Part-1

The Theoretical Comparison of Sorting Algorithms.

For each type of sorting algorithm, there are multiple time complexities, whether it's the best case, worst case or average case. Due to this, it means that for six different sorting algorithms there are at least 18 different comparisons that can be made and that's before we look at specific methods for each algorithm which in some cases has its own time complexity. A prime example of this is quicksort as when the pivot changes so do the time complexity of the algorithm.

The first algorithm being looked at is insertion sort, this is where the elements are inserted in place until the data set is sorted. The best-case time complexity for this type of sort is O(n) and this occurs when the data set is already sorted as the sort is a linear sort. The worst-case time complexity for this sort is O(n^2) and this occurs if the data set is inverted and each set of data needs to be rearranged into its correct place. And finally, the average-case for Insertion Sort is also O(n^2) and this is due to most data set not being pre-sorted and therefore the time-complexity is the same as the worst case. This scenario happens when the data set is partly sorted and as a result still requires some sorting.

The second algorithm is selection sort, it works by constantly looking for the smallest element until there are no elements remaining and the data set is sorted. The best-case for selection sort is O(n^2) and this occurs if the data set is already sorted. The worst-case for selection sort is also O(n^2), this is due to the way that the sort works as it always looks for the smallest value so theoretically the best case and worst case will have the same number of operations if the size of the data set is the same. And lastly the average-case for selection sort is also O(n^2) as it still has the same number of operations as the best and worst case. The one thing to point out about selection sort is that as the data set increases in length the efficiency of the algorithm drops and the number of operations increase.

The third algorithm is bubble sort, it works by swapping out the adjacent element until the data set is sorted. As a result, the best case for bubble sort is O(n) and this occurs when the data set is pre-sorted, so the number of operations is the length of the data set as they check to see if the element to the right is smaller than the one to its left. The worst case for bubble sort is O(n^2) and this is because the number of times the algorithm must go over the data set, as one "pass" of the data set if the data is sufficiently out of place won't be enough to sort it correctly. And since this can happen n times the time complexity will be n times n which in n^2. The average case for bubble sort is also O(n^2) as the number of passes to sort the data set is more than one which gives the n^2 time complexity.

The fourth algorithm is merge sort, which is a divide and conquer algorithm. This means that the data set is divided into two and then sorted and merged back together. The best-case for merge sort is O(nlogn) as the data set will be divided and compared no matter what the data set looks like. As a result of this the worst case is also O(nlogn) and the average case is this too as the data set will be divided and compared.

The fifth algorithm is quick sort, this algorithm works a little differently than the others in this report. Whilst being a divide and conquer algorithm quicksort uses a pivot point to decide where the divide in the data set will be. Due to this the time complexity of the algorithm depends on a few things, one mainly being the pivot point and the type of pivot as different pivots have different time complexities which causes the algorithm to have different time complexities too. But in the case of "standard" quicksort the best-case time complexity is O(nlogn) this occurs if the selected pivot is in the centre or middle of the data set. The worst case time complexity happens when either the first or last element is chosen for the pivot point and the data is either already sorted in the case for the last element pivot point or is inverted for the case of the first element pivot point and this gives the time complexity of O(n^2). Another thing to consider is the random pivot which gives the time complexity of O(nlogn) for the worst case however this type of pivot is classed as a Monte Carlo algorithm which is the classification for randomized algorithms. The randomized quicksort is another type of the quicksort algorithm. And finally, the average case for quicksort is also O(nlogn) and this is due to the division and comparison nature of the algorithm.

The sixth algorithm is heap sort, it works by finding the largest element and placing it in the last place of the data structure and then working backwards. In a way it is inverted selection sort. As a result of this the time complexity is the same across all three scenarios for best worst and average case and has a complexity of O(nlogn). This is due to the algorithm having the same number of operations no matter the order of the data set but as long as the length of the data is constant.

The Generation of Data for Experimental Setup.

