Introduction:

Sorting involves the analysis of data, such as numbers and words, and determining their order in an array, via numerical order or alphabetical order respectively. Sorting is a useful concept in solving problems in daily life, for example, searching for a specific item in a list/database would be more efficient if the list has been sorted prior to searching.

Sorting algorithms are used in various fields such as Mathematics and Computer Science to organise an array from smallest values to largest values in the most efficient, least time-consuming manner. In order to achieve this, an understanding of algorithms and how each type of algorithm operates is vital.

There are many varieties of sorting algorithms, the focus will be on five sorting algorithms referred to later in this report, along with numerous factors which can impact on how efficient (or inefficient in some cases) an algorithm's code can potentially be, and comparing each algorithm with another.

Not all algorithms are created equal, this is relevant in the case of efficiency. Efficiency is how quickly (or slowly) an algorithm starts and finishes its tasks. There are two types of efficiency regarding algorithms: time and space complexity, the focus being on time complexity.

Time efficiency takes into account the time required for the computer to run an algorithm.

Complexity measures the efficiency of an algorithm along with various factors that can impact on its efficiency overall, such as the size of input n the algorithm will process, any issues with hardware specifications and also the quality of the compiler.

These factors can define the performance of the algorithm, such as time, memory, disk usage that the computer's hardware is capable of processing in order for the algorithm to run as efficiently as possible.

The complexity of algorithms can be determined by calculating the running time for each algorithm to process input data of various sizes of n.

Big O notation is a measure of how fast a function grows or declines, this is commonly stated to describe an algorithm's complexity in its worst-case running time behaviour.

As the value of n increases, the running time of an algorithm would also increase in a particular order of growth depending on the algorithm that is implemented.

In-place sorting refers to a sorting algorithm that uses a fixed additional amount of memory which is separate of the input size n, whereas stable sorting focuses on how the sorting algorithm operates when it encounters elements that are repeated throughout the array.

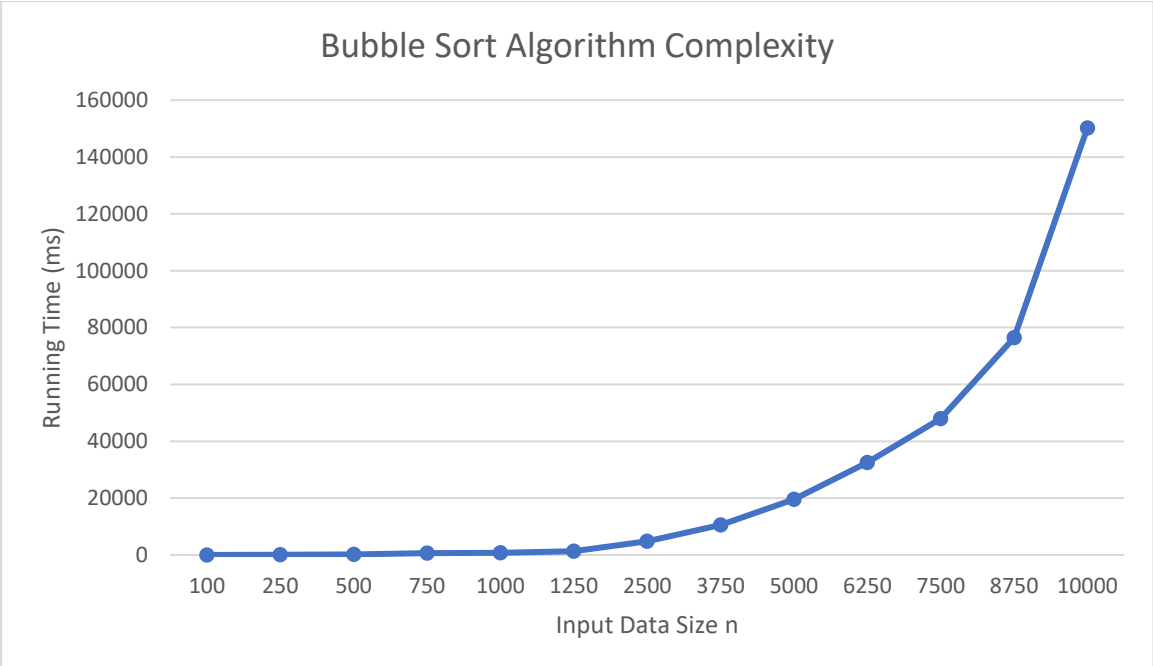
Comparator functions are used in an algorithm to determine the ordering of elements within an array via pre-defined calculations.

