# COMPUTATIONAL THINKING WITH ALGORITHMS (46887)

## Sorting Algorithm Report

Student Name: Keith Ryan

Student ID: G00387816

Student Email: G00387816@gmit.ie

# Table of Contents

# Introduction:

Sorting is an important topic that we encounter daily even though in general it probably is not given much consideration, its usefulness and the important role it plays in our day to day lives can't be underestimated. Having things sorted, allow us to easily find what we are looking for, when looking at numeric values we know the smallest and largest value at a glance, when looking up definitions for words in a dictionary we can easily find a word we're looking for in amongst many thousands because we know it is sorted alphabetically. [1]

## Sorting in General:

In essence the things you are sorting will be numeric or character based and can be sorted ascending or descending depending on your need. It can be done categorically, for example, sorting a set of tasks based on difficulty, in this situation you are defining categories which have some relation to one another e.g., easy is lower on the scale than medium which is lower on the scale than hard. Similarly, colour might be a category to sort over how someone might want colours sorted will vary, some might want colours sorted alphabetically, others may want them sorted by hue while someone else might want them sorted by their preference, ultimately it really depends on the situation and the type of data for how things should be sorted, especially with categorical data. [1]

While sorting in general is something we may not think about, in the world of computing sorting is a topic that gets much attention and is an ongoing area of research [2]. Sorting in computing is used for lookups/search efficiency such as for finding a movie in a database, merging arrays together if two different arrays need to be combined into one, allowing items to be processed in a defined order.

## Sorting in Computing:

For performing sorting with computing there are many different sorting algorithms which tend to have advantages in specific areas, some may perform very well on nearly sorted arrays while others may perform best when dealing with very randomly distributed arrays [2]. As there are many kinds of sort algorithms there are important characteristics to consider when trying to pick the optimal algorithm; how the sort is performed, length of time it takes to run, amount of memory it occupies on the hardware [3]. Which are all very important factors and will vary depending on the use case, for example it may very well be desirable to have a long running sort so long as the memory used is very low [2].

# Evaluating Sorting Algorithms

## Time and Space Complexity

With algorithms of any kind we need to have a way to measure how fast they are to run and how much memory they consume when running. The term for how much memory a search algorithm requires is called it's "space-complexity", for the run-time it's termed "time-complexity", these are measured not in terms of exact units like seconds or bits but using big O notation [4]. The rationale behind this is that in general, the length of time that an algorithm will take to run will vary based on the underlying hardware, so instead of determining time-complexity around X seconds to run, it is instead determined using the number of operations required relative to the input. For example, if an array of size n is being sorted and

it takes the algorithm a single for loop over the size of the array then that algorithm would have a time-complexity of O(n) [5].

Space-complexity refers to the memory used by the algorithm, like time-complexity it is also measured with big O notation and essentially it is the amount of memory taken up while the algorithm is running [6]. Based on this it is important to consider how much memory may be available on the hardware which is ultimately going to be running the algorithm and so will be especially important on devices which have relatively low memory.

Both time and space complexity are important measures of sorting algorithms [2], and the figure for each that generally gets referred to is the worst-case [7]. What the worst-case is will depend on the algorithm in question, and it ties into what the algorithm is good at doing, so if an algorithm is exceptional at sorting a nearly sorted array then it's worst case will be providing it with a completely unsorted array. Best-case and average-case are other useful measures but not as important as worst-case [7]. Best-case is determined based on the absolute ideal situation for running an algorithm and so its relevance is only when dealing with an optimal input to the sorting algorithm, for example with bubble sort the best-case is being provided an already sorted list [8]. Average-case is more useful as it describes in general the performance you could expect with a typical set of input's, that is where the inputs are not really advantageous or disadvantageous to the algorithm [9].

## Sorting In-Place:

Having an algorithm sorted in-place simply means that the array that is passed into the algorithm is the same object returned [10], in Python this means that you do not necessarily need to return anything from the sorting algorithm function as the array itself has been modified.

Whether the algorithm is sorted in-place will generally have an impact on increasing the space-complexity [10], if not in-place then additional space is needed to store a new sorted version of the array and the additional space required is depends on the size of the input.

Sorting an algorithm in-place may increase the time-complexity and typically for embedded systems where memory will be an issue a trade-off between reducing space-complexity and increasing time-complexity might be sought.

Depending on the sorting algorithm it may be possible to have in-place and non in-place version of the algorithm and as described these different versions will have advantages in the in-place versions with time-complexity when and with space-complexity when not in-place [11].

## Comparison and non-comparison based sort algorithms:

# Sorting Algorithms:

In this section the different sorting algorithms will be introduced with an overview of their function, visualised using bespoke diagrams describing their operation and have their space and time complexity discussed. Each algorithm will be explained how it works and how it handles different kinds of inputs.

The different types of sorting algorithm that were looked at and the specific ones chosen for this project are listed below, before expanding on each in more detail in their own sections.

1. A simple comparison-based sort - Bubble Sort

2. An efficient comparison-based sort – Quicksort

3. A non-comparison sort - Counting Sort

4. Any other sorting algorithm of your choice – Insertion Sort

5. Any other sorting algorithm of your choice – Timsort

## 1. Bubble Sort

### Overview:

Bubble Sort is a very simple sorting algorithm, it works by looping through the input array and comparing the current index element against every subsequent index, swapping their positions if the indexes value is smaller than the current one. This process repeats, iterating to the next index and comparing against each subsequent index until the entire array is sorted, after each iteration of the loop the number of indexes to compare against reduces and so the final loop can be skipped as the array is already sorted at this point. It is not a very practical algorithm due to its time complexity which will be discussed below but thanks to its relative simplicity, it is a good example for teaching and as an intro into sorting algorithms as a concept [8].

