Computational Thinking with Algorithms: Project 2021

Eoin Lees – G00387888 - Higher Diploma in data analytics

Introduction

Sorting is the action of "arranging a collection of items according to some pre-defined ordering rules." (Mannion, 2021)

In computing it is an important concept as a large porting of computing work is spent on sorting. It has therefore since the beginning of modern computing been an important filed of study for optimization. By sorting information in advance, many tasks can be simplified greatly and take up less computational power. It is often an important part of other algorithms too, so its optimization to a specific task is important.

Sorting algorithms in computing began in 1945 with the development of merge sort by John von Neumann. Multiple other sorting methods were also developed in the early days of computing e.g Radix sort (1954), Counting sort (1954), Quick sort (1962) etc.

"To sort a collection, you must reorganize the elements A such that if $A[i] < A[j]$, then $i < j$." (G Heineman, 2016)

If elements are the same then they must be ordered alongside each other in the resulting sorted array.

The final sorted array must be a permutation of the original array of elements.

Comparator Functions

The elements in an array are sorted by using comparator functions in order to determine their final order. This is done by taking two elements of any type and comparing one to the other.

“Any two elements p and q in a collection, exactly one of the following three predicates is true: p = q, p < q, or p > q.” (G Heineman, 2016)

Which input, p or q should appear first is decided according to predefined definition of comparator function.

This project is focused on integer sorting. So greater than or less than will be used.

It is possible to sort strings using lexicographical ordering. i.e how letters appear in alphabet, or items appear in dictionary.

Comparison-based-sorts

“A comparison sort is a type of sorting algorithm which uses comparison operations only to determine which of two elements should appear first in a sorted list.” (Mannion, 2021)

A sorting algorithm is called ***comparison-based*** if the only way to gain information about the total order is by comparing a pair of elements at a time via the order ≤. (Mannion, 2021)

In comparison-based sorting no algorithm that sorts by this method can do better than n log n performance in the average or worst cases. This is reflected in table 1.

There are many sorting algorithms that are in this category (e.g. Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, Quicksort, Heapsort).

They are the most widely applicable sorts for a diverse type of input data.

There are no assumptions made about the data. They compare all elements against each other.

O(n log n) is the ideal worst case scenario for a comparison-based sort.

Non-comparison-based-sorts

“Under some special conditions relating to the values to be sorted, it is possible to design other kinds of non-comparison sorting algorithms that have better worst-case times (e.g. Bucket Sort, Counting Sort, Radix Sort)” (Mannion, 2021)

These methods can achieve better “worst case” times as shows in the table below however they are not as stable.

O(n) time is possible if assumptions are made about the data. This linear result comes from not having to compare every element against each other.

Examples include: Counting sort, Bucket sort and radix sort.

O(n) is the minimum sorting time possible since each element must be examined at least once.

Stability

Stability in a sorting algorithm as it is important to preserve the order of already sorted data. This is a desirable property in sorting algorithms.

"When the comparator function determines that two elements, *ai* and *aj*, in the original unordered collection are equal, it may be important to maintain their relative ordering in the sorted set–that is, if $i < j$, then the final location for *ai* must be to the left of the final location for *aj*." (G Heineman, 2016)

Sorting algorithms that guarantee this property are considered to be *stable*.

Example of stable sorting algorithms are Merge Sort, Insertion Sort, Bubble Sort, and Binary Tree Sort.

In-place sorting

Sorting algorithms have different memory requirements, which depend on how the specific algorithm works. A sorting algorithm is called in place if it uses only a fixed additional amount of working space, independent of the input size.

Other sorting algorithms may require additional working memory, the amount of which is often related to the size of the input n.

In place sorting is a desirable property if the availability of memory is a concern (Mannion, 2021)

Performance and complexity

Deciding which algorithm to use in a specific situation is made easier by understanding how the algorithms perform under specific circumstances.

The complexity of an algorithm can be analysed in both the time required to complete the algorithm and how much memory it requires to run. These are known as time and space complexities.

This analysis thankfully has previously been studied in most common sorting algorithms used today. The results of these analysis will be attached to each algorithm and described in three separate use cases; The worst case, the average case and the best case.

Worst case

"Defines a class of problem instances for which an algorithm exhibits its worst runtime behaviour." (G Heineman, 2016) When studying this case it is best to try and identify the properties that cause this to occur and try and avoid them when running the algorithm in real world scenarios. This should prevent the algorithm from running in the worst-case scenario.

Average case

"Defines the expected behaviour when executing the algorithm on random problem instances." (G Heineman, 2016)

This describes the average use case for this algorithm. It should match this case in the vast majority of instances. You should expect this sort of result when planning to use the particular algorithm .

