Computational Thinking with Algorithms Project

Benchmarking Sorting Algorithms

Patrick Moore - G00364753

1. Introduction

1.2 Sorting

Sorting is defined as arranging a collection of items into a particular order according to some pre-defined rules. Examples of this would include:

- Sorting a dictionary or telephone directory in alphabetical order,
- Ordering a list of employees from maximum salary to minimum salary,
- Listing a doctor's patients from youngest to oldest

Formally the conditions for sorting are a follows:

- A collection of items is said to be sorted if every item in the list is less than or equal to its successor
- If there is a list of items called A, then if A[i] < A[j], then i < j, i.e. if an item comes before another item in the list – it must be less than it
- For duplicate items, if A[i] = A[j] then there can be no k such that i<k<j if A[i] is not equal to A[k], or in other words duplicate items must be stored consecutively in the list
- The final sorted list must be a permutation of the original list – with no additions or ommisions

Sorting is an extremely important and useful exercise as it a makes many other operations on datasets easier. For example:

- Finding a word in a dictionary is easy because the dictionary is already sorted
- Determining particular descriptive statistics (such median, mode, inter-quartile range, maximum value, minimum value etc.) from a dataset is much easier if the dataset is sorted in advance

Due the importance of sorting, much time has been spent by computer scientists developing various sorting algorithms with a view to improving the performance of the sorting for a particular application. As result of this work, there are many different sorting algorithms available each with their own strengths and weaknesses. The objectives of this project are to:

- Introduce the concept of sorting and sorting algorithms
- Describe some of the key considerations when choosing a particular sorting algorithm

- Introduce five different sorting algorithms and describe how each one is expected to behave with respect to each of the key considerations
- Benchmark the performance of these algorithms by writing a Python application
- Discuss the results of the benchmarking exercise with respect to expected performance

When choosing a particular sorting algorithm for an application the following are the main considerations:

- Complexity in both time and space
- Performance
- In-place sorting
- Stable sorting
- Comparator functions,
- Comparison-based or non-comparison-based sorting

Each of these will be described in turn in the next sections of this report.

1.3 Complexity

With sorting algorithms, and with computing in general, the term complexity is used to describe the theoretical efficiency of an algorithm, and how it is likely to change depending on the size of the input data. It is a useful measure because it does not depend on real world constraints such as available memory or processing power, instead looking at the individual tasks in an algorithm and determining what impact these should have on run time as a dataset grows to a non-trivial size. Complexity can be analysed for both time and space, where time complexity looks at the impact of input data on run time, and space complexity considers how memory requirements are affected.

Complexity is considered to be a function of the input size, so for an input dataset with n data points, complexity is represented in terms of n. Some examples of how time complexity varies with input size are:

- An algorithm which takes an array as an input and returns the first item in the array will run in constant time (denote by O(1)), order of 1), as regardless of the size of the input set it will always take the same amount of time to run.
- An algorithm which takes an input array and loops through each item in it to find the sum of the items in the array will have a run time that varies in direct proportion to the size of the input data set, n. This is denoted by O(n).
- An algorithm that uses a nested for loop to determine if a dataset contains duplicates can have complexity that varies in proportion to the square on the size of the input data (denote by O(n^2)) in the worst case scenario (where no duplicates exists and the algorithm must make n^2 comparisons).

Space complexity considers how much additional memory would be required for larger input data sets. This would be important when using recursive algorithms which create a new memory stack for each recursive call of the function.

1.4 Performance

If complexity can be considered a theoretical measure of algorithmic efficiency, then performance can be though of as a practical measure. In other words, performance of an algorithm is measured by implementing it, and then taking real world measurements of time, disk space etc., to determine how the algorithm actually performs in the real word. There are a few interesting points to note on this. Firstly, as complexity is theoretical analysis of the algorithm it doesn't change based on real world factors such as operating system, processor, RAM available, programming language used, etc. This is useful when considering the best algorithm to use for a particular application. Performance, on the other hand, will change based on the actual implementation parameters of the algorithm. This means that performance is of limited use when describing an algorithm in general terms, however it is useful if the hardware and software to be used for the implementation are fixed, as it can help determine expected run times.

It should also be noted that one of the objectives of this project is to measure algorithmic performance from a real world implementation. The hardware and software used for running the comparisons in this project are as follows:

- Mid 2012 Apple MacBook Pro
- 2.5 GHz Intel Core i5 processor
- 8GB 1600 MHz DDR3 RAM
- Running macOS Mojave 10.14.4
- The algorithms are implemented using the Python programming language, version 3.6.8

1.5 In-place sorting

When a sorting program runs, it must first read the input data from the storage location (such as the computers hard drive) to memory (RAM). Depending on the size of this dataset and the memory available, the algorithm chosen may have to employ in-place sorting.

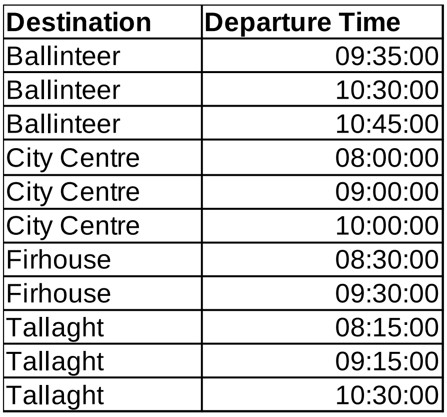
An in-place sorting algorithm is one that sorts the input using only a small, fixed amount of additional memory while completing the sort. For example, while sorting an array, one method might be to create an empty array and fill it with sorted copies of the elements from the input array. This method will require additional memory of the O(n), i.e. as the size of the input array increases, the additional memory requirements for this array of copies will grow linearly. If memory available is limited, or large input datasets are expected, then it might not be possible to allocate extra memory in this manner. Using an in-place sorting algorithm can overcome this limitation. With in-place sorting, the individual elements in the input array have their relative positions in the input array swapped during the sorting procedure, without the need for any additional memory.

1.6 Stable sorting

Until now we have only considered simple one dimensional arrays when talking about sorting, and sorting algorithms. In real life, data to be sorted is usually part of a two dimensional table that is to be sorted based on one column in that table. In this case, the column upon which the data is to be sorted is known as the 'sort key', and all other data in the table is called 'satellite data'. An example of this is shown as a bus departures timetable below:

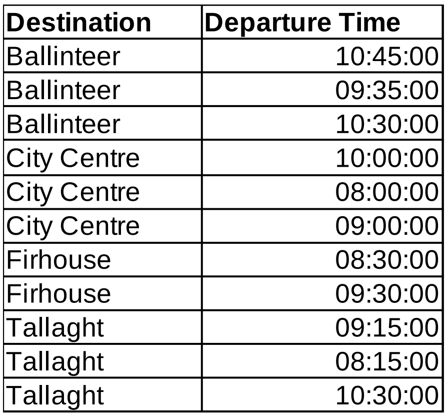
Destination	**Departure Time**
City Centre	08:00:00
Tallaght	08:15:00
Firhouse	08:30:00
City Centre	09:00:00
Tallaght	09:15:00
Firhouse	09:30:00
Ballinteer	09:35:00
City Centre	10:00:00
Ballinteer	10:30:00
Tallaght	10:30:00
Ballinteer	10:45:00

In this case we have a table of data that shows the destination and departure times of all buses leaving the station sorted by ascending departure time. If a particular passenger had planned on travelling to the City Centre and wanted to sort the timetable by destination to achieved this, it would be desirable for the sorting algorithm to preserve the current sort i.e. for items that have equal destinations the current sorting order (by ascending departure time) should still be preserved. This is known as stable sorting. If the table shown above is sorted based on destination using a stable sorting algorithm the result would be as follows:



Destination	Departure Time
Ballinteer	09:35:00
Ballinteer	10:30:00
Ballinteer	10:45:00
City Centre	08:00:00
City Centre	09:00:00
City Centre	10:00:00
Firhouse	08:30:00
Firhouse	09:30:00
Tallaght	08:15:00
Tallaght	09:15:00
Tallaght	10:30:00

Unstable sorting of the same data would not preserve the original order of the data, and in this case it would be more difficult for a passenger to determine what time the next City Centre bus is leaving at! The result of an unstable sort of this data is shown below:



Destination	Departure Time
Ballinteer	10:45:00
Ballinteer	09:35:00
Ballinteer	10:30:00
City Centre	10:00:00
City Centre	08:00:00
City Centre	09:00:00
Firhouse	08:30:00
Firhouse	09:30:00
Tallaght	09:15:00
Tallaght	08:15:00
Tallaght	10:30:00

The data is still sorted by ascending destination, however the relative positioning of items with the same destination has not been preserved in this case.

