## Table of Contents

## Introduction

The concept of sorting is regarded as the process of ordering of a series of objects relative to some predefined parameters e.g., sorting a deck of cards based on their picture card and their numbers in descending order i.e., from 10 to 2, followed by an ace. Another example of sorting being that of ordering a series of books on a bookshelf by the authors surname in alphabetical order. With reference to the theme of efficiency, the utilisation of approaches that adopt the process of sorting, can greatly benefit one's efficiency, in contrast to approaches that have not adopted an aspect related to the theme of order. Further expanding on the example above regarding sorting numerous books by the authors surname in alphabetical order, having undergone the process of sorting, the element of randomisation is then removed from the equation, when one wishes to search for a particular book (Matzinger, 2019).

## Defining a sorting algorithm

While cognisant of the fact that an algorithm can be defined as a series of step-by-step instructions that lead one towards obtaining a desired outcome (Krishna, 2016, pg.29) e.g., a search algorithm that displays the skillset of a company's employees and the number of years of experience that each employee has. A sorting algorithm may then be regarded as a series of step-by-step instructions of which the processing of a particular input occurs and based on the prior specification of ones needs, a desired outcome is then achieved as the algorithm engages in the process of sorting.

## Time and space efficiency

Just as there are several recipes for various chicken dishes, there are also various algorithms that engage in the process of sorting. The question that is laid to us here is that of, which algorithm should we select for our particular purpose?

Various algorithms have different space and time efficiency. Time efficiency takes into consideration the system runtime or number of operations required for an algorithm to be run, while space efficiency takes into consideration the amount of memory or storage required by the computer in order to run an algorithm. It is also beneficial to be cognisant of the fact that the parameters of system runtime and memory requirement are affected by external environmental factors such as: the handling of data structures, system software and hardware configurations, the style of writing and compiling codes and the programming language utilised (Krishna, 2016, pg.30).

## A Priori analysis and A posteriori analysis

With reference to the theme of efficiency within the realm of algorithm analysis. It is possible to analyse an algorithm for its efficiency through two various means of analysis, A priori or A posteriori.

A priori analysis, is the process of evaluating an algorithms efficiency from a theoretical perspective i.e., Asymptotic analysis (Krishna, 2016, pg.301). Asymptotic analysis is based on the principle that that growth rate of the input data is directly proportional to the system runtime (Krishna, 2016, pg.301). When implementing this approach, the effect of implementation details is discarded from the equation i.e., one's processor or system architecture.

A posteriori analysis, in contrast to a priori analysis, implements the process of evaluating the efficiency of an algorithm empirically. In such circumstances algorithms are implemented and executed in order to evaluate their performance i.e., the system runtime. Empirical analysis is performed on the underlying assumption that the system properties and configuration remain the same for all the running algorithms under consideration (Krishna, 2016, pg.130).

## Stable, hybrid and in place-sorting

Having previously brought this question to mind, which algorithm should we select for our particular purpose? The analysis of an algorithm's asymptotic runtime complexity can provide direction with regards to obtaining an answer to the question posed above (Matzinger, 2019, pg.193). Sorting algorithms are categorised based upon their properties. Stable sorting algorithms maintain a relative order when comparing equal values i.e., if $i < j$, then the final location for A[i] must be to the left of the final location for A[j]. Hybrid sorting algorithms combine two or more sorting approaches e.g., Introsort and Timsort. In-place sorting algorithms use indices instead of full copies for passing collections around i.e., it uses only a fixed additional amount of working space, independent of the input size (Matzinger, 2019, pg.204).

## Comparison-based and non-comparison-based sorting

Sorting algorithms can also be categorised, based on the type of sorting that they engage with, that of which can be comparison-based sorting or non-comparison-based sorting. Comparison based sorting occurs when all the key values of an input e.g., an array of size n, are directly compared with each other prior to engaging in the process of ordering (Krishna, 2016, pg.102).

In contrast to comparison-based sorting, non-comparison-based sorting takes place when computations are performed on each key value, and then the ordering is performed based on the computed values (Krishna, 2016, pg.102).

## Big O notation

When engaging in the process of asymptotic analysis in order to measure and distinguish the efficiency level of a particular algorithm, the utilisation of Big O notation (Landau notation) often occurs (Matzinger, 2019, pg.192). Big O notation allows for the running time and space complexity of an algorithm to be expressed in a simplified and easily comprehensible manner.

*The table below provides an insight into how this simplification process occurs*

| Running Time | Big-O notation |
|---|---|
| ***$2n+7$*** | *$O(n)$* |
| ***$N^2+6n+25$*** | *$O(n^2)$* |
| ***$45n^4+n^3+4$*** | *$O(n^5)$* |

*Table 1*

## Best average and worst cases

There are three possible cases that are taken into consideration when evaluating the time complexity of an algorithm; they are worst, average, and best cases (Anggoro, 2018). These cases arise due to the fact that different inputs may produce a series of varying results with reference to the domains of space and time (Erik, Alebicto, 2016, pg. 223).

Worst case analysis is a calculation of the upper bound on the running time of an algorithm (Anggoro, 2018, pg.45) e.g., this algorithm may take up to two minutes to conclude. With reference to a searching algorithm, it may potentially occur that all elements within a data structure were iterated through and the item at which one was looking for was not found.

Average case analysis is a calculation of all possible inputs on the running time of an algorithm (Anggoro, 2018, pg.45), including an element that is not present in an array that is being searched. With reference to a searching algorithm, a desired item may be in any position within a data structure with equal probability.

Best case analysis is a calculation of the lower bound on the running time of the algorithm (Anggoro, 2018, pg.45) i.e., a case that requires the least amount of running time. With reference to a searching algorithm, when iterating through a data structure, while seeking to find a desired item, such an item is then found on the first iteration.

## Sorting Algorithms

The following section will introduce a series of sorting algorithms and discuss their space and time complexity, alongside their explanation a series of diagrams will be provided in order to depict how each algorithm operates i.e., what processes are occurring as the algorithm itself engages in the act of sorting a particular input e.g., an array of size n. Having previously addressed the concepts of comparison-based and non-comparison-based sorting algorithms, the algorithms dealt with in this report will be categorised by the following properties.

The following algorithms will be addressed throughout the course of this report:

- Two Simple comparison-based sorts i.e., **Bubble Sort** and **Selection Sort**
- Two Efficient comparison-based sorts i.e., **Merge Sort** and **Heap Sort**
- A non-comparison-based sort i.e., **Counting Sort**

