CTA Project

Benchmarking Sorting Algorithms

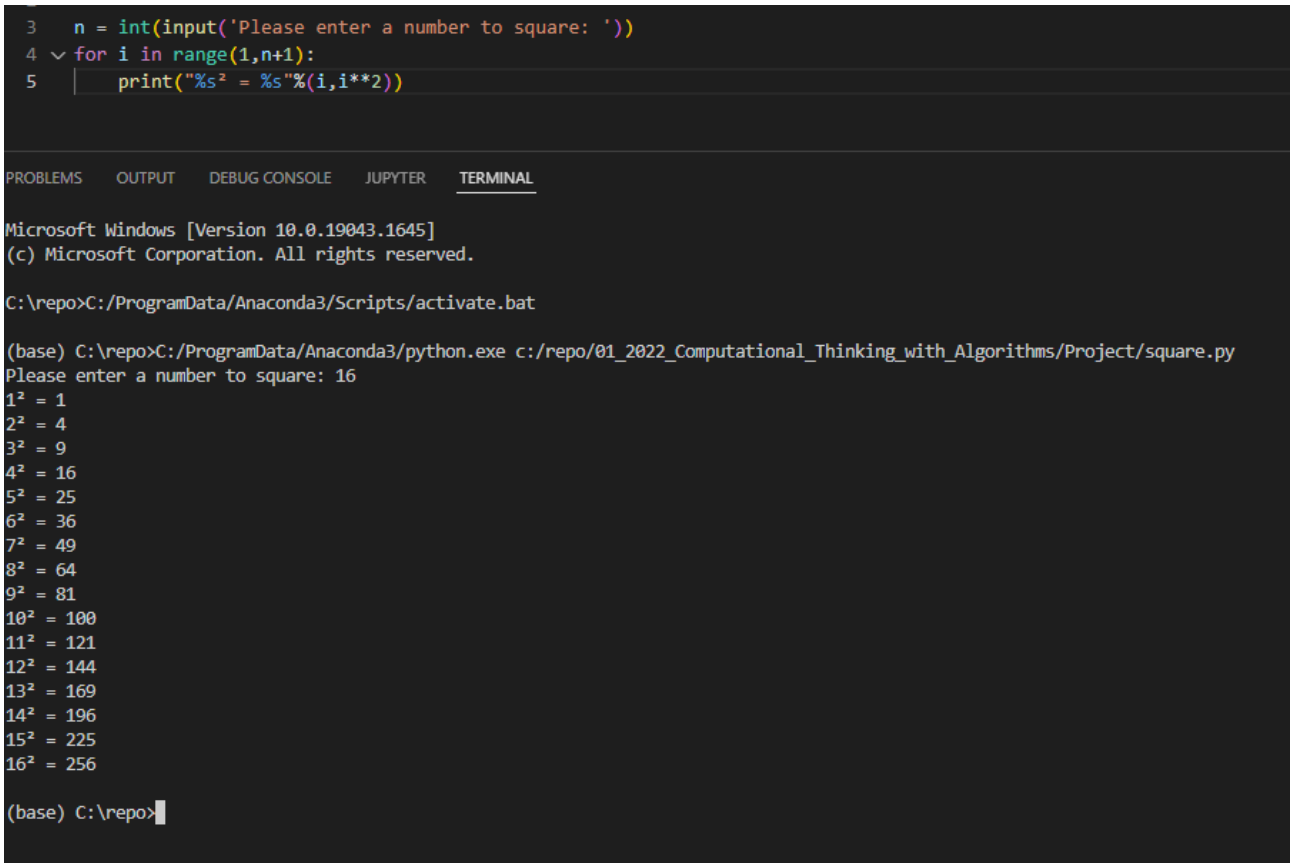
Introduction

This section of the report is to introduce different concepts of sorting and the use of sorting algorithms. This introduction will hopefully demonstrate an understanding of concepts such as performance, complexity, comparator functions, comparison/non-comparison based sorts, and in-place sorting.

This project outcome is to demonstrate a well-designed algorithm, as an algorithm is to get an expected output from input with a set of rules. The request is to investigate and demonstrate this using a sorting algorithm, producing the correct output by demonstrating the efficiency of the sorting algorithm over space/time and varying sizes of the input (n=100 to n=10000).

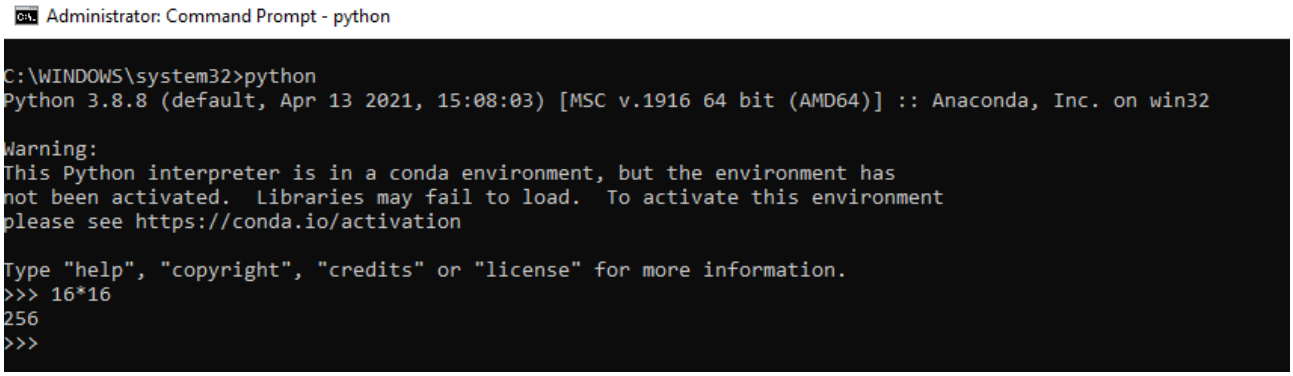
What is algorithms' time complexity?
The time complexity of an algorithm is estimated by counting the number of steps performed till its final execution. By the number of steps which approach would you consider is the best solution? A python for loop or using a mathematical operator of *

To demonstrate let us try two methods to output the square of 16. Example 1 uses a for loop to cycle from 1*1 eventually reaching 16*16 and printing out each step.



Example 1 For loop *[3]*

Example 2 uses a Mathematical operator of *



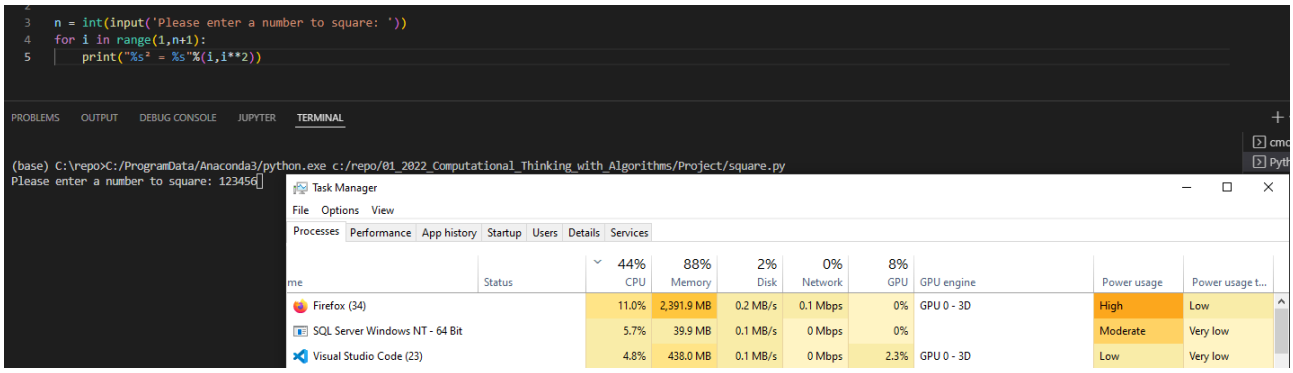
Example 2 Mathematical operator of *

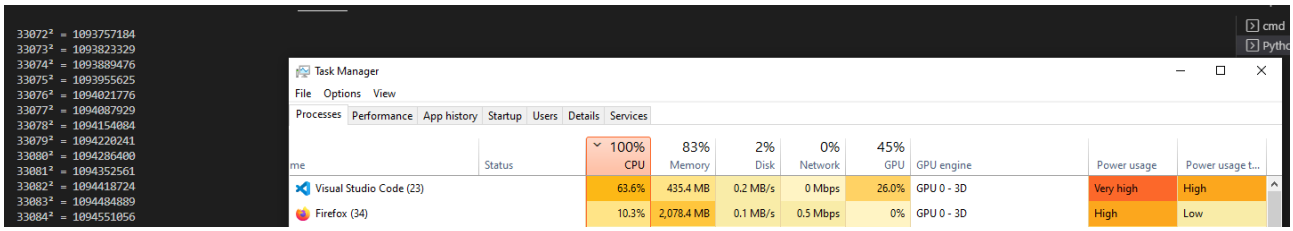
The time complexity of steps performed till their final execution using the mathematical operator of * returning the result in one line [4] is a better solution.

What is algorithms' space complexity?

Space complexity is considered the amount of memory a computer requires to solve concerning the input size, another aspect that can also affect the outcome can include the type of machine, programming language or how the program is compiled. [1]

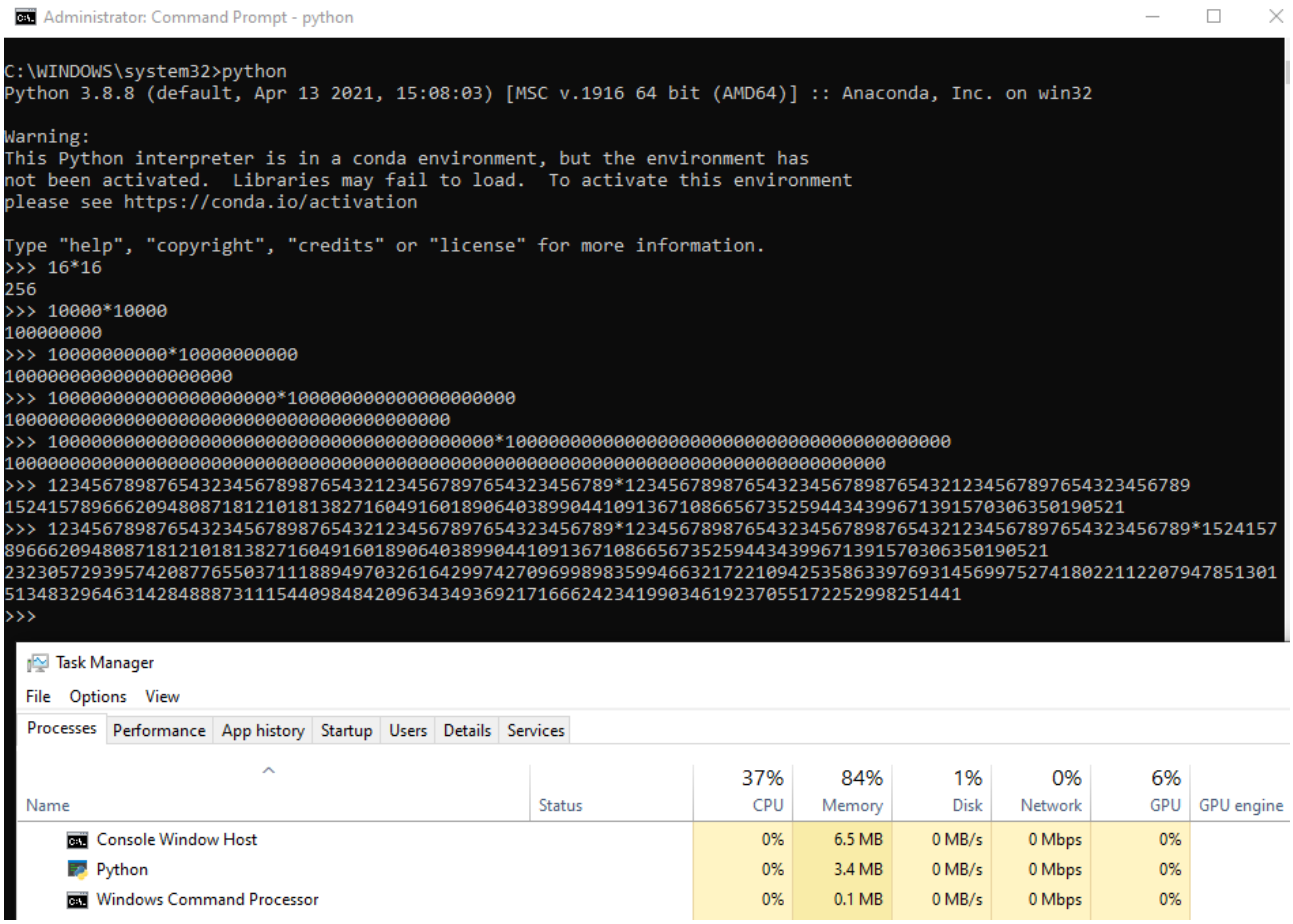
So if we re-run example 3 of the for loop with a bigger number (123456) this will take up more space in memory and resources, making the PC unusable until the program finishes executing.