Best case

"Defines a class of problem instances for which an algorithm exhibits its best runtime behaviour." (G Heineman, 2016)

Here the algorithm has to do the least amount of work possible in the shortest amount of time possible. This may be down to the input being already partly or mostly sorted. In reality, the best case rarely occurs.

Figure 1: Overview of sorting algorithms - Shows an overview of a number of common sorting algorithms along with their time and space complexities and whether they are stable or not. A simple table like this one can be enough to make a quick decision on which algorithm to use.

Algorithm	Best case	Worst case	Average case	Space Complexity	Stable?
Bubble Sort	n	n^2	n^2	1	Yes
Selection Sort	n^2	n^2	n^2	1	No
Insertion Sort	n	n^2	n^2	1	Yes
Merge Sort	$n \log n$	$n \log n$	$n \log n$	$O(n)$	Yes
Quicksort	$n \log n$	n^2	$n \log n$	n (worst case)	No*
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No
Counting Sort	$n + k$	$n + k$	$n + k$	$n + k$	Yes
Bucket Sort	$n + k$	n^2	$n + k$	$n \times k$	Yes
Timsort	n	$n \log n$	$n \log n$	n	Yes
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	No

*the standard Quicksort algorithm is unstable, although stable variations do exist

COMP08033 Computational Thinking with Algorithms, Patrick Mannion GMIT

Figure 1: Overview of sorting algorithms

By understanding how an algorithm performs in each of these situations you can decide whether it is appropriate for use in your specific situation.

We compare algorithms by evaluating their performance on problem instances of size n. This methodology is the standard means developed over the past half century for comparing algorithms.

This allows us to determine which algorithms scale best to solve problems that are much larger and could take a considerable amount of time. The running time is looked at in relation to the provided input.

The following classifications are used, they are ordered by decreasing efficiency:

- Constant: O(1)
- Logarithmic: O(log n)
- Sublinear: O(n^d) for d < 1
- Linear: O(n)
- Linearithmic: O(n log n)
- Quadratic: O(n^2)
- Exponential: O(2^n)

“When evaluating the performance of an algorithm, keep in mind that you must identify the most expensive computation within an algorithm to determine its classification” (G Heineman, 2016)

This means that if two or more methods are use you must calculate the overall performance based on which classification is the worst performing.

Sorting Algorithms

Bubble Sort

Bubble sort takes its name from having larger values in a list “bubble up” to the end of the list as the sorting takes place. It was first analysed formally as early as 1956. It is a comparison-based sorting method.

It has a time complexity of n (linear) in best case and n2 (quadratic) in the worst and average cases. (table 1)

It is an in-place sorting algorithm. It uses only the size of the input data set plus a constant amount of additional working space to function. The additional working space is just a few variables required to make the swaps of the elements in different places. A temporary variable is used here.

It is a slow sorting method. It is impractical for most problems. It only really works well on small data sets, or on data that is already nearly sorted.

Procedure:

- Compare each element to the element on it right (exclude last element)
  - If out of order, swap. Then repeat to next element.
  - Largest element ends up at end of list in its final position.
- Repeat process (exclude last two elements)
  - If out of order, swap. Then repeat to next element.
  - Second largest element ends up in its final position.
- Continue until there are no more remaining unsorted elements on the left.

The algorithm assumes every element on the left is unsorted, and the right is sorted. It then has two separate arrays of data. Sorted and un-sorted.

