Computational Algorithms Sorting Algorithms Project.

By Aidan Summerville – g00302722

Introduction

What is a sorting algorithm? “A sorting algorithm is a method for reorganizing a large number of items into a specific order, such as alphabetical, highest-to-lowest value or shortest-to-longest distance. Sorting algorithms take lists of items as input data, perform specific operations on those lists and deliver ordered arrays as output” (Hawking, 2000). Sorting algorithms rearrange input lists into a certain order. The most common rearrangement methods are numerical and lexicographical in ascending or descending order. Sorting algorithms is one of the most studied aspects of computer science because of its base use across many other problems and the amount of bandwidth it can take up on a computer (Blelloch 1996).

Computational complexity

Computational complexity in terms of algorithms in computer science is also known as time complexity and is the given time an algorithm will take to complete. Complexity measures an algorithm’s efficiency with respect to internal factors, such as the time needed to run an algorithm. (Patrick Mannion “Analysing Algorithms Part 1”2019). Complexity is measured by big O notation. Big O notation refers to order of magnitude. , Big O notation issued to classify algorithms by their running times grouped by their input value.

Performance

Computational performance affects the speed at which an algorithm will complete that is not connected to the efficiency of the algorithmic code. The same code would run at vastly different speeds and different machines. However the relation of the results to one another should be similar and while the results on different machines differ a clear relation should be seen when the results are plotted. The performance is affected by the computers specifications and operating system. Parts of the system that will have an effect on the computer's ability to complete the algorithm in as quick as time as possible include the CPU(how many cores , clock speed), the amount of ram the system has, the type of internal memory the system uses(HDD, SSD, nand SSD, PCIE SSD), the graphics card, the cooling system(fan, liquid cooled), the case(cooling), the operating system and how many programs are installed or are already running on the background of the system. (Patrick Mannion "Analysing Algorithms Part 1"2019).

In place sorting

An in place sorting algorithm is an algorithm that does not need any extras pace in order to output the sorted algorithm, other than a small space for its operation. These algorithms do not create separate arrays to sort the outputs and the space remains constant within the array.

Examples of in place sorting: Bubble sort, Selection Sort, Insertion Sort, Heapsort.

Stable sorting

"A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted." (GeeksforGeeks, 2019) This means an algorithm is stable if two elements with equal ordering keys appear in an array. The element that appears first initially will appear first in the output array. This kind of sorting algorithm has more use in sorting words (names, places etc) or integers with other pieces off data attached. It has no effect on the outcome of integer lists on their own that I will be sorting in this project as stability is not an issue for integer arrays.

Examples of stable sorting algorithms: Insertion Sort, Merge Sort, Bubble Sort, Tim Sort, Counting Sort.

Unstable sorting

Unstable sorting algorithms do not take into account the original position of two elements with equal sorting keys into account in the output array. These algorithms can output elements off matching sorting keys in an array in any order. Because of this they are often faster than there stable counterparts. Those are of the same Big O notation. These algorithms work well with integer arrays. But are unsuited to most strings.

Examples of unstable sorting algorithms: Heap Sort, Selection sort, Shell sort, Quick Sort.

Examples of stable and unstable sorting algorithms

Comparator functions

Comparator functions are used to compare integer or string values in python. The comparator function is "cmp()". It can be used to see if numbers are greater than ">", less than "<" or equal to "=" each other. Instead of returning true or false it then assigns an -1 the number is less than the number it is being compared against, 0 if it I equal and 1 if it is greater. The numbers can then be rearranged by the theses values as the algorithm cycles through all the numbers in an array to produce a sorted array. Strings can be compared by the function in a similar way by associating numbers with the ascending order of the alphabet (Includehelp.com, 2019).

Comparison-based sorts

A sorting algorithm is comparison based if it uses the comparator functions discussed above. The less than, equal to and greater than (<, =, >)operator to determine the output of the sorting algorithm, these can be collectively called a three way comparison . If the algorithms are stable and two elements are equal the element that appears first will appear in the output array first. If the algorithm is not stable the equal elements may appear in any order. (GeeksforGeeks, 2019)

Examples of Comparison based sorts: Quicksort, Heapsort, Shellsort, Merge sort.

