G00007918
Project – Benchmarking Sorting Algorithms

Introduction

Sorting algorithms are used to rearrange elements in a list/array. Elements are compared using a comparison operator (or non-comparison orperations as in the case of counting sort), and then rearranged in the correct order either chronologically or alphabetically, in descending or ascending order. There are 3 main types of sorting operators that I will describe in this document, ones that involve the simple comparison of adjacent elements, an efficient comparison sort that uses a 'divide and conquer' paradigm, and counting sort, which uses mathematical operations to count and then rearrange values into their correct order.

Complexity – time

The time complexity of a sorting algorithm describes the amount time it takes for an algorithm to complete its sorting operation. It is calculated by counting the amount of elementary operations involved in the sort. Each operation is assumed to be performed at a constant rate. Time complexity is generally expressed as a function of the size of the input.(1)

In measuring an algorithm's complexity, it is most useful to measure the algorithm's asymptotic behavior, or rate of growth as the input size increases. The worst-case, or asymptotic upper bound time complexity of an algorithm is the fastest rate of growth of the algorithm as input size increases. The worst case describes the algorithm's running time at most. The best-case time complexity, or asymptotic lower bound, describes the slowest rate of growth of an algorithm, or least running time. The average-case time complexity of an algorithm is the average amount of time used by the algorithm, over all inputs of equivalent complexity.

Average-case complexity is often a more accurate measure of an algorithm's performance than worst-case measurements, as many worst-case scenarios would never happen in practice, and are therefore not effective in determining the efficiency of an algorithm in a real world scenario.

Big O notation - O(n)
The upper bound of algorithmic time complexity is expressed using big O notation. It is a form of asymptotic notation that means "the running time grows at most this much, but it could grow more slowly”. (2) The 'n' in the formula represents the input size in units of bits.

Omega/big Ω Notation - Ω(n)
For large enough input sizes, omega notation describes the asymptotic lower bound time complexity of an algorithm. The rate of growth of an algorithm will never grow slower than this value. (3)

Theta/big Θ notation - Θ(n)
Once the input size n is large enough, we use theta notation to give us an asymptotically tight bound above and below, which is to say that the best and worst running times are between 2 complexities.(4)

Algorithmic Performance

O(1)
Performance time in this instance is constant, as it does not depend on the size of the input. Only 1 operation needs to be performed to access a single element in the input array. Practically, this would be an scenario where we need the algorithm to find the first element in an array sorted in ascending order. (5)

O(log n)
Logarithmic time complexity is highly efficient. When n increases, the number of operations compared to

the size of the input decreases and tends to zero.(5)

O(n)
This is linear time complexity, wherein running time increases linearly in proportion to the size of the input. It is the best time complexity for algorithms that need to scan through each element in an array sequentially to find a particular value. If the number of elements in the list is constant however, the algorithm can be said to run in constant time.(5)

O(n log n)
Linearithmic time complexity is the fastest possible complexity for any comparison sorting algorithm. Quick sort and heap sort have n log n as their best case time complexities.(5)

O(n^2), O(n^3), O(n^4)
This represents polynomial time complexity. An algorithm has polynomial time if its running time is upper bounded by a polynomial expression in the size of the input for the algorithm. N^2 is said to run in quadratic time, n^3 is cubic, etc. Quicksort is an example of an algorithm that has this complexity.(5)

O(2^n), n!
2^n and n! represent exponential time complexity, or the fastest asymptotic rate of growth of an algorithm. (5)

Complexity – space

The space complexity of an algorithm is total space taken up in computer memory by the algorithm in relation to its input size. It also describes the auxiliary space used by the algorithm itself. Increasing the size of the input increases the memory required by the algorithm to function. (6)

For linear search algorithms, time complexity is O(n), however the space used by the algorithm remains constant O(1), as the search target is always the same. Bubble sort, insertion sort, and selection sort all have this amount of auxilary space. Merge Sort uses O(n) auxiliary space, whereas insertion sort and heap sort use O(1) auxiliary space. Space complexity of all these sorting algorithms is O(n) though.(7)

Quicksort has O(n) time complexity but O(n) space complexity in the worst case. Mergesort has O(n log n) linearithmic time complexity in the best, worse and average case, whereas its linear space complexity is O(n). (7)

In-place sorting algorithms

In-place sorting describes algorithms that sort input using no auxiliary data structure. They update the input sequence/array by swapping elements without using extra arrays to store data. They usually overwrite their input and with output, but may however require a small extra space for the replacement operation. Two examples of this type of algorithm are insertion sort and quick sort. (8)

Not-in-place sorting algorithms

Not-in-place sorting algorithms require extra memory to sort input data. This type of algorithm generally requires O(n) or more space to function. Extra arrays may be needed to store and count data. Merge sort, a 'divide and conquer' algorithm, requires extra linear space which is not constant and is thus considered out-of-place. (8)

Stable and unstable sorting algorithms

Stable sorting algorithms preserve the order of duplicate keys. Two objects with equal keys (ie: they both

have the same initial in an alphabetical sort) must appear in the same order in the sorted output as they appear in the input order in order for the algorithm to be considered stable. Insertion sort, bubble sort, and merge sort are considered stable. Heap sort and quick sort are considered unstable. (9)(10)

From a practical perspective, stable storing algorithms are important when we need to sort an input by multiple keys. A unstable algorthm will not be able to maintain ordering achieved by previous sorts in subsequent sorts.

Comparator functions

Comparator functions take the values from two indexes in an array as arguments and decide their relative order, which is then output. They can be used to to obtain sorted lists in increasing or decreasing order. (11)

Comparison-based sorting

Comparison based algorithms use a single comparison operation (a "less than or equal to" operator or a three-way comparison) to sort through adjacent elements in an input list and decide upon their correct alphabetical or numeric output order. A sorting algorithm that involves comparing pairs of values can never have a worst-case time better than O(n log n). (12)

Non-comparison-based sorts

Non comparison based sorting algorithms do not perform a simple direct comparison of input values. Counting sort for example uses several mathematical operations to count the incidence of elements in an input list and then maps the count to the index of each element. (21)

Sorting Algorithms

Bubble Sort

Bubble Sort is a simple comparison-based sort that grows linearly. Its best case time complexity is O(n) where the input list is already sorted, and it's average time complexity is O(n^2). Its space complexity is constant at O(1). Bubble sort makes multiple passes through a list, it compares the size of adjacent values in the list, and swaps values that are out of order. (13)(14)(15)

Code (15)

```
def bubbleSort(self,randomList):
   # Get the length of the input size list minus 1
   outer_start = randomList.size - 1

   # Loop backwards over the input list
   for passnum in range(outer_start,0,-1):
      # Create an inner loop to interate through the outer loop
      for i in range(passnum):
         # Use a condition to check if the current iteration value is larger than its adjacent value
         if randomList[i]>randomList[i+1]:
            # if true, store the larger value in the temp variable
            temp = randomList[i]
            # Make the current iteration of the input list equal to its adjacent value
```

```
randomList[i] = randomList[i+1]
# make the adjacent value equal to the current value of the iteration – sthus swapping the values
randomList[i+1] = temp
```

Suitability

Bubble sort is a very simple algorithm to code and implement. It is also a stable sorting algorithm, so it is useful for sorting lists by multiple keys, as it will maintain the order of each successive sort. Bubble sort needs little memory to operate, as it doesn't use temporary storage during the sorting operation. It is useful for small input lists, however it does not deal well with lists containing a large number of values. This is because it has O(n^2) worst case time complexity, and as such is not useful for practical sorting applications.