# Diagram:

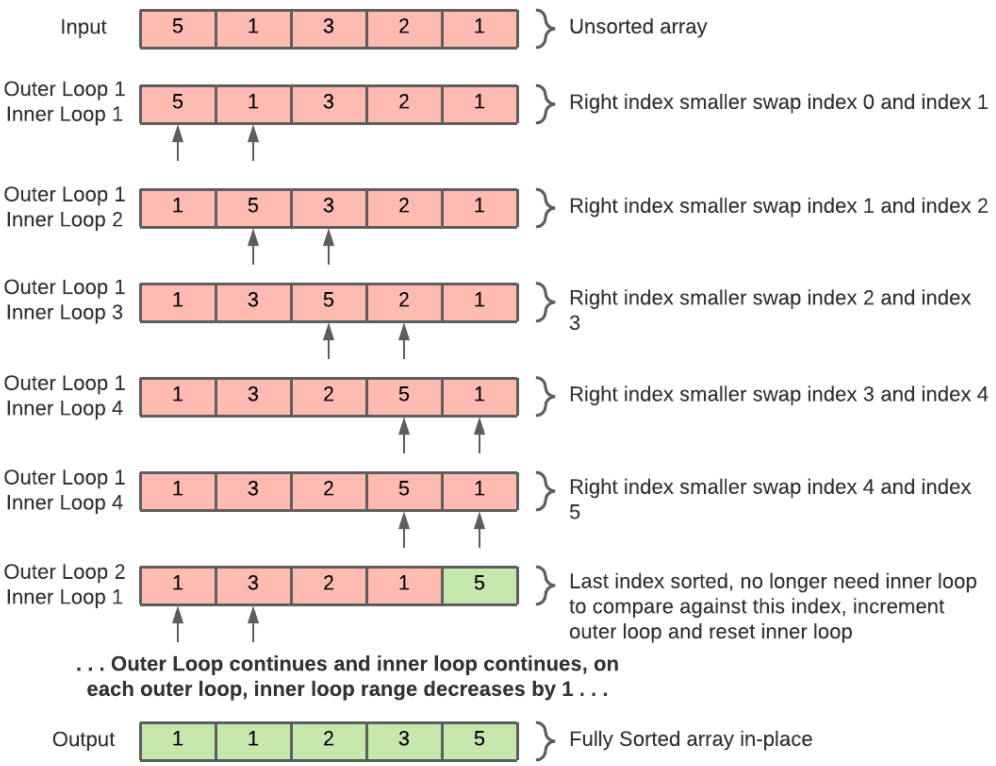


**Figure 1: Bubble Sort bespoke diagram**

# Space and Time Complexity:

The bubble sort function I implemented is performed iteratively and not recursively, It has a space-complexity of O(1) or 1 as it performs it's sort on the array in-place.

For Time Complexity bubble sort is in best-case O(n) – when the input array is already sorted.

On average it has $O(n^2)$

It has time complexity in its worst case of $O(n^2)$ – when the input array is totally unsorted. [3]

# Explanation:

The implementation for bubble sort used was based off example found from a realpython article on sorting [12].

How Bubble sort works is it makes use of loops to control stepping through the array, the outer loop iterates through each index of the array with the inner loop iterating over every index up till what is sorted. Per iteration of the outer loop the right most unsorted index gets sorted, with the largest remaining unsorted value in the array 'bubbling' up to the top. For a small performance improvement can take advantage of the fact that the largest value that is unsorted in the current loop of the array will be sorted by the end of that loop, which means we no longer need to compare against a sorted index once it has been sorted, so this means that we can reduce the number of iterations on the inner loop on each completed loop of the outer.

# 2. QuickSort

## Overview:

Quicksort is a recursive sorting algorithm and is a more practical algorithm (due to it having better space and time complexity in general) than the Bubble Sort and is still often used. At a high level it works using a pivot point (this can be picked a number of ways but for my implementation it is the first element in the array) to split the array to be sorted into left and right halves which are less than and greater than the pivot elements value. It then recursively does the same operation on each half until there are only 1 or 2 elements, at which point the array has been sorted [13].