### Bubble Sort

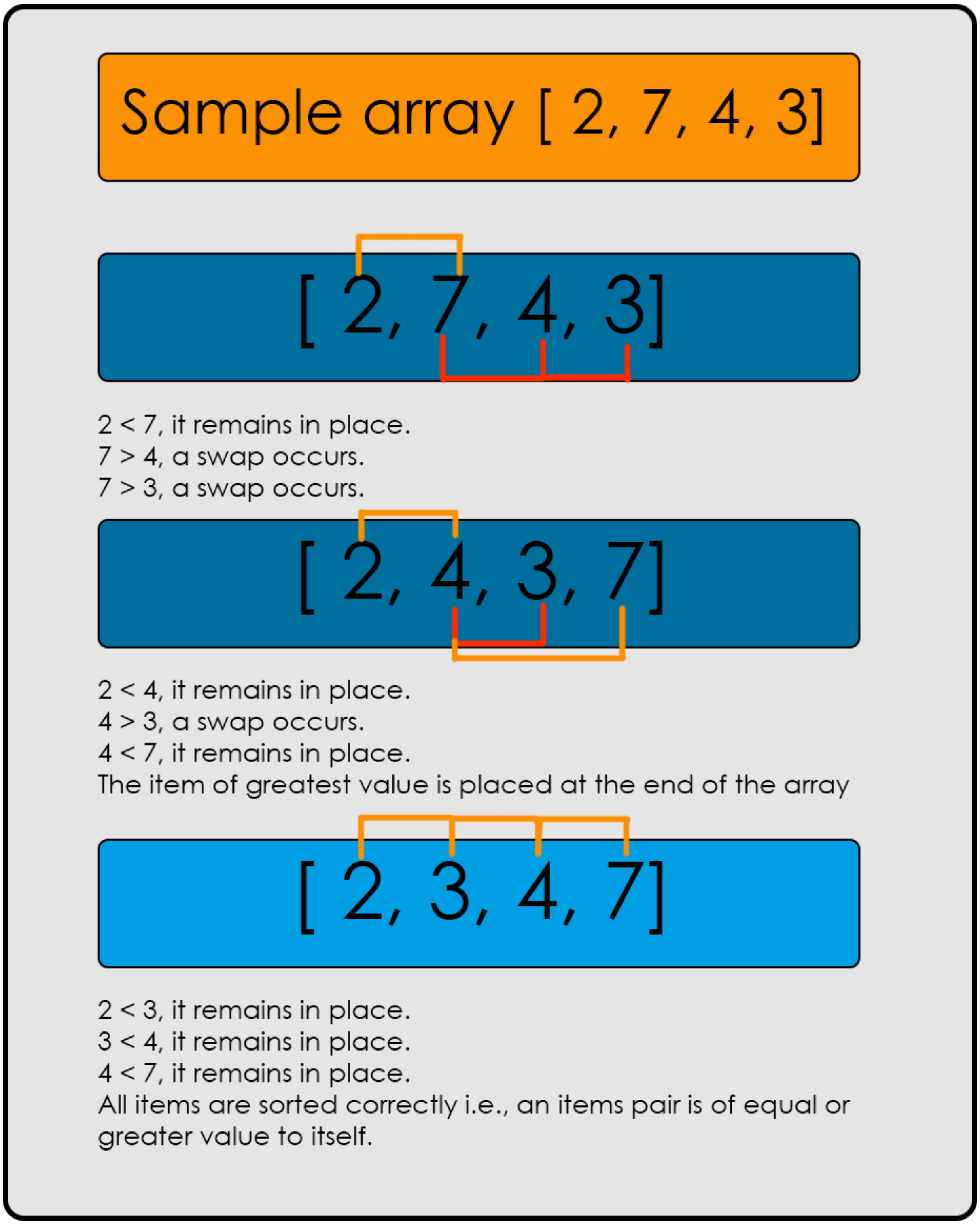
As previously addressed bubble sort is a simple comparison-based sorting algorithm i.e., all the key values of an input e.g., an array of size n, are directly compared with each other prior to engaging in the process of ordering (Krishna, 2016, pg.102). With reference to its space and time complexity it is regarded as quiet a poor approach to sorting a data structure in contrast to other sorting algorithms (Matzinger, 2019).

Should one use bubble sort in order to sort an array of n items in ascending order, during the processing or reading of the items within an array, each item is compared to its partner within the array and swapped if the item to its right is of a value less than the item that it is in direct comparison with. Upon reaching the conclusion of the bubble sort algorithm, the item of greatest value is transferred to the right position and then proceeded by the second greatest value; this pattern then continues until all elements have a greater or equal value to their right. The greatest value that is to be sorted is suggested to *bubble up* into its correct position.

When engaging in the process of asymptotic analysis with the sorting algorithm of bubble sort, it is suggested to have the following complexities:

- Best case: O(n)
- Average case: O($n^2$)
- Worst case O($n^2$)

Diagram (1): a depiction of the processes that occur while bubble sort is active:


Sample array [ 2, 7, 4, 3]
[ 2, 7, 4, 3]
2 < 7, it remains in place.
7 > 4, a swap occurs.
7 > 3, a swap occurs.
[ 2, 4, 3, 7]
2 < 4, it remains in place.
4 > 3, a swap occurs.
4 < 7, it remains in place.
The item of greatest value is placed at the end of the array
[ 2, 3, 4, 7]
2 < 3, it remains in place.
3 < 4, it remains in place.
4 < 7, it remains in place.
All items are sorted correctly i.e., an items pair is of equal or greater value to itself.

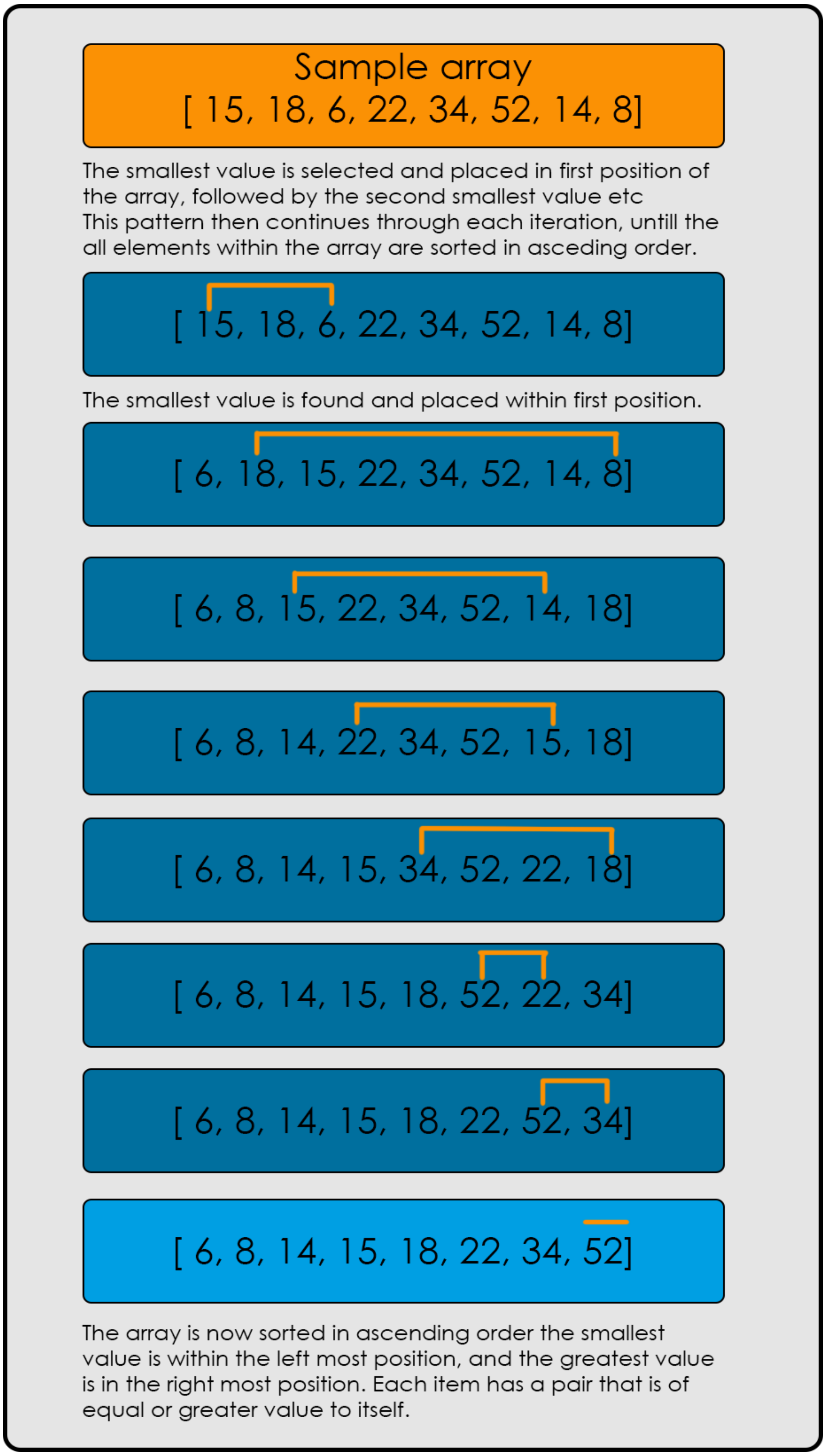
## Selection Sort

Similar to bubble sort, selection sort is also categorised as a simple comparison-based sorting algorithm. In contrast to the approach taken by bubble sort in which the highest value *bubbles up* to the right most position of an input data structure i.e., an array. Selection sort strives to place the smallest value of an item within the first position, followed by the second smallest value of an item of which is then placed in second position. This pattern then continues, until all elements within the given input have been sorted in ascending order (Krishna, 2016, pg.108).

When engaging in the process of asymptotic analysis with the sorting algorithm of selection sort, it is suggested to have the following complexities:

- Best case: $O(n^2)$
- Average case: $O(n^2)$
- Worst case: $O(n^2)$

Diagram (2): a depiction of the processes that occur while selection sort is active:



The smallest value is selected and placed in first position of the array, followed by the second smallest value etc
This pattern then continues through each iteration, untill the all elements within the array are sorted in asceding order.