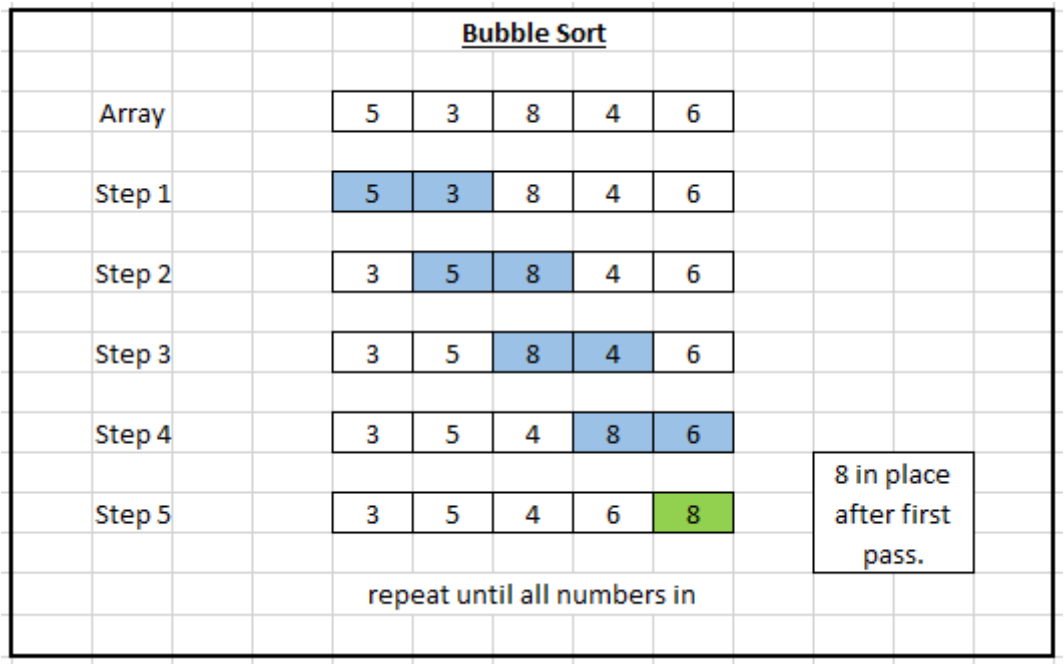

Figure 2: Bubble Sort

Merge Sort

Merge sort is an efficient comparison sort. It was proposed by John von Neumann in 1945. Is a recursive algorithm. It repeatedly calls itself until a base case is reached rather than using for loops or while loops.

It has a worst-case running time of 0(n log n). Its best and average cases are also n log n.

It is a good all-around performance algorithm and therefor has many use cases. It is also a stable algorithm.

There are versions of merge sort that are particularly good for sorting data with slow access times, e.g. data that cannot be held in RAM. Merge Sort is particularly well-suited for sorting data in secondary storage. (G Heineman, 2016)

Merge sort must take the least time that is linear in the total size of the two lists in the worst case. Every element must be looked at so the correct order can be determined.

Procedure:

- When called:
  - If size of input is 0 or 1, return.
  - Otherwise separate input array into two arrays approximately equal in size
- Apply merge sort to left and right sides.
  - Repeat until every element is 0 or 1.
- Call merge
  - Positions swap relevant to one another in sorted position.
  - Combine sub arrays
- Call merge
  - Positions swap relevant to one another in sorted position.
  - Combine sub arrays
- Sorted array achieved.

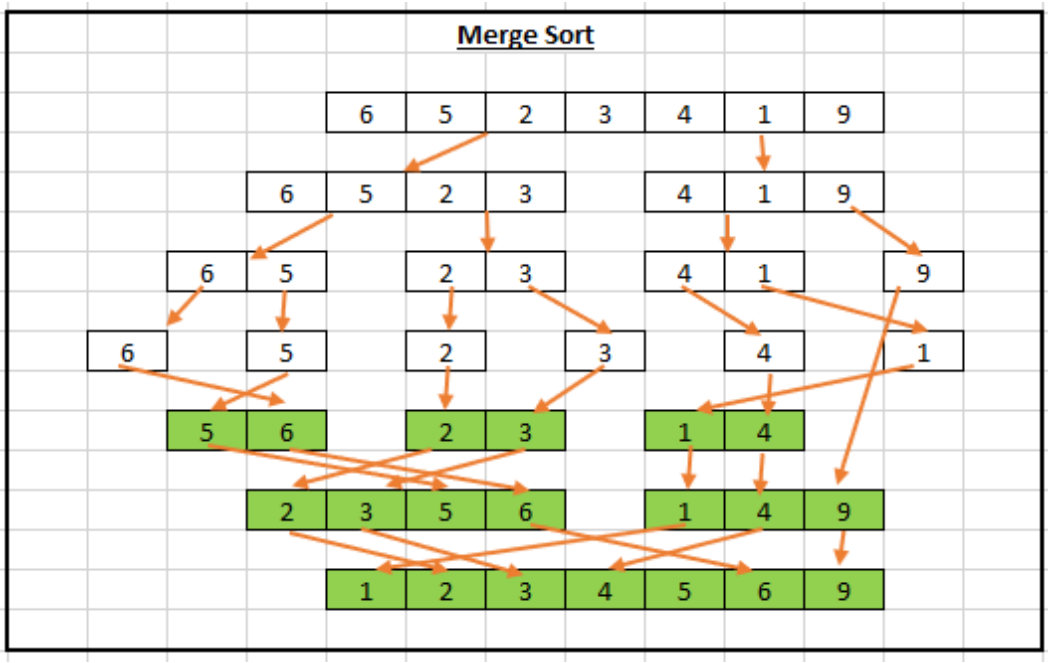

Figure 3: Merge Sort

Counting Sort

A non-comparison-based sort. It was proposed by Harold H. Seaward in 1954, who also proposed radix sort. It allows a collection of items to be sorted in close to linear time.

