46887 -- COMPUTATIONAL THINKING WITH ALGORITHMS

Project: Benchmarking Sorting Algorithms

Submitted by: John Dunne

Student No: G00273895

This report is submitted in a .zip folder accompanied by 6 .py files, 5 of which contain the implementations of my chosen sorting algorithms sourced from online resources with my own comments added. There is a main method called Benchmarking.py which I wrote to benchmark the run time of the five sorting algorithms. I have also included a csv file containing the results of the benchmarking process and a .png file containing the plot that I generated of the results.

Project Specification

Introduction (10%):

Introduce the concept of sorting and sorting algorithms, discuss the relevance of concepts such as complexity (time and space), performance, in-place sorting, stable sorting, comparator functions, comparison-based and non-comparison-based sorts.

Sorting Algorithms (5 x 5 = 25%):
Introduce each of your chosen algorithms in turn, discuss their space and time complexity, and explain how each algorithm works using your own diagrams and different example input instances.

The five sorting algorithms which you will implement, benchmark and discuss in this project must be chosen according to the following criteria:

1. A simple comparison-based sort (Bubble Sort, Selection Sort or Insertion Sort)

2. An efficient comparison-based sort (Merge Sort, Quicksort or Heap Sort)

3. A non-comparison sort (Counting Sort, Bucket Sort or Radix Sort)

4. Any other sorting algorithm of your choice

5. Any other sorting algorithm of your choice

Implementation & Benchmarking (25%):
This section will describe the process followed when implementing the application above, and will present the results of your benchmarking. Discuss how the measured performance of the algorithms differed – were the results similar to what you would expect, given the time complexity of each chosen algorithm? In this section you should use both a table and a graph to summarise the results obtained

Table of Contents

Section 1: Introduction

In the opening section of this project I will briefly introduce the concept of sorting and sorting algorithms. I will introduce the concepts of space and time complexity and use examples to explain their relevance when comparing the performance of algorithms. I will also introduce the concepts of analysing performance of sorting algorithms, in-place sorting, stable sorting and comparator functions. Finally, I will explain the difference between a comparison-based and non-comparison-based sort.

Section 1.1: Introduction to sorting

In the world of computer science sorting is deemed to be an important problem. According to the lecture series on Sorting we studied in this module - It has been claimed that up to 25% of all CPU cycles are spent sorting, which provides great incentive for further study and optimization. Sorting is the process of placing some sort of order on a collection. This could be sorting an array on integers in order of size or sorting a list of strings in alphabetical order.

I found the below explanation of sorting to be helpful in understand the concept:

"Sorting is arranging/re-arranging given data in an ordered sequence based on certain defined criteria. Simple example is to arrange boxes from smallest to largest size or from largest to smallest size. Only one caveat to the sorting is that, the criteria on which the sorting needs to be done should be available with all the given data and also must apply to all the given data equally. For example if we want to arrange given list of employees in a company based on their salaries, every employee must have salary information."

Source: ROBA's World - Sorting Algorithms in Software Development

A collection of items is deemed to be "sorted" if each item in the collection is less than or equal to its successor. To put this in more practical terms - the elements should be reorganised such that if an element at the index i in a collection has a value less than the element at index j in a collection then the element at index i must be less than the element at index j.

Sorting can also be a benefit when working to solve other problems. For example, if you wanted to search for an item in a list or find the median item in a list this would be much easier when working with a sorted list rather than an unsorted list.

Many different types of sorting algorithms have been developed over the years and I will now introduce the concept of sorting algorithms.

Section 1.2: Sorting algorithms

An algorithm is thought of as a set of rules to be followed in calculations or problem-solving operations. In the case of sorting, these rules would be focused on taking in an input and providing a sorted output.

There have been many different types of sorting algorithms developed and analysed over the years. Different sorting algorithms have varying levels of complexity and performance and there is no one sorting algorithm that is the most suitable one. One of the major factors involved in the performance of a sorting algorithm is the size of the input or the number of items to be processed. For a small input size a simple sorting algorithm may be adequate, whereas for a large input size a more complex sorting algorithm could be more suited to the job.

Sorting algorithms could be broken into two classes:

Comparison sorting algorithms which are a type of sorting algorithm which use comparison operators only to determine which of two elements should appear first in a sorted list.

Examples include:

- Bubble sort
- Insertion sort
- Selection sort
- Merge sort
- Quicksort
- Heapsort

Non- Comparison sorts on the other hand can make assumptions about the data.

Examples include:

- Counting sort
- Bucket sort
- Radix sort

I have included a section on Comparison sorting algorithms and Non-comparison sorting algorithms further down this document.

Section 1.3: Time and space complexity

Time complexity of an algorithm is essentially a measure of the time taken for the algorithm to run to completion. This is usually measured as a function of the length of the input.

Space complexity of an algorithm is a measure of the amount of space or memory taken by an algorithm to run as a function of the length of the input.

I have included an extract from a website which I think explains the difference between time and space complexity nicely:

"Algorithm analysis is concerned with comparing algorithms based upon the amount of computing resources that each algorithm uses. There are two different ways to look at this. One way is to consider the amount of space or memory an algorithm requires to solve the problem. As an alternative to space requirements, we can analyze and compare algorithms based on the amount of time they require to execute. This measure is sometimes referred to as the "execution time" or "running time" of the algorithm."

(Source: Brad Miller and David Ranum, Luther College, Problem Solving with Algorithms and Data Structures using Python [Online], Chapter 3.2 What is Algorithm Analysis?)

Time and space complexity of an algorithm depend on several factors including the operating system in use, the system processer, and the number of other operations being performed in the background. Algorithms are platform independent and so we should evaluate the complexity of algorithms in a way that is platform independent. Algorithms can be compared by evaluating their running time complexity on input of size n. Doing this will identify which algorithms should scale well to solve problems of a varying sizes. In other words, we would have a measure of the complexity of the algorithm as a function of the input size n.

The following classifications are generally used when describing the growth rate in complexity of algorithms by decreasing efficiency:

- Constant
- Logarithmic (log n)
- Sublinear
- Linear (n)
- n log(n)
- Quadratic (n^2)
- Cubic (n^3)
- Polynomial
- Exponential

A sorting algorithm that has a complexity that falls into ones of these classes would be expected to display the behaviour associated with this class when dealing with different input sizes n.

Another factor that would have an impact on the time taken for an algorithm to run is the characteristics of the data in the input. Taking a sorting algorithm for example, there could be a marked difference in performance when dealing with a nearly sorted input in comparison to dealing with a completely unsorted input. In this way, the performance of an algorithm can be broken further into three classes:

- **Worst case**

The Worst case occurs when an algorithm displays its worst possible runtime behaviour. In the case of a sorting algorithm example, the worst case could occur when the sorting algorithm has to deal with a completely unsorted input. Big O notation is generally used to describe the execution time of an algorithm in the worst-case scenario and can be used to describe both the time and space complexity of algorithms. When evaluating algorithms if they have a similar Big O notation then we could expect them to display similar time and space complexity.

- **Best case**

In the best case, an algorithm displays its best possible runtime behaviour. In the case of a sorting algorithm, this may occur when an almost fully sorted input is passed into the algorithm. Omega notation is used to describe the complexity of an algorithm in the best case. The best case usually doesn’t occur too often.

- **Average case**

The average case is the expected behaviour an algorithm should display when dealing with random input instances. In other words, the performance an average user could expect when running the algorithm.

Big O notation can be used to represent the relationship between the input size n and the number of steps an algorithm needs to take. The relationship is represented with a big O followed by the relationship between input size and number of steps needs for the algorithm to run.

Some examples below:

- **O(n)**

An algorithm that displays O(n) complexity will grow linearly in line with the input size. In other words, one operation per item in the input.

- **O(1)**

An algorithm that will always execute in the same time or space regardless of the size of the input data set would be described as having a complexity of O(1).

- **O(n^2)**

An algorithm which would check each element in the input twice would display quadratic complexity.

Section 1.4: Performance

Performance is a measure of how much system resources such as time, memory and disk space is used when a program runs. Performance is therefore different to complexity which measures how well the resource requirements of an algorithm scale to different input sizes. Algorithms are platform independent and can be run on different platforms and varying levels of performance would be recorded on different platforms. Therefore, performance is essentially platform dependent whereas complexity of an algorithm would remain the same regardless of the platform. Complexity affects performance but performance does not affect complexity.

Section 1.5: In-place sorting

In-place sorting is a classification of space complexity. Sorting algorithms have different memory requirements, which depend on how the specific algorithm work. A sorting algorithm is known as in-place if it requires only a fixed amount of working space, no matter the input size.

Examples of in-place sorting algorithms include:

- Bubble Sort
- Selection Sort
- Insertion Sort

I have sourced the below explanation on in-place sorting algorithms:

"In-place means that the algorithm does not use extra space for manipulating the input but may require a small though nonconstant extra space for its operation."

(Source: GeeksforGeeks, In-Place Algorithm)

This small non constant extra space mentioned in the above quotation refers to some small additional memory an algorithm may use, for example, when storing a value in a variable. An example of this would be in Bubble Sort when a temporary storage location is used to store the value on the left that is to be swapped. This is necessary to ensure the value on the right doesn't overwrite the value on the left when the swap is being made.

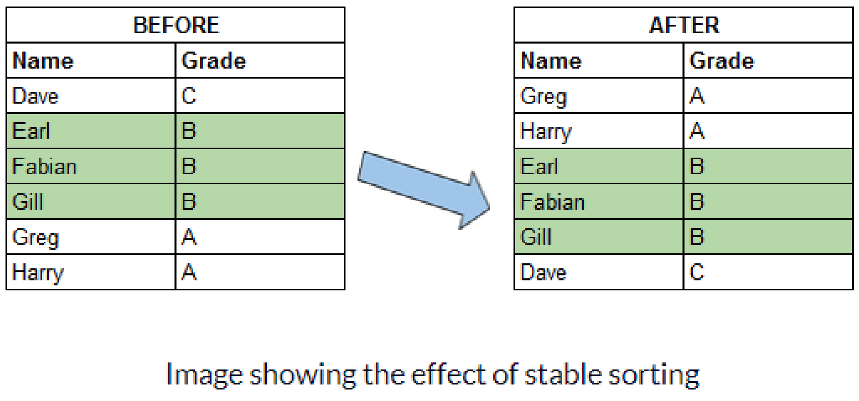
Algorithms that are not in-place may require additional working memory, the amount of this additional memory is usually related to the input size n.

Section 1.6: Stable sorting

Stable sorting refers to the ability of a sorting algorithm to preserve the order of elements from the input where the elements have an equal sort key. A sort key is a key used to determine the ordering of items in a collection, for example – sorting names in alphabetical order. Elements of equal value have the same sort key and so when they are output by the sorting algorithm, they should maintain their positions in relation to one another based on the input.

Sorting algorithms are deemed to be stable if they can guarantee this behaviour. Unstable sorting algorithms are algorithms that do not preserve the order

I have sourced the below graphics for a visual look at how a stable sorting algorithm preserves the order from the input collection for elements that have an equal sort key whereas an unstable algorithm cannot guarantee this will be done.

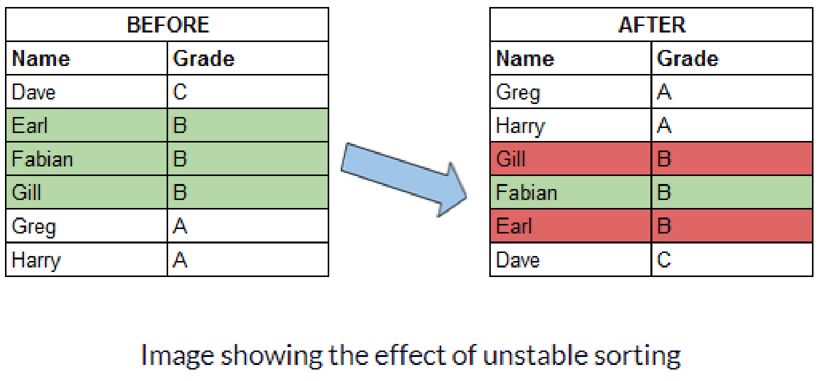


BEFORE	
Name	**Grade**
Dave	C
Earl	B
Fabian	B
Gill	B
Greg	A
Harry	A

AFTER	
Name	**Grade**
Greg	A
Harry	A
Earl	B
Fabian	B
Gill	B
Dave	C

Image showing the effect of stable sorting

Figure 1: Stable sorting: Image sourced from: Freecodecamp, Stability in sorting algorithms – A treatment of equality



BEFORE	
Name	**Grade**
Dave	C
Earl	B
Fabian	B
Gill	B
Greg	A
Harry	A

AFTER	
Name	**Grade**
Greg	A
Harry	A
Gill	B
Fabian	B
Earl	B
Dave	C

Image showing the effect of unstable sorting

Figure 2: Unstable sorting: Image sourced from: Freecodecamp, Stability in sorting algorithms – A treatment of equality

Examples of stable sorting algorithms are:

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

Section 1.7: Comparator functions

Comparator functions are used to compare objects in a collection. When sorting custom collections, a custom ordering scheme may have to be developed.