The smallest value is found and placed within first position.

The array is now sorted in ascending order the smallest value is within the left most position, and the greatest value is in the right most position. Each item has a pair that is of equal or greater value to itself.

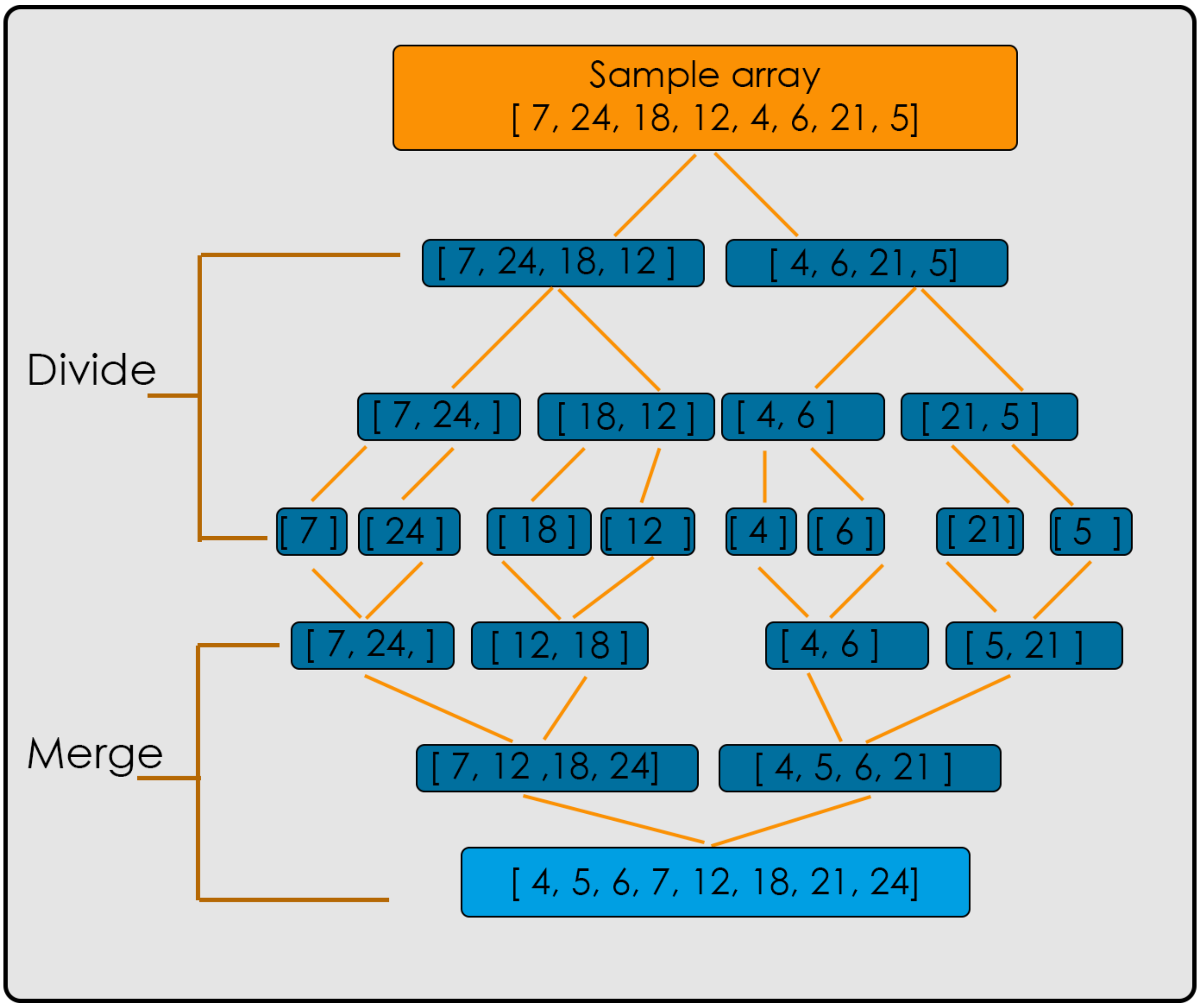
## Merge Sort

Merge sort is classified as an efficient comparison-based sort. This sorting algorithm implements the strategy of divide and conquer. This occurs when the input data structure i.e., an array is split in half recursively until only a single element remains (Matzinger, 2019). Through the process of dividing the input i.e., an array, the input data is split up into a series of chunks of which then allow the processing of sorting to occur more easily. These single elements are then merged together in the correct order e.g., ascending.

When engaging in the process of asymptotic analysis with the sorting algorithm of merge sort, it is suggested to have the following complexities:

- Best case: O(nlog n)
- Average case: O(nlog n)
- Worst case: O(nlog n)

Diagram (3): a depiction of the processes that occur while merge sort is active:



## Heap Sort

Heap sort is of a similar classification to merge sort as it is also regarded as an efficient comparison-based sorting algorithm. Heap sort utilises heaps which are tree-like data structures that contain either the highest (max-heap) or lowest element (min-heap) at its root of which allows ordering to take place when inserting or removing elements (Claus Matzinger, 2019 hands on data structures).

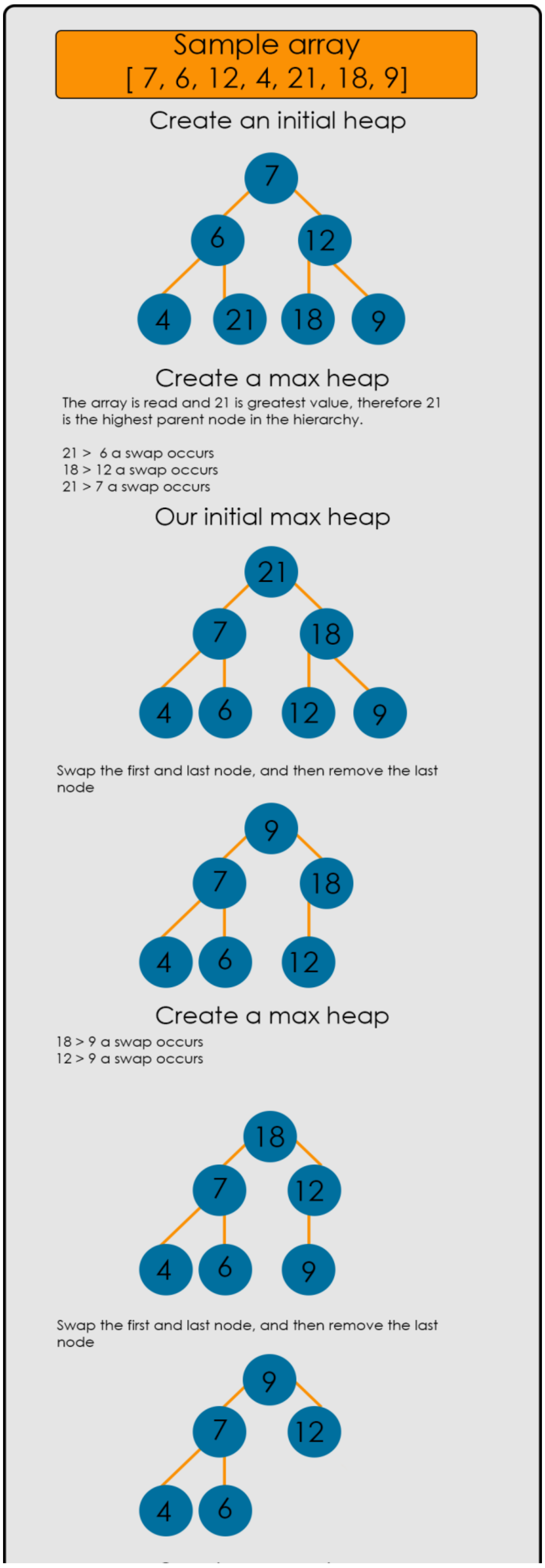
With reference to a max heap, when sorting input i.e., an array of random numbers, the greatest value is moved to the top of the tree in order to establish a max heap. This greatest value is removed from the tree and placed at the end of the array. Upon removing the greatest value i.e., the item in first position, this item is then replaced with the item in last position. This pattern of establishing a max heap is repeated, followed by the act of removing the item of greatest value and replacing it with the item within last position. When there is only one remaining item within the tree, the algorithm is complete, and the processing of sorting stops as all items within the array have been sorted relative to the predefined specifications of the algorithm.