Comparison sort algorithms tend to determine which of two elements should appear in an array first. This is the only option for the algorithm to read two elements at any time and compare these to implement an order in the array, whereas non-comparison sort algorithms use an internal character of two elements to be sorted.

Sorting Algorithms:

1. **Bubble Sort – A Simple Comparison-Based Sort:**

The Bubble Sort algorithm is regarded as the most simply structured sorting algorithm in programming, serving as a suitable introduction to sorting algorithms.
This algorithm simply compares the first element in an array with its adjacent value to check whether the first element is greater or less than its adjacent element. If greater, the elements are swapped, this trend will repeat throughout the array until all elements are sorted successfully.
From the results of running the code and creation of the graph below, the time complexity of Bubble Sort is deemed to be quadratic, $O(n^2)$, meaning that the worst-case runtime is directly proportional to the square of the input data n, as a result of the algorithm double-checking values in the event of any duplicate values which may be present in the array.
This level of efficiency is a result of the Bubble Sort code containing an initial for loop followed by a nested for loop, where the worst-case running time is hinging on the square of the number of elements that are within the array.
In the case where Bubble Sort would be working on an already sorted array, also with the 'already_sorted' function in the algorithm's code, these factors ensures the running time complexity can be reduced from an inefficient quadratic time $O(n^2)$ to a slightly more efficient linear time O(n), this is due to the algorithm not needing to revisit any element again once it has been initially sorted.
In conclusion, the Bubble Sort algorithm is straightforward to implement and is efficient in dealing with smaller arrays (below 1,000 n), but its main disadvantage is the lack of efficiency while attempting to sort larger arrays (above 1,000 n).



2. **Merge Sort – An Efficient Comparison-Based Sort:**

The Merge Sort algorithm uses recursion (also referred to as divide and conquer) to organise and rearrange elements within an array. The concept involves splitting the array down into two smaller, equal length and more manageable subarrays.

By using recursion, the elements within each subarray are sorted and then appended to an empty array that will serve as the final definitively sorted array.

The first step of the Merge Sort algorithm begins with a while loop, and the while loop ends when the final array contains every sorted element from both subarrays. Elements at the start of both subarrays are then compared and the smaller value is then appended to the final array, these steps repeat until all elements are analysed and appended to the final array appropriately.

From the results of running the code and creation of the graph below, the time complexity of Merge Sort differs based on both functions in the code:

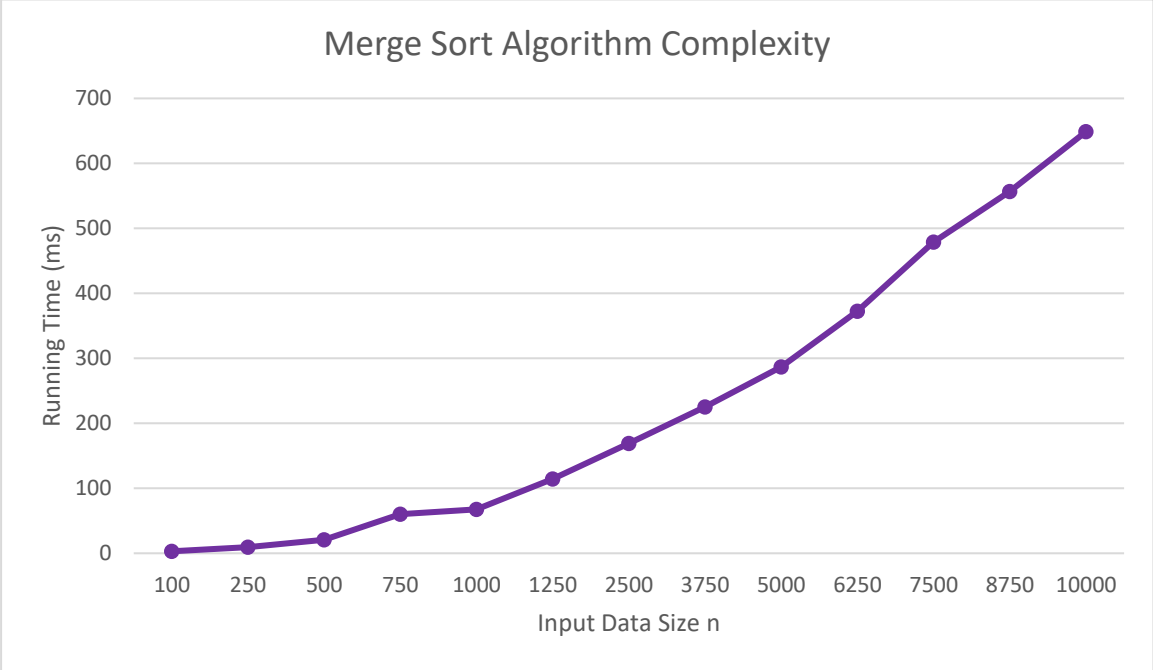
Firstly, with **merge (left, right)**; the time complexity is deemed to be linear, O(n), as a result of receiving two subarrays with a combined length of n (the length of the original array). The subarrays will then combine at the end as sorted, looking at each element once only.