"Most sorting algorithms rely on the correct implementation of a comparison function that returns the ordering between two elements – that is, a function that takes two elements and returns whether the first is less than, equal to, or greater than the second"

(Source: Hackerrank, Sorting: Comparator)

An example implementation could be writing a function that takes two objects as inputs and returns a -1 if the first element is less than the second element, a zero if the first and the second element are equal or +1 if the first element is greater than the second element.

Section 1.8: Comparison-based sorting

A comparison-based sorting algorithm is an algorithm which only uses comparison operations to determine which of two elements should appear first in a collection. In other words, the only way to gain information about the positions of elements in the collection is by comparing a pair of elements at a time. No comparison-based sorting algorithm can perform better than n log n in the worst or average cases. The reason for this is that the algorithm works only by comparing one pair of elements at a time.

Some common comparison-based sorting algorithms are:

- Bubble sort
- Selection sort
- Insertion sort
- Merge sort
- Quicksort
- Heapsort

Section 1.9: Non-comparison-based sorting

Non-comparison-based sorting algorithms involves making some assumptions about the data in a collection. As mentioned in the previous section, comparison-based sorting algorithms don't make any assumptions about the data and use comparison operations to compare all elements in the collection against each other.

An example of non-comparison-based sorting algorithms:

- Count Sort
- Bucket Sort
- Radix Sort

Non-comparison-based sorting algorithms can therefore achieve better worst-case run times than comparison-based sorting algorithms.

Section 2: Sorting Algorithms

In this section I will introduce each of the five sorting algorithms that I will use in this benchmarking project. I will give some details about their space and time complexity in the best, worst and averages cases. I will give a brief explanation on how each algorithm works using diagrams and examples of different input instances. I will also include a screenshot of the code I have used in my implementation of the algorithms in this project along with some explanations of how the algorithm works.

Section 2.1: A simple comparison-based sort (Bubble Sort, Selection Sort or Insertion Sort)

The simple comparison-based sorting algorithm that I have chosen to focus on is **Bubble sort.**

Section 2.1.1: Bubble Sort Introduction

Bubble Sort is a simple comparison-based sorting algorithm that works by comparing a pair of elements at a time and exchanging elements that are out of order. Bubble Sort works by comparing the first two elements in the list. If the first element is larger than the second element, then the elements are swapped. If the first element is not larger than the second element, then they are left as they are.

This is where the name comes from, as the larger elements "bubble up" to the end of the list. The process is continued until the last pair of elements in the list is reached and the list is sorted.

"This simple sorting algorithm iterates over a list, comparing elements in pairs and swapping them until the larger elements "bubble up" to the end of the list, and the smaller elements stay at the "bottom"."

Source: Stackabuse – Sorting Algorithms in Python

Bubble sort is a very basic sorting algorithm in that it works through each pair of elements in a list one at a time.

Section 2.1.2: Bubble Sort procedure

For an input size of n, we can say that there are n-1 pairs of items that need to be compared the first time the algorithm passes through the list. An example would be for an input size of ten elements there would be 9 pairs of elements to be compared on the first pass.

In the below image the algorithm is making its first pass through an unsorted list. The first two elements on the left are compared and if the element on the left is larger than the element to the right an exchange is made. In the example below 54 is larger than the value 26 to its right and so it is exchanged or in other words swaps positions with 26. The second pair of elements to be examined are 54 on the left and 93 on the right in this case 54 is smaller than 93 and so the elements hold their position and no exchange is made.

The process is continued whereby each pair of elements are compared until the largest element in the list can no longer be compared to any other element, in other words, is positioned at the end of the list. When the first pass has been complete the largest element in the list is now in its correct place at the end of the list.

I have sourced an image and included it below which gives a graphical look at how Bubble sort would operate on the first pass through the list. At the end of the first pass the largest element has bubbled up to the end of the list.

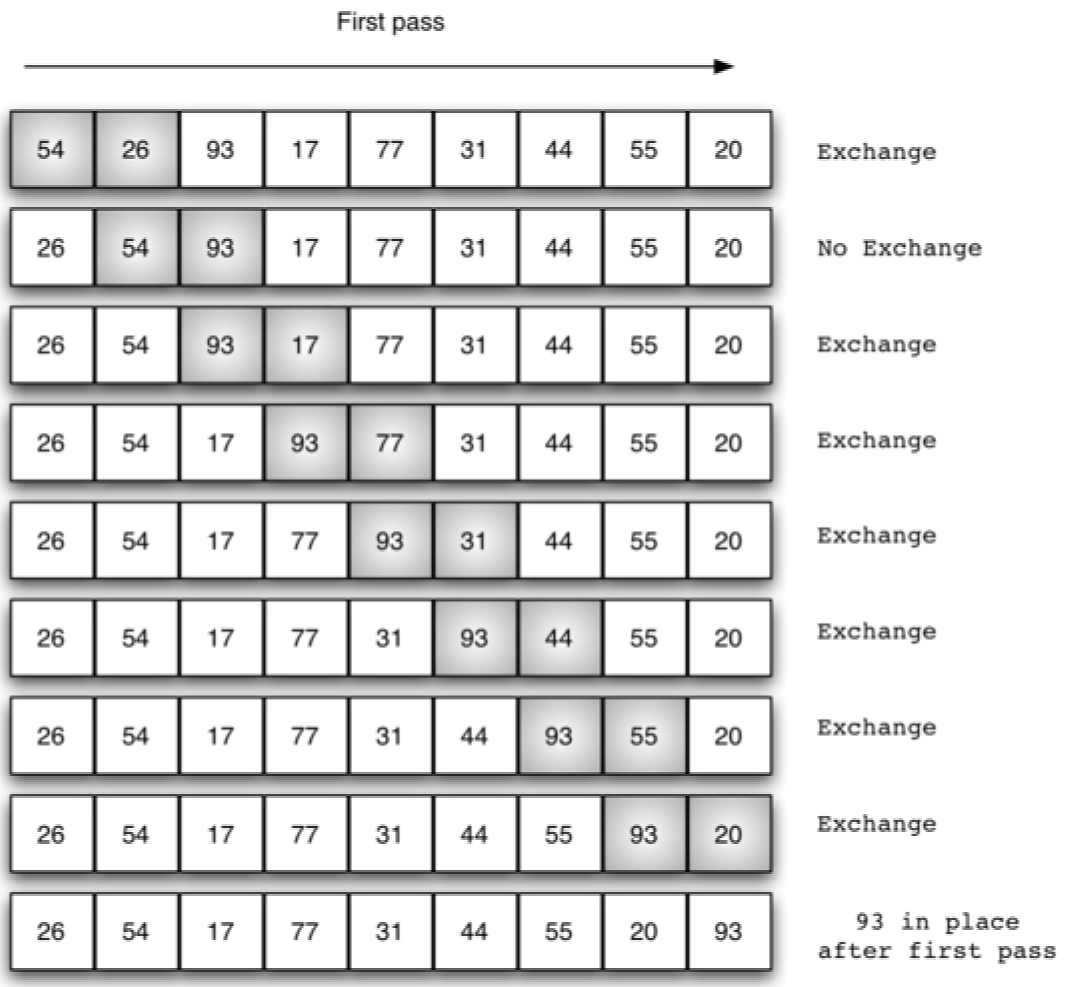

First pass
54 26 93 17 77 31 44 55 20 Exchange
26 54 93 17 77 31 44 55 20 No Exchange
26 54 93 17 77 31 44 55 20 Exchange
26 54 17 93 77 31 44 55 20 Exchange
26 54 17 77 93 31 44 55 20 Exchange
26 54 17 77 31 93 44 55 20 Exchange
26 54 17 77 31 44 93 55 20 Exchange
26 54 17 77 31 44 55 93 20 Exchange
26 54 17 77 31 44 55 20 93 93 in place after first pass

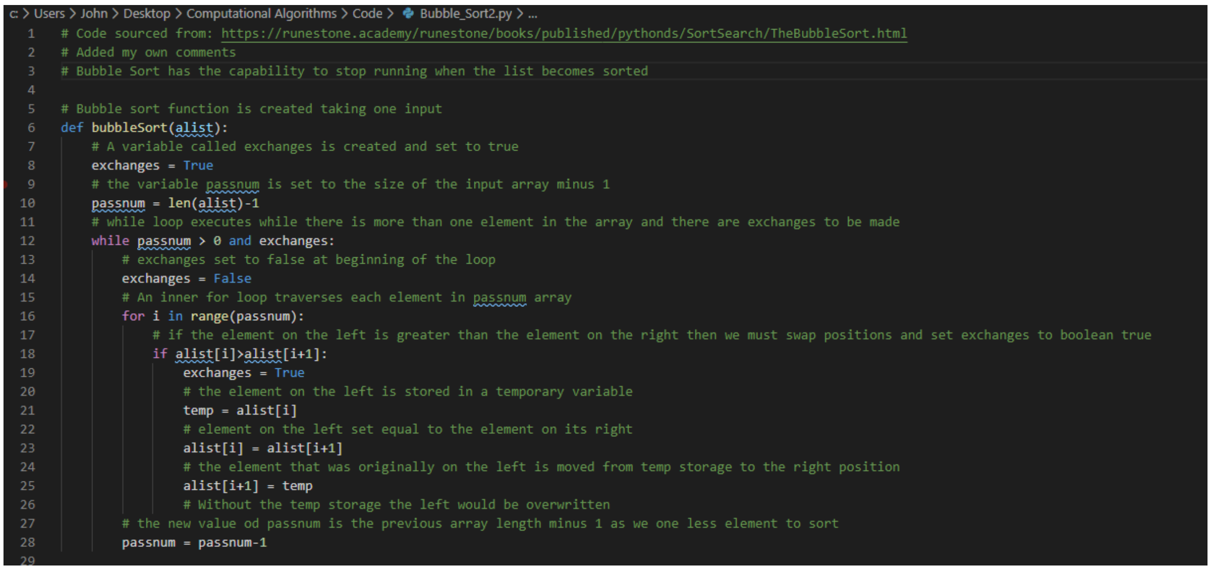
Figure 3: Bubble Sort: Source: Brad Miller and David Ranum, Luther College, Problem Solving with Algorithms and Data Structures using Python [Online], Chapter 6.7 Bubble Sort.

The procedure is repeated n-1 times until all elements are in their correct positions and the list is sorted.

Section 2.1.3: Bubble Sort implementation in Python

The implementation of Bubble Sort algorithm I have sourced includes a modification to the basic Bubble Sort algorithm that allows the algorithm to recognise when a list is fully sorted and to stop early in this case. This is one advantage that Bubble Sort has over other types of sorting algorithms.

I have included below a screenshot of the code used in my project including the comments I added to explain how the Bubble Sorting algorithm works:


c: > Users > John > Desktop > Computational Algorithms > Code > Bubble_Sort2.py > ...

```python
# Code sourced from: https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheBubbleSort.html
# Added my own comments
# Bubble Sort has the capability to stop running when the list becomes sorted

# Bubble sort function is created taking one input
def bubbleSort(alist):
    # A variable called exchanges is created and set to true
    exchanges = True
    # the variable passnum is set to the size of the input array minus 1
    passnum = len(alist)-1
    # while loop executes while there is more than one element in the array and there are exchanges to be made
    while passnum > 0 and exchanges:
        # exchanges set to false at beginning of the loop
        exchanges = False
        # An inner for loop traverses each element in passnum array
        for i in range(passnum):
            # if the element on the left is greater than the element on the right then we must swap positions and set exchanges to boolean true
            if alist[i]>alist[i+1]:
                exchanges = True
                # the element on the left is stored in a temporary variable
                temp = alist[i]
                # element on the left set equal to the element on its right
                alist[i] = alist[i+1]
                # the element that was originally on the left is moved from temp storage to the right position
                alist[i+1] = temp
                # Without the temp storage the left would be overwritten
        # the new value od passnum is the previous array length minus 1 as we one less element to sort
        passnum = passnum-1
```

Figure 4: Screenshot of the code I sourced for Bubble Sort with my own comments added

In Python, in order to exchange the elements in the list we must store the element that will be exchanged in a temporary storage location to ensure it is not overwritten. This is achieved by storing the value in a variable.

