Comparison-based Sorting Algorithms

M. Brian Curlee

February 2022

1 Introduction

Algorithms have for many years provided the structure that becomes the bases for many applications in computing. Simply put, an algorithm is nothing more than set of prescribed steps for solving a specific problem in a finite amount of time. Many common algorithms take a set of data elements as input and output a the same set of elements in the desired sorted order. This project implements several different sorting algorithms, and tests them on several sets of data with varying size. The Time complexity of each algorithm is then compared to shed some light on which may be suited for certain cases.

1.1 Time Complexity

Algorithms can be rated for their use of time by considering how increasing the data size (n) will increase the overall time to completion. By simply examining the pseudocode of an algorithm, or the steps of primitive operations written in more standard English, time complexity can described as a function of input size. For example, in simply looping through an array of size n will result in linear complexity, where looping through an array and comparing each value with every other would result in a quadratic complexity. An algorithm with quadratic complexity will take much more time as the value for n increases, versus that of a linear algorithm. These complexities are not affected by the hardware on which algorithms run, thus creating a method for providing the most efficiency. It should also be noted that complexity is a property assigned by worse case scenario.

In the following sections each of the implemented algorithms will be briefly described. Afterwards the implementation and test results for this project will be discussed.

2 The Algorithms

2.1 Insertion Sort

Is a simple sorting algorithm that removes values from an unsorted list sequentially and arranges each one at a time. As mentioned in the lecture, this process can be thought of as removing cards from a deck in your right hand and placing them in order in the left. This process is implemented by swapping the cards on the values on the left until the new value sits between one that is less and one that is greater (if said value is not the greatest). This process continues until all values have been evaluated.

In the best case the unsorted deck was already in order, and in this case Quick Sort runs linearly. The worst case, and thereby the time complexity of Quick Sort is when the entire array is in reverse order, thus becoming quadratic in complexity.

2.2 Merge Sort

Merge Sort is another simple algorithm that uses divide and conquer paradigm. First the array is divided into smaller parts. Each part is either recursively or iteratively divided again and each part is solved. Finally, each part is merged back together in much the same order that it was split.

For Merge Sort, as the full array is split where by each value will only be compared with a partial set resulting in a logarithmic complexity. Specifically n log(n).

2.3 Heap Sort

First, a **Heap** is a binary tree that stores keys at internal nodes. Each key stored is greater than or equal to that of its parent. In a complete tree, meaning that for each level (i) there are at most 2^i nodes, all internal nodes are to the left of external.

In the Heap Sort algorithm, we have another case where elements are split in a divide and conquer paradigm. An element is inserted at the bottom of the tree and recursively moved up the tree until its parent has a smaller value, and children a larger. Heap Sort has a worst case complexity of n log(n). Here again inserting and heapifying will only ever work over a portion of the heap.

2.4 Quick Sort

Quick Sort is another divide and conquer algorithm, with one random element being established as the pivot. Each element is ether less than, equal to, or greater than the pivot. As suggested, elements are pivoted around our pivot

element to create partitions in relation. This process continues recursively, with pivots being drawn from each partition until no elements remain. Each partition is then reassembled resulting in a sorted array.

Again here, as the original array is partitioned, our best time is n log(n). However in the worst case, and thus overall complexity, the pivot chosen is either the maximum, or minimum in the array resulting in a quadratic complexity.

2.5 Modified Quick Sort

Also known as the median of three, this modified quick search selects the first, last and middle items in the array. These elements are then swapped to be in the correct order with lowest on the left, greatest on the right. The pivot, or median is then swapped to the end of the array at length - 1. This process results in hopefully only swapping half the array.

Modified Quick Sort theoretically has complexity of n log(n).

3 The Test

For my implementation, I chose to use Python as my implementation language. I chose values ranging from 10 - 10000 and implemented each of the algorithms recursively. I did not realize that Python has a recursion depth limit of 1000. This means that with multiple arrays of varying size and tests including several algorithms, I ran into the limit quickly in my first several runs. In consideration of the submission date and the time I had already invested, I decided to implement iterative versions of the algorithms. Upon examination, complexity was essentially the same, however much less sleek in implementation.

I created my project in Google Collab, as I envisioned that this format would be pleasant and well organized for submission. I found, however, that with larger arrays Collab would stall for several minutes. I pulled the notebook and ran it locally, but found that raw code ran substantially more quickly. The TLDR of this experience is that I have created several versions of my project.

After implementation, I decided to test each algorithm on each array five times, capturing the time in seconds of each run, and averaging the times for each algorithm. Tests were completed for several size array and quantitative data was visualized as seen below.

The arrays for my tests were randomized, with the range matching the size of the array. The first test was on the unsorted arrays. The second, on sorted arrays and the third on arrays sorted in reverse order.