Secondly, with **merge ( )**; the starting array is split recursively and calls the first merge function for each subarray. Each subarray is then halved until one single element is left over, this results in logarithmic, $\log_2 n$, as the merge ( ) function is called upon for each half in the subarrays.

Finally, combining the time efficiency of both functions of the Merge Sort algorithm yields the total time efficiency to be $O(n \log_2 n)$.

Merge Sort is an efficient algorithm due to its recursive abilities and that its runtime can scale efficiently as the input size n is increased, judging by the results obtained.

But the drawback with the Merge Sort algorithm is the fact that copies of the subarrays are created when the function calls upon itself via recursion, this results in higher memory usage required as opposed to Bubble Sort.



3. **Bucket Sort – A Non-Comparison Sort:**

The Bucket Sort algorithm works by scattering elements in an array into a number of empty buckets. These empty buckets are created based on the largest number found in the array which is divided by the total length of the input array. This determines the size of each bucket.

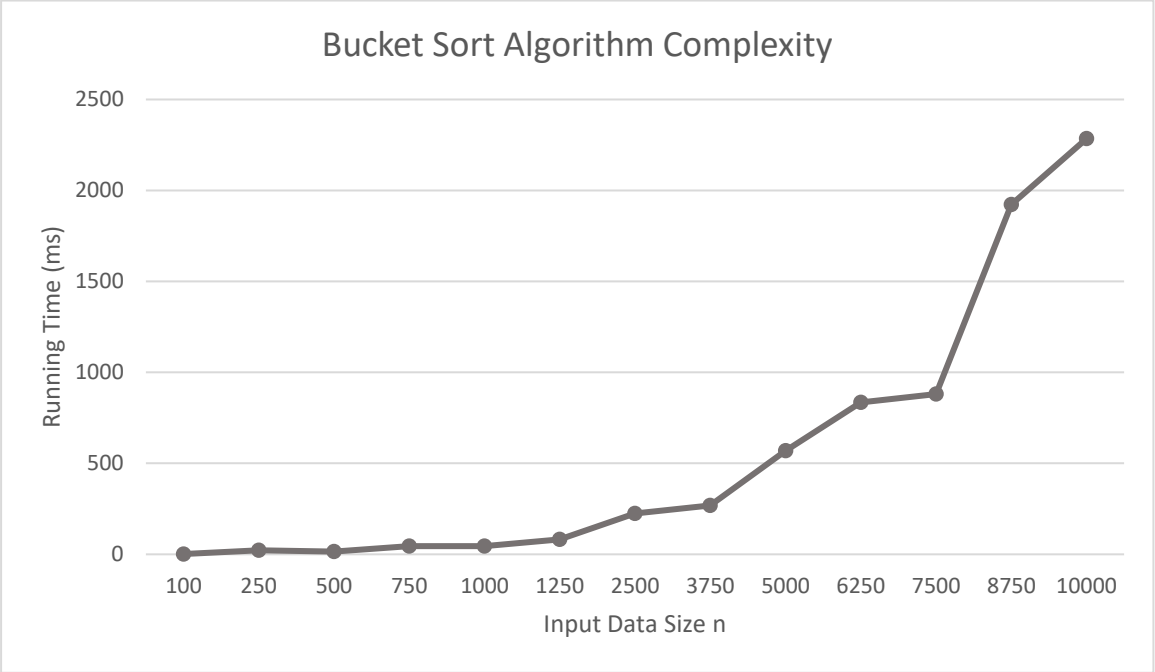
Next, each element is divided by the calculated bucket size, this results in the applicable index of a bucket where the applicable element is to be placed in. The elements inside each bucket are compared with one another and are sorted via the Insertion Sort algorithm (more information on Insertion Sort later in the report).

