BENCHMARKING SORTING ALGORITHMS

Computational Thinking with Algorithms

JENNIFER RYAN
G00376325

3rd May 2020

Table of Contents

1.0 Introduction

Data sorting is an important step in the efficient implementation of many tasks, such as searching. If data is in some discernible order, it is easier to find certain elements and one can avoid a time-consuming sequential search, where each element needs to be examined to find the target value. For example, the tables below contain the same set of numbers but one is in random order and the other is sorted. Not only is it possible for a person to quickly locate specific elements in the sorted table, it is also easier to check for duplicates and frequencies of elements as well as statistical information like minimum, maximum and median values.

10	3	4	37	50	41	27	12	20	2
31	14	25	7	39	23	18	34	6	32
28	5	34	19	8	49	29	50	27	16
18	30	4	35	37	1	47	33	44	48
36	40	27	10	43	22	27	17	5	1
10	46	12	11	38	33	49	13	38	29
33	24	1	46	9	29	45	4	4	35
27	16	36	18	15	42	26	10	21	17

1	1	1	2	3	4	4	4	4	5
5	6	7	8	9	10	10	10	10	11
12	12	13	14	15	16	16	17	17	18
18	18	19	20	21	22	23	24	25	26
27	27	27	27	28	29	29	29	30	30
31	32	33	33	33	34	34	35	35	36
36	37	37	38	38	39	40	41	42	43
44	45	46	46	47	48	49	49	50	50

Similarly, sorted data allows computer programs to work more efficiently, whether it's for organising data in the hard drive of a machine or as part of more complex algorithms and procedures. There are many sorting algorithms that can be used to sort data and it is important to choose one based, not only on the particular problem, but also the resources or number of operations required to run it. There are several factors that should be considered when assessing sorting algorithms:

- Time Complexity
    - How long it takes an algorithm to run.
- Space Complexity
    - The amount of memory required to run an algorithm.
- Stability
    - Whether an algorithm preserves the relative order of equal elements, if it is important to retain this order in the final sorted output.

- In-Place Sorting
    - When an algorithm does not need to use more memory to process the input data, if memory is a concern.
- Suitability
    - An algorithm is well-suited to the particular problem set, in terms of the size and type of input.

For the purposes of this project, the focus will be on time complexity in terms of the runtime of five specific sorting algorithms:

- Two simple comparison-based sorting algorithms:
    - Insertion Sort
    - Selection Sort
- Two efficient comparison-based sorting algorithms:
    - Merge Sort
    - Quicksort
- A non-comparison sorting algorithm:
    - Counting Sort

These algorithms have been coded and implemented with Python in the accompanying Jupyter Notebook (also available here) and their runtimes are benchmarked with different random integer input sizes. Due to the inefficiency of some of the simpler algorithms with large input sizes, it will likely take several minutes for the code to finish. Note that the specific data figures that are referenced in the discussion section will likely be different each time the notebook is run as the randomised input will change, but the general trends should remain stable.

2.0 Efficiency

Algorithmic efficiency can be measured in terms of space complexity and time complexity and is based on the size of the input that is to be processed by an algorithm, denoted as *n*. It is often assessed using asymptotic notation which measures the rate of growth of either space needed or time elapsed when an algorithm runs, relative to the size of the data passed into the algorithm. There are several forms of asymptotic notation but the main ones are as follows:[1]

- Big-O notation measures the efficiency of an algorithm in the worst case.
- Theta (Θ) notation measures the efficiency of an algorithm in the average case.
- Omega (Ω) notation measures the efficiency of an algorithm in the best case.

These best, worst and average cases are usually all communicated with 'O' notation,[2] e.g. *O(n)* describes linear growth in that the time/space needs of an algorithm will increase in a linear relationship with the input size. Generally, the worst and average cases are considered more than the best case when evaluating an algorithm as such scenarios are more likely to occur in reality.

2.1 Space Complexity

Space Complexity refers to the amount of working memory required to run an algorithm in the worst case. The memory assigned to the operation of an algorithm can be implicit, in the case of recursive calls to a function, or explicit, such as storing input elements temporarily outside of the original input array.[3] Simple sorting algorithms can achieve constant *O(1)* space complexity, meaning that their memory needs do not grow as the input size grows. The more efficient sorting algorithms often utilise recursion, which uses more memory. In such cases, a decision needs to be made about balancing good runtime against the memory cost. Below are the worst case space complexity values for each of the sorting algorithms; these will be addressed individually in the following sections.

Algorithm	**Space Complexity**
Selection Sort	*1*
Insertion Sort	*1*
Merge Sort	*n*
Quicksort	*log(n)*
Counting Sort	*n + k*

2.2 Time Complexity

The time complexity of a sorting algorithm describes the time it takes for data to be sorted based on the input size *n*. It does not refer to the actual measured time an algorithm takes to run as that depends on a lot of other factors such as the particular programming language, implementation, and the machine it's run on. Rather, time complexity measures the number of times each statement in an algorithm is executed.[4] Thus, the fewer executed statements, the more efficient the algorithm in terms of time complexity. For example, an algorithm with $O(n^2)$ performance runs in quadratic time, which usually involves the use of many nested iterations over the input data. The best, worst and average runtime efficiencies of the sorting algorithms discussed herein are summarised below and each will be addressed in the sections below.

Algorithm	**Best Case**	**Worst Case**	**Average Case**
Selection Sort	n^2	n^2	n^2
Insertion Sort	n	n^2	n^2
Merge Sort	$n \log n$	$n \log n$	$n \log n$
Quicksort	$n \log n$	n^2	$n \log n$
Counting Sort	$n + k$	$n + k$	$n + k$

3.0 Sorting Algorithms

The majority of the sorting algorithms that will be examined are comparison-based, meaning that they use comparison operators to determine the order of elements. These comparison operators assess whether elements are less than, greater than or equal to one another, which helps to place them in correct order. However, not all comparison-based sorting algorithms are created equally. The two simple algorithms, Selection and Insertion Sorts, are not very fast or efficient, particularly when it comes to large input sizes of random data, as they function by performing several iterations through the input in order to sort it. The more efficient algorithms, Merge Sort and Quicksort, work by breaking the input data down into smaller parts to avoid excessive iterations. The final sorting algorithm that will be examined, Counting Sort, is non-comparison-based and highly efficient but, unlike the other algorithms, will only function well in particular cases and needs to make certain assumptions about the input data. There is no single sorting algorithm that is suitable in every instance. Even the most simple sorting algorithms are valuable in specific cases and the most powerful ones have limitations.

3.1 Selection Sort

Selection sort is a simple comparison-based algorithm that works by selecting each element of the unsorted input in turn and comparing it to every other element using a comparison operator in order to find the next minimum value and rebuild the data in sorted order.

3.1.1 Procedure

Input is split into a sorted side and an unsorted side. In the beginning, the sorted side is empty and the unsorted side contains the entire array.[5]

8	1	9	4	5

Unsorted array

The first element of the array is then compared to every other element and the one with the minimum value is placed into the sorted side of the array. To achieve this, the minimum valued element switches place with the first element.

8	1	9	4	5

First element, 8, is compared to every other element

8	1	9	4	5

The second element, 1, is less than 8 and there are no other elements less than 1

1	8	9	4	5

1 and 8 switch positions and the sorted side begins

This process is repeated on the remaining unsorted elements and the minimum value found in each iteration is placed into the next slot in the sorted side until all items have been sorted.

