**Computational Thinking with Algorithms Project**

**By Kevin Dooley**

**Introduction**

**What is sorting?**

Sorting is the process of arranging a collection of elements according to some pre-defined ordering rules. A human can intuitively perform the task of sorting but a computer program must follow an exact set of instructions in order to accomplish the same. This sequence of instructions is called an algorithm. Sorting was a fundamental algorithmic problem that was faced in the early days on computing. In fact, most of the computer science research was centred on finding the best way to sort a set of data. Sorting is a term/area that has been widely studied in Computer science since 1945 when the first sorting algorithm was invented by John von Neumann in 1945 [1]

Sorting algorithms can be used on their own or in conjunction with other programs to simplify numerous computational tasks such as searching for an exact item in a list, identifying duplicate items in a list and finding the frequency of each distinct item. [1]

There many different types of sorting algorithms which are split into two categories: comparison based and non - comparison based algorithms, both of which will be discussed in this project.

**Complexity**

As will be seen in this project, some algorithms are more efficient than others. That will become very evident when comparing the different types of sorting algorithms. The complexity of an algorithm is a function describing the efficiency of the algorithm in terms of the amount of data the algorithm must process. There are two main complexity measures of the efficiency of an algorithm:

1. Time Complexity: Describes the amount of time an algorithm takes in terms of the amount of input to the algorithm.
2. Space Complexity: Describes the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm.

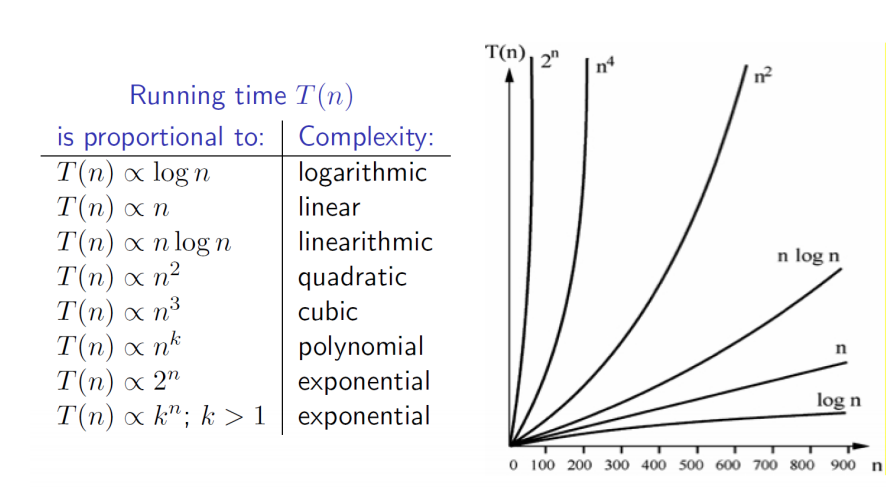
**Analyse Complexity**

Algorithms are platform dependent, therefore comparing complexity via empirical comparison is of little use because the results are dependent on the specific platform which was used to run the algorithm. Therefore, the best option is to analyse complexity mathematically. The standard approach to compare algorithms is to evaluate running time complexity on input data size n. By using this method, you can determine which algorithm scales better than others and which is the most efficient. (Analysing Algorithms Part 1, 2020)

Algorithms are often quite different from one another, though the objective of these algorithms are the same. Analysis of an algorithm is the process of analysing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation). As mentioned above, the main concern of analysis of algorithms is the required time or performance. Analysis of algorithms is performed under 3 main categories:

1. Worst-case − The maximum number of steps taken on any instance of size n. Worst Case behaviour is quite easy to analyse and explains how slow the program could be in any situation. Referred to as Big O Notation.
2. Best-case − the minimum number of steps taken on any instance of size n.
3. Average case − an average number of steps taken on any instance of size n.

