CTA Project Report on Benchmarking Sorting Algorithms

Student Name: Sarah McNelis Student Number: G00398343

Introduction

This report explains the concept of sorting and sorting algorithms, concepts of time and space complexity, performance, in-place sorting and comparator functions alongside the implementation and benchmarking process of five sorting algorithms.

Sorting is a means of strategically arranging items in an ordered sequence (wikipedia, 2022). In computer science, sorting is a fundamental building block for other algorithms (Valdarrama, 2020). One of the reasons for this is that it reduces the complexity of a problem (freecodecamp, 2020). When you consider a basic everyday task, we as humans actually use sorting strategies to tackle these. For example, putting the groceries away. We must first organize items out of the shopping bags, figure out where they will be stored, are they perishable, do they need to be frozen, maybe some items are household cleaning products which need to be stored in a different cupboard or a different room. This is considered to be a sorting process. Therefore, if we consider the complexity of a computer and computer programs, it is no surprise that sorting algorithms would be an essential part of their foundation. Computers have to sort through vast amounts of information to return key values to it's users. This is why algorithms have been developed for the sole purpose of sorting. However, the question arises, which sorting algorithm is the best?

One factor to deliberate when analysing the efficiently of an algorithm is a computer's performance. Computers are built differently. They vary with regard to central processing units, motherboards, storage units, input and output units, graphics processing units and random-access memory. Although computer contain most of the same components, it does not mean they will all perform at the same level. A computer's performance is measured by the amount of useful work completed (geekforgeeks, 2018). There are three components analysed when measuring this; response time, throughput, and execution time (geekforgeeks, 2018). In other words, measuring from the start time to end time of a task, the amount of work completed in this time and the total time a CPU spends on competing this task.

Furthermore, given the number of algorithms out there, each one behaves differently in terms of time and space complexity. Time complexity refers to the amount of time it takes an algorithm to run (interviewkickstart, 2022). In other words, how efficient it may be. Space complexity refers to the amount of memory an algorithm uses to run (interviewkickstart, 2022). Complexity is a vital part in comparing algorithms. This is why computer scientist measure efficiency with Big O notation (Valdarrama, 2020). Big O notation refers to the speed

of the algorithm's runtime relative to the size of the input (Valdarrama, 2020). In other words, comparing algorithms based of the same input size would return a true reflection of their growth and efficiently in terms of time and complexity.

This table in Fig 1, was extracted from <https://realpython.com/sorting-algorithms-python/#the-significance-of-time-complexity> and gives a brief explanation of Big O notation in terms of runtime complexity. As we can see from Fig 1, there are five different types of Big O notation depending on the growth or failure of an algorithm.

Big O	Complexity	Description
O(1)	constant	The runtime is constant regardless of the size of the input. Finding an element in a hash table is an example of an operation that can be performed in constant time**.**
O(n)	linear	The runtime grows linearly with the size of the input. A function that checks a condition on every item of a list is an example of an *O(n)* algorithm.
O(n^2)	quadratic	The runtime is a quadratic function of the size of the input. A naive implementation of finding duplicate values in a list, in which each item has to be checked twice, is an example of a quadratic algorithm.
O(2^n)	exponential	The runtime grows exponentially with the size of the input. These algorithms are considered extremely inefficient. An example of an exponential algorithm is the three-coloring problem.
O(log n)	logarithmic	The runtime grows linearly while the size of the input grows exponentially. For example, if it takes one second to process one thousand elements, then it will take two seconds to process ten thousand, three seconds to process one hundred thousand, and so on. Binary search is an example of a logarithmic runtime algorithm.

Fig 1. Five examples of the runtime complexity of different algorithms (Valdarrama, 2020).

Therefore, if we refer to O(1) notation, it is considered the fastest in terms of complexity. This is because the runtime stays the same regardless of the size of an array that needs to be sorted. Whereas, O(n) means that the complexity increases with the size which leads to a longer runtime. Furthermore, O(n^2) is a step further than O(n) because it checks elements in an array twice. This means it will produce a longer runtime. O(2^n) notation is described as being inefficient. This type of exponential growth starts with a low runtime but then doubles with the increasing size of the input. Finally, O(log n) is similar to O(n) in that is has a linear growth. However, the difference between the two is that O(log n) continues at this type of growth regardless of the input size which means it has an exceptionally low and quick runtime in terms of complexity.

We must also consider other factors such as in-place sorting. An in-place sorting algorithm does not use additional space to produce a sorted output. Instead, it transforms the input in-place using the same memory (geeksforgeeks, 2022). In other words, it doesn't use any

additional arrays to temporarily store the sorted data. It changes the sequence of the data within the original array.

Likewise, we must consider the stability of algorithms. This refers to when a sorting algorithm contains two elements with equal keys which appear in the same order in the sorted array as they did in the unsorted array (geeksforgeeks, 2019). Stability varies on the input. If key elements in an array are unique, then the algorithm would not be stable. Some sorting algorithms are stable by nature for example; bubble sort, insertion sort, merge sort and count sort, whereas, quick sort and heap sort are not stable by nature (geeksforgeeks, 2019). However, it is possible to give a sorting algorithm stability by changing the comparison element so that two elements are compared for elements with equal values (geeksforgeeks, 2019).

Moreover, there are different types of sorting algorithms. For the purpose of this report I will focus on comparison-based sorting algorithms and non-comparison-based sorting algorithms. A comparison-based sort compares element of an array with each other in order to return a sorted array (geeksforgeeks, 2021). Examples of comparison-based algorithms are; Bubble sort, Selection sort, Insertion sort, Merge sort, Quick sort, and Heap sort. Element are compared with the help of comparator functions. A non-comparison-based sort uses internal values to sort elements in an array (geeksforgeeks, 2021). Examples of these are; Counting sort, Bucket sort and Radix sort. In other words, elements are sorted without comparing them to other elements.

Sorting Algorithms:

I have chosen the following five sorting algorithms which I will describe, implement and benchmark using python.

1. Bubble Sort
2. Insertion Sort
3. Merge Sort
4. Quick Sort
5. Counting Sort

Bubble Sort:

A bubble sort is a simple comparison-based sorting algorithm. It operates by comparing and swapping adjacent elements in an array if they are not in the correct order (geeksforgeeks, 2022). This is done by iteration. In other words, the process is repeated until the entire array has been sorted. It was given it's name because with every iteration, the elements in the list bubble up towards their correct position (Valdarrama, 2020).

I created the diagram below, Fig 2, to explain how bubble sort works. A bubble sort can either start at the beginning or end of an array. In this diagram I have started at the beginning

comparing the first two elements. As we can see the first element; 8 is greater than the second element; 6 so they swap places. Next 8 is compared to the next element to it's right. The same comparison is made again, and the process is repeated until 8 has reached its rightful position within the array. Then, the process is repeated with the new array comparing 6 with its neighbour and so on.