1	4	9	8	5

4 is identified as the next minimum number and is switched with 8

1	4	5	8	9

5 is identified as the next minimum number and switched with 9

1	4	5	8	9

8 is less than to 9 and so is in the correct position and the array is sorted

The algorithm can also be implemented by searching for the maximum value in each iteration or building the sorted array from the opposite side.[6]

3.1.2 Performance

Time Complexity

Selection Sort runs in $O(n^2)$ time in the best worst and average cases. Regardless of the input size or how sorted the data is already, it will always compare each element with every other through nested iterations. Algorithms with quadratic time do not scale well and should generally only be used for smaller input sizes.

Space Complexity

The algorithm has constant *O(1)* space complexity, meaning that it does not require more memory to keep up with a growing input size.[7] The memory needs of the algorithm are the same whether the input has 10 or 10,000 elements. It does not require any recursive calls or temporary lists; all iterations and swaps occur within the bounds of the input data.

Stability

Selection Sort does not operate in a stable manner and the relative order of elements cannot be guaranteed. Take the following illustrative example:

3	5	3	1	4	*In this unsorted list the green 3 comes before the orange 3*
1	5	3	3	4	*When the algorithm runs the green 3 will be swapped with 1*
1	3	5	3	4	*Next 5 will be swapped with the orange 3*
1	3	3	5	4	*5 is swapped with the green 3*
1	3	3	4	5	*Finally, 5 is swapped with 4. The list is sorted but now orange 3 comes before green 3*

In-Place Sorting

Selection Sort performs sorts in-place by swapping elements within the original list and thus does not require any additional memory to store elements temporarily. This makes it a useful option if space is a concern as the memory required is constant.

Suitability

Even though Selection Sort is one of the slowest sorting algorithms, it is useful in certain cases. The algorithm works well with small input lists and, because it is simple and sorts in-place, it does not require a lot of memory to operate. It generally outperforms the similar Insertion Sort on larger, random inputs

but does less well with smaller, more sorted data.[8] Selection Sort can also be useful to check if a list is already sorted as no swaps would need to be performed in such a case.[9]

3.1.3 Implementation

The code for Selection Sort has been implemented with Python as follows [10] (see Jupyter notebook for commented code):

```
def selection_sort(alist):

    for i in range(len(alist)):
        min_index = i

        for j in range(i+1, len(alist)):
            if alist[min_index] > alist[j]:
                min_index = j

        alist[i], alist[min_index] = alist[min_index], alist[i]
```

3.2 Insertion Sort

Insertion sort works by comparing unsorted elements to already sorted elements in order to insert them into the correct position among the sorted elements. It is similar to Selection Sort except that it iterates over the sorted list to insert an element into the correct position instead of only iterating over the unsorted list.[11]

3.2.1 Procedure

Like Selection Sort, it starts by splitting the input into a sorted and unsorted side. In the beginning, the sorted side contains the first element of the input array as a single item is sorted by default.

8	1	9	4	5

Sorted side contains the number 8

The first element of the unsorted side is selected, compared to the element in the sorted side and placed in the correct position, based on whether it is less than, greater than or equal to it. Sorted elements are

shifted one step to the right until the correct position for the unsorted element is found, whereupon it is inserted into the empty space.

The first element of the unsorted array, 1, is selected and compared to 8

1 is less than 8 so 8 shifts to the spot previously occupied by 1

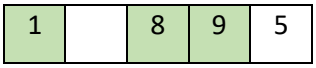
1 is placed in the empty position before 8

This procedure is repeated with each element in the unsorted side being compared to the sorted side and the element being inserted into the correct position until there are no items remaining in the unsorted side.

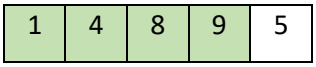
9 is already in the correct position as it is greater than 8

4 is the next element in the unsorted side

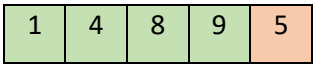
4 is less than 9 so 9 shifts to the right



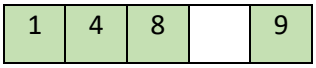
4 is less than 8 so 8 shifts to the right



4 is not less than 1 so is inserted into the empty slot



5 is the final element to be sorted



5 is less than 9 so 9 shifts to the right

5 is less than 8 so 8 shifts to the right

5 is not less than 4 so is inserted into the empty slot and the list is sorted

3.2.2 Performance

Time Complexity

Even though they share worst and average case $O(n^2)$ time, Insertion Sort and Selection Sort perform differently with certain input instances. While Selection Sort is faster with randomised input, Insertion Sort usually performs better with small arrays that are largely already sorted. This is because it only needs to iterate through the sorted side until it finds the correct position to insert an element, whereas Selection Sort will always iterate through the entire input data to find the next value to add to the sorted side.

Unlike Selection Sort, Insertion Sort it is capable of running in $O(n)$ linear time in the best case. Linear time is possible when the input is already completely sorted and so the algorithm does not need to iterate through the sorted side as all elements are already in their sorted position.

Space Complexity

Another similarity between Insertion and Selection Sorts is that they both have constant $O(1)$ space complexity. Again, as sorting takes place inside the input, Insertion Sort does not require any extra memory to sort input of larger sizes.

Stability

Insertion Sort is differentiated from Selection Sort in that it is a stable algorithm. Because it chooses each element in order from the unsorted side and shifts rather than swaps elements, it is able to maintain the relative relationships between elements of the same value when placing them in the sorted side.

In-Place Sorting

Sorting occurs in-place as the sorted and unsorted sides only occupy space within the original input array and are not stored in another location. Elements are shifted and inserted only within the confines of the array.

Suitability

Insertion Sort is very efficient when used for smaller, near sorted lists. In fact, it is so efficient at this that it is often used as part of hybrid sorting algorithms, such as Timsort, to deal with smaller sub-lists. Even some implementations of Quicksort utilise Insertion Sort once the partitions get below a certain size to avoid the need for further recursions.[12] Insertion Sort is, however, highly inefficient for larger and/or completely random input cases as the iterations through both unsorted and sorted sides, while also shifting elements, can be very time-consuming.

3.2.3 Implementation

The code for Insertion Sort has been implemented with Python as follows[13] [14] (see Jupyter notebook for commented code):

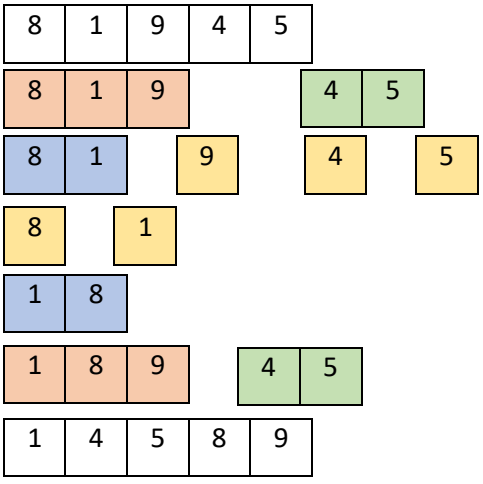
```
def insertion_sort(alist):

    for i in range(1, len(alist)):
        current_value = alist[i]

        while i > 0 and alist[i-1] > current_value:
            alist[i] = alist[i-1]
            i -= 1

        alist[i] = current_value
```

3.3 Merge Sort

Merge Sort is the first of the efficient comparison-based algorithms addressed here. It is known as a "Divide and Conquer" algorithm that recursively breaks the input problem down into smaller versions of itself until it reaches a base case that can be quickly sorted. The sorted data is then merged back together in the correct order.