The above code example is only one example implementation of the Bubble Sort algorithm and there are different implementations of the Bubble Sort algorithm. I will now proceed to talking about the space and time complexity of Bubble Sort algorithm.

Section 2.1.4: Bubble Sort space and time complexity

- Bubble sort is an in-place sorting algorithm which uses a constant amount of additional working space in addition to the memory required for the input.
- Bubble sort is generally is slow and impractical for most problems.
- Bubble sort works well in lists that are already partially sorted. In the worst case, the algorithm would have to swap every element in the list.

*"Therefore, if we have n elements in our list, we would have n iterations per item - thus Bubble Sort's time complexity is **O(n^2)**."*

Source: Stackabuse – Sorting Algorithms in Python

- In the worst-case Bubble Sort has a Big O notation of O(n^2) and in the best case the algorithm can achieve a linear running time of n.
- Bubble sort is a stable sorting algorithm in that is preserves the order of elements from the input that have equal sort keys.
- I performed some tests running Bubble sort with varying input sizes of between 100 items and 10,000 items. The time complexity of Bubble sort seems to grow in a linear fashion on input sizes of less than 1,000 items.
- Larger input sizes seem to cause much slower run times. This is consistent with the resources I have read online, which suggest Bubble sort can perform well on small input instances but struggles with larger input sets.

Bubble Sort characteristics:

Worst case time complexity	**n²**
Best case time complexity	**n**
Average case time complexity	**n²**
Stable	**Yes**
In-Place	**Yes**
Space complexity	**1**

Section 2.2: An efficient comparison-based sort (Merge Sort, Quicksort or Heap Sort)

The efficient comparison-based sorting algorithm I have chosen to implement is **Merge Sort.**

Section 2.1.1: Merge Sort Introduction

Merge sort in a comparison-based sorting algorithm that uses a divide and conquer method to sort elements in a collection. Merge sort was proposed by John Von Neumann in 1945 and remains an important sorting algorithm today. Merge sort is a stable sorting algorithm that preserves the order of equal elements from the unsorted collection. The Merge Sort algorithm works by breaking the input list into equal or close to equal sub-lists repeatedly until there are sub-lists containing just 1 element each left. These lists containing 1 element each are merged into sorted lists containing 2 elements each and so on until a final sorted list remains.

In Python Merge sort is usually implemented recursively to sort a collection. Recursion is a programming technique whereby the problem is broken into smaller instances. Recursive methods use a base case that can be solved without the use of recursion. A recursive function calls itself until this base case is true at which point the program stops running. Recursion can be used to achieve simpler and cleaner looking algorithms when compared to iterative solutions to a similar problem.

Section 2.1.2: Merge Sort procedure

Merge sort works by:

- Checking the size of the input. If the size of the input is 0 or 1 then the list is returned without any action (base case).
- Separating the collection into two lists of equal or almost equal halves.
- If the list contains an odd number of elements, then one half would be larger by 1 element.
- These sub lists are repeatedly divided until the algorithm ends up with lists containing just 1 element each.
- The lists containing 1 element each are then compared and sorted into lists containing 2 elements each.
- The lists containing 2 elements are compared and merged into lists containing 4 elements each and so on until there are 2 sorted lists.
- Merge the two sorted halves of the list back together into one sorted list.

In a sample input containing 8 elements the steps would be:

- Check the list contains more than 1 element. If the list only contained 1 element it would be sorted and no action would be taken as the base case would be true.
- The list contains 8 elements, break the list into two halves of 4 elements each.
- Further break the two lists of 4 elements each into two halves containing 2 elements each.

- Further break the lists into lists containing just 1 element each.
- Merge the sorted lists containing 2 elements each back together into an element containing 4 elements.
- Merge the sorted lists containing 4 elements each back together into an element containing the 8 elements.
- The final sorted list is output.

I have drawn a diagram to give a visual representation of how the lists are broken into halves until each list contains just one element:

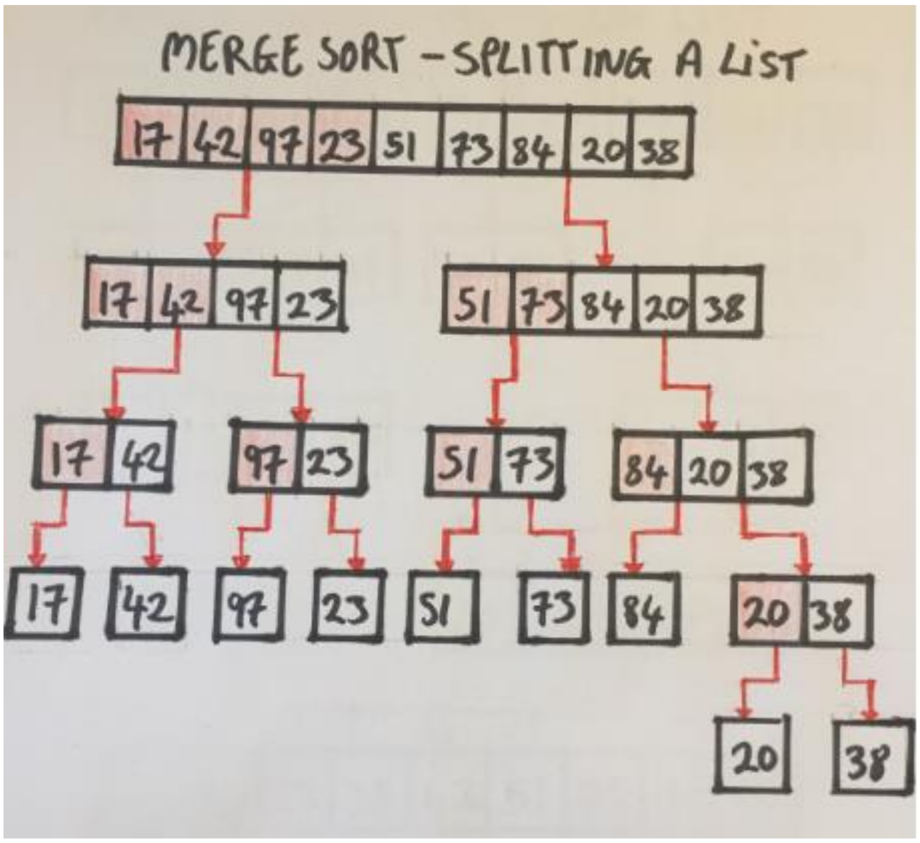


Figure 5: My own hand drawn diagram of Merge sort splitting a list.

I have drawn another diagram to give a visual representation of how the lists are then merged back together into a sorted list:

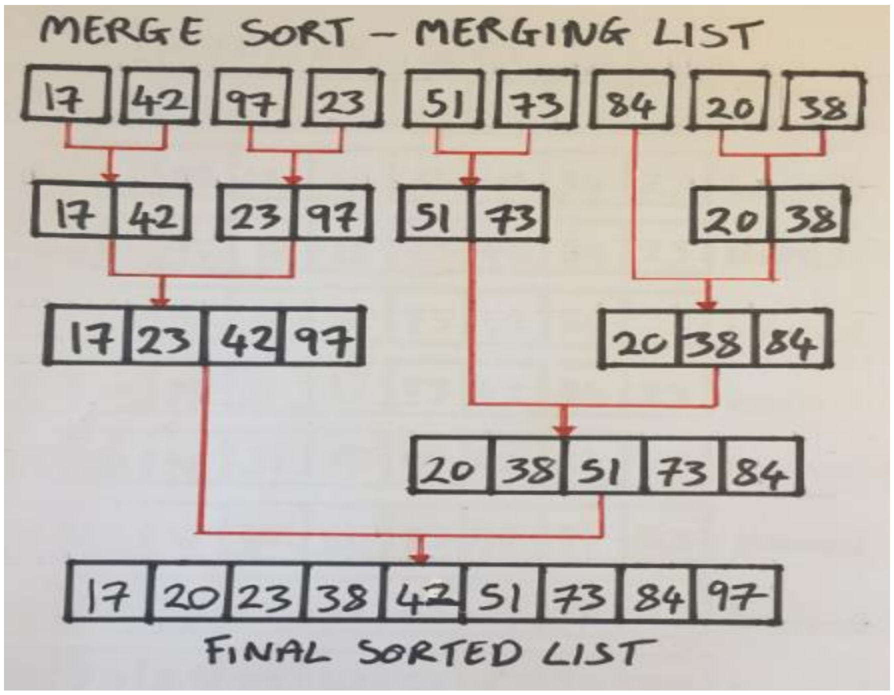
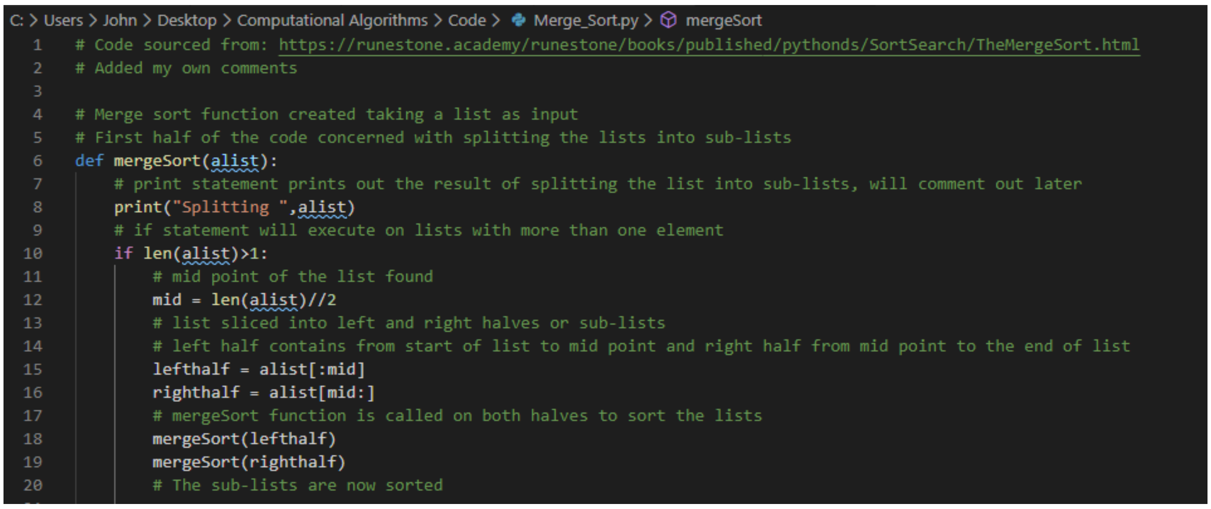


Figure 6: my own hand drawn diagram of Merge sort merging the sorted sub-lists into the final sorted list

Section 2.1.3: Merge Sort implementation in Python

The implementation of the Merge Sort algorithm that I have chosen to use in this project uses recursion. The base case question that is asked is if the length of the input list is not greater than 1 then the list is deemed to be already sorted and no action will be taken. If the length of the list is in fact greater than 1 then the Python slicing function is used to slice the list into 2 smaller sub-lists.