The PC we will be using is a Samsung Laptop. It has an Intel Core i7-4500U CPU @ 1.8 GHZ. It has 8 gb of DDR3 ram. For implementation of our timing mechanism, we used a chrono library. In the data sets that were created, all possible cases for each and every sorting algorithm were taken into account. This includes the data sets that range from 10 – 5000. This was incorporated for examination purposes. It helps because it provides differing values that exponentially increases. We can plot the data in an exponential growth curve and can do a side-by-side comparison of each algorithm.

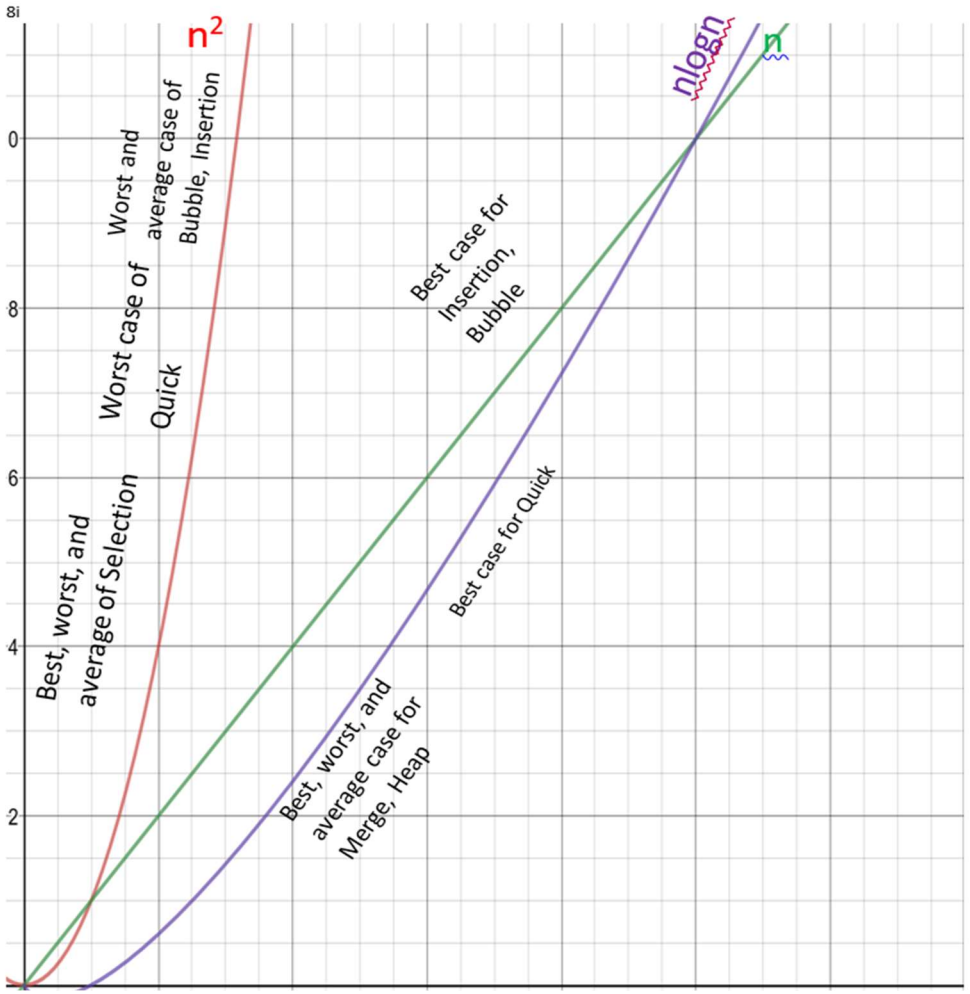
The positions of each value changes with each instance of creation. The data that was implemented has a random set, a reverse set, and also an ordered set. The reverse set was made to outline any cases that may have fallen between the cracks. The ordered set is to see which sorting algorithm ends up performing the slowest in an ordered state (Quick Sort). The random set is incorporated to include best cases for some algorithms and worst cases for others. So, there are different scenarios being considered in our examination of the sorting algorithms. The random sets can be computed 9 different times – creating a different data set each time. This helps to make the

median value clear. It also must be an odd number to help draw the picture better. So the inputs are chosen at random besides a few unique instances with the sorted and the reverse set.