When engaging in the process of asymptotic analysis with the sorting algorithm of heap sort, it is suggested to have the following complexities:

- Best case: O(nlog n)
- Average case: O(nlog n)
- Worst case: O(nlog n)

Contained within the following pages is a diagram of which breaks down the functioning of the sorting algorithm heap sort, please note the diagram itself spans two pages.

Diagram (4): a depiction of the processes that occur while heap sort is active.



Sample array
[ 7, 6, 12, 4, 21, 18, 9]

Create an initial heap

Create a max heap

The array is read and 21 is greatest value, therefore 21 is the highest parent node in the hierarchy.

21 > 6 a swap occurs
18 > 12 a swap occurs
21 > 7 a swap occurs

Our initial max heap

Swap the first and last node, and then remove the last node

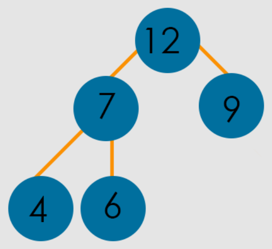
Create a max heap

18 > 9 a swap occurs
12 > 9 a swap occurs

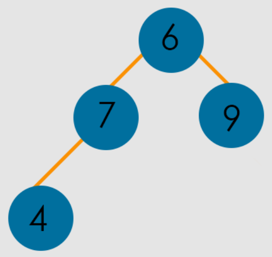
Swap the first and last node, and then remove the last node

## Create a max heap

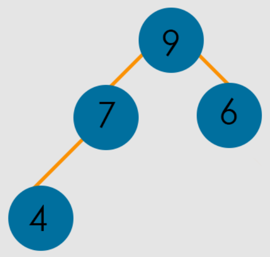
12 > 9 a swap occurs



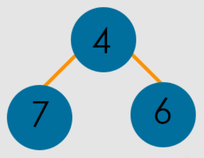
Swap the first and last node, and then remove the last node



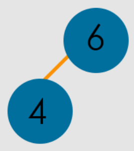
9 > 6 a swap occurs



Swap the first and last node, and then remove the last node



7 > 4 a swap occurs



Swap the first and last node, and then remove the last node

Max heap already established as 6 > 4

Remove the parent node i.e., 6
As there is only one element remaining the algorithm stops.
The array is now sorted in ascending order from smallest to greatest value (from left to right).

Initial array
[ 7, 6, 12, 4, 21, 18, 9]

Sorted array
[ 4, 6, 7, 9, 12, 18, 21]

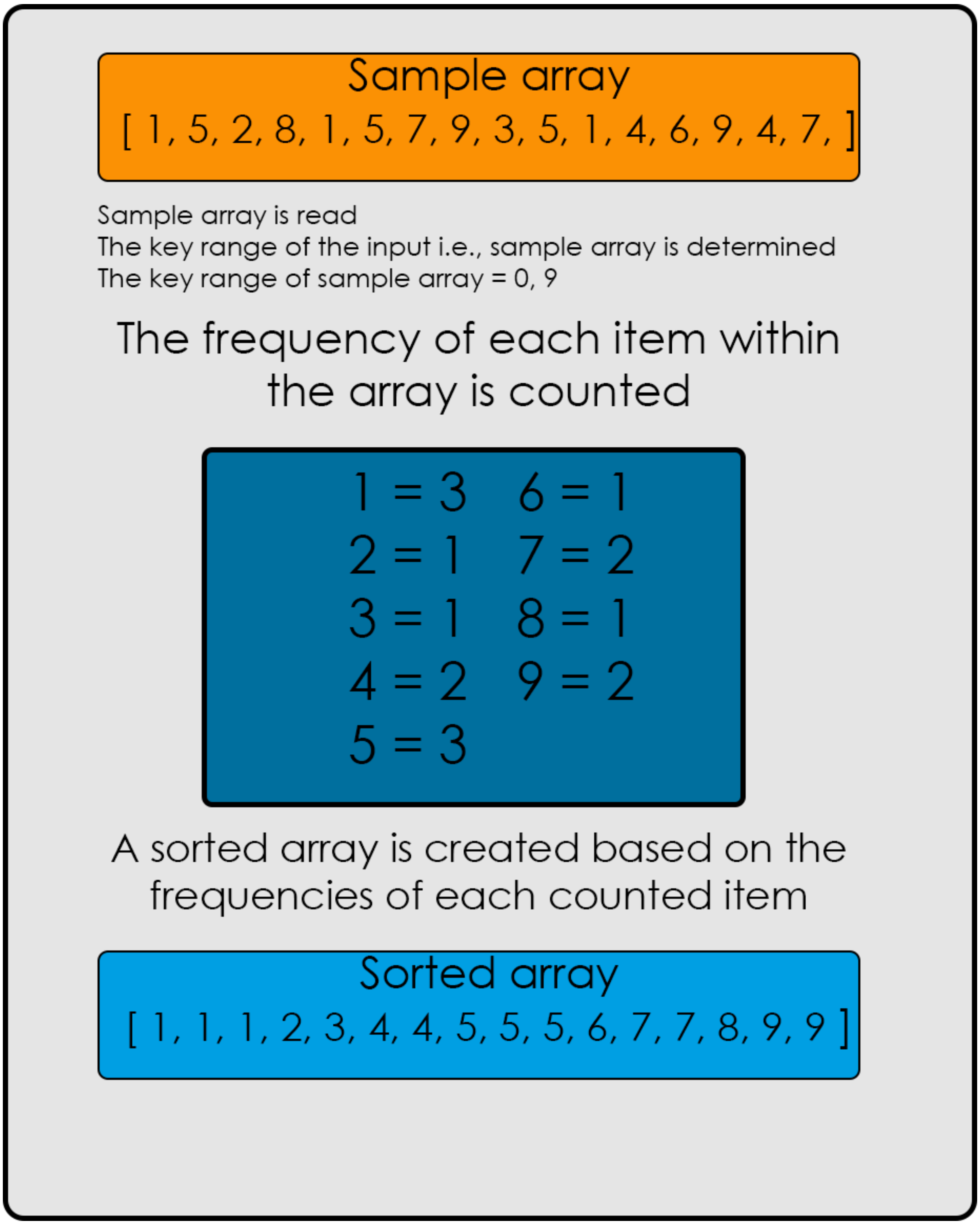
## Counting Sort

Counting sort is classified as a non-comparison-based sorting algorithm i.e., direct comparisons are not made with elements within a data structure. Counting sort engages in the process of arranging items based on a key (Anggoro, 2018). Counting sort calculates the frequency at which each key value occurs within the data structure i.e., an array of randomly generated numbers. This array is then iterated through, the algorithm then outputs the occurrences of a particular item in a sorted order.

When engaging in the process of asymptotic analysis with the sorting algorithm of counting sort, it is suggested to have the following complexities:

- Best case: n + k
- Average case: n + k
- Worst case: n + k

Diagram (5): a depiction of the processes that occur while counting sort is active.