Example 3 for loop Heavy on CPU and memory consumption.

Now let us get the square of 123456 using the mathematical operator of *. Example 4



Example 4 negligible CPU and memory consumption

The second squared example of a mathematical operator * requires little to no memory to achieve the same result and even with larger numbers did not reach the same requirements in example 3.

What methods measure complexity?

With an algorithm, there are two methods of calculating this Fig 1 gives a further breakdown.

- A Priori analysis looks at from a theoretical, device agnostic and measure of complexity.
- A Posteriori analysis measures the performance on the same platform

Posteriori analysis	A priori analysis
Evaluating efficiency empirically	Evaluating efficiency from a theoretical perspective
Algorithms which are to be compared are implemented and run on a target platform	Analysis removes the effect of implementation details (processor/system architecture)
relative efficiency of algorithms is analysed by comparing actual measurements collected during experimentation	Relative efficiency of algorithms is analysed by comparing their order of growth
Measure of performance	Measure of complexity

Fig 1 Difference between Posteriori and Priori [1]

These methods indicated performance versus complexity and if we refer to the Examples 1 and 3 python for loops, performance never affects complexity but as the complexity increased input size n it affected the performance. This is shown in Fig 2 when the increased input size n growth functions

value of n	constant	$log\ n$ (logarithmic)	n (linear)	$n\ log\ n$ (linearithmic)	n^2 (quadratic)	n^3 (cubic)	2^n (exponential)
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4096	65536
32	1	5	32	160	1024	32768	4294967296
64	1	6	64	384	4096	262144	1.84467E+19
128	1	7	128	896	16384	2097152	3.40282E+38
256	1	8	256	2048	65536	16777216	1.15792E+77
512	1	9	512	4608	262144	1.34E+08	1.3408E+154

Fig 2 Comparing growth functions *[5]*

What is the Big O notation? (Worst case)

From a theoretical perspective, the Big O notation is the most common metric used for calculating time complexity. Referring to examples Example 1 and 3 pythons for loops increasing the size of the input n, increased the number of the operation taken. Big O notation measures the growth rate of a function increase or decrease in a worst-case scenario representing the lower bounds

If Example 2 has a less complex Big 0 notation than Example 1 it can be inferred that Example 2 is more efficient in terms of space/time requirements. [5]

What is Ω (omega) notation? (Best Case)

This is the best-case scenario with a linear growth rate in execution time as n is increased, representing the upper bounds [5]

What is Θ (theta) notation? (Average Case)

Θ (theta) notation is the upper and lower bounds of the runtime and is used for analysing the metric used for calculating the average time of complexity.[5]

What is a sorting algorithm?

A sorting algorithm is to take an input (an array or list) to arrange the output in a particular order and has desirable properties of stability, efficiency and in-place sorting.[1] There are several sorting algorithms each with its efficiency in complexity by the number of operations or time it takes to complete known as Time and the amount of memory required to run also known as Space.

Why do we need sorting algorithms?

It is said that 25 per cent of the running time of computers in the 1960s was spent sorting with some tasks responsible for more than half of the computing time. [2]

While we humans deal with sorting information on a day to day for example your shopping list and finding food in a supermarket. We generally use technology to simplify tasks for us, this is usually through sorting, how we search the web and the results and the order of the return results, how to find a car's make, model, by year, mileage, etc.

Sorting memory usage using in-place

Different sorting techniques require different allocations of memory, to keep memory usage to a minimum the most desirable is in-place. In-place sorting does not require extra memory as the input data is usually overwritten by the output, only swapping elements within the input size n. This avoids creating an empty array for copying data to double the memory requirements.

In-place Definition: *In-place means that the algorithm does not use extra space for manipulating the input but may require a small though non-constant extra space for its operation. Usually, this space is O(log n), though sometimes anything in O(n) (Smaller than linear) is allowed. [6]*

Comparison-based and non-comparison-based sorts

Comparison based means that elements within input size n are compared with each other. This is called a Comparator function which compares one element against another element and returns a value. Comparing elements can never have a worst-case time better than O(N log N) Fig 2.

A Non-Comparison is a way to design an algorithm not to compare elements, for instance making an assumption about data, taking a range of values from the data and the distribution of the data. Non-comparison sorts can achieve linear n running time in the best case but are less flexible.

Algorithm	Best case	Worst case	Average case	Space Complexity
Bubble Sort	n	n^2	n^2	1
Selection Sort	n^2	n^2	n^2	1
Insertion Sort	n	n^2	n^2	1
Merge Sort	$n\ log\ n$	$n\ log\ n$	$n\ log\ n$	$O(n)$
Quicksort	$n\ log\ n$	n^2	$n\ log\ n$	n (worst case)
Heapsort	$n\ log\ n$	$n\ log\ n$	$n\ log\ n$	1
Counting Sort	$n + k$	$n + k$	$n + k$	$n + k$
Bucket Sort	$n + k$	n^2	$n + k$	$n \times k$
Timsort	n	$n\ log\ n$	$n\ log\ n$	n
Introsort	$n\ log\ n$	$n\ log\ n$	$n\ log\ n$	$log\ n$

Fig 3 Overview of sorting algorithms [7]

Sorting Algorithms

This section of the report is to introduce the five chosen sorting algorithms referencing their space and time complexity, how each algorithm works using bespoke diagrams and different example input instances. [8]

The five chosen sorting algorithms are as follows:

1. Bubble Sort is a simple comparison-based sort.
2. Merge Sort as an efficient comparison-based sort
3. Counting Sort as a non-comparison sort
4. Selection Sort is a simple comparison-based sort.
5. Insertion Sort is a simple comparison-based sort.

Bubble Sort is a simple comparison-based sort.

Bubble Sort is a comparison based sort which compares two elements, swapping them and continuing until they are in the correct order. The name comes from how the highest number bubbles up to the top as the largest element is repeatedly compared to the neighbour on the right and swapped. This is repeated until the largest element is in the furthest position to the right, then the next highest element and so on. See the Diagram 1 to show the example [5]

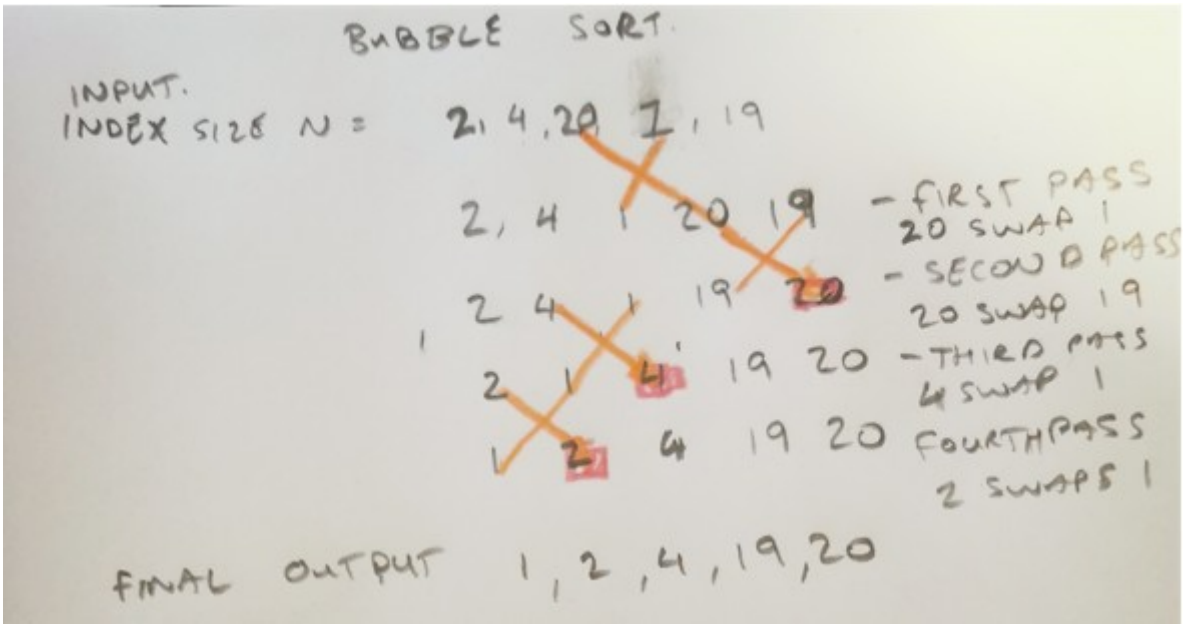


Diagram 1

It may better visualise if the input the bubbling up affects if the array is transposed, this diagram shows 20 bubbling up, then 4, then 2.

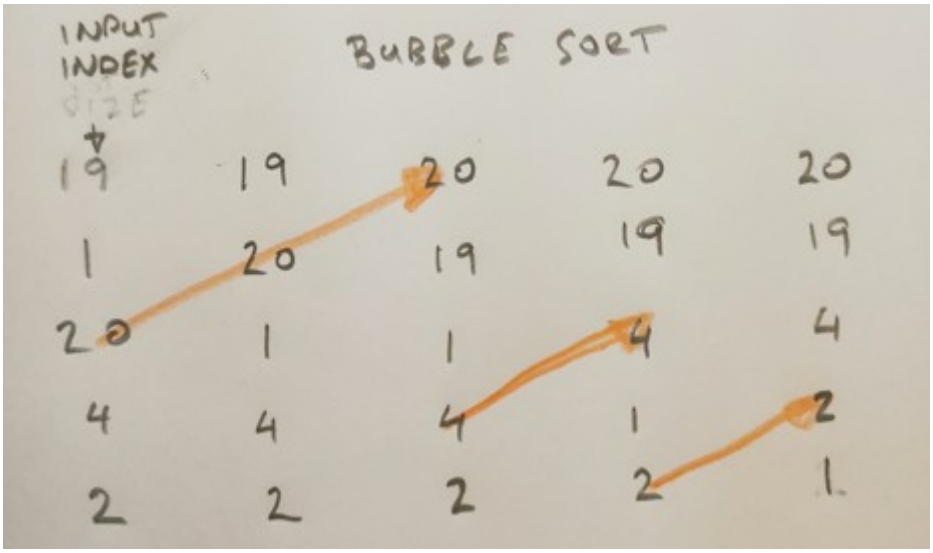


Diagram 2

Space complexity [9]

- O(1) The algorithm only requires using a temporary variable while swapping the largest element

Time complexities [9]

- Worst Case Time Complexity of Bubble Sort is O(N^2)
  - This is the case when the array is reversely sort
- Best Case Time Complexity of Bubble Sort is O(N)
  - This case occurs when the given array is already sorted.
- Average Case Time Complexity of Bubble Sort is O(N^2)
  - The number of comparisons is constant in Bubble Sort so in the average case, there are O(N^2) comparisons.