3.3.1 Procedure

The input data is split at the midpoint, or as close to the midpoint as possible. The sub-arrays are then recursively split in the same manner until there is only one element in each. The recursion ends at this stage as the base case has been reached and only single-element sorted arrays remain. The algorithm now works back up through the callers, merging the smaller lists into ordered larger lists.



Unsorted array

First split occurs close to the midpoint

Second split creates 3 single element sub-arrays

Final split achieves base case – no array with more than one item

Merging begins with the last split list arranged in sorted order

Next split lists are merged in sorted order

Final split lists merged and the input has now been sorted

3.3.2 Performance

Time Complexity

The best, worst and average cases of Merge Sort are all *O(n log n)*, which makes it a good choice if predictability is an important factor as the algorithm should deliver a similar performance in all cases. The recursive calls to split the input account for some of this time, but the majority of the algorithm's runtime is involved with merging the smaller sorted lists back together to create the sorted output. It should be noted as well that, even if a sub-array is already sorted, it will still be split and merged like any other. This makes Merge Sort unsuitable for near sorted arrays as it will not work any faster than it would for a random array.

Space Complexity

Merge Sort has *O(n)* space complexity meaning that the memory needs of the algorithm grow in parallel with the input size. This is because the input is split into sub-arrays that are stored outside of the original input before being merged. Thus, it can be quite an expensive algorithm in terms of memory usage for very high input sizes.

Stability

Merge Sort offers stable sorting as, because the lists are so small when the merging step begins, elements can maintain their relative positioning.

In-Place Sorting

Sorting does not occur in-place with Merge Sort. Elements are split into sub-arrays and merged together in order outside of the original input array.

Suitability

Merge sort is a very efficient comparison-based sorting algorithm for larger randomised input arrays. However, it does not perform as efficiently for nearly-sorted or smaller sized data[15] as it still has to recursively split and merge the input whether it is sorted or not. It usually performs slower than the similar Quicksort but is generally the favoured algorithm for sorting data with slow access times or linked lists because it more evenly splits data and does less work in the partitioning step.[16] [17]

3.3.3 Implementation

The code for Merge Sort has been implemented with Python as follows[18] [19] (see Jupyter notebook for commented code):

```
def merge_sort(alist):

    if len(alist) > 1:
        midpoint = len(alist) // 2

        left_list = alist[:midpoint]
        right_list = alist[midpoint:]

        merge_sort(left_list)
        merge_sort(right_list)

        l = 0
        r = 0
        a = 0

        while l < len(left_list) and r < len(right_list):

            if left_list[l] <= right_list[r]:
                alist[a] = left_list[l]
                l += 1
            else:
                alist[a] = right_list[r]
                r += 1
            a += 1

        while l < len(left_list):
            alist[a] = left_list[l]
            l += 1
            a += 1

        while r < len(right_list):
            alist[a] = right_list[r]
            r += 1
            a += 1
```

3.4 Quicksort

Like Merge Sort, Quicksort is a “Divide and Conquer” algorithm that works recursively to break the problem down into smaller versions of itself for more efficient sorting. Instead of always splitting data at the midpoint as in Merge Sort, Quicksort uses a pivot to partition the input data. This pivot can be any element in the input array and successful implementation of the algorithm is based on choosing a good pivot. It may be beneficial to choose random or median elements as the pivot if you are unsure of the input data as it could help to avoid unbalanced partitions.[20]

3.4.1 Procedure

The first step is to choose a pivot around which the partition is formed. This can be any element in the input array but generally is chosen by:

- Always picking the first element
- Always picking the last element
- Picking the median/middle element
- Picking a random element

In this example, the first element is chosen to act as pivot.

7	9	1	8	5	4

First element of the array is selected as pivot

The rest of the elements will be compared to this and those that are less than its value will be placed on its left while those with a greater value will be placed on its right. The pivot element is now in its final position as every other element is either greater than, less than or equal to it. The algorithm now recursively partitions the sub-arrays on the first side of the pivot and then the other side until all elements are sorted.

1	5	4	7	9	8	*Elements organised around the pivot, 7, which is in its final position*
1	5	4	7	9	8	*The first element of the left sub-array is selected as pivot*
1	5	4	7	9	8	*The elements are already on the correct side of the pivot, which is in its final position*
1	5	4	7	9	8	*Recursively move on to next sub-array on same side and choose first element as pivot*
1	4	5	7	9	8	*Sort element around pivot. Single elements are already sorted. Left partition is sorted.*
1	4	5	7	9	8	*Move to right partition and select first element*
1	4	5	7	8	9	*Switch elements and array is now sorted*

3.4.2 Performance

Time Complexity

Quicksort is one of the most efficient sorting algorithms, and generally considered to be the fastest across diverse input types.[21] Even though it shares the same time complexity as Merge Sort in best and average cases, *O(n log n)*, it often has faster actual runtimes with some inputs, such as arrays, because its inner loops can be more efficiently implemented and it has low memory requirements.[22]

The efficiency of Quicksort largely depends on the choice of pivot, with the worst case, $O(n^2)$, occurring if the pivot is poorly chosen, particularly when the input is almost entirely sorted. For instance, if the input is a sorted list and the pivot chosen is always the first element, when the data is partitioned, one of the resulting sub-lists will always be disproportionately larger than the other and Quicksort will take a long time to finish. The efficient partitioning of data is vital to efficient sorting with Quicksort.[23]

Space Complexity

The space complexity of Quicksort depends largely on its implementation with versions that do not sort in-place having *O(n)* complexity in the worst case, just like Merge Sort. Most implementations, however, generally work in-place and so have a worst case of *O(log n)*. This accounts for the recursive elements of the algorithm when it runs. Quicksort does not require the extra memory that Merge Sort needs to store the sub-arrays.

Stability

Unlike Merge Sort, Quicksort is not stable. The partitioning of sub-arrays around a pivot can potentially alter the relative ordering of unique elements.

In-Place Sorting

Unlike Merge Sort, Quicksort usually performs in-place sorting as it does not require the use of temporary lists to store input elements. Data is recursively partitioned and then sorted within the input space.

Suitability

Quicksort is often regarded as one of the best sorting algorithms. Like Merge Sort, it is very efficient for large amounts of random data but less so for nearly-sorted input. However, it still generally outperforms this and other comparison-based algorithms in most input cases as it has very little memory use and excellent runtimes.

3.4.3 Implementation

For efficiency, it is important that Quicksort is implemented in-place with a good pivot choice. With that in mind, Quicksort was implemented as follows[24] (see Jupyter notebook for commented code):

```
def quick_sort(alist):

    def quick_helper(items, low, high):

        if low < high:
            split_index = partition(items, low, high)

            quick_helper(items, low, split_index)
            quick_helper(items, split_index + 1, high)

    quick_helper(alist, 0, len(alist) - 1)

def partition(alist, low, high):

    pivot = alist[(low + high) // 2]

    l = low - 1
    r = high + 1

    while True:

        l += 1
        while alist[l] < pivot:
            l += 1

        r -= 1
        while alist[r] > pivot:
            r -= 1

        if l >= r:
            return r

        alist[l], alist[r] = alist[r], alist[l]
```

3.5 Counting Sort

Counting Sort is fundamentally different from the other algorithms described above. While they work by comparing input elements against one another, Counting Sort is a non-comparison-based algorithm that relies on a set of assumptions about the input data. This makes it much more efficient but also more limited in that it only works with integers and needs to know something about their values in order to operate.