From running the code, we can determine from the graph that from 250 to 2000 inputs it has a O(n) or linear gradient. Then between 2000 and 10000 input sizes the algorithm appears to move to a steeper quadratic O(n^2) curve.

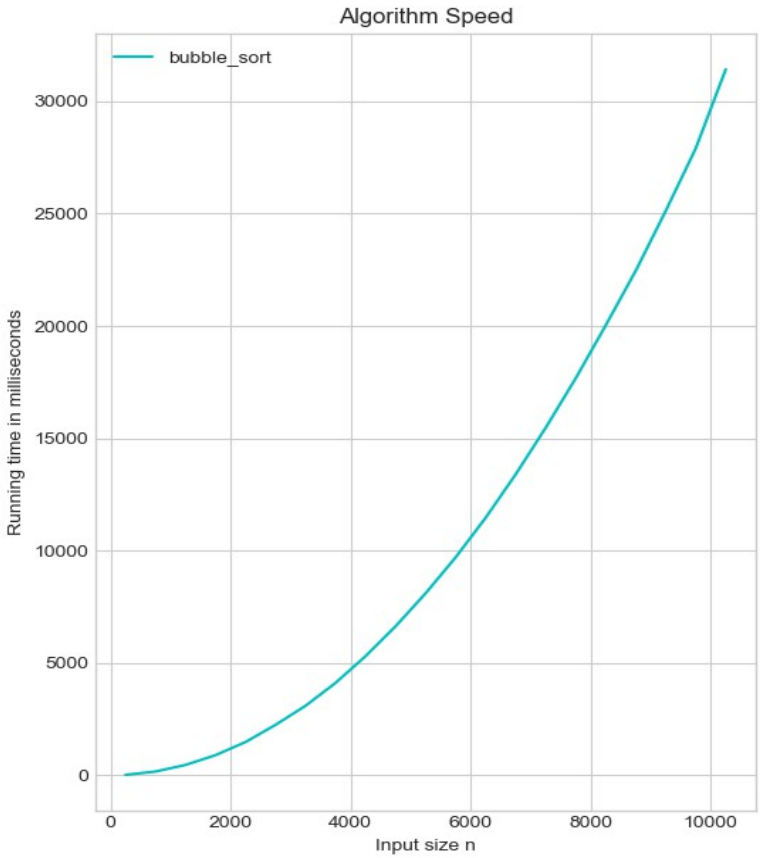

Figure 1

Insertion Sort

Insertion sort is a simple comparison-based sort. Its time complexity is O(n) or linear in the best case and O(n^2) or quadratic in the average or worst case. Its space complexity is O(1) as it is in-place algorithm and only requires a constant amount of memory to run. Insertion sort works by removing each iteration of the input list and comparing it to all of the elements before it in the list. The iteration is then inserted into its correct place in order, the sorted part of the list grows one element bigger, and the algorithm goes to the next iteration of the unsorted part of the list. As it doesn't need to loop through the entire input list to complete each iteration, it is substantially faster than bubble sort. (13)(14)(16)

Code

```
def insertionSort(self,randomList):
   # Get the size of the input array
   outer_start = randomList.size
   # Loop through the input array and store each value in a variable called 'currentvalue', store the index of each iteration in a variable called 'position'
   for index in range(1,outer_start):
      currentvalue = randomList[index]
      position = index

      #  Create an inner loop iterate backwards through the input array checking if each iteration is smaller than the 'currentvalue', or value of each index in the input index
      while position>0 and randomList[position-1]>currentvalue:
         # Set the value of each index in the input array to its previous value in the input list
         randomList[position]=randomList[position-1]
         # Decrement the position variable by 1
         position = position-1
         # If the condition is true, the each index of the input list will be set to the currentvalue
         randomList[position]=currentvalue
```

Suitability

Insertion sort is a very simple algorithm to implement. It also exhibits good performance when dealing with a small list. It also requires little memory, as it is an in-place sorting algorithm. As its average time complexity is quadratic, it is quite slow and therefore not suitable for large input sizes, however it is more efficient than other quadratic algorithms such as bubble sort or selection sort. It is also a stable sorting algorithm, so it can be used for implementing multiple sorts on arrays with multiple keys. (16)

From the graph below, we can see that its time complexity is similar to that of selection sort, however it is slightly faster than selection sort after the 4000 input mark. Before the 4000 input mark the algorithm exhibits linear time complexity and then curves upwards to a more quadratic time complexity after the 4000 mark.