Certain assumptions are made in order to achieve this. These assumptions are also limitations on the type of input data the sorting algorithm can handle.

The assumption with counting sort is: assuming an input of n, where each item is a non-negative integer key with a range of k. If using zero-indexing, the keys are in the range [0,…,k-1]. (Mannion, 2021)

The best, worst and average case time complexity is n + k, the space complexity is also n + k.

The running time advantage comes at the cost of the algorithm not being as widely applicable as comparison-based sorts. It is more complex to use with data containing strings. A mapping system mapping the strings to an integer would need to be put in place prior to running the sort.

It is a stable algorithm if used correctly.

Procedure:

- The key range k of the input array is determined.
- Initialise array (the count array) that is the size of k. This is used to count the number of times that each key value appears in the input instance.
- Initialise the result array that is of size n. This stores the sorted output.
- Iterate through the input array and record the number of times each input value occurs.
- Construct the sorted results array from count array as shown in Figure 4: Counting Sort.

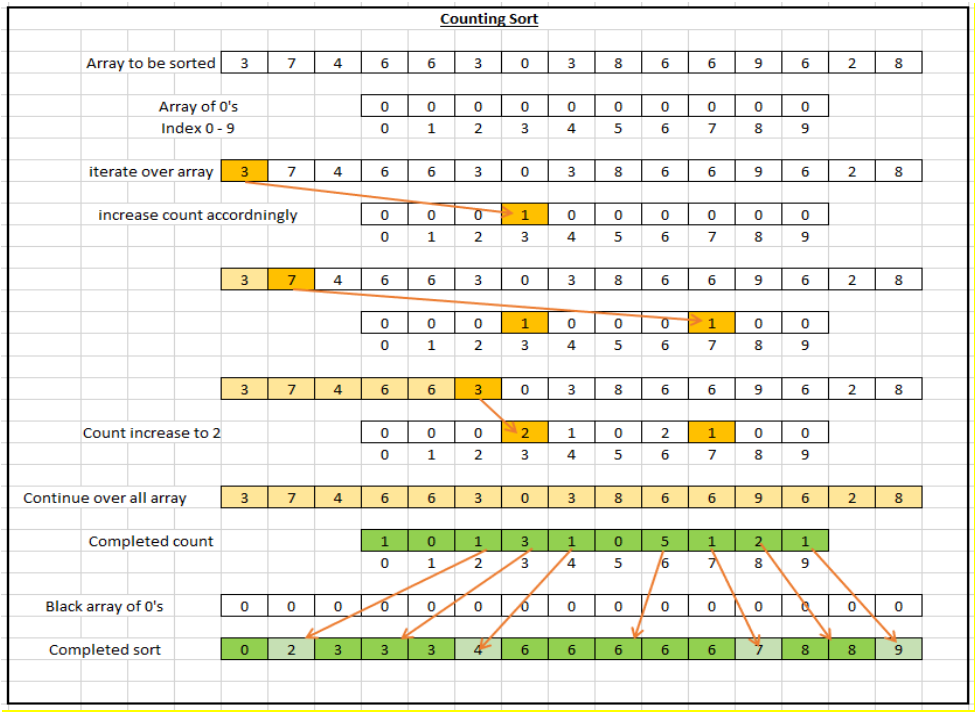

Figure 4: Counting Sort

Insertion Sort

A simple comparison-based sorting algorithm. It is similar to the method used usually by card players sorting cards. It is intuitive to understand. It is easy to implement, is a stable sorting algorithm, it works in-place (efficient with memory) and is very good for small and partially sorted lists.

In best case scenarios it runs in n + d time, where d is the number of inversions in the input instance.

It is very inefficient for large lists that are randomly ordered. It is an iterative approach. It splits the list into sorted and unsorted sub lists. (head and tail)

Complexity:

The total number of data comparisons is the number of inversions d plus n-1. On average

Procedure:

- Starting on the left of the array, the key is set as the element at index 1.
    - Any elements on the left greater than the key get moved right by one position and the key gets inserted.
- Set the key as the element at index 2.
    - Any elements on the left greater than the key get moved right by one position and the key gets inserted. In Figure 5: Insertion Sort this moves 3 to the left of 5.
- Set the key as the element at index 3.
    - Any elements on the left greater than the key get moved right by one position and the key gets inserted.
- Continue until the key is set at n-1 index.
    - Any elements on the left greater than the key get moved right by one position and the key gets inserted.
- The array is sorted.