3.5.1 Procedure

Counting Sort needs to know the maximum key value in the input.

5	2	3	4	**5**	3	4	4

Maximum key value in the array is 5

The algorithm then creates a count list whose indexes represent the integer values of the input array. It is this operation that makes Counting Sort an integer-only sorting algorithm.[25] Each box below represents an integer between the values 0-5. The frequency of each number is recorded in this list:

- There are no instances of numbers 0 or 1
- There is one instance of number 2
- There are three instances of the number 3
- There are two instances of the number 4
- There are two instances of the number 5

0^{0}	0^{1}	1^{2}	3^{3}	2^{4}	2^{5}

Counting the instances of each unique value matched to the index

The algorithm then cycles through this counting list, rebuilding the input data in sorted order.

2

0 and 1 do not exist so one 2 is the first element added to the sorted list

2	3	3	3

Next three 3s are added

2	3	3	3	4	4	5	5

Finally, two 4s and two 5s are added to complete the sorted list

3.5.2 Performance

Time Complexity

The most powerful comparison-based algorithms can only ever achieve *O(n log n)* runtime because they must cycle through the data to compare elements.[26] However, because Counting Sort does not need to make comparisons, its time complexity is based on the input size (*n*) and maximum input value (*k*) added

together, making it *O(n + k)*. It iterates through the input list to populate the count list and sorted list (*n*) and also through the count list (*k*). *n* and *k* are usually different sizes but it is important for efficiency that the value of *k* is not significantly larger than the value of *n*.[27] For instance, if the input has just 10 elements but the maximum value is 5,000, the algorithm would not work very efficiently as it would be creating a counting array of 5,001 indexes to represent every value between 0-5,000 for just 10 elements.

Space Complexity

The algorithm's space complexity is the same as its time complexity at *O(n + k)*. It requires memory to make a list to store the counts (*k*) and another for the sorted values (*n*).

Stability

Counting Sort maintains the stability of the original input values as when it is counting occurrences of unique values it maintains their relative order in a linked list.[28]

In-Place Sorting

This algorithm does not sort data in-place but creates a new list of sorted data from the values in the count list.

Suitability

Because it depends on using indexes to count the unique elements in the input, Counting Sort can only sort positive integers but does so very efficiently. It works best when the maximum value (*k*) is not significantly larger than the number of input elements (*n*). Often Counting Sort is used as part of another non-comparison integer sorting algorithm, Radix Sort, which can handle larger values of *k* more efficiently.[29]

3.5.3 Implementation

The code for Counting Sort has been implemented with Python as follows[30] (see Jupyter notebook for commented code):

```
def counting_sort(alist, max_val=100):
    m = max_val + 1
    count = [0] * m

    for a in alist:
        count[a] += 1

    i = 0
    for a in range(m):
        for c in range(count[a]):
            alist[i] = a
            i += 1

    return alist
```

4.0 Benchmarking

The benchmarking process begins with the generation of an array of arrays of different sizes that contain random integers. These arrays are then passed to each sorting algorithm one by one and the time taken for each one to perform the sort is recorded. This is repeated ten times for each input size to calculate average times, which are then displayed in a table and plot.

The Python code was implemented as follows:

- The function *random_array(n)* returns an array of size *n*. The array contains random integers between the values of 0-100 inclusive. The size of *n* ranges from 100 to 10,000 elements and is stored in the variable *n_size*.
- The function *all_arrays()* takes *n_size* as a parameter and returns an array of arrays equal to the specified input sizes. For instance, if an *n_size* of [100, 200, 300] is passed to the function, it returns an array of three arrays that contain 100, 200 and 300 random integers respectively. The function uses *random_array()* to generate the random numbers.
- Function *benchmarking()* takes a sorting function as an argument and returns the time taken for it to sort a series arrays produced by *all_arrays()*.
- Function *average_time()* runs the *benchmarking()* function for a sorting function ten times and returns the average time taken for each input size (*n_size*).
- Function *all_averages()* collates the benchmarking results of each sorting algorithm in a single array. This array is then used to create a Pandas dataframe to present the findings and create plots.[31] [32]

While the focus of this project is on benchmarking the five sorting algorithms with random arrays, the following were also examined:

- Python's built-in sorted() function is added to the mix to demonstrate how the hybrid algorithm behind it, TimSort, fares against the algorithms addressed here.
- Smaller random and sorted arrays were used to demonstrate how differently the algorithms manage such inputs.

5.0 Discussion

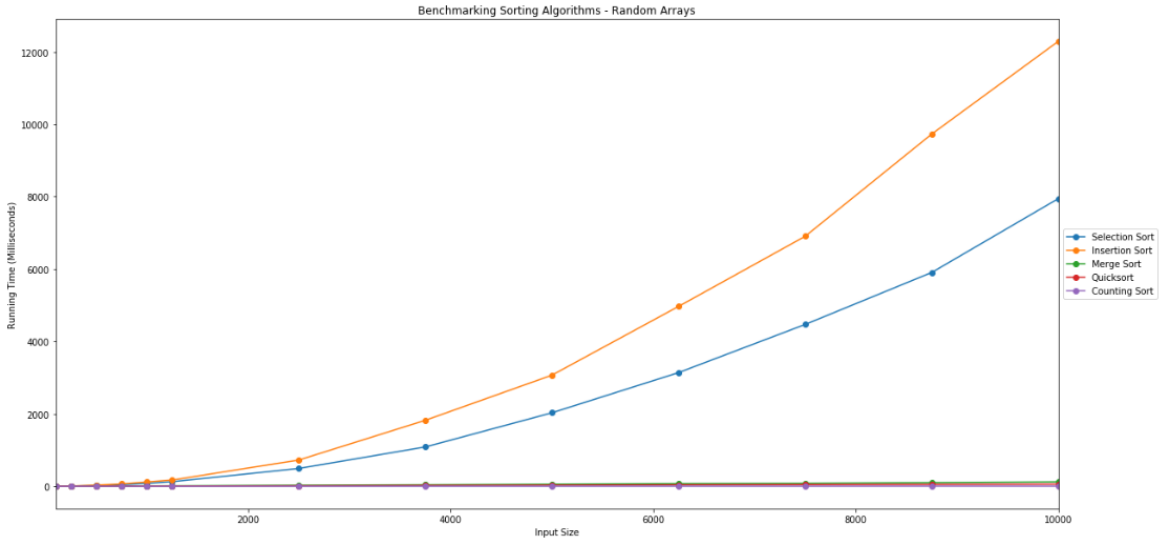
Based on what is known about the best, worst and average time complexities for each algorithm as well as how they operate, several assumptions were made:

- Selection Sort will perform better than Insertion Sort for large random arrays.
- Insertion Sort will perform better than Selection Sort with smaller and sorted arrays.
- Quicksort will perform better than Merge Sort in all input cases.
- The efficient comparison-based algorithms will outperform the simple comparison-based algorithms for large random arrays.
- Counting sort will vastly outperform all comparison-based algorithms with large random arrays.
- Differences between algorithms will be less stark for smaller and sorted arrays.