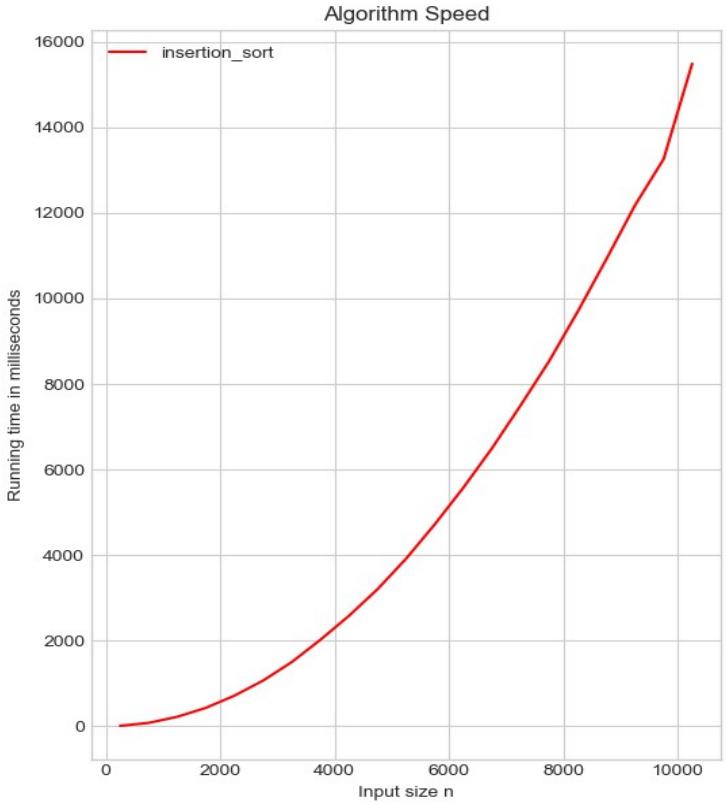

Figure 2

Selection Sort

Selection sort is a simple comparison-based sort. It has n^2 time complexity in the best and worst cases. It loops through an input list and selects the either the lowest value or the highest value in the list (depending on whether the sort is ascending or descending) and places that value at the beginning of the list. It then starts at the next index in the list and again picks out the lowest/highest value, placing it next to the previously chosen lowest/highest value. It continues in this manner until it builds a sorted sublist from the main list, requiring n−1 passes to sort n items as the final item will be in place after the n-1st pass. It doesn't need any extra space in order to perform the sort, therefore its space complexity is O(1). (13)(17)

Code

```
def selectionSort(self,randomList):
   # Get the length of the input list and reduce it by 1
   outer_start = randomList.size - 1

   # Create an outer loop to iterate through the input array and initiate the positionOfMax variable to zero
   for fillslot in range(outer_start,0,-1):
      positionOfMax=0

       # Create an inner loop to iterate through
      for location in range(1,fillslot+1):
         # The below condition compares the value at each iteration of the input array to check if it's smaller than its adjacent value
         if randomList[location]>randomList[positionOfMax]:
```

```
            # If true, the positionOfMax becomes the larger value
            positionOfMax = location

    # Store the value of the current iteration of the input array in the 'temp' variable
    temp = randomList[fillslot]
    # Swap the value in the input list with the value of the larger value
    randomList[fillslot] = randomList[positionOfMax]
    # Make the index position of the max value equal to the current iteration of the input array
    randomList[positionOfMax] = temp
```

Suitability

Selection sort performs well on a small list, and does not require additional memory so is therefore useful for use on processors that have low RAM. It is unsuitable however for large input sizes, as it's time complexity is too large (as we can see from the graph below after the 4000 input mark).

From the graph below, selection sort appears to be tied with insertion sort in a linear fashion until just before the 4000 input size mark. From that point on the time complexity becomes worse and starts to look more like a quadratic polynomial time complexity. From the 4000 to 10000 input mark, it has a worse time complexity than insertion sort, but not by much.

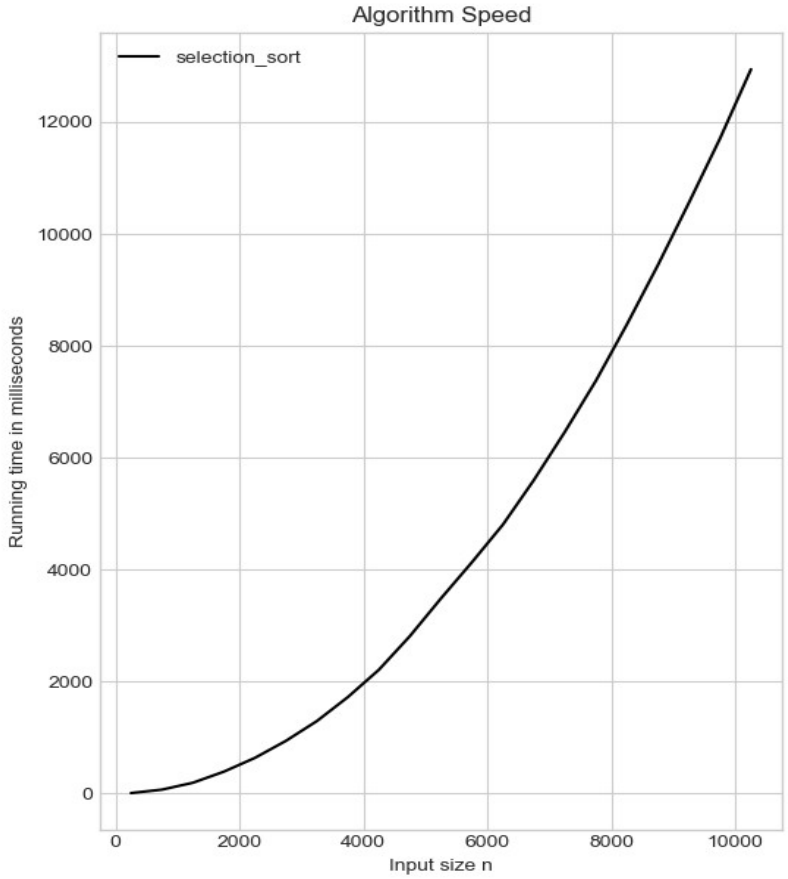

Figure 3

Counting Sort

Counting sort is a non-comparison based integer sorting algorithm. Iits time complexity is O(n+k), and its space complexity is also O(n+k). The space complexity increases when the difference between the keys to be sorted is large. Similar to a hashing function, it counts the number of unique values in the input array and then stores the counts in a separate count array, with each integer value mapped to its respective index. It then accumulatively adds each pair of consecutive values in the count array, shifts them by 1 index, and then store the values in order in an output array. (18)

Code (19)

```
def countingSort(self, randomList):
    # Get the length of the input list
    n = len(randomList)
    # Turn the numpy array into a regular python array
    randomList = randomList.tolist()

    # Initialise the output array elements that will store the sorted array to 0
    output = [0] * (n)

    # Initialize the count array to 0
    count = [0] * (n)
    resultList = [0] * (n)
    resultList2 = []

    # Store the count of unique occurrences from the input array in the count array
    for i in range(0, n):
        index = randomList[i]
        count[ index ] += 1

    # Caccumulatively add consecutive pairs of values in the count array
    for i in range(1,n):
        count[i] += count[i-1]

    # Build the output array – the values in the input array 'randomList' are used to iterate through the indices of the count array. 1 is subtracted from each iteration of count and this is used as the keys for the output array. Each key from the output array is set to the input array value. 1 is then subtracted from each value in the count array.
    for i in range(len(randomList)):
        index = randomList[i]
        output[ count[ index ] - 1] = randomList[i]
        count[ index ] -= 1

    # Copy the output array to arr[], so that arr now contains sorted numbers
    i = 0
    for i in range(0,len(randomList)):
        resultList[i] = output[i]
```

Suitability

Counting sort is a fast efficient algorithm, whose running time is linear. However it is only suitable for use in situations where the values to be sorted are not greater than the number of indices in the input list. Counting sort can be combined with Radix sort to handle larger values more efficiently. (20) The values in the input list must be integers in a range from min to max. Counting sort can also be used to sort integers that store some information with them. (19)

As we can see from the graph below, the running time of counting sort increases as input size increases, in an a linear fashion. O(n) is its best case running time, which is demonstrated below, abeit with minor fluctuations between the 4000 and 8000 input marks. Running time should be linear if none of the integers in the input array go above the maximum size of the array.