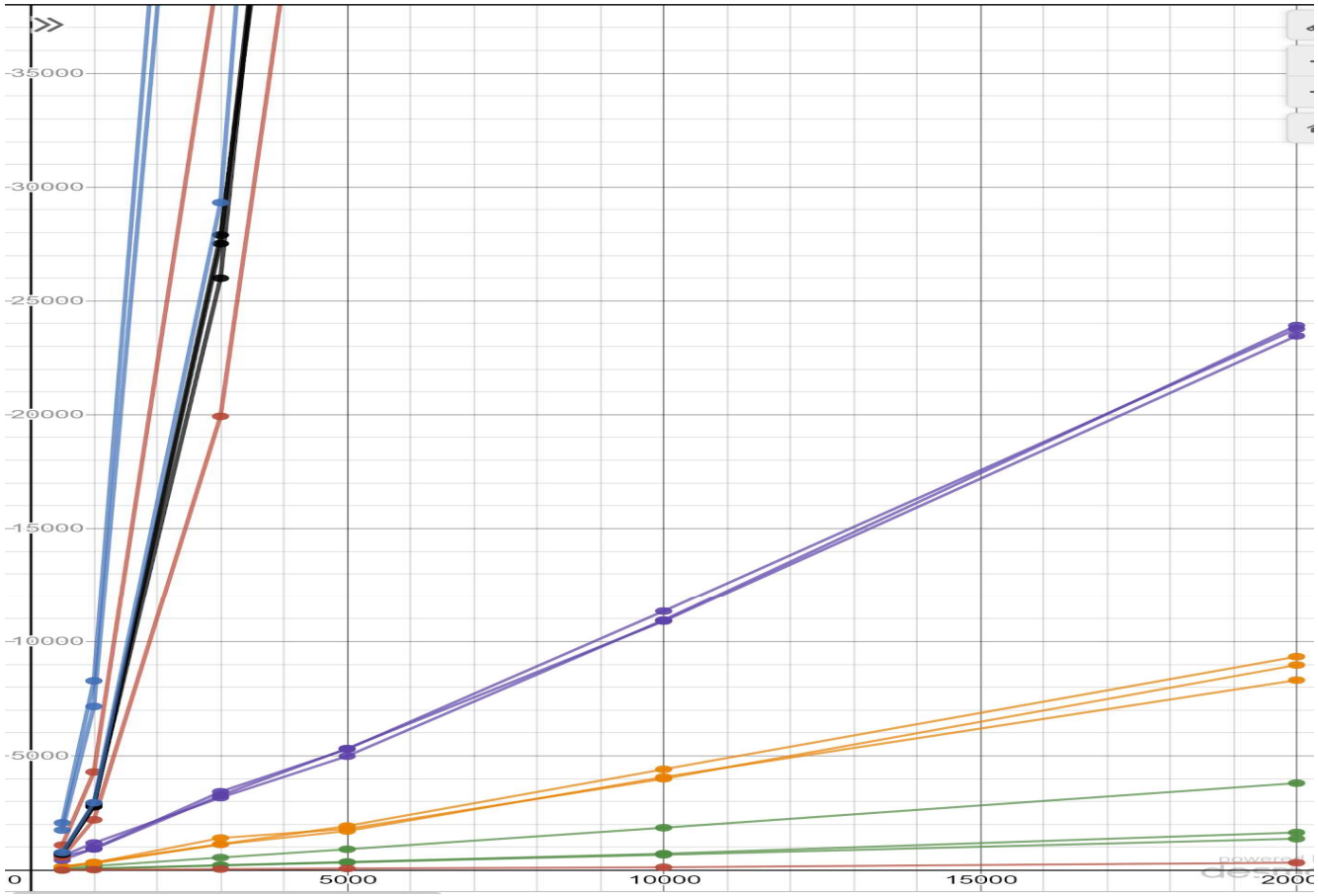
The aim was to include as many differing scenarios as possible. The experiment will be run across the different types of data sets and will be generated according to a selection of data sizes. For the reverse and sorted sets, we are running it N times based on how many different sizes are available. Then we are creating 9 different data sets of the random data set with one size N. We will increment the size and repeat the same process until the median is clear.

