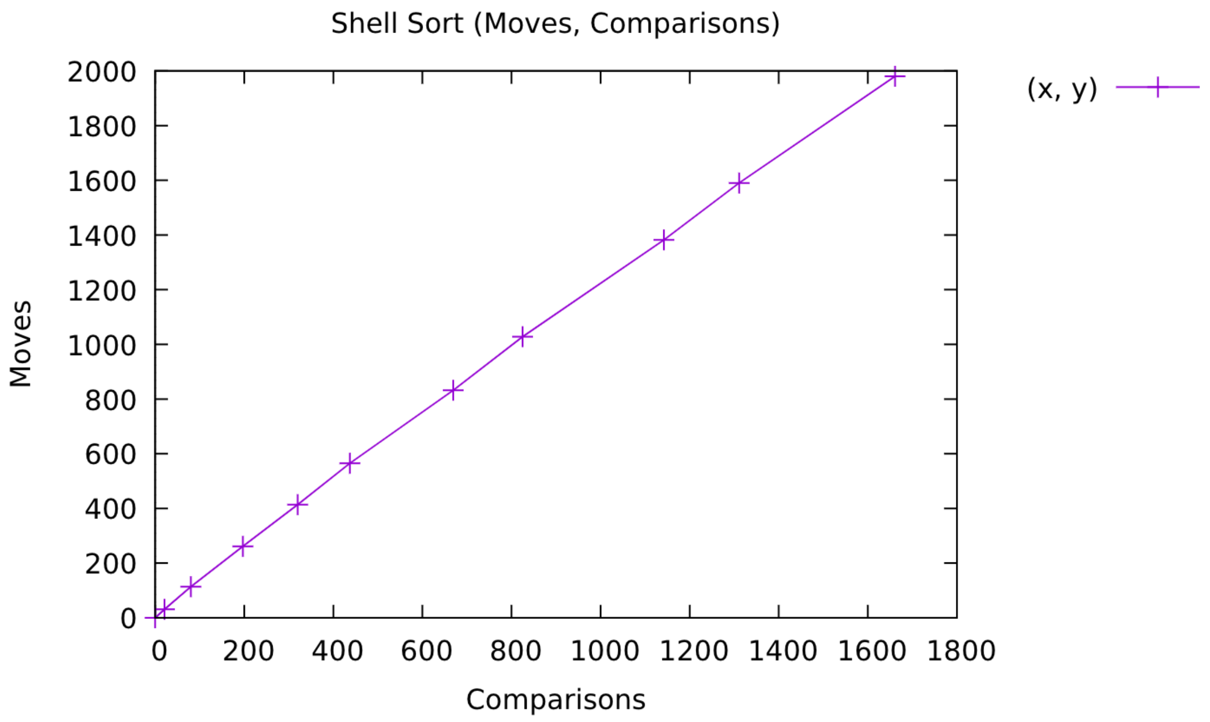
Introduction

Sorting numbers is a central concept in Computer Science, and can be performed about in a number of ways. Although the various methods end in the same result of a sorted array of numbers, these algorithms differ in what we call time complexity. Time complexity is defined as the computational complexity of an algorithm, denoting how much time an algorithm takes to run. Today we will study four sorting methods, each with differing time complexities, advantages and disadvantages.