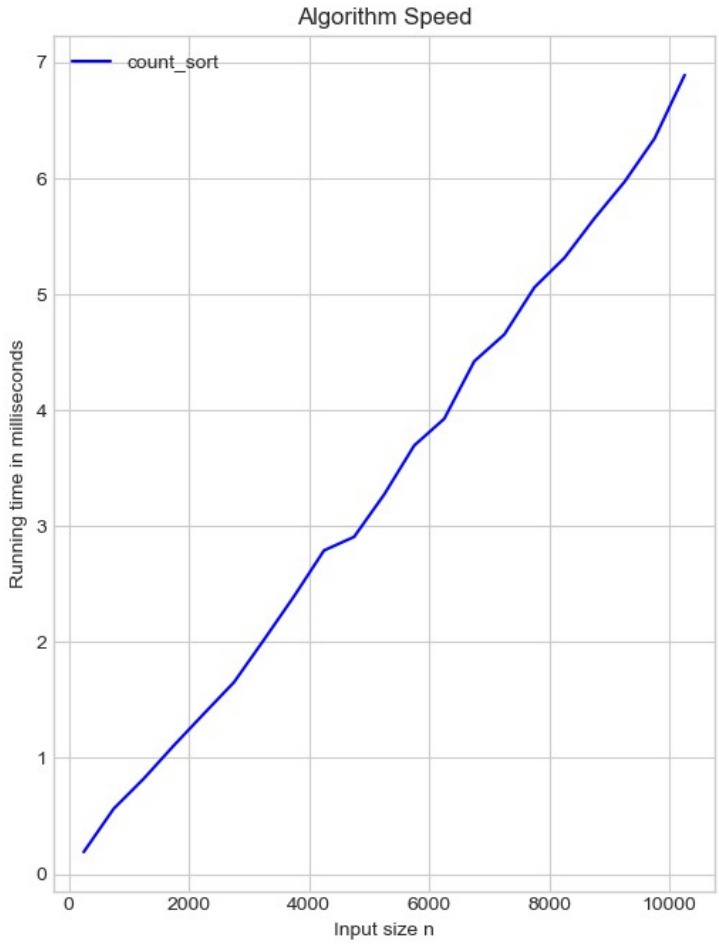

Figure 4

Quick Sort

Quick sort is an efficient comparison sort. It has O(n log n) time complexity in the best and average case, and O(n^2) in the worst case. Quick sort is an in-place algorithm requiring very small amounts of extra memory to perform the sort. Its space complexity is O(log n), even in the worst case. (21) Quick sort is a recursive 'divide and conquer' algorithm, and it operates by creating a partition in the input list. The partition splits the list at a point or 'pivot' and recursively calls itself to 'iterate' through the partitioned sublist from first to last index. At each 'iteration' the values to the left of the pivot value that are larger than the pivot value are swapped with the values to the right of the pivot value that are smaller than the pivot value until the value from the right of the pivot is smaller compared to the value to the left of the pivot (at this point all leftmost values are smaller than the pivot, and all rightmost values are larger). Then the right value is swapped with the pivot value. We now have several groupings or 'sublists' of numbers that have more similar ranges. Another pivot value is then chosen and the process begins again until all values in the sublists are in order. The best choice of pivot value is the index as close as possible to the halfway point of the algorithm to even out the workload.(25)

Because the comparison operations are completed on small sublists, quick sort is able to sort the input list without needing to pass over the entire input array repeatedly carrying out comparison operations on every adjacent value, unlike less the efficient comparison algorithms described above. Hence quick sort is significantly faster than bubble, selection and insertion sorts.

Code (21)

```
def quickSort(self,randomList):

    self.quickSortHelper(randomList,0,len(randomList)-1)

def quickSortHelper(self, randomList,first,last):
   if first<last:
      # Create the first partition by calling the partition function
      splitpoint = self.partition(randomList,first,last)
      # Recursively call the quickSortHelper function to iterate through the list and create another partition
      self.quickSortHelper(randomList,first,splitpoint-1)
      # Recursively call the quickSortHelper function to iterate through the list and create another partition  until the last index of the
input list
      self.quickSortHelper(randomList,splitpoint+1,last)

def partition(self,randomList,first,last):
   # Get the first value of the input list to set the pivot value
   pivotvalue = randomList[first]
   # Set variables for the indexes adjacent to the first index and for the last index
   leftmark = first+1
   rightmark = last

   done = False
   # Create while loop with a boolean condition if not true
   while not done:
      # Create nested while loop with condition that checks if the leftmark is less than or equal to the rightmark
      # and the value of the leftmark index is less than or equal to the pivot value
      while leftmark <= rightmark and randomList[leftmark] <= pivotvalue:
         # If true, increment 'leftmark' by 1
         leftmark = leftmark + 1
      # Create another nested while loop with condition to check if the 'rightmark' or last input list value is greater than or equal to
the pivot value
      # and greater than or equal to the leftmark value
      while randomList[rightmark] >= pivotvalue and rightmark >= leftmark:
         # If true, decrement the last value by 1
         rightmark = rightmark -1
      # Once rightmark iterates down the array and leftmark iterates up the array and they meet, and rightmark is finally less than
leftmark, then the outside while loop ends
      if rightmark < leftmark:
          done = True
   # Until leftmark and rightmark converge swap the values of the leftmark and rightmark index values
   else:
      # Create temporary variable to store the value of the leftmark index position
      temp = randomList[leftmark]
      # Make the leftmark value equal to the rightmark value
      randomList[leftmark] = randomList[rightmark]
      # Make the rightmark value equal to the temp value (or leftmark value)
      randomList[rightmark] = temp

   # Swqp the first value of the list with the rightmark value
   temp = randomList[first]
   randomList[first] = randomList[rightmark]
   randomList[rightmark] = temp

   return rightmark
```

Suitability

Quick sort is highly efficient and has a slow rate of growth, so would be suitable for sorting arrays with a large input size. It as it in an in-place algorithm with a relatively low space complexity, it would be suitable

for use on systems with low memory capacity. However it is not a stable sorting algorithm, so the relative order of sort items is not preserved, and so it could not be used for sorting based on multiple keys. From the graph below (figure 5) it appears as though the running time from my implementation is linear in nature. As its average running time is n log n, I would have expected it to have a slower, linearithmic running time. (21)

However, when comparing quick sort in a graph against counting sort directly (figure 6), we can see that the running time of counting sort is significantly faster (and thus its asymptotic rate of growth is slower).