1.7 Comparable elements and comparator functions

In order for any sorting operation to take place, it must be possible to compare each item in a collection to the other items in the collection and decide which order to put them. More formally, for any 2 elements *p* and *q* in a collection, exactly one of the following must be true:

- p = q
- p > q
- p < q

It is quite straightforward to compare commonplace items such as integers, floating point numbers (which are sorted based on numerical value) or strings (which can sorted lexicographically). For more complex sorts, where the desirable sort is less intuitive a ‘comparator function’ must be defined that can take 2 elements in the collection (p,q) and return a different value for the case when p =q, p>q or p<q. An example of this might me sorting books based on genre where we want the books sorted in the following order:

- Non- fiction
- Romance
- Classics
- Sci- Fi
- Fantasy
- Horror

In this case the ordering of genres is not lexicographical, so a comparator function would need to be defined to compare elements based on the following logic:

Non-fiction > Romance > Classics > Sci-Fi > Fantasy > Horror

It is important to stress that sorting algorithms are independent of the definition of “less than” which is to be used so we need not concern ourselves with the specific details of the comparator function used when designing sorting algorithms.

1.8 Comparison based or non-comparison based sorting

A comparison based sorting algorithm is one that one uses comparisons between pairs of elements in the collection to determine the final order of the sorted data. Most of the well-known sorting algorithms fall into this category such as bubble sort, quick sort, merge sort and insertion sort. Benefits to using comparison-based sorting is that it is widely applicable and can be used on any collection of data that needs to be sorted (provided, of course the data can be compared in some way to determine the final order!). However, comparing elements in this manner can be very inefficient, and one of the peculiarities of this comparison based sorting is that no algorithm can do better that nlogn time complexity in the average or worst cases.

It is possible to improve sorting performance using a non-comparison based algorithm such as bucket sort or counting sort. However there are limitations placed on the input data that can be sorted in this manner. This concept will be discussed in more detail in section 2.

1.9 Comparison of sorting algorthims

In the next section of this report the sorting algorithms to be considered in this project will be described in turn. The algorithms to be covered are as follows:

1. Bubble sort (a simple comparison based algorithm)

2. Insertion sort (a simple comparison based algorithm)

3. Selection sort (a simple comparison based algorithm)

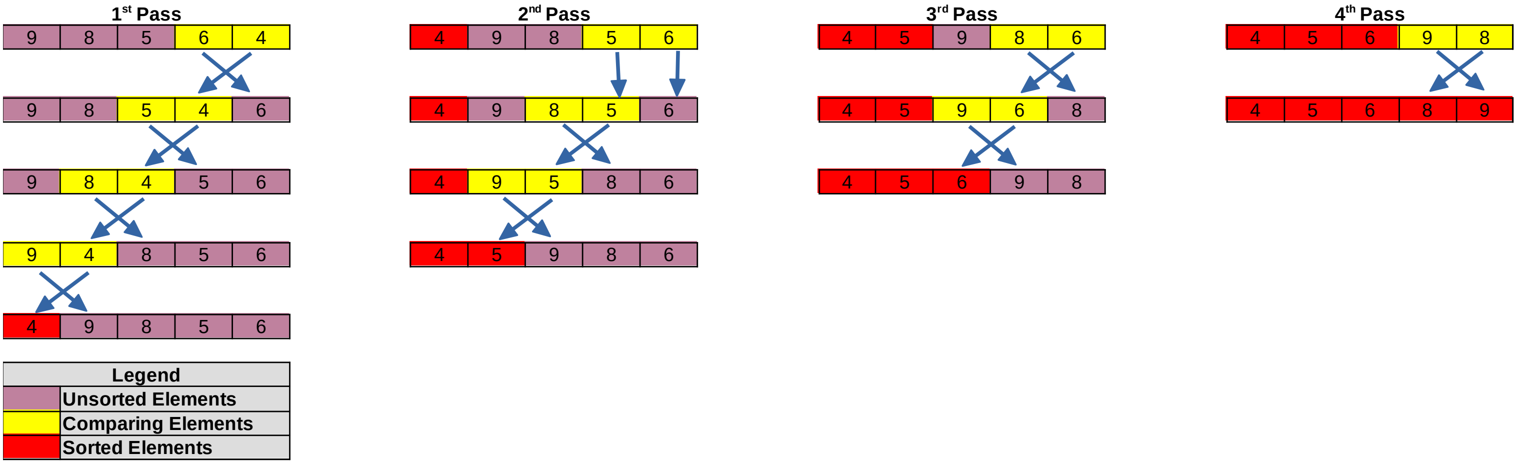
4. Merge sort (an efficient comparison based algorithm)

5. Counting sort (a non-comparison based algorithm

2. Sorting Algorithms

2.1 Bubble Sort

Bubble sort is a simple, comparison based sorting algorithm that is used to sort an array of elements by comparing them one by one and sorting them based on their values. Bubble sort works by starting at the end of the array and comparing the value of the last element with the one previous to it and then sorting them. It then moves to the second last element and compares it to the third last element and sorts based on the value. It continues in this manner until the smallest value in the array is in the first position. It then starts again at the last position and repeats this process until the second lowest value in the array is in the second position in the array. It repeats this process until all items in the array are sorted. Bubble sort gets its name from the way the values 'bubble' to the end of the array like bubbles rising in water. The diagram below describes an implementation of this algorithm on a real set of numbers.



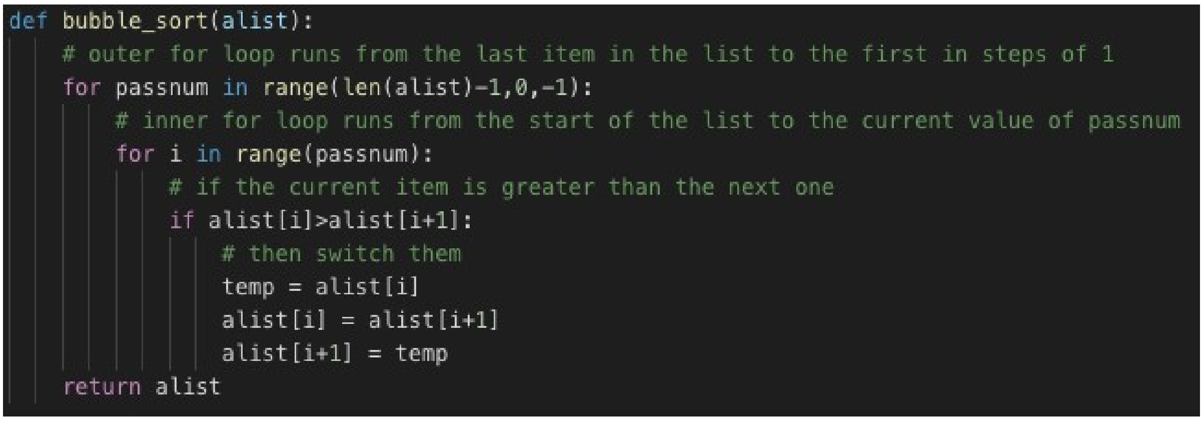
In the implementation above, we start at the last two elements in the array (6 and 4), compare them and swap them. We then move to the next two elements (4 and 5), compare them and swap them. We then move onto the next two elements (8 and 4), compare them and swap them. Finally we move on to the first two elements (4 and 9), compare them and swap them. After the first pass through the smallest item in the list (4), is in its correct position.