The running efficiency of an algorithm can be categorised into the below classifications. Logarithmic are the most efficient and exponential the least. Some of the sorting algorithms that will be discussed below are quite inefficient and cannot improve on a worst case notation of 𝑛2. (Analysing Algorithms Part 1, 2020)

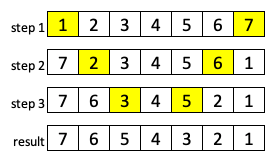
**Image 1: Overview of Complexity**

(Analysing Algorithms Part 1, 2020)

##### **In-Place Sorting**

In-place algorithms play an important role, because they maximize the size of data that can be processed in the main memory without input/output operations. [Hanan Ahmed-Hosni Mahmoud, and Nadia Al-Ghreimil, 201] A sorting algorithm is called in-place if it uses only a fixed additional amount of working space, independent of the input size. In-place algorithms may require a small amount of additional memory but the memory is not dependent on the input size. [In place sorting] In-place sorting can be a very desirable property if memory availability is limited. Several of the sorting algorithms sort in-place such as Bubble sort, insertion sort and selection sort.

**Image 2: Example of in place sorting reverse algorithm.**



Source: (<https://www.baeldung.com/java-in-place-sorting>)

**Stable Sorting**

A sorting algorithm is considered stable if two elements of equal value appear in the same order in its sorted output as they appear in the input array to be sorted. [17] Basically, if there is a list with an 8 at the start of the list and an 8 in the middle of an unsorted list. The 8 at the start of the list will remain in the position in front of the 8 in the middle of the list in the sorted array. Bubble Sort is a prime example of a sample sorting algorithm. Unstable sorting is then the opposite. It may not maintain the position of two equals in the sorted list.

**Comparison and Non-Comparison based Sorting**

A sorting algorithm is comparison based if it uses comparison operators to find the order between two numbers. It goes through the array, compares two elements in adjacent positions and swaps there position if in an unordered state. Examples of comparison based sorts include Bubble Sort, Insertion Sort and Selection Sort. [Lecture Notes]

It is also possible to sort items in a list with comparing the items with each other. They do this by making certain assumptions about the elements they are sorting rather that comparing each element to one another. [18]

**Sorting Algorithms**

**Bubble Sort**

The bubble sort is a simple sorting algorithm that repeatedly steps through a list, compares adjacent elements (from left to right) and swaps their positions if they are in the wrong order. [1] It does this through the entire list until it cannot find any two elements that are required to be swapped and the list is completely sorted. It is a comparison based sort due to the way it compares the element adjacent to it and determines if they need to be swapped or not. It also uses in place sorting therefore uses a constant amount of additional working space in addition to the memory required for the input. [2]

It was first used by Iverson in 1962 but it has been analysed from as early as 1956. It received its name because the biggest elements in the list always ‘bubble’ to the end or top of the list [2].

As mentioned above, a comparison based algorithm complexity cannot do no better than n log n performance in the average or worst cases. Bubble sort is a comparison-based algorithm so that applies analysis will apply when reviewing benchmarking results.

The bubble sort is very impractical for large datasets and has an average and worst complexity of Ο(n2). It can achieve n performance in the Best Case but that is unlikely to ever be achieved using random pseudocode as per this project. It is much more likely that worst or average case will be achieve where it will compare every element in the list with every other one. This will result in Ο(n2) or exponential to the size of the input.

**How Bubble Sort Works**

Irrespective of the nature of input, the number of passes to be made is n – 1. [4] By the end of every pass, at least one element is placed in its right position.

As per figure below, input [a] has 9 elements in unsorted positions.