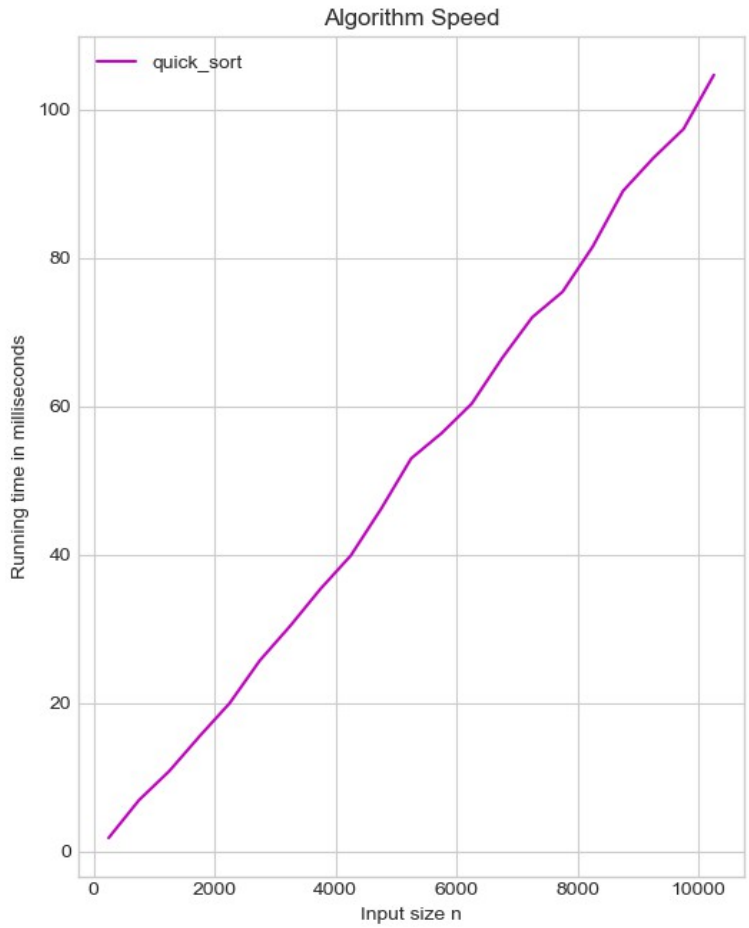

Figure 5

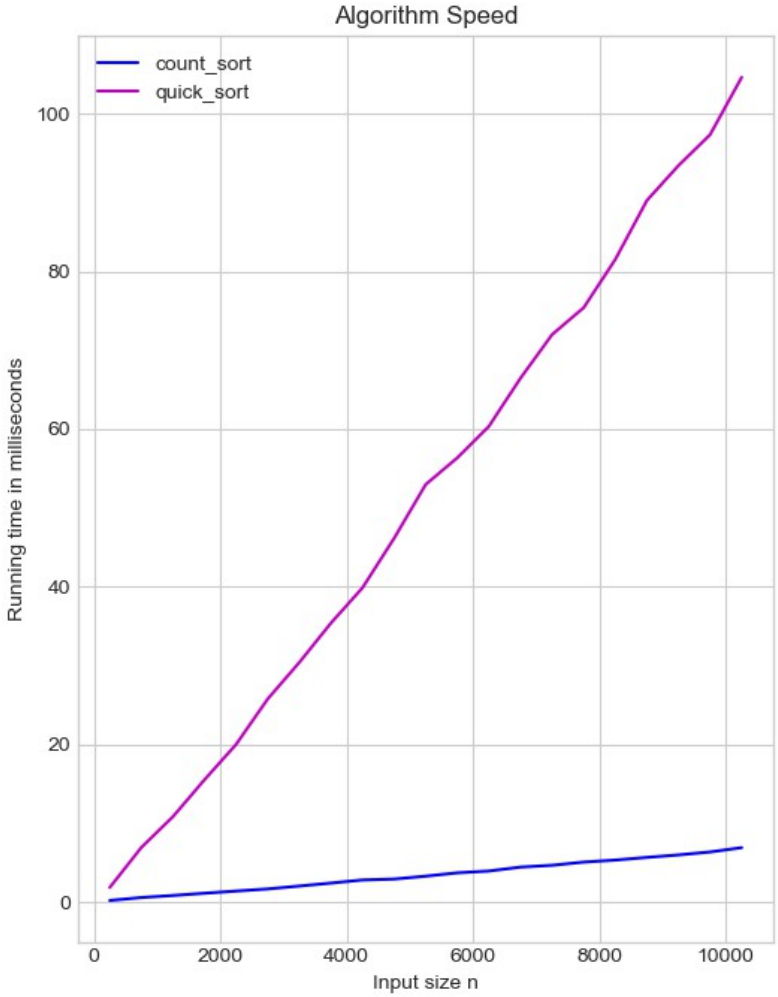
Algorithm Speed
count_sort
quick_sort
Running time in milliseconds
Input size n
0
20
40
60
80
100
0
2000
4000
6000
8000
10000

Figure 6

Implementation & Benchmarking

Implementation

Rather than create a separate script to run each algorithm, I created a class called BenchmarkClass with helper methods to automate as much of the benchmarking implementation as possible.

The sizeLoop method iterates through a range of inputs from 250 to 10500, calls the benchmark method, and then outputs the result of benchmark in a pandas dataframe, which is useful for wrangling data and building tables.

```
def sizeLoop(self,numRuns,maxInputSize,algoType):
    algoSpeedArray = []
    algoSizeArray = []
    minInputSize = 250
    inputIncrement = 500
    countInputSizes = 0

    for listLength in range(minInputSize,maxInputSize,inputIncrement):
        benchmarked = self.benchmark(numRuns,listLength,algoType)
        algoSpeedArray.append(benchmarked)
        algoSizeArray.append(listLength)
        countInputSizes += 1

    df = pd.DataFrame({'Algorithm_type': algoType,
                       'Algorithm_size': algoSizeArray,
                       'Algorithm_speed': algoSpeedArray})
    return df
```