We then move on to the second pass, comparing the last two elements in the array (5 and 6), compare them but there is no need to swap them this time. We then move onto the next two elements in the array (8 and 5), compare them and swap them. Finally we compare the next 2 elements (9 and 5) and swap them. At the end of this pass the first 2 elements are in their correct positions.

In the third pass we again start at the back of the list, and compare the last 2 elements (8 and 6) and swap them. We then move onto the next 2 elements (9 and 6) compare them and swap them. After this step, the first 3 elements in the array are sorted.

For the final run through the array, the last two elements (9 and 8) are compared and swapped. The array is now sorted in ascending order.

The Python code to implement this algorithm is shown below:



```
def bubble_sort(alist):
    # outer for loop runs from the last item in the list to the first in steps of 1
    for passnum in range(len(alist)-1,0,-1):
        # inner for loop runs from the start of the list to the current value of passnum
        for i in range(passnum):
            # if the current item is greater than the next one
            if alist[i]>alist[i+1]:
                # then switch them
                temp = alist[i]
                alist[i] = alist[i+1]
                alist[i+1] = temp
    return alist
```

The Python code uses a nested for loop to carry out the comparisons and sorts. The outer for loop runs from the second last item in the list to the first item in increments of 1. The inner for loop runs from the first item in the list to the item currently indexed by the outer for loop. It compares the item with the next one in the list and swaps if it is bigger.

As shown above, in the first pass there are n-1 (4) comparisons completed. There are n-2 (3) comparisons in the second pass, n-3 (2) comparisons in the third and so on. So the total number of comparisons for an array with n elements will be:

Sum = (n-1) + (n - 2) + (n - 3) + ... 3 + 2 + 1

This equates to n(n-1)/2

So the time complexity of bubble sort is ***$O(n^2)$*** in the worst and average cases. Note that in the best case this is ***O(n)*** and it occurs when the array is already sorted.

The space complexity of this algorithm is constant ***O(1)*** as it only needs additional memory for one extra variable (temp). So it can be said to sort 'in place'. It should always be pointed out that this is a stable sorting algorithm that only swaps the relative positions of items if necessary.

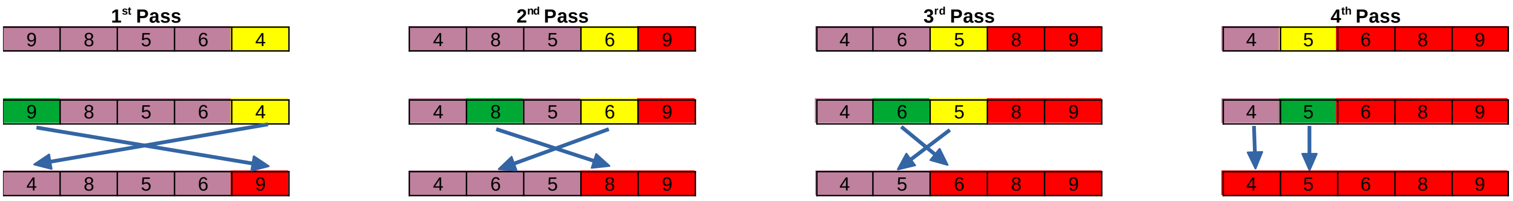
Bubble sort is really only useful as a teaching tool to help explain the concept of sorting algorithms and the analysis of them. It is used in this way because of its simplicity. However in practical terms, it is way too slow to be considered for even the most basic tasks.

2.2 Selection Sort

Another simple comparison based sorting algorithm is selection sort. Conceptually, selection sort is even more straightforward than bubble sort. Selection sort is implemented on an array containing n integers as follows:

- Start at the last element in the array (the item at index n-1), then check every item in array to find the maximum value in the array
- Once the item with the maximum value is found it is swapped with the last item in the list
- Move on to the second last item in the list (the item at index n-2), then check every item in the array between index n-2 and 0 to find the maximum value in this sub-array.
- Once the item with the maximum value in the sub array is found, it is swapped with the item at index n- 2
- This process is repeated until the array is sorted

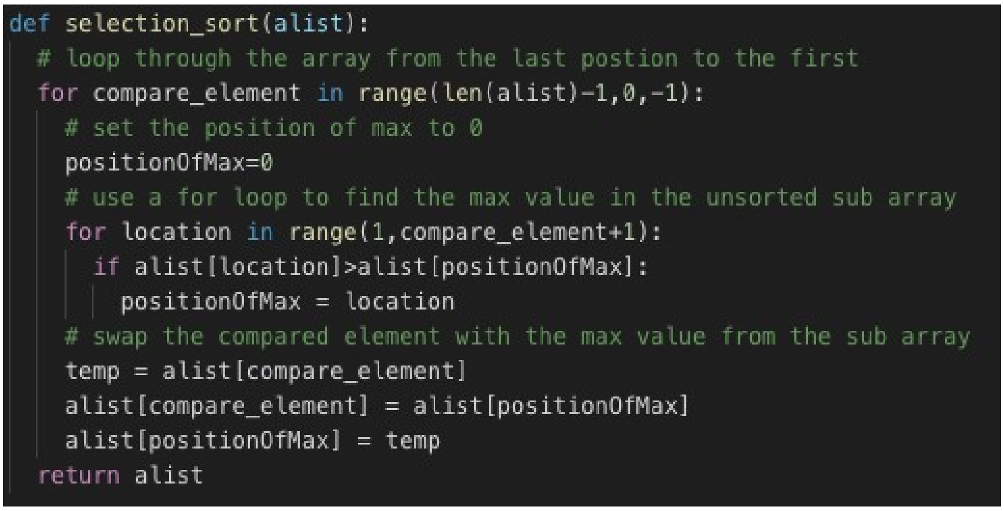
The diagram below describes an implementation of this algorithm on a real set of numbers.



Legend	
🟪	Unsorted Elements
🟨	Comparing Elements
🟩	Located Max Value
🟥	Sorted Elements

In the first pass we start with the last element in the array, we then check the entire array for the largest value (9). We can say that 9 is now in the sorted portion of the array, while the rest of the array can be called the unsorted subarray. We swap these values so the maximum value is in the last position in the array. Next, we consider the element in the second last position in the array (6), we then find the maximum value in the unsorted subarray (8) and swap these elements. On the third pass we consider the element in the third last position in the array (5). We again find the maximum number in the unsorted subarray (6) and swap it with this element. Finally we consider the item in the second position (or index 1 – value is 5 here). In this case it is the maximum value in the sub array so there is no swap required.

The Python code for this implementation of selection sort is shown below:



```python
def selection_sort(alist):
  # loop through the array from the last postion to the first
  for compare_element in range(len(alist)-1,0,-1):
    # set the position of max to 0
    positionOfMax=0
    # use a for loop to find the max value in the unsorted sub array
    for location in range(1,compare_element+1):
      if alist[location]>alist[positionOfMax]:
        positionOfMax = location
    # swap the compared element with the max value from the sub array
    temp = alist[compare_element]
    alist[compare_element] = alist[positionOfMax]
    alist[positionOfMax] = temp
  return alist
```

Just like bubble sort, a nested for loop is used for selection sort. In the case the outer for loop is used to pick the elements to be swapped, whereas the inner loop is used to find the maximum value of the unsorted subarray on each iteration of the outer loop.

As can be seen from the code above, the outer loop will run n – 1 times, while the inner loop with run n/2 times on average (somewhere between n and 2 times). So the time complexity of selection sort is ***$O(n^2)$*** in the best, worst and average cases. As for bubble sort, only a fixed small amount of additional memory is required for the temporary variables used for the loops and swaps, so the space complexity of selection sort is ***O(1)***.

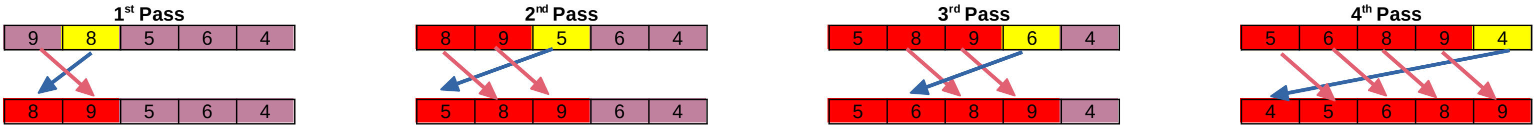
It should also be noted that this implementation of selection sort in unstable so it should not be used in cases where stability is important. But in reality, while selection sort usually performs better than bubble sort, it is still too slow to be considered useful for any real world applications. Like bubble sort, it is useful in the teaching of concepts such as sorting, algorithms ‘big O’ notation etc., as this sorting algorithm is conceptually quite simple.