## Diagram:

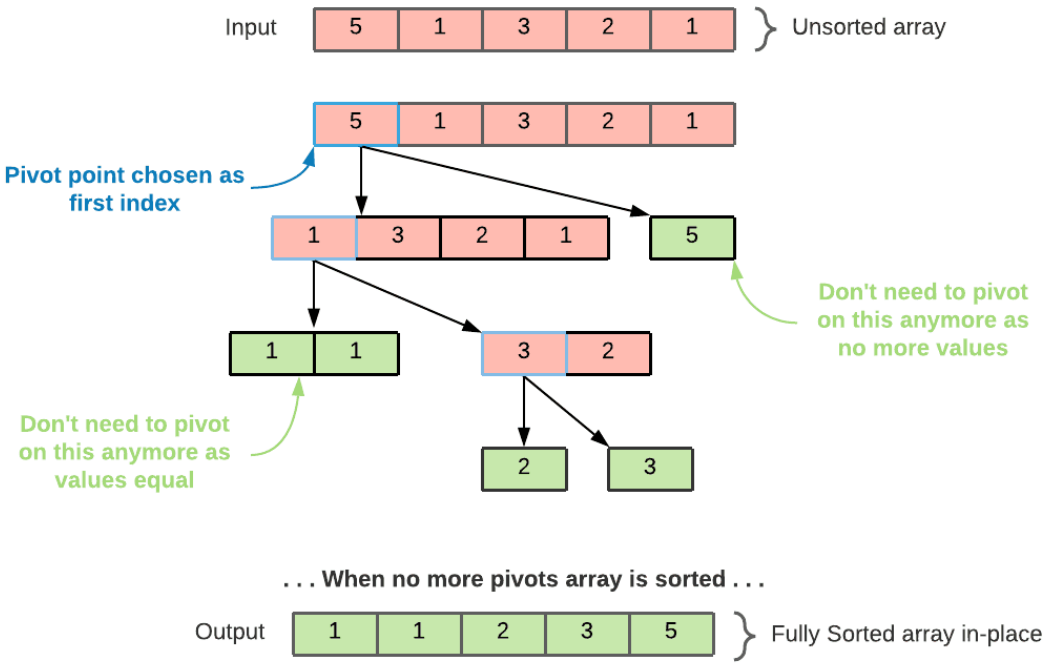


**Figure 2: Quicksort bespoke diagram**

## Space and Time Complexity:

The Space Complexity for this algorithm is O(n), meaning it takes more memory than the simple Bubble Sort to run, this is due to the recursive nature of quicksort, while the array is modified in place and so no auxiliary data structures are required, the stack is increasing on each recursive call.

Time Complexity in the best case is O(n log n).

Time Complexity in the average case is O(n log n).

Time Complexity in the worst case is $O(n^2)$. [3]

Quicksort's time complexity is the same on average as it is in the best case, which makes it a generally good and reliable method for sorting with its worst case happening only rarely, such as when the array

has already been sorted, this makes it a far more efficient algorithm especially on larger arrays than something like Bubble Sort [13].

## Explanation:

The implementation of Quicksort used was based off example code from askpython article on Quicksort [14].

The implementation of Quicksort used in my code has two functions, pivot and quicksort.

Pivot takes in three inputs, the array to be sorted and start and end which determine the scope within the array to look at. On a first call of pivot start and end will be the first and last elements of the array but on subsequent calls this will change. How the pivot function works is it chooses an element in the array between start and end that will be the pivot value (in my case the first element) and using this value it sorts the other values between start and end in the array into low and high halves, those lower than the pivot value go into the low and those higher go to the high, this is all done in-place and so it swaps elements within the array to achieve this. This function returns the index at which the pivot point is.

Quicksort function similarly takes in three inputs, the array to be sorted and start and end, again start and end determine the scope within the array to look at, however one part that differs here are that I have the quicksort function set up so that start and end are optional arguments and so if not specified supply default values (I have done so to allow reusability of my timer function which will be discussed in the later "Implementation and Benchmarking" section.). The function works by first checking start is greater than or equal to end, this is the base case and it returns True to denote that it has finished sorting. After this it calls the pivot function to sort the array into high and low and correctly position the pivot element, following this, the function recursively calls itself twice for the first it sends the array and the start and end for the left or low half of the array and for the second call it sends the array and the start and end for the right or high half of the array. These calls perform the same steps but on increasingly smaller parts of the array until each recursive call hits their base case and have sorted their respective parts of the array.

# 3. Count Sort

## Overview:

Count Sort is a non-comparison-based sorting algorithm which is used on integers, performed iteratively and is not in-place [15]. What makes this such an interesting type of algorithm is that unlike comparison-based ones it does not rely on comparing one value against another to see if it is larger and then swapping, instead it like the name implies counts how many times each value occurs and uses that to sort. Conceptually it is relatively simple, it steps through the input array to be sorted, using a second array it keeps track of the number of occurrences of each value using its index and the values at each index. Once the entire array has been iterated over and been unpacked into the counting array the values get unpacked from the count array where they are greater than 0, so if the first value greater than 0 was at index 1 and it had a value of 5, then the sorted array would have 5 1's and so on through the rest of the count array until there is a sorted array.

There are however some limitations to the count sort algorithm and some assumptions it relies on to work. Regarding these limitations, it does not handle too large a range of values well and it can become a

burden on memory and time to run, generally counting sort should only be applied on situations where the range of values or number of keys will not be too large [16]. For assumptions counting sort expects non-negative numbers, and in a lot of cases expects to be given the maximum value of the array as an input, in the case of my implementation I have this as an optional parameter which can be determined at run-time however this does impact run-time.