The benchmark method iterates through a list called numRuns (this is an argument for the method). In my code it loops 10 times, calls the randomListGen method, which generated a random list of numbers from 250 to the randomListLength argument, and then records the length of time it takes for the chooseAlgo method to run. It then calculates the average speed of the algorithm, converts it to milliseconds, and rounds the output to 3 decimal places.

```
def benchmark(self,numRuns,randomListLength,algoType):
    outputAggregate = 0
    generatedRandomList = []

    for run in range(numRuns):
        generatedRandomList = self.randomListGen(randomListLength)
        # Get start time in seconds #
        startTime = time.time()
        self.chooseAlgo(algoType,generatedRandomList)
        endTime = time.time()
        # Subtract the start and end times to get the difference #
        timeElapsed = endTime - startTime
        # Sum the speed of each algorithm #
        outputAggregate += timeElapsed
        # Convert seconds to milliseconds and get the average speed of 10 runs
    averageOutput = (outputAggregate/numRuns)*1000
    # Round the speed output values to 3 decimal places to make them more readable #
    averageOutputRound = round(averageOutput,3)
    return averageOutputRound
```

The chooseAlgo method calls the method of the particular algorithm that is in the algoType argument.

```
def chooseAlgo(self,algoType,randomList):
```

```
    if algoType == 'bubble_sort':
        return self.bubbleSort(randomList)
    elif algoType == 'selection_sort':
        return self.selectionSort(randomList)
    elif algoType == 'insertion_sort':
        return self.insertionSort(randomList)
    elif algoType == 'count_sort':
        return self.countingSort(randomList)
    elif algoType == 'quick_sort':
        return self.quickSort(randomList)
```

The code for the lgorithms themselves is outlined in the second section of this document. Each algorithm method calls the randomListGen method, which generates a numpy array of random numbers, based on the inputSize argument.

```
def randomListGen(self,inputSize):
    return np.random.randint(inputSize, size=inputSize)
```

The loopThroughAlgorithms method then calls the sizeLoop method and iterates through the array of algorithms and returns a dataframe with all of the time complexity information about all of the algorithms.

```
def loopThroughAlgorithms(self,numRuns,maxInputSize,algoArray):
    collectiveOutputDf = pd.DataFrame()
    for algoType in algoArray:
        algoOutput = self.sizeLoop(numRuns,maxInputSize,algoType)
        collectiveOutputDf['algorithm_size'] = algoOutput['Algorithm_size']
        collectiveOutputDf[algoType] = algoOutput['Algorithm_speed']
    return collectiveOutputDf
```

The method then uses the output returned from loopThroughAlgorithms to generate a matplotlib graph including each of the algorithms time complexity over the amount of input values. (23)(24)

```
def plotOutput(self,algo_speed_results_df):
    plt.style.use('seaborn-whitegrid')
    plt.title("Algorithm Speed")
    plt.xlabel("Input size n")
    plt.ylabel("Running time in milliseconds")
    plt.plot(algo_speed_results_df.algorithm_size, algo_speed_results_df.bubble_sort, '-c')
    plt.plot(algo_speed_results_df.algorithm_size, algo_speed_results_df.selection_sort, '-k')
    plt.plot(algo_speed_results_df.algorithm_size, algo_speed_results_df.insertion_sort,'-r')
    plt.plot(algo_speed_results_df.algorithm_size, algo_speed_results_df.count_sort, '-b')
    plt.plot(algo_speed_results_df.algorithm_size, algo_speed_results_df.quick_sort,'-m')
    plt.legend()
    plt.show()
```

Below is the code to set the input size of the array, which will have random values generated for it. You can also set the number of runs wanted and add in the selection of algorithms chosen to benchmark against.

```
input_size = 30000
num_runs = 10
algo_array = ['bubble_sort','selection_sort','insertion_sort','count_sort','quick_sort']
```

The method loopThroughAlgorithms is called and the results are printed and exported to a CSV file for persistent storage. The plotOutput method is also called, which renders the graph of the running times of each of the algorithms.

```
benchmark1 = BenchmarkClass()
algo_speed_results_df = benchmark1.loopThroughAlgorithms(num_runs,input_size,algo_array)
```

```
print(algo_speed_results_df)
export_csv = algo_speed_results_df.to_csv (r'/Users/joanhealy1/Documents/GMIT-algorithms/project/algorithm-speeds/combined_algo_speeds-5.csv', index = None, header=True)
benchmark1.plotOutput(algo_speed_results_df)
```

Below is the typical CLI output (figure 7) from run 6 through 9 of the insertion sort algorithm:

```
Run number:  6
random list length in benchmark:  23750
random list benchmark() :  [ 5487 16481 14628  7351 18962 12847  6740  5986  2187 16107]
chosen algorithm:  insertion_sort
insertionsort input list length  23750
time elapsed 78.22104811668396
output aggregate 549.4008929729462


Run number:  7
random list length in benchmark:  23750
random list benchmark() :  [ 5941 13143  5153 14181  2389  2972 12786  9207 22115  3926]
chosen algorithm:  insertion_sort
insertionsort input list length  23750
time elapsed 78.56405401229858
output aggregate 627.9649469852448


Run number:  8
random list length in benchmark:  23750
random list benchmark() :  [ 7501 23216 10396  9910 14660 13895  7034  3144  6125  1789]
chosen algorithm:  insertion_sort
insertionsort input list length  23750
time elapsed 77.40641808509827
output aggregate 705.371365070343


Run number:  9
random list length in benchmark:  23750
random list benchmark() :  [ 4821 21595 11571  4484  3716  8944  2931 17059 13286 15918]
chosen algorithm:  insertion_sort
insertionsort input list length  23750
time elapsed 78.30114698410034
output aggregate 783.6725120544434


average_output  78367.251
countInputSizes  48
```

Figure 7

Benchmarking Results

Below is a table of results (figure 8) collected on the performance of the 5 algorithms measured. Algorithm_size is the number of inputs in the input list. Count sort is the fastest and lowest time complexity by a significant margin. In second place is the efficient comparison sort quick sort, and then the 3 quadratic simple comparison sorts in order of increasing time complexity; selection sort, insertion sort and bubble sort, with bubble sort being over twice as slow as insertion sort.