Python code and execution of the above diagrams

```python
# Bubble Sort
# https://www.geeksforgeeks.org/python-program-for-bubble-sort/
def bubblesort(elements):
  # Looping n-1 times
  print("input: " + str(elements))
  for n in range(len(elements)-1, 0, -1):
    # Loop is short 1 each time as the other element is sorted
    for i in range(n):
      # Comparing each element with the element beside
      if elements[i] > elements[i + 1]:
        # Swapping out of order with the largest element to the right
        elements[i], elements[i + 1] = elements[i + 1], elements[i]
        print("Sorting in-place: " + str(elements))
  return elements
```

✓ 0.8s

```python
list = [2, 4, 20, 1, 19]
# list = [7,5,2,3,1]

print(bubblesort(list))
```

✓ 0.1s

```
input: [2, 4, 20, 1, 19]
Sorting in-place: [2, 4, 1, 20, 19]
Sorting in-place: [2, 4, 1, 19, 20]
Sorting in-place: [2, 1, 4, 19, 20]
Sorting in-place: [1, 2, 4, 19, 20]
[1, 2, 4, 19, 20]
```

Example 5 **cta-sorts.ipynb** code and execution of the Bubble sort of above diagrams

Merge Sort is an efficient comparison-based sort

Merge Sort is a recursive divide-and-conquer approach by diving the index into halves and subdividing until the elements can be easily solved and then merging the results [10]. To visualise the diagram shows the index split into sub-indexes [2,4] [20] [1,19]

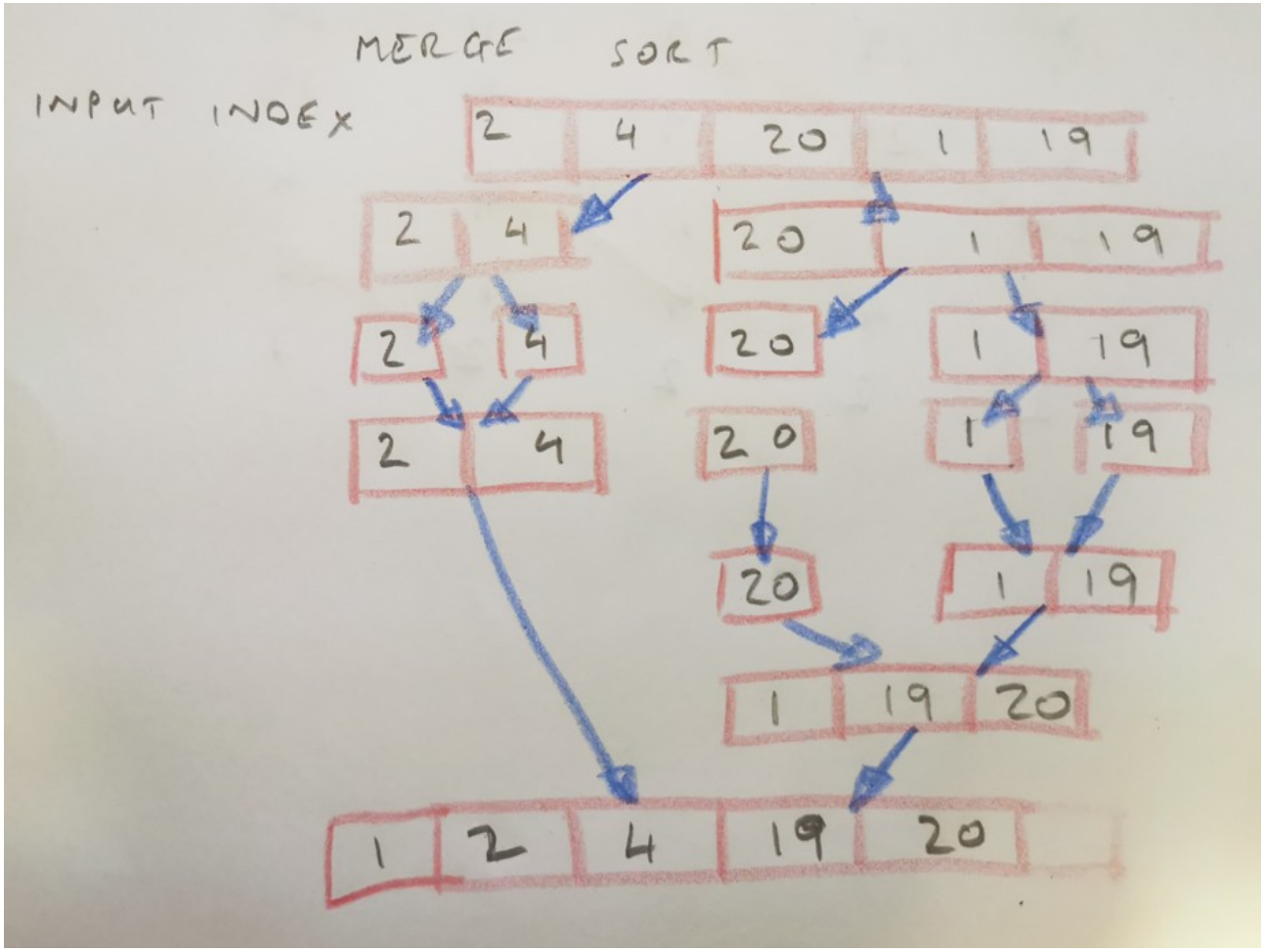


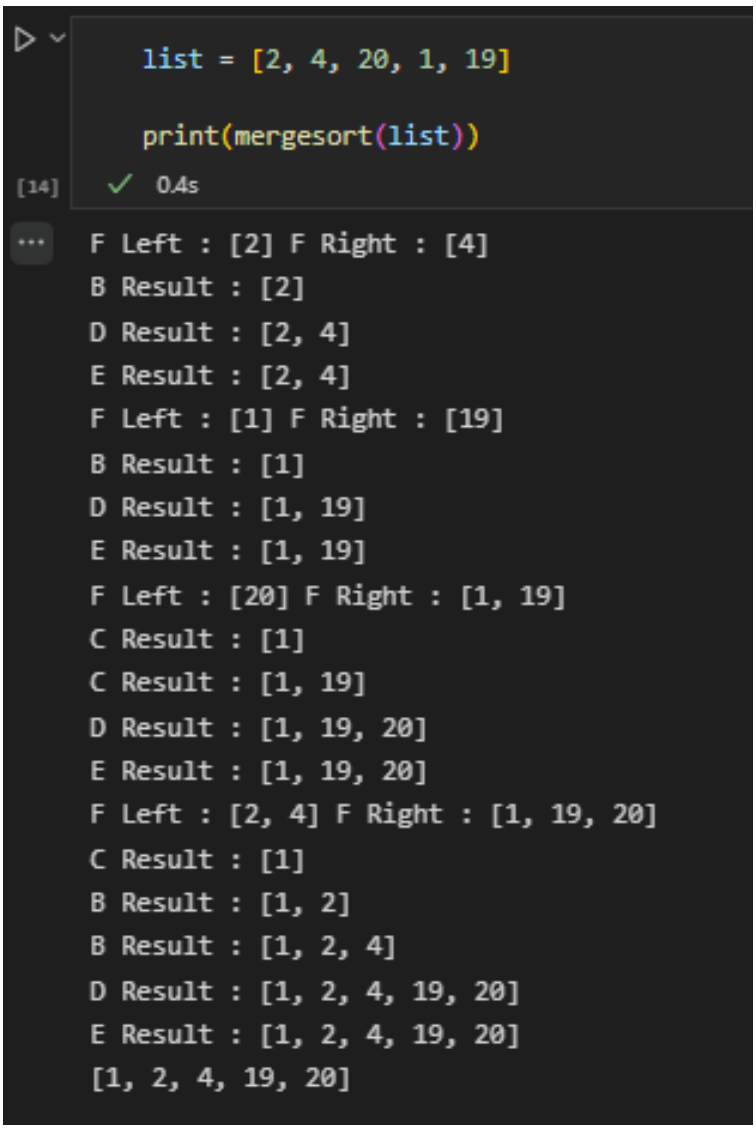
Diagram 3

Space complexity *[11]*

- O(N) as N is the space required for several elements in the index, this is divided and stored for the total space and merged back to a single array.

Time complexities *[12]*

- Merge Sort is O(n*Log n) in all the 3 cases (worst, average and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.



Example 6 **cta-sorts.ipynb** code output of a merge sort of the above diagrams

Counting Sort as a non-comparison sort

A Counting Sort is a non-comparison based algorithm based on keys in a range. It counts the number of elements with distinct values. The distinct value is held in a temporary array and used that to sort the input array [13] Please see the diagram below