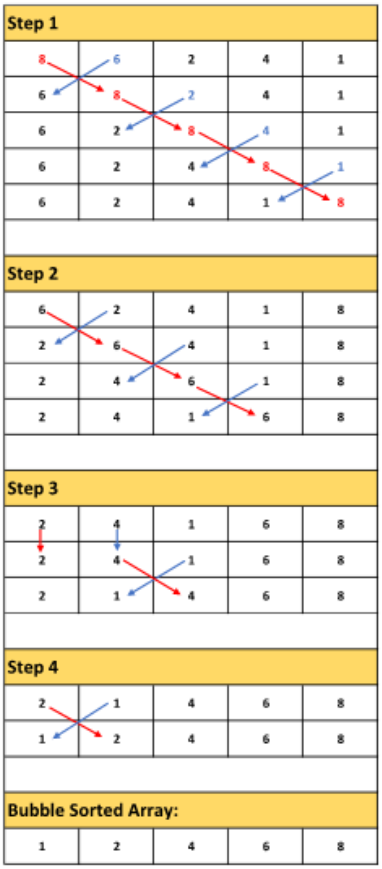


Fig 2. Custom Bubble Sort Diagram.

This bubble sort algorithm uses comparator functions throughout. And in Step 3 in Fig 2, we can see the use of stability. Please see the python code I developed in Fig 3 for my bubble sorting algorithm. This code was sourced and adapted from < https://www.geeksforgeeks.org/python-program-for-bubble-sort/>.

```
# A simple comparison-based sort - Bubble Sort
# Code sourced and adapted from <https://www.geeksforgeeks.org/python-program-for-bubble-sort/>

def bubble_sort(array):
    # Looping through length of array from last to first index.
    for i in range(len(array)-1, 0, -1):
        for s in range(i):
            # if s is is greater than the next element/s+1 - swap them.
            if array[s] > array[s + 1]:
                array[s], array[s + 1] = array[s + 1], array[s]
```

Fig 3. Image of python code found in bubble_sort.py.

In Fig 3, I have started by creating a function specifically for a bubble sort algorithm. I loop thought the length of the array from the last to the first index. In my python code I am iterating the opposite direction to Fig 2 in order to display that a bubble sort can be implemented starting at either end of an array. Next I check if the element is greater than the element to it's right. If yes than I swap the order. This process is repeated for every element in the array until the loop is complete.

Although a bubble sort is considered simple and straightforward to understand and implement, its runtime complexity tends to be quite slow (Valdarrama, 2020). Moreover, Valdarrama (2020) explains that this algorithm contains two nested for loops in which it performs n-1 comparisons and then n-2 comparisons and so on until the sort is complete. In terms if Big O complexity, constants should be removed as they have no impact on the input size. Furthermore, Valdarrama (2020) states that a bubble sort performs with a best-case of *O(n)* which is described in Fig 1 as having linear growth with a function that checks a condition on every element in an array. Valdarrama (2020) also describes a bubble sort as having an average and worst-case complexity of O(n^2) which if we refer back to Fig 1, is considered to have a runtime of quadratic complexity which checks each item twice.

Insertion Sort:

An insertion sort is another simple comparison-based sorting algorithm. It virtually splits an array into a sorted and unsorted part (geekforgeeks, 2021). The element in an array is compared to the previous element. If it is greater than the first, then no swap is needed. Otherwise the value is repeatedly moved left until it meets a value less than itself (bbc, 2022).

I designed another diagram to explain how an Insertion sort operates – see Fig 4. The algorithm starts with the second element in the array; 6 and compares it to the first element; 8. If the second is less than the first then a swap is performed. This continues for the third element; 2 which is less than the second and first elements and moves to the first position in the array. This process is repeated until the entire array is sorted. We can also see from this diagram comparator operators are used to sort the elements in the array.

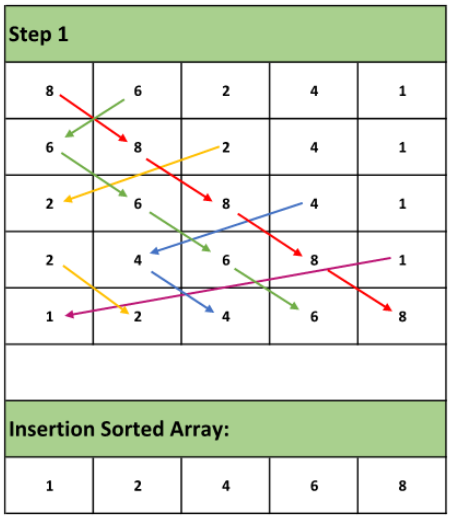


Fig 4. Custom Insertion Sort Diagram.

In Fig 5 below, you will find an image of my code which was sourced and adapted from <https://www.geeksforgeeks.org/insertion-sort/>. In this function a for loop is used to check every element in the array from the second element. If the element to it's left is greater than the key than it is moved one position to the right and a new key is given. This is repeated until every element in the loop has been sorted.

```
# A second simple comparison-based sorting algorithm - Insertion Sort
# Code sourced and adapted from <https://www.geeksforgeeks.org/insertion-sort/>

def insertion_sort(array):
    # For every element fron 1 to length of the array.
    # 1 and not 0 as all element will move one position to the right each time.
    for i in range(1, len(array)):

        key = array[i]

        # Move elements greater than the key one position to right
        p = i-1
        while p >= 0 and key < array[p] :
                array[p + 1] = array[p]
                p -= 1
        array[p + 1] = key
```

Fig 5. Image of python code found in insertion_sort.py.

Insertion sort is as equally straightforward and simple as Bubble sort. It also contains nested loops; however, the inner loop of an Insertion sort is more efficient as it passes through the array until it finds it's correct position (Valdarrama, 2020). However, Valdarrama (2020) states that insertion sort has an O(n^2) runtime complexity on an average case and worst case and O(n) in best case which is the same a Bubble sort even though the insertion inner loop is more efficient. Nevertheless, both Bubble and Insertion sorts are better suited to smaller arrays.

Merge Sort

Merge sort is an example of an efficient comparison-based sorting algorithm. A merge sort divides an array in half and recursively called itself on the subarrays and then merges the subarrays until only one array remains (geekforgeeks, 2022). This has been described as a divide and conquer method (geekforgeeks, 2022). As we can see in Fig 6 below, the array is halved, and the same is done with the subarrays. Then the arrays are sorted and merged together. We can see from this diagram that comparator operators are used for sorting along with in-place sorting.