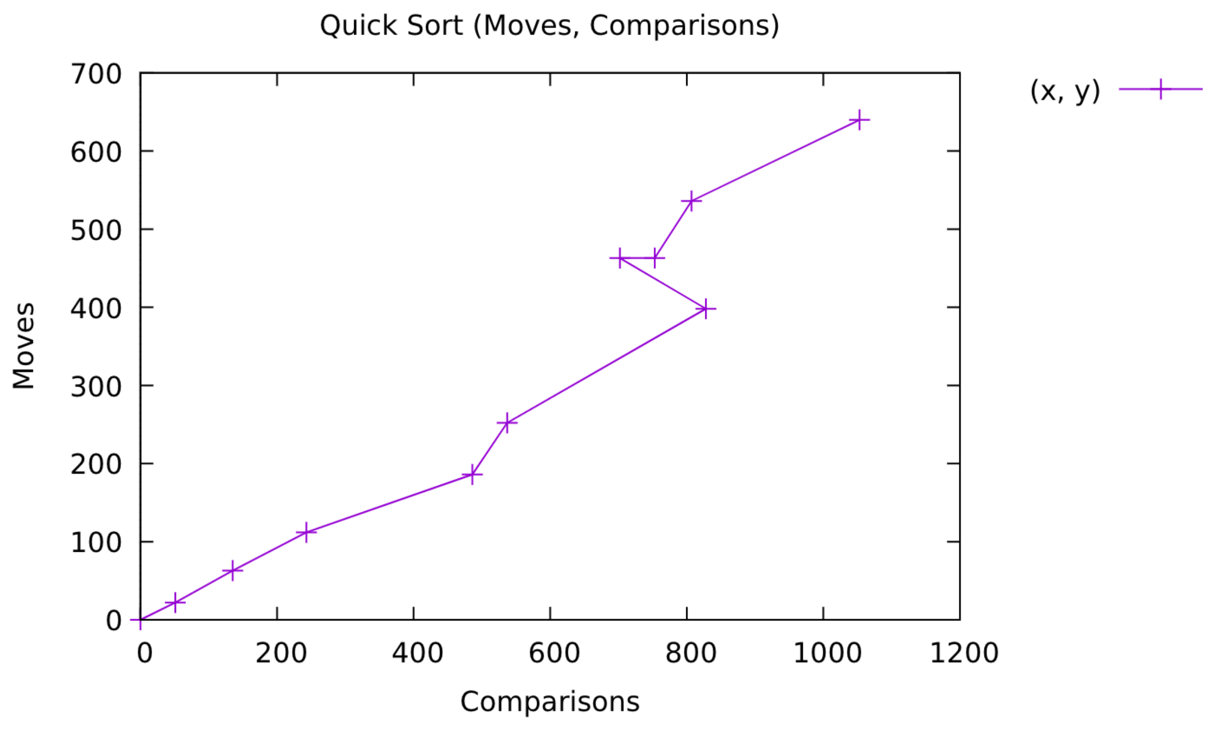
Discussion



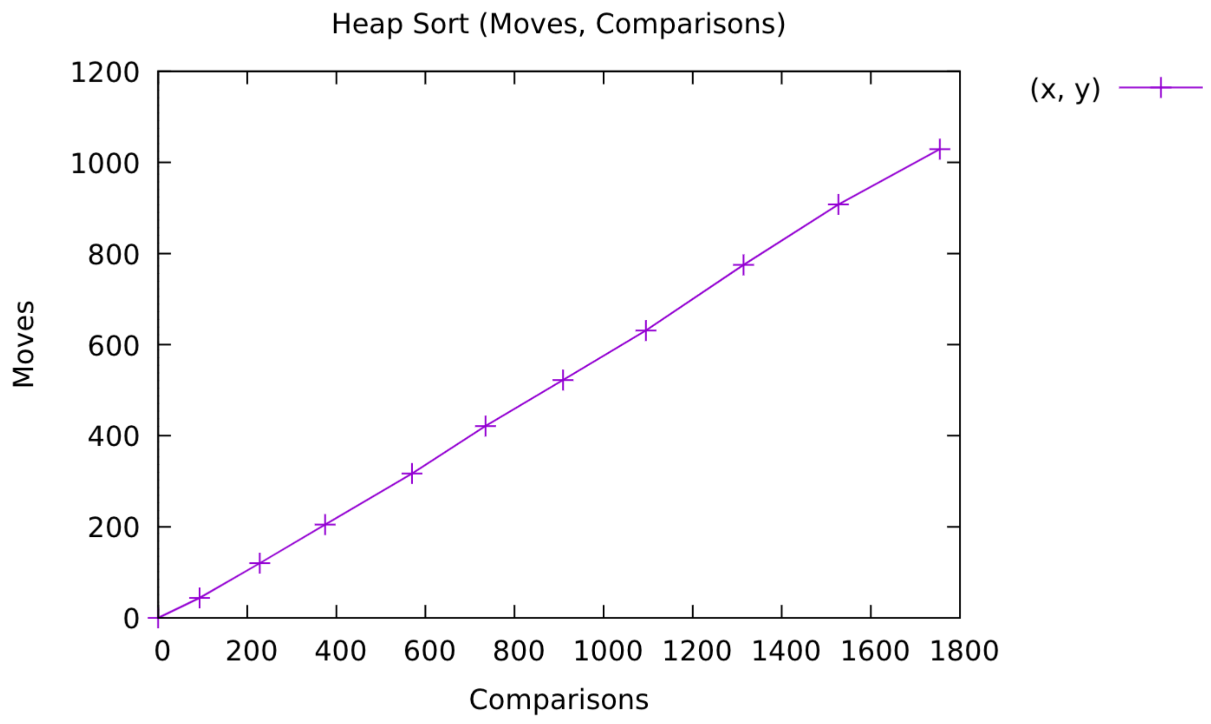
The first sort we will look at is the most basic of the four: insertion sort. Insertion sort involves comparing elements one at a time, checking the current value with that of the preceding elements. If the algorithm finds that the element at the current array index is smaller than that of the directly preceding element, the preceding element will be moved to the current array index. The original element that was in the current array index is then compared to the next preceding element that was right before the first element it was compared to. This algorithm is very straightforward with no room for fluctuations in performance. The algorithm involves two nested loops, meaning that it will take no longer than n * n to sort the given array, offering an average and worse case time complexity of O(n^2). Under a circumstance of a very small array that may already be sorted, the best case is O(n).