I have included a screenshot of the code below which I sourced online and added my own comments. The first half of the code is concerned with recursively splitting the list into sub-lists:

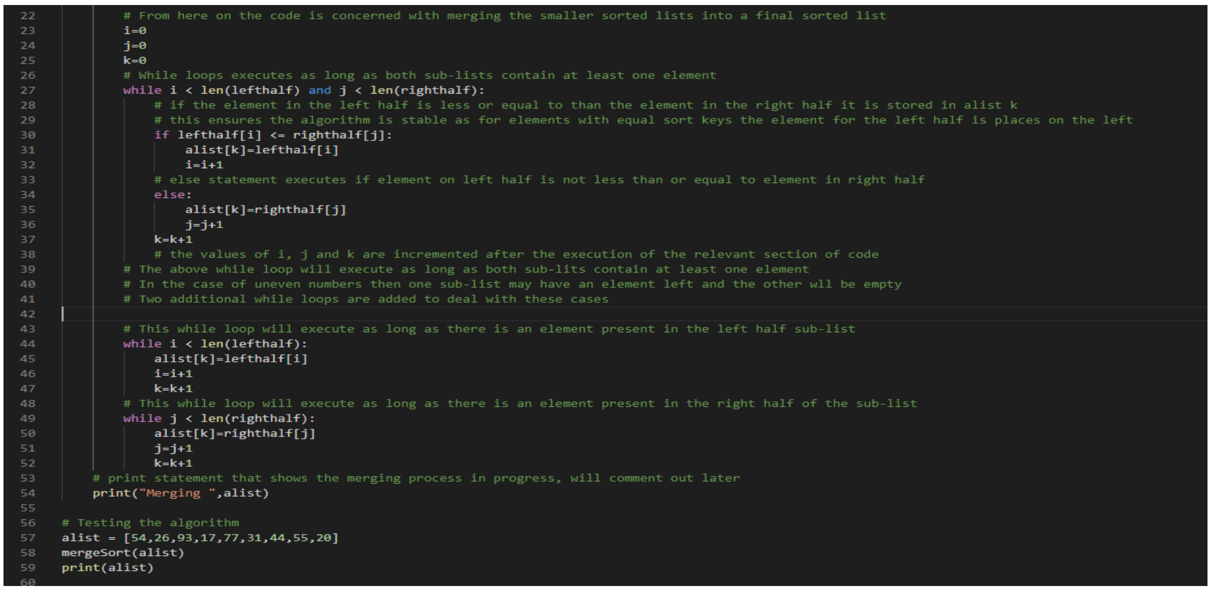

C: > Users > John > Desktop > Computational Algorithms > Code > Merge_Sort.py > mergeSort

```python
# Code sourced from: https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheMergeSort.html
# Added my own comments

# Merge sort function created taking a list as input
# First half of the code concerned with splitting the lists into sub-lists
def mergeSort(alist):
    # print statement prints out the result of splitting the list into sub-lists, will comment out later
    print("Splitting ",alist)
    # if statement will execute on lists with more than one element
    if len(alist)>1:
        # mid point of the list found
        mid = len(alist)//2
        # list sliced into left and right halves or sub-lists
        # left half contains from start of list to mid point and right half from mid point to the end of list
        lefthalf = alist[:mid]
        righthalf = alist[mid:]
        # mergeSort function is called on both halves to sort the lists
        mergeSort(lefthalf)
        mergeSort(righthalf)
        # The sub-lists are now sorted
```

Figure 7: Screenshot of the first half of the Merge sort code I sourced with my own comments added.

The second part of the code is concerned with merging the sorted sub-lists back together. The merge operation works by repeatedly taking the smallest items from the sorted sub-lists and placing them back in the sorted list. There is a provision in the code that ensures where elements with equal sort keys are met in both sub-lists the element from the left sub-list will be placed on the left of the element with the same sort key value from the right sub-list in the final sorted list. This is vital in ensuring the sorting algorithm is stable. In other words, it maintains the order of items that have equal value from the input.



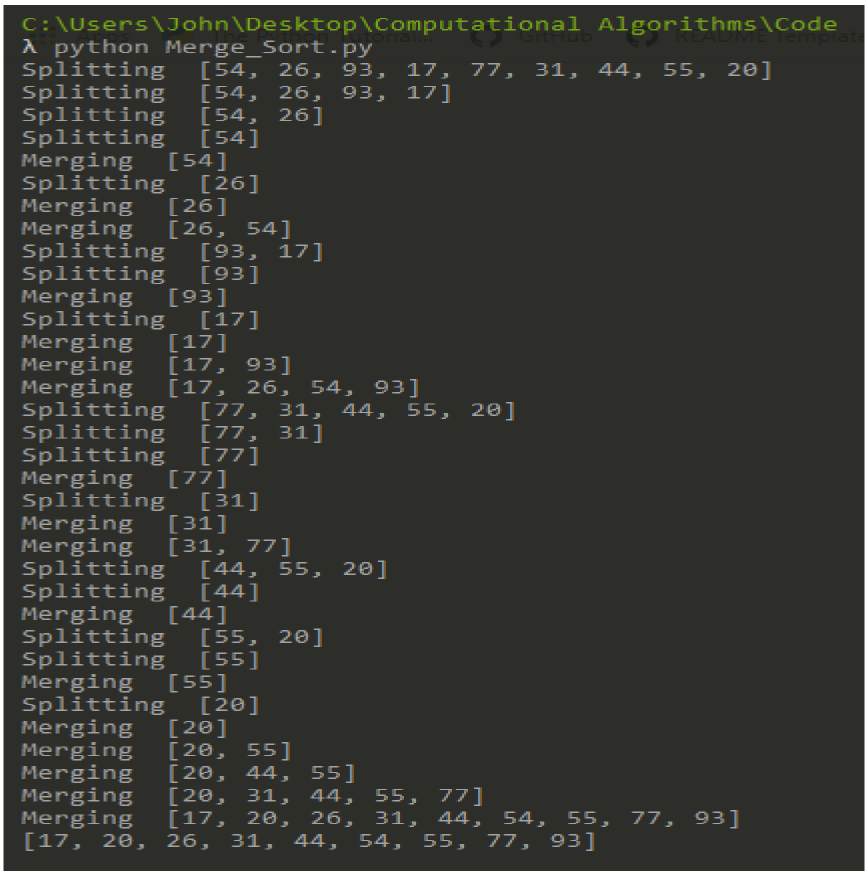
```python
        # From here on the code is concerned with merging the smaller sorted lists into a final sorted list
        i=0
        j=0
        k=0
        # While loops executes as long as both sub-lists contain at least one element
        while i < len(lefthalf) and j < len(righthalf):
            # if the element in the left half is less or equal to than the element in the right half it is stored in alist k
            # this ensures the algorithm is stable as for elements with equal sort keys the element for the left half is places on the left
            if lefthalf[i] <= righthalf[j]:
                alist[k]=lefthalf[i]
                i=i+1
            # else statement executes if element on left half is not less than or equal to element in right half
            else:
                alist[k]=righthalf[j]
                j=j+1
            k=k+1
            # the values of i, j and k are incremented after the execution of the relevant section of code
        # The above while loop will execute as long as both sub-lits contain at least one element
        # In the case of uneven numbers then one sub-list may have an element left and the other wll be empty
        # Two additional while loops are added to deal with these cases

        # This while loop will execute as long as there is an element present in the left half sub-list
        while i < len(lefthalf):
            alist[k]=lefthalf[i]
            i=i+1
            k=k+1
        # This while loop will execute as long as there is an element present in the right half of the sub-list
        while j < len(righthalf):
            alist[k]=righthalf[j]
            j=j+1
            k=k+1
    # print statement that shows the merging process in progress, will comment out later
    print("Merging ",alist)

# Testing the algorithm
alist = [54,26,93,17,77,31,44,55,20]
mergeSort(alist)
print(alist)
```

Figure 8: Screenshot of second half of the Merge sort code I sourced with my own comments added.

I have included below a screenshot from the command line showing a test run of the sorting algorithm which has printed out the splitting and merging process to give a visual representation of how the code works:



```
C:\Users\John\Desktop\Computational Algorithms\Code
λ python Merge_Sort.py
Splitting  [54, 26, 93, 17, 77, 31, 44, 55, 20]
Splitting  [54, 26, 93, 17]
Splitting  [54, 26]
Splitting  [54]
Merging  [54]
Splitting  [26]
Merging  [26]
Merging  [26, 54]
Splitting  [93, 17]
Splitting  [93]
Merging  [93]
Splitting  [17]
Merging  [17]
Merging  [17, 93]
Merging  [17, 26, 54, 93]
Splitting  [77, 31, 44, 55, 20]
Splitting  [77, 31]
Splitting  [77]
Merging  [77]
Splitting  [31]
Merging  [31]
Merging  [31, 77]
Splitting  [44, 55, 20]
Splitting  [44]
Merging  [44]
Splitting  [55, 20]
Splitting  [55]
Merging  [55]
Splitting  [20]
Merging  [20]
Merging  [20, 55]
Merging  [20, 44, 55]
Merging  [20, 31, 44, 55, 77]
Merging  [17, 20, 26, 31, 44, 54, 55, 77, 93]
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

Figure 9: Screenshot of Merge Sort splitting and merging a list taken from command line.

Section 2.1.4: Merge sort space and time complexity

- When analysing the time complexity of the Merge Sort algorithm it must be considered there are two processes within the algorithm – splitting the list into sub-lists and merging the sorted sub-lists into the final sorted list.
- The merging of the elements from the sub-lists into the sorted list will require all elements from the input n to be handled which means the program would require n executions to complete. This means the algorithm cannot achieve better than linear running time in the best case as it much handle all the elements of the input data set.
- In terms of space complexity, the Merge Sort algorithm requires additional space to store the sub-lists as they are extracted from the input list. This may cause problems when dealing with very large input instances. As I mentioned previously the algorithm is a stable sorting algorithm.
- I performed some tests running Merge sort on different sizes of input ranging from 100 to 10,000 and the algorithm seems to perform well on both small and large input instances. The time complexity appears to grow linearly along with the input size.

Merge Sort characteristics:

Worst case time complexity	**n log n**
Best case time complexity	**n log n**
Average case time complexity	**n log n**
Stable	**Yes**
Space complexity	**O(n)**

Section 2.3: A non-comparison sort (Counting Sort, Bucket Sort or Radix Sort)

The non-comparison-based sorting algorithm I have chosen to implement in this project is **Counting Sort.**

Section 2.3.1: Counting Sort Introduction

Counting sort was first proposed by Harold H. Seward in 1954. Counting sort can achieve running times of close to linear time. This can be achieved by making several assumptions about the data in the input instance. Counting sort works by sorting a collection according to a set of keys. Counting sort is a non-comparison-based sorting algorithm in that it doesn't compare the elements in a collection against each other. Counting sort is commonly used as a subroutine in another non-comparison-based sorting algorithm – Radix sort.

Section 2.3.2: Counting Sort procedure

Counting sort takes in an input collection and sorts it by using keys which are non-negative integers. It counts the number of occurrences of each key value in the input and stores it in an array. Counting sort then creates an output array based on the number of occurrences of each key. Important to note that in the case of multiple key values that have the same value, counting sort maintains the order from the input data which makes it a stable sorting algorithm.

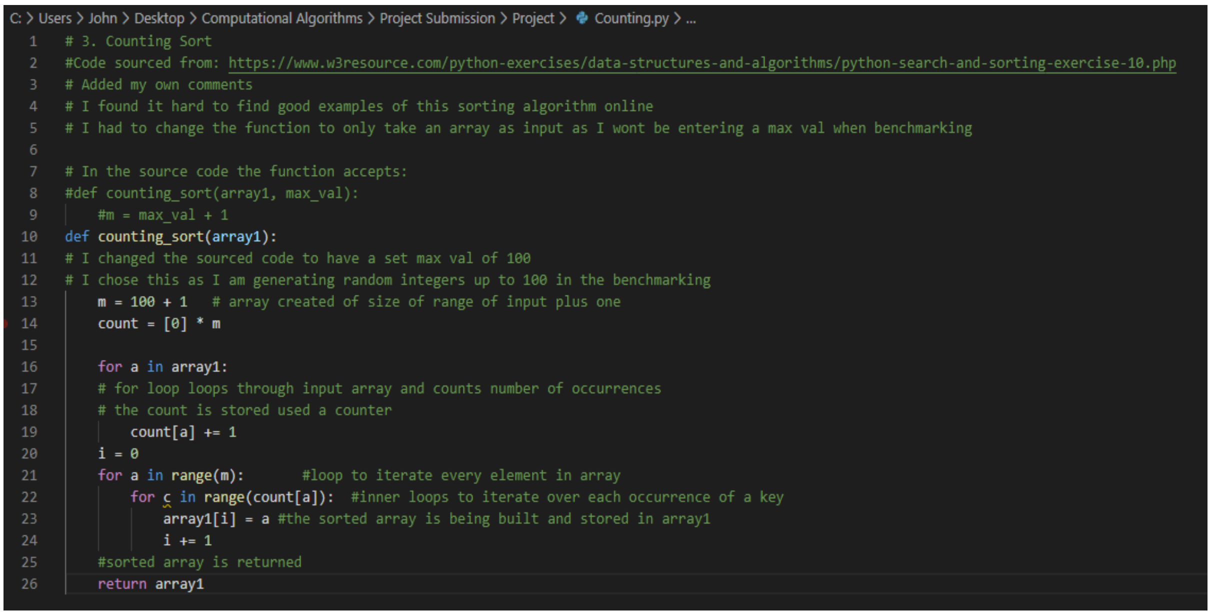
Counting sort procedure:

- Finding the key range k of the collection. In other words, find the maximum element in the input instance.

- Creates an array of size k.
- This array is then used to count the occurrence of each key value in the input instance and the number of occurrences are stored in this array.
- A result array of input size is created which will be used to store the sorted collection.
- Counting sort iterates through the input and records the number of occurrences of each key value.
- The output is constructed using the frequencies of occurrence of key values.
- Ordering is maintained from the input in the case of equal key values.

Section 2.3.3: Counting Sort implementation in Python

I have included a screenshot below of the Python implementation of Counting sort that I have sourced to use in this project including my own comments:



