Computational Thinking with Algorithms

Submitted in partial fulfillment of H.Dip in Data Analytics in Computing

Author: Katie O’Brien G00398250

Introduction:

Algorithms are the building blocks of computer programming. They are a set of rules that must be followed when solving a particular problem. From the most basic “Hello World” script that is commonly used as a first program, to programs used for areas as complex as automation and AI. They are all fundamentally based on the concept that computers are not intuitive, they can only do exactly what you tell them.

While this on the surface might seem like an easy concept to grasp, looking at the popular concept that an algorithm is like a recipe, potential pitfalls emerge. The simple instruction “Pour cake batter into pan” immediately presents a number of additional questions such as “What cake batter,” “Pour from where?,” “Into what pan” and “For how long.” For humans, this is easily answered, however, as a computer, these details need to be expressly laid out in advance in order for the algorithm to run as intended, and the related problem to be solved.

There are many types of algorithms, for many types of problems. They include Brute Force Algorithms, Greedy Algorithms, Recursive Algorithms, Backtracking Algorithms, Divide & Conquer Algorithms, Dynamic Algorithms, and Randomized Algorithms (Dev et al., 2022). These distinct types of algorithms have pros and cons, and many options may be impractical depending on the problem that requires solving. There are several factors to consider when looking at the type of problem that a user is hoping to solve- for example, in certain tasks, speed is a key consideration. One such area where this is important is mapping and route planning such as that offered by Google Maps, where results must be almost instantaneous. A search query in this application would be useless if it did not return for an hour or more. For this reason, in some cases there is a tradeoff to be made with time against accuracy, with some developers recognizing that a margin of error may in fact be a reasonable compromise in order to get the speed necessary.

In this assignment we are going to look specifically at sorting algorithms. This is a key area in algorithmics and there are many developments associated with it. Research into sorting algorithms has been ongoing since the 1950’s and still attracts a great deal of research, as despite its simplicity it is a complex problem to solve efficiently. The use of sorting algorithms is often used in conjunction with searching algorithms as it is operationally faster in most cases to run unsorted data through a sorting algorithm and then a searching algorithm, than it is to try and run a searching algorithm on unsorted data.

The concept of sorting an array such as [1,4,2,3,5] may seem a very straightforward process for us as humans to process and correctly sort. However, when looking at a data input with 1000 numbers, etc. then it becomes more difficult, which is where we need alternative strategies to sort the data. A computer solving an algorithm is no different, with the only difference being the amount of space and time required to complete the task.

There are a number of different sorting algorithms, all with different properties. There is no “best” algorithm to use when selecting an algorithm to sort a user’s inputted data as there are a number of factors that make up a user’s decision to choose the correct one. This can be the amount of data that needs to be sorted, the “sortedness” of the data, and the type of data. For example, there are some algorithms that work better with data that is almost sorted, whereas with a different algorithm there may be no appreciable difference in time or complexity efficiency. The role of the user is to accurately assess and understand the data that they are working with to correctly use appropriate algorithms to solve their problem.

One of the ways that can be used to determine what the best algorithm is, is to benchmark the algorithm. This is a process by which an algorithm can be ran over a series of data inputs, varying in size to determine the time and space complexity of the algorithm in question. Time and space complexity is a crucial factor, and it plays a huge part in determining the suitability of an algorithm.

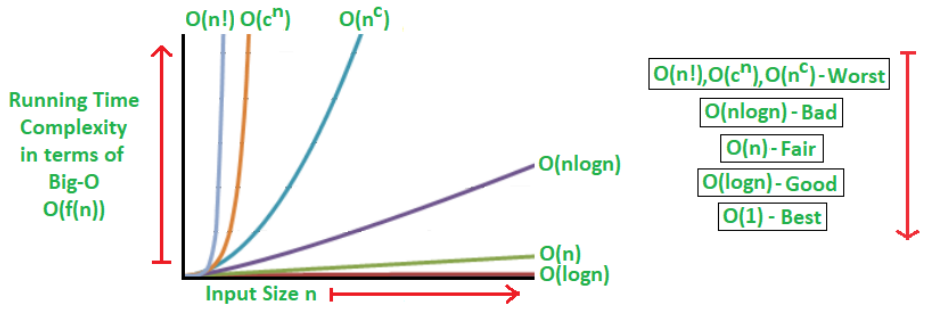
Time and space complexity is a way of measuring what resources will be required to run an algorithm, and thus determining its efficiency. As the name suggests, they measure the amount of time (or number of operations) and memory required to run the algorithm. There are 2 ways of measuring time complexity, *a priori* and *a posteriori* analysis.

A posteriori analysis, or benchmarking, is an empirical evaluation whereby algorithms to be compared are run on a platform and their relative performances are analyzed based on the actual measurements. While this will give an exact answer to the time complexity of the algorithm in question, it is dependent on the compiler and hardware in the machine running the analysis and the results will vary between systems. (Difference between Posteriori and Priori analysis - GeeksforGeeks, 2022)

A priori analysis on the other hand is a theoretical analysis. It is totally independent of complier, hardware or machine type and compares algorithms based on order of magnitude to give an excellent approximation of the complexity of the algorithm. Order of magnitude, or Big-O notation is an "asymptotic notation" that determines how difficult the algorithm will be to solve as the size of the input increases by orders of magnitude ie, from 1 to 10 to 100 to 1000, as opposed to 100-102. There a number of asymptotic notations, which are basically mathematical notations, used to describe the running time of an algorithm, (Big-O Notation, Omega Notation and Big-O Notation (Asymptotic Analysis), 2022). These notations represent best, average, and worst cases, with Big-O representing the worst-case scenario. When looking at the time complexity results, we often see them also denoted with an "*n"* which represents the input size, or *"k"* which can represent "key" values.

There are a number of complexity families that are used to classify the efficiency of algorithms. In decreasing deficiency these are (DAA Complexity of Algorithm - javatpoint, 2022):

- Constant - (denoted as **O(1)**)
- Logarithmic - **O(log(n))**
- Linear - **O(n)**
- Polynomial - (I.e., **O(n^2), O(n^3), O(n^4)**
- Exponential - **O(2^n), O(N!)**



(Analysis of Algorithms | Big-O analysis - GeeksforGeeks, 2022)

There are 3 cases that can be used to determine the time complexity of an algorithm- Best, Average, and Worst cases. In the vast majority of instances, the Best case is a conceptual idea, and is not going to be seen in "real world" scenarios all that often but is a helpful comparison between algorithms. For a lot of users, when picking an algorithm, they will consider the Average case scenario as this is likely to cover the majority of input scenarios. However, for some instances, it may be better to consider the worst case. For example, an average case might offer a *n*log*n* time complexity, but worst case may end up being a quadratic time complexity, and this may not be suitable for the project/problem in question.