The next sort we will analyze is shell sort, a variation of insertion sort. This algorithm involves comparing pairs of elements that are a gap of size *x* away. Each iteration of the algorithm will decrease the gap between compared elements until a gap of 1 is reached, as at this point the array will have been sorted. This sorting method does involve far less moves and comparisons than that of its relative insertion sort, but will find weakness in arrays with too many or too few gaps. This algorithm has a ranging time complexity, greatly dependent on the array that is passed into it. It has a worst case of O(n $log^2 n$), where the gap sequence is very large, and a best case if O(n$log\ n$), where the array is already sorted, and an average case of O(n).



Another sorting algorithm we will discuss is quick sort. It is generally described to be one of the fastest sorting algorithms. However, it has the potential to be just as slow as it is fast depending on the array passed. This sort involves partitioning an array into two sub arrays in which a pivot between the two is chose. Elements found to be less than the pivot are put into the left array, and elements larger than or equal to partition are put in the right array. The algorithm has a best case time complexity of O(n$log\ n$), where the partitions on each side of the array are balanced, and a worse case of O(n^2).



The final sort we will study today is that of heap sort. This algorithm makes use of structures called heaps, which are synonymous to that of binary trees. The sort uses max heap specifically, where up to two child nodes must be less than or equal to their shared parent node. First a heap is built in max heap fashion, with the largest values coming to the beginning of the array. Then, the heap is fixed, being sorted by removing the largest elements of the root of the heap, and placing them at the end of the array. This algorithm yields a time complexity of O(n$log\ n$) for any case, making it a very stable predictable algorithm.

Conclusion

______Of the algorithms studied today, we can see which are most efficient, and which are not so much. We can get direct comparisons of performance by looking at each sorting method's time complexity. We see that quick sort, as denoted by its name is the quickest sorting algorithm of the bunch with its best case of O(n$log\ n$). However we also saw that it has equal potential to be of the slowest with the worst case being O(n^2). Next we have heap sort with an equivalent time complexity of O(n$log\ n$). This raises the question of why quick sort is considered faster despite heap sort having the same best case time complexity as quick, even in its worst case. This is because of the speed in which quick sort's inner loop runs. Third fastest, we have shell sort with its average case of O(n), and lastly insertion sort of complexity O(n^2).