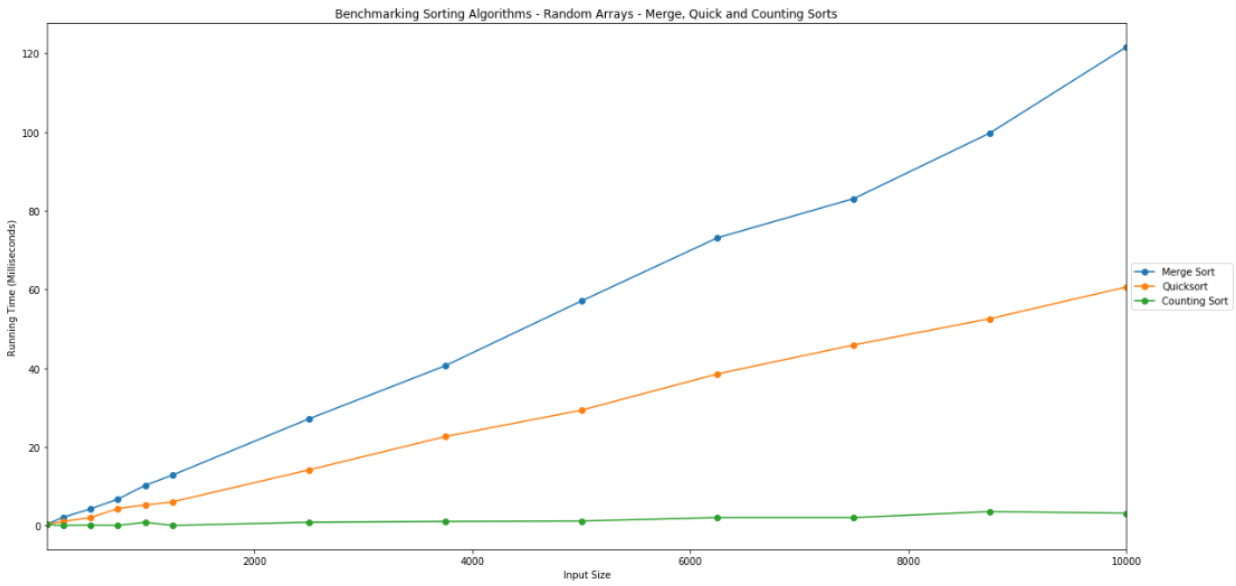
The dataframe below contains the results of benchmarking the five sorting algorithms with random integer arrays of various sizes:

	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
Selection Sort	1.408	5.083	20.449	44.982	80.039	126.315	490.922	1085.357	2029.152	3138.620	4468.785	5904.450	7944.112
Insertion Sort	1.003	5.587	31.994	64.625	119.241	171.837	724.134	1822.246	3068.668	4967.182	6903.063	9727.334	12288.765
Merge Sort	0.399	2.195	4.289	6.782	10.271	12.867	27.129	40.598	57.043	73.118	83.063	99.730	121.549
Quicksort	0.396	1.106	2.095	4.379	5.290	6.080	14.166	22.634	29.334	38.552	45.912	52.578	60.622
Counting Sort	0.301	0.100	0.200	0.100	0.895	0.100	0.899	1.098	1.196	2.094	2.094	3.592	3.187

These results are plotted below:



Unsurprisingly, Selection and Insertion Sorts are significantly slower than the three other algorithms by several orders of magnitude, taking thousands of milliseconds for larger input sizes while the others are largely in double digits only. The difference is so stark, that it is nearly impossible to distinguish between Merge, Quick and Counting Sorts in the plot above. These are plotted separately to visualise their relationships more clearly:



5.1 Time Complexities

It is interesting to see that the sorting algorithms which share the same time complexities trend in the same directions in these plots. Selection Sort is consistently faster than Insertion Sort even though they share $O(n^2)$ time complexity in most cases and Quicksort is faster than Merge Sort across all input sizes despite their largely shared $O(n \log n)$ efficiency. Visualising these figures in the plots above show that the algorithms that share time complexity measurements are quite well-aligned with one another, reflecting similar growth rates. Thus, while time complexity does not translate to actual runtimes, it does reflect the rate that time increases as input sizes grow. The individual algorithmic operations and implementations have more of an impact on the actual time they take to run and likely accounts for the differences between similarly efficient algorithms.

5.2 Selection Sort and Insertion Sort

Even though Selection Sort is generally considered to be a slower algorithm than Insertion Sort,[33] it achieves faster running time across all large random inputs in this investigation. This demonstrates Insertion Sort's weakness when it comes to large input arrays of completely random data.

	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
Selection Sort	1.408	5.083	20.449	44.982	80.039	126.315	490.922	1085.357	2029.152	3138.620	4468.785	5904.450	7944.112
Insertion Sort	1.003	5.587	31.994	64.625	119.241	171.837	724.134	1822.246	3068.668	4967.182	6903.063	9727.334	12288.765

While they start out with very similar running times when *n* sizes are smaller (100 / 250), the gap between them grows along with the input size, until there's more than four seconds between them when sorting 10,000 elements. However, neither can be said to have sorted these large arrays very efficiently relative to the other algorithms and it is their inefficiency with such input that causes the code to run very slowly. These simple algorithms were not designed to manage such large inputs and so should not be judged as poor algorithms because of the results shown here. They can be very effective with other input types, particularly small arrays and near-sorted data.

5.3 Merge Sort and Quicksort

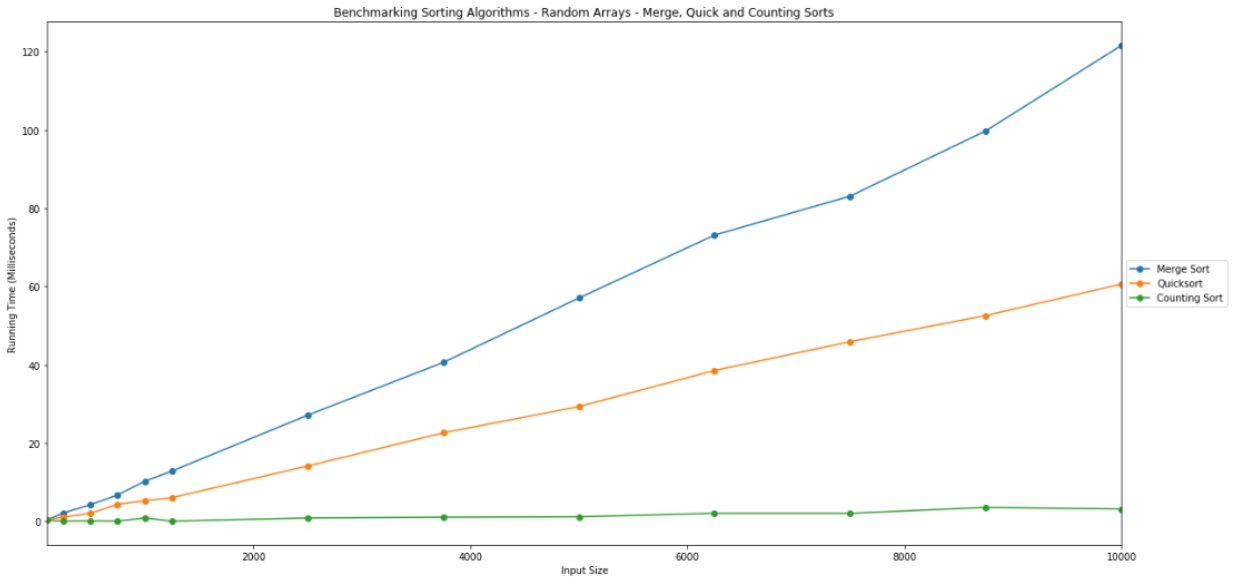
As expected, the more efficient comparison-based algorithms outperform the simple algorithms. Between them, however, Quicksort is consistently faster than Merge Sort in all cases of the large random array input.