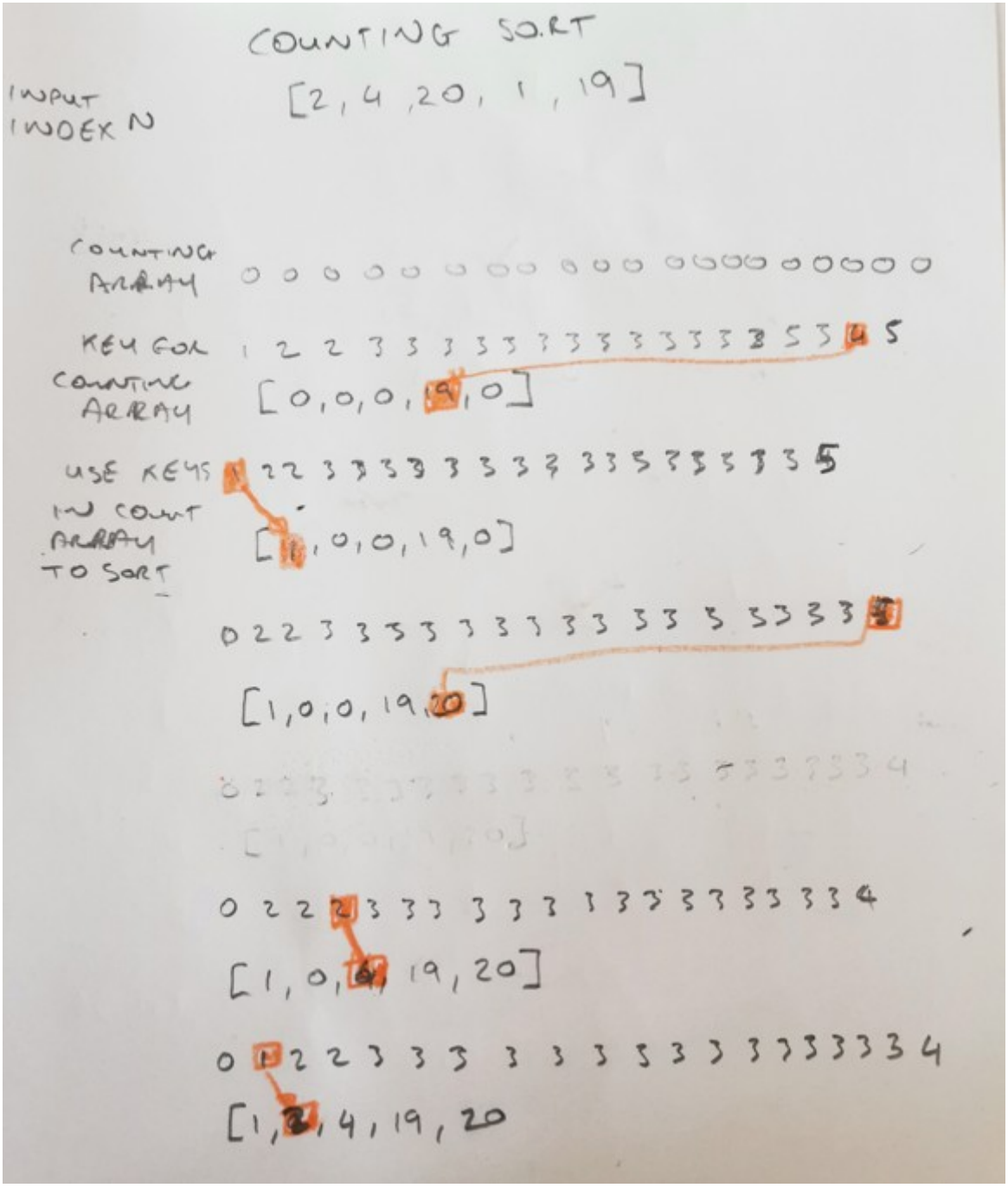


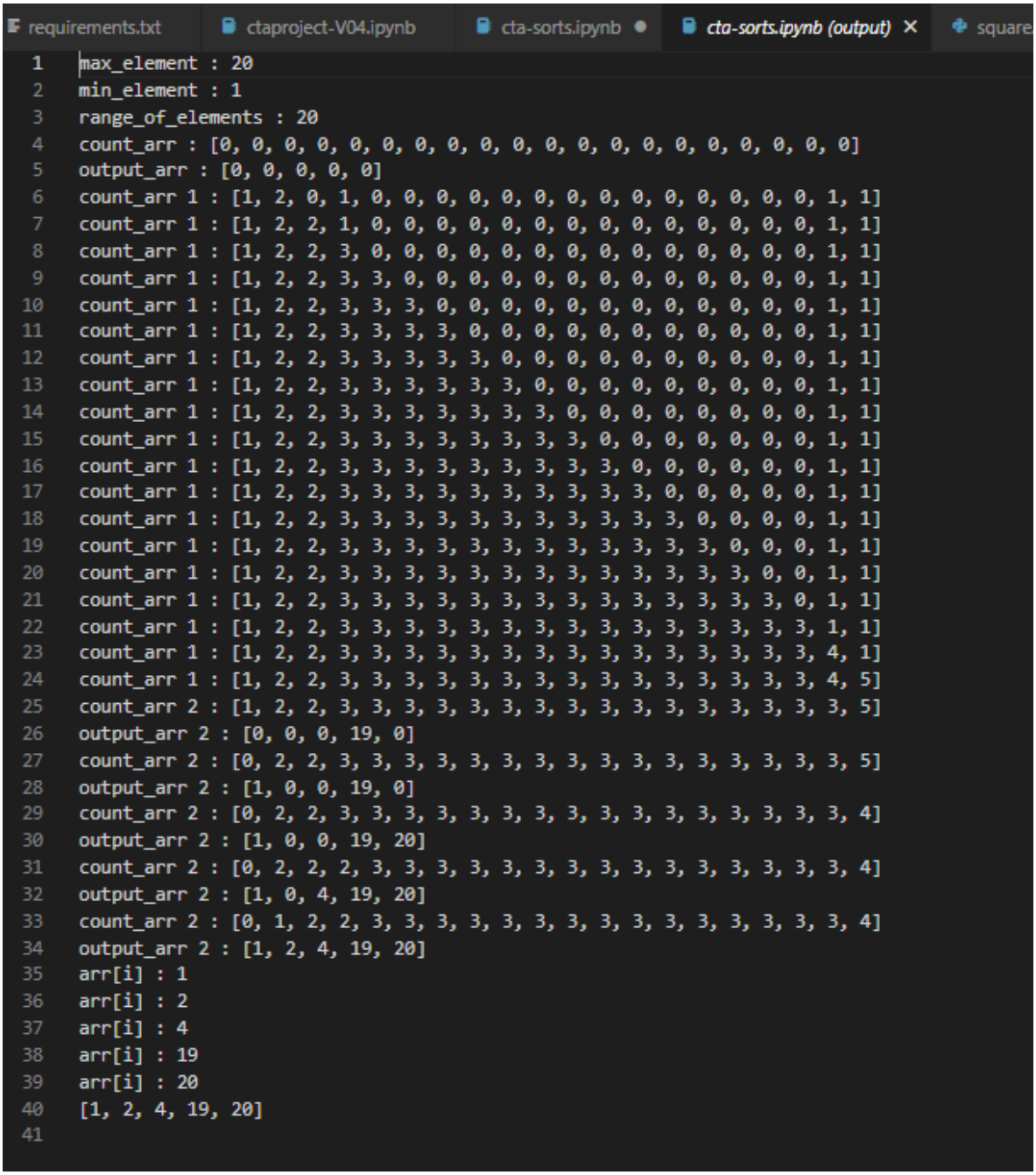
Diagram 4

Space complexity *[14]*

- O(n+k) as an auxiliary array is created to the size of k, k in the example is 20 there for the auxiliary array is the size of 20 with only 5 elements (Example 7) on the input array or create an auxiliary array of 3 if the range is between input is between 1-3 with 5 elements (Example 8). This indicates that space complexity is not great for a wide range of integers as the auxiliary array of that size is required to be created.

Time complexities [9]

- O(n+k) range k of the elements is significantly larger than the other elements deepening the range of elements, Comparing example 7 with 40 steps and example 8 with 23 steps



Example 7 **cta-sorts.ipynb** input index of 5 elements with range 1-20 with the auxiliary of 20

```
max_element : 3
min_element : 1
range_of_elements : 3
count_arr : [0, 0, 0]
output_arr : [0, 0, 0, 0, 0]
count_arr 1 : [3, 4, 1]
count_arr 1 : [3, 4, 5]
count_arr 2 : [2, 4, 5]
output_arr 2 : [0, 0, 1, 0, 0]
count_arr 2 : [2, 3, 5]
output_arr 2 : [0, 0, 1, 2, 0]
count_arr 2 : [2, 3, 4]
output_arr 2 : [0, 0, 1, 2, 3]
count_arr 2 : [1, 3, 4]
output_arr 2 : [0, 1, 1, 2, 3]
count_arr 2 : [0, 3, 4]
output_arr 2 : [1, 1, 1, 2, 3]
arr[i] : 1
arr[i] : 1
arr[i] : 1
arr[i] : 2
arr[i] : 3
[1, 1, 1, 2, 3]
```

Example 8 **cta-sorts.ipynb** input index of 5 elements with range 1-3 with the auxiliary of 3

Selection Sort is a simple comparison-based sort.

A Selection Sort essentially finds the minimum element in an array and inserts it into its position with the element in its place. [15]

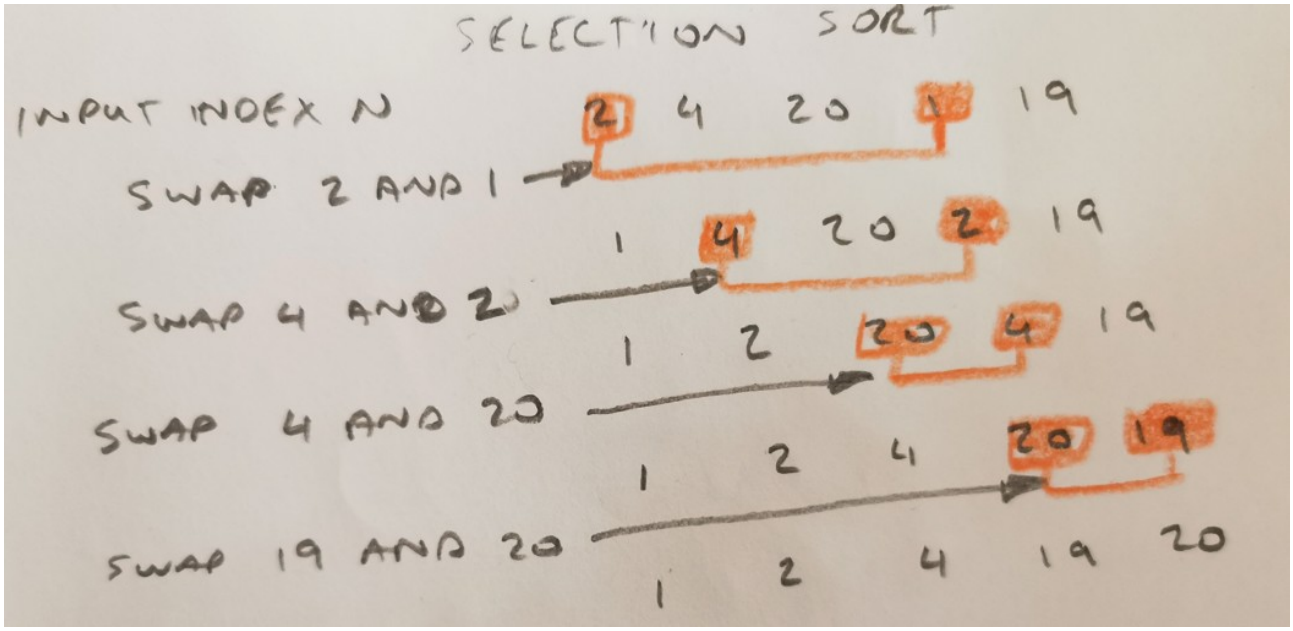


Diagram 5

Space complexity *[16]*

- O(1) Similar to bubble sort the algorithm only requires using a temporary variable for the to hold an element in the array while swapping the elements

Time complexities [16]

- Worst Case Time Complexity of Selection Sort is O(N^2)
  - This is the case if the given array is sorted except for the last element is the smallest
- Best Case Time Complexity of Selection Sort is O(N^2)
  - This case occurs when the given array is already sorted.
- Average Case Time Complexity of Selection Sort is O(N^2)
  - The number of comparisons is constant in Selection Sort so in an average case, there are O(N^2) comparisons.

```
Sorting Stable 0: [2, 4, 20, 1, 19]
outer: 0
inner: 1
inner: 2
inner: 3
Sorting Stable 1: 3
inner: 4
Sorting Stable 2: 2
Sorting Stable 3: 1
Sorting Stable 4: 2
Sorting Stable 5: [1, 4, 20, 2, 19]
```

Example 9 **cta-sorts.ipynb** code output of a Selection sort of the above diagrams

Insertion Sort is a simple comparison-based sort.

An Insertion Sort compares each pair in if the minimum element in an array is found it inserts that element into the correct position. [17]

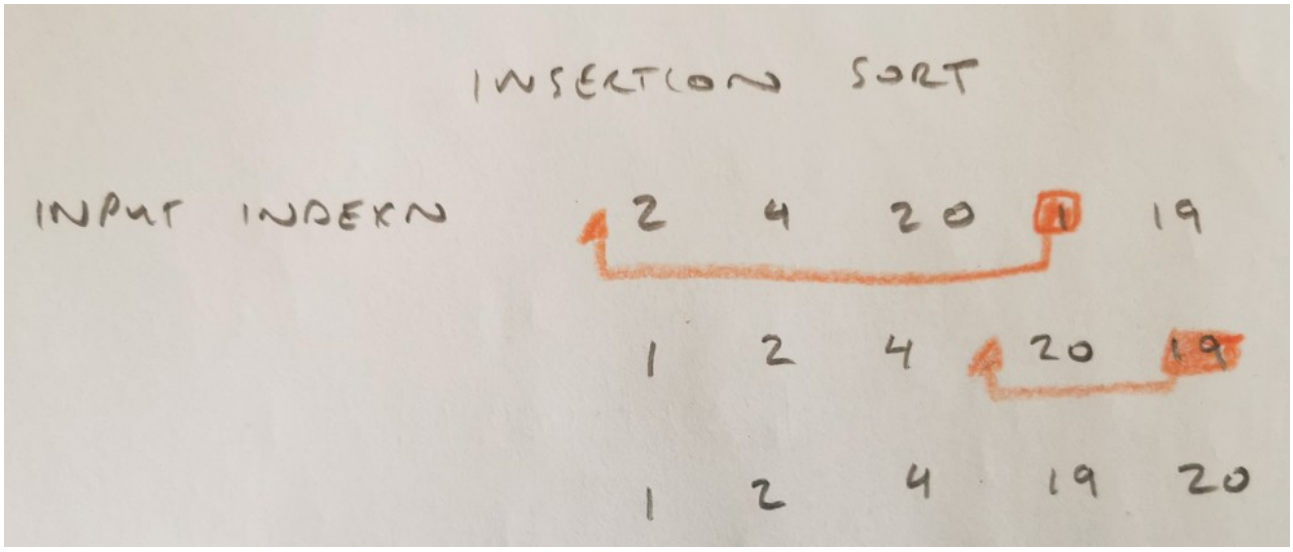


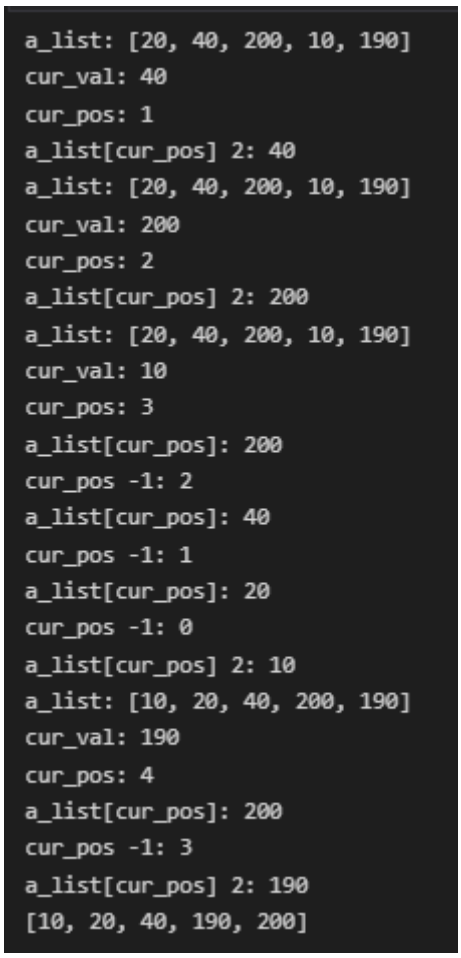
Diagram 6

Space complexity *[18]*

- O(1) Similar to Bubble sort and Selection sort the algorithm only requires space by shifting the element to right and inserting the element at the appropriate position.

Time complexities [18]

- Worst Case Time Complexity of Insertion Sort is O(N^2)
  - This is the case if the array is reversely sorted
- Best Case Time Complexity of Insertion Sort is O(N)
  - This case occurs when the given array is already sorted.
- Average Case Time Complexity of Insertion Sort is O(N^2)
  - The number of comparisons is constant in Insertion Sort so in an average case, there are O(N^2) comparisons.



Example 10 **cta-sorts.ipynb** code output of an Insertion sort of the above diagrams

Implementation & Benchmarking

This section of the report is to introduce the implementation using a Jupyter notebook to time and test all the python sorting functions and output the required table and graphs. The results of each of the benchmarks are discussed and broken into 3 parts. Information on extra notes and scripts used in the project and a conclusion.