A consideration that will impact the space complexity is in-place versus out-of-place sorting. An in-place algorithm sorts the input without using any extra memory- i.e., the input is overwritten by the output. That is not to say that the algorithm won't use any space to sort, more so that the input size does not affect this extra memory usage. An out-of-place algorithm will sort into a separate array, or alternative holding option, as the algorithm is running. In this instance the extra space used will depend wholly on the input size. (In-place vs out-of-place algorithms | Techie Delight, 2022)

It should be noted that there is often a trade-off between optimal memory use and running time performance (Analysis of Algorithms | Big-O analysis - GeeksforGeeks, 2022). Generally, with algorithms, space and time efficiency are at polar ends of a scale, so the less time efficiency you have, the greater space efficiency and vice versa. Overall, having an algorithm that can have the smallest possible time and space complexity can make a huge difference in performance.

Sorting Algorithms:

As mentioned, there are many types of sorting algorithms. Bubble sort was one of the earliest algorithms developed and was analyzed as early as 1956. They can broadly be broken into 2 distinct categories- comparison based and non-comparison sorts.

Comparison based algorithms sort by reading the list of elements through a comparison operation i.e., a function where the 2 elements are compared with a comparison operator- the most common being less than (<), greater than (>) or equals (=). This simple comparison determines which of the elements should occur first. (Comparison of Sorting Algorithms, 2022)

Contrast this with the concept of non-comparison sorts which do not compare each individual element to each other, instead using keys, buckets, or other grouping mechanisms to sort the data, either alone or with the aid of a comparison algorithm in the case of hybrid sorting algorithms.

Where the 2 elements are the same, the ability of an algorithm to maintain the original input order is called stable sorting, although it should be noted that not all sorting algorithms can offer this feature. Stable sorting can be important, especially if the input being sorted is only a small component of the overall data, with other values and variables attached to the input. For example, a dataset of deliveries for a company may have the customers already sorted by delivery date, but may now wish to sort by name within that delivery date. If the algorithm used cannot provide a stable sort, the data table may be sorted by date, but the names are now no longer in alphabetical order, which can be problematic and frustrating. In an instance such as this, stable sorting is a key requirement.

In depth look at 5 sorting algorithms.

We are going to look now at 5 commonly used sorting algorithms. These are a combination of comparison and non-comparison sorts, and we will briefly look at their history, time and space complexities, and how they are used.

Bubble Sort:

Bubble Sort was one of the first algorithms to be studied shortly after the development of computers. It is a very simple comparison sorting method that does not perform well with large input groups. This is due to the characteristics of the algorithm, which compares one element with the element on its immediate left (or right) and swaps if the element is smaller. It repeats this process with each element until the whole array is sorted. As each element is being iterated through it can be a very time intensive process.

As a result, unsurprisingly, Bubble Sort has a worst case and average case complexity of $O(n^2)$, where n is the number of items in the array, which is slower than most other practical sorting algorithms. As a result, this is often only used as a teaching method. Bubble sort has one significant advantage over most other algorithms, and that is that it can efficiently detect that the list, or array is sorted by default. When you compare this to most other algorithms, even those with better average case complexity, they must perform the entire sorting process on the input data. (Bubble sort - Wikipedia, 2022)

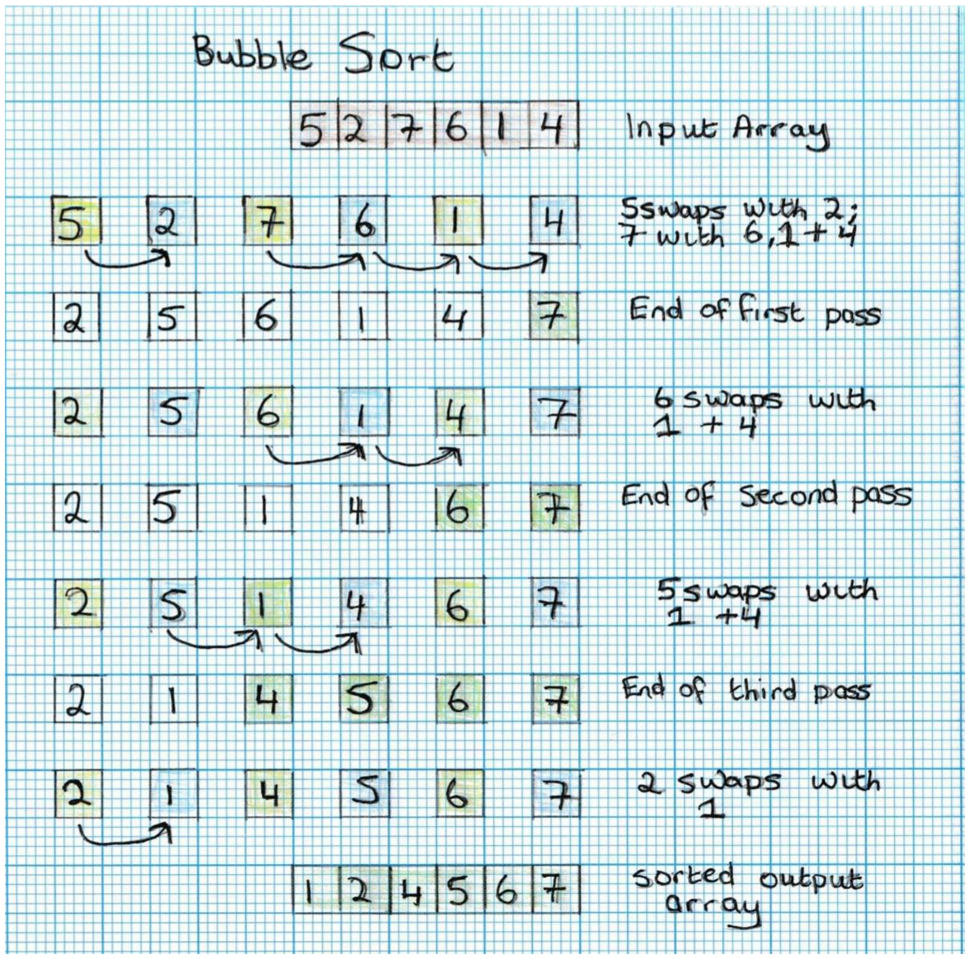
- How Bubble sort works

Consider an array with unsorted elements such as [5,2,7,6,1,4]. In the first pass the element 5 is compared against 2. 2 is less than 5 so 5 and 2 swap. Next 5 and 7 are compared, and in this instance 5 is less than 7 so they stay as they are. 7 is then compared with 6 and swapped, as it is with 1 and 4. After the first pass the numbers are [2,5,6,1,4,7].