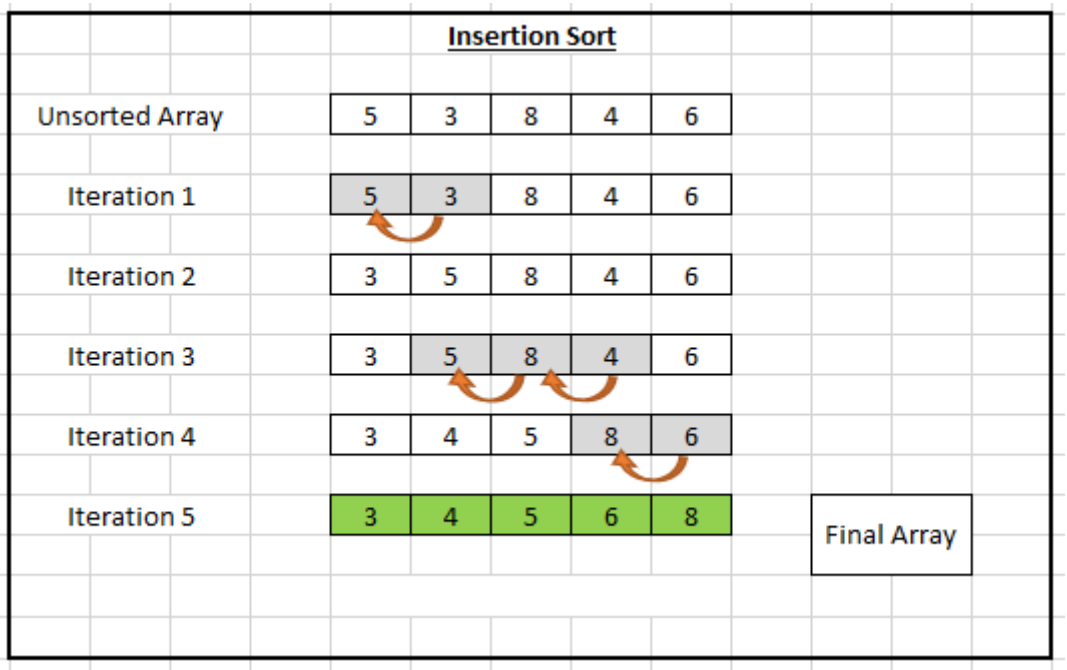

Figure 5: Insertion Sort

Quick Sort

One of the most important algorithms developed. It was developed in 1959 by C.A.R Hoare. It is a recursive/divide and conquer algorithm. It is on average one of the fastest known sorting algorithms in practice.

It is not a stable algorithm, however stable modifications do exist.

The performance is n^2 in the worst case, however this is rare. The average and best cases are n log n.

If the pivot chosen is poor then the n^2 performance may happen. E.g if the first or last element is chosen in an array where the data is already nearly sorted.

The median element is usually the best choice, as it makes it so that a fairly equal distribution will occur in the left and right arrays.

Its memory usage is O(n) – linear memory usage in standard usage. Some variations exist with O(n log n) memory performance.

Procedure:

- Pick an element from the array – the "pivot" (in Figure 6: Quick Sort the pivot is the right most entry)
- Partitioning – takes every array element that has a value less than pivot and positions them to the left of the pivot. It also takes every array element greater than the pivot value and positions them to the right. The pivot is now in its final position with two subarrays on either side.
- Recursion – the above steps are then applied to (G Heineman, 2016) each sub array. It is repeated until a base case is reached: a subarray of length 1 or 0.