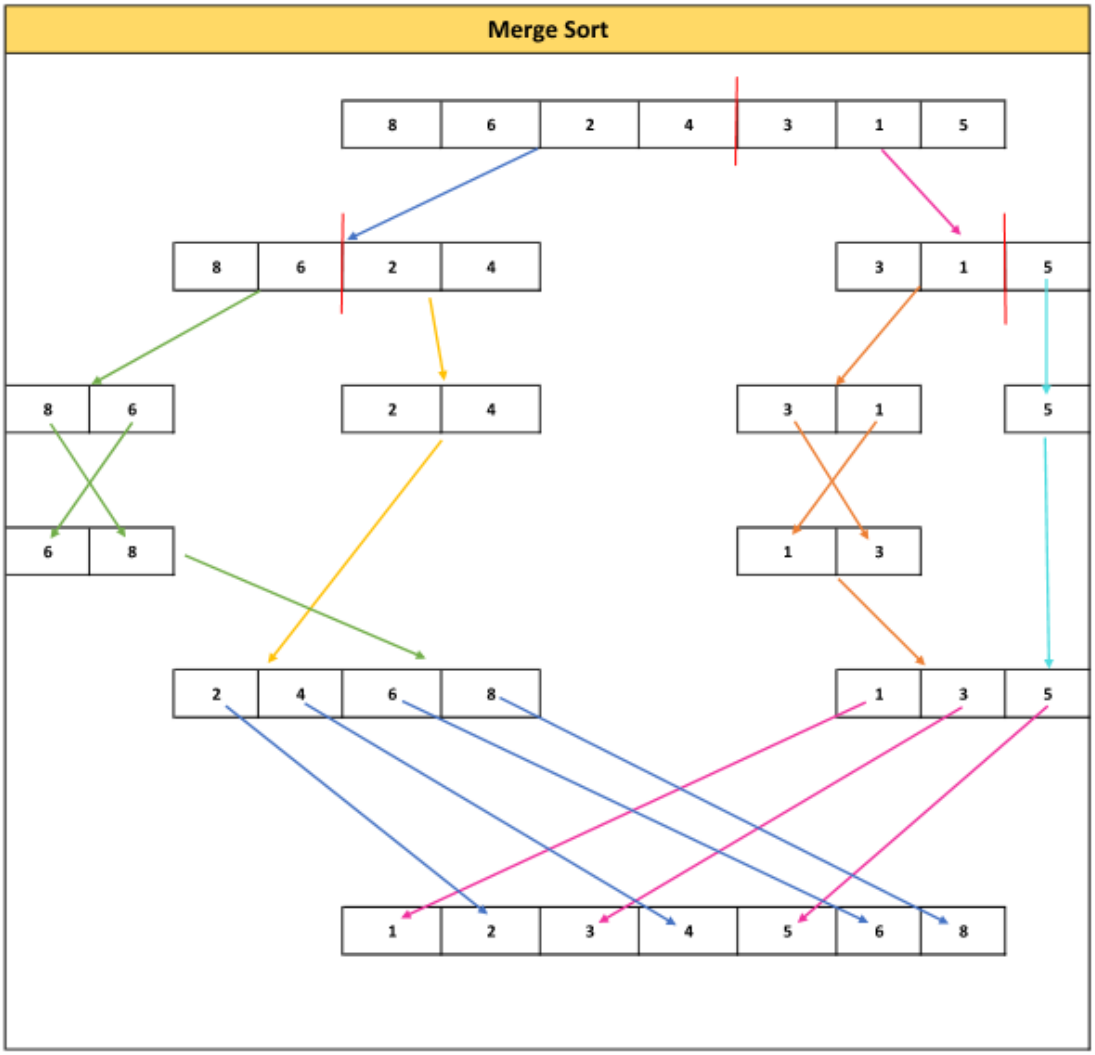
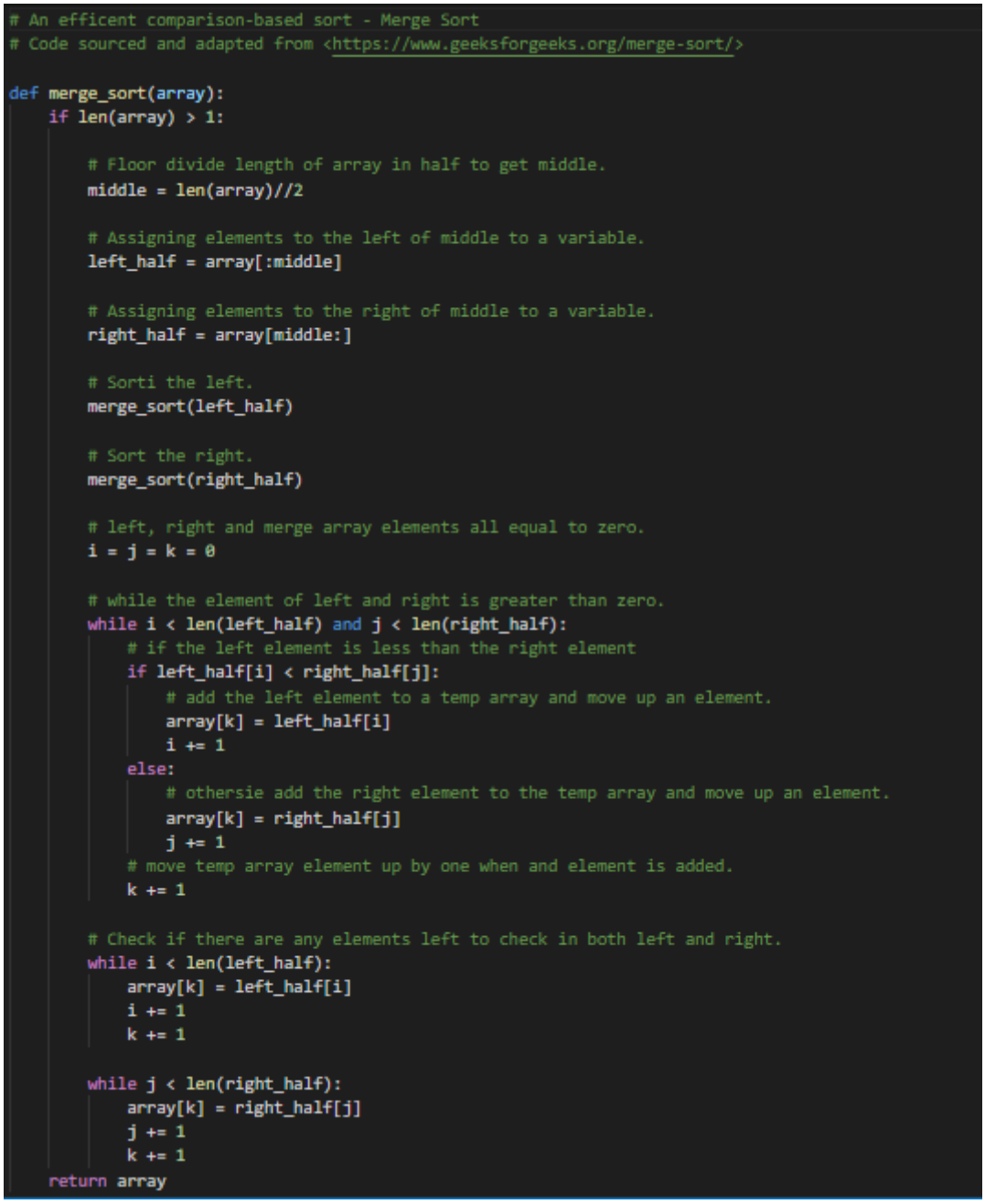


Fig 6. Custom Merge Sort Diagram.