## Diagram:

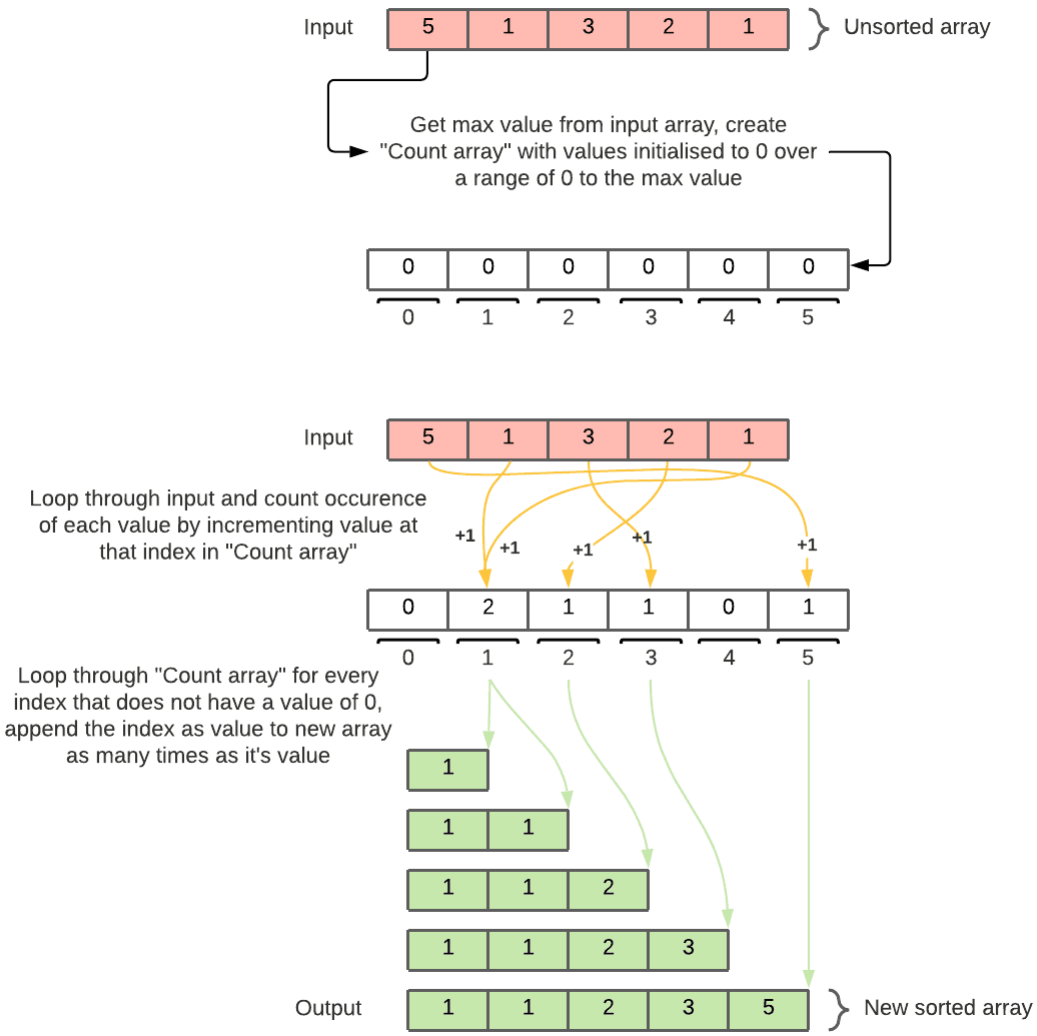


**Figure 3: Counting Sort bespoke diagram**

## Space and Time Complexity:

The Space Complexity for this algorithm is O(n+k) (k here represents the max key value), as both the size of the array to be sorted and the maximum value have an impact on the amount of memory needed [15].

Time Complexity in the worst case is O(n + k), similar to the space complexity as the size of both the array and the maximum value in the array increase so too will the run-time [3] [15].

## Explanation:

The implementation for count sort used was based off example found from a realpython article on sorting [12].

The count sort function implemented takes in two input arguments, the input array to be sorted and an optional argument for the max value in the input array, if no value supplied for the max value, then this is determined using python's max function.

A new list/array for counting the number of times each value occurs is initialised with all 0's with a range of 0 to the max value from the input array plus 1, as the new array has indices for each possible value of the input array the value at these indices will represent a count for how many times each value occurs.

A second array is initialised as an empty array, this array will ultimately be used to store the sorted list and will be what the function returns, as this is not an in-place sorting algorithm the original input array will remain the same.

Incrementing through each item in the input array it uses that item to add 1 to the index representing that item in the count array.

Finally, the code increments through the count list using enumerate (I did this to remove the need for a nested for loop and from testing this seems to have improved the run-time reasonably, I also experimented with using len(range(array) but saw no real difference in run-time over several tests, ultimately chose enumerate for readability), this allows the function to track the current index and the value at that index in two separate variables. As it loops through if the current item is 0 continue the loop as there is nothing there to sort, if the value is greater than 0 then to the sorted list it appends a list object enclosing index multiplied by the value for the current index, this is a useful python way for getting multiple of the same values into a list and is the same logic as how the count array was created earlier. Once this loop finishes the sorted input array has been stored in the sorted array and this is returned.

# 4. Insertion Sort

## Overview:

Insertion sort is a comparison-based sorting algorithm and performs the sort on the array in-place [17]. It works by stepping through the array and based on the current index compare it against the previous indexes. The value at the current index being sorted is compared against the previous indexes by looping backwards through the array from the current index, when it hits an index, whose value is greater than it, shift that value to the next index. Continue to loop backwards until it hits an index where either the value is less than it or have compared against every index before it and then insert the value at the last index it was less than. Then increment the index to be sorted and repeat this process comparing it against every index before it until you have reached the end of the array, at which point the array will be sorted in-place.