	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
Merge Sort	0.399	2.195	4.289	6.782	10.271	12.867	27.129	40.598	57.043	73.118	83.063	99.730	121.549
Quicksort	0.396	1.106	2.095	4.379	5.290	6.080	14.166	22.634	29.334	38.552	45.912	52.578	60.622

The trend is similar to the differences between Insertion and Selection Sorts, with comparable times achieved during smaller input instances but growing further apart at the input gets larger. With the largest input size of 10,000 elements, Quicksort is twice as fast as Merge Sort. This is to be expected considering it is more suitable for large arrays than Merge Sort, which would be better applied to data such as linked lists. Quicksort's lower memory use may also have an effect on the running times. It is the fastest comparison-based sorting algorithm assessed in this project, beaten out only by Counting Sort. However, Quicksort does not need the extra information about the data that is required by Counting Sort.

5.4 Counting Sort

Counting Sort looks almost like a straight line in the plot relative to the comparison-based algorithms, taking just over 3ms for the largest input size of 10,000.



To put that into perspective, the simple algorithms take much longer to sort just 250 elements and Merge Sort and Quicksort take slightly longer to sort 500 and 750 elements respectively.

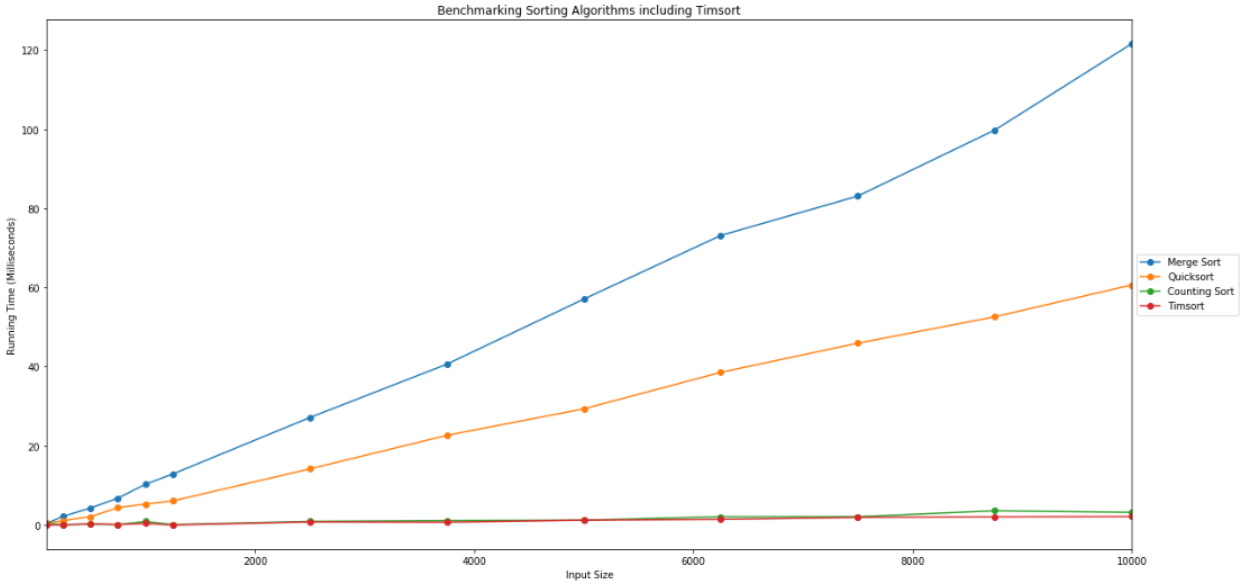
	100	250	500	750
Selection Sort	1.408	5.083	20.449	44.982
Insertion Sort	1.003	5.587	31.994	64.625
Merge Sort	0.399	2.195	4.289	6.782
Quicksort	0.396	1.106	2.095	4.379

The high efficiency being achieved by Counting Sort is likely down to the *k* value, which is set to 100 across all input sizes as that is the highest number generated by the *random_array()* function. The low *k* value relative to the *n* value allows Counting Sort to work much more efficiently as it only ever has to create a count list of 101 indexes (zero is included), even for arrays with several thousand elements.

5.5 Timsort

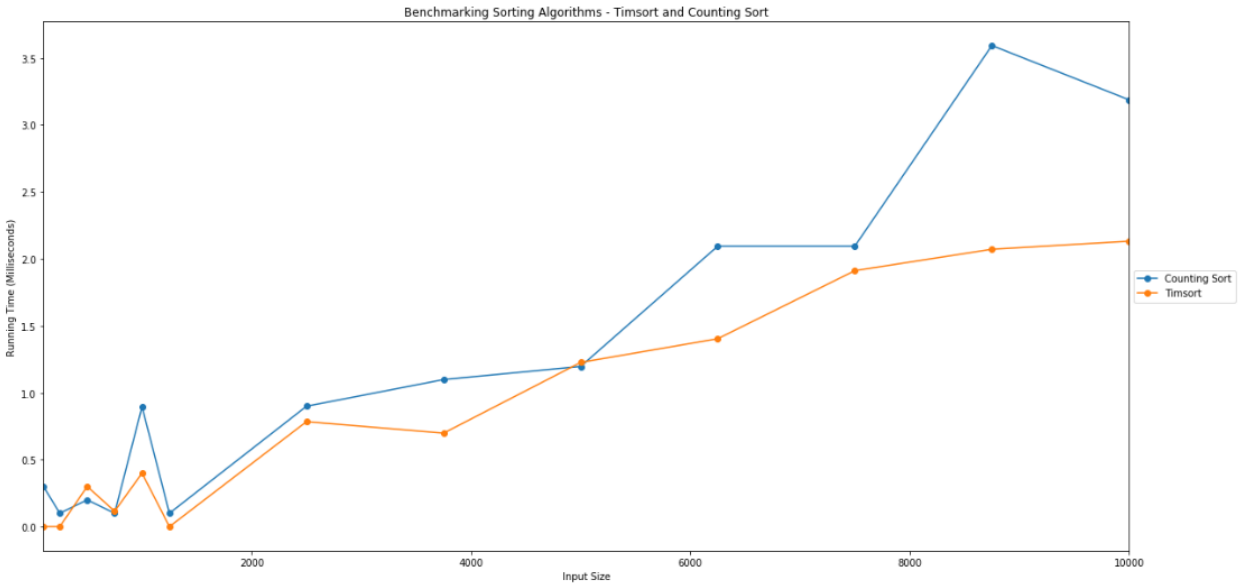
Python has a built-in sorting function called *sorted()* and it may be interesting to see how it fares against the other algorithms. It is based on Timsort, a hybrid sorting algorithm that combines Merge Sort and Insertion Sort. It works by finding smaller subsequences (runs) of ordered data in the input, using Insertion Sort to add elements to a run until it reaches a certain size and then merging runs using Merge Sort.[34] In the worst and average cases, it runs at *O(n log n)* efficiency and linear *O(n)* in the best case.