Finally, all elements are then gathered and merged into the original bucket (array) via iteration, from smallest to largest values.

From the results of running the code and creation of the graph below, the running time complexity is deemed to be quadratic, $O(n^2)$, in the worst-case. When the buckets are established at the beginning of the algorithm, similar elements may be placed within the same bucket which causes an uneven dispersion of elements between the total amount of buckets.

Also, the use of an intermediate sorting algorithm (in this case, Insertion Sort) midway through the Bucket Sort algorithm determines the overall complexity.

In conclusion, Bucket Sort tends to work efficiently with smaller arrays in a linear fashion, but as the input data size n increases gradually, use of the Insertion Sort will result in a less efficient (quadratic) means of organising the entire array, particularly with larger arrays.



4. **<u>Insertion Sort – A Simple Comparison-Based Sort (Choice #1):</u>**

The Insertion Sort algorithm operates by placing an unsorted element in an array to its correct right-hand place and repeating these steps for every unsorted element throughout the entire array.

According to the code, the first element (Index 0) is assumed to have been sorted, so the sorting begins at the second element (Index 1), this element will be labelled as the 'key' when traversing the array.

This 'key' is used to compare itself to all elements in the array, if the adjacent element is greater than the 'key', the 'key' is placed in front of the first element.

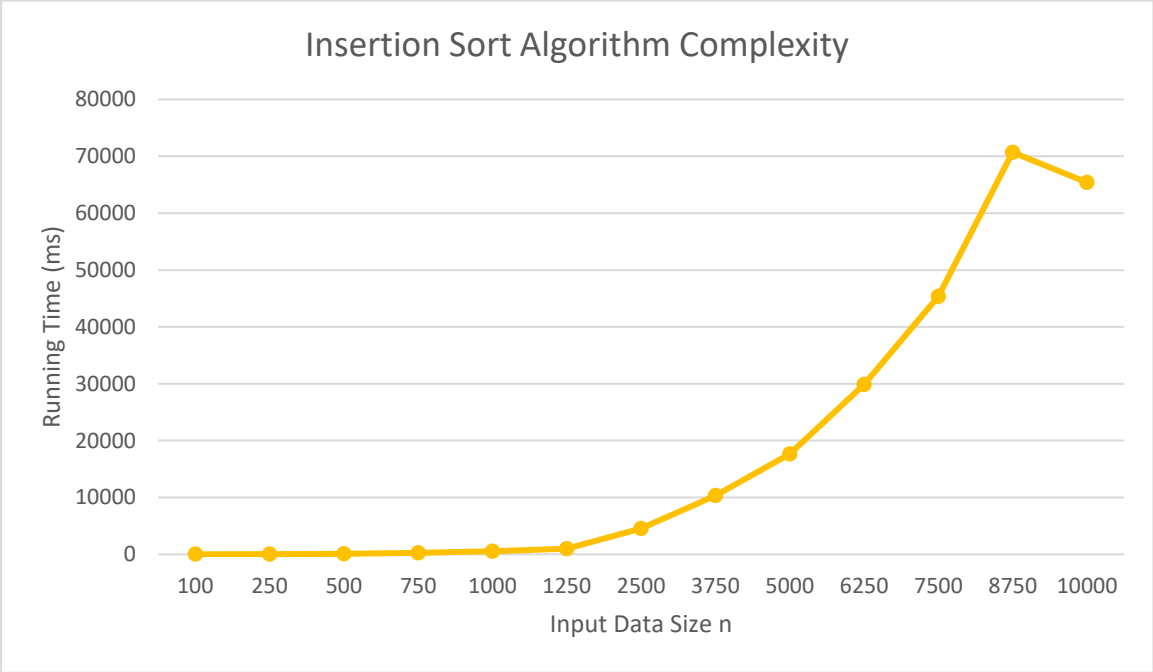
This process repeats, comparing each adjacent element one at a time to sort in comparison with the 'key' until the array is fully sorted.

From the results of running the code and creation of the graph below, the time complexity of Insertion Sort is deemed to be quadratic, $O(n^2)$, which is indicative for a simple comparison-based sort such as Bubble Sort.

The Insertion Sort code has a for loop and a nested while loop; this while loop is used to find the correct position of an element in comparison to the 'key' for each case in the array.