2.3 Insertion Sort

Insertion sort is another of the simple comparison based sorting algorithms. It works in a way that's similar to how card players sort a hand of cards. If you had 5 cards in your hand and they were already sorted from smallest to largest, and you were given an extra card. The process you would follow would be to go through your sorted hand and see what position your new card should take. You would then insert the new card at is correct position.

The insertion sort algorithm would work on an array (a[], with n elements) by assuming that the first item in the array(a[0]), is the 'already sorted list'. It then looks at the second item in the list (a[1]) and treats this as it if was a 'new card' (note that the item is known as the 'key'). If the key is smaller than a[0] it shifts everything in the array right one space and inserts this item at position a[0]. Now the items in a[0] and a[1] are in the "already sorted list". The algorithm next considers a[2] as the key. It finds the position in the array where where this item should go, moves everything in the already sorted list that is greater than the key right one place and inserts the key in its correct position. It continues in this manner from a[3] to a[n-1] until the array is completely sorted.

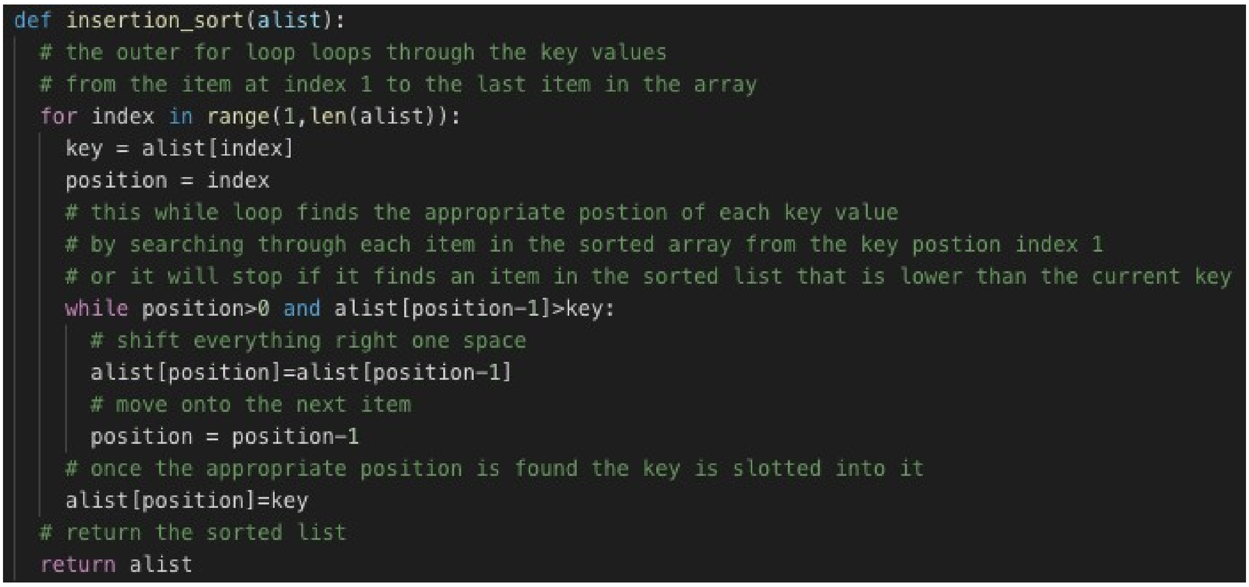
An example of this is shown below:



	Legend
🟪	Unsorted Elements
🟨	Current Key
🟥	Sorted Elements

- In the first pass, the element at index 1 is set as the key (8). It is compared to the item at index 0 (9). As it is smaller than 9, the 9 is pushed right by one place and the 8 is inserted at index 0.
- In the second pass the item at index 2 (5) is the key. It is smaller than both 8 and 9 so they are both shifted right by one position and the 5 is inserted at index 0.
- In the third pass the element at index 4 is the key. Note that in this case, the 6 is greater than 5, so 5 stays where it is. It is less than 8, so the 8 and everything to the right of it in the sorted portion of the list is shifted right by one position. The 6 is inserted at position once held by the 8.
- In the final pass, the key (4) is smaller than all other elements in the array, so everything is shifted right by one position and the 4 is inserted at index 0.
- The list is now sorted

The Python code for an insertion sort implementation is shown below:



```python
def insertion_sort(alist):
  # the outer for loop loops through the key values
  # from the item at index 1 to the last item in the array
  for index in range(1,len(alist)):
    key = alist[index]
    position = index
    # this while loop finds the appropriate postion of each key value
    # by searching through each item in the sorted array from the key postion index 1
    # or it will stop if it finds an item in the sorted list that is lower than the current key
    while position>0 and alist[position-1]>key:
      # shift everything right one space
      alist[position]=alist[position-1]
      # move onto the next item
      position = position-1
    # once the appropriate position is found the key is slotted into it
    alist[position]=key
  # return the sorted list
  return alist
```

Like selection sort and bubble sort, insertion sort uses a nested loop. For an input array of size n, The outer for loop will always need to run n-1 times as it assigns the key to the various items in the array. However, the inner while loop only runs until it finds the correct position in the array for the current key. This means than if an array is nearly sorted (or said to have few inversions), the while loop will have less iterations to complete.

The time complexity of insertions sorted is related to the number of inversions in the input array. The total number of comparisons made by insertion sort is the number of inversions plus n-1 (the number of times the for loop must run). The number of inversions for an already sorted array is 0, so in the best case insertion sort run in linear time (O(n)). In the worst case (when a reverse sorted list is passed to the sorting function) the number of inversions is:

$$\frac{(n-1)xn}{4}$$

The time complexity in the worst case is approximately O(n^2). Note that in the average case the number of inversions is half of the number of inversions in the worst case so the time complexity is still O(n^2).

The space complexity for insertion sort in constant as only a small amount of additional memory is required for the ‘key’ and ‘position’ variables. Insertion sort provides stable sorting so this makes it advantageous in situations where stability is important.

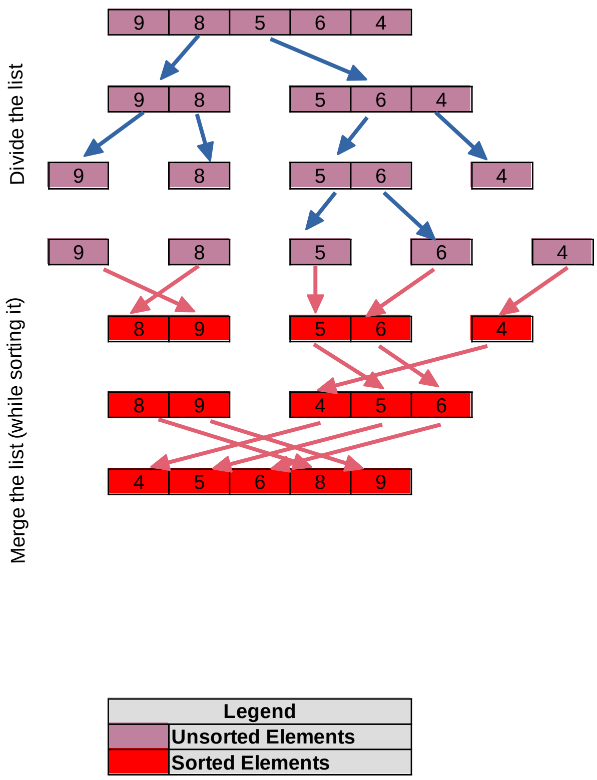
Of the 3 simple comparison based algorithms discussed here, insertion sort is the only one that could really be considered for any real world application. If performs very well on data that is already mostly sorted and also for smaller lists of data. In some cases insertion sort may outperform more complex algorithms.