Experimental Analysis of the Variations of the Sorting Algorithms.

Theoretical Graph



Experimental graph



X-axis Data set sizes
Y-axis Time taken by each sort

Green-Quick sort
Purple-Merge sort
Black-Selection sort
Red-Insertion sort
Orange-Heap Sort

Blue-Bubble Sort

Comparison between the theoretical analysis and the experimental analysis of the algorithms.

Looking at the graphs above we can see that:

Bubble sort-

Theoretically:

Average and worst-case time complexity: n^2

Best case time complexity: n when array is already sorted.

Worst case: when the array is reverse sorted.

Experimentally:

Bubble Sort *can* terminate early -- if we break because a sweep didn't result in any two elements being swapped, the function returns faster. We know that Bubble Sort will not run for more than n sweeps (where n = len(L)), just because the outer loop will not run for more than n iterations.

Insertion sort –

Theoretically:

Average and worst-case time complexity: O(n^2)

Best case time complexity: n when array is already sorted.

Worst case: when the array is in reverse order.

Experimentally:

Insertion Sort is good only for sorting small arrays (usually less than 100 items). In fact, the smaller the array, the faster insertion sort is compared to any other sorting algorithm. However, being an O(n^2) algorithm, it becomes very slow very quick when the size of the array increases. It was used in the tests with arrays of size 100.

Selection sort –

Theoretically:

Best, average and worst-case time complexity: n^2 which is independent of distribution of data.

Experimentally:

Since the selection sort is independent of distribution of data it gets very slow with the size of the array. The more the data more the running time. Selection sort is an unstable sort i.e. it might change the occurrence of two similar elements in the list while sorting.

Merge sort –

Theoretically:

Best, average and worst case time complexity: nlogn which is independent of distribution of data.

Experimentally:

The virtue of merge sort is that it's a truly O(*n*log*n*) algorithm (like heap sort) and that it's stable (iow. it doesn't change the order of equal items like e.g. heap sort often does). Its main problem is that it requires a second array with the same size as the array to be sorted, thus doubling the memory requirements.

Heap sort –

Theoretically**:**

Best, average and worst-case time complexity: nlogn which is independent of distribution of data.

Experimentally:

Heap Sort is the other (and by far the most popular) *in-place* non-recursive sorting algorithm used in this test. Heap sort is not the fastest possible in all (or most) cases, but it's the *de-facto* sorting algorithm when one wants to make *sure* that the sorting will not take longer than O(*n*log*n*). Heap sort is generally appreciated because it is trustworthy: There aren't any "pathological" cases which would cause it to be unacceptably slow. Besides that, sorting in-place and in a non-recursive way also makes sure that it will not take extra memory, which is often a nice feature.

Quick sort –

Theoretically:

It is a divide and conquer approach with recurrence relation:

T(n) = T(k) + T(n-k-1) + cn

Worst case: when the array is sorted or reverse sorted, the partition algorithm divides the array in two subarrays with 0 and n-1 elements. Therefore,

T(n) = T(0) + T(n-1) + cn

Solving this we get, T(n) = O(n^2)

Best case and Average case: On an average, the partition algorithm divides the array into two subarrays with equal size. Therefore,

T(n) = 2T(n/2) + cn

Solving this we get, T(n) = O(nlogn)

Experimentally:

Quicksort is the most popular sorting algorithm. Its virtue is that it sorts *in-place* (even though it's a recursive algorithms) and that it is usually very fast. The reason for its speed is that its inner loop is very short and can be optimized very well.