At the minimum input size count sort just took 0.189 of a millisecond, whereas, bubble sort took 18.678 ms. At the maximum input size they take 6.892 and 31428.247 ms respectively, which means that bubble sort is ~4560 times slower at the maximum input size. The min and max time complexities for selection and insertion sort are almost identical. The time for quick sort at the maximum input size is 104.646 ms, which is ~15 times slower than counting sort, but it is still ~ 123 times faster than its nearest rival selection sort.

algorithm_size	selection_sort	insertion_sort	bubble_sort	count_sort	quick_sort
250	7.452	8.637	18.678	0.189	1.83
750	68.966	77.679	165.644	0.563	6.91
1250	192.185	220.036	459.543	0.822	10.82
1750	390.85	429.287	895.504	1.108	15.511
2250	636.816	714.751	1479.275	1.382	19.996
2750	941.3	1066.92	2244.73	1.653	25.761
3250	1295.213	1498.55	3093.648	2.02	30.395
3750	1724.545	2018.855	4112.858	2.395	35.385
4250	2212.574	2580.347	5305.2	2.791	39.868
4750	2814.297	3203.67	6633.997	2.908	46.147
5250	3486.706	3917.492	8100.659	3.273	52.977
5750	4133.493	4719.074	9702.29	3.696	56.361
6250	4806.734	5576.921	11463.264	3.927	60.356
6750	5604.453	6496.843	13395.223	4.423	66.466
7250	6466.788	7498.914	15468.468	4.654	71.999
7750	7378.792	8537.184	17670.446	5.06	75.416
8250	8385.86	9696.099	20028.067	5.315	81.532
8750	9445.681	10920.094	22457.02	5.656	89.017
9250	10560.095	12178.259	25124.571	5.971	93.447
9750	11704.502	13258.939	27907.173	6.345	97.366
10250	12942.809	15491.801	31428.247	6.892	104.646

Figure 8

In figure 9, we can see the average length of time to run each of the algorithms, plotted against the size of each input, in increasing order. As expected, bubble sort shows the greatest increase in performance time as input size increases. It has quadratic time complexity, so the curve of the line corresponds to the n^2 line in the reference graph (figure 10). Even though insertion sort and selection sort also have quadratic time complexity, their curves appear to follow a more n log n time complexity, when compared to the reference graph and to the curve of bubble sort.

Both counting sort and quick sort have such small time complexities that when graphed alongside the simple comparison sorts they appear to take no time at all to complete. For this reason I have separated them into another graph (figure 11) to show their relative time complexities. On this smaller scale graph, we can see that quick sort has a linear or O(n) time complexity, however it is supposed to only have n log n in the best case.

Similarly couningt sort appears to have log n time complexity as demonstrated on this graph, when it actually should have linear time complexity, so again the algorithm appears to perform faster in my implementation.

Even though each of the algorithms implemented appear to have slightly faster run times than expected, their relative positions to each other are exactly what we would expect from them. In conclusion, unless I have an extremely fast computer processor, there may just be something about the graphing mechanism that is slightly skewing the plots. The only alternate explanation is that the numpy random number generator wasn't very effective in generating randomly ordered lists of input arrays, and as such the algorithms might have had faster run times given that the sorting performed was 'easier' for them to compute.

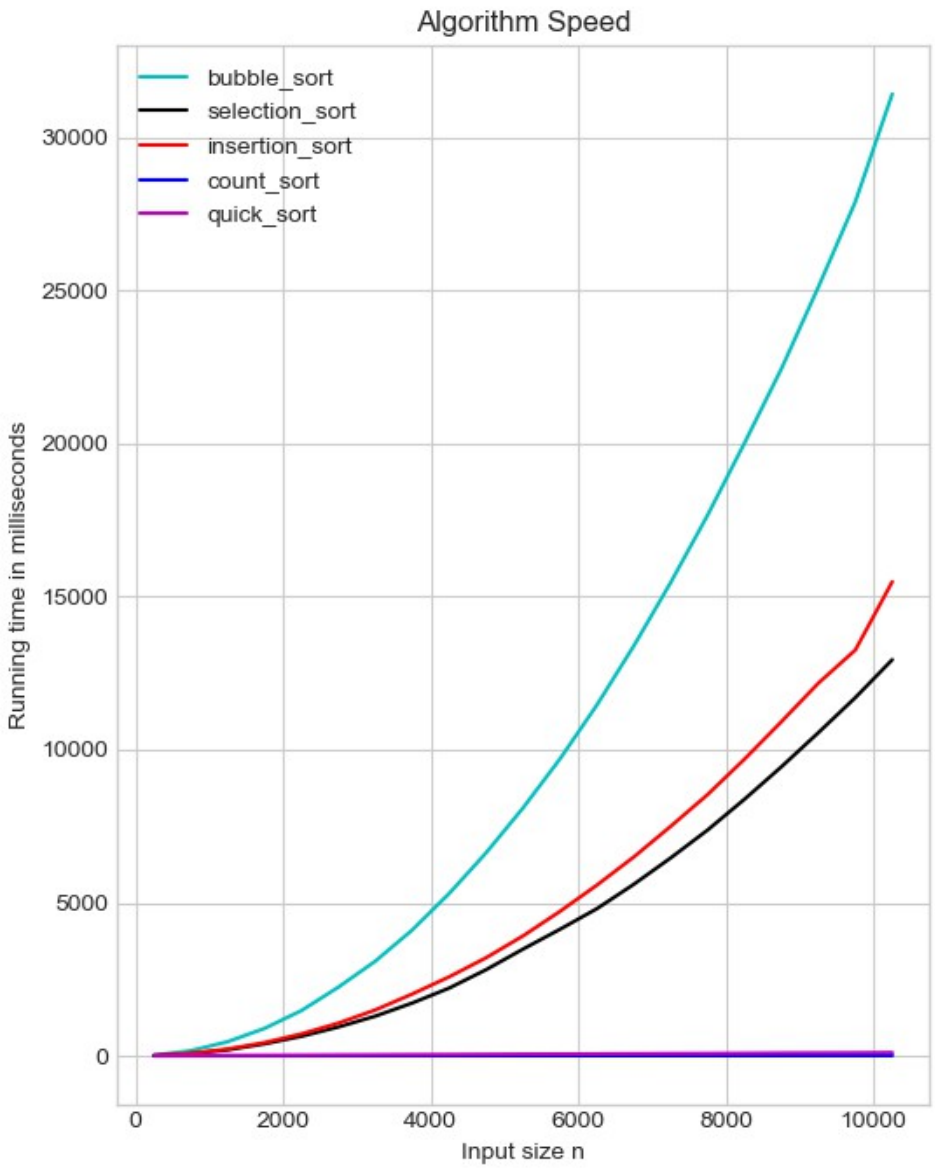
Algorithm Speed
bubble_sort
selection_sort
insertion_sort
count_sort
quick_sort
Running time in milliseconds
Input size n
0
5000
10000
15000
20000
25000
30000
0
2000
4000
6000
8000
10000

Figure 9

Running time $T(n)$

is proportional to:	Complexity:
$T(n) \propto \log n$	logarithmic
$T(n) \propto n$	linear
$T(n) \propto n \log n$	linearithmic
$T(n) \propto n^2$	quadratic
$T(n) \propto n^3$	cubic
$T(n) \propto n^k$	polynomial
$T(n) \propto 2^n$	exponential
$T(n) \propto k^n;\ k > 1$	exponential