```
C: > Users > John > Desktop > Computational Algorithms > Project Submission > Project > Counting.py > ...
# 3. Counting Sort
#Code sourced from: https://www.w3resource.com/python-exercises/data-structures-and-algorithms/python-search-and-sorting-exercise-10.php
# Added my own comments
# I found it hard to find good examples of this sorting algorithm online
# I had to change the function to only take an array as input as I wont be entering a max val when benchmarking

# In the source code the function accepts:
#def counting_sort(array1, max_val):
    #m = max_val + 1
def counting_sort(array1):
# I changed the sourced code to have a set max val of 100
# I chose this as I am generating random integers up to 100 in the benchmarking
    m = 100 + 1   # array created of size of range of input plus one
    count = [0] * m

    for a in array1:
    # for loop loops through input array and counts number of occurrences
    # the count is stored used a counter
        count[a] += 1
    i = 0
    for a in range(m):       #loop to iterate every element in array
        for c in range(count[a]):  #inner loops to iterate over each occurrence of a key
            array1[i] = a #the sorted array is being built and stored in array1
            i += 1
    #sorted array is returned
    return array1
```

Figure 10: Screenshot of the Counting sort algorithm I sourced with my own comments added.

Section 2.3.4: Counting Sort space and time complexity

- The best, worst and average time complexity of Counting sort is n + k where n is the input size and k is the range of items we could have for each item in the input.
- Space complexity of Counting sort algorithm is also n + k as the algorithm must create an array of n size to hold the sorted output and an array of k size to check the occurrence of each key value.
- Counting sort is a stable sorting algorithm, any items that have the same key values will maintain their relative positions in output from the input instance.
- I did some tests running Counting sort over different input sizes and the algorithm seems to perform well on small and large input sizes. The input sizes I ran tests on ranged from 100 items to 10,000 items and the time of execution seemed to be consistent across all input sizes and displays a linear growth pattern.

Counting sort characteristics:

Worst case time complexity	**n + k**
Best case time complexity	**n + k**
Average case time complexity	**n + k**
Stable	**Yes**
Space complexity	**n + k**

Section 2.4: Any other sorting algorithm of choice

The sorting algorithm I have chosen to use here is **Insertion Sort.**

Section 2.4.1: Insertion Sort Introduction

Insertion sort is a simple in-place comparison-based sorting algorithm. Insertion sort works by maintaining a sorted sub-list at the lower end of the collection. Starting at the first element in an array and working through one element at a time, it inserts each element back into a sorted collection that it maintains throughout. Each time insertion sort deals with an element the sorted sub-list grows by one and the unsorted collection decreases in size by one. The procedure continues until all elements in the collection are sorted.

Section 2.4.2: Insertion Sort procedure

Insertion sort assumes a sorted collection of one element, that being the first element in the collection. Insertion sort then compares the second element with the first element and if the second element is less than the first element it moves the second element to the left and it forms the first element in the new sorted sub-list that has now grown in size from one to two. Insertion sort then moves on and compares the second and third elements in the collection and again if the element in the third position is less than the element in the second position they will be swapped. The procedure continues whereby at each iteration one element is compared to the elements in the sorted sub-list and inserted correctly into position in the sorted list.

I have drawn a diagram to give a graphical view of how insertion sort deals with an unsorted collection one element at a time inserting the element back into the sorted collection:

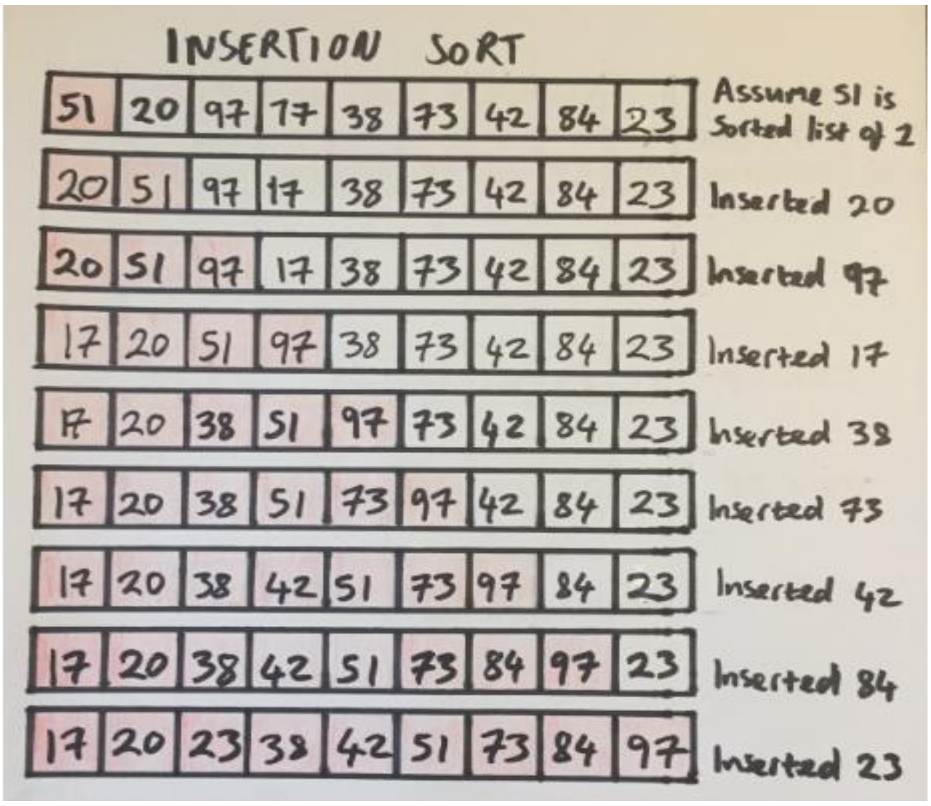


Figure 11: my own hand drawn diagram showing insertion sort in operation.

I have drawn a diagram below that shows how an element is inserted back into the sorted sub-list. In this example instance this is the fifth pass of the algorithm through the unsorted collection and there now exists a sorted sub-list of five elements on the left. The two elements being compared at this pass are 51 on the left and 23 on the right. 23 is less than 51 and so must be inserted back into the sorted sub-list. 23 is inserted to the left of 51 and is then compared to the other elements in the sorted sub-list in turn and inserted back to the left of any elements that are greater than 23. At the end 23 is now in its correct position in the sorted sub-list and the size of the sorted sub-list has grown to six elements.

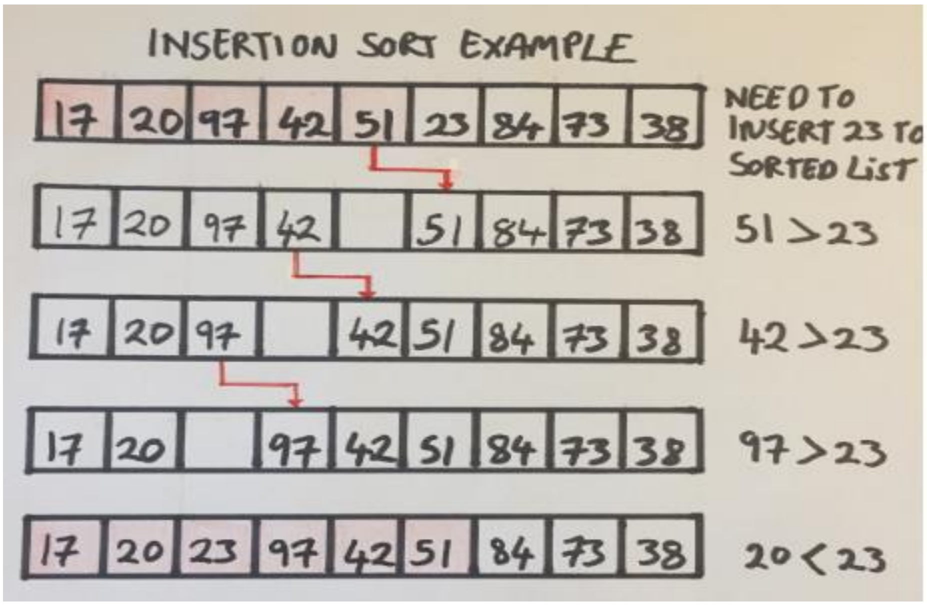
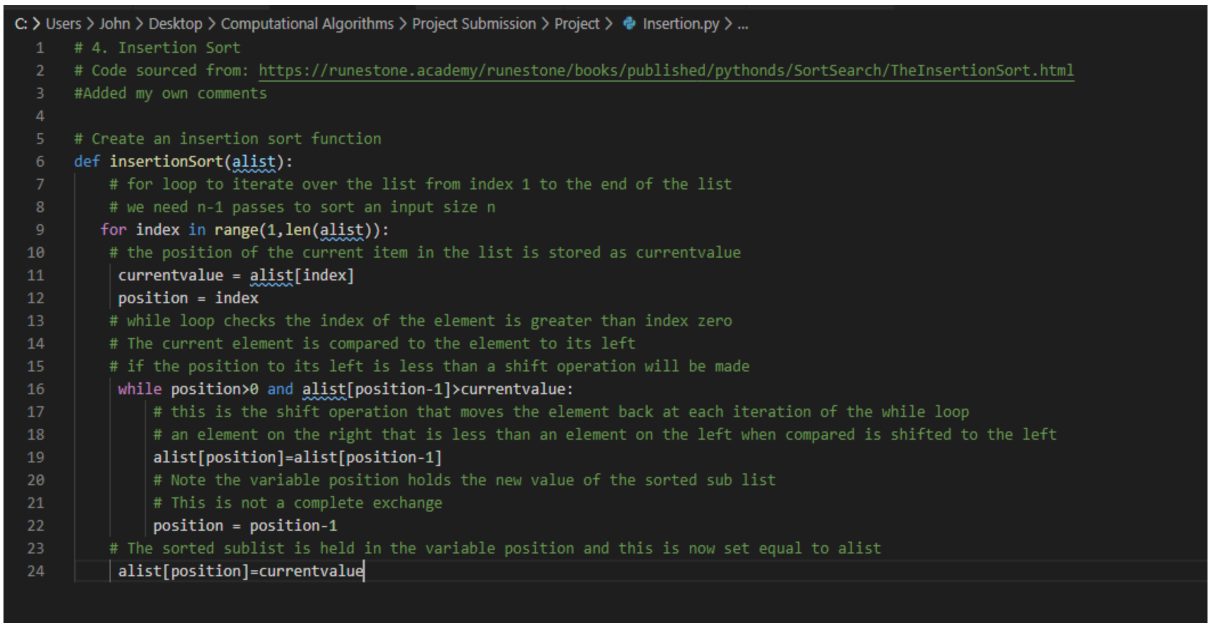


Figure 12: my own hand drawn diagram show Insertion sort on the 5th pass through the input.

Section 2.4.3: Insertion Sort implementation in Python

In order to sort a collection of size n, Insertion sort would have to make n-1 passes through the collection. For example, in a sample input size of 100 items, Insertion sort would iterate through the list 99 times to compare all the elements and produce the final sorted output. In Python, a while loop can be used to iterate over the list of while the condition that the current element has an index greater than zero remains true. Elements are compared to the element on their right at every iteration and if the element is found to have a greater value than the element on the right then a shift operation is made to shift the positions of the elements in the sorted part of the collection.

I have included a screenshot below of the implementation of insertion sort that I have chosen to use in this project which includes my own comments:



C: > Users > John > Desktop > Computational Algorithms > Project Submission > Project > Insertion.py > ...