See Fig 7 below, an image of my code which was sourced and adapted from <https://www.geeksforgeeks.org/merge-sort/>. We can see in this function that the length of the array is divided in two and each half is assigned a variable name. Then, the function is recursively called to spilt the two halves in half again if needed. Next the elements are compared to one another and swapped if needed. Then the sub-sorted arrays are merged together and returns as one sorted array.



```python
# An efficent comparison-based sort - Merge Sort
# Code sourced and adapted from <https://www.geeksforgeeks.org/merge-sort/>

def merge_sort(array):
    if len(array) > 1:

        # Floor divide length of array in half to get middle.
        middle = len(array)//2

        # Assigning elements to the left of middle to a variable.
        left_half = array[:middle]

        # Assigning elements to the right of middle to a variable.
        right_half = array[middle:]

        # Sorti the left.
        merge_sort(left_half)

        # Sort the right.
        merge_sort(right_half)

        # left, right and merge array elements all equal to zero.
        i = j = k = 0

        # while the element of left and right is greater than zero.
        while i < len(left_half) and j < len(right_half):
            # if the left element is less than the right element
            if left_half[i] < right_half[j]:
                # add the left element to a temp array and move up an element.
                array[k] = left_half[i]
                i += 1
            else:
                # othersie add the right element to the temp array and move up an element.
                array[k] = right_half[j]
                j += 1
            # move temp array element up by one when and element is added.
            k += 1

        # Check if there are any elements left to check in both left and right.
        while i < len(left_half):
            array[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            array[k] = right_half[j]
            j += 1
            k += 1
    return array
```

Fig 7. Image of python code found in merge_sort.py.

Valdarrama (2020) explains that merge sort receives two arrays and combines these arrays by checking each element only once. This leads to an O(n) runtime complexity for best case. However, as it recursively called itself for each half, the runtime complexity is O($n \log_2 n$) for best, average and worst case. This is because a merge sort always divides itself into two halves and uses linear time to merge the halves together (geekforgeeks, 2022). Therefore, a Merge sort is a very efficient comparison-based sorting algorithm and would be exceptionally good for larger arrays. However, as it creates copies of the array inside the function it would use more memory than Bubble or Insertion sort (Valdarrama, 2020). This is something to consider if memory was limited.

Quick Sort

Quicksort is another example of an efficient comparison-based sorting algorithm. A quick sort divides an array into subarrays selecting a pivot element from that array (programiz, 2022). This pivot element should be positioned where elements less than the pivot are situation on it's left and elements greater are position on it's right (programiz, 2022). The same approach is repeated for the two subarrays until all elements are sorted and combined.

In Fig 8 below, the Quicksort uses a similar divide and conquer method to Merge sort. The array is split in two using a pivot element as a partition. The first subarray contains smaller element and the second with larger ones. Next the pivot and partition method is recursively called until the array is sorted and returned.

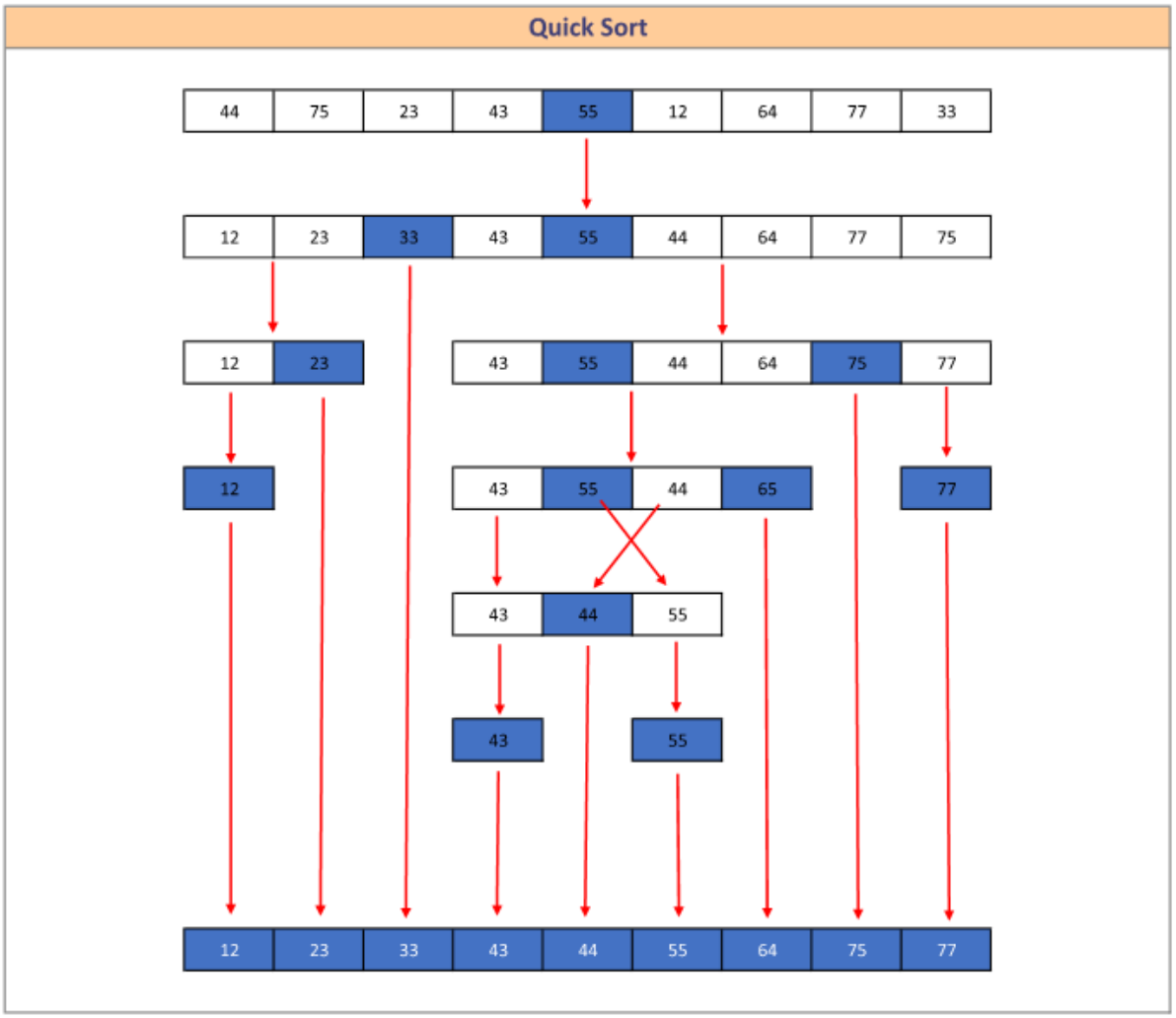


Fig 8. Custom Quick Sort Diagram.