3.1 Results

My first set of iterations, in the Fig 1 - 3, I used values ranging from 10 - 20000. With such a wide range of values it was hard to assume any true patterns. In Fig 4 - 6, I reran the tests on values ranging from 10 -1000. These images show a more loosely related spread of complexity. Still much as would be expected.

Figure 1: Unsorted Arrays with Max 20000

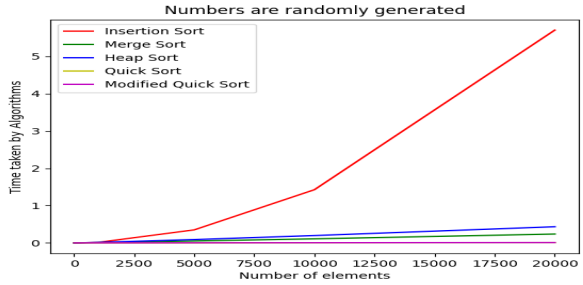


Figure 2: Sorted Arrays with Max 20000

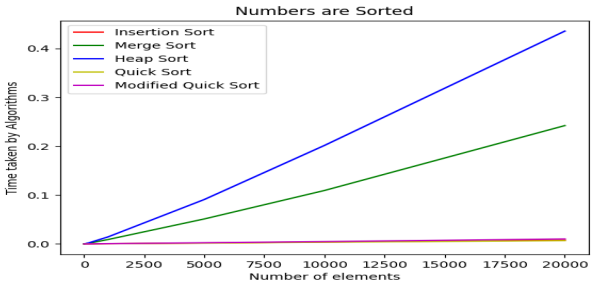
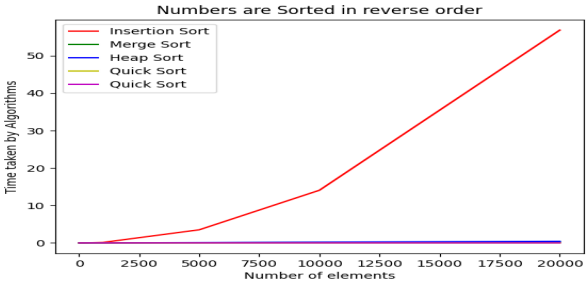


Figure 3: Reversed Arrays with Max 20000



These next images show the more compact range of 10 - 1000. The trajectories are as expected and described within the prior explanations.

Figure 4: Unsorted Arrays with Max 1000

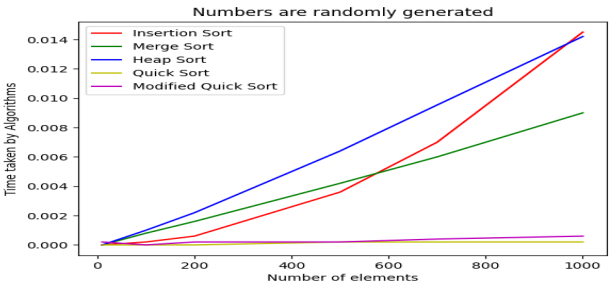


Figure 5: Sorted Arrays with Max 1000

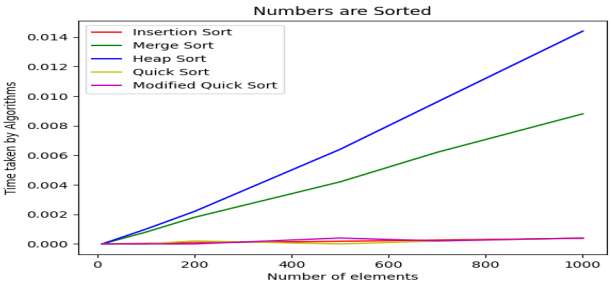
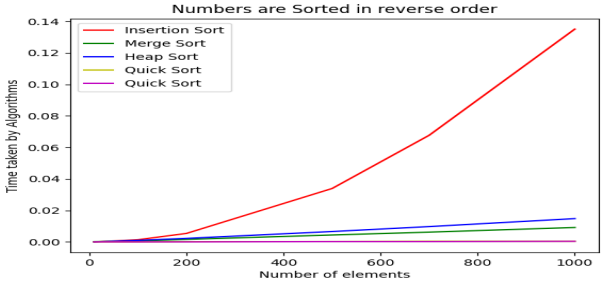


Figure 6: Reversed Arrays with Max 1000



4 Conclusion

After testing and learning a ton, about both these algorithms and Python. It should be noted that certain algorithms and implementations are better optimized for certain cases. Python may not be the best adapted for recursive sorting algorithms, but part of the journey is the steps we make on the way. Insertion sort is going to do the best for those sets of data that are small and nearly sorted. More robust algorithms like Heap Sort will fare much better in larger and more loosely sorted data.

4.1 Project Repositories

GitHub: https://github.com/mcurlee3/AlgorithmComparison.git

Collab: $https://colab.research.google.com/drive/1gLhgqwIxJO_fTk8XkPFq4dmjUsbOimmL?usp=sharing$