```python
# 4. Insertion Sort
# Code sourced from: https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheInsertionSort.html
#Added my own comments

# Create an insertion sort function
def insertionSort(alist):
    # for loop to iterate over the list from index 1 to the end of the list
    # we need n-1 passes to sort an input size n
   for index in range(1,len(alist)):
    # the position of the current item in the list is stored as currentvalue
     currentvalue = alist[index]
     position = index
    # while loop checks the index of the element is greater than index zero
    # The current element is compared to the element to its left
    # if the position to its left is less than a shift operation will be made
     while position>0 and alist[position-1]>currentvalue:
         # this is the shift operation that moves the element back at each iteration of the while loop
         # an element on the right that is less than an element on the left when compared is shifted to the left
         alist[position]=alist[position-1]
         # Note the variable position holds the new value of the sorted sub list
         # This is not a complete exchange
         position = position-1
    # The sorted sublist is held in the variable position and this is now set equal to alist
     alist[position]=currentvalue
```

Figure 13: Screenshot of the implementation of Insertion sort that I sourced with my own comments added.

Section 2.4.4: Insertion Sort space and time complexity

- Insertion sort is a stable in-place comparison-based sorting algorithm.
- In the worst-case insertion sort has a time complexity of $O(n^2)$ and this could be expected when we pass a completely unsorted array of items to be the algorithm where each element would have to be dealt with.
- In the best case, a fully sorted collection would be passed in as input and the algorithm could run in n time where n is the size of the input.
- The time complexity of insertion sort is therefore dependent on the number of inversions in the input instance.
- Insertion sort is most effective when dealing with small input instances and in input instances which are mostly already sorted.
- I performed some tests and Insertion sort seems to perform well on small input sizes up to approximately 1,000 items but has a longer running time when dealing with larger input instances.
- Insertion sort has a constant space complexity using a fixed amount of additional space.
- One point to note about the performance of insertion sort that I came across:

"One note about shifting versus exchanging is also important. In general, a shift operation requires approximately a third of the processing work of an exchange since only one assignment is performed. In benchmark studies, insertion sort will show very good performance."

(Source: Brad Miller and David Ranum, Luther College, Problem Solving with Algorithms and Data Structures using Python, Chapter 6.9 The Insertion Sort)

Insertion Sort characteristics:

Worst case time complexity	**n^2**
Best case time complexity	**n**
Average case time complexity	**n^2**
Stable	**Yes**
In-Place	**Yes**
Space complexity	**1**

Section 2.5: Any other sorting algorithm of choice

The algorithm that I have chosen to implement is **Selection Sort.**

Section 2.5.1: Selection Sort Introduction

Selection sort is a comparison-based in-place sorting algorithm. Selection sort works in a similar way to Bubble sort however tends to give better performance than Bubble sort. Selection sort only makes one exchange at each pass through the collection, whereas Bubble Sort can make multiple exchanges before an element is placed in its correct position. Even though Selection sort performs better than Bubble sort, most of the reading I have done suggests it is not very practical for dealing with large input instances. There are different implementations of Selection sort, some are unstable, and others have been modified to be stable.

Section 2.5.2: Selection Sort procedure

Selection sort divides a collection into two parts, a sorted part on the left and the unsorted on the right. At the beginning the sorted part on the left is empty, Selection sort then works on populating this sorted part of the collection with the sorted elements. Selection sort works by searching through the elements in the collection and finding the smallest element. This element is then moved to the index 0 in the collection and forms the start of the sorted sub-list. The next search is performed on elements between index 1 and the end of the collection. The smallest element is again found and moved to index 1. The sorted sub-list has now grown from 1 element to 2 elements. This I continued until the collection is fully sorted.

I have included an example below for illustration purposes:

An unsorted collection: [44, 18, 14, 52, 23]

- On the first iteration the element at index 0 is 44 and it is compared with the remaining elements in the collection. The element at index 2 is found to be the smallest in the collection and so is exchanged with 44.
- On the next iteration a sorted sub-list of one element exists at element 0 and so the remaining elements from index 1 to the end of the collection are searched for the smallest element.

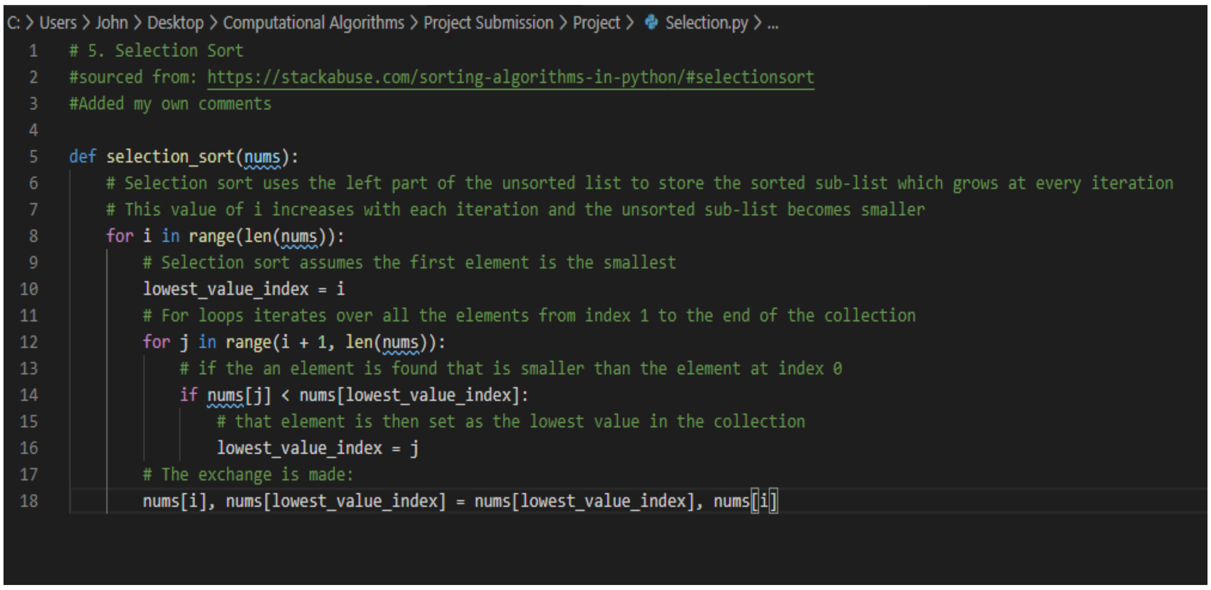
- The smallest element from the unsorted sub-list on the right is found to 18 and it is placed at index 1 in the sorted sub-list.
- The procedure is repeated n-1 times until the final sorted list is fully populated.

The result: [14, 18, 23, 44, 52]

As you can see, an exchange is only made once on each iteration where the smallest element is moved into the next available index in the sorted sub-list on the left. This in contrast to Bubble Sort where the element must be exchanged with elements that already are in their correct positions in the sorted sub-list which requires multiple exchanges.

Section 2.5.3: Selection Sort implementation in Python

I have included a screenshot of the implementation of Selection sort that I sourced online, with my own comments added:



C: > Users > John > Desktop > Computational Algorithms > Project Submission > Project > Selection.py > ...

```python
# 5. Selection Sort
#sourced from: https://stackabuse.com/sorting-algorithms-in-python/#selectionsort
#Added my own comments

def selection_sort(nums):
    # Selection sort uses the left part of the unsorted list to store the sorted sub-list which grows at every iteration
    # This value of i increases with each iteration and the unsorted sub-list becomes smaller
    for i in range(len(nums)):
        # Selection sort assumes the first element is the smallest
        lowest_value_index = i
        # For loops iterates over all the elements from index 1 to the end of the collection
        for j in range(i + 1, len(nums)):
            # if the an element is found that is smaller than the element at index 0
            if nums[j] < nums[lowest_value_index]:
                # that element is then set as the lowest value in the collection
                lowest_value_index = j
        # The exchange is made:
        nums[i], nums[lowest_value_index] = nums[lowest_value_index], nums[i]
```

Figure 14: Screenshot of the implementation of Selection sort that I sourced along with my own comments.

The code is quite easy to understand, and the procedure seems quite straightforward. The unsorted collection is repeatedly traversed and the smallest element each time is moved into position in the sorted sub-list on the left side. The process is repeated until the sorted list is fully populated.

Section 2.5.4: Selection Sort space and time complexity

- Selection sort requires n-1 passes to sort a collection of size n as it needs to look at each pair of elements in the collection.
- There are two loops in the implementation that I have used in this project. The outer loop iterates over a collection of size n, n times. The inner loop loops over the collection n-1 times, then n-2 times and so on until each element has been placed in the correct order.
- Time complexity if n^2 in worst, average and best cases.
- Space complexity is O(1) as the maximum amount of exchanges made by Selection sort is equal to the input size n.
- Stable sort in general is an unstable sorting algorithm but modifications can be made to make the algorithm stable.
- I performed some tests with varying input sizes and Selection sort seems to perform quiet well on input sizes up to 1,000 items. Larger input sizes result in a longer run time.

Selection Sort characteristics:

Worst case time complexity	**n^2**
Best case time complexity	**n^2**
Average case time complexity	**n^2**
Stable	**No**
In-Place	**Yes**
Space complexity	**O(1)**

Section 3 -Implementation & Benchmarking

In this section, I will describe the process I followed when implementing the benchmarking application. I will present the results of my benchmarking. I will discuss how the measured performance of the algorithms differed and discuss how the results compared to the expected performance based on the research I conducted about each algorithm.

Section 3.1: My Implementation of benchmarking process

First, I will describe the way that I decided to implement the benchmarking process. I decided to break the benchmarking process into several smaller tasks:

- Generate a random array of integers for a variety of input sizes.
- For each input size run each of the five sorting algorithms ten times and measure the time taken to execute and store the time taken to execute in a list.
- Calculate the average execution time of the ten runs for each input size.
- Create a pandas dataframe to display the average run time for each input size for each of the five sorting algorithms with the running time in milliseconds and the floating-point numbers rounded to 3 decimal places.
- Send the results to a csv file, which could be useful if the user wanted to further manipulate the data.
- Generate a plot to summarise the results.

Generating random arrays of integers:

I used the code that was provided in the project specification document to generate random arrays of integers. The Python random module is imported and within the module the randint function is used to generate random integers within a specified range. I decided to specify the range of random integers to be between 0 and 99. I made a list of the sample input sizes that I wanted to run my benchmarking on and the used a for loop to loop through this list of sample sizes and create arrays of that size which I stored in a variable called benchmark arrays. When passing these randomly generated arrays of input size as input to the sorting algorithms I passed a copy of the array as some sorting algorithms sort the array in place and so I wanted to ensure every time the sorting algorithm was run it was operating on an unsorted array.

Timing the algorithms:

As suggested in the project specification I used the built in Python time module to time the algorithms. The definition of the time.time() function within the time module is given on the Python time documentation page:

“time.time() → float

Return the time in seconds since the epoch as a floating point number.”

Sourced: Python time docs

In order to run each sorting algorithm ten times and to measure the time of execution each time, I used a nested for loop to loop through each input size ten times. I imported the time module and called time.time() right before the sorting algorithm was called and called time.time() again after the sorting algorithm had finished running on each iteration of the loop. I stored the difference between start time and finish time in a time elapsed variable. As mentioned earlier I passed a copy of the randomly generated array to the sorting algorithm on each run to ensure an unsorted array was being operated on. In the first tests that I conducted I didn’t do this, and I noticed that the first run of ten was taking longer for all the sorting algorithms than the remaining runs. I created a list for each sorting algorithm to hold the run times and appended the run times to this list.

Calculate the average time of ten runs:

I decided to use numpy.mean to calculate the mean running time of the list of times that I had stored. There are 1000 milliseconds in a second so I multiplied the average time by 1000 when calculated as the project specification requests the outputted average run time to be in milliseconds. I appended the average times to an empty list that I had created for each sorting algorithm outside of the for loops. This average list for each of the five sorting algorithms then contained the average time for each input size.

Outputting the average time to the user in the requested format:

I decided to use a pandas dataframe as the method of displaying the average time for each input size in milliseconds. I have worked with pandas in previous projects on this course and am familiar with dataframe objects and so it seems like a good choice.

Some advantages of storing data in a pandas dataframe:

- Data can be sent to a csv file from the dataframe.
- Built in pandas analysis tools can be used to return statistics about the data contained in the dataframe.
- Data visualization tools can be used to return visual representations of the data.

I created a pandas dataframe and added the input size and the average run times for each input size as columns in the dataframe. I set the index as input size in order to transpose the dataframe for a nicer display. I read about the transpose function online and I have included some references in the code and in the references section of this report. I also rounded the format of the pandas dataframe to 3 decimal places as this is also requested in the project specification. I did this using the built in dataframe.round function in pandas.

Sending the benchmarking results to a csv file:

Although not explicitly asked for in the project specification, I decided to use the built-in pandas function dataframe.to_csv to send the data from the pandas dataframe to a csv file. This is a handy option for storing results which the user could go on to later manipulate in Microsoft Excel for example.

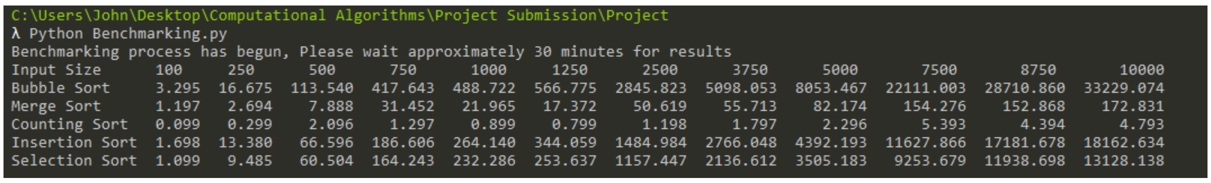
Plotting the results:

I imported matplotlib.pyplot in order to create a line plot of the results of my benchmarking. I was familiar with working with both pandas and matplotlib to plot data from some previous assignments and so I thought this would be a good tool to use. Some of the functionality I used included:

- Rcparams to customize the appearance of the plot.
- The legend function to add a legend to the plot.
- I added a title and x and y labels using the title, xlabel and ylabel functions.
- I added a grid to the plot.
- I added markers to the lines in order to have each input size marked on the line.

Section 3.2: Results of the benchmarking process

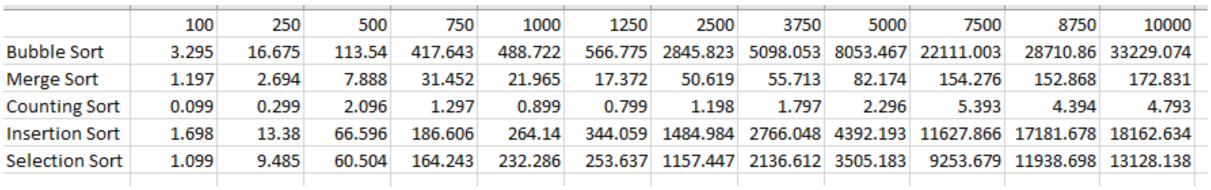
In this section I will describe the results of the benchmarking of the five sorting algorithms I chose in this project along with an analysis of the measured performance of each of the individual algorithms. I have taken a screenshot of the pandas dataframe generated on the command line after running my benchmarking code and I have included it below. The times shown are the average of ten runs of each sorting algorithm for each input size measured in milliseconds and rounded to 3 decimal places.