Sorting Algorithms

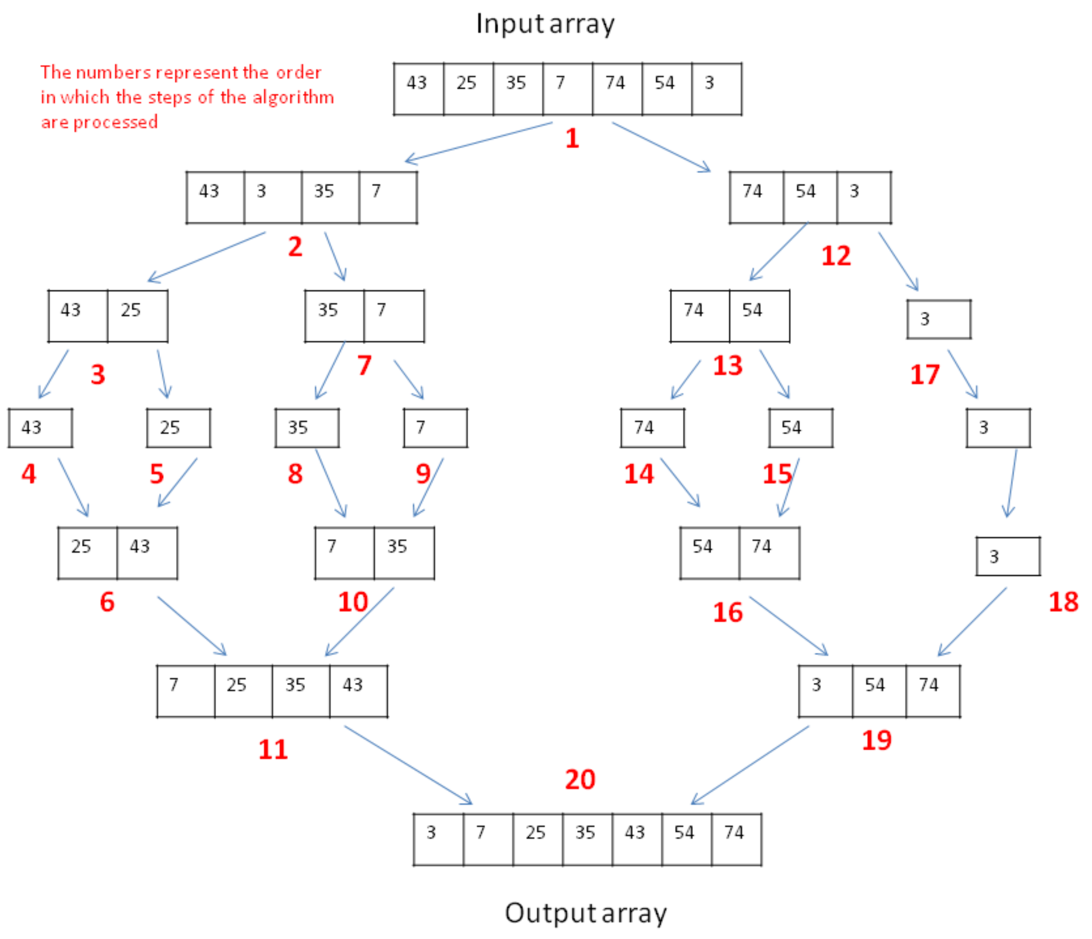
Merge sort

Merge sort is a divide and conquer algorithm (GeeksforGeeks, 2019). It divides arrays into too distinct pieces and continues to divide and split pieces until all parts of the array are individualised. If the split occurs across an array of uneven length the split will take place with the larger side being on the right (by one).

Merge sort is a stable array with its best case, worst case, and average case for completion being the same. Mergesorts time complexity is nlogn for best worst and average completion case. This would mean when graphed that the time for the algorithm to complete would increase evenly in proportion to n (n=the input size of the array.)