In the second pass, 2 and 5 are compared and as they are all in order they do not move. 5 and 6 are also compared and do not move. 6 is then compared with 1 and 4, and swaps are done with both of them. After the 2nd pass the elements are now [2,5,1,4,6,7].

The 3rd pass checks 2 and 5 and they are still in order, however 5 swaps with 1 and 4 to leave the array looking like so: [2,1,4,5,6,7].

At the 4th pass 2 and 1 are compared and swap, then 2 and 4 are compared and stay where they are. As the rest of the row is now greater than 4 the input is now sorted and looks like this: [1,2,4,5,6,7], and the algorithm is complete. This whole sequence can be seen in the diagram below.



As can be seen from the number of passes the time complexity on this is vast, particularly when the algorithm must iterate through an input that is far greater than the example above. This *can* be reduced by adding in an optimization in the code used. This will check the code and prevent comparisons being made if the array is already sorted, preventing the code iterating through further passes. (Bubble Sort (With Code in Python/C++/Java/C), 2022)

Quick Sort:

Quicksort is a comparison sorting algorithm developed in the late '50's by Tony Hoare, a well-known computer scientist. It is what is known as a "divide and conquer" algorithm that works by selecting a "pivot" element from the input array and partitioning the elements into 2 subarrays. It was developed as a more efficient version of insertion sort when Hoare was working in Moscow. After publication Quicksort became the default sort library in Unix and was also used in early versions of C and Java (Quicksort - Wikipedia, 2022).

Quicksort is still commonly used today, and a good implementation of the algorithm can be faster than similar algorithms such as merge sort and heapsort. However, efficient implementations are not stable, meaning that the order of equal sort items is not preserved. This can be a factor for the user, depending on the input data that is required to be sorted.

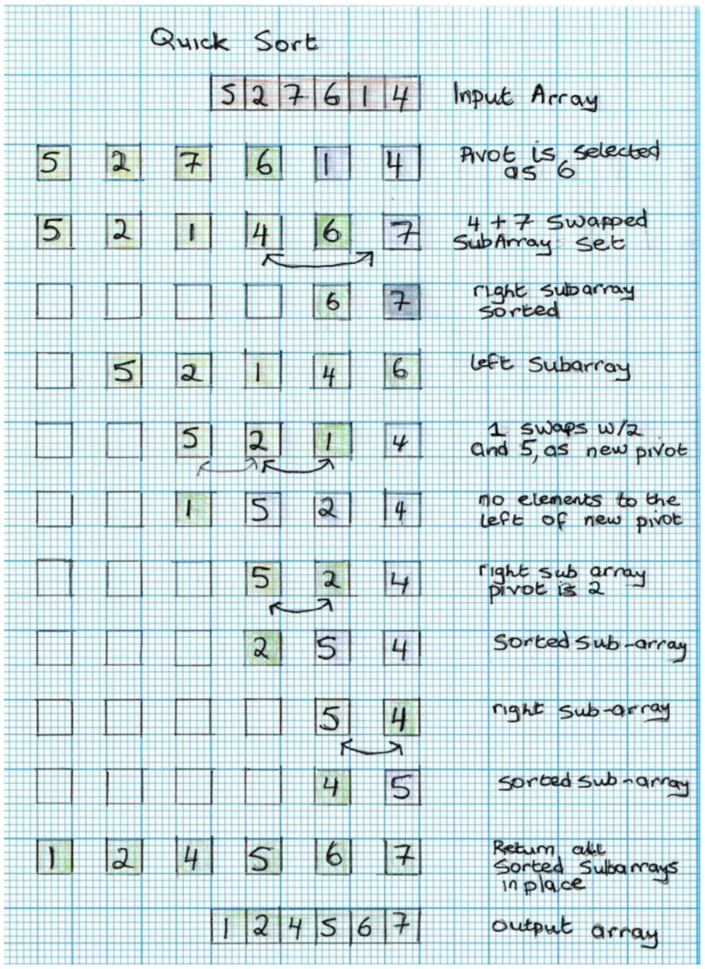
Quicksort's time complexity is O(n log n) for best case and average case, O(n^2) for the worst case (Quick Sort, 2022). Quicksort can sort in place- requiring only small additional amounts of memory to perform the sorting.

As mentioned, quicksort is a divide-and-conquer algorithm that works by selecting a "pivot" element from the array. The array is split into 2 sub-arrays depending on whether they are greater or less than the "pivot" element. These sub-arrays are then sorted recursively.

- How quicksort works

Consider an array such as the one used above in Bubble Sort [5,2,7,6,1,4]. A pivot is then chosen- this can be any element; however, the middle element is often used. In the past, the left or right-most element was used which caused worst-case runtimes on almost sorted arrays. This is because one sub-array may have little or no elements in it, and the bulk of the elements will be in the other. (Quicksort - Wikipedia, 2022), (QuickSort (With Code in Python/C++/Java/C), 2022)

So, in this instance we will use 6 as the pivot element. In this case the elements are "sorted" in the first run to have the elements less than 6 on the left-side, and the elements greater on the right: [5,2,1,4,6,7]. This is done by pointers and swapping. So where 7 is greater than 6 a pointer is set on that and is swapped directly with an element less than 6, in this case 4. At this point the subarray on the right of the pivot is sorted as there is only one element there. The steps are then repeated on the left subarray. [5,2,1,4,**6,7**]. In the 2nd pass, 1 is the new pivot so 1 swaps with 2 and then with 5 to reach its final position and the array now looks like this: [**1,** 5,2,4,**6,7].** As there is no subarray to be sorted on the left of the pivot (1) then the process repeats on the new right subarray. The pivot is now 2 and this swaps with 5 to leave the array looking as so: [**1,2,**5,4,**6,7].** Finally, as there are 2 elements left in the subarray to the right of the pivot, the process repeats. The pivot is selected as 4, this swaps with 5 and as there are no subarrays left to be sorted the algorithm ends. (Quick Sort visualize | Sorting | Algorithms | HackerEarth, 2022). This process can be seen below:



Radix Sort:

Radix sort is a non-comparative sorting algorithm which shares close links to another sorting algorithm called Bucket sort and is primarily used to sort strings or integers. In this algorithm, elements from an input array are split into “Buckets,” or sub-groups, based on their radix, or base number (Radix - Wikipedia, 2022). While this sorting method dates back to the late 1800’s, the first memory-efficient computer version was developed by Harold Seward in 1954. He proposed using a linear scan of the array beforehand allowing for concise use of bucket size and offsets before the sorting begins. Compared to other general-purpose algorithms, including Quick sort, it is shown to be significantly faster in many cases. (Radix Sort - GeeksforGeeks, 2022)

Radix is used in modern computing to sort records that have multiple keys, such as datetime stamps. It has a Best, Average and Worse Case time complexity of O(nk), where n is the number of values in the array, and k is the number of digits in the largest number. However, the space required for the intermediate sort can make this algorithm impractical for use, especially in software libraries (Radix Sort (With Code in Python, C++, Java, and C), 2022).

- How Radix sort works

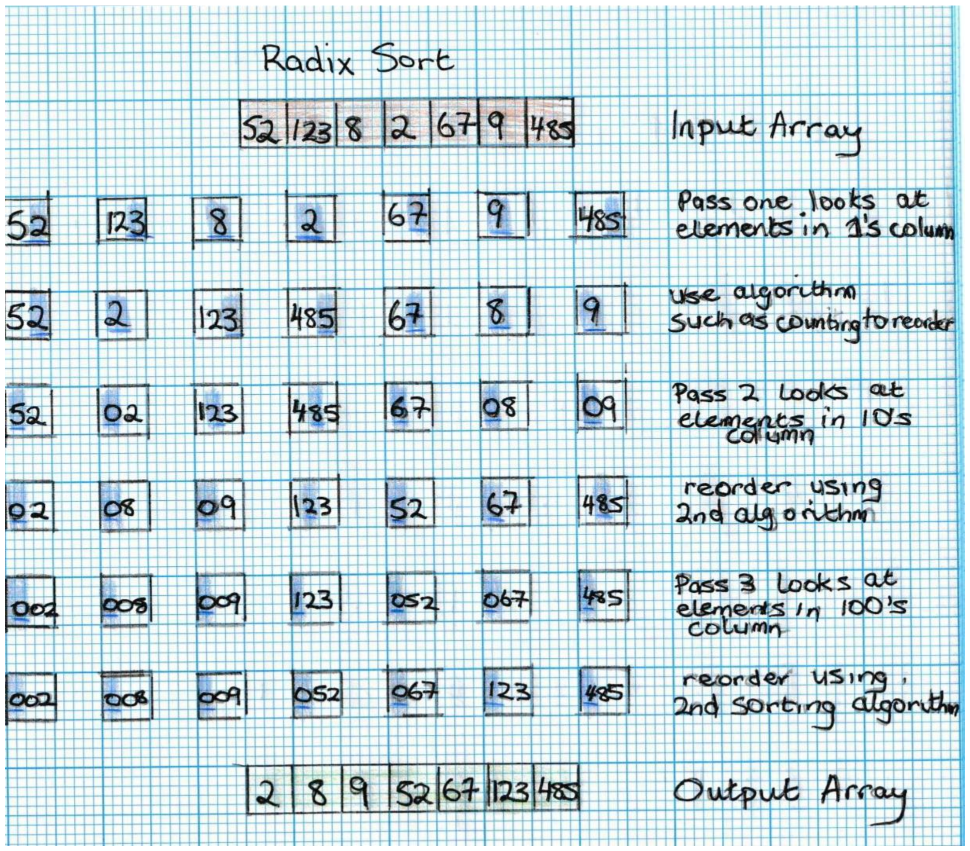
Radix sort can be implemented in one of 2 ways. It can start with either the most significant digit (MSD) or the least significant digit (LSD). This would be either 1 (MSD) or 3 (LSD) in the number 123. Although both are used it should be noted that MSD Radix sorts are not usually stable sorts, so if the original layout of equal sort items is important, it might be worth considering using an LSD-based radix sort instead.

The following array [52,123,8,2,67,9,485] shall be used as a sample input in order to explain how Radix sort works. We shall use an LSD based sort in this instance. To determine how many passes of the algorithm are required we should first get the largest number in the array. In this instance it is 485, so we know that the algorithms will only require 3 passes to complete.

For the first pass we compare numbers in the 1's column. This is demonstrated here: [5**2**,12**3,8,2,**6**7,9,**48**5].** We then use another comparative sorting algorithm such as counting sort to arrange the array in order based on the 1's column which results in the output from pass 1 looking like so: [52,2,123,485,67,8,9] (note that although 52 is clearly bigger than 2 it appeared first in the array so relative position is in place for now).

On the 2nd pass we are looking at the 10's column for the array. Where there are no 10's, such as in the case of 2, 8, and 9, the algorithm will treat them as 0's and sort accordingly. At the end of the second pass the sort will look like this: [2,8,9,123,52,67,485].

Finally, the 3rd pass looks at the digits in the 100's column and sorts within this digit. At the end of this final pass the result should look like this: [2,9,9,52,67,123,485]. At this point the array is fully sorted so the algorithm ends. This process can be seen below:


Radix Sort
52 123 8 2 67 9 485
Input Array
52 123 8 2 67 9 485
Pass one looks at elements in 1's column
52 2 123 485 67 8 9
use algorithm such as counting to reorder
52 02 123 485 67 08 09
Pass 2 looks at elements in 10's column
02 08 09 123 52 67 485
reorder using 2nd algorithm
002 008 009 123 052 067 485
Pass 3 looks at elements in 100's column
002 008 009 052 067 123 485
reorder using 2nd sorting algorithm
2 8 9 52 67 123 485
Output Array

Merge Sort:

Like Quicksort, merge sort is a divide and conquer algorithm. It was developed by John Von Neumann in 1945 and is an efficient multi-purpose algorithm that can produce stable results in most versions. It produces particularly good time complexity results in worst case (Merge Sort, 2022) when compared to quicksort, with one source suggesting that in its worst case, merge sort uses approximately 39% fewer comparisons than quicksort does in its average case. (Merge sort - Wikipedia, 2022). As the algorithm sorts through the whole array, it can be slower than other algorithms for almost sorted data, and for smaller inputs.