# Diagram:

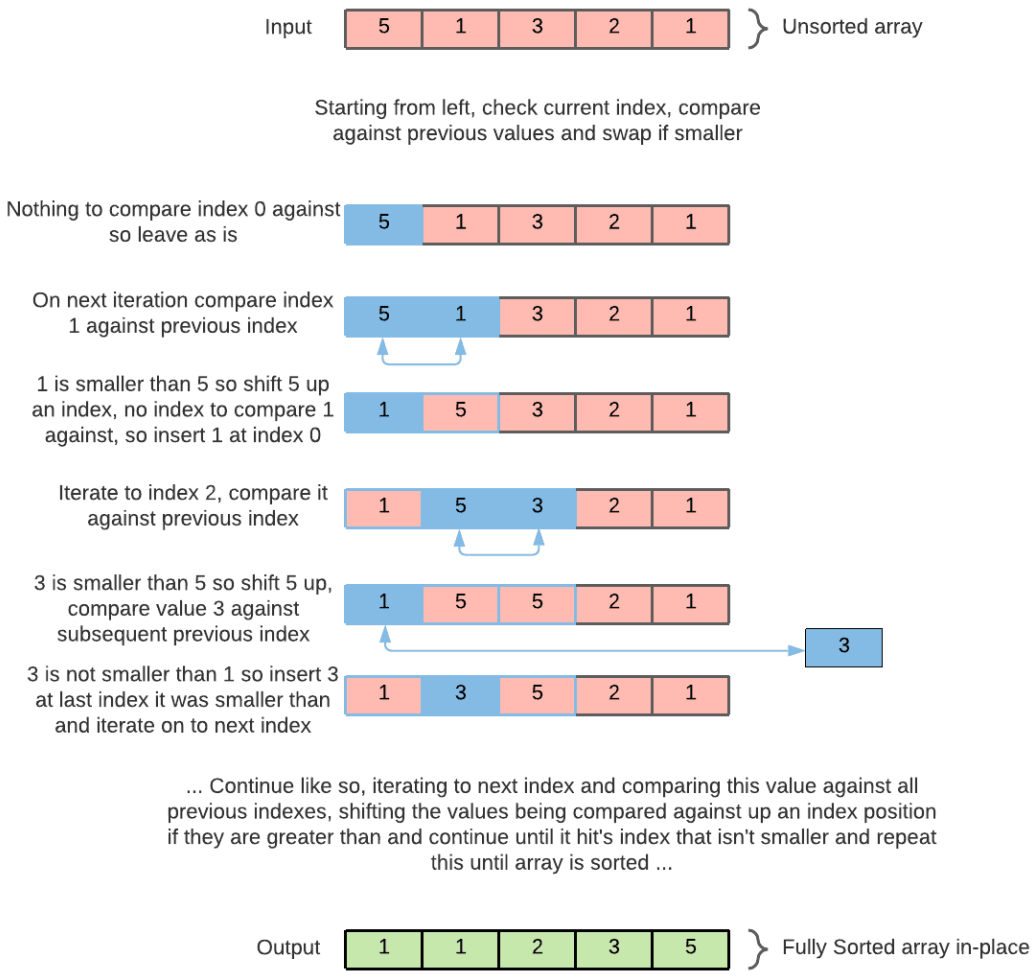


**Figure 4: Insertion Sort bespoke diagram**

# Space and Time Complexity:

The insertion sort function used in this project is performed iteratively and, It has a space-complexity of O(1) or 1 similar to bubble sort and it performs it's sort on the array in-place.

For time-complexity insertion sort is in best-case O(n) – when the input array is already sorted.

On average it has $O(n^2)$

It has time complexity in its worst case of $O(n^2)$ – when the input array is totally unsorted.

[3] [17]

# Explanation:

The implementation for Insertion sort used was based off example found from a realpython article on sorting [12].

The function for insertion sort takes in three arguments, the array to be sorted and two optional arguments for the left position and the right position, these are optional as when this function is being directly tested the default values for the optional arguments are what are used, however when used as part of the timsort algorithm different values will be specified for these arguments.

It first checks whether the right argument is “None”, if it is then set right as the index of the last element in the array, this is done as if no value is supplied for the right argument, then insertion sort will be performed over the entire length of the array.

After that initial check it uses a for loop to iterate over the range of the array to be sorted, as it iterates across this range the current indexes value (e.g. the index as determined by the iterator) is compared against every value before it, this is done as part of a while loop which continues until it encounters a value greater than the value at the current index. Within the while loop it shifts the value of those indexes who are greater than the current index value one position to the right and continues back through the array. A variable that keeps track of the index to compare the current index value against is decremented, and this process continues to until it either hits the start of the array or the value is greater than the current value. Once either of those conditions have been met the while loop exits and the value at the index we started this iteration of the for loop on, gets inserted back to the last position where it was greater than the value at that index. The for loop then continues to iterate starting this process again with the next index in the array and continues to do so for every index in the array.

# 5. Timsort

## Overview:

Timsort is a hybrid sorting algorithm that is comparison based and is not sorted in-place, hybrid sorting algorithms are ones that make use of other sorting algorithms to take advantage of their distinct benefits [18]. Timsort is a combination of both Insertion sort and Merge sort, it takes advantage of Insertion sort’s efficiency over small arrays [17] by splitting the input array into segments that are then merged, it does so by having a maximum size and if the array is smaller than this size timsort will instead simply use insertion sort. Merge sort is used to combine each of the chunks of the array that were insertion sorted, it is good at handling large arrays as it doesn’t need to go through the array several times [11]. The merge sort in timsort differs from the original in that it is not performed recursively and is not responsible for splitting up the array, instead it is used for merging two arrays together by looping through the indexes of the two arrays to be merged taking whatever is the smallest value from each array and appending them to a new array, continuing to do so until the new array is the length of both the arrays.

# Diagram:

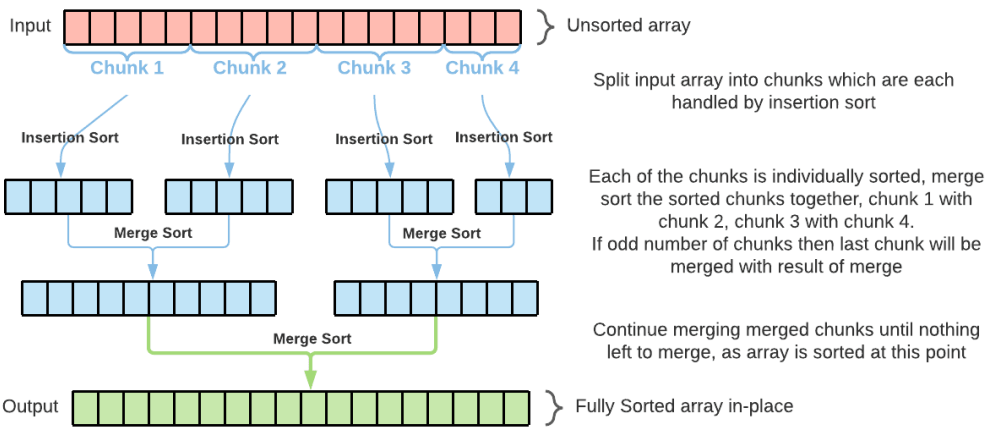


**Figure 5: Timsort bespoke diagram**

# Space and Time Complexity:

Timsort has a space-complexity of O(n), this implementation is not performed in-place however there are versions of the algorithm that can be performed in-place but this has a significant impact worsening the time-complexity.