**Step 1**: Pass 1, compare 54 and 26. 54 is greater than 26 so the elements are swapped.

**Step 2:** Pass 1, compare 54 and 93. 54 is smaller than 93 so the elements are not swapped

**Step 3:** Pass 1, compare 93 and 17. 93 is greater than 17 so the elements are swapped.

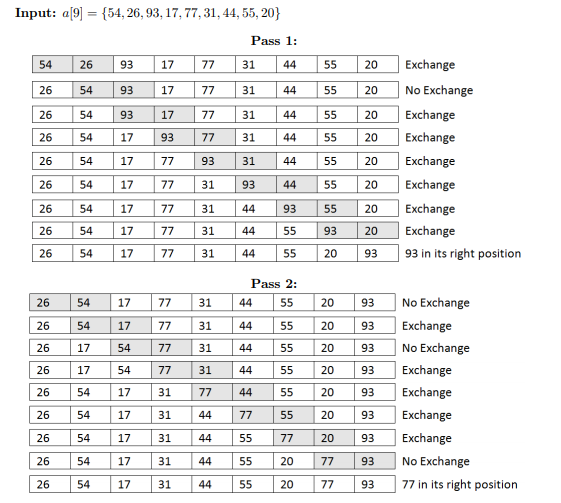
 Pass 1 continues in this process until the largest element is at the end, in this case its 93. Then, Pass 2 completes and so on until the list/array is sorted. There will be 8 passes in the example because there are 9 elements.

Diagram Source [4]

**Selection Sort**

Selection sort is an in place comparison based sorting algorithm which is simple to implement. It divides the list into two separate parts, the sorted list on the left and the unsorted list on the right. It is based on the process of finding the minimum and maximum element of an unsorted array and placing it in its correct position [5].

The algorithm starts at element 0 and works its way through until n-1. It selects the smallest element and puts that element in position 0 (sorted sub array). It then goes to element 1, sorts through n-1 to find the smallest element on the right side (in the unsorted sub array) and swaps it to position 1. There are now 2 elements in the sorted array on the left. The algorithm continues in this way until there is nothing left to search and sort. [2]

Selection sort, while performs better than the Bubble Sort, is still an impractical sorting algorithm for real world data. It can be beneficial on small datasets [2]. Time complexity of selection sort in best, worst and average case is n2. It does not matter the state in which the list is in if selection sort is used. If it is already sorted or in reverse order, O(n2) is returned. This quadratic because of the number of comparisons required to be completed. At every step, you have to find the minimum element and put it in the right place. The minimum element is not known until the end of the array is reached. [6]

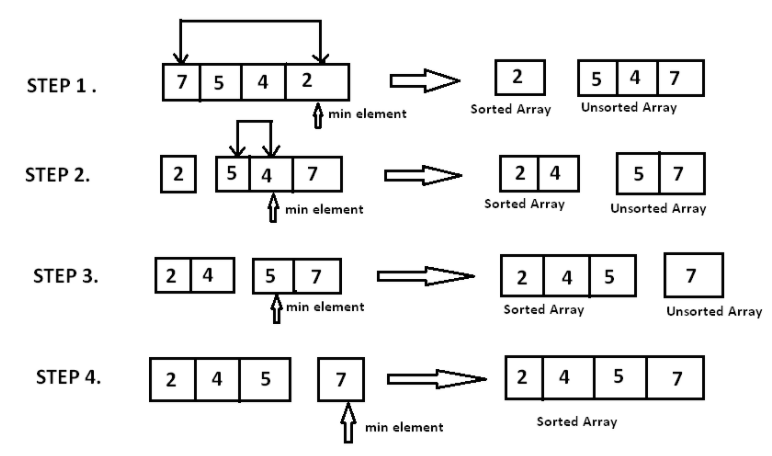
**How Selection Sort Works**

**Step 1:** The minimum value is identified, in this case 7. 2 is then swapped into position 0. There are now 2 sub arrays. A sorted sub array and an unsorted subarray.

**Step 2:** The minimum value in the unsorted sub array is identified as 4. It is swapped into position 1 in the sorted sub array

**Step 3:** 5 is identified as the minimum value in the unsorted subarray and moved into position 2 in the sorted array.

**Step 4:** 7 is the last number in the unsorted array and therefore the minimum value. It is swapped into position 3 of the sorted array. Sort complete.