The main problem with quicksort is that it's not trustworthy: Its worse-case scenario is O(n^2) (in the worst case it's as slow, if not even a bit slower than insertion sort) and the pathological cases tend to appear too unexpectedly and without warning, even if an optimized version of quicksort is used.

We used two versions of quicksort in this project: A plain vanilla quicksort, implemented as the most basic algorithm descriptions tell, and an optimized version. The optimized version chooses the median of the first, last and middle items of the partition as its pivot.

Sorting Algorithm	Best Case	Average Case	Worst Case	Stable	In-Place
Bubble Sort	**O(n)**	**O(n^2)**	**O(n^2)**	**Yes**	**Yes**
Heap Sort	**O(nlogn)**	**O(nlogn)**	**O(nlogn)**	**No**	**Yes**
Insertion Sort	**O(n)**	**O(n^2)**	**O(n^2)**	**Yes**	**Yes**
Merge Sort	**O(nlogn)**	**O(nlogn)**	**O(nlogn)**	**Yes**	**Yes**
Quick Sort	**O(nlogn)**	**O(nlogn)**	**O(n^2)**	**No**	**Yes**
Selection Sort	**O(n^2)**	**O(n^2)**	**O(n^2)**	**No**	**Yes**

Determining whether the number of comparisons is an accurate method of analysing the algorithms.

In all likelihood, **comparisons is roughly correlated with execution time but cannot accurately predict the execution time.**

Firstly, comparison and assignment are two factors for predicting the execution time for Bubble Sort, Selection Sort and Insertion Sort. Although all of these sorting algorithms have the same time complexity when the bias is eliminated from the test data, the order of execution time always be Insertion Sort < Selection Sort < Bubble Sort (from fast to slow). In other words, Insertion Sort is fastest while Bubble Sort is slowest. Why does it happen?

Insertion Sort –

Insertion sort may skip some comparisons in the inner loop when the current number is greater or equal than the previous number. Otherwise, assign the current value to the successor number(A[i+1] = A[i]). Therefore, Insertion Sort has the minimum times of comparisons and assignments, it is the fastest method in O(n^2) methods.

Bubble Sort –

Bubble Sort cannot skip any comparison. Also, when it needs swap two elements, there should be three assignments (temp = a, a = b, b = temp), that's why Bubble Sort become the slowest method.

Selection Sort –

Selection Sort cannot skip any comparison either. But for each inner loop, it only swaps current element with the smallest element once. Since the comparisons and assignments for Selection Sort is between Insertion Sort and Bubble Sort, the execution time is also in the middle.

Secondly, comparison and the characteristics of each algorithm are two factors for predicting the execution time for Merge Sort, Quick Sort and Heap Sort. Although all of these sorting algorithms have the same time complexity when bias is eliminated from the test data, the order of execution time always be Quick Sort < Heap Sort < Merge Sort. In other words, Quick Sort is fastest while Merge Sort is slowest.

Quick Sort –

Quick Sort has similar comparison times as Merge Sort, but Quick sort is in-*place* sorting method, it doesn't need extra space like Merge Sort does. Therefore, Quick Sort is the fastest method in O(nlogn) methods.

Merge Sort –

Merge sort need extra space to hold the sorted elements, even though the other characteristics are the same as Merge Sort, It is more slower than Quick Sort.

Heap Sort –
Heap Sort has the most comparisons in these three methods. In order to choose the biggest value in a subtree, it should compare left child with right child, then compare the bigger one with parents. While the advantage of Heap Sort is that it is not a recursion method, so it avoids the cost to maintain a stack. Heap Sort is an in-space solution either, Therefore, it is faster than Merge Sort and slower than Quick Sort.