Sample array
[ 1, 5, 2, 8, 1, 5, 7, 9, 3, 5, 1, 4, 6, 9, 4, 7, ]
Sample array is read
The key range of the input i.e., sample array is determined
The key range of sample array = 0, 9
The frequency of each item within the array is counted
1 = 3
2 = 1
3 = 1
4 = 2
5 = 3
6 = 1
7 = 2
8 = 1
9 = 2
A sorted array is created based on the frequencies of each counted item
Sorted array
[ 1, 1, 1, 2, 3, 4, 4, 5, 5, 5, 6, 7, 7, 8, 9, 9 ]

## Summary Table

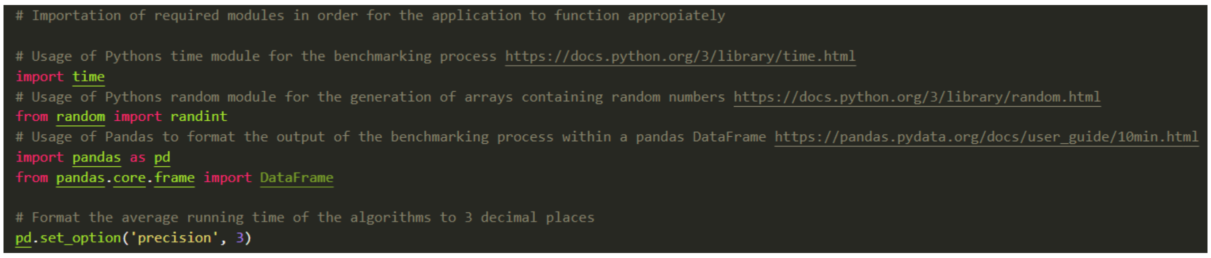
Asymptotic complexities of the sorting algorithms discussed above

| Algorithm | Type of sort | Best case | Average case | Worst case |
|---|---|---|---|---|
| **Bubble sort** | Simple comparison-based | O(n) | O($n^2$) | O($n^2$) |
| **Selection sort** | Simple comparison-based | O($n^2$) | O($n^2$) | O($n^2$) |
| **Merge sort** | Efficient comparison-based | O(nlog n) | O(nlog n) | O(nlog n) |
| **Heap sort** | Efficient comparison-based | O(nlog n) | O(nlog n) | O(nlog n) |
| **Counting sort** | Non-comparison-based | n + k | n + k | n + k |

*Table 2*

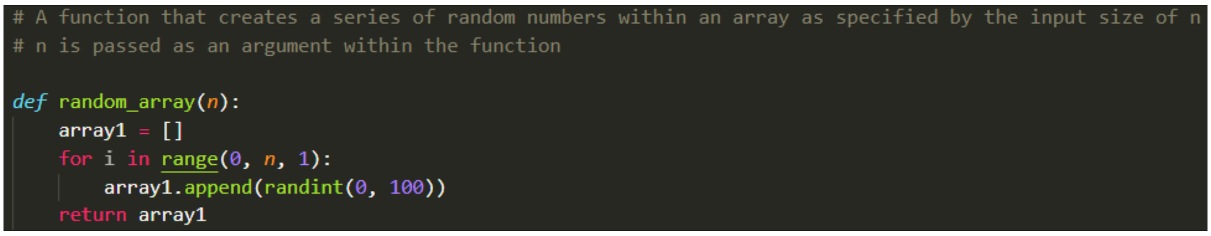
## Implementation and Benchmarking

In order for the application to function as intended, a series of modules were implemented to enhance the functioning of the code and its output.



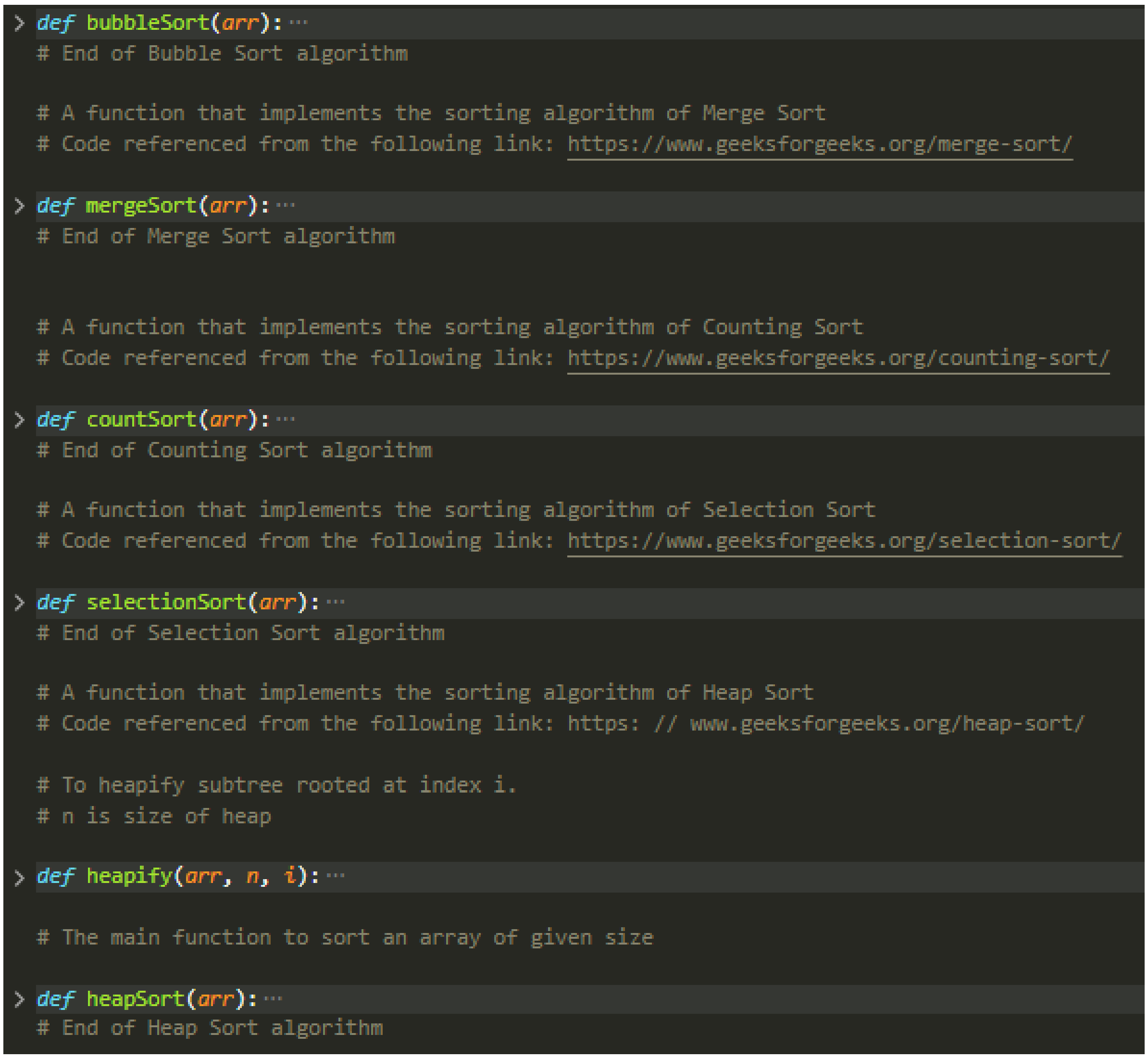
```
# Importation of required modules in order for the application to function appropiately

# Usage of Pythons time module for the benchmarking process https://docs.python.org/3/library/time.html
import time
# Usage of Pythons random module for the generation of arrays containing random numbers https://docs.python.org/3/library/random.html
from random import randint
# Usage of Pandas to format the output of the benchmarking process within a pandas DataFrame https://pandas.pydata.org/docs/user_guide/10min.html
import pandas as pd
from pandas.core.frame import DataFrame

# Format the average running time of the algorithms to 3 decimal places
pd.set_option('precision', 3)
```

The following code was utilised in order to allow the generation of arrays that contain a series of random numbers. This code was established as a function so that a function call could be made.



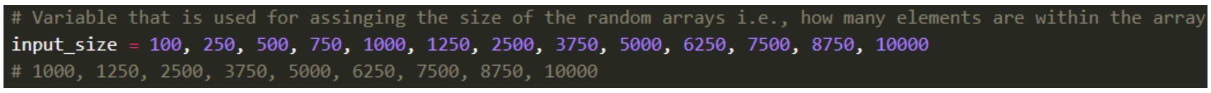
```
# A function that creates a series of random numbers within an array as specified by the input size of n
# n is passed as an argument within the function

def random_array(n):
    array1 = []
    for i in range(0, n, 1):
        array1.append(randint(0, 100))
    return array1
```

A series of functions were created each of which contain code related to a specific sorting algorithm.