This algorithm in Fig 9 was sourced and adapted from <https://runestone.academy/ns/books/published/pythonds/SortSearch/TheQuickSort.html> and <https://www.geeksforgeeks.org/python-program-for-quicksort/#:~:text=The%20key%20process%20in%20quickSort,be%20done%20in%20linear%

20time>. The process is the same as my diagram in Fig 8. The quick_sort function calls a helper function. In the helper function, the array , first and last elements are passed in. Next, the partition function is called, and the helper function is recursively called on both halves. The partition function finds the pivot value and creates the partition. And then sorts the two halves swapping places if needed. Finally the sorted array is returned.

```python
# A second efficent comparison-based sort - Quick Sort
# Code sourced and adapted from <https://runestone.academy/ns/books/published/pythonds/SortSearch/TheQuickSort.html>
# and <https://www.geeksforgeeks.org/python-program-for-quicksort/#:~:text=The%20key%20process%20in%20quickSort,be%20done%20in%20linear%20time.>

# Main function.
def quick_sort(array):
    # See funciton below which is called here to help complete quick sort.
    quick_sort_helper(array, 0, len(array)-1)
    return array

# Helper function.
def quick_sort_helper(array, first, last):
    # If first element is greater than the last find the halfway point
    if first < last:
        halfway = partition(array, first, last)
        # Now recursively call function again to sort both halves.
        quick_sort_helper(array, first, halfway-1)
        quick_sort_helper(array, halfway+1 ,last)

# a function to partition the array using the pivot element,
def partition(array, first, last):
    # Pivot value is the first value in the array
    pivot_value = array[first]
    # The left mark is the first element of the remaining unsorted array hence first plus one.
    left_mark = first+1
    # The right mark is the last element of the unsorted array.
    right_mark = last

    done = False
    while not done:
        # While the left mark is less than the right and less than the pivot
        # move the left mark one to the right.
        while left_mark <= right_mark and array[left_mark] <= pivot_value:
            left_mark = left_mark + 1
        # While the right mark is greater than the pivot and the left mark
        # move the right mark one to the left.
        while array[right_mark] >= pivot_value and right_mark >= left_mark:
            right_mark = right_mark -1
        # If the right and left marks crossover then the function stops.
        if right_mark < left_mark:
            done = True
        else:
            # Otherwise swap the order.
            array[left_mark], array[right_mark]  = array[right_mark], array[left_mark]
    # swap with the pivot value with the right mark.
    array[first], array[right_mark] = array[right_mark], array[first]

    return right_mark
```

Fig 9. Image of python code found in quick_sort.py.

Valdarrama (2020) explains that the pivot value can affect the runtime complexity. This is why it is best to assign the pivot to the hallway point in an array. Doing so will allow for O(n) runtime complexity for best case. And an average and worst case of $O(n \log_2 n)$ runtime complexity (Valdarrama, 2020). Similar to Merge sort, Quicksort is very efficient and fast. However, it uses more memory than Bubble or Insertion sort. Thus, this could be a limitation with regard to larger input sizes (Valdarrama, 2020).

Counting Sort

Counting sort is a non-comparison-based sorting algorithm. A counting sort counts the number of occurrences of each element in an array (programiz, 2022). This count is stored in another array and the sorting is done by using the count as an index in this other array. In

other words, the distinct elements are used to calculate the position of each element in the array.

As we can see below in Fig 10, the elements of the input array on the left are counted and stored in appropriate index of an empty array. Then it calculates the cumulative sum in order to produce the correct sequence on the right.

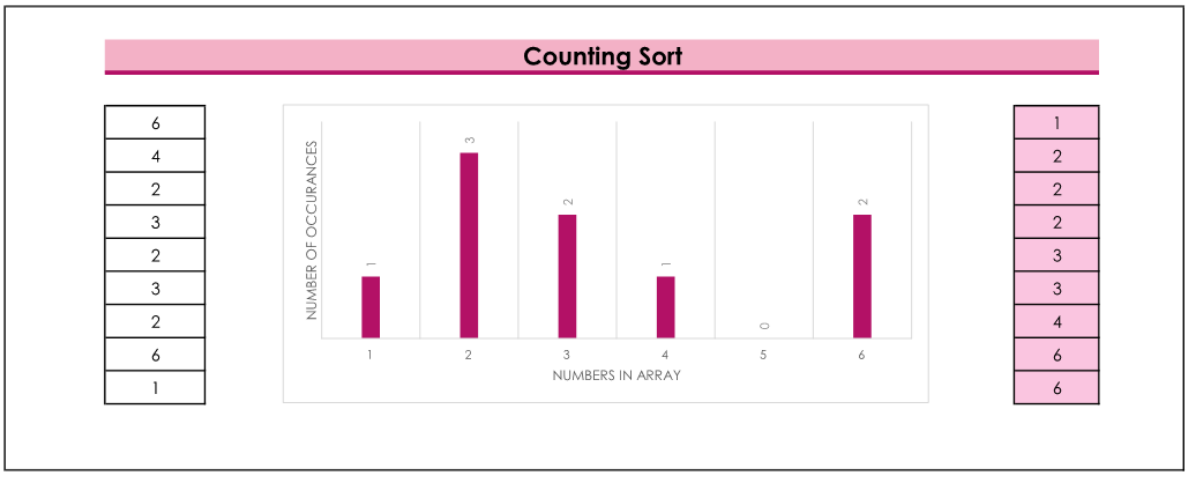


Fig 10. Custom Counting Sort Diagram.

My code in Fig 11 was sourced and adapted from <https://www.w3resource.com/python-exercises/data-structures-and-algorithms/python-search-and-sorting-exercise-10.php>, <https://www.geeksforgeeks.org/counting-sort/> and <https://www.programiz.com/dsa/counting-sort >.

```python
# A non-comparison sort- Counting Sort
# Code sourced and adapted from <https://www.w3resource.com/python-exercises/data-structures-and-algorithms/python-search-and-sorting-exercise-10.php>
# and from <https://www.geeksforgeeks.org/counting-sort/>
# and <https://www.programiz.com/dsa/counting-sort>

def counting_sort(array):
    # Count array to store each element - initalized as zero to the max value of the orgional array.
    count_array = [0 for i in range(max(array)+1)]
    # Empty var for sorted elements.
    sorted_arr = []

    # each time an element appears in the array - add 1 to count_array
    for e in array:
        count_array[e] += 1

    # For every element while count is greater than zero.
    for e in range(0, (len(count_array))):
        while count_array[e] > 0:
            # Add the element to sorted array
            sorted_arr.append(e)
            # And decrease the count by 1 each time.
            count_array[e] -= 1
    return sorted_arr
```

Fig 11. Image of python code found in counting_sort.py.

As we can see from this figure above, the count array is created as an empty array of zeros for them max value of the array plus one. Then an empty array is created to return the sorted array into. Each time a unique key element appears in the array one is added to the count

array. Next, the elements are added to the empty array in a sorted manner and returned to the user.

Geeksforgeeks (2022) suggests that counting sort has *O(n + k)* runtime complexity where n is the input size and k is the range of elements in that array. In other words, a counting sort algorithm is efficient if the given range of the input data is not significantly greater than the number of elements to be sorted (geeksforgeeks, 2022).

Implementation & Benchmarking

This section focuses on the process of implementing each of the five sorting algorithms using random arrays of various input sizes in order to measure the average runtime complexity for each algorithm.

The code used for the benchmarking process was sourced and adapted from the following; <https://learnonline.gmit.ie/pluginfile.php/579189/mod_resource/content/0/CTA_Project.pdf> , <https://learnonline.gmit.ie/course/view.php?id=5341>, and <https://realpython.com/sorting-algorithms-python/>.