```
C:\Users\John\Desktop\Computational Algorithms\Project Submission\Project
λ Python Benchmarking.py
Benchmarking process has begun, Please wait approximately 30 minutes for results
Input Size      100     250     500     750     1000    1250     2500     3750     5000      7500      8750     10000
Bubble Sort     3.295  16.675  113.540  417.643  488.722  566.775  2845.823  5098.053  8053.467  22111.003  28710.860  33229.074
Merge Sort      1.197   2.694    7.888   31.452   21.965   17.372    50.619    55.713    82.174    154.276    152.868    172.831
Counting Sort   0.099   0.299    2.096    1.297    0.899    0.799     1.198     1.797     2.296      5.393      4.394      4.793
Insertion Sort  1.698  13.380   66.596  186.606  264.140  344.059  1484.984  2766.048  4392.193  11627.866  17181.678  18162.634
Selection Sort  1.099   9.485   60.504  164.243  232.286  253.637  1157.447  2136.612  3505.183   9253.679  11938.698  13128.138
```

Figure 15: Screenshot from command line of the results of benchmarking. The time average running time of ten runs for each input size is shown in milliseconds rounded to 3 decimal places.

I have also included below a screenshot of the results in csv format which I generated using the built-in function in pandas to send pandas dataframe to a csv file. I have included the csv file in my submission alongside this report.



	100	250	500	750	1000	1250	2500	3750	5000	7500	8750	10000
Bubble Sort	3.295	16.675	113.54	417.643	488.722	566.775	2845.823	5098.053	8053.467	22111.003	28710.86	33229.074
Merge Sort	1.197	2.694	7.888	31.452	21.965	17.372	50.619	55.713	82.174	154.276	152.868	172.831
Counting Sort	0.099	0.299	2.096	1.297	0.899	0.799	1.198	1.797	2.296	5.393	4.394	4.793
Insertion Sort	1.698	13.38	66.596	186.606	264.14	344.059	1484.984	2766.048	4392.193	11627.866	17181.678	18162.634
Selection Sort	1.099	9.485	60.504	164.243	232.286	253.637	1157.447	2136.612	3505.183	9253.679	11938.698	13128.138

Figure 16: Screenshot from csv file of the results of benchmarking. The time average running time of ten runs for each input size is shown in milliseconds rounded to 3 decimal places.

I have also included below a screenshot of the plot that I generated using matplotlib.pyplot and included it below. This plot has also been included in my submission alongside this report.

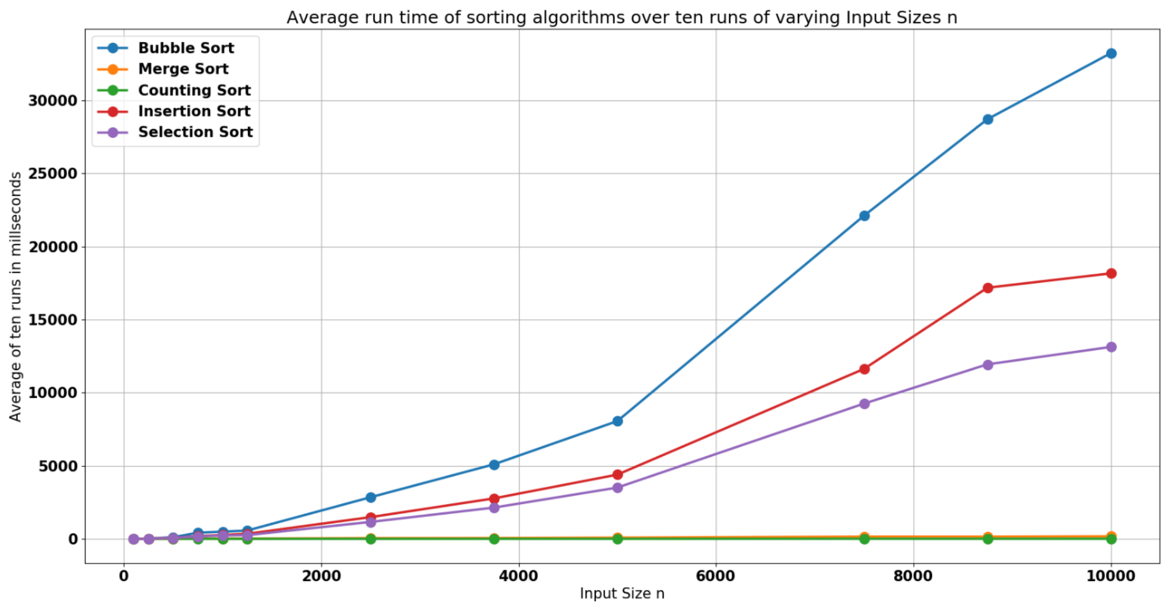


Figure 17: Screenshot of the plot generated showing the average running time for each of the five sorting algorithms for each input size.

I have given my interpretation of the results of the benchmarking process below:

Bubble Sort

- Bubble sort had the longest running time of all the 5 sorting algorithms across all input sizes in this benchmarking study.
- On input sizes of less than 1,000 Bubble sort performed reasonably well and displayed a linear order of growth.
- For larger input instances Bubble sort performed the worst of all 5 sorting algorithms.
- This is consistent with the research that I conducted with most sources suggesting Bubble sort is unsuitable for dealing with larger input instances.
- However, there may be a case to be made for using Bubble sort on small input sizes as the code is easy to understand and the algorithm is effective working with small input sizes.
- The results are what I expected based on the research that I completed about the time complexity of Bubble sort in section 2 of this report.

Merge Sort

- Merge sort was the best performing comparison-based sort in this benchmarking study.
- The average execution time grew linearly in line with the increased input size and overall the algorithm had a consistent performance across all input instances.
- My research indicated Merge sort displays n log n performance in the best, worst and average cases and this seems to be reflected in the results of my benchmarking study.
- I found the results for Merge sort to be very impressive and I didn't expect it perform as good as it is a comparison-based sorting algorithm.
- The execution times of Merge sort in this benchmarking study would lead me to recommend the algorithm for use on large input instances as it is reasonably easy to understand and implement.
- I find Merge sort easier to understand and implement than Counting sort and for this reason I would be more inclined to use Merge sot than Counting sort in the future.

- I would recommend Merge sort for use on large input instances.

Counting Sort

- Counting sort displayed the lowest average execution time of all 5 sorting algorithms benchmarked.
- This is consistent with the information I gathered in my research for section 2 of this report.
- Counting sort was the only non-comparison-based sorting algorithm that I used in this benchmarking study and I expected the execution time to be low for all input sizes.
- I found Counting sort to be harder to understand and implement than the comparison-based sorting algorithms in this benchmarking study but when I understood the logic of how the algorithm works, I could see that it could be a very effective sorting algorithm to use.
- I would recommend Counting sort for use on large input sizes.

Insertion Sort

- Insertion sort had the second longest execution times after Bubble sort of all the five sorting algorithms in this benchmark study.
- This didn't come as a surprise to me as the algorithm is a simple comparison-based sorting algorithm and operates in a similar way to Bubble sort.
- However, Insertion sort outperforms Bubble sort as it performs a shift operation instead of an exchange which is computationally less expensive which I highlighted in section 2 of this report.
- Insertion sort is said to display time complexity of n in the best case and n^2 in the worst and average cases. This is consistent with the result of this benchmarking study.
- Insertion sort displayed good performance when dealing with small input sizes of less than 500 items.
- Based on the research I conducted and the results of this benchmarking study I would not recommend Insertion sort for large input instances.

Selection Sort

- Selection sort performed reasonably well in this benchmarking study and outperformed the other two simple comparison-based sorting algorithms in the study.
- Selection sort outperformed Bubble sort with execution times of almost 100% less on some of the larger input sizes. As my research indicated, Selection sort performs well in benchmarking studies as the algorithm only makes one exchange per pass compared to Bubble sort which makes multiple exchanges per pass.
- Overall, the algorithm would not be suitable for use on large input sizes, but I would recommend it as suitable for input sizes of less than 1,000 items.
- My research indicated the algorithm displays time complexity of n^2 in the best, worst and average cases. The times measured in this benchmarking are consistent with this.

Section 4: Bibliography

In this section I have listed all references and resources that I used when researching and compiling this report. In section 3 I have listed the references I drew on when writing the code for the benchmarking element of the project, these references are also included in the comments in the Benchmarking.py file.

Section 1: Introduction

Section 1.1: Introduction to sorting

- GMIT Lecture slides – Sorting Algorithms Part 1 [Online] Available from: https://learnonline.gmit.ie/pluginfile.php/191466/mod_resource/content/0/07%20Sorting%20Algorithms%20Part%201.pdf
- Brad Miller and David Ranum, Luther College, Problem Solving with Algorithms and Data Structures using Python [Online] Available from: https://runestone.academy/runestone/books/published/pythonds/index.html
- Sorting Algorithms in Python: https://stackabuse.com/sorting-algorithms-in-python/ accessed 07/04/2020
- Blog Post – ROBA's World, Sorting Algorithms in Software Development [Online] Available from: https://www.robasworld.com/sorting-algorithms/

Section 1.2: Sorting algorithms

- Blog Post – ROBA's World, Sorting Algorithms in Software Development [Online] Available from: https://www.robasworld.com/sorting-algorithms/
- Sorting Algorithms in Python: https://stackabuse.com/sorting-algorithms-in-python/ accessed 07/04/2020
- Brad Miller and David Ranum, Luther College, Problem Solving with Algorithms and Data Structures using Python [Online], Chapter 6.6 Sorting, Available from: https://runestone.academy/runestone/books/published/pythonds/SortSearch/sorting.html
- Santiago Valdarrama, Real Python, Sorting Algorithms in Python [Online] Available from: https://realpython.com/sorting-algorithms-python/

Section 1.3: Time and space complexity

- Brad Miller and David Ranum, Luther College, Problem Solving with Algorithms and Data Structures using Python [Online], Chapter 3.2 What is Algorithm Analysis? Available from: https://runestone.academy/runestone/books/published/pythonds/AlgorithmAnalysis/WhatIsAlgorithmAnalysis.html
- Santiago Valdarrama, Real Python, Sorting Algorithms in Python [Online] Available from: https://realpython.com/sorting-algorithms-python/
- Akash Sharma, hackerearth.com, time and space complexity [Online], Available from: https://www.hackerearth.com/practice/basic-programming/complexity-analysis/time-and-space-complexity/tutorial/
- GeeksforGeeks, Understanding time complexity with simple examples, [Online], Available from: https://www.geeksforgeeks.org/understanding-time-complexity-simple-examples/
- GMIT Lecture slides, Analysing Algorithms Part 1 [Online], Available from: https://learnonline.gmit.ie/pluginfile.php/186006/mod_resource/content/0/03%20Analysing%20Algorithms%20Part%201.pdf

- GMIT Lecture slides, Analysing Algorithms Part 2 [Online], Available from: https://learnonline.gmit.ie/pluginfile.php/186006/mod_resource/content/0/03%20Analysing%20Algorithms%20Part%201.pdf
- Studytonight.com, Time complexity of algorithms, [Online], Available from: https://www.studytonight.com/data-structures/time-complexity-of-algorithms

Section 1.4: Performance

- GMIT Lecture slides – Sorting Algorithms Part 1 [Online] Available from: https://learnonline.gmit.ie/pluginfile.php/191466/mod_resource/content/0/07%20Sorting%20Algorithms%20Part%201.pdf

Section 1.5: In-place sorting

- GeeksforGeeks, In-Place Algorithm, [Online] Available from: https://www.geeksforgeeks.org/in-place-algorithm/
- GMIT Lecture slides, Analysing Algorithms Part 1 [Online], Available from: https://learnonline.gmit.ie/pluginfile.php/186006/mod_resource/content/0/03%20Analysing%20Algorithms%20Part%201.pdf
- Baeldung, How an in-place sorting algorithm works [Online], Available from: https://www.baeldung.com/java-in-place-sorting

Section 1.6: Stable sorting