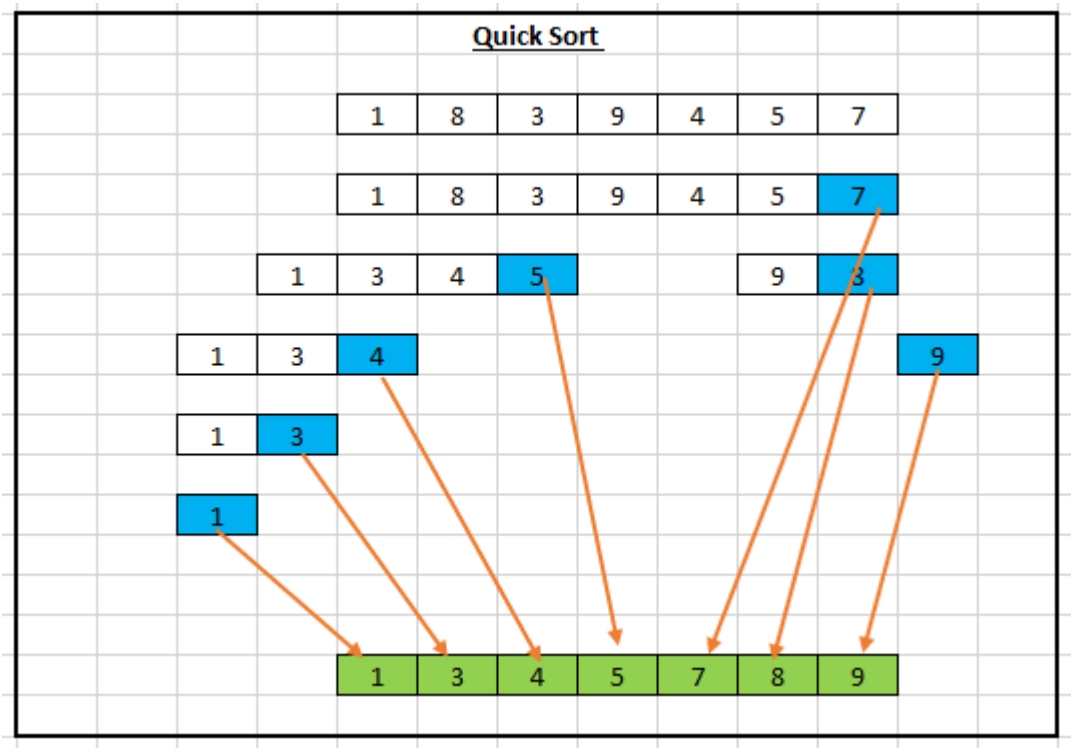

Figure 6: Quick Sort

Implementation and Benchmarking

Implementation

The results of this project are produced by running the program.py program in python. It will produce a table of results shown below in *Figure 7: Table of results for 10,000 inputs* and also if specified in the code *Figure 8: Plot of results*.

The algorithms were each defined as functions separately. The source for each algorithm is stated in the comments. They may also be ran in individual instances if required.

A function was also created called `randomArray(n)` which provided the array to the functions. Each array produced a 'n' number of integers between 1 and 100. The 'n' number was specified when calling the function. This allows the function to be passed into the sorting algorithm function while specifying an array of 'n' values to be sorted.

A globalTimeArray is created to hold all of the average time data that results from the program.

The `runTest(numberOfTests, inputSize)` function is the function that allows the program to achieve the desired results as specified in the project brief.

It runs as follows:

- Takes two inputs from function call.
    - numberOfTests = how many times the function would run through each sorting algorithm.
    - inputSize = 'n' value for `randomArray(n)`
- Initiates blank time array to record each average time value for each input size.
- Initiates an individual algorithm time array to hold the times taken for each iteration of the algorithm function.
- Initiates a for loop to run the algorithm the number of times specified (in this case 10)
    - Creates a random array using the `randomArray(n)` function
    - Starts the python timer
    - Calls the sorting function
    - Ends the timer
    - Calculates the time taken for the function to run.
    - Converts this value to milliseconds and rounds it to 3 decimal places
    - Appends this value to average time array
- The 10 time values in the average time array are then divided by 10 (or whatever value is given) to calculate the average time taken to run the sorting algorithm.
- This final average time value is appended to the time array for this input value.
- The same procedure is followed for each sorting algorithm.
- The final time array is appended to the global array.
- The function then runs again with the next inputSize value.
- It is completed once all of the input values are run through the sorting algorithms.

This function is called in the '__main__' function. This is the function that runs once the program is executed in python.

It runs as follows:

- Initiate timer to time entire program.
- Set size variable array.
    - For testing a smaller array of variables is used.
    - For getting the results a larger array is used.
- Iterate the 'runTest(numberOfTests, inputSize)' function with the 'numberOfTests' = 10 and the 'inputSize' = i in the size array.
- Set up dataframe to organise results of function
    - sortTypes holds the headings for each sort type
    - df holds the globalTimeArray with the time taken to run each iteration of each input size for each sorting algorithm.
- End the total program timer.
- Print the data frame in a neat table.
- Print the time taken to run.
- Plot the results in a seaborn line plot.

The program will follow these steps each time it is called. Its variables can be changed when needed.

Creating it as a function allows it to be reused to test other sorting algorithms if required in the future.

Benchmarking

"Benchmarking or a posteriori analysis is an empirical method to compare the relative performance of algorithm implementations" (Mannion, 2019)

This means that bench marking works using experimental measurements. Experimental data, e.g running times, can be used to verify the initial analysis of the algorithms.

It is important to note that software and hardware factors can influence the performance of an algorithm and benchmarking process. Therefore, multiple statistical runs were undertaken using the same conditions (laptop and background programs running) in order to obtain a reliable set of benchmarked average results. The python program is provided also so the same results can be verified independently.

As described in the implementation section above the `time.time()` function in python is used to get a specific time to measure the algorithms at the start and end of their function call. This time is represented as "the number of seconds that have elapsed since midnight on January 1st 1970" (Mannion, 2019) or Epoch time. An epoch is an instant in time chosen as the origin of a particular calendar era.