Jupyter notebook ctaproject-V05.ipynb

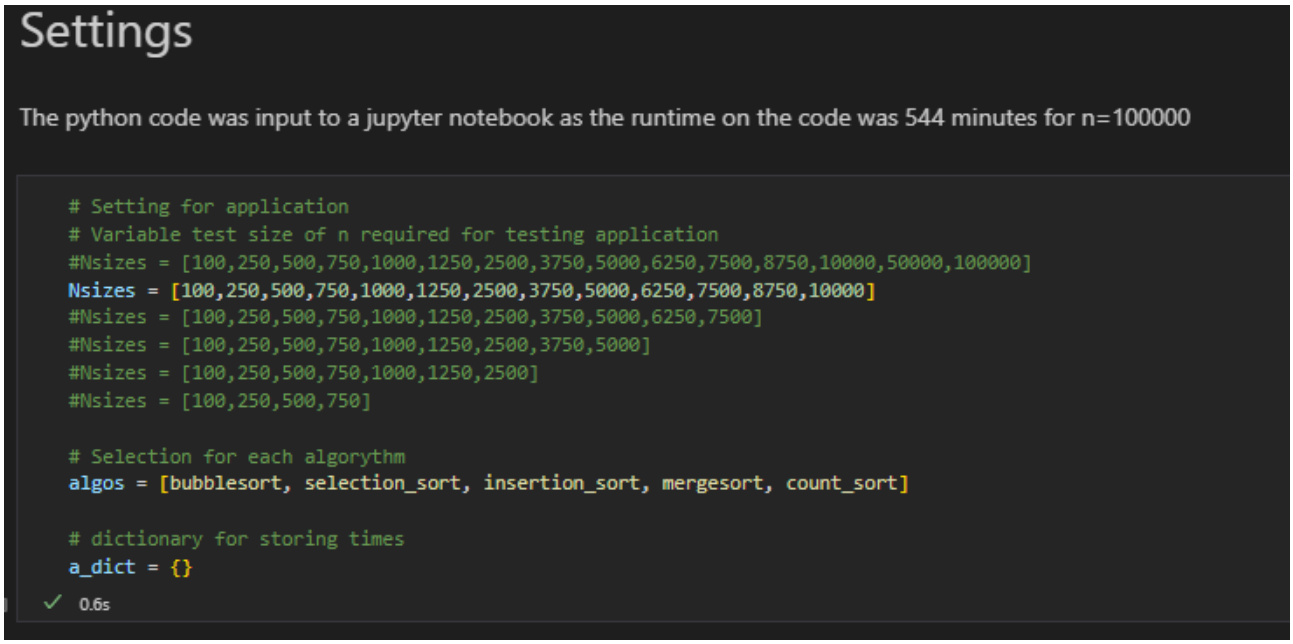
The project was implemented in a Jupyter notebook called **ctaproject-V05.ipynb**, with the five sorting algorithms. Each of the sorting functions is labelled and commented on appropriately and included some information from this project.

- Bubble Sort is a simple comparison-based sort [20]
- Merge Sort is an efficient comparison-based sort [21]
- Counting Sort as a non-comparison sort [22]
- Selection Sort is a simple comparison-based sort. [23]
- Insertion Sort is a simple comparison-based sort. [24]

Settings Example 11

Settings (not including imports) have the following

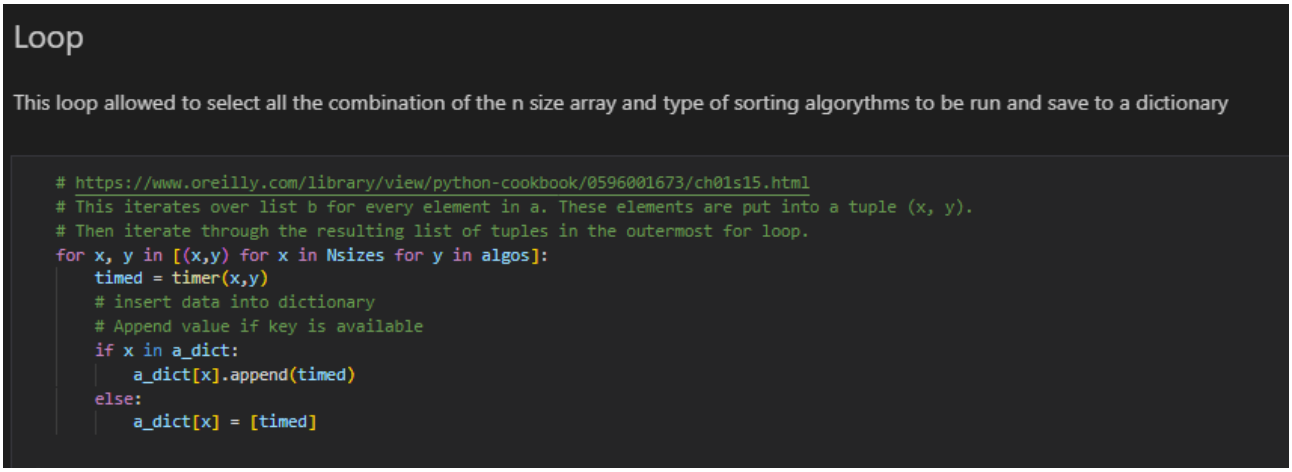
- A list of sorting algorithms names **(algos)**
    - To call the functions
- A list with varying integers sizes **(Nsizes)**
    - To create the sizes of N
- A dictionary **(a_dict)**
    - To hold the results of the timer.



Example 11 Setting Code Snippet

For Loop Example 12

A for loop [19] iterates over list **Nsizes** for every element in **algos** calling the function **timer** and adds them to a **a_dict**



Example 12 For Loop Code Snippet

Benchmarking Example 13

For benchmarking a **timer** function was adapted from 1.2.2 Python of the CTA Project Specification pdf [8]. This accepted two inputs x (**Nsizes**) and y (**algos**). A random array is created using Numpy with a range of numbers 1 – 100 and size (**Nsizes).** A section is commented out on a second random array to check how a larger range of 1 - 100000 would affect the timing on the Count Sort. A While Loop is used to count 10 runs of sorting algorithms, the time.time module records the times in seconds before and after each run of an algorithm and array size. The recorded times are subtracted and multiplied by 1000 to get the milliseconds as required by the Project Specifications. All 10 runs record are added and then divided by 10 to get the average run. The average run is returned and stored in the dictionary **a_dict**.

Benchmarking

Timer code and random array

This code was a adapted from 1.2.2 Python of the CTA Project Specification pdf [8]. The notable difference is the use of numpy to generate the random array.

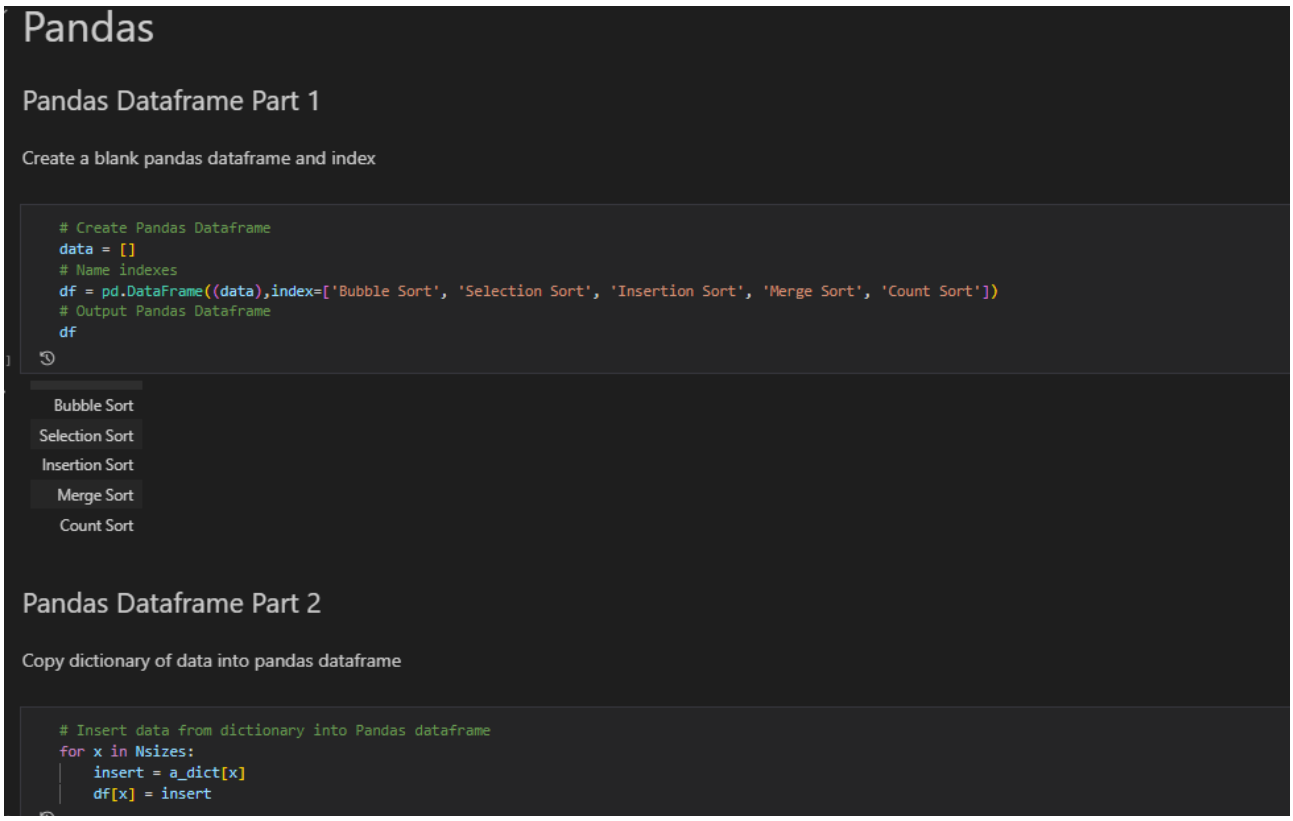
```
# Create a timer using a while loop
def timer(x,y):
    # Set i to 0
    i = 0
    # Set total time to 0
    totaltime = 0

    # Begin 10 runs for algorithm
    # While i is less than 10
    while i < 10:
        # Create a random array
        #getarray = random.randint(100, size=(x))
        # Create a second array to test larger range of number
        getarray = random.randint(100000, size=(x))
        # Extracted from CTA Project Specification
        # Start time
        start_time = time.time()
        # call your function
        y(getarray)
        # End time
        end_time = time.time()
        # Get time for 1 run in milliseconds
        time_elapsed = (end_time - start_time)*1000
        # Add 1 run to totaltime
        totaltime = totaltime + time_elapsed
        # Increase i by 1
        i = i + 1
    # get the averge time by dividing all runs by 10
    getavgtime = totaltime/10
    return getavgtime
```

Example 13 Benchmarking Code Snippet

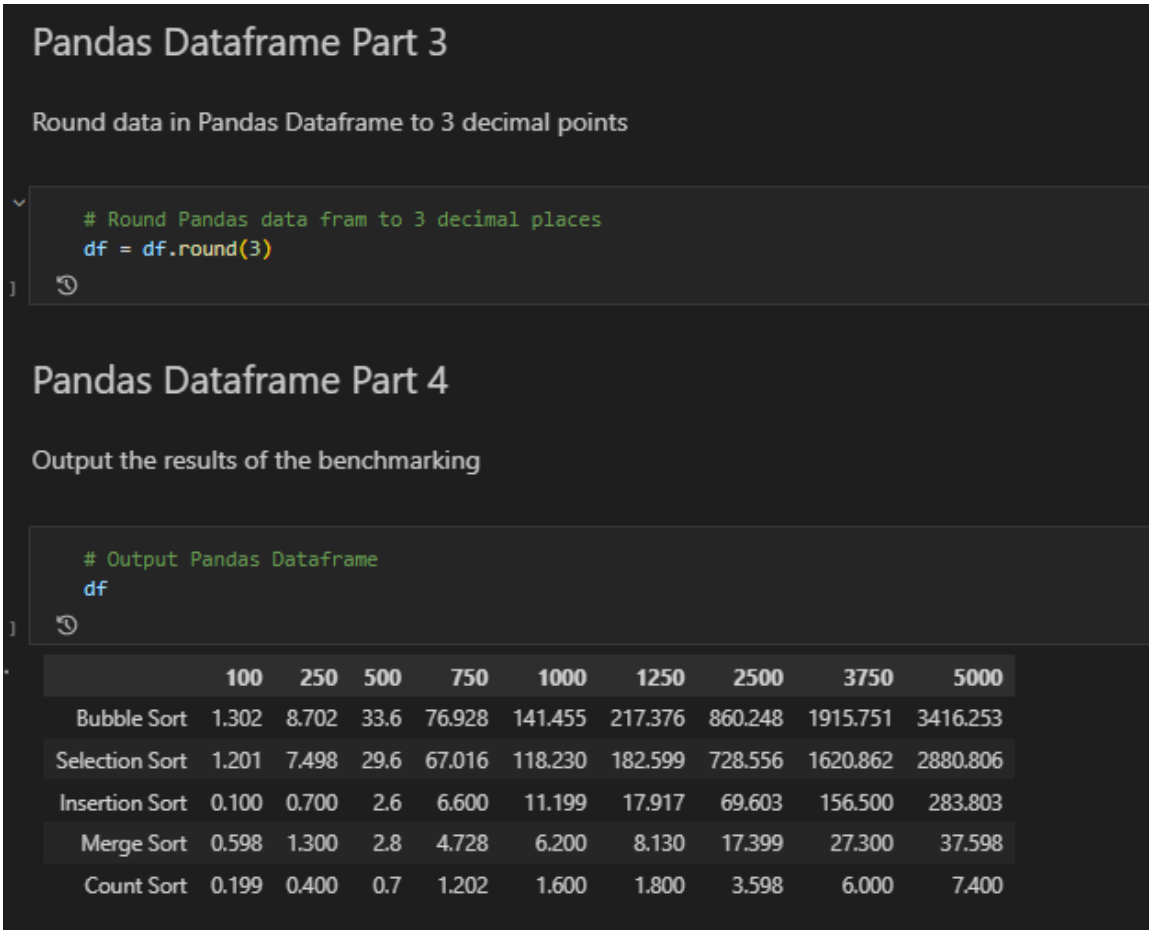
Pandas Example 14 – 17

A pandas dataframe is created with the index for each of the sorting algorithms and **a_dict** is inserted into the Pandas Dataframe.