Adding the average benchmarked times achieved by the *sorted()* function shows it is well-aligned with Counting Sort in comparison to the efficient algorithms:



Looking more closely, Timsort is faster than Counting Sort across most input sizes:

	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
Counting Sort	0.301	0.1	0.2	0.100	0.895	0.1	0.899	1.098	1.196	2.094	2.094	3.592	3.187
Timsort	0.000	0.0	0.3	0.115	0.399	0.0	0.784	0.698	1.226	1.403	1.911	2.071	2.131

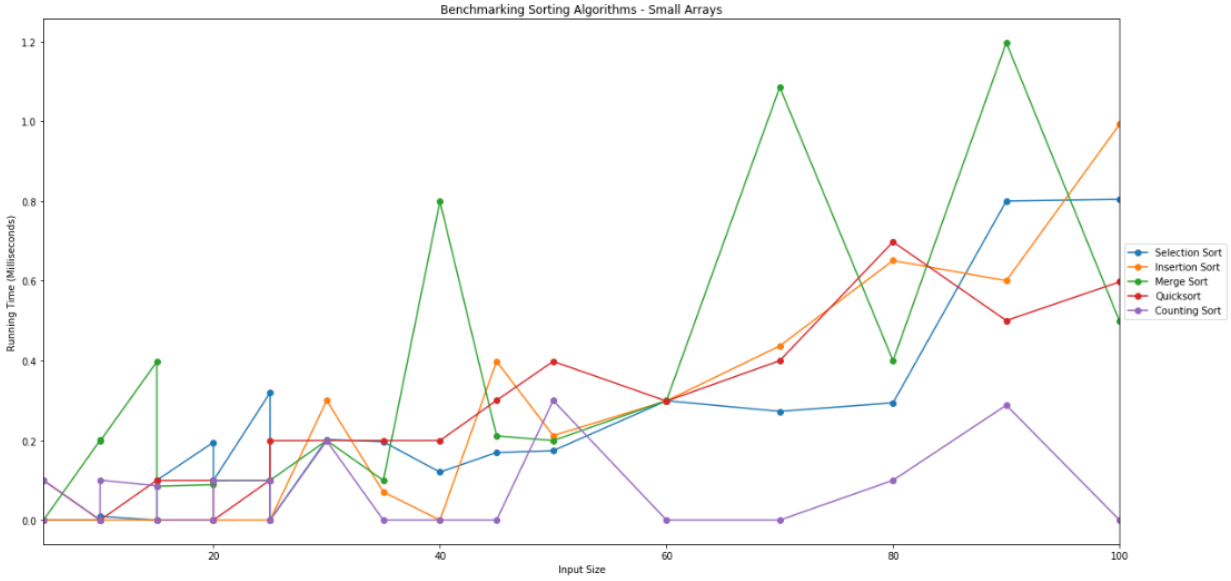


This reflects the extreme speed of this highly optimised hybrid algorithm which, not only outperforms a non-comparison algorithm that has more knowledge about the input data, but also beats the extremely fast Quicksort by a factor of 30 with the largest input size. It is thus unsurprising that it was chosen to be the default sorting method of the Python programming language.

5.6 Small Input Size

It is interesting to see that when benchmarking is run with smaller input sizes, the stark differences previously observed between the five sorting algorithms all but disappear:

	5	5	10	10	15	15	20	20	25	25	30	35	40	45	50	60	70	80	90	100
Selection Sort	0.1	0.0	0.0	0.010	0.000	0.100	0.195	0.099	0.319	0.000	0.202	0.196	0.120	0.170	0.174	0.299	0.273	0.294	0.800	0.804
Insertion Sort	0.0	0.0	0.0	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.300	0.071	0.000	0.398	0.212	0.299	0.437	0.651	0.600	0.992
Merge Sort	0.0	0.0	0.2	0.198	0.397	0.085	0.089	0.098	0.100	0.100	0.200	0.099	0.799	0.211	0.199	0.300	1.085	0.399	1.197	0.498
Quicksort	0.0	0.1	0.0	0.000	0.100	0.099	0.100	0.000	0.100	0.199	0.199	0.199	0.199	0.300	0.398	0.298	0.400	0.698	0.500	0.597
Counting Sort	0.0	0.1	0.0	0.100	0.086	0.000	0.000	0.100	0.099	0.000	0.199	0.000	0.000	0.000	0.300	0.000	0.000	0.100	0.288	0.000



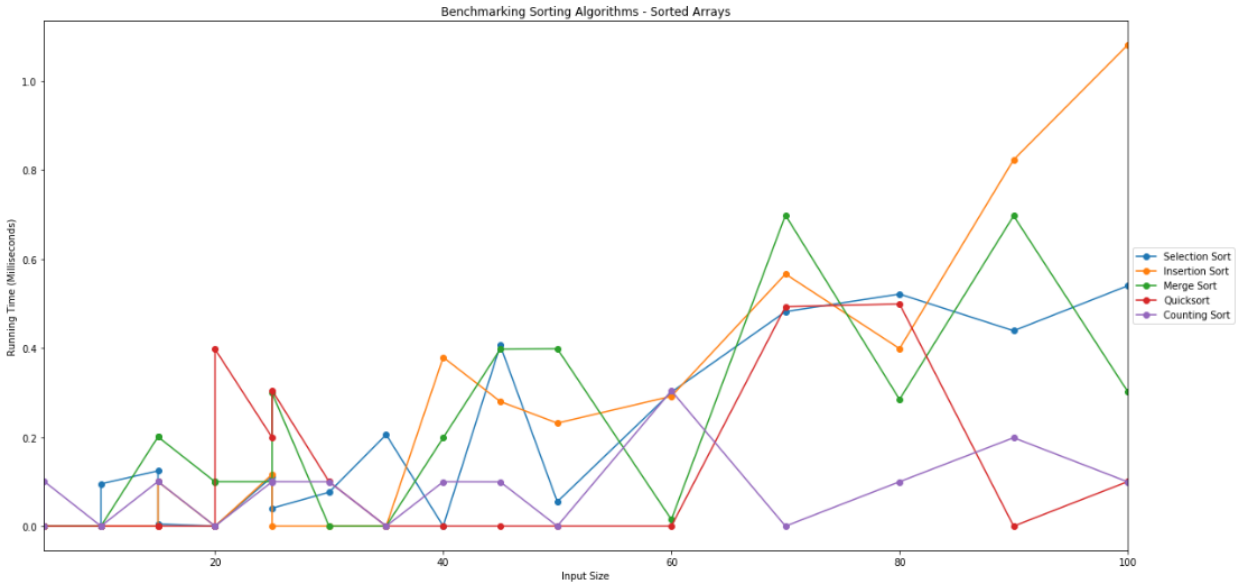
Insertion and Selection Sorts are more closely aligned, with Insertion Sort generally running in the same amount of time or faster than Selection Sort up to an input size of about 40 elements. As mentioned previously, Merge Sort and Quicksort are less efficient with smaller input sizes and that is certainly reflected here. Counting Sort appears to be the only algorithm that maintains quite low runtimes.