2.4 Merge Sort

Merge sort is known as an efficient comparison based sorting algorithm. Instead of using nested iteration to make the comparisons, it instead uses a recursive approach to sorting. The approach it uses is known as a 'divide and conquer' method. A divide and conquer approach to problem solving is one that divides a large problem into smaller sub-problems, solves the sub-problems easily and then uses these small solutions to solve the larger problem. In our case the problem to be solved is sorting an array of data. Merge sort works as follows:

- Split the array into 2 sub arrays (approximately equal in length)
- Split each of the sub arrays into 2 sub arrays (approximately equal in length)
- Continue until each sub array has exactly one element in it (which is sorted by default)
- Merge the sub arrays back together in a sorted order

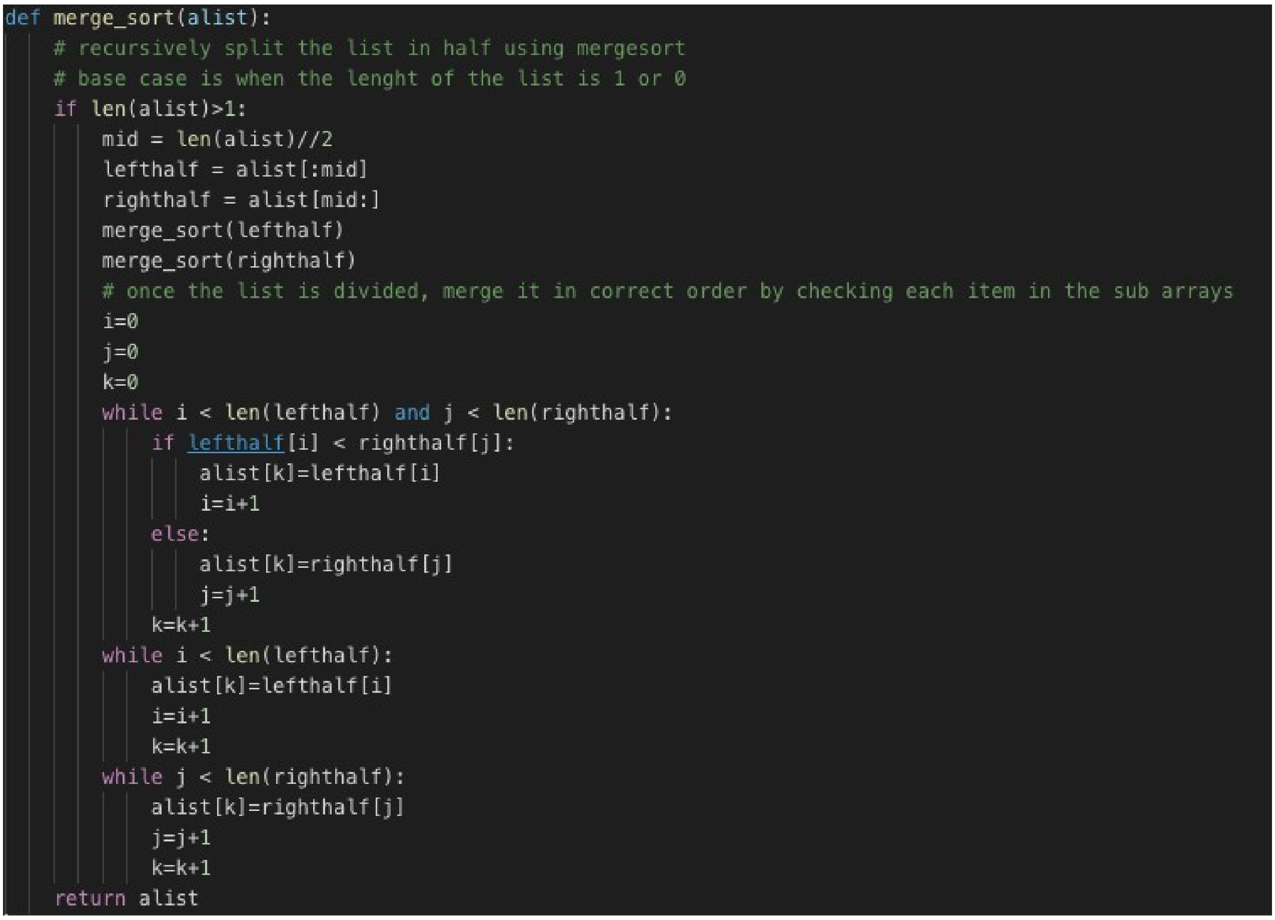
An example of merge sort is shown below:



Conceptually this is quite simple, in the first 3 steps the list is broken down into lists on length 1. Then the hard work begins merging these small lists into sorted list. The merge works as follows:

- 8 and 9 are merged in the correct order (a swap is required here)
- 5 and 6 are merged in the correct order (no swap required)
- 4 does nothing
- In the next step the 8 and 9 do nothing, but 4 is merged with 5 and 6
- In the last step the entire list is merged back together in the correct ordering

The Python code for merge sort is shown below:



```python
def merge_sort(alist):
    # recursively split the list in half using mergesort
    # base case is when the lenght of the list is 1 or 0
    if len(alist)>1:
        mid = len(alist)//2
        lefthalf = alist[:mid]
        righthalf = alist[mid:]
        merge_sort(lefthalf)
        merge_sort(righthalf)
        # once the list is divided, merge it in correct order by checking each item in the sub arrays
        i=0
        j=0
        k=0
        while i < len(lefthalf) and j < len(righthalf):
            if lefthalf[i] < righthalf[j]:
                alist[k]=lefthalf[i]
                i=i+1
            else:
                alist[k]=righthalf[j]
                j=j+1
            k=k+1
        while i < len(lefthalf):
            alist[k]=lefthalf[i]
            i=i+1
            k=k+1
        while j < len(righthalf):
            alist[k]=righthalf[j]
            j=j+1
            k=k+1
    return alist
```

Merge sort is quite fast and has a time complexity of O(nlogn) in the best worst and average cases. This determined because constant division in half has a complexity of logn, and merging a list of items with n elements will require n operations. Note that due to the recursive method used, a new memory stack is required for each call. This means that the space complexity is O(n). This might make merge sort less attractive on large list if there is only a small constant amount of memory available.

Merge sort is also a stable sorting algorithm as equal items will retain the relative positions in the sorted array. Note that merge sort can be used for real world problems due to it’s speed (note that nlogn is the best time complexity that can be achieved with comparison based sorting). Another benefit of this algorithm is that its time complexity is similar for the best, worst and average cases. This can be useful if predictable run times are required.