Diagram Source: [5]

**Insertion Sort**

Insertion sort is a simple comparison based algorithm that builds the sorted list one item at a time. It works similar to the way a person would sort playing cards in your hand. That is because if you have 5 cards in your hand that are in order and you pick up another card, you go through the deck 1 by 1 comparing the new card to the cards you already have in order and you insert it into the correct position [7][8]. The Insertion sort algorithm follows a similar process.

Insertion sort can be easy to implement, it is stable, follows in place sorting and can be effective and efficient on small lists or lists already nearly sorted. However, similar to bubble sort and selection sort, it is not inefficient on large random datasets [2].

It is an iterative sorting algorithm and works by splitting a list of input size n into sorted and unsorted sub lists (similar to selection sort). The algorithm starts at the left of the unsorted array and sets a key at index 1. The key value then compares itself to the element to its left. If the key value is greater than the element to its left, nothing swaps but if the element to its left is greater, then the element moves right by one position. Set the key at index 2. This keeps iterating for index n-1 until the list is sorted.

Insertion sort can achieve O(n), linear, in the best case. That is because, if it receives an already sorted list, it will still execute the outer for loop, thereby requiring n steps to sort an already sorted array/list of n elements, which makes its O(n) complexity. [9]. Average and worst case for insertion sort is O(n2). The algorithm takes an element to compare with every other element: n x n comparisons, quadratic.

**How Insertion Sort Works**

Array = [9, 6, 4, 1]

**Step 1:** Nothing to compare to the left of index 0, therefore add key value to index 1

|  |  |  |  |
| --- | --- | --- | --- |
| 9 | 6 (key) | 4 | 1 |

**Step 2:** Compare Key (6) with element to its left (9). 9 > 6, therefore they swap position. 5 is now the key as it gets assigned to the next element to the right.

|  |  |  |  |
| --- | --- | --- | --- |
| 9 | 6 (key) | 4 | 1 |

|  |  |  |  |
| --- | --- | --- | --- |
| 6 | 9 | 4 (key) | 1 |

**Step 3:** 4 is now the key. Compare 4 to 9. 9 > 4, swap position. Compare 4 to 6. 6 > 4. 4 is now in index 0 and 1 is the key value

|  |  |  |  |
| --- | --- | --- | --- |
| 4 | 6 | 9 | 1(key) |

|  |  |  |  |
| --- | --- | --- | --- |
| 6 | 9 | 4 (key) | 1 |

**Step 4:** 1 is now the key value. Compare 1 to 9, 9 > 1. Compare 1 to 6, 6 > 1. Compare 1 to 4, 4 > 1. All elements are now sorted in the correct order and the algorithm is complete.

|  |  |  |  |
| --- | --- | --- | --- |
| 4 | 6 | 9 | 1(key) |

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 4 | 6 | 9 |

**Quick Sort**

Quicksort is a recursive divide and conquer algorithm. The approach is based on choosing one element as the pivot value and then partitioning the array (into two sub arrays) around that pivot element. The left side of the pivot contains elements that are less than the pivot value and elements to the right are greater than the pivot value. [10]. The sub arrays are sorted recursively and is completed in place.

It was developed by Tony Hoare in 1959 and when implemented well can be very efficient and quick to sort as its name suggests. [11]. It is a comparison based sorting algorithm such as bubble, selection and insertion sort but it is much more efficient than any of those mentioned algorithms.

The partitioning process can determine the running time of Quick sort. In the best and average case, Quick sort will run in 𝑛 log 𝑛 but if a bad pivot is continually chosen, running time performance will be O(n2) in the worst case. Worst case can occur if too many of the elements are in the sub array < the pivot or in the sub array > pivot. [2]. There are many ways to choose the pivot: the 1st or last element in the array, the median element of the array or choose the pivot randomly. Worst case (quadratic) time would most only likely occur when the 1st or last elements are selected. By selecting the median element, there is a better chance of getting balanced sub arrays and therefore an average running time of O(n log n).