While it could not be said that the simple comparison-based algorithms vastly outperform the more efficient ones with these input sizes, it does illustrate why these algorithms should be chosen for such cases. It makes more sense to use simpler code that requires less memory for small inputs if the runtime is essentially the same. Using more efficient algorithms such as Merge Sort or Quicksort would be overkill in these cases as their recursive nature is quite expensive memory-wise and there would be no added efficiency time-wise; it is best to save such algorithms for larger data inputs. In fact, Merge Sort in particular does not perform very well compared to Insertion and Selection Sorts, often being beaten by both and reinforcing its unsuitability for such input.

5.7 Sorted Input

To examine how the algorithms manage sorted data, the arrays of random integers were replaced with arrays of completely sorted integers, using the same small input sizes as above.

	5	5	10	10	15	15	20	20	25	25	30	35	40	45	50	60	70	80	90	100
Selection Sort	0.0	0.000	0.0	0.095	0.124	0.005	0.0	0.000	0.111	0.040	0.076	0.206	0.000	0.406	0.055	0.301	0.482	0.521	0.439	0.540
Insertion Sort	0.0	0.000	0.0	0.000	0.000	0.100	0.0	0.000	0.116	0.000	0.000	0.000	0.379	0.280	0.231	0.292	0.566	0.399	0.823	1.081
Merge Sort	0.0	0.000	0.0	0.000	0.201	0.201	0.1	0.100	0.100	0.299	0.000	0.000	0.199	0.397	0.399	0.015	0.699	0.285	0.697	0.302
Quicksort	0.0	0.000	0.0	0.000	0.000	0.000	0.0	0.398	0.199	0.305	0.100	0.000	0.000	0.000	0.000	0.000	0.493	0.499	0.000	0.100
Counting Sort	0.0	0.101	0.0	0.000	0.100	0.100	0.0	0.000	0.100	0.100	0.100	0.000	0.100	0.099	0.000	0.304	0.000	0.100	0.199	0.099



Again, the simple algorithms fare quite well in comparison to the others and Insertion Sort marginally beats Selection Sort with the smaller input sizes up to about 40 elements. Merge Sort performs even more poorly than with the randomised small arrays and does not get substantially better as the input size increases, remaining in line with the simple algorithms throughout. This reflects its unsuitability for both small and sorted input arrays.

It should be noted that Counting Sort is not running as efficiently as it could with these small sorted input cases as the *k* value was not altered in the function's code. It is hardcoded as 100, even though most of the smaller arrays do not have such a range. It was decided to leave the code unchanged as this benchmark was implemented to illustrate the strengths of the simpler algorithms, not reinforce the power of Counting Sort, which has already been well-established.

6.0 Conclusion

The benchmarking results for the five algorithms align with expectations in terms of their time complexities and operations. The simple algorithms, Insertion and Selection Sorts, take quite a long time to sort large random inputs as they are not designed to manage this type of data. Their usefulness is observed with smaller and/or sorted arrays. Merge Sort and Quicksort are much more efficient at dealing with large randomised input data, and therefore operate faster. Lastly, Counting Sort runs at much higher speeds than any of the comparison-based algorithms, but this is offset by its requirement to know the maximum value of the data. It could be argued that, when dealing with large inputs of random integers, Quicksort should be the algorithm of choice as it has excellent runtimes, even with a maximum *O(n log n)* time complexity, uses very little memory, and does not require any special knowledge about the input array.

7.0 References

[1] Asymptotic notation: https://www.programiz.com/dsa/asymptotic-notations

[2] O-notation: https://medium.com/cantors-paradise/basics-of-big-o-sorting-94d0c04d0f53

[3] Different types of memory used by algorithms: https://stackoverflow.com/a/36363720

[4] Time complexity as statements executed: https://medium.com/@info.gildacademy/time-and-space-complexity-of-data-structure-and-sorting-algorithms-588a57edf495

[5] Selection Sort procedure: https://stackabuse.com/selection-sort-in-python/

[6] Implementation of Selection Sort using maximum values: https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheSelectionSort.html

[7] Selection Sort space complexity: https://en.wikipedia.org/wiki/Selection_sort

[8] Selection Sort versus Insertion Sort study:
Furat, G. F. (2016) *A Comparative Study of Selection Sort and Insertion Sort Algorithms.* International Research Journal of Engineering and Technology (vol. 3, no. 12) Available from: https://www.irjet.net/archives/V3/i12/IRJET-V3I12115.pdf

[9] Selection Sort and sorted lists: https://itnext.io/selection-sort-in-a-nutshell-how-when-where-932275135c00

[10] Selection Sort code adapted from: https://www.geeksforgeeks.org/selection-sort/

[11] Difference between Selection and Insertion Sort iterations: https://stackoverflow.com/a/15799532

[12] Insertion sort used as part of other algorithms: https://en.wikipedia.org/wiki/Insertion_sort

[13] Insertion Sort code adapted from (1): https://www.geeksforgeeks.org/insertion-sort/

[14] Insertion Sort code adapted from (2): https://www.pythoncentral.io/insertion-sort-implementation-guide/

[15] Merge Sort and nearly sorted data: https://www.toptal.com/developers/sorting-algorithms/nearly-sorted-initial-order

[16] Merge Sort uses over Quicksort (1): https://www.tutorialcup.com/interview/linked-list/merge-sort-better-quick-sort-linked-lists.htm

[17] Merge Sort uses over Quicksort (2): https://stackoverflow.com/a/7629969

[18] Merge Sort code adapted from (1): https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheMergeSort.html?highlight=merge%20sort

[19] Merge Sort code adapted from (2): https://www.geeksforgeeks.org/merge-sort/

[20] Quicksort pivot choice: https://towardsdatascience.com/essential-programming-sorting-algorithms-76099c6c4fb5

[21] Quicksort speed: https://medium.com/human-in-a-machine-world/quicksort-the-best-sorting-algorithm-6ab461b5a9d0

[22] Quicksort faster runtimes than Merge Sort: https://stackoverflow.com/questions/70402/why-is-quicksort-better-than-mergesort

[23] Efficient partitioning in Quicksort: https://www.codementor.io/@garethdwyer/quicksort-tutorial-python-implementation-with-line-by-line-explanation-p9h7jd3r6

[24] Quicksort code adapted from: https://stackabuse.com/sorting-algorithms-in-python/#quicksort

[25] Counting Sort as integer sorting algorithm: https://en.wikipedia.org/wiki/Counting_sort

[26] Comparison-based algorithms best time complexity: https://www.geeksforgeeks.org/lower-bound-on-comparison-based-sorting-algorithms/

[27] Importance of a lower pivot in Counting Sort: https://medium.com/basecs/counting-linearly-with-counting-sort-cd8516ae09b3

[28] Counting sort stability: https://stackoverflow.com/a/2572215

[29] Counting Sort as part of Radix Sort: https://en.wikipedia.org/wiki/Counting_sort

[30] Counting Sort code adapted from https://www.w3resource.com/python-exercises/data-structures-and-algorithms/python-search-and-sorting-exercise-10.php

[31] (1) Plotting code adapted from https://stackoverflow.com/a/31734795

[32] (2) Plotting code adapted from https://stackoverflow.com/a/43610256

[33] Selection Sort slower than Insertion Sort:
Heineman, G. T., Pollice, G. & Selkow, S. (2009). *Algorithms in Nutshell*. California: O'Reilly Media. p. 92

[34] Timsort description: https://en.wikipedia.org/wiki/Timsort