- Stack overflow, What is stability in sorting and why is it important? [Online], Available from: https://stackoverflow.com/questions/1517793/what-is-stability-in-sorting-algorithms-and-why-is-it-important
- GeeksforGeeks, Stability in sorting algorithms [Online], Available from: https://www.geeksforgeeks.org/stability-in-sorting-algorithms/
- Freecodecamp, Stability in sorting algorithms – A treatment of equality [Online], Available from: https://www.freecodecamp.org/news/stability-in-sorting-algorithms-a-treatment-of-equality-fa3140a5a539/
- GMIT Lecture slides, Analysing Algorithms Part 1 [Online], Available from: https://learnonline.gmit.ie/pluginfile.php/186006/mod_resource/content/0/03%20Analysing%20Algorithms%20Part%201.pdf

Section 1.7: Comparator functions

- Rufflewind's Scratchpad, Sorting with a random comparator [Online], Available from: https://rufflewind.com/2016-09-16/random-comparator-sort
- GMIT Lecture slides, Analysing Algorithms Part 1 [Online], Available from: https://learnonline.gmit.ie/pluginfile.php/186006/mod_resource/content/0/03%20Analysing%20Algorithms%20Part%201.pdf
- Hackerrank, Sorting: Comparator [Online], Available from: https://www.hackerrank.com/challenges/ctci-comparator-sorting/problem
- Cs.furman,edu, Writing your own Comparators [Online], Available from: http://cs.furman.edu/~chealy/cs025/Comparators.pdf

Section 1.8: Comparison-based sorting

- GeeksforGeeks, Lower bound for comparison based sorting algorithms [Online] Available from: https://www.geeksforgeeks.org/lower-bound-on-comparison-based-sorting-algorithms/
- Cs.cmu.deu, Lecture 5 Comparison-based lower bounds for sorting [Online] Available from: https://www.cs.cmu.edu/~avrim/451f11/lectures/lect0913.pdf
- Quora, What is a comparison based sorting algorithm [Online] Available from: https://www.quora.com/What-is-a-comparison-based-sorting-algorithm
- GMIT Lecture slides – Sorting Algorithms Part 2 [Online] Available from: https://learnonline.gmit.ie/pluginfile.php/191471/mod_resource/content/0/08%20Sorting%20Algorithms%20Part%202.pdf

Section 1.9: Non-comparison-based sorting

- GMIT Lecture slides – Sorting Algorithms Part 3 [Online] Available from: https://learnonline.gmit.ie/pluginfile.php/191472/mod_resource/content/0/09%20Sorting%20Algorithms%20Part%203.pdf
- Cs.wisc.edu, Non-Comparison sorting algorithms [Online], Available from: http://pages.cs.wisc.edu/~paton/readings/Old/fall08/LINEAR-SORTS.html

Section 2: Sorting Algorithms

Section 2.1: Bubble Sort

- GMIT Lecture slides – Sorting Algorithms Part 2 [Online] Available from: https://learnonline.gmit.ie/pluginfile.php/191471/mod_resource/content/0/08%20Sorting%20Algorithms%20Part%202.pdf
- Python: BubbleSort sorting algorithm Youtube: https://www.youtube.com/watch?v=YHm_4bVOe1s
- Stackabuse, Sorting algorithms in Python [online] Available from: https://stackabuse.com/sorting-algorithms-in-python/
- Brad Miller and David Ranum, Luther College, Problem Solving with Algorithms and Data Structures using Python [Online], Chapter 6.7 Bubble Sort Available from: https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheBubbleSort.html
- Python Data Structures and Algorithms: Bubble sort [Online], Available from: https://www.w3resource.com/python-exercises/data-structures-and-algorithms/python-search-and-sorting-exercise-4.php

Section 2.2: Merge Sort

- Brad Miller and David Ranum, Luther College, Problem Solving with Algorithms and Data Structures using Python, Chapter 6.11 The Merge Sort [Online], Available from: https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheMergeSort.html
- GMIT Lecture slides – Sorting Algorithms Part 3 [Online] Available from: https://learnonline.gmit.ie/pluginfile.php/191472/mod_resource/content/0/09%20Sorting%20Algorithms%20Part%203.pdf
- Oliviera Popovic, Stack Abuse, Merge Sort in Python [Online] Available from: https://stackabuse.com/merge-sort-in-python/

- GeeksforGeeks, Python Program for Merge Sort [Online] Available from: https://www.geeksforgeeks.org/python-program-for-merge-sort/
- W3resource, Python Data Structures and Algorithms: Merge Sort [Online] Available from: https://www.w3resource.com/python-exercises/data-structures-and-algorithms/python-search-and-sorting-exercise-8.php
- Marcus Sanatan, Stack Abuse, Sorting Algorithms in Python [Online], Available from: https://stackabuse.com/sorting-algorithms-in-python/#mergesort

Section 2.3: Counting Sort

- GeeksforGeeks, Counting Sort [Online] Available from: https://www.geeksforgeeks.org/counting-sort/
- Tutorialspoint, Counting Sort [Online] Available from: https://www.tutorialspoint.com/Counting-Sort
- Hackerearth, Counting Sort [Online] Available from: https://www.hackerearth.com/practice/algorithms/sorting/counting-sort/tutorial/
- Wikipedia, Counting Sort [Online] Available from: https://en.wikipedia.org/wiki/Counting_sort
- Youtube, Learn counting sort algorithm in less than 6 minutes! [Online} Available from: https://www.youtube.com/watch?v=OKd534EWcdk
- GMIT Lecture slides – Sorting Algorithms Part 3 [Online] Available from: https://learnonline.gmit.ie/pluginfile.php/191472/mod_resource/content/0/09%20Sorting%20Algorithms%20Part%203.pdf

Section 2.4: Insertion Sort

- Tutorialspoint, Data Structure and Algorithms Insertion Sort [Online], Available from: https://www.tutorialspoint.com/data_structures_algorithms/insertion_sort_algorithm.htm
- GMIT Lecture slides – Sorting Algorithms Part 2 [Online] Available from: https://learnonline.gmit.ie/pluginfile.php/191471/mod_resource/content/0/08%20Sorting%20Algorithms%20Part%202.pdf
- GeeksforGeeks, Insertion Sort [Online}, Available from: https://www.geeksforgeeks.org/insertion-sort/
- Hackerearth, Insertion Sort [Online], Available from: https://www.hackerearth.com/practice/algorithms/sorting/insertion-sort/tutorial/
- Brad Miller and David Ranum, Luther College, Problem Solving with Algorithms and Data Structures using Python, Chapter 6.9 The Insertion Sort [Online], Available from: https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheInsertionSort.html

Section 2.5 Selection Sort

- Tutorialspoint, Data Structure and Algorithms Selection Sort [Online] Available from: https://www.tutorialspoint.com/data_structures_algorithms/selection_sort_algorithm.htm
- GMIT Lecture slides – Sorting Algorithms Part 2 [Online] Available from: https://learnonline.gmit.ie/pluginfile.php/191471/mod_resource/content/0/08%20Sorting%20Algorithms%20Part%202.pdf
- Geeksforgeeks, Selection Sort [Online] Available from: https://www.geeksforgeeks.org/selection-sort/

- Brad Miller and David Ranum, Luther College, Problem Solving with Algorithms and Data Structures using Python, Chapter 6.8 The Selection Sort [Online] Available from: https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheSelectionSort.html
- Marcus Sanatan, Stack Abuse, Sorting Algorithms in Python [Online] Available from: https://stackabuse.com/sorting-algorithms-in-python/#selectionsort

Section 3 -Implementation & Benchmarking

Section 3.1: My Implementation of benchmarking process

I have included below some of the resources I used in writing the code for the benchmarking part of this project. They are also included in the comments accompanying the code submitted alongside this report.

Generating random array of integers:

- Python random module [Online] Available from: https://docs.python.org/3/library/random.html

Timing the algorithms and appending the times to a list:

- Python time docs [Online] Available from: https://docs.python.org/3/library/time.html
- time.time() [Online] Available from: https://docs.python.org/3/library/time.html#time.time
- W3schools, Python list append() method: https://www.w3schools.com/python/ref_list_append.asp

Passing a copy of an array as input to the sorting algorithm:

- Stack Overflow, How to clone or copy a list in Python [Online] Available from: https://stackoverflow.com/questions/2612802/how-to-clone-or-copy-a-list

Calculating average time:

- numpy.mean available from: https://docs.scipy.org/doc/numpy/reference/generated/numpy.mean.html

Creating a pandas dataframe:

- Pandas.dataframe documentation [Online] Available from: https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html
- GeeksforGeeks, Python Pandas Dataframe [Online] Available from: https://www.geeksforgeeks.org/python-pandas-dataframe/

Set index and transpose dataframe:

- Pandas.dataframe.set_index documentation [Online] Available from: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.set_index.html

- Pandas.dataframe.transpose documentation [Online] Avilable from: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.transpose.html
- Geeksforgeeks, Python pandas dataframe.transpose [Online] Available from: https://www.geeksforgeeks.org/python-pandas-dataframe-transpose/

Rounding dataframe output to 3 decimal places:

- Pandas.dataframe.round documentation [Online] Available from: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.round.html
- Geeksforgeeks, Python pandas dataframe.round() documentation [Online] Available from: https://www.geeksforgeeks.org/python-pandas-dataframe-round/

Sending data from pandas dataframe to a csv file:

- Pandas.dataframe.to_csv documentation [Online] Available from: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_csv.html
- GeeksforGeeks, Saving a Pandas Dataframe as a CSV [Online] Available from: https://www.geeksforgeeks.org/saving-a-pandas-dataframe-as-a-csv/

Plotting data with matplotlib.pyplot

- Matplotlib.pyplot documentation [Online] Available from: https://matplotlib.org/3.2.1/api/_as_gen/matplotlib.pyplot.html
- Matplotlib customizing matplotlib with style sheets and rc params [Online] Available from: https://matplotlib.org/tutorials/introductory/customizing.html
- Matplotlib.pyplot.plot [Online] Available from: https://matplotlib.org/3.2.1/api/_as_gen/matplotlib.pyplot.plot.html#matplotlib.pyplot.plot
- Matplotlib.pyplot.title [Online] Available from: https://matplotlib.org/3.2.1/api/_as_gen/matplotlib.pyplot.title.html#matplotlib.pyplot.title
- Matplotlib.pyplot.ylabel [Online] Available from: https://matplotlib.org/3.2.1/api/_as_gen/matplotlib.pyplot.ylabel.html#matplotlib.pyplot.ylabel
- Matplot.pyplot.xlabel [Online] Available from: https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.xlabel.html
- Matplotlib.pyplot.grid [Online] Available from: https://matplotlib.org/3.2.1/api/_as_gen/matplotlib.pyplot.grid.html#matplotlib.pyplot.grid
- Matplotlib.pyplot.legend [Online] Available from: https://matplotlib.org/3.2.1/api/_as_gen/matplotlib.pyplot.legend.html#matplotlib.pyplot.legend
- Matplotlib.pyplot.savefig [Online] Available from: https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.savefig.html