Example 14 Pandas Code Snippet

All data is then rounded to 3 decimal places and a table of the results is presented to the user.



	100	250	500	750	1000	1250	2500	3750	5000
Bubble Sort	1.302	8.702	33.6	76.928	141.455	217.376	860.248	1915.751	3416.253
Selection Sort	1.201	7.498	29.6	67.016	118.230	182.599	728.556	1620.862	2880.806
Insertion Sort	0.100	0.700	2.6	6.600	11.199	17.917	69.603	156.500	283.803
Merge Sort	0.598	1.300	2.8	4.728	6.200	8.130	17.399	27.300	37.598
Count Sort	0.199	0.400	0.7	1.202	1.600	1.800	3.598	6.000	7.400

Example 15 Pandas Code Snippet

This produces a graph overlay of each of the chosen sorting algorithms.

Pandas Dataframe Part 5

Graph to summarise the results obtained This overlays all the results from each of the chosen sorting Algorithms

- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Count Sort

```
# https://pandas.pydata.org/pandas-docs/version/0.23/generated/pandas.DataFrame.plot.html
# Create a comparison plot for sorts
ax = df.T.plot(figsize=(14, 12), marker='x')
# Label graphs
ax.set_ylabel('Average time for 10 runs for each algorithm', fontsize=12)
ax.set_xlabel('Input Sizes', fontsize=12)
```

Example 16 Pandas Code Snippet

The second graph is produced for each sorting algorithm

Pandas Dataframe Part 6

Graph to summarise the results obtained This outputs individual graphs for each of the chosen sorting Algorithms

- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Count Sort

```
# https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.plot.line.html
# Create a indidviual plot for each sort
axes = df.T.plot(figsize=(14, 12), marker='x', subplots=True)
```

Example 17 Pandas Code Snippet

Results of Benchmarking

Benchmarking was tested in 3 different ways

1. n size 100 – 10000 with a random range of 1 – 100
2. n size 100 – 10000 with a random range of 1 – 100000
3. n size 100 – 100000 with a random range of 1 – 100

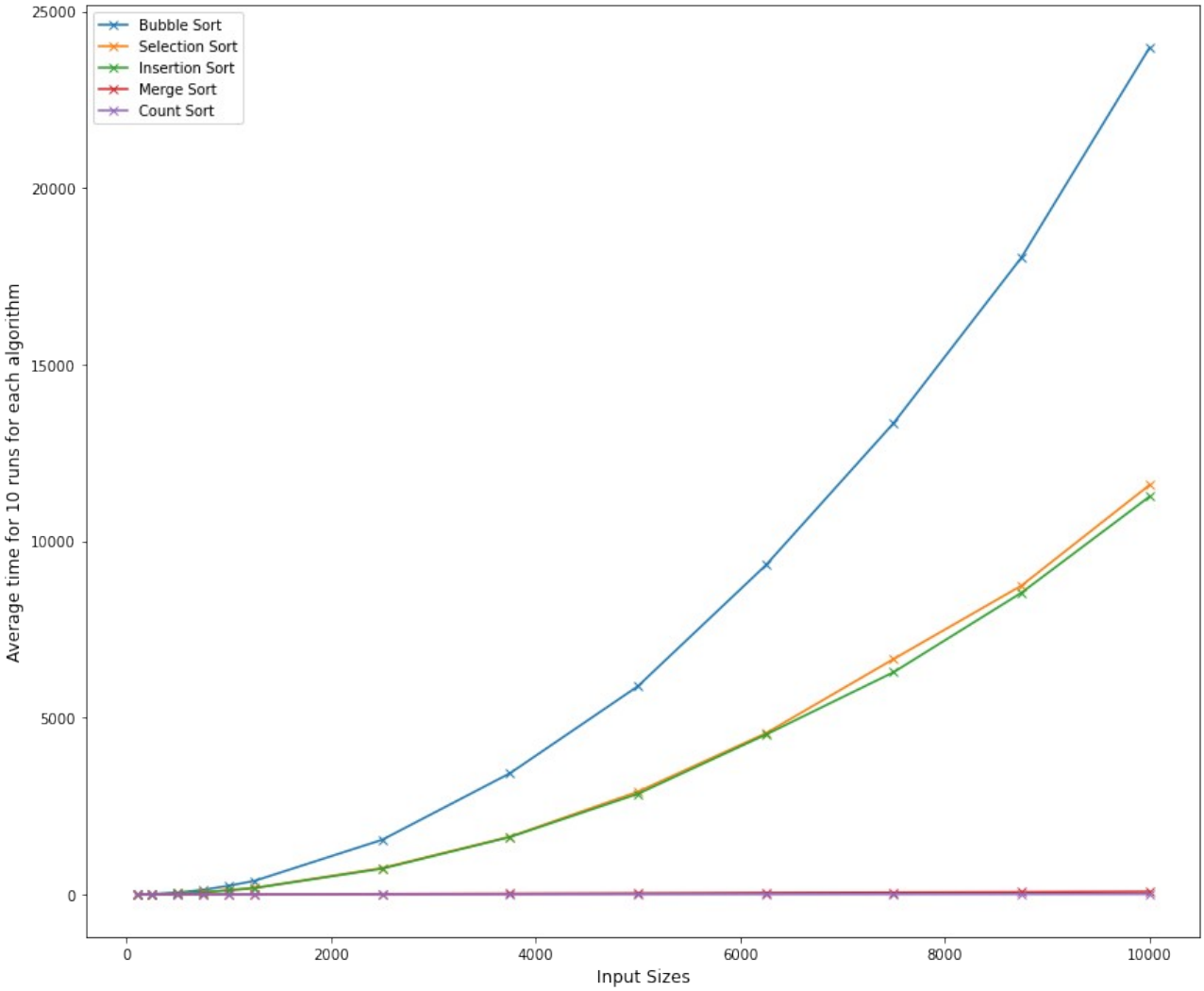
Part 1 n=10000 Range 1-100 Example 18 – 20

The results of the benchmarking are presented show simple comparison sorts, Bubble Sort taking over 23 seconds to complete n=10000 and followed close behind with Selection and Insertion sort over 11 seconds. All three of the simple comparison-based sort displayed the growth rate of O(N^2).

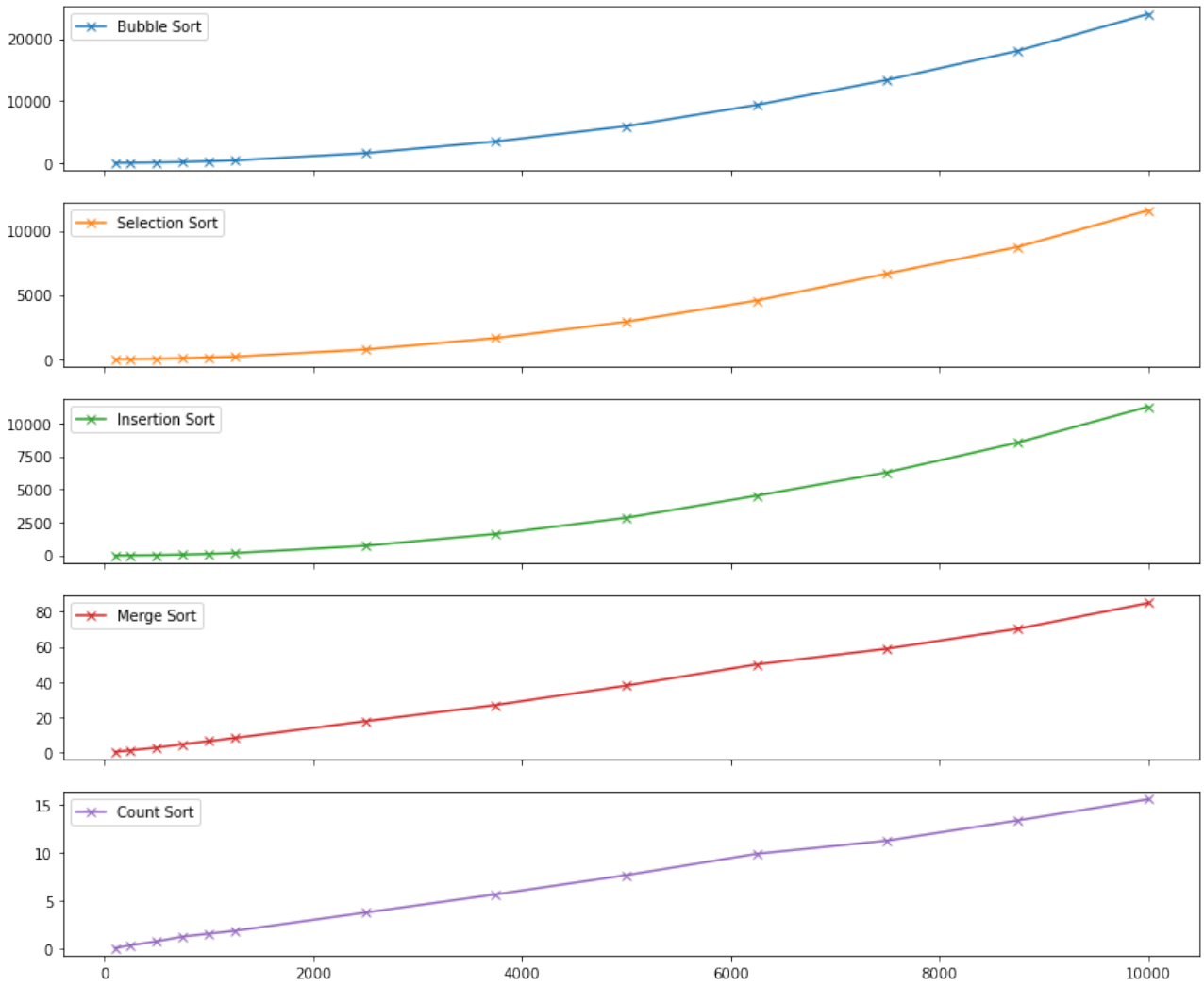
Merge sort in 84 milliseconds and Counting Sort in 15 milliseconds

	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
Bubble Sort	2.199	14.742	57.943	134.275	251.473	386.383	1551.050	3435.959	5899.007	9322.027	13345.634	18035.095	23965.236
Selection Sort	1.200	7.800	29.316	67.915	135.226	189.402	752.499	1640.735	2913.865	4565.328	6668.350	8745.737	11590.342
Insertion Sort	1.198	6.899	27.982	64.002	114.577	181.900	733.452	1628.932	2851.018	4526.685	6295.182	8544.478	11262.623
Merge Sort	0.599	1.400	2.932	4.899	6.600	8.401	17.910	27.097	38.097	50.100	59.002	70.298	84.872
Count Sort	0.100	0.400	0.798	1.300	1.598	1.898	3.799	5.702	7.699	9.925	11.299	13.402	15.599