Merge Sort has a best, average, and worst-case complexity of O(nlogn)- which is the same as quicksort's best-case complexity. However, Merge sort's most common implementation does not sort in place so the memory size of the input must be allocated for the sorted output to be stored in. This may be a consideration for a user with limited space resources.

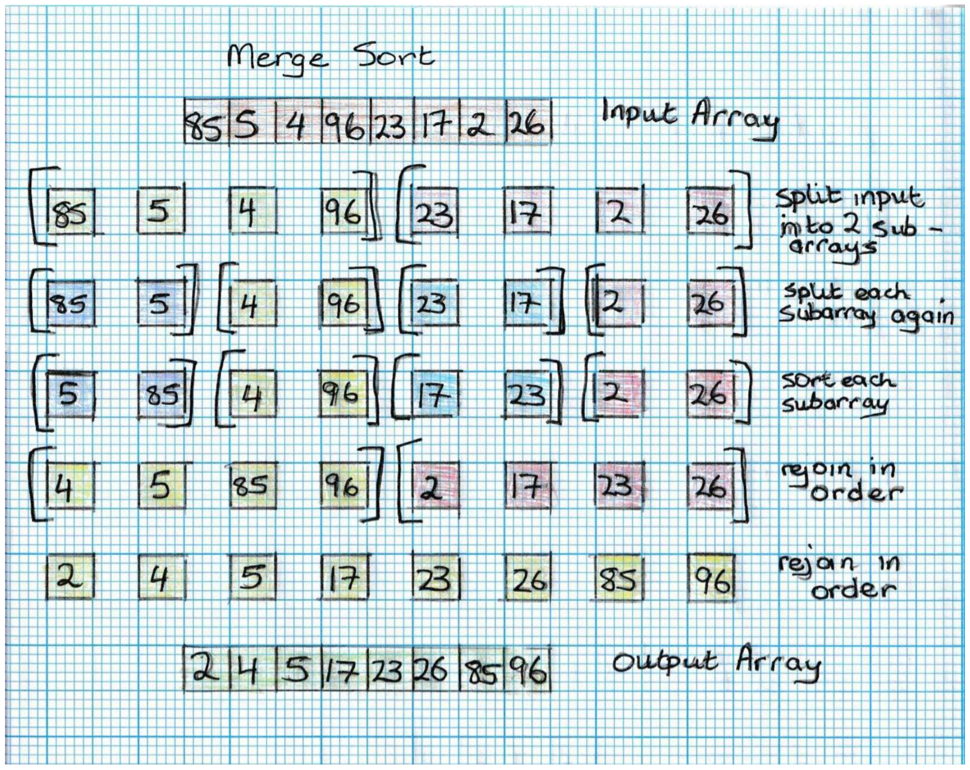
-How merge sort works

In order to best understand how merge sort works its best to describe using an array such as this list of numbers [85,5,4,96,23,17,2,26]. The first step is to split the array into 2 sub-arrays and so there is [85,5,4,96] and [23,17,2,26].

In subarray 1 the array is split into 2 further subarrays [85,5] and [4,96]. These are then split into their elements and are rejoined in correct order- [5,85] and [4,96]. The two sub-arrays are then merged in sorted order to the original subarray [4,5,85,96]

In subarray 2 the array is again split into 2 more subarrays [23,17] and [2,26]. From there this first sub array is again split into 2 leaving this subarray at [23,17] and [2,26]. These arrays are again split into their individual elements and are rejoined in correct order. So [23,17] becomes [17,23] while [2,26] stays as is. These 2 arrays are then rejoined together in order so [2,17,23,26] and sub array 2 is now also sorted.

Finally, the 2 sorted subarrays are rejoined in order with [4,5,85,96] and [2,17, 23, 26] becoming [2,4,5,17, 23,26,85,96] and the array is then sorted. (Merge Sort - javatpoint, 2022), (Merge Sort - GeeksforGeeks, 2022). This process is displayed visually below:



Counting Sort:

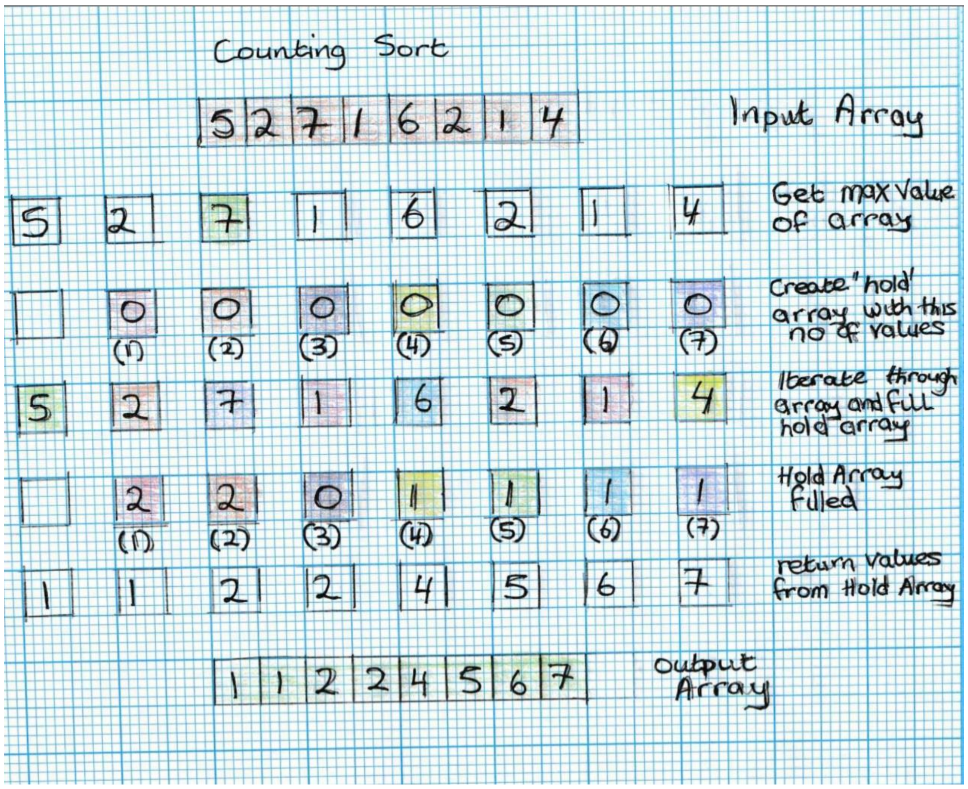
Counting sort is a non-comparison sort created by Harold Seward in 1954. It sorts according to "keys" that are small positive integers. This means that the algorithm counts the number of objects in the input that have distinct "key" values. It works on a similar concept to bucket sort, however, because it does not hold each element in memory, rather, only holds the count of each "key," it can be highly space- efficient, and so may be more suitable for sorting large arrays. (Counting sort - Wikipedia, 2022)