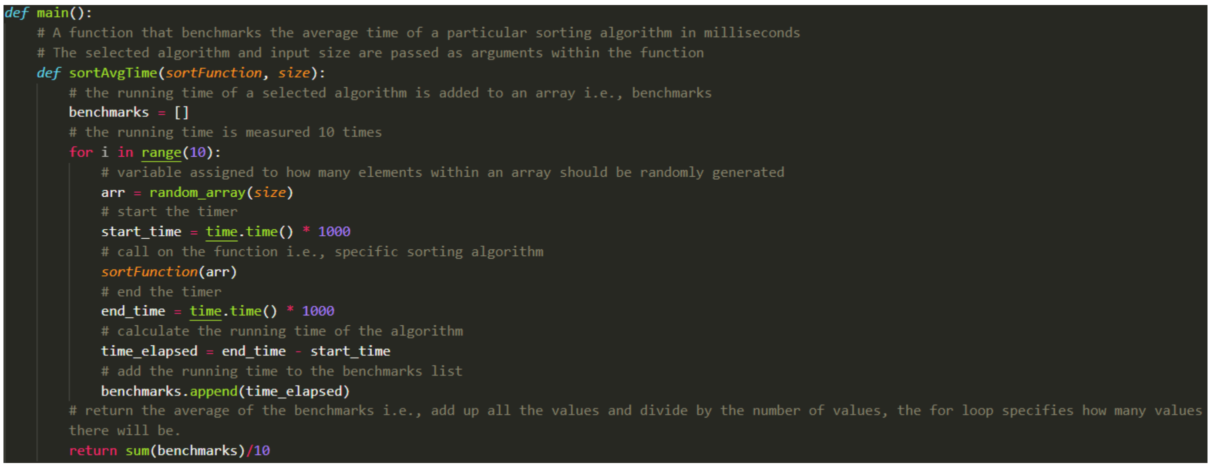
```
> def bubbleSort(arr): ···
  # End of Bubble Sort algorithm

  # A function that implements the sorting algorithm of Merge Sort
  # Code referenced from the following link: https://www.geeksforgeeks.org/merge-sort/

> def mergeSort(arr): ···
  # End of Merge Sort algorithm


  # A function that implements the sorting algorithm of Counting Sort
  # Code referenced from the following link: https://www.geeksforgeeks.org/counting-sort/

> def countSort(arr): ···
  # End of Counting Sort algorithm

  # A function that implements the sorting algorithm of Selection Sort
  # Code referenced from the following link: https://www.geeksforgeeks.org/selection-sort/

> def selectionSort(arr): ···
  # End of Selection Sort algorithm

  # A function that implements the sorting algorithm of Heap Sort
  # Code referenced from the following link: https: // www.geeksforgeeks.org/heap-sort/

  # To heapify subtree rooted at index i.
  # n is size of heap

> def heapify(arr, n, i): ···

  # The main function to sort an array of given size

> def heapSort(arr): ···
  # End of Heap Sort algorithm
```

A variable was created to establish the varying input data of size n



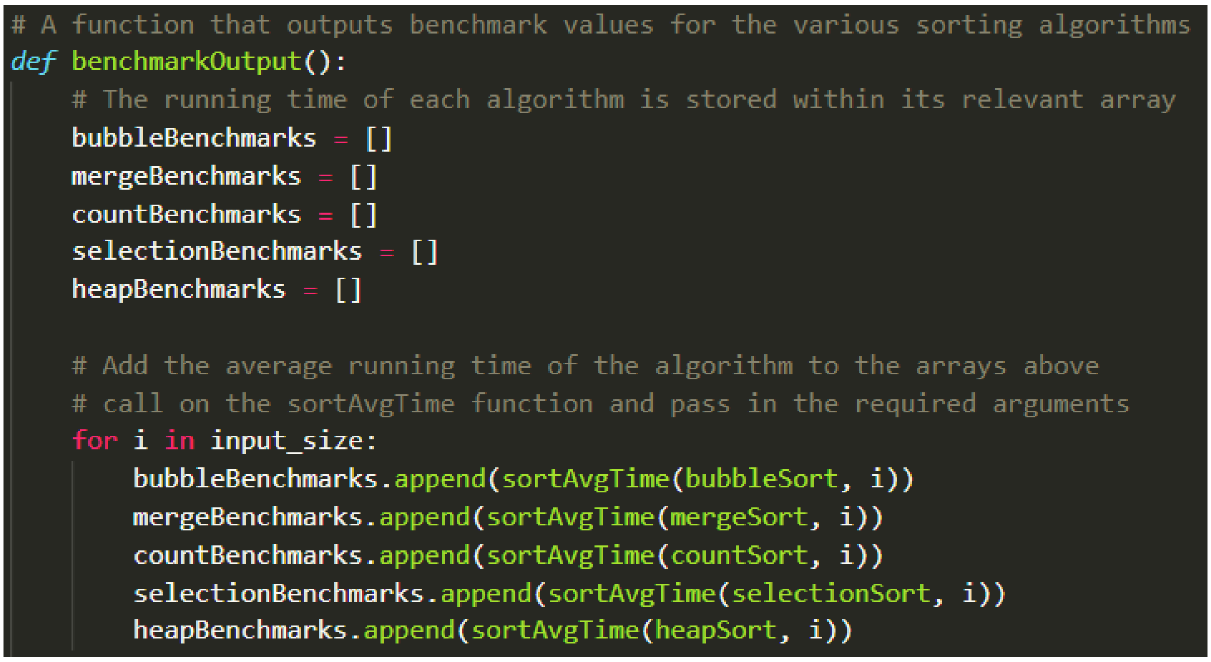
```
# Variable that is used for assinging the size of the random arrays i.e., how many elements are within the array
input_size = 100, 250, 500, 750, 1000, 1250, 2500, 3750, 5000, 6250, 7500, 8750, 10000
# 1000, 1250, 2500, 3750, 5000, 6250, 7500, 8750, 10000
```

A main method was created in order to allow the benchmarking of the various sorting algorithms to occur.



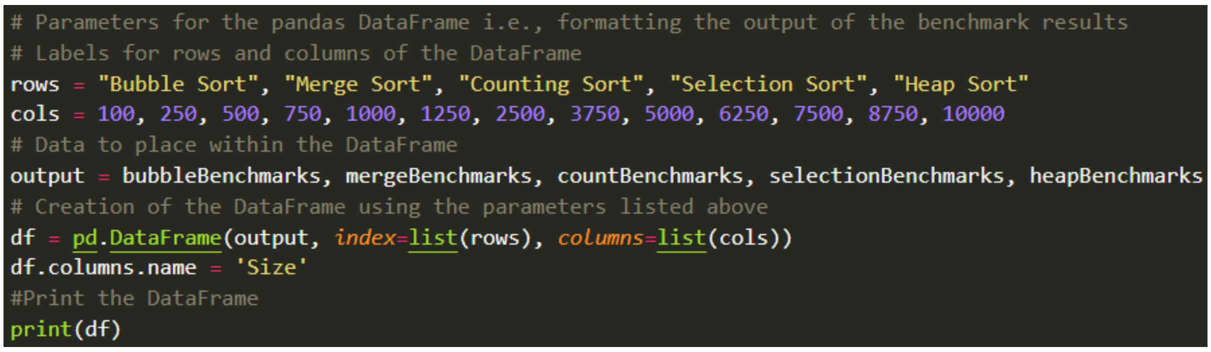
```
def main():
    # A function that benchmarks the average time of a particular sorting algorithm in milliseconds
    # The selected algorithm and input size are passed as arguments within the function
    def sortAvgTime(sortFunction, size):
        # the running time of a selected algorithm is added to an array i.e., benchmarks
        benchmarks = []
        # the running time is measured 10 times
        for i in range(10):
            # variable assigned to how many elements within an array should be randomly generated
            arr = random_array(size)
            # start the timer
            start_time = time.time() * 1000
            # call on the function i.e., specific sorting algorithm
            sortFunction(arr)
            # end the timer
            end_time = time.time() * 1000
            # calculate the running time of the algorithm
            time_elapsed = end_time - start_time
            # add the running time to the benchmarks list
            benchmarks.append(time_elapsed)
        # return the average of the benchmarks i.e., add up all the values and divide by the number of values, the for loop specifies how many values
        there will be.
        return sum(benchmarks)/10
```

The function sortAvgTime is then called, this function calls on a particular sorting algorithm and then iterates through the variable created previously i.e., input_size. The output of this function is then stored within an array relative to the sorting algorithm defined within the function argument e.g., the benchmark timings of the bubblesort algorithm are stored within the array bubbleBenchmarks.



```
# A function that outputs benchmark values for the various sorting algorithms
def benchmarkOutput():
    # The running time of each algorithm is stored within its relevant array
    bubbleBenchmarks = []
    mergeBenchmarks = []
    countBenchmarks = []
    selectionBenchmarks = []
    heapBenchmarks = []

    # Add the average running time of the algorithm to the arrays above
    # call on the sortAvgTime function and pass in the required arguments
    for i in input_size:
        bubbleBenchmarks.append(sortAvgTime(bubbleSort, i))
        mergeBenchmarks.append(sortAvgTime(mergeSort, i))
        countBenchmarks.append(sortAvgTime(countSort, i))
        selectionBenchmarks.append(sortAvgTime(selectionSort, i))
        heapBenchmarks.append(sortAvgTime(heapSort, i))
```

In order to format the output of the benchmarks in a more comprehensible and appropriately formatted manner a pandas dataframe was created to allow this to occur.