**How Quick Sort Works**

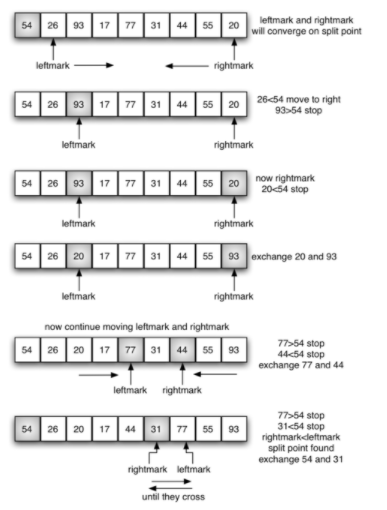
The pivot value taken in this example is the first element in the array 54.

Diagram Source [12]

**Step 1:** Pivot value identified as 54. The two position markers: left mark and right mark are identified. They are the most left and most right elements remaining in the list.

**Step 2:** Left mark is incremented right until a value greater than the pivot value is located. Then right mark is decremented until a value less than pivot value. This is evident between values, 93 (93 >54) and 20 (20 < 54). They are out of order so they can be swapped with each other. Then the left mark increments again.

**Step 3:** Neither 20 nor 17 are greater than the pivot value 54 therefore left mark increments until 77 which is greater than 54. The right mark starts to decrement because 55 is not less than 54. Right mark is not 44 which is less than 54. 77 and 44 can be swapped.

**Step 4:** Now right mark is less than left mark. The split point has been identified and the split point (rightmark) can be swapped with the pivot value. Now all the values less than 54 are to the left and all the values greater than 54 are to its right.

**Step 5:** The algorithm then recursively follows the above to sort the sub arrays until completely sorted.

**Counting Sort**

Counting sort is a non-comparison-based sorting algorithm that sorts the elements of an array by counting the number of instances of each unique element in the array [13]. It was proposed by Harold H. Seward in 1959. Counting sort is a very interesting sorting algorithm because it proposes an almost impossible task – to sort a list in linear time. To achieve this, assumptions on the elements in the list are required. For instance, if a list of n elements is required to be sorted (all non-negative) where each element is in the range [0,k] where k is the size of the helper array. [Lecture Notes] [14]

Counting sort, albeit efficient from a running time perspective does have its disadvantages in that it is not widely applicable as much as comparison-based sorting algorithms. That’s is because counting sort only works when the range of potential elements is known ahead of time. [15]

Counting Sort works by iterating through the input array, it counts the number of occurrences of each individual item and then using those counts, computes an items index in the final sorted array. To achieve this, the key range k must be determined. Create an array of size k to count and store the number of times each value occurs (iterate through). Create a final array to store the result values. It should be based on the frequency of each individual element [lecture notes]

Counting sort takes O(n + k) time complexity where n is the number items being sorted and k number of possible values. This is the case in best, average and worst case because no matter how the elements are placed in the array, the algorithm iterates at least n + k times.

**How Counting Sort Works:**

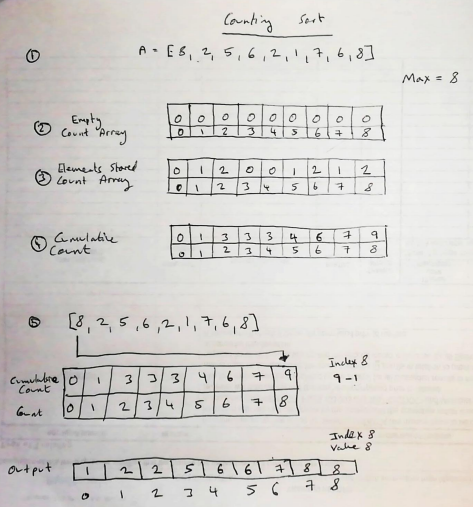
**Step 1:** Determine the max element value of the input array

**Step 2:** Create an array in length max + 1 with elements 0 assigned. This will be used for counting the number of elements in the input array