By running each input instance of a sorting array 10 times with different generated random array each time we can take an average of the time taken. This gives us a good, benchmarked time value to compare out initial investigation to in the conclusion. It also allows us to investigate further by

changing the variables in the input to see if it effects the results, however this is not necessary in this project.

Results

The results of running the algorithm are shown in "Figure 7: Table of results for 10,000 inputs" and plotted in Figure 8: Plot of results and Figure 9: Log scale Plot.

These results show the time taken for each algorithm to run as specified in the implementation section above. The input values are shown on the left of figure 7, and the time taken in milli seconds is shown for each sort; Bubble sort, Merge sort, Counting sort, Insertion sort and Quick sort.

The line plots show this data plotted. Figure 8 shows this data plotted linearly, while figure 9 shows the y axis on a log scale. It allows the data to be read more clearly for the quicker working algorithms. Each individual algorithm is stated in the legend according to the colour on the plot it represents.

Average time taken to run each sorting algorithm in milliseconds

	Bubble Sort	Merge Sort	Counting Sort	Insertion Sort	Qucik Sort
100	0.921	0.300	0.200	0.848	0.400
250	10.841	1.200	0.300	4.800	0.800
500	35.112	2.004	0.200	12.201	1.499
750	68.722	3.215	0.599	28.286	2.260
1000	113.752	4.504	0.755	51.833	3.650
1250	173.950	5.648	0.800	80.790	4.905
2500	697.503	11.921	1.662	332.926	13.595
3750	1600.631	19.008	2.400	753.790	27.298
5000	3217.585	25.911	3.027	1339.052	42.632
6250	4562.753	33.077	3.806	2141.389	62.937
7500	6458.208	40.917	4.800	3221.998	85.942
8750	8657.143	48.818	6.199	4069.129	122.145
10000	11450.725	55.997	6.692	5456.171	147.478