When comparing the run-time efficiency of both Insertion Sort and Bubble Sort, Insertion Sort is slightly more efficient overall as the nested loop makes less comparisons in order to sort the list, resulting in a faster run-time in comparison to Bubble Sort.

In conclusion, Insertion Sort is easy to implement like Bubble Sort, it works well for sorting smaller arrays but not so well in sorting larger arrays.



5. **Quicksort – An Efficient Comparison-Based Sort (Choice #2):**

The Quicksort algorithm operates by dividing the array into subarrays, unlike Merge Sort, Quicksort consists of three subarrays, consisting of a subarray for small values, equal values and large values.

Quicksort utilises a pivot, this is a randomly selected element and this will divide the total array around the designated pivot, resulting in low value, equal value and high value subarrays.

The positioned pivot ensures the function can use recursion to all subarrays until all elements are sorted.

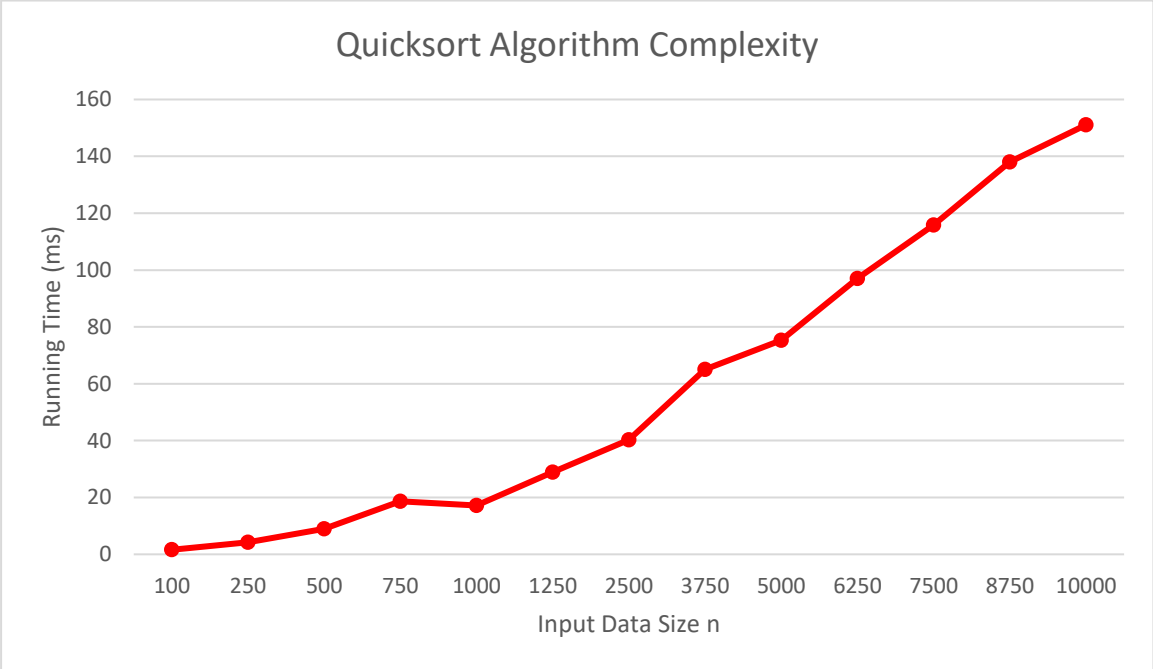
From the results of running the code and creation of the graph below, the time complexity of Quicksort consists of linear running time, O(n), from splitting the array into three subarrays equal to the total length of the input array.

This is followed by a logarithmic running time, $n \log_2 n$, from the use of recursion with three subarrays.