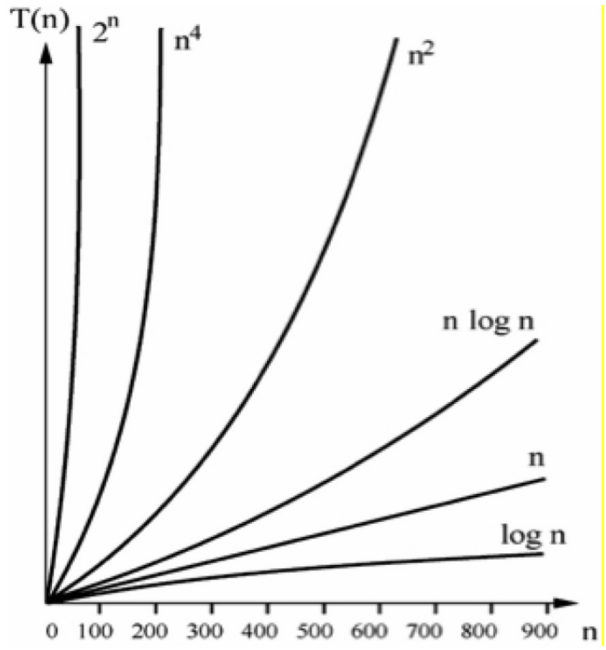

Figure 10

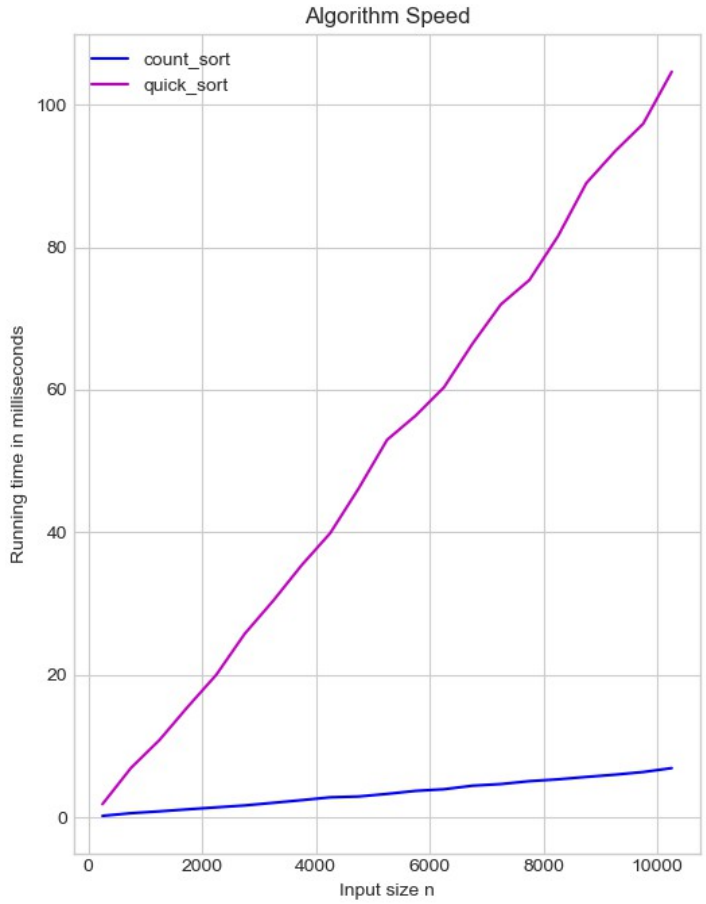

Figure 11

Bibliography

1. M. Sipser, Introduction to the theory of computation, Boston : Thomson Course Technology, 2010, pp.
2. Khan Academy, accessed 10th April 2019, <https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/big-o-notation>
3. Khan Academy, accessed 10th April 2019, <https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/big-big-omega-notation>
4. Khan Academy, accessed 10th April 2019, <https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/big-big-theta-notation>
5. Khan Academy, accessed 10th April 2019, <https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/functions-in-asymptotic-notation>
6. Geeks for Geeks, accessed 28 April 2019, <https://www.geeksforgeeks.org/g-fact-86/>
7. Computer Science, accessed 16 April 2019, <https://www.youtube.com/watch?v=bNjMFtHLioA>
8. Tutorials Point, accessed 1 May 2019, <https://www.tutorialspoint.com/data_structures_algorithms/sorting_algorithms.htm>
9. Saint Mary's University, accessed 20th April, <http://cs.smu.ca/~porter/csc/common_341_342/notes/sorts_stable.html>
10. Stack Overflow, accessed 26th April 2019, <https://stackoverflow.com/questions/1517793/what-is-stability-in-sorting-algorithms-and-why-is-it-important>
11. Geeks for Geeks, accessed 28 April 2019, <https://www.geeksforgeeks.org/comparator-function-of-qsort-in-c/>
12. Open Data structures, accessed 2 May 2019, <https://opendatastructures.org/ods-java/11_1_Comparison_Based_Sorti.html>
13. Sciencing.com, accessed 17th April 2019, <https://sciencing.com/the-advantages-disadvantages-of-sorting-algorithms-12749529.html>
14. Geeks for Geeks, accessed 28 April 2019, <https://www.geeksforgeeks.org/sorting-algorithms/>
15. Interactive Python, accessed 5 April 2019, <https://interactivepython.org/runestone/static/pythonds/SortSearch/TheBubbleSort.html?highlight=bubblesort>
16. Interactive Python, accessed 5 April 2019, <http://interactivepython.org/courselib/static/pythonds/SortSearch/TheInsertionSort.html>
17. Interactive Python, accessed 5 April 2019, <https://interactivepython.org/runestone/static/pythonds/SortSearch/TheSelectionSort.html>
18. Base CS, accessed 29 April 2019, <https://medium.com/basecs/counting-linearly-with-counting-sort-cd8516ae09b3>
19. Geeks for Geeks, accessed 29 April 2019, <https://www.geeksforgeeks.org/counting-sort/>
20. University of Wisconsin Madison, accessed 5 May 2019, <http://pages.cs.wisc.edu/~paton/readings/Old/fall08/LINEAR-SORTS.html>
21. Interactive Python, accessed 5 April 2019,<https://interactivepython.org/runestone/static/pythonds/SortSearch/TheQuickSort.html?highlight=quick%20sort>
22. P. Mannion, “Sorting Algorithms Part 1”, pp. 16, 2019.
23. Python Data Science Handbook, accessed 1 May 2019, <https://jakevdp.github.io/PythonDataScienceHandbook/04.01-simple-line-plots.html>
24. Matplotlib.org, accessed 1 May 2019, <https://matplotlib.org/users/pyplot_tutorial.html>
25. Quick sort in 4 minutes, accessed 2 May 2019, <https://www.youtube.com/watch?v=Hoixgm4-P4M>