This algorithm has a linear run time based on the number of items but also the difference between the maximum and minimum key values. As a result, this algorithm is only suitable for use in input scenarios where the variation in keys is less than the number of items. For an input where there is significant variation in the number of elements compared to the keys, it might be better to consider using counting sort in conjunction with radix sort, which can handle the larger key differences more effectively. As it is often used in conjunction with Radix sort, most iterations of Counting Sort are stable, I.e., the relative positions of equal value elements are preserved. (Counting sort - Wikipedia, 2022). The Best, Average and Worst times are all the same at O(n + k), where n is the number of elements, and k is the range of element values.

- How Counting Sort works

For this example, we will use the following array to demonstrate how counting sort works. [5,2,7,1,6,1,2, 4]. The first step in counting sort is to find the max value of the input array (in this instance it is 7). This information is then used to create an array with 7 elements of value 0 in it, in preparation for the keys to be held. This array looks similar to this: [0(for "1" values),0(for 2's),0(3's),0(4's),0(5's),0(6's),0(7's)]

The next step is for this array to be "filled" with the key values of the array. This means the algorithm will iterate through the input array, count the values, and store them in the output array. In this instance this will look like so: [2(for "1" values), 2(2's), 0(3's), 1(4's),1(5's),1(6's), 1(7's)]. The final stage of the algorithm is to return the values that have been counted into their keys: [1,1,2,2,4,5,6,7] (Counting Sort (With Code in Python/C++/Java/C), 2022). This process is shown below:



While this is a clear, concise, and relatively straightforward algorithm, it is easy to see how it may become unsuitable for an array such as this [1,10,100,1000] which would create 1,000 key values just to sort 4 elements. Therefore, it might be more practical to consider another algorithm in this instance.

Implementation and Benchmarking

As mentioned above, Benchmarking, or *a posteriori* analysis is an effective way to compare algorithms on the same machine. In this instance python was used to create 2 files- one which contains the functions for each sorting algorithm, and a second which takes in each algorithm and runs the benchmarking.

We are going to use the 5 sorting algorithms described above- Counting Sort, Merge Sort, Radix Sort, Bubble Sort and Quick Sort. Our aim is to create a program that can benchmark these 5 algorithms, run them with a series of array sizes, and run them 10 times each to get an average result for each size and algorithm.

This main file uses 3 "for loops"- the first one runs 5 times, once for each algorithm. The second "for loop" runs for each size of input array. The 3rd loop then runs 10 times. This ensures that we are getting an average reading of the time coming back and this is important to ensure that the results are consistent. As we are using python's *random* package in a range of 0-99 there is no way to ensure that on any certain occasion that the results may be nearly sorted, and therefore for accuracy it is important that benchmarking is done over a number of runs.

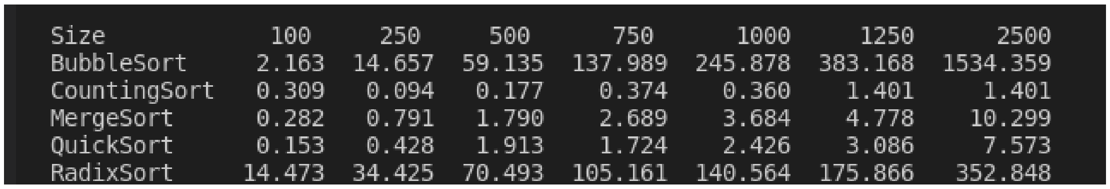
In order to time the results, we used python's *time* module, which takes a timestamp just before the algorithm runs, and then takes a second reading immediately after. The second time is then subtracted from the first to get the length of time taken to run the algorithm.

We then append the times into a dataframe, and used python's *pandas* module to get the average of each of the 10 runs. Once we have that information, it is then straightforward enough to create a CSV file, and a plot using *Seaborn* and *pyplot*, demonstrating the runtime of each algorithm with their various input sizes. Finally, we can output a table with results to allow a quick and easy comparison.

Using the CSV, we can then use excel, or pandas to create a tabular version of the output data- this is displayed below.

As the algorithms are stored in a separate program, and the sizes of inputs are in one easily accessed array, it becomes straightforward to modify the program in order to benchmark other algorithms and smaller/larger array sizes if so desired.

We can see below the output of one of the runs from the program. It should be noted at this point that as the data is randomized, and changes slightly every time the program runs, it is expected that there will be small variations to the results every time. Additionally, due to variations in compiler, machine and programming language used the results may differ quite significantly between machines.


```
Size            100     250     500      750     1000     1250      2500
BubbleSort    2.163  14.657  59.135  137.989  245.878  383.168  1534.359
CountingSort  0.309   0.094   0.177    0.374    0.360    1.401     1.401
MergeSort     0.282   0.791   1.790    2.689    3.684    4.778    10.299
QuickSort     0.153   0.428   1.913    1.724    2.426    3.086     7.573
RadixSort    14.473  34.425  70.493  105.161  140.564  175.866   352.848
```

It can be seen in the graphic above and below, that the time complexity increased as expected. As a brief summary, these are the expected Best, Worst, and Average times for each of the algorithms tested

	Best	Average	Worst
Bubble Sort	O(n)	O(n^2)	O(n^2)
Counting Sort	O(n+k)	O(n+k)	O(n+k)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)
Radix Sort	O(nk)	O(nk)	O(nk)