The first step in developing the python program for benchmarking the sorting algorithms is to import various python packages and modules – see Fig 12 below. The randint module is imported from the random package in order to generate random arrays for the sorting process. The time module is imported because part of the benchmarking process is to track the start and end time of each run for each algorithm. Next, Pandas is used when working with dataframes. In this program the data created in the benchmarking process is stored in a dataframe which can then be used to compare, display, and plot the data for each sorting algorithm. Matplotlib Pyplot is a module which is used for data visualisation and plotting. Allowing the data created and gathered to be displayed on a graph can help us visualise the performance of each sorting algorithm. Finally, I am importing the five sorting algorithm python files I created. Importing them allows me to call upon these functions during the benchmarking process instead of having to re-write them within this program file.

```
# Import python packages.
# Randint to help generate random numbers.
from random import randint

# Using time to record time complexity.
import time

# Pandas for working with dataframes.
import pandas as pd

# This is used for data visualisation.
import matplotlib.pyplot as plt

# Import individual py function files for my sorting algorithms.
import bubble_sort
import merge_sort
import counting_sort
import insertion_sort
import quick_sort
```

Fig 12. Partial image of python code found in project.py.

Next I created a function for the benchmark process which can be seen in Fig 13 below. This function takes in three parameters; the algorithms, the input size, and the number of runs. I have then created empty variables to append and store the data created in the benchmarking process. I then use a for loop with two inner loops nested inside. This means that for each run of each size for each algorithm complete the following action. Next I call a separate function which can seen in Fig 14 below, which generates random arrays of integers between 0 and 100 with sizes of n. These random arrays are then returns to the benchmarking function to continue the process. Next each algorithm in the list of algorithms is selected and a timestamp is created for the start time. This is done using the time.time method which returns the current time in seconds (geeksforgeeks, 2019). Then the algorithm is called with the randomly generated array to sort. Following this, the end time is recorded using the python time.time function again. The reason for recording both the start and end times is to calculate the difference between them. This allowed me to determine the time elapsed. It is important to note here that I have multiplied the difference by one thousand so that I get my results in milliseconds. The next step is to append the results to the empty variables created in the beginning of the function. The time elapsed is recorded in time_taken, the number of runs is recorded in trial_runs, the input size is recorded in sizes_used and the algorithm is stored in the sort_type. The reason for storing this data in empty variables is so I can now use these variables of data to create the dataframe using pandas. I have done this by using the pd.Dataframe method while passing the data created into this method (geeksforgeeks, 2019).

```python
# Main function for benchmarking the sorting algorithims.
def benchmarking(algorithms, n, runs):
    # Variables to append results to.
    time_taken = []
    sizes_used =[]
    trial_runs =[]
    sort_type =[]
    # Loop for each algorithm.
    for sort_algo in algorithms:
        # Loop to run each size in array to each algorithm.
        for size in n:
            # Loop again to do it 10 times.
            for run in range(runs):
                # call function below to generate random numebrs to use.
                nums = get_random_array(size)
                # Select each algo to monitor time.
                algorithm = algorithms[sort_algo]
                # Use time method to record start time.
                start_time = time.time()
                # Call the algo after start time recorded.
                algorithm(nums)
                # Record the end time.
                end_time = time.time()
                # Subtract to get the time elapsed and by 1000 to return it in milliseconds.
                difference = (end_time - start_time)* 1000
                # Save results to var above.
                time_taken.append(difference)
                # Record the amount of runs in the var above.
                trial_runs.append(run+1)
                # Add the size used for each to var above.
                sizes_used.append(size)
                # And record the algo name for each.
                sort_type.append(sort_algo)
    # Using pandas to create a dataframe of results from the four variables above.
    df = pd.DataFrame({"Sort":sort_type, "Size":sizes_used, "Times":time_taken, "trialNo":trial_runs})
    return df
```

Fig 13. Partial image of python code found in project.py.

```python
# Seperate function to generate random numbers.
def get_random_array(n):
    array = []
    for i in range(0, n, 1):
        # Append random numbers to an array to use for benchmarking.
        array.append(randint(0, 100))
    return array
```

Fig 14. Partial image of python code found in project.py.

The next section of code in this file is a function to calculate the average running time of each sorting algorithm for each input size over the 10 runs – see Fig 15. This is done by passing in the dataframe which has been created and setting the index to the size column using the pandas set_index function in order to drop the default index from the dataframe (geeksforgeeks, 2020). Next I am using the pandas iloc method to select the column up until the second column (stackoverflow, 2016) and then I am grouping them by algorithm and input size. The pandas groupby function is a simple way to group data by category (geekforgeeks, 2021). In other words, I am grouping each time by each algorithm and by each input size n. Next I am using the mean function (geeksforgeeks, 2021), and round function to get the

average runtimes rounded to three decimal places (geekforgeeks, 2021). Finally, the unstack method will return the dataframe using new level for the column labels (w3resource, 2022).

```
# A function to calculate averages of benchmarking process.
def mean_for_each(df):
    # Set index as size column.
    df.set_index('Size', inplace=True)
    # Using mean and round to get averages and groupby the sort and size.
    avg_speeds = (df.iloc[:, 0:2].groupby(['Sort','Size']).mean()).round(3)
    # Using unstack method to change format of dataframe.
    return avg_speeds.unstack()
```

Fig 15. Partial image of python code found in project.py.

Following this I have created another function to plot the data – see Fig 16. This is done by passing the amended dataframe df2 into the function. Firstly, I am setting the figure size to 10x5. Next I am using a T plot. This is a type of plot used in pandas in order to re-arrange the data so that I can plot multiple series together on one plot (swcarpentry, 2022). Within this T.plot function I am setting the line width, marker, marker-size along with an alpha parameter (pandas, 2022). The alpha parameter will allow the lines to be slightly transparent in order to differentiate between them in the case of crossovers. Next, I set a title and labels for both axis (adamsmith, 2022). And finally, I am saving the figure as a png file.

```
# A function to plot the data.
def plot_the_benchmarks(df2):
    # Set the figure size.
    plt.rcParams["figure.figsize"]=(10,5)
    #T.plot to plot multiple series on on plot.
    df2.T.plot(lw=1 , marker='.', markersize=5, alpha=0.5)
    plt.title('Benchmarking Sorting Algorithms')
    plt.ylabel("Running time in Milliseconds")
    plt.xlabel("Array Size")
    #plt.show()
    # Save plot as png file.
    plt.savefig('plot.png')
```

Fig 16. Partial image of python code found in project.py.

Next I have the main function – see Fig 17. This is the function which calls each of the individual functions above to implement the benchmarking process. First, I have assigned the algorithms to be used. Next, I have chosen the input sizes for n. After that, I have called the benchmarking function which we saw earlier in Fig 13. I have passed in the algorithms, input sizes and have chosen to perform 10 runs on each. The dataframe that is returned from the benchmarking function is now assigned to the variable named benchmark. This is then passed

into the mean_for_each function which we saw in Fig 15. As already discussed this will return the average runtimes for each algorithm and size. This is now stored in a new variable named df2. This axis is then removed from the dataframe using the rename_axis method (w3resource, 2022). Furthermore, the column levels are dropped again using the droplevel function (w3resource, 2022). Next, the dataframe of the results is saved to a csv file for later use and the dataframe is printed to the console. Finally, I have called the plot_the_benchmarks function which we saw in Fig 16 which will plot the results.

```
# Main function called when program is run.
def main():
    # List of algos importing them from their seperate py files.
    algorithms = {"Bubble Sort": bubble_sort.bubble_sort,"Insertion Sort":insertion_sort.insertion_sort,"Merge Sort":merge_sort.merge_sort,
                  "Quick Sort":quick_sort.quick_sort, "Counting Sort": counting_sort.counting_sort}
    # Array sizes for each algo.
    n = [100, 250, 500, 750, 1000, 1250, 2500, 3750, 5000, 6250, 7500, 8750, 10000]
    # Call the main function using the algos, array sizes and 10 runs for each.
    benchmark = benchmarking(algorithms, n, 10)
    # Storing the average running times in a new df.
    df2 =  mean_for_each(benchmark)
    # Renamaing the axis to none to remove the sort heading for the algorithims.
    df2.rename_axis(None, inplace=True)
    # Dropping the sizes down a level so that Size heading is above the algos with the sizes horizontally across the table.
    df2.columns = df2.columns.droplevel()
    # Saving the df2 to a csv in order to create a table of console results.
    df2.to_csv('averages'+'.csv')
    # Print df to console.
    print(df2)
    # Call function to plot df2.
    plot_the_benchmarks(df2)
```

Fig 17. Partial image of python code found in project.py.