2.5 Counting Sort

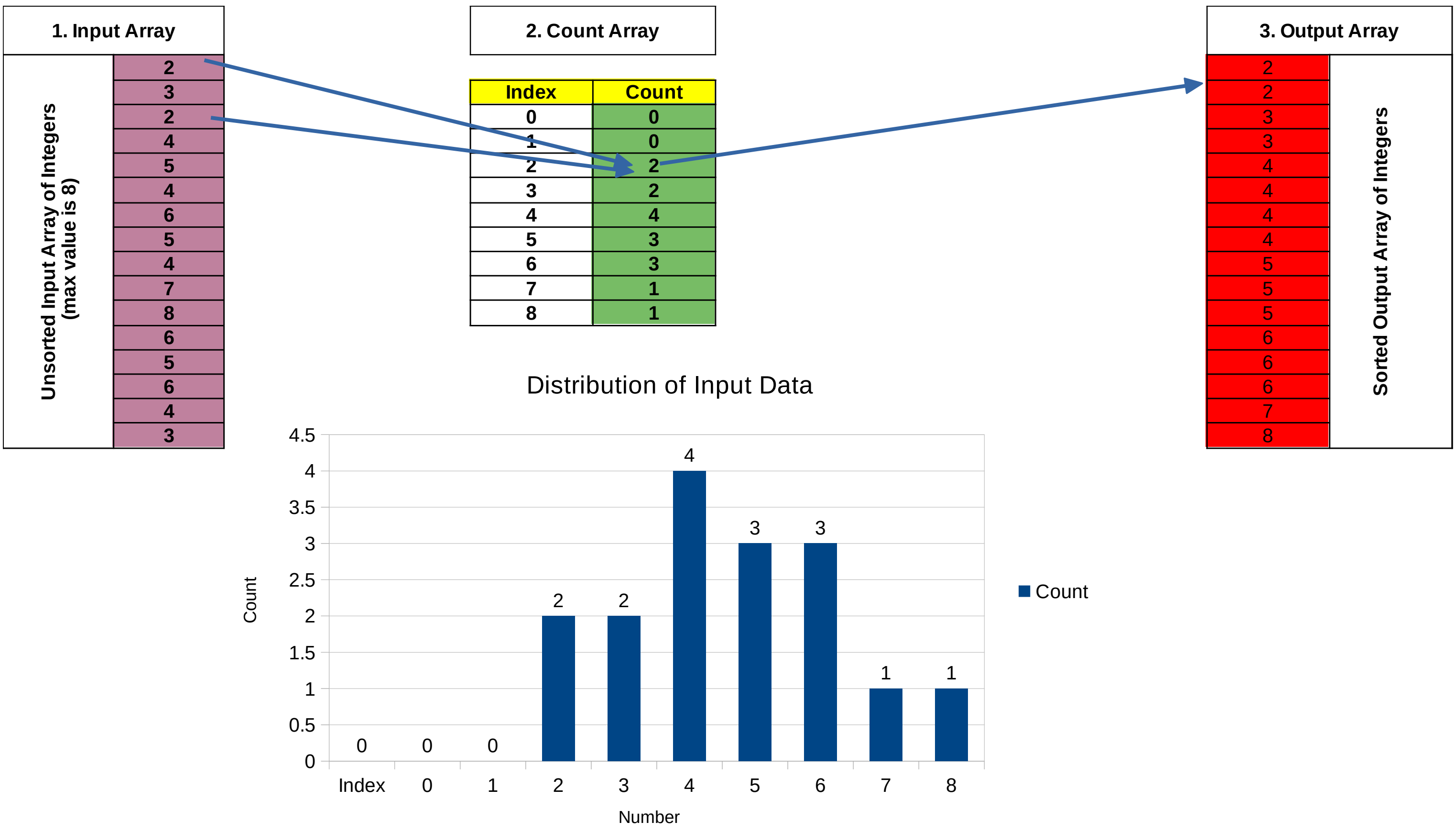
So far, the four sorting algorithms that were discussed were known as comparison based sorting algorithms. These all work by comparing the various elements in a list or array to each other, in order to determine the correct order the items should be sorted in. These types of algorithms (especially the efficient ones) are useful because they can be used on anyinput data, without the user having to know anything about the data or make any assumptions about the data. This means that they are widely applicable to a broad range of sorting problems. The downside to comparison based sorting is that it can be mathematically proven that no comparison based sorting algorithm can ever perform better than nlogn time complexity in the average or worst complexity.

It is possible to improve upon this time complexity, but the algorithms used must operate in a different way to comparing each element to every other element. There is a trade off when doing this, because in order to use one of these non-comparison based algorithms, the user must have some idea of the type, content and range of the input data. One such non-comparison based sorting algorithm is counting sort.

In order for counting sort to be used, we must know what the data to be sorted will look like, and over what range it is to be distributed. So if we are sorting a list of integers, we need to know what the maximum value of the data is going to be. The procedure for implementing counting sort is as follows:

- Determine the range of input data, and use it to define a number of keys (values that the data can possibly take on). So if we are sorting integers in the range of 1 to 1000 we will need a count[] array of 1000 keys to store the count.
- Iterate through the input[] array and count how many instances of each key are in it. Store the results of this iteration in the count[] array.
- Create a new results[] array, and enter the data based on the count of each key. Note that it is unstable unless the algorithm references the index from the input array when constructing this results array.

An example of counting sort is shown below:



1. Input Array

Unsorted Input Array of Integers (max value is 8)

2
3
2
4
5
4
6
5
4
7
8
6
5
6
4
3

2. Count Array

Index	Count
0	0
1	0
2	2
3	2
4	4
5	3
6	3
7	1
8	1

3. Output Array

Sorted Output Array of Integers

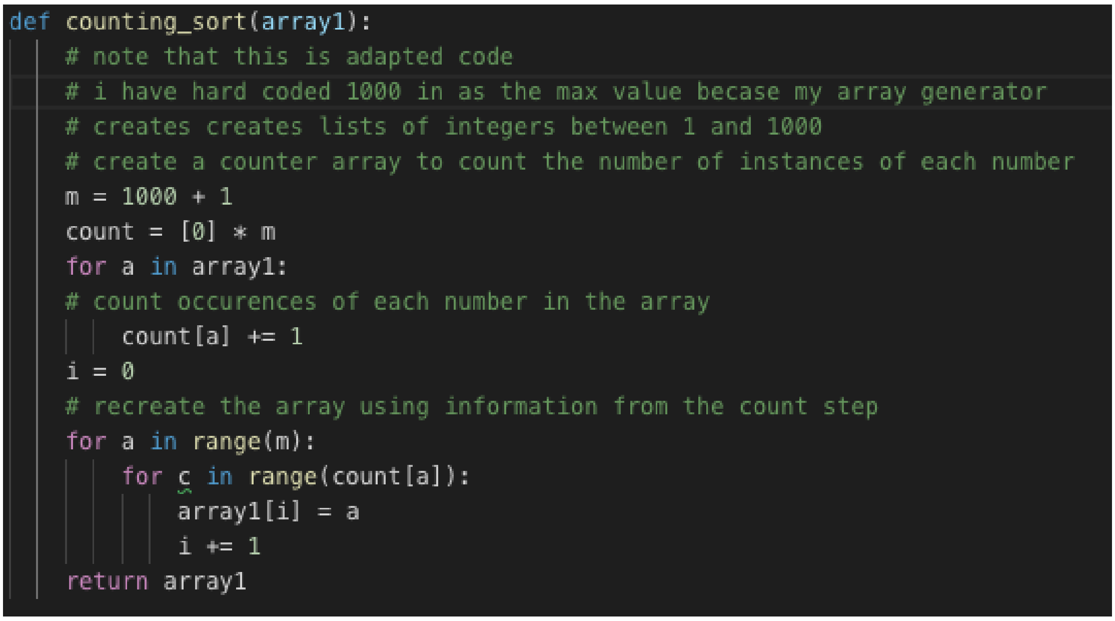
2
2
3
3
4
4
4
4
5
5
5
6
6
6
7
8

The above demonstrates how counting sort can be used on an array of 16 integers with a maximum value of 8. As there the maximum value in the input array is 8, we need an array with 9 elements (i.e. an array with maximum index of 8) to store the count. The values at each index in the count array are initially set to zero. The next step is to iterate through the input array and check the value of each element. We then increment the element at that index in the count array. In our example, every time the value in the input array is 2, we increment the element at index 2 in the count array. This allows to determine the distribution of values in the input array. The final step is to construct an output array using the information we have in the count array. In our case we have :

- no 0's
- no 1's
- 2 x 2's
- 2 x 3's
- 4 x 4's
- 3 x 5's
- 3 x 6's
- 1 x 7
- 1 x 8

The output array is constructed to reflect this.

The Python code for counting sort is shown below:



```python
def counting_sort(array1):
    # note that this is adapted code
    # i have hard coded 1000 in as the max value becase my array generator
    # creates creates lists of integers between 1 and 1000
    # create a counter array to count the number of instances of each number
    m = 1000 + 1
    count = [0] * m
    for a in array1:
    # count occurences of each number in the array
        count[a] += 1
    i = 0
    # recreate the array using information from the count step
    for a in range(m):
        for c in range(count[a]):
            array1[i] = a
            i += 1
    return array1
```

Personally I believe this is a great solution for sorting, the algorithm is elegant in its simplicity. However it can only be used on lists of positive integers when the maximum possible value in know. But the benefit to using this over some of the comparison based algorithms is that the best, worst and average case time complexity is n + k, where n is the size of the input array and k is the size of the count array. The space requirements are also n + k. My implementation of counting sort is only suitable for sorting simple lists of integers, as such stability is not really a problem - but this implementation would not be stable if used on an integer sort key of a 2 dimensional array of data. In order to maintain stability the relative indexing of the input array can be used.