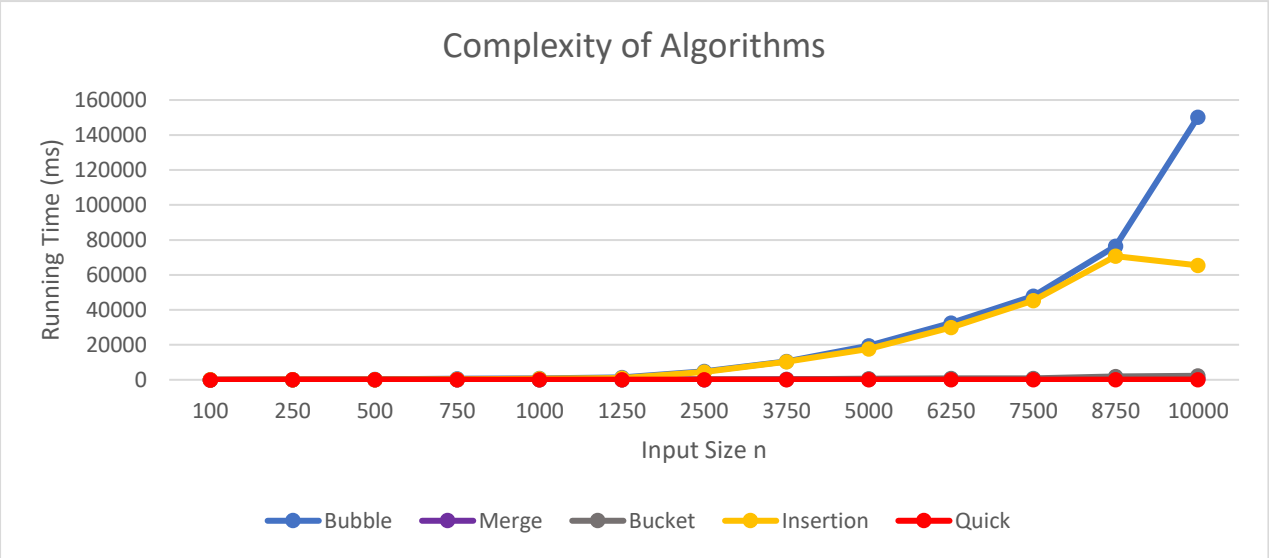
Finally, combining the time efficiency of the Quicksort algorithm yields the total time efficiency to be $O(n \log_2 n)$.

In conclusion, Quicksort is an efficient algorithm like Merge Sort, due to its recursive abilities and that its runtime can scale efficiently as the input size n is increased.

The drawback with the Quicksort algorithm are similar to Merge Sort, due to the fact that copies of elements can occur within the created subarrays, this results in higher memory usage required and a slightly slower running time efficiency.



Implementation & Benchmarking:



Size n	100	250	500	750	1,000	1,250	2,500	3,750	5,000	6,250	7,500	8,750	10,000
Bubble Sort	11.367	60.515	238.719	670.802	780.822	1,370.614	4,782.241	10,608.344	19,533.795	32,574.205	47,892.430	76,452.963	150,222.478
Merge Sort	3.046	9.218	20.558	60.166	67.591	114.585	169.175	225.445	286.592	372.506	479.156	557.002	648.810
Bucket Sort	1.711	21.685	15.764	45.095	45.388	82.703	225.280	267.982	570.177	834.597	881.324	1,922.266	2,284.259
Insertion Sort	8.478	22.636	110.816	240.805	567.049	996.064	4,543.862	10,343.063	17,669.608	29,891.253	45,362.031	70,715.461	65,384.764
Quicksort	1.603	4.261	8.958	18.702	17.226	28.845	40.258	65.047	75.308	97.058	115.842	138.000	151.055

Reference Material:

Mannion, P., Carr, D. (2021). *Computational Thinking with Algorithms PowerPoint Presentations.* Galway-Mayo Institute of Technology (GMIT).

Valdarrama, S. *Sorting Algorithms in Python.* Real Python. https://realpython.com/sorting-algorithms-python/

Galler, L., Kimura, M. (2019, April 21). *What are Sorting Algorithms?* LAMFO. https://lamfo-unb.github.io/2019/04/21/Sorting-algorithms/

Srivastava, P. (2020, October 20). *Stable Sorting Algorithms.* Baeldung. https://www.baeldung.com/cs/stable-sorting-algorithms

Sanatan, M. *Sorting Algorithms in Python.* Stack Abuse. https://stackabuse.com/sorting-algorithms-in-python/

Shaik, H.K. (2021, January 23). *Sorting Algorithms Implementations in Python*. Geekflare. https://geekflare.com/python-sorting-algorithms/

Seif, G. (2018, November 26). *A tour of the top 5 sorting algorithms with Python code*. Medium. https://medium.com/@george.seif94/a-tour-of-the-top-5-sorting-algorithms-with-python-code-43ea9aa02889

Khalid, M.J. (2020, December 3). *Bucket Sort Algorithm & Time Complexity | Python Implementation.* Ingenio Pvt Ltd. https://mjunaidkhalid.com/2020/12/03/bucket-sort-algorithm-time-complexity-python-implementation/