Example 18 Results of Benchmarking Part 1



Example 19 Results of Benchmarking Part 1



Example 20 Results of Benchmarking Part 1

Part 2 n=10000 Range 1-100000 Example 21 – 24

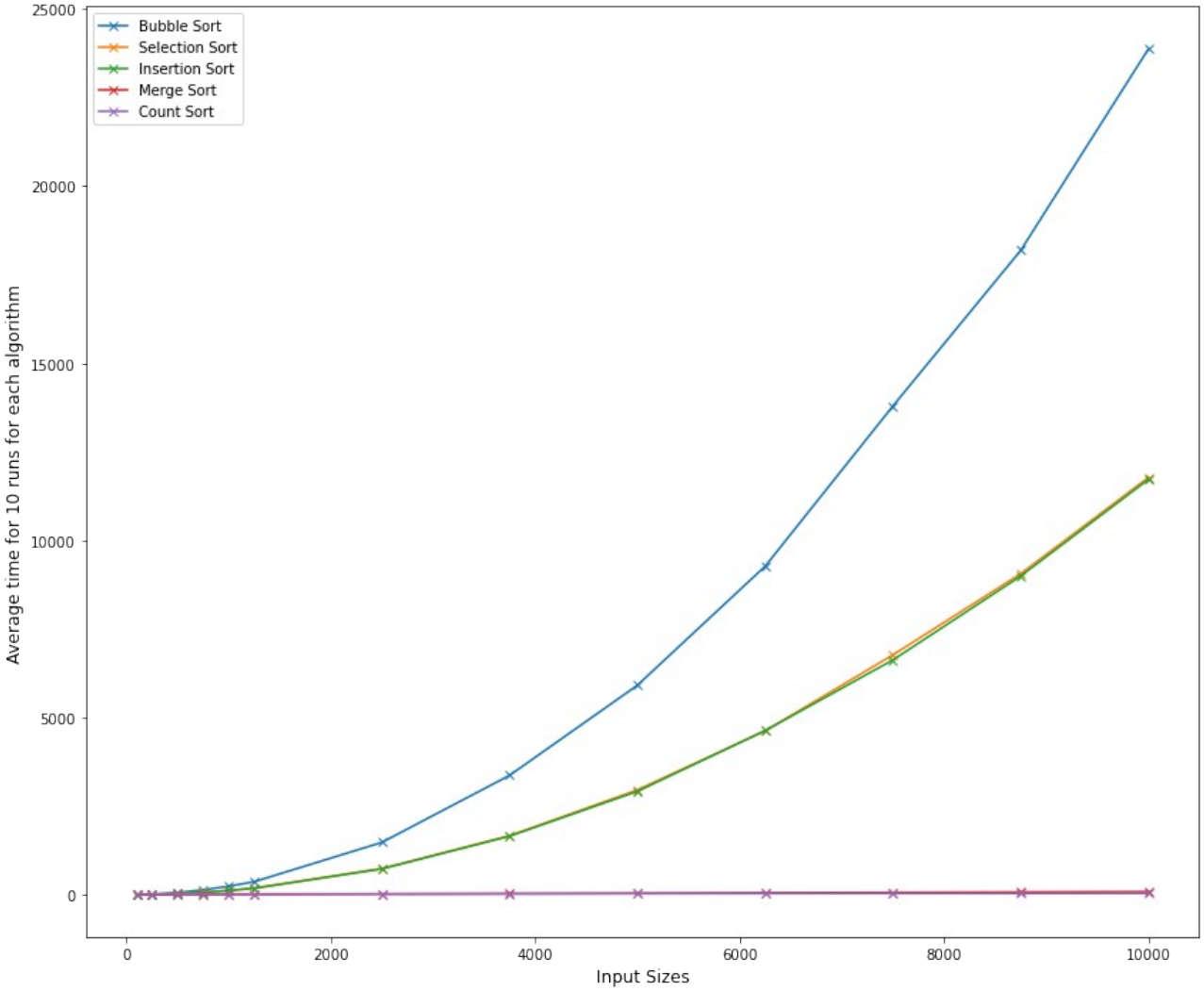
Part 2 was to test if the range of numbers was increased 1000 times from a range of 100 to a range of 100000 and what effect this would have on the benchmark results. The results showed no significant changes in 4 of the 5 sorting algorithms. The time and space were affected in the Counting Sort indicating a nearly ~ 2.4 increase in the time to sort the range with an n=10000

	N10000 * Range 1-100	N10000 * Range 1-100000	Time and Space Difference
Bubble Sort	23965.236	23876.229	0.996
Selection Sort	11590.342	11778.225	1.016
Insertion Sort	11262.623	11724.101	1.041
Merge Sort	84.872	83.016	0.978
Count Sort	15.599	37.402	2.398

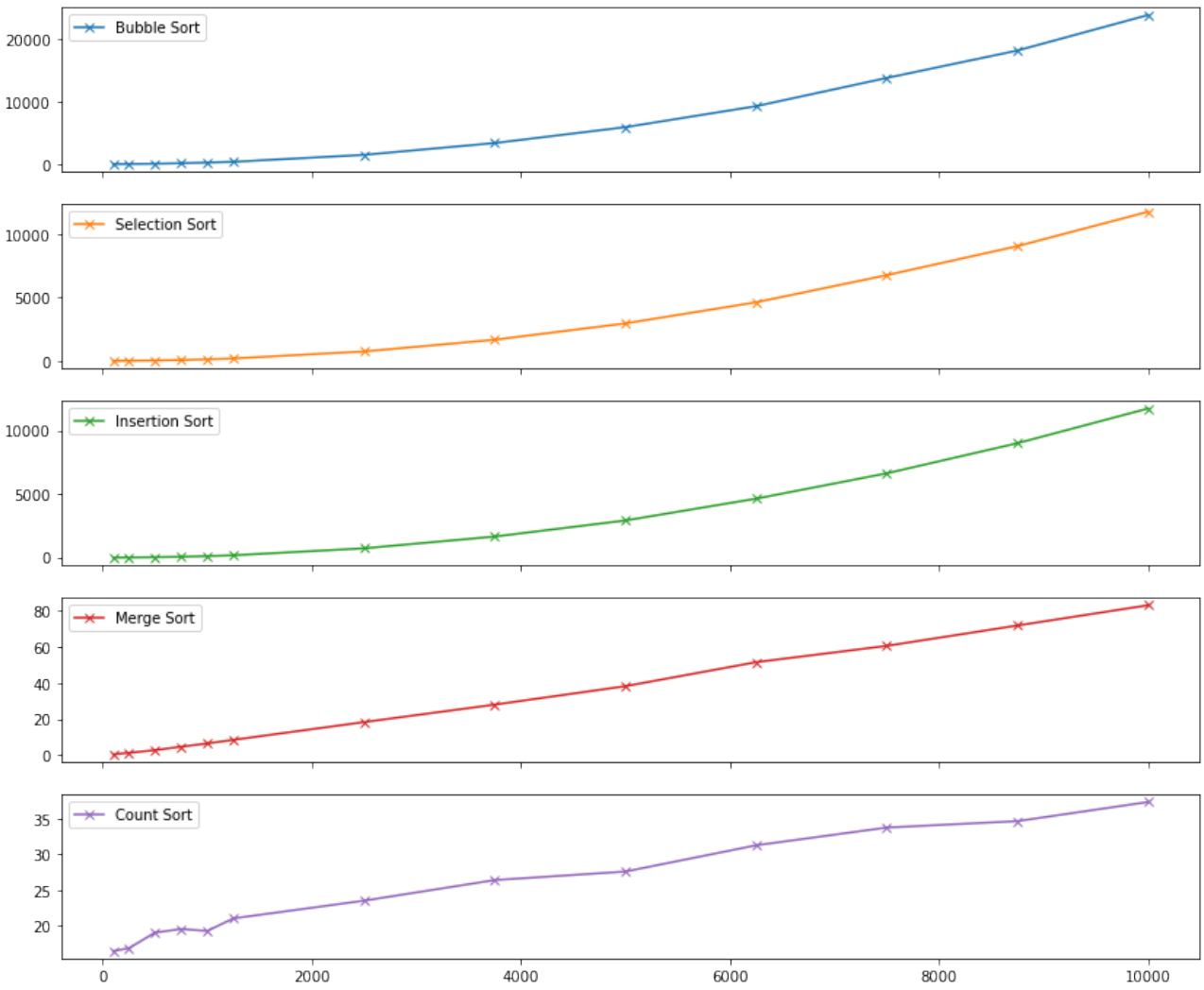
Example 21 Comparing Solve times of ranges 100 versus 100000

	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
Bubble Sort	2.500	14.101	57.798	134.639	236.272	365.42	1476.499	3369.596	5912.256	9272.345	13795.078	18186.140	23876.229
Selection Sort	1.200	7.402	29.601	67.658	118.128	186.942	738.637	1666.270	2957.652	4625.845	6767.717	9061.130	11778.225
Insertion Sort	1.001	6.597	27.201	62.827	114.933	178.327	727.090	1652.653	2917.457	4632.351	6625.599	8995.003	11724.101
Merge Sort	0.499	1.303	2.800	4.700	6.499	8.504	18.325	28.002	38.206	51.499	60.526	71.797	83.016
Count Sort	16.398	16.799	19.001	19.499	19.229	20.999	23.498	26.398	27.600	31.298	33.798	34.704	37.402

Example 22 Results of Benchmarking Part 2



Example 23 Results of Benchmarking Part 2



Example 24 Results of Benchmarking Part 2

Part 3 n=100000 Range 1-100 Example 25 – 31

This was the initial testing of using n=100000 was not efficient due to time constraints and an error in the layout of the script. Both of these issues produced some interesting results and contributed to my understanding of sorting algorithms' time and space. These are the discoveries:

- This was only run once as the total runtime was 544 minutes, as all the sorting algorithms required a higher time and space complexity
- When compared to a run of n=10000 table it was noted the average time was ~100 times slower. This means an estimation could be made for all n sizes, for example below estimating an n=1000000

Size	N=100	N=10000	Difference N100 & N10000	N=100000	Difference N10000 & N100000	N=1000000 Estimated
Bubble Sort	136.347	13503.966	99.041	1326520.505	98.232	130306762.247
Selection Sort	118.05	11447.026	96.968	1163724.774	101.662	118306587.974
Insertion Sort	11.222	1140.233	101.607	109400.346	95.946	10496525.597
Merge Sort	6.214	80.602	12.971	961.851	11.933	11477.768
Count Sort	1.502	14.801	9.854	144.971	9.795	1419.991

Example 25 Estimating n=10000000

- An error in the coding when the run on the same random array 10 times (**getarray**) Insertion time did not follow the curve and had significantly lower times than expected of ~10 times faster, Bubble Sort was ~2 times as fast. Selection, Merge and Count times were not significant enough to note. See Example 28 for comparison

```
    # Set total time to 0
    totaltime = 0
    # Create a random array
    getarray = random.randint(100, size=(x))
    # Create a second array to test larger range of number
    #getarray = random.randint(100000, size=(x))
    # Begin 10 runs for algorithm
    # While i is less than 10
    while i < 10:
        # Extracted from CTA Project Specification
        # Start time
        start_time = time.time()
```

Example 26 **getarray** outside while loop

```
# Set total time to 0
totaltime = 0

# Begin 10 runs for algorithm
# While i is less than 10
while i < 10:
    # Create a random array
    #getarray = random.randint(100, size=(x))
    # Create a second array to test larger range of number
    getarray = random.randint(100000, size=(x))
    # Extracted from CTA Project Specification
    # Start time
    start_time = time.time()
```

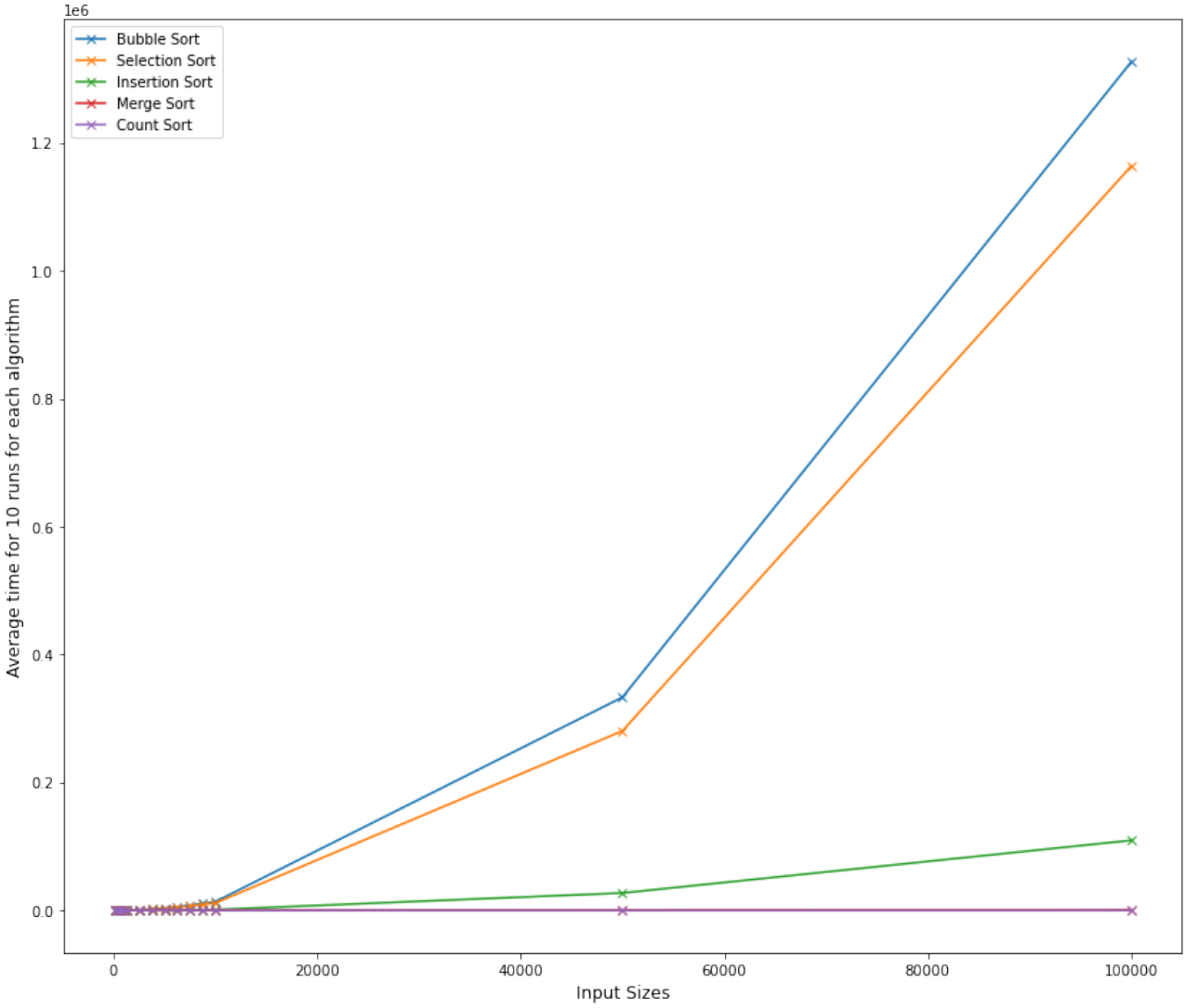
Example 27 **getarray** inside while loop