**Step 3:** Store each value and the number of times they occur in their respective position in the count array.

**Step 4:** Store the cumulative sum of the elements of the count array

**Step 5:** Find the index of each element from the input array in the count array. Use the cumulative count and decrement its value by one. This then gives the correct index of your input value in the output array (final sorted array). As per the example below: First element in input array is 8, go to its index in the count array (index 8). The cumulative value is 9. Take away 1 from 9 = 8. Then value 8 goes in position/index 8 in the sorted array. [4]



**Diagram Concept adapted from [16]**

**Further information on complexity of chosen sorting algorithms**

As discussed above and can be seen from table 1 below, all the simple comparison based sorting algorithms have an average and worst case time complexity of n2 (quadratic) and have a space complexity of 1. Bubble sort and insertion sort can achieve time complexity of n (linear) in the best case but the probability of that occurring in a real world occurrence are unrealistic. Using real word data or random data as per this project, it is expected to get an average /worst case and a time complexity of n2 will be seen. The 3 simple sorting algorithms all have a space complexity of 1 meaning there is no need extra space required to sort the list. In each case, the algorithm is only comparing 2 elements at a time and swapping them if required.

Quick sort is an efficient comparison based sorting algorithm and hence why it is more practical that the simple comparison based algorithms. The running time can vary dramatically depending on how partitioning is performed resulting in either balanced or unbalanced sub arrays. Balanced sub arrays will result in an average or best case and with that an efficient running time of O(n log (n)). However, it is also possible to see O(n2) if a poor pivot has been selected. While I have mentioned in the below table that Quick Sort is stable, it can be unstable in certain instances in which elements may change two similar elements during the process.

Counting Sort has a O(n + k) complexity across worst, average and best. Meaning it can achieve linear complexity once certain assumptions about the input is known.

**Table 1. Overview of complexity**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Algorithm** | **Best Case** | **Worst Case** | **Average Case** | **Space Complexity** | **Stable** |
| Bubble Sort | n | n2 | n2 | 1 | Yes |
| Selection Sort | n2 | n2 | n2 | 1 | No |
| Insertion Sort | n | n2 | n2 | 1 | Yes |
| Quicksort | n log n | n2 | n log n | n (worst case) | Yes |
| Counting Sort | n + k | n + k | n + k | n + k | No (Unstable in standard form) |

**Implementation & Benchmarking**

As per the project specification, this section will include the process followed when implementing each of my chosen sorting algorithms; bubble sort, selection sort, insertion sort, quick sort and counting sort. This section will not include an in depth description of the code used to implement the sorting algorithms. All code is clearly explained within the python file of each sorting algorithm and the benchmarking. This section will contain an overview of the steps completed.

Extensive research was completed when completing the implementation of these sorting algorithms. There is a lot of sources online with information on sorting algorithms and therefore I felt it was best to investigate the topic extensively first to identify the sorting algorithms to discuss. I decided to choose 3 simple sorting algorithms, one efficient comparison based sorting algorithms and 1 non comparison sorting algorithm.

As discussed above, bubble, selection and insertion sort are not very efficient using big data sets. But, they are very well known and as an introduction to sorting algorithms, I thought it would be good to investigate these 3 against other and see how they compare. Bubble sort has an easy implementation and in my opinion the easiest algorithm to both understand and code. While not being the most efficient sorting algorithm, it was a great way to start the project and introduce me into the ore complex sorting algorithms. Selection and insertion sort were also easier to understand as they are comparison based algorithms.

I began to first develop and adapt sources for each of the sorting algorithms. Once I understood the logic behind the algorithm I would try to review to numerous sources of code to develop the algorithms themselves. Before completing the benchmarking of any of the algorithms, I first coded them and tested them ensure they could sort data correctly. Once all 5 sorting algorithms were sorting elements correctly, I began the benchmarking aspect of the project. Each sorting algorithm was individually completed and tested in its own python file. Then all 5 were benched marked in another python file.