For time-complexity Timsort in the best-case has O(n) and this is when the input is already sorted.

Time Complexity in the average-case is O(n log n).

Time Complexity in the worst case is O(n), this is the case because there are no nested loops.

[5] [19]

# Explanation:

The implementation for Timsort used was based off example found from a realpython article on sorting [12].

This implementation for Timsort starts by first defining a minimum run, this will be the size of each section of the input array as it is sliced up. Using a for loop the array is iterated over in steps based on the minimum run which in this implementation is 32 and at each iteration of the for loop a slice or chunk of the array is sent to the previously defined insertion sort function and this continues until the end of the array is reached. At this point the array will be sections of itself that are sorted, but it is not sorted as a whole.

Next it takes each of the sections that have been insertion sorted and it merge sorts these together, first it sorts them in the same steps of 32 (the minimum run) as defined at the start of the function and this is done as part of a for loop where again it iterates through the array in steps of 32 taking the first slice and merging with the second, the third with the fourth and so on. Once this is complete the size of the slice to merge over doubles as we now have sections of the original array in slices of 64 sorted, this process repeats until the array is fully sorted. This is all controlled within a while loop which exits when the size of slices exceeds the length of the array.

I have not fully implemented merge sort, instead I have just implemented the merge functionality which I will describe here as it is used by timsort yet not described elsewhere in this report. The merge function takes in a left and right slice of the array from timsort function and it assumes that these input arrays are already sorted, if either the left or the right is empty then it simply returns the opposite as there is nothing to merge. After this check a new array/list called result is initialised, this is what will be returned as we are not sorting in place as part of merge sort. The index to start at for both the left and right arrays are both set to 0 and this will be the start index for both as they are merged together. The merge itself is done within a while block which continues until the new result array size is equal to the combined size of the left and right arrays. The merge is carried out by simply comparing the value at the current position in the left array with the value at the current position in the right array, whichever has the smaller value gets appended to the results array. After the value from one of the arrays has been appended the current position for that array is incremented so that on the next iteration it is comparing that index against the other arrays current position, again depending on which value from which array is smaller that gets appended to the result array and the position is incremented. This continues until the last value from one of the arrays is appended to the result array, at which point the rest of the other array is appended to the result array and the result array gets returned.

# Implementation and Benchmarking

## Implementation:

A single file “main.py” contains all the code for the various sorting algorithm functions, the code to run these functions and any ancillary functions like the timing and random number generator functions.

For testing the sorting algorithm functions two functions have been defined, the first which returns an array of random numbers called “random_array_generator()” which utilizes the numpy package and defines a random number generator in order to return an integer array of specified size, it takes in three arguments, the size of the array to generate and return, the minimum value in the range for the array and the maximum value for the range. The random array generator function is in turn used within the second function for testing, “timer_function()”, which is for returning the elapsed time to run a sorting algorithm in milliseconds. This function takes two arguments, the sorting algorithm function to be tested and the size of the array to test over, within this function it runs the algorithm with different random arrays ten times and collects the elapsed time (in milliseconds) for each run into a list and returns the average of those results to give the average elapsed time for the sorting algorithm.

These testing functions are all called within a default main function where each algorithm is tested over different array sizes and the average elapsed time is captured and printed to the command line as a formatted table showing the elapsed time of each algorithm at differing array sizes.

A commented-out section is also present here which imports some additional packages (pandas, matplotlib, seaborn) and has code for storing the benchmarking results and trends these producing the charts found below. Two charts are generated, one which simply trends the results using a linear y-axis, however this obscures the better performing algorithms as both Bubble Sort and Insertion Sort take exponentially longer to run for the larger array sizes. To address that problem a second chart is generated

which has a logarithmic y-axis, that more clearly distinguishes between each of the sorting algorithm run-times. I have left in this section to firstly demonstrate further how the benchmarking results are handled and secondly so that whoever is looking through the code also has the option to have a new set of charts generated to easily visualise the benchmark results on their own computer.

This code can simply be ran from the command line using “python main.py” (assuming you are in the same directory). The only required packages are numpy and time, the commented-out section at the end for placing the results in a dataframe and visualising them are done requires the packages matplotlib, seaborn and pandas.

## Benchmarking Results:

| Size | 100 | 250 | 500 | 750 | 1000 | 2000 | 4000 | 6000 | 8000 | 10000 |
|---|---|---|---|---|---|---|---|---|---|---|
| Bubble Sort | 1.696 | 9.905 | 41.076 | 95.808 | 163.346 | 694.972 | 3351.406 | 6867.645 | 12660.635 | 20163.813 |
| Quick Sort | 0.202 | 0.595 | 1.506 | 2.29 | 3.9 | 12.501 | 31.755 | 39.604 | 64.696 | 103.324 |
| Count Sort | 0.094 | 0.104 | 0.196 | 0.2 | 0.2 | 0.3 | 0.6 | 1.0 | 1.1 | 1.5 |
| Insertion Sort | 0.905 | 6.095 | 28.099 | 48.806 | 80.231 | 316.354 | 1403.562 | 3027.261 | 5348.717 | 8450.252 |
| Timsort | 0.4 | 1.3 | 3.0 | 5.4 | 7.2 | 15.9 | 29.307 | 32.693 | 44.404 | 58.3 |

**Figure 6: Benchmarking tabulated results, average elapsed time of algorithms at different array sizes**

Across all the benchmarking tests count sort performed the best by a significant margin, especially as array size increased. Compared against all the other algorithms there was no array size tested where it did not perform clearly better, sorting an array of size 100 on average it was more than twice as fast as the next best; quicksort. At sorting an array of size 10000 on average it was nearly 39 times faster against the next best; timsort.

Of the comparison-based algorithms timsort performed the best at higher array sizes, taking 58.3 milliseconds on average for an array size of 10000. Presumably timsort performs so well due to it being a hybrid algorithm, taking the best characteristics of merge and insertion sort.