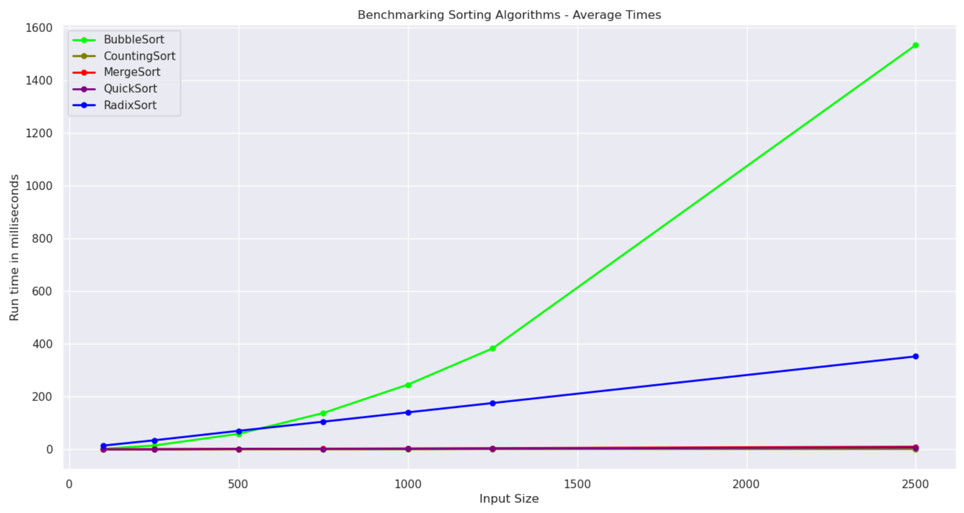
We can see, looking at the results, that Bubble Sort is by far the most inefficient of the algorithms tested.

At the lower array sizes Radix Sort has a greater time complexity than Bubble Sort. However, as soon as the input sizes increase past the 500-750 element input Bubble Sort begins to overtake and by the time the array hits 2500 it is almost 5 times slower than the closest equivalent.

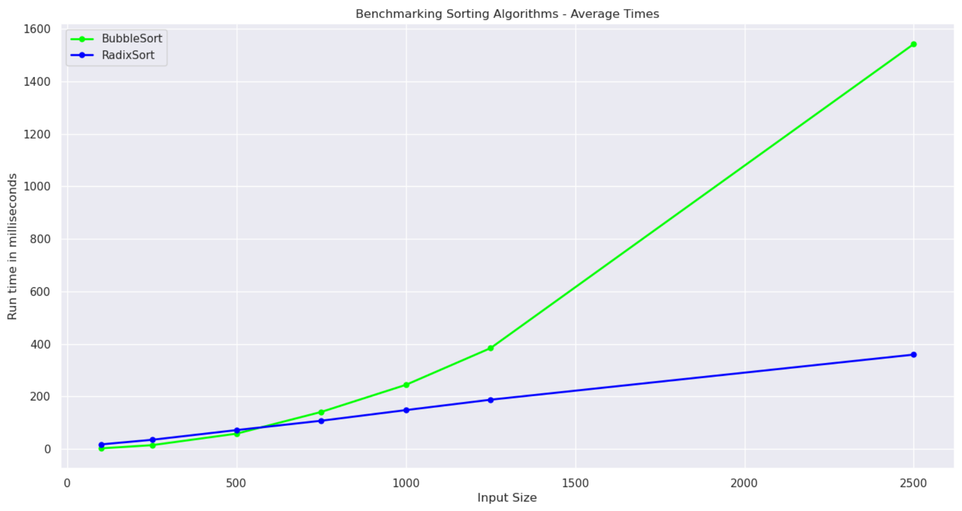
Counting Sort offers excellent efficiency in this case and performed the best, especially at the higher input figures. However, at the lower range, quicksort performed better, although this was only .15 milliseconds in the difference. As the random arrays are of a relatively small sample range (0-99) this may have an impact on how the algorithms perform. Further testing on a larger sample range and size would be recommended to further push the performance of the individual algorithms.

	Array Sizes						
Algorithm Types	100	250	500	750	1000	1250	2500
BubbleSort	2.163	14.657	59.135	137.989	245.878	383.168	1534.359
CountingSort	0.309	0.094	0.177	0.374	0.36	1.401	1.401
MergeSort	0.282	0.791	1.79	2.689	3.684	4.778	10.299
QuickSort	0.153	0.428	1.913	1.724	2.426	3.086	7.573
RadixSort	14.473	34.425	70.493	105.161	140.564	175.866	352.848

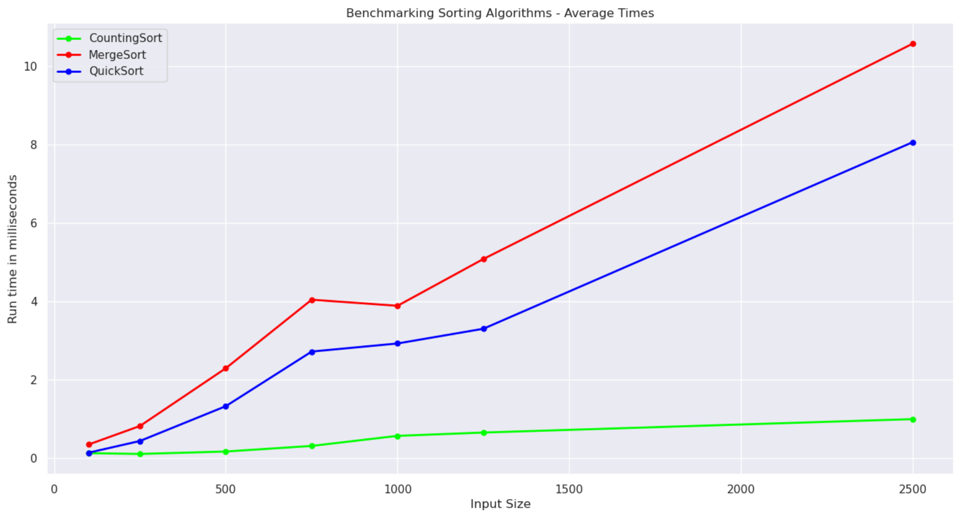
Finally, we can take a look at the data as a graphic representation. It is easy to see how the plot of Bubble Sort is increasing on a polynomial curve and with a greater input that time complexity is going to keep getting larger and larger. In fact, the growth of the Bubble Sort curve is such that it is obscuring the plots on the slower growing sorting algorithms. Therefore, we can split the chart up, and this is done below. Figure 1 has the 5 sorting algorithms laid out on the same plot. Figure 2 has the 2 fast growing algorithms- Bubble Sort and Radix Sort. Finally, Figure 3 has counting sort, and in this instance, we see an interesting phenomenon, where the time complexity actually appears to drop between the 750 and the 1000 input arrays. This could be due to the data at this point being closer to best case scenario and causing a brief improvement in time complexity results.