As part of the benchmarking, there were a couple of modules that were required to be imported:

* import **time –** required to represent time in code in forms such as objects, numbers & strings
* Import **numpy –** library that provides fast and efficient operations on arrays
* Import **randint** from **random –** generate random numbers

For the benchmarking specification, each algorithm must run 10 times (at each input size n) and then the average of the 10 run times should be outputted on the screen. I found this to be the most difficult coding aspect of the project. There is substantial knowledge on each of the sorting algorithms but less so on the bench marking.

The project specifications stated how to generate random numbers and obtain the running time of each algorithm but I still needed to determine how to run each algorithm 10 times at each input size n. First I used randint to create a set of random numbers at different input sizes from 100 to 10000. Each sorting algorithm was ran at every input size n (table below) for 10 runs. The average runtime at each input size was then average using numpy.average. There were other options for completing this task, such as more manual calculations but I felt numpy was the best option as it completed the calculation while keeping code to a minimum. As per specifications, each average was then rounded to 3 decimal places. The project also stated to output results in milliseconds, therefore I multiplied the time taken to complete each run x 1000 to output in the correct format.

Another consideration to make was how to benchmark counting sort compared to the other 4 sorting algorithms. Counting sort takes 2 arguments while the other algorithms I designed only took 1 argument (quicksort has 3 arguments at the recursive function but not for overall function). Therefore, I had to create an additional argument for the benchmarking of counting sort. I created a max value and the array that was to be sorted. Finally to ensure the benchmarking was outputted in a nice and readable form, I created a dataframe and changed it to transpose mode to ensure it was in the correct format.

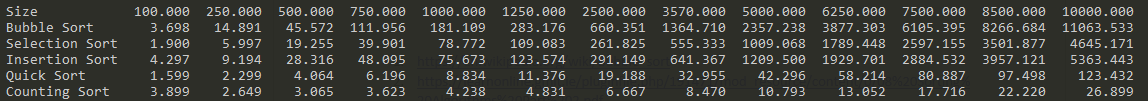
**Results**

As the best case is highly ever to occur, I decided to look at complexity from the average or worst case scenarios as they are much more likely to occur. Based historical data and previous studies on the sorting algorithms, I was expecting counting sort to be the most efficient as it had a time complexity of (n + k). The next most efficient in theory should quicksort. In the average case, quicksort can achieve (n log n). The 3 simple comparison based algorithms would all be expected to achieve (n2) in both the worst and average case. While bubble sort and insertion can technically achieve a complexity of n in the best case, the odds make it basically impossible. Of the 3 simple comparison based algorithms, I would expect bubble sort to demonstrate the worst complexity as input size drastically affects the run time of this algorithm. Then insertion sort and selection sort should be similar in run time performance but research suggests that insertion sort is the more efficient of the two. [7]

**Graph 1:** Complexity Comparison of 5 Sorting Algorithms

**Table 2:** Run times (milliseconds) of 5 Sorting Algorithms



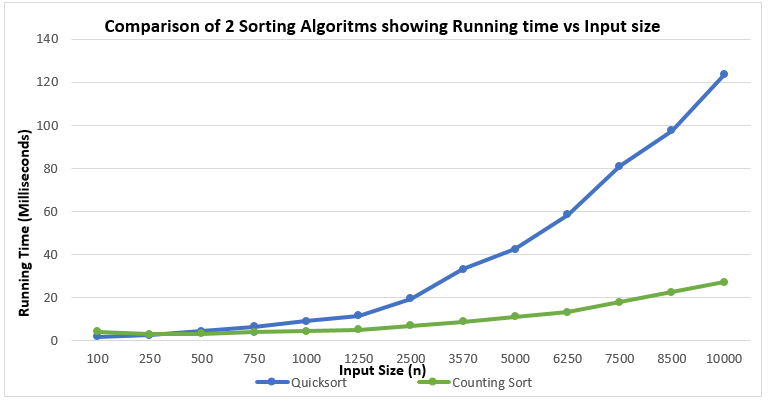
**Table 3:** Run times (milliseconds) of 5 Sorting Algorithms taken

Source:Screenshot from console

Based on the graph above we can see that bubble sort (as expected) is by far the worst performer of all the sorting algorithms. Both Graph 1 and table 2 clearly indicate that as the input size n increases, the running time proportionally increases based on the input size.