Quicksort performed the best at the lower range for array sizes outperforming timsort up until an array size of 4000. It performed the best at array size of 2000 at 12.501 milliseconds, beating out timsort by more than 3 milliseconds, making it a fifth faster on average at that size.

Insertion sort did second worst performing particularly worse on average at higher array sizes, taking approximately 82 times the time to run as quick sort and 145 times the length to run as timsort.

Bubble sort performed the worst by a significant margin and took consistently greater than double the time to run as insertion sort.

Both bubble sort and insertion sort were so much worse than the other algorithms at high array sizes that in a linechart with linear scale for run time it completely obscured the other results (see figure 7).To distinguish the results more clearly from one another I used a logarithmic scale for the run time which allows the different algorithms results to be distinguished (see figure 8).

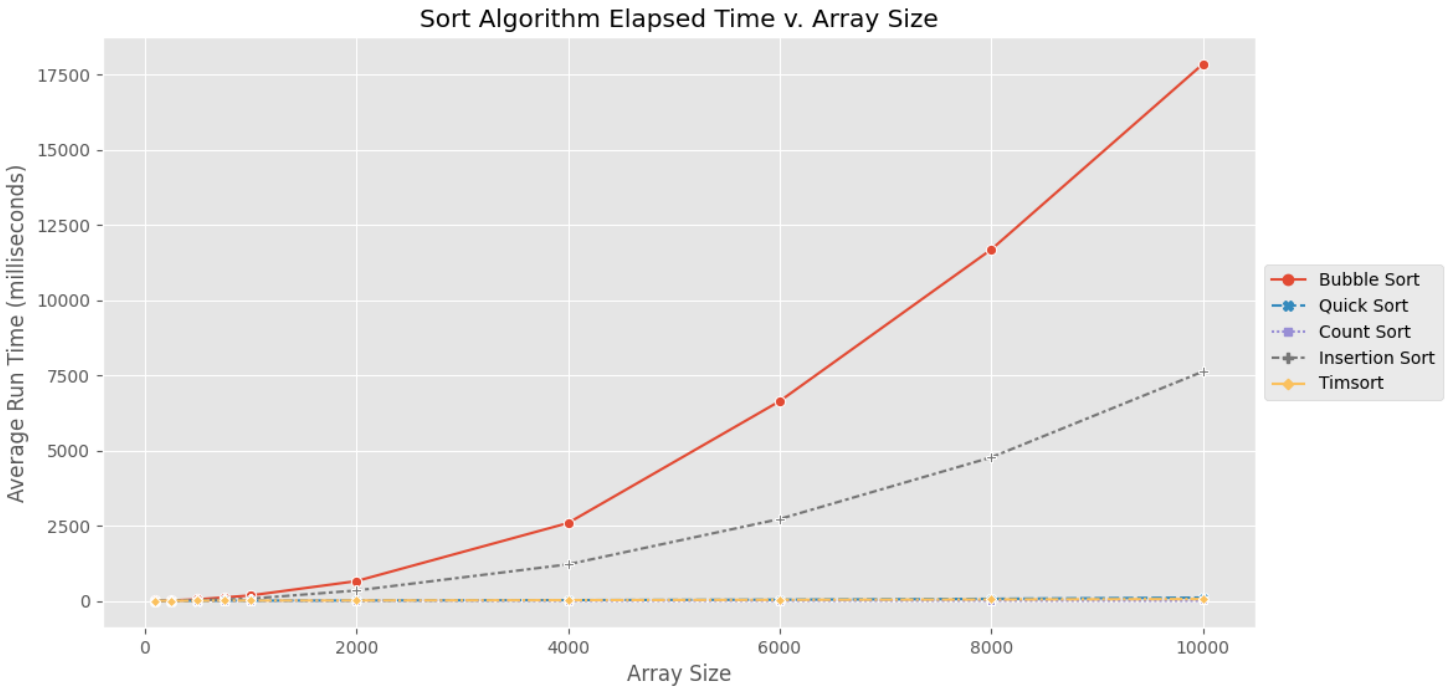


**Figure 7: Average elapsed time trended against array size w/ linear scale for elapsed time**

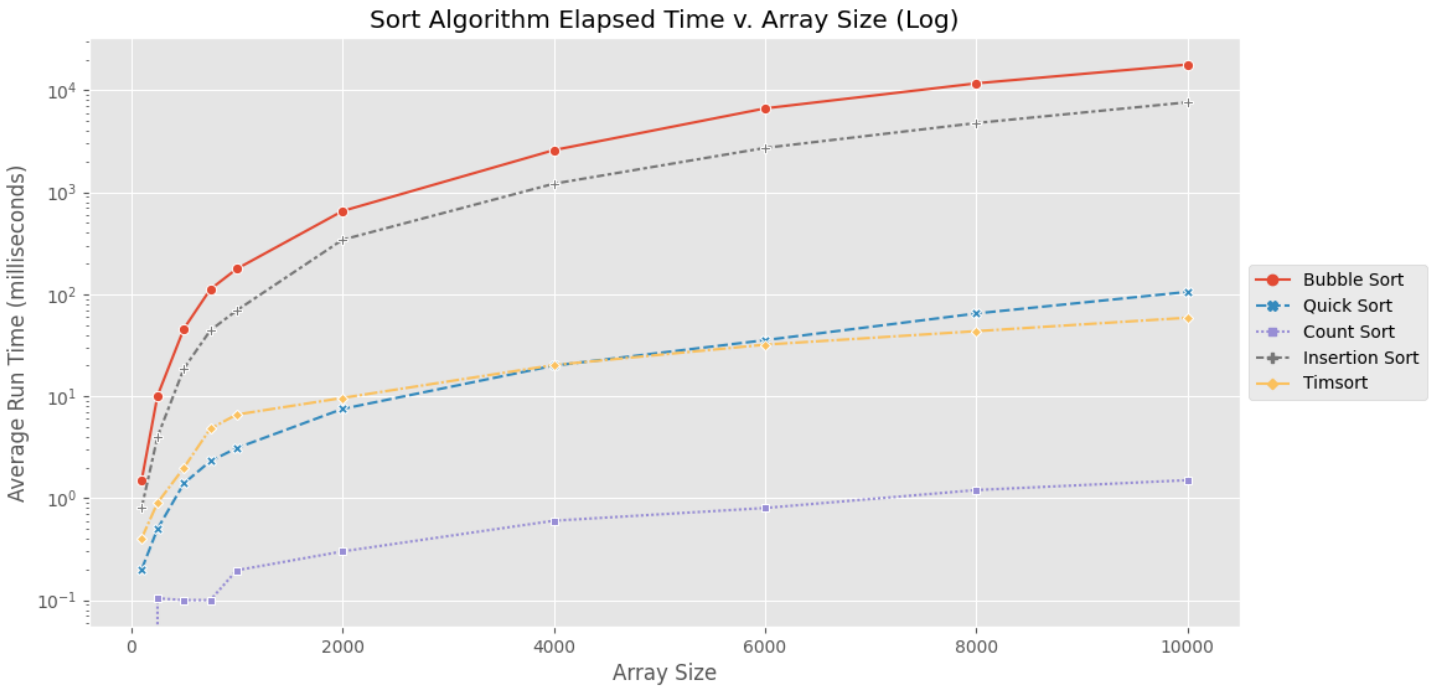


**Figure 8: Average elapsed time trended against array size w/ logarithmic scale for elapsed time**

# Benchmarking Conclusion:

The elapsed times across the different sorting algorithms in general matched my expectations. Predictably Bubble Sort and Insertion Sort performed exponentially worse as the size of the array increased, in keeping with their express limitations as defined by their worst case time-complexity of $O(n^2)$ [3]. Both bubble sort and insertion sort both serve as being introductions to sorting algorithm and so are relatively simple in their implementation, based on this it was not surprising that these were the worst algorithms in general. However, I would have expected Insertion sort to have performed better at smaller sizes than it did and interestingly Timsort performed significantly faster even with an array size of 100, which demonstrates quite clearly that a combination of several small insertion sorts (in the code implemented for timsort the insertion sorts were on arrays of size 32 and less) merged together with a basic merge sort perform faster than a single insertion sort.

Quick Sort behaved very well at small array sizes, getting worse at higher sizes and when compared against Timsort shows that it performs better up till an input array size of 4000 at which point the advantage of the hybrid sorting algorithm become clear, this is quite interesting to me as it clearly highlights cases where you may pick quick sort e.g., when generally going to be sorting arrays of size 4000 and less.

Count Sort was the most surprising performance-wise with the arrays, which in hindsight does make sense as it has a very linear run time [16]. For count sort I had to increase the range of values (from 0-100 to 0-1000) for the input arrays to increase the run time so as to get more useful benchmarking results, for lower array sizes it tended to result in an elapsed time of 0 milliseconds being returned across multiple instances and so to avoid this I increased the range. Another justification for increasing the range is that prior to that the tests were more advantageous to a non-comparison-based algorithm like count sort as while the range of values increase so too will the time-complexity [3], which isn't the case with the other comparison-based algorithms. As such count sort is the only algorithm that is affected by the change in range.