3. Implementation & Benchmarking

This section describes the Python application that was written to carry out the benchmarking process, detailing the various functions and what they do. It also covers the benchmarking process by running the application and discussing the results using both a table and graph.

3.1 Implementation of the python application

The python application used for carrying out this benchmarking project is included with the submission in a python file called `project.py.` The application is divided into 5 sections using comments to make it more human readable. Here each of these sections is described in detail.

3.1.1 – Section 1 - Import the required libraries for the project

In this section the required libraries are imported. These libraries are:

time: used for timing each of the sorting algorithms

numpy: used for generating lists of random integers

pandas: used to create a dataframe to store the output from the trials

matplotlib.pyplot: used for creating a graphical representation of the benchmark tests

3.1.2 - Section 2 - Define the 5 Sorting Algorithms to be Used

In this section the five sorting algorithms to be tested are defined. As described in section 2, these are:

- Bubble sort
- Selection sort
- Insertion sort
- Merge sort
- Counting sort

Note that each function takes an input array as an argument and returns a sorted array, using the respective sorting algorithm.

3.1.3 – Section 3 - Define the timer functions to benchmark the algorithms

In this section the timer functions that are to be used to carry out the benchmarking are created. A description of what each one does is provided below:

array_create(): Takes an integer as argument and returns a list of randomly generated integers (between 1 and 1000) of the specified length. It uses the numpy package to generate the list.

timer(): Takes an input array and a sorting algorithm as arguments. Runs the sorting algorithm on the input array and returns the time it takes in seconds.

average_time(): Takes a number of runs, test size and sorting algorithm as inputs. It runs for the specified number of times, each time it creates a new random array of the specified length and times how long it takes to sort using the specified algorithm. It returns the average run time.

3.1.4 - Section 4 - Formatting the output

In this section the functions that are used to carry out the trials and format the output are defined. A description of each is provided below:

algo_trial(): This is the function that actually carries out the trials. It takes a sorting algorithm and test size as arguments. It creates a random array of the specified size, and uses the average_time function to find the average run time for 10 runs and returns the average time in milliseconds formatted to 3 place of decimal.

col_create(): This is a function used to create columns for a pandas dataframe. Each column will contain the data for a particular test size. So it takes a list of sorting algorithms and a test size as arguments. It then loops through the list of algorithms and calls the algo_trial function for each for the specified test size. It returns a column of data for the dataframe.

df_create(): takes a data dictionary as an input and converts it to a pandas dataframe. This is done because it is simple to format the output of a dataframe.

results_plot(): Takes a data dictionary, list of test sizes and list of sorting algorithms as input and plots the relative performance of each using matplotlib.pyplot.

results_export(): Takes a data dictionary as input and exports it to timestamped .csv file. Note that a copy of this has been included with the submission.

3.1.5 - Section 5 - User Interface

The first part of the user interface is to create some variables to store the master data to be used for the application.

sorts: this is a list of the names of the sorting algorithms. It is used to index the pandas dataframe.

algorithms: this is a list of the sorting algorithm functions. It is passed to the col_create function when creating the columns for the dataframe.

n_trial: this is a list of the various test sizes that should be used to test each algorithm

trial: this is the data dictionary that is used to create columns for the dataframe. Note that it is the declaration of this variable that starts the benchmarking project. It can can take several minutes for this to finish. Once this variable is defined it can be passed to the df_create, results_plot and result_export functions in order to be output to the user in the desired format

The next part of the user interface is to define a main() function to interact with the user on the command line. The main() function is quite straightforward. The user is allowed to enter the following values:

graph: creates a graphic using matplotlib.pyplot

table: returns a formatted grid of data to the command line

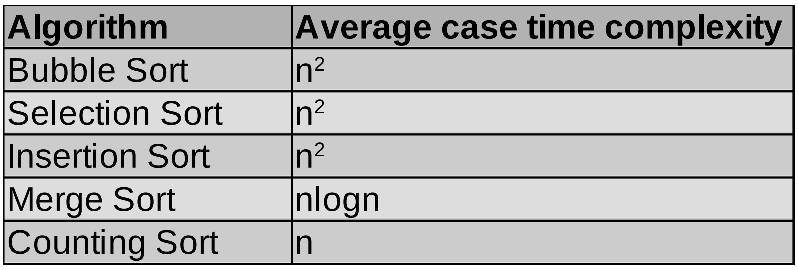
export: exports the results to a timestamped .csv file

quit: stops the application

Note that user input is validated using a simple while loop, and that once the main function returns the users selection, it will then recursively call itself until the user chooses quit

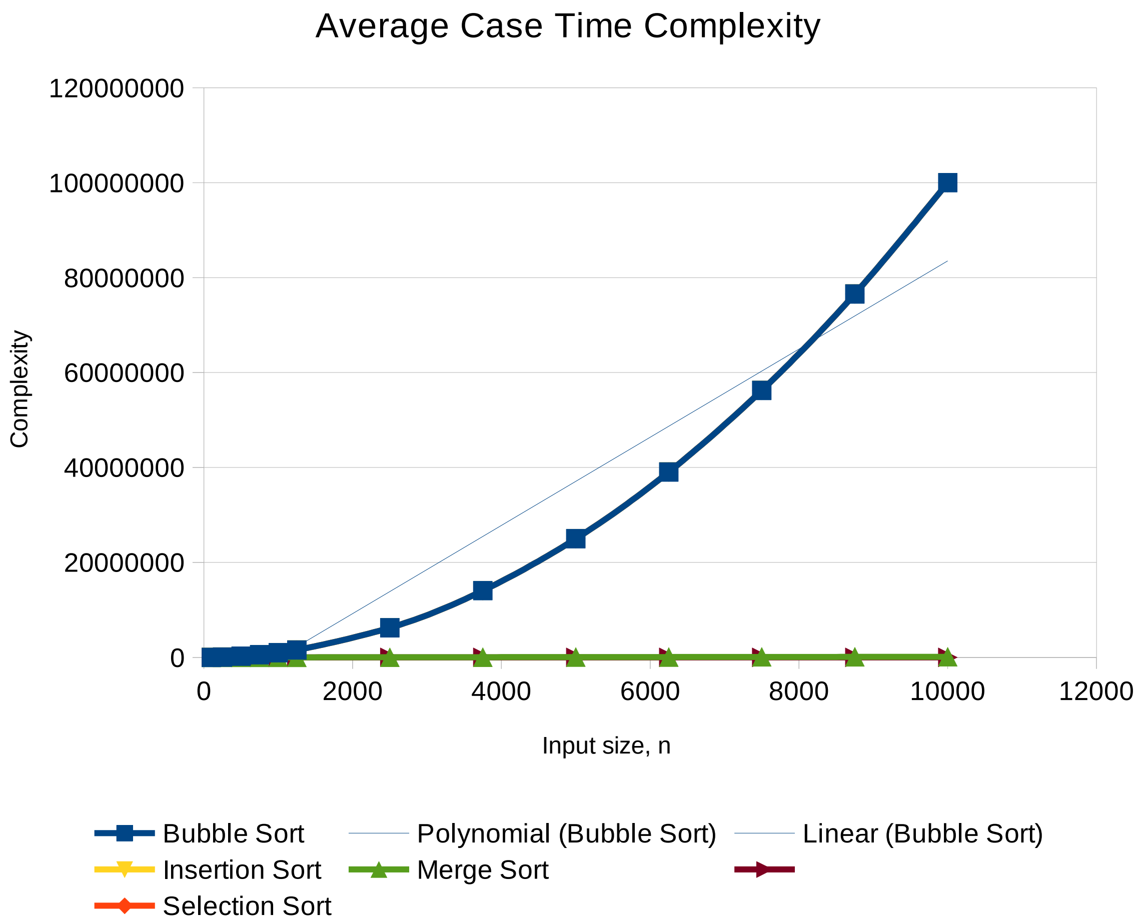
3.2 Results of the benchmarking exercise

In this final section of the project, the results of the benchmarking exercise are presented for discussion. Before doing this however, we should first consider what the expected outcome of this should be. The application for this project compared run times for each algorithm that were averaged over 10 runs each. Therefore when discussing expected performance we will concern ourselves with the expected average case performance of the sorting algorithms. The table below shows a list of the five sorting algorithms together with their average case time complexity.