	Example 26 getarray outside while loop n=10000	Example 27 getarray inside while loop n=10000	Time and Space Difference
Bubble Sort	23965.236	13503.966	0.563
Selection Sort	11590.342	11447.026	0.988
Insertion Sort	11262.623	1140.233	0.101
Merge Sort	84.872	80.602	0.95
Count Sort	15.599	14.801	0.949

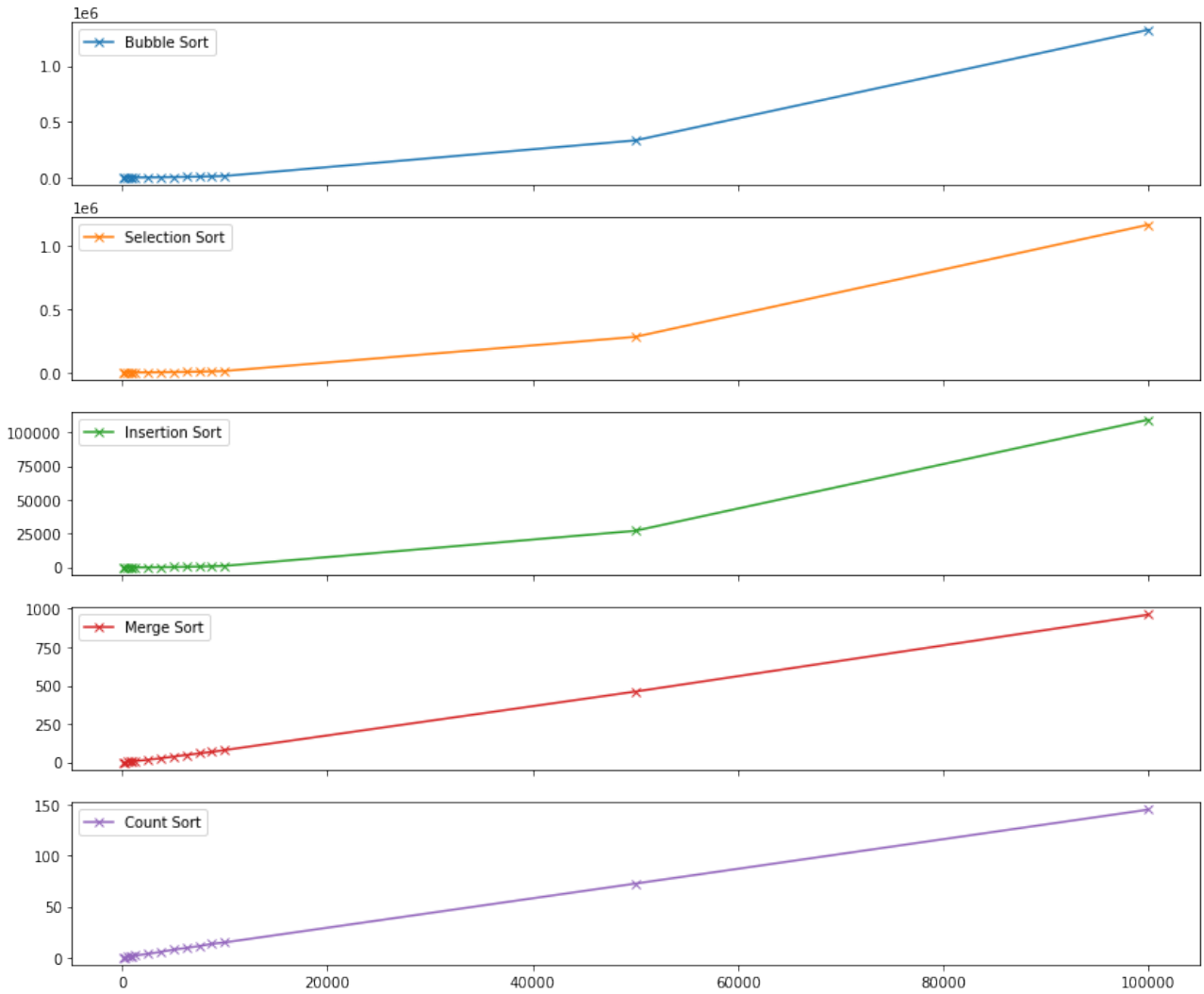
Example 28 Compare results of error in code

	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000	50000	100000
Bubble Sort	1.397	8.702	34.320	77.302	136.347	219.997	851.961	1943.923	3442.274	5360.560	7912.948	10455.078	13503.966	333145.420	1326520.505
Selection Sort	1.401	7.297	29.503	65.900	118.050	198.759	733.154	1683.706	2938.304	4649.582	6610.630	8915.093	11447.026	280528.138	1163724.774
Insertion Sort	0.200	0.800	2.797	6.498	11.222	18.526	72.700	164.400	292.741	455.131	646.770	885.578	1140.233	27115.555	109400.346
Merge Sort	0.500	1.400	2.900	4.800	6.214	8.600	17.200	26.703	37.899	48.399	59.102	68.902	80.602	461.208	961.851
Count Sort	0.200	0.400	0.700	1.101	1.502	1.915	3.600	5.498	7.700	9.400	11.198	13.400	14.801	72.457	144.971

Example 29 Results of Benchmarking Part 3



Example 30 Results of Benchmarking Part 3



Example 31 Results of Benchmarking Part 3

Conclusion Example 32

This project has allowed me a greater understanding of sorting algorithms and space and time complexities. The times in the table and graphs comparisons are the expected results given in the course. As demonstrated various factors came into play including the machine's performance. Testing the Jupyter notebook on another machine will not exactly match output tables and graphs but the expected behaviour of each of the chosen sorting algorithms. The test was run over ten times for each **Nsize** on a random NumPy array with different ranges. Changing the ranges did affect the outcome of the Counting Sort increasing the times to solve by ~2.4 times. Counting Sort was still overall the fast algorithm of the five chosen. The changes made also proved that Comparisons based (In-place) sorting does not require extra memory as the input data is usually overwritten by the output, only swapping elements within the input size n. In the larger range of numbers 1 – 100000, a Non-comparison sort can achieve linear n running time in the best case but are less flexible, as demonstrated with the Counting Sort creating a larger second key array significantly increasing the memory requirements. It was also noted that the error made on the script using the same random array saw a significant decrease in times for Insertion (10 times) and Bubble Sort (2 times), and would require further investigation.

In Summary, the results of the benchmarking are in line with the introduction of each of the chosen sorting algorithms.

	Best Case	Worst Case	Average Case	Space complexity
Bubble Sort	O(N)	O(N^2)	O(N^2)	O(1)
Selection Sort	O(N^2)	O(N^2)	O(N^2)	O(1)
Insertion Sort	O(N)	O(N^2)	O(N^2)	O(1)
Merge Sort	O(n*Log n)	O(n*Log n)	O(n*Log n)	O(N)
Count Sort	O(n+k)	O(n+k)	O(n+k)	O(n+k)

Example 32 Conclusion

Extra Scripting and notes

A second Jupyter notebook **cta-sorts.ipynb** is included to demonstrate the bespoke diagrams run in code and the outputs at the different stages. The code cta-sorts.ipynb is a reference in the examples 5 – 10

A third script is included square.py used in Examples 1 – 3

Requirement.txt contains the required imports for the Jupyter notebook.

References

[1] Mannion, P., 2022. *Sorting Algorithms Part 1*
[2] John D. Cook | Applied Mathematics Consulting. 2022. Sorting. [online] Available at: <https://www.johndcook.com/blog/2011/07/04/sorting/>
[3] square, P. and Parikh, J., 2022. Python for loops program with square. [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/41876325/python-for-loops-program-with-square>
[4] Studytonight.com. 2022. Time Complexity of Algorithms | Studytonight. [online] Available at: <https://www.studytonight.com/data-structures/time-complexity-of-algorithms>
[5] Mannion, P., 2022. Sorting Algorithms Part 2
[6] GeeksforGeeks. 2022. In-Place Algorithm - GeeksforGeeks. [online] Available at: <https://www.geeksforgeeks.org/in-place-algorithm/>
[7] Mannion, P., 2022. Sorting Algorithms Part 3
[8] Mannion, P., 2022. CTA_Project.
[9] Simplilearn.com. 2022. Time and Space complexity of Bubble Sort [online] Available at: <https://www.simplilearn.com/tutorials/data-structure-tutorial/bubble-sort-algorithm>
[10]GeeksforGeeks. 2022. Merge Sort - GeeksforGeeks. [online] Available at: <https://www.geeksforgeeks.org/merge-sort/>
*[11]Opengenus.org. 2022 . Time & Space Complexity of Merge Sort. [online] Available at: <*https://iq.opengenus.org/time-complexity-of-merge-sort/*>*
[12]Studytonight.com. 2022. Merge Sort Algorithm | Studytonight. [online] Available at: <https://www.studytonight.com/data-structures/merge-sort>
*[13]*GeeksforGeeks. 2022. *Counting Sort - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/counting-sort/>
[14]Opengenus.org. 2022 . Time & Space Complexity of Counting Sort. [online] Available at: <https://iq.opengenus.org/time-and-space-complexity-of-counting-sort/>
[15]GeeksforGeeks. 2022. Selection Sort - GeeksforGeeks. [online] Available at: <https://www.geeksforgeeks.org/stable-selection-sort/?ref=gcse>

[16]Opengenus.org. 2022 . Time & Space Complexity of Selection Sort. [online] Available at: <*https://iq.opengenus.org/time-complexity-of-selection-sort/*>
[17]GeeksforGeeks. 2022. Insertion Sort - GeeksforGeeks. [online] Available at: <*https://www.geeksforgeeks.org/insertion-sort/*>
[18]Opengenus.org. 2022 . Time & Space Complexity of Insertion Sort. [online] Available at: <*https://iq.opengenus.org/insertion-sort-analysis/*>
[19] O'Reilly Online Learning. 2022. Python Cookbook. [online] Available at: <*https://www.oreilly.com/library/view/python-cookbook/0596001673/ch01s15.html*>
[20] GeeksforGeeks. 2022. *Python Program for Bubble Sort - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/python-program-for-bubble-sort/>
[21] GeeksforGeeks. 2022. *Iterative Merge Sort - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/iterative-merge-sort/>
[22] GeeksforGeeks. 2022. *Counting Sort - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/counting-sort/>
[23]Runestone.academy. 2022. *6.8. The Selection Sort — Problem Solving with Algorithms and Data Structures 3rd edition*. [online] Available at: <https://runestone.academy/ns/books/published/pythonds3/SortSearch/TheSelectionSort.html>
[24] Runestone.academy. 2022. *6.9. The Insertion Sort — Problem Solving with Algorithms and Data Structures 3rd edition*. [online] Available at: <https://runestone.academy/ns/books/published/pythonds3/SortSearch/TheInsertionSort.html>
[25]Pandas.pydata.org. 2022. *pandas.DataFrame.plot — pandas 0.23.1 documentation*. [online] Available at: <https://pandas.pydata.org/pandas-docs/version/0.23/generated/pandas.DataFrame.plot.html>
[26]Pandas.pydata.org. 2022. *pandas.DataFrame.plot.line — pandas 1.4.2 documentation*. [online] Available at: <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.plot.line.html>