(fig.1)



(fig.2)



(fig.3)

Conclusion:

In this project we looked at algorithms, their importance in modern computing and how they can be used with searching algorithms very effectively. We then looked at 5 different sorting algorithms and demonstrated how these worked with diagrams. Finally, we benchmarked these 5 algorithms with a number of different inputs and demonstrated that they performed as expected.

End

References:

Dev, A., Prep, A., Learning, D., Mining, D., Learning, M., Experiences, I. and Dev Ruby React JavaScript, W., 2022. Types of Algorithms and Their Uses - Coding Ninjas Blog. [online] Coding Ninjas Blog. Available at: <https://www.codingninjas.com/blog/2020/06/04/types-of-algorithms-and-its-uses/> [Accessed 2 May 2022].

En.wikipedia.org. 2022. *Bubble sort - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Bubble_sort> [Accessed 8 May 2022].

Programiz.com. 2022. *Bubble Sort (With Code in Python/C++/Java/C)*. [online] Available at: <https://www.programiz.com/dsa/bubble-sort#:~:text=Bubble%20sort%20is%20a%20sorting,is%20called%20a%20bubble%20sort.> [Accessed 8 May 2022].

Tutorialspoint.com. 2022. *Quick Sort*. [online] Available at: <https://www.tutorialspoint.com/Quick-Sort> [Accessed 8 May 2022].

En.wikipedia.org. 2022. *Quicksort - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Quicksort> [Accessed 9 May 2022].

Programiz.com. 2022. *QuickSort (With Code in Python/C++/Java/C)*. [online] Available at: <https://www.programiz.com/dsa/quick-sort> [Accessed 9 May 2022].

HackerEarth. 2022. *Quick Sort visualize | Sorting | Algorithms | HackerEarth*. [online] Available at: <https://www.hackerearth.com/practice/algorithms/sorting/quick-sort/visualize/> [Accessed 9 May 2022].

GeeksforGeeks. 2022. *Radix Sort - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/radix-sort/> [Accessed 9 May 2022].

En.wikipedia.org. 2022. *Radix - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Radix> [Accessed 9 May 2022].

Programiz.com. 2022. *Radix Sort (With Code in Python, C++, Java and C)*. [online] Available at: <https://www.programiz.com/dsa/radix-sort> [Accessed 9 May 2022].

Error! Hyperlink reference not valid.. 2022. *Merge Sort - javatpoint*. [online] Available at: <https://www.javatpoint.com/merge-sort> [Accessed 9 May 2022].

GeeksforGeeks. 2022. *Merge Sort - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/merge-sort/> [Accessed 9 May 2022].

Tutorialspoint.com. 2022. *Merge Sort*. [online] Available at: <https://www.tutorialspoint.com/Merge-Sort> [Accessed 9 May 2022].

En.wikipedia.org. 2022. *Merge sort - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Merge_sort#cite_note-:1-7> [Accessed 9 May 2022].

GeeksforGeeks. 2022. *Counting Sort - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/counting-sort/> [Accessed 10 May 2022].

Programiz.com. 2022. *Counting Sort (With Code in Python/C++/Java/C)*. [online] Available at: <https://www.programiz.com/dsa/counting-sort#:~:text=Counting%20sort%20is%20a%20sorting,index%20of%20the%20auxiliary%20array.> [Accessed 10 May 2022].

En.wikipedia.org. 2022. *Counting sort - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Counting_sort> [Accessed 10 May 2022].

Tutorialspoint.com. 2022. *Counting Sort*. [online] Available at: <https://www.tutorialspoint.com/Counting-Sort> [Accessed 10 May 2022].

HackerEarth. 2022. *Time and Space Complexity Tutorials & Notes | Basic Programming | HackerEarth*. [online] Available at: <https://www.hackerearth.com/practice/basic-programming/complexity-analysis/time-and-space-complexity/tutorial/#:~:text=Time%20complexity%20of%20an%20algorithm,the%20length%20of%20the%20input.> [Accessed 10 May 2022].

GeeksforGeeks. 2022. *Difference between Posteriori and Priori analysis - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/difference-between-posteriori-and-priori-analysis/#:~:text=Posteriori%20analysis%20is%20a%20relative,compiler%20and%20types%20of%20hardware.> [Accessed 10 May 2022].

Programiz.com. 2022. *Big-O Notation, Omega Notation and Big-O Notation (Asymptotic Analysis)*. [online] Available at: <https://www.programiz.com/dsa/asymptotic-notations#:~:text=Asymptotic%20notations%20are%20the%20mathematical,linear%20i.e.%20the%20best%20case.> [Accessed 10 May 2022].

[duplicate], O. and Hoff, R., 2022. *Order of magnitude using Big-O notation*. [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/32364872/order-of-magnitude-using-big-o-notation> [Accessed 10 May 2022].

GeeksforGeeks. 2022. *Analysis of Algorithms | Big-O analysis - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/?ref=rp> [Accessed 10 May 2022].

Error! Hyperlink reference not valid.. 2022. *DAA Complexity of Algorithm - javatpoint*. [online] Available at: <https://www.javatpoint.com/daa-complexity-of-algorithm> [Accessed 10 May 2022].

GeeksforGeeks. 2022. *Stability in sorting algorithms - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/stability-in-sorting-algorithms/> [Accessed 10 May 2022].

Afteracademy.com. 2022. *Comparison of Sorting Algorithms*. [online] Available at: <https://afteracademy.com/blog/comparison-of-sorting-algorithms> [Accessed 10 May 2022].

Medium. 2022. *Benchmarking Sorting Algorithms in Python*. [online] Available at: <https://medium.com/geekculture/benchmarking-sorting-algorithms-in-python-437ba864b799> [Accessed 11 May 2022].

Harel, D. and Feldman, Y., 2012. *Algorithmics*. Berlin: Springer.

Tuckfield, B., 2020. *Dive into algorithms*. San Francisco: No Starch Press.

Techie Delight. 2022. *In-place vs out-of-place algorithms | Techie Delight*. [online] Available at: <https://www.techiedelight.com/in-place-vs-out-of-place-algorithms/> [Accessed 13 May 2022].