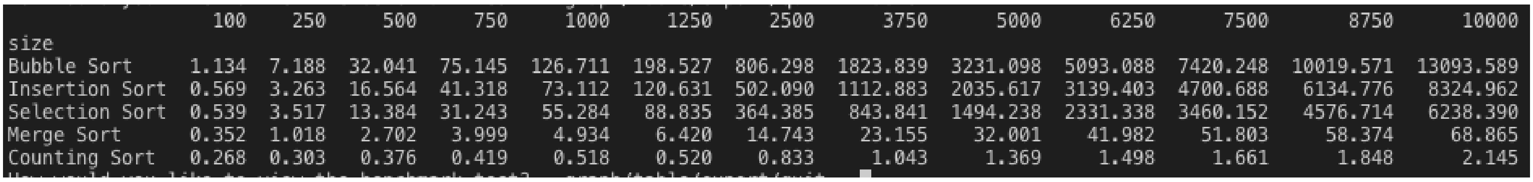
Algorithm	Average case time complexity
Bubble Sort	n^2
Selection Sort	n^2
Insertion Sort	n^2
Merge Sort	nlogn
Counting Sort	n

Graphically this can be represented as follows:



As shown above, the expected performance should be that bubble sort, selection sort and insertion sort should have similar performance – which will be quite poor for larger input sizes! Merge sort should perform much better while counting sort should perform the best on larger lists.

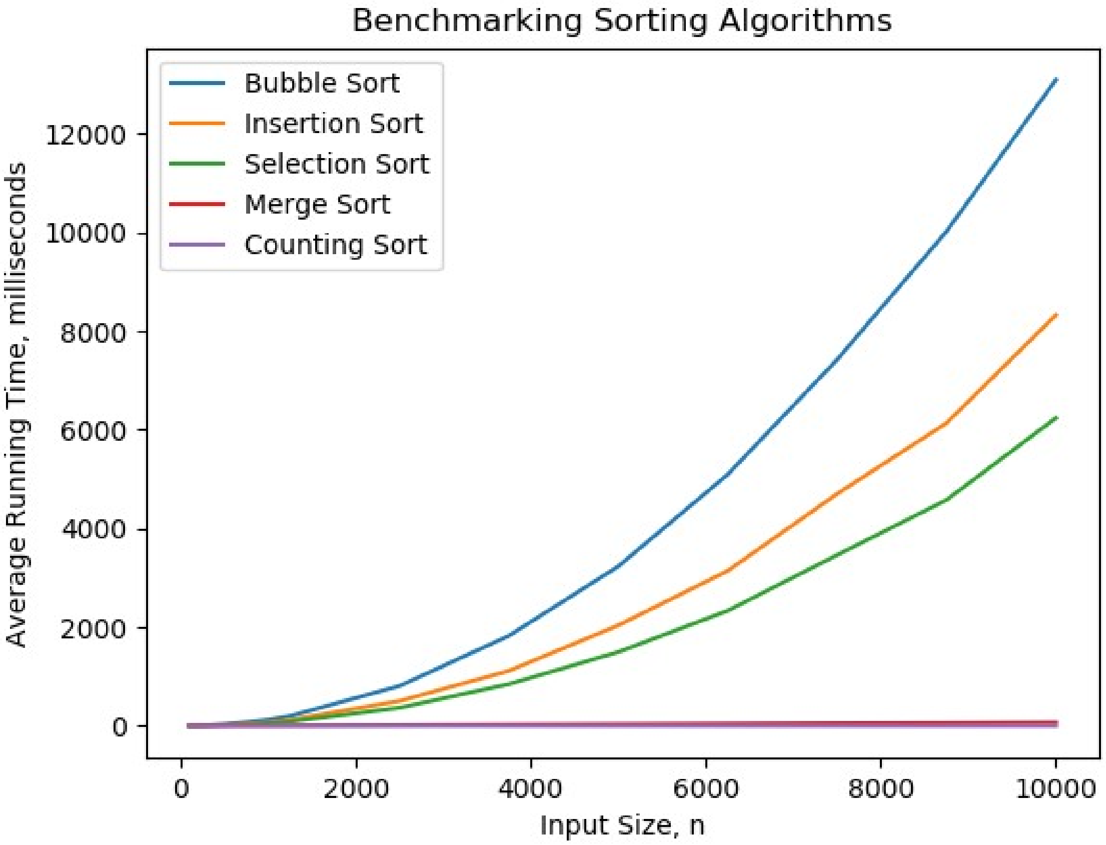
The table below shows the results of the benchmarking exercise, as generated using the `project.py` Python application.



```
                 100    250     500     750     1000     1250     2500      3750      5000      6250      7500       8750      10000
size
Bubble Sort     1.134  7.188  32.041  75.145  126.711  198.527  806.298  1823.839  3231.098  5093.088  7420.248  10019.571  13093.589
Insertion Sort  0.569  3.263  16.564  41.318   73.112  120.631  502.090  1112.883  2035.617  3139.403  4700.688   6134.776   8324.962
Selection Sort  0.539  3.517  13.384  31.243   55.284   88.835  364.385   843.841  1494.238  2331.338  3460.152   4576.714   6238.390
Merge Sort      0.352  1.018   2.702   3.999    4.934    6.420   14.743    23.155    32.001    41.982    51.803     58.374     68.865
Counting Sort   0.268  0.303   0.376   0.419    0.518    0.520    0.833     1.043     1.369     1.498     1.661      1.848      2.145
```

As expected, bubble sort is worst performing sorting algorithm over all. It takes, on average, over 13 seconds to sort a random list of 10,000 numbers. Insertion sort fared slightly better at just over 8 seconds. In this trial, selection sort performed the best of the three simple comparison based algorithms taking just over 6 seconds, on average to sort the largest lists. While there there are differences between the actual time taken for the three algorithms – they are all of the same order of magnitude – approximately 10 seconds. Merge sort was, on average, 100 times faster than selection sort – in other words a completely different order of magnitude. Likewise counting sort was nearly 30 times faster that merge sort, taking just 2 milliseconds to sort lists containing 10,000 random numbers. However, it should be noted that the implementation of counting sort here can only be used on lists of integers whose values are between 1 and 1000. If an array containing a number larger that this is passed to it, it will throw an error. The trade off for the better performance is that the algorithm can only be used in certain circumstances.

The results are also displayed graphically below:



The results of the exercise are broadly in line with what we would have expected to achieve. Bubble sort would have been expected to perform the worst, and it did. Insertion might be expected to perform better that selection sort in some instances, However, if the input data is very unordered and has lots of inversions in it (as randomly generated data would be expected to), insertion sort can be very inefficient which is what the results here are showing. Merge sort would be expected to perform much better the three of these and it has. Counting sort would be expected to perform the best and it has.

Finally, I would just like to comment on the performance of my program in general. As described in the introduction, this project was carried out using a 2012 macbook pro with an i5 processor and 8GB of RAM. While these specs are adequate for most day-to-day tasks, I would say that benchmarking sorting algorithms really was a stretch. The performance of the program was quite poor taking 15 minutes to complete all of the testing and then to load to the user interface.

3.3 Conclusions

Having introduced the five sorting algorithms, and then benchmarked them in python, the following conclusions can be drawn:

1) Simple comparison based sorting algorithms such as bubble sort, selection sort and insertion sort are simply far too slow to be useful for any real world applications

2) Merge sort is perfectly adequate for sorting large arrays of random integers, and can be used on arrays containing unknown ranges of data.

3) Counting sort is a very efficient method for sorting large arrays of data, and will provide extremely fast sorts provided the range of likely input values is known in advance