From the trends we can further see the differences in time complexities for the different sorting algorithms, with bubble sort and insertion sort both growing noticeably exponentially especially from figure 7. With figure 8 we can see that count sort run time appears to grow nearly linearly which it should have. Quick sort and timsort it is harder to see their growth as it is obscured in figure 7 thanks to bubble and insertion sort taking exponentially longer but from figure 8 we can see that they are taking an order of magnitude longer on average than count sort and this is likely to become more pronounced at large array sizes. However over larger ranges of numbers in the input arrays it is likely that count sort would itself start to perform much worse.

# References


[1] “Sorting,” [Online]. Available: https://en.wikipedia.org/wiki/Sorting.

[2] “Sorting Algorithm,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Sorting_algorithm.

[3] D. M. Aditya and G. Deepak, “SELECTION OF BEST SORTING ALGORITHM,” *International Journal of Intelligent Information Processing,* pp. 363-368, 2008.

[4] “Big O Notation,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Big_O_notation.

[5] “Time complexity,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Time_complexity.

[6] “Space complexity,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Space_complexity.

[7] “Big O Notation Explained With Examples,” FreeCodeCamp, [Online]. Available: https://www.freecodecamp.org/news/big-o-notation-explained-with-examples/.

[8] “Bubble sort,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Bubble_sort.

[9] “Average-case complexity,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Average-case_complexity.

[10] “In-place algorithm,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/In-place_algorithm.

[11] “Merge sort,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Merge_sort.

[12] “Sorting Algorithms in Python,” RealPython, [Online]. Available: https://realpython.com/sorting-algorithms-python/.

[13] “Quicksort,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Quicksort.

[14] “Quicksort Algorithm,” AskPython, [Online]. Available: https://www.askpython.com/python/examples/quicksort-algorithm.

[15] “Counting Sort,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Counting_sort.

[16] J. Su, “Sorting in Linear Time,” [Online]. Available: https://web.stanford.edu/class/archive/cs/cs161/cs161.1168/lecture7.pdf.

[17] “Insertion sort,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Insertion_sort.

[18] “Timsort,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Timsort.

[19] T. Peters, “[Python-Dev] Sorting,” 20 July 2002. [Online]. Available: https://mail.python.org/pipermail/python-dev/2002-July/026837.html.