Selection and insertion sort are the next poorest performers. As mentioned above, insertion sort can achieve a best case scenario of (n) but that is not realistic using real world data. The graph demonstrates that both achieve an n2 complexity in that the running time drastically increases at higher input sizes of n.

From graph 1 above, we can clearly tell that both quick sort and counting sort are by far the 2 most efficient algorithms of the 5. However, it is difficult to tell their growth from the above graph due do the difference in running times. The graph does not represent counting sort and quicksort appropriately. See graph 2 below for close up of counting sort and quicksort.

**Graph 2:** Complexity Comparison of 2 Sorting Algorithms

As can be seen in graph 2, quicksort is less efficient than counting sort. As we know, comparison based algorithms cannot improve on n log n. The above graph indicates that in this instance, quicksort achieved a logarithmic complexity. It is possible for quicksort to demonstrate n2 but based on its performance compared to the other comparison based sorts which did achieve n2 we can show quick sort did not and is the most efficient of all the comparison sorting algorithms discussed in this project.

Counting Sort, as expected is the most efficient algorithm. From graph 2, we can see that it has achieved a linear time complexity. A time complexity which is not achievable by any of the others. The figures in the tables and graphs show that bubble sort is significantly slower than any of the other sorting algorithms while counting sorting is the best performer. Quick sort, as its name suggests, can sort efficiently in the average case and based on the data above would be the best comparison based algorithm for real world application.

1. <https://en.wikipedia.org/wiki/Bubble_sort>
2. <https://learnonline.gmit.ie/pluginfile.php/191471/mod_resource/content/0/08%20Sorting%20Algorithms%20Part%202.pdf>
3. <https://www.tutorialspoint.com/data_structures_algorithms/bubble_sort_algorithm.htm>
4. <http://www.iiitdm.ac.in/old/Faculty_Teaching/Sadagopan/pdf/DAA/SortingAlgorithms.pdf>
5. <https://www.hackerearth.com/practice/algorithms/sorting/selection-sort/tutorial/>
6. <https://www.programiz.com/dsa/selection-sort>
7. <https://en.wikipedia.org/wiki/Insertion_sort>
8. <https://www.studytonight.com/data-structures/insertion-sorting>
9. <https://www.interviewbit.com/tutorial/insertion-sort-algorithm/>
10. <https://www.hackerearth.com/practice/algorithms/sorting/quick-sort/tutorial/>
11. <https://en.wikipedia.org/wiki/Quicksort>
12. <https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheQuickSort.html>
13. <https://www.programiz.com/dsa/counting-sort>
14. <https://www.oreilly.com/library/view/algorithms-in-a/9780596516246/ch04s07.html>
15. <https://www.interviewcake.com/concept/java/counting-sort>
16. <https://www.programiz.com/dsa/counting-sort>
17. <https://www.geeksforgeeks.org/stability-in-sorting-algorithms/>
18. <https://javarevisited.blogspot.com/2017/02/difference-between-comparison-quicksort-and-non-comparison-counting-sort-algorithms.html#:~:text=No%20comparison%20sorting%20includes%20Counting,which%20examines%20bits%20of%20keys.&text=On%20the%20other%20hand%2C%20non,on%20integer%20arithmetic%20on%20keys.>

<https://www.tutorialspoint.com/design_and_analysis_of_algorithms/analysis_of_algorithms.htm>

<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.113.4254&rep=rep1&type=pdf>

<https://www.techiedelight.com/in-place-vs-out-of-place-algorithms/>