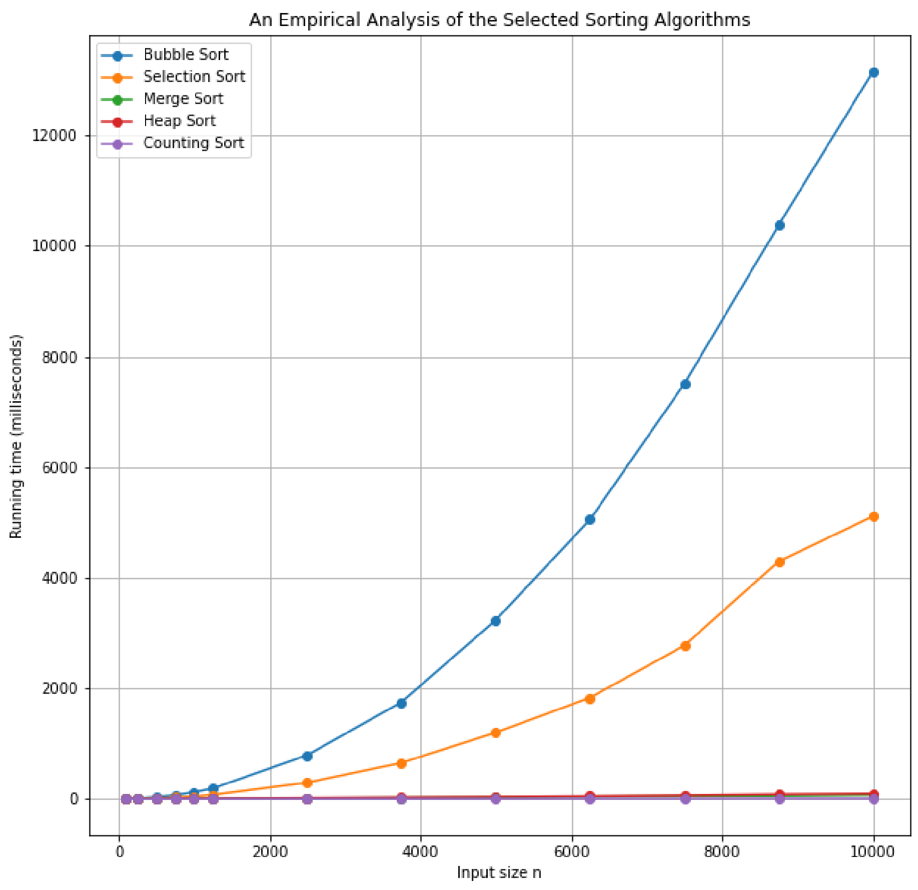


```python
# Parameters for the pandas DataFrame i.e., formatting the output of the benchmark results
# Labels for rows and columns of the DataFrame
rows = "Bubble Sort", "Merge Sort", "Counting Sort", "Selection Sort", "Heap Sort"
cols = 100, 250, 500, 750, 1000, 1250, 2500, 3750, 5000, 6250, 7500, 8750, 10000
# Data to place within the DataFrame
output = bubbleBenchmarks, mergeBenchmarks, countBenchmarks, selectionBenchmarks, heapBenchmarks
# Creation of the DataFrame using the parameters listed above
df = pd.DataFrame(output, index=list(rows), columns=list(cols))
df.columns.name = 'Size'
#Print the DataFrame
print(df)
```

## Results table depicting the system runtime of the various sorting algorithms

| Input Size | 100 | 250 | 500 | 750 | 1000 | 1250 | 2500 | 3750 | 5000 | 6250 | 7500 | 8750 | 10000 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bubble Sort | 1.2 | 7.101 | 34.703 | 69.105 | 121.309 | 189.214 | 789.458 | 1748.333 | 3231.896 | 5049.497 | 7503.218 | 10370.583 | 13146.455 |
| Selection Sort | 0.4 | 2.600 | 12.201 | 26.602 | 46.904 | 73.707 | 293.522 | 650.850 | 1198.191 | 1829.037 | 2770.859 | 4291.227 | 5107.924 |
| Merge Sort | 0.2 | 0.900 | 2.700 | 3.000 | 4.500 | 5.600 | 12.601 | 19.900 | 26.800 | 42.003 | 47.603 | 54.006 | 70.404 |
| Heap Sort | 0.2 | 1.100 | 2.500 | 4.700 | 6.100 | 8.301 | 17.601 | 27.401 | 37.003 | 48.904 | 63.206 | 82.506 | 94.007 |
| Counting Sort | 0.1 | 0.400 | 0.500 | 0.500 | 0.700 | 1.000 | 1.900 | 2.500 | 3.801 | 5.900 | 5.600 | 7.901 | 7.302 |

*Table 3 – All values are in milliseconds, and are the average of 10 repeated runs*



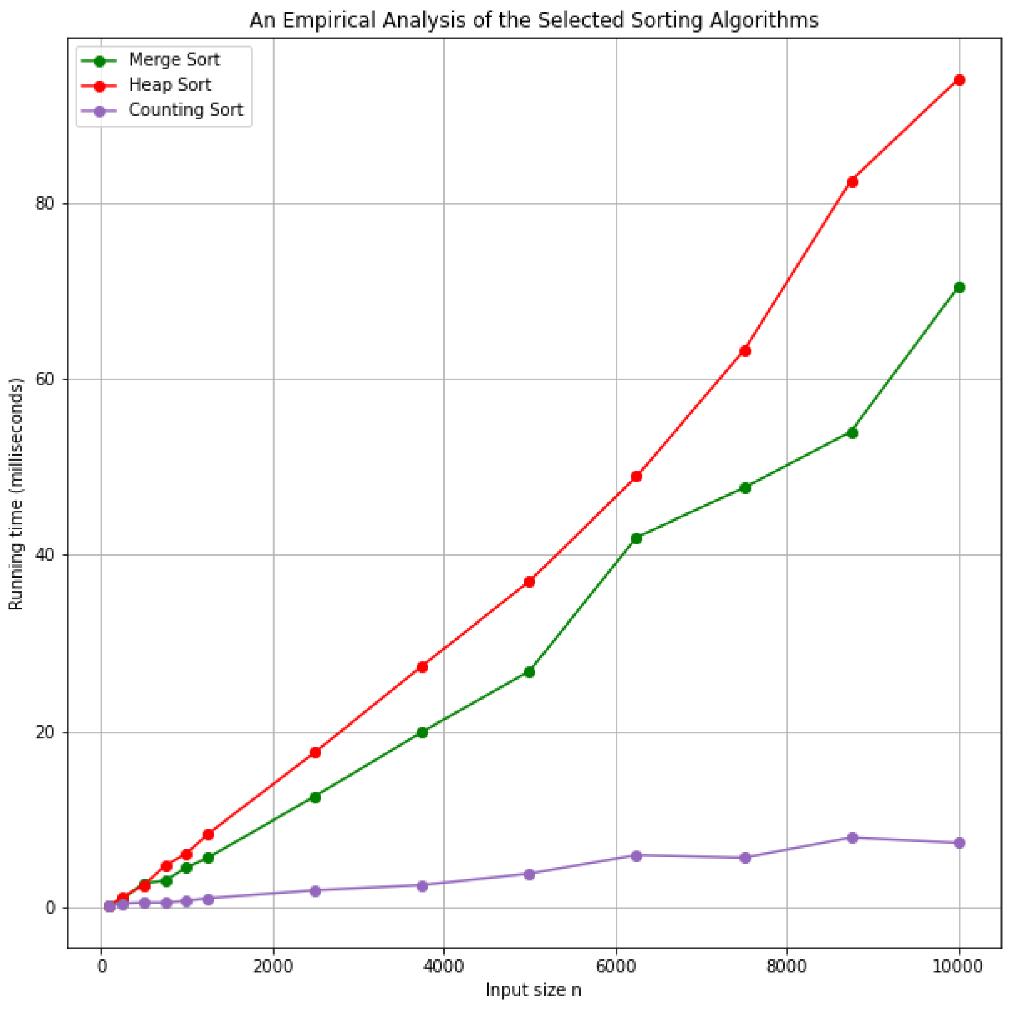
*Graph 1*

The line graph above, provides an overview of the relationship between the various sorting algorithms and their performance with reference to the variables of system runtime and the sorting of a series of inputs i.e., arrays of varying sizes, of which contain random numbers. As previously established the sorting algorithms of bubble sort and selection sort fall under the category of simple comparison-based sorts. Both algorithms have a worst and average case complexity of O($n^2$). While taking their expected performance into consideration, it is evident by the graphs and the runtime information within the results table, that both algorithms provided results that were aligned to their expectations. As conveyed by the *results table* and *graph 1,* the simple comparison-based sorts of bubble sort and selection sort displayed rather poor performances, in comparison to that of the other sorting algorithms.