As we can see from Fig 18, the use of various python functions has allowed for the desired format of the data to be printed out in the console. We can clearly see the input sizes as column headers and the algorithms as the index and within each column and row is the average running time for each.

```
C:\Users\sarah\Desktop\CTA_Project>python project.py
Size             100    250     500     750      1000     1250     2500      3750      5000      6250      7500      8750      10000
Bubble Sort     1.054  8.379  37.263  82.271  142.819  266.736  993.164  2177.667  3928.533  6211.315  8162.090  11555.446  15100.282
Counting Sort   0.100  0.200   0.100   0.200    0.199    0.602    0.501     2.334     4.496     2.023     5.658     3.754      5.472
Insertion Sort  0.966  4.349  19.975  45.864   81.395  125.676  508.536  1148.380  2085.514  3207.144  4589.734  6307.143   8195.569
Merge Sort      1.904  0.816   2.809   5.517    3.829    8.335   17.103    30.117    39.259    45.153    57.878    67.731     88.309
Quick Sort      0.300  0.114   1.223   4.416    3.224    3.666   14.300    25.927    38.476    56.173    70.485    90.818    124.016
```

Fig 18. Image of Dataframe produces in the console.

However, it is important to note that this python file will not run unless this piece of code in Fig 19 is read. In other words, once the python interpreter will read the name main it will begin running the program (geekforgeeks, 2020). However, it does not start to run until this point.

```
# Will only run when it reaches the main function.
if __name__ == "__main__":
    main()
```

Fig 19. Partial image of python code found in project.py.

The Results

As we can see from Fig 20, Bubble Sort and Insertion Sort scored extremely high average runtimes in comparison to Counting Sort, Merge Sort and Quick Sort. Bubble Sort has the worst running time taking up to 15100.280 milliseconds to run an array of ten thousand elements. Insertion Sort was the second worst scoring at an average of 8195.569 milliseconds runtime for an array of ten thousand elements. However, considering the comments from Valdarrama (2020), both Bubble and Insertion Sort have passable average running times up until an array of one hundred elements. This supports Valderrama's (2020) theory that these two comparison-based sorting algorithms are less efficient on larger arrays and more suited to sorting smaller ones.

On the other hand, the other comparison-based sorting algorithms; Merge and Quick Sort returned satisfactory average runtimes. Merge sort appears to have a steady average runtime through all the input sizes. However, Quick Sort appears to have extremely low runtimes up until array sizes of five thousand at which point it's runtimes are more than that of Merge Sort. This again supports Valderrama's (2020) suggestion that both Merge and Quick Sort are efficient and quick and are suited to larger arrays.

Nevertheless, the clear winner from this benchmarking process was the non-comparison based Counting Sort Algorithm. It's average runtimes are exceptional requiring only 5.472 milliseconds to sort an array of ten thousand elements. That is over two thousand, seven hundred times the milliseconds it took the Bubble sorting algorithm to sort the same array size. This supports Geeksforgeeks (2022) suggestion that Counting Sort is a far more efficient way of sorting arrays.

Average Running Times													
Size	**100**	**250**	**500**	**750**	**1000**	**1250**	**2500**	**3750**	**5000**	**6250**	**7500**	**8750**	**10000**
Bubble Sort	1.054	8.379	37.263	82.271	142.819	266.736	993.164	2177.667	3928.533	6211.315	8162.09	11555.45	15100.28
Counting Sort	0.1	0.2	0.1	0.2	0.199	0.602	0.501	2.334	4.496	2.023	5.658	3.754	5.472
Insertion Sort	0.966	4.349	19.975	45.864	81.395	125.676	508.536	1148.38	2085.514	3207.144	4589.734	6307.143	8195.569
Merge Sort	1.904	0.816	2.809	5.517	3.829	8.335	17.103	30.117	39.259	45.153	57.878	67.731	88.309
Quick Sort	0.3	0.114	1.223	4.416	3.224	3.666	14.3	25.927	38.476	56.173	70.485	90.818	124.016

Fig 20. Table of console results.

These results can be clearly visualised in the plot in Fig 21 below. Bubble and Insertion sort have a clear growth reaching upwards towards their respective 15100.28 and 8195.569 milliseconds for an array size of ten thousand elements. Whereas, Quick, Merge and Counting Sort can barely be distinguished from one another which ties in with the results from the table in Fig 20 above. In fact, we can just about see the Counting Sort line below the other two which again clearly illustrates the results from the benchmarking process.

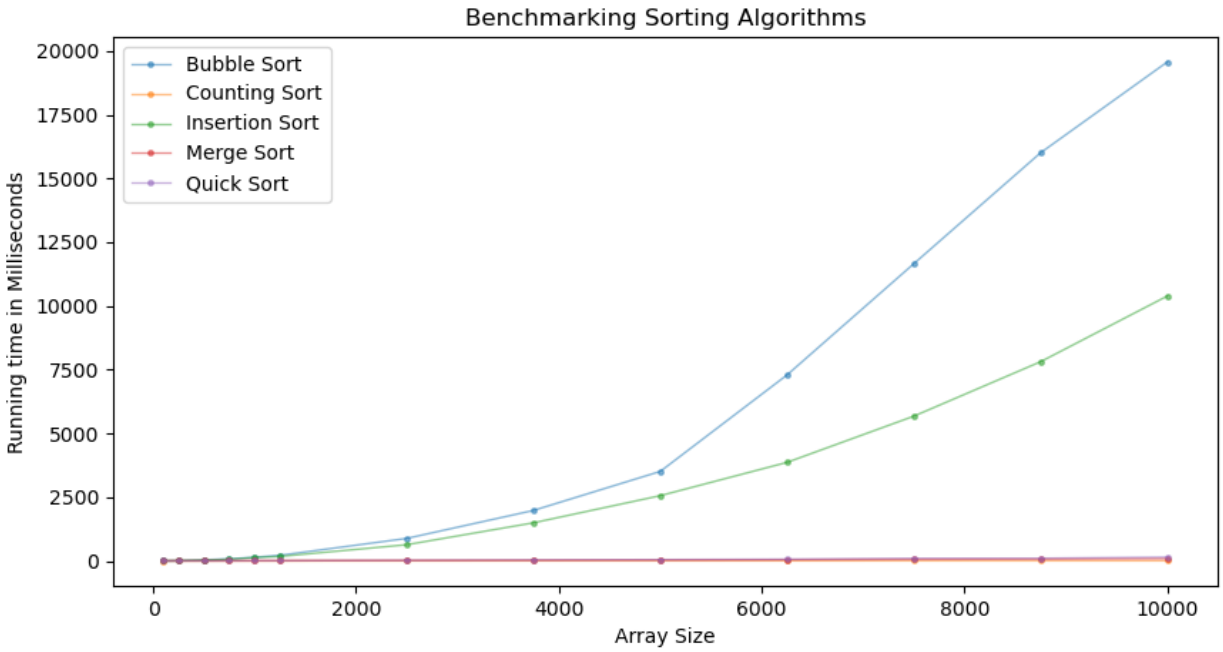


Fig 21. Plot of console data.