Program took 556.009 seconds to run

Figure 7: Table of results for 10,000 inputs

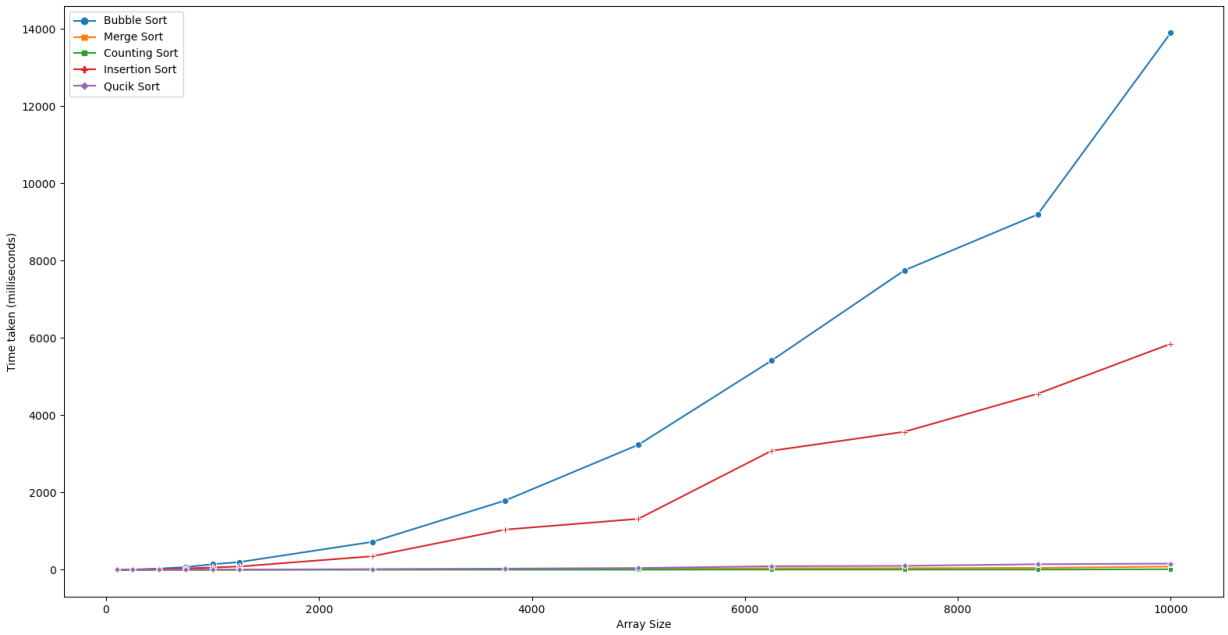

Figure 8: Plot of results

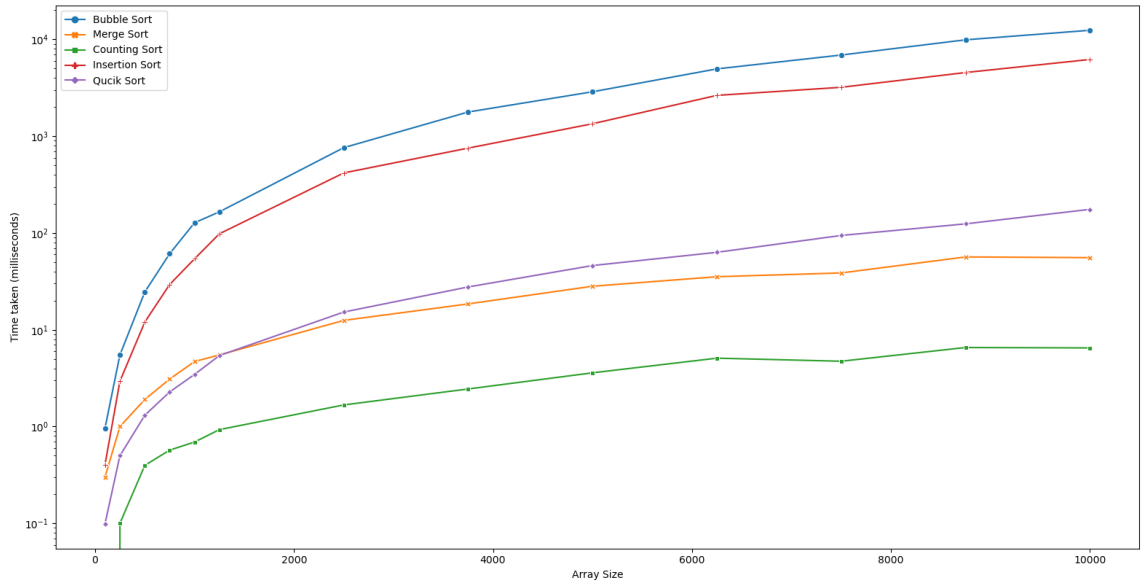

Figure 9: Log scale Plot

Discussion

The results shown above can be compared to Figure 1: Overview of sorting algorithms to show if they were similar to what was expected given the time complexity of each input instance.

In each algorithm it is possible to see how they change with larger input.

The results of each are discussed as follows:

Bubble Sort

As expected from studying Figure 1: Overview of sorting algorithms Bubble sort performed slowest.

Looking at Figure 7: Table of results for 10,000 inputs it is immediately slower than all the other algorithms. This isn't immediately apparent from studying figure 1, as the best case and average cases are the same at n and n^2 however after undertaking some research it is clear that this algorithm is historically known as a slow algorithm. They both do follow a quadratic curve as shown in figure 7.

Merge Sort

In merge sort, worst case and average case has same complexities O(n log n). Plotting on a linear scale it is difficult to differentiate between it, Quick sort and Counting sort. However if we switch to the log scale plot in figure 8 we can see that while it function worse than both quick sort and counting sort in lower inputs, it does however end up quicker than quick sort at larger inputs.

Counting Sort

Counting sort has best, worst, and average cases of n + k suggesting that it will be quicker than all studied. This is in face true with counting sort finishing quickest in all input instances in this test. It is very efficient even at much larger inputs with its 10,000 input finishing quicker than bubble sort at 250 input.

Insertion Sort

Insertion Sort is expected to follow a quadratic curve when plotted as its best, worst and average cases are n, n^2 and n^2. It's linear best case scenario could be a reason for it performing better than bubble sort.

Quick Sort

The worst case complexity of quick sort is n^2. However, both the best and average case is n log n. These better cases are apparent in lower number of inputs as it performs better than merge sort until around 2000 input instances. It then runs slower than both merge sort and counting sort.

Conclusions

Following research and experimentation it is clear that using a counting sort algorithm is the best-case scenario of all of the algorithms studied in a use case that involves an array of integers.

References

G Heineman, G. P. S., 2016. Algorithms in a Nutshell. In: *Algorithms in a Nutshell 2nd Edition.* s.l.:O'Reilly.

Mannion, P., 2019. Benchmarking Algorithms in Python. In: *COMP08033 Computational Thinking with Algorithms.* s.l.:GMIT.

Mannion, P., 2021. Sorting Algorithms Part 3. In: *COMP08033 Computational Thinking with Algorithms.* s.l.:GMIT.

Mannion, P., 2021. Sorting Algorithms, Part 1. In: *COMP08033 Computational Thinking with Algorithms,.* s.l.:GMIT.