The disparity that occurred between the simple comparison-based sorts and the other sorting algorithms is overwhelmingly apparent when viewing *graph 1*. The other sorting algorithms are depicted as having identical runtimes, and what may be suggested to be that of an instant processing time. However, we know that this is not the case, due to the fact that O(n) is the minimum sorting time possible, as every item within a data structure must be examined at least once. Based on the information obtained within *graph 1*, a conclusion can made of which informs us of the efficiency levels of the simple comparison-based sorting algorithms i.e., bubble sort and selection sort. When dealing with large inputs of data, both bubble sort and selection sort display poor performance, although selection sort did sort the arrays of randomly generated numbers at a speed of more than double that of the runtime of bubble sort.

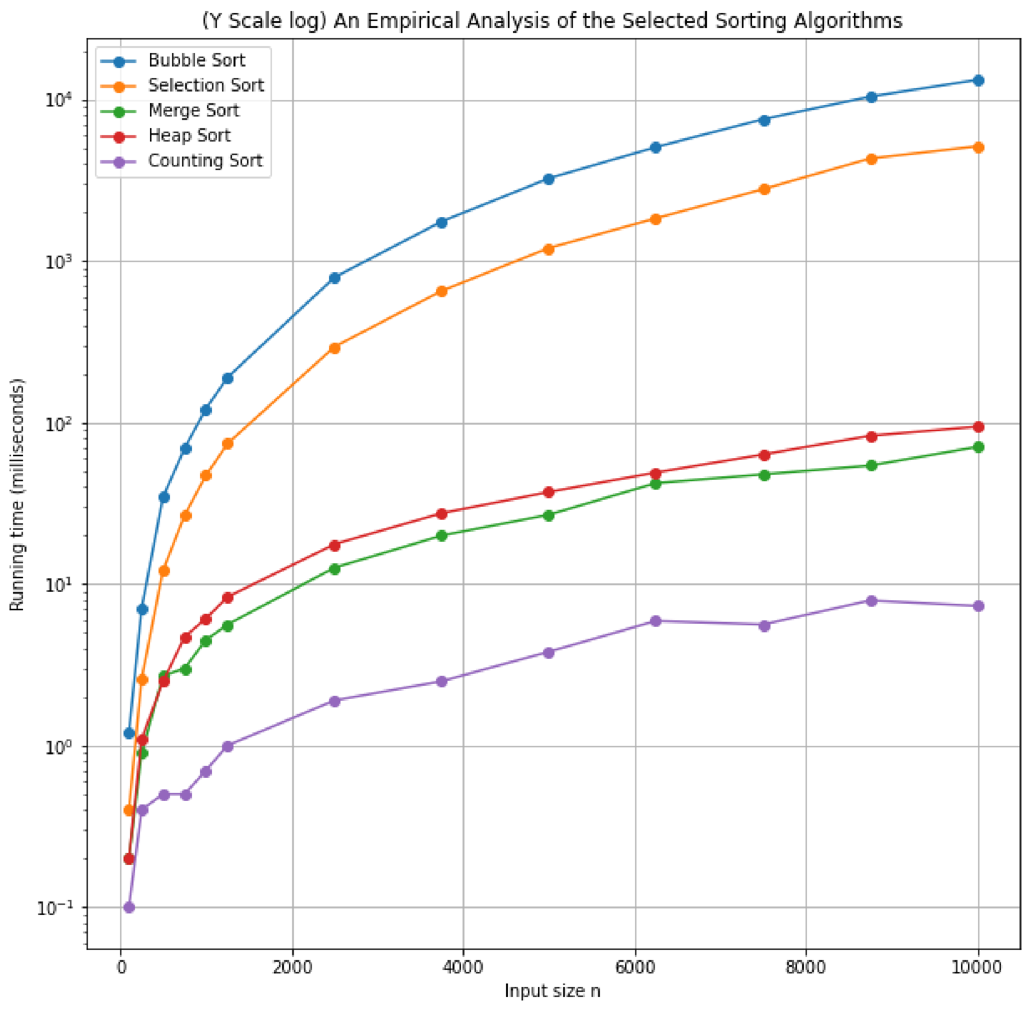
In order to establish a more accurate depiction of the relationship between that of efficient comparison-based and non-comparison-based sorting algorithms. Another graph was created i.e., *graph 2*, in order to provide a greater insight into the limitations that were previously discussed with reference to *graph 1*. *Graph 2,* informs us that of the efficient comparison-based sorting algorithms of which have a complexity of O(nlog n), merge sort has a better performance in comparison to heap sort. When comparing the performance of the simple comparison-based sorting algorithms to that of the efficient comparison-based sorting algorithms. The efficient sorting algorithms firmly represent their title, as evidently depicted by the results table and the graphs.

Lastly, we are met the with the category of non-comparison sorting algorithms, namely that of counting sort. Counting sort is regarded as the most efficient sorting algorithm that we have dealt with yet; this algorithm displays a worst and average case complexity of n + k. When viewing the results table and the graphs, counting sort displays the best performance in comparison to all of the other sorting algorithms that were discussed throughout the course of this report.


An Empirical Analysis of the Selected Sorting Algorithms
Merge Sort
Heap Sort
Counting Sort
Running time (milliseconds)
0
20
40
60
80
0
2000
4000
6000
8000
10000
Input size n

*Graph 2*

In order to highlight the varying performance of the sorting algorithms, *graph 3* was created. *Graph 3* plots the y axis on a log scale, this allows the benchmarking process of the sorting algorithms to be depicted in a form that promotes accurate conclusions to be made, regarding the performance of the various sorting algorithms.



*Graph 3*

## References

Azar, E. and Eguiluz Alebicto, M. (2016) *Swift Data Structure and Algorithms*. Birmingham, UK: Packt Publishing. Available at: GMIT library, http://search.ebscohost.com/login.aspx?direct=true&AuthType=ip,sso&db=nlebk&AN=1424569&site=ehost-live&scope=site (Accessed: 12 July 2021).

Claus Matzinger (2019) *Hands-On Data Structures and Algorithms with Rust: Learn Programming Techniques to Build Effective, Maintainable, and Readable Code in Rust 2018*. Birmingham, UK: Packt Publishing. Available at: GMIT library, http://search.ebscohost.com/login.aspx?direct=true&AuthType=ip,sso&db=nlebk&AN=2012512&site=eds-live&scope=site (Accessed: 12 July 2021).

Rao, A. S. K. (2016) *R Data Structures and Algorithms*. Birmingham: Packt Publishing. Available at: GMIT library, http://search.ebscohost.com/login.aspx?direct=true&AuthType=ip,sso&db=nlebk&AN=1426016&site=ehost-live&scope=site (Accessed: 12 July 2021).

Wisnu Anggoro (2018) *C++ Data Structures and Algorithms: Learn How to Write Efficient Code to Build Scalable and Robust Applications in C++*. Birmingham: Packt Publishing. Available at: GMIT library, http://search.ebscohost.com/login.aspx?direct=true&AuthType=ip,sso&db=nlebk&AN=1801030&site=ehost-live&scope=site (Accessed: 12 July 2021).