Conclusion:

This report started by explaining the concept of sorting and sorting algorithms, time and space complexity performance, in-place sorting, and comparator functions. I then described five sorting algorithms in detail, two simple comparison-based sorting algorithms; Bubble and Insertion Sort, two efficient comparison-based sorting algorithms; Merge and Quick Sort and finally a non-comparison-based sorting algorithm; Counting Sort. Next, I explained the implementation of benchmarking each of these five sorting algorithms. Ultimately, the conclusion in which I have arrived is that Bubble and Insertion Sorting algorithms are simple and easy to use, however, their runtime on larger array sizes leaves a lot to be desired. In contrast, Merge, Quick and Counting Sort algorithms appear to be the winners with regard to average runtime complexity. They produced quick runtimes especially with larger arrays. However, Counting Sort was the clear winner will the lowest average runtime of all five. In spite of this, it would be interesting to perform this analysis on another computer to compare how the results might differ. In any case, it is fair to say that sorting algorithms can reduces the complexity of problems and allows us efficiently to reach our desired results.

Bibliography

(2016, May 29). Retrieved from stackoverflow: https://stackoverflow.com/questions/37512079/python-pandas-why-does-df-iloc-1-values-for-my-training-data-select-till#:~:text=iloc%5B%3A%2C%20%3A2%5D%20or,a%20negative%20index%2C%20nothing%20changes.

(2018, Jul 27). Retrieved from geekforgeeks: https://www.geeksforgeeks.org/computer-organization-performance-of-computer/

(2019, Sep 26). Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/stability-in-sorting-algorithms/

(2019, Aug 28). Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/python-time-time-method/

(2019, Jan 10). Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/python-pandas-dataframe/

(2020, Jan 18). Retrieved from freecodecamp: https://www.freecodecamp.org/news/sorting-algorithms-explained/

(2020, Sep 01). Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/python-pandas-dataframe-set_index/

(2020, Dec 11). Retrieved from geekforgeeks: https://www.geeksforgeeks.org/what-does-the-if-__name__-__main__-do/

(2021, Jun 28). Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/analysis-of-different-sorting-techniques/

(2021, Jul 08). Retrieved from geekforgeeks: https://www.geeksforgeeks.org/insertion-sort/

(2021, Jul 08). Retrieved from geekforgeeks: https://www.geeksforgeeks.org/insertion-sort/

(2021, Jun 28). Retrieved from geekforgeeks: https://www.geeksforgeeks.org/python-pandas-dataframe-groupby/

(2021, Sep 27). Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/python-statistics-mean-function/

(2021, Sep 30). Retrieved from geekforgeeks: https://www.geeksforgeeks.org/round-function-python/

(2022, May 07). Retrieved from wikipedia: https://en.wikipedia.org/wiki/Sorting#:~:text=Sorting%20is%20any%20process%20of,grouping%20items%20with%20similar%20properties.

(2022, Apr 29). Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/bubble-sort/

(2022, May 07). Retrieved from interviewkickstart: https://www.interviewkickstart.com/learn/time-complexities-of-all-sorting-algorithms

(2022, May 07). Retrieved from bbc: https://www.bbc.co.uk/bitesize/guides/zjdkw6f/revision/6#:~:text=An%20insertion%20sort%20compares%20values,again%20with%20the%20next%20value.

(2022, Apr 22). Retrieved from geekforgeeks: https://www.geeksforgeeks.org/merge-sort/

(2022, May 07). Retrieved from programiz: https://www.programiz.com/dsa/quick-sort

(2022, May 07). Retrieved from programiz: https://www.programiz.com/dsa/counting-sort

(2022, Apr 25). Retrieved from wikipedia: https://en.wikipedia.org/wiki/Computer_performance#:~:text=In%20computing%2C%20computer%20performance%20is,of%20executing%20computer%20program%20instructions.

(2022, Feb 24). Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/in-place-algorithm/

(2022, Apr 22). Retrieved from geekforgeeks: https://www.geeksforgeeks.org/merge-sort/

(2022, April 22). Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/counting-sort/

(2022, Apr 18). Retrieved from w3resource: https://www.w3resource.com/pandas/dataframe/dataframe-unstack.php

(2022, Apr 18). Retrieved from w3resource: https://www.w3resource.com/pandas/dataframe/dataframe-rename_axis.php

(2022, Apr 18). Retrieved from w3resource: https://www.w3resource.com/pandas/series/series-droplevel.php

(2022, May 08). Retrieved from swcarpentry: https://swcarpentry.github.io/python-novice-gapminder/09-plotting/index.html

(2022, May 08). Retrieved from pandas: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.plot.html

(2022, May 08). Retrieved from adamsmith: https://www.adamsmith.haus/python/answers/how-to-add-axis-labels-to-a-plot-in-matplotlib-in-python

(2022, Jan 19). Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/python-program-for-bubble-sort/

(2022, Jan 20). Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/python-program-for-quicksort/#:~:text=The%20key%20process%20in%20quickSort,be%20done%20in%20linear%20time

(2022, Apr 22). Retrieved from geeksfrogeeks: https://www.geeksforgeeks.org/counting-sort/

(2022, Apr 04). Retrieved from w3resource: https://www.w3resource.com/python-exercises/data-structures-and-algorithms/python-search-and-sorting-exercise-10.php

(2022, Jan 20). Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/python-program-for-quicksort/#:~:text=The%20key%20process%20in%20quickSort,be%20done%20in%20linear%20time

(2022, May 08). Retrieved from runestone.academy: https://runestone.academy/ns/books/published/pythonds/SortSearch/TheQuickSort.html

Barney. (2020, Mar 22). Retrieved from towardsdatascience: https://towardsdatascience.com/how-to-export-pandas-dataframe-to-csv-2038e43d9c03

Carr, D. D. (2022, May 09). *Computational Thinking with Algorithms.* Retrieved from learnonline.gmit: https://learnonline.gmit.ie/course/view.php?id=5341

Carr, D. D. (2022, May 09). *CTA Project PDF.* Retrieved from learnonline.gmit: https://learnonline.gmit.ie/pluginfile.php/579189/mod_resource/content/0/CTA_Project.pdf

Valdarrama, S. (2020, Apr 15). *Sorting Algorithms in Python*. Retrieved from realpython: https://realpython.com/sorting-algorithms-python/