The diagram below shows how the mergesort sorting algorithm works. It takes the initial input array and splits the arrays recursively until they are individual numbers. It then joins the numbers back together in smaller sorted arrays. This is repeated across all the individual numbers until a number of sorted smaller arrays exist. These smaller sorted arrays are merged together into increasingly larger arrays until all the numbers are together in a final outputted sorted array.

The diagram below depicts the how the Merge sort algorithm works.

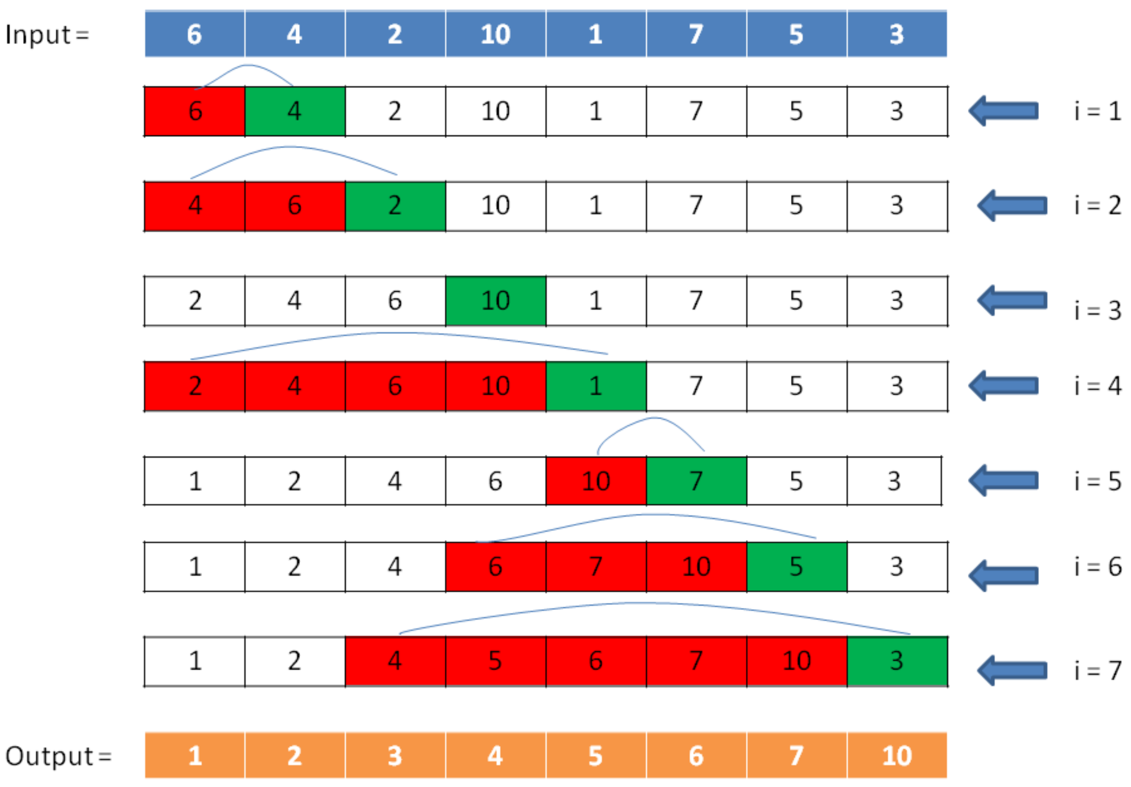


Insertion sort

Insertion sort is a stable sorting algorithm. It works by sorting individual integers at a time. Because of this it is inefficient and slow as the input the size of the array increases. However it is efficient at sorting datasets of low input sizes. Insertion sort is an in place algorithm meaning it doesn't use any more space than the initial input to sort the array.

Insertion sort sorts the input array by selecting the second value in the input array. It then compares this value to the value before it. If this value is greater than the selected value it moves the selected value into this position and the value before moves back one position. This is then repeated until the selected value becomes the first value or it meets a value that is less than itself. The algorithm then moves on to the next value in the input array and repeats the process until it reaches the final input value once this final selected value is sorted the algorithm outputs the array.

The diagram below depicts the how the insertion sort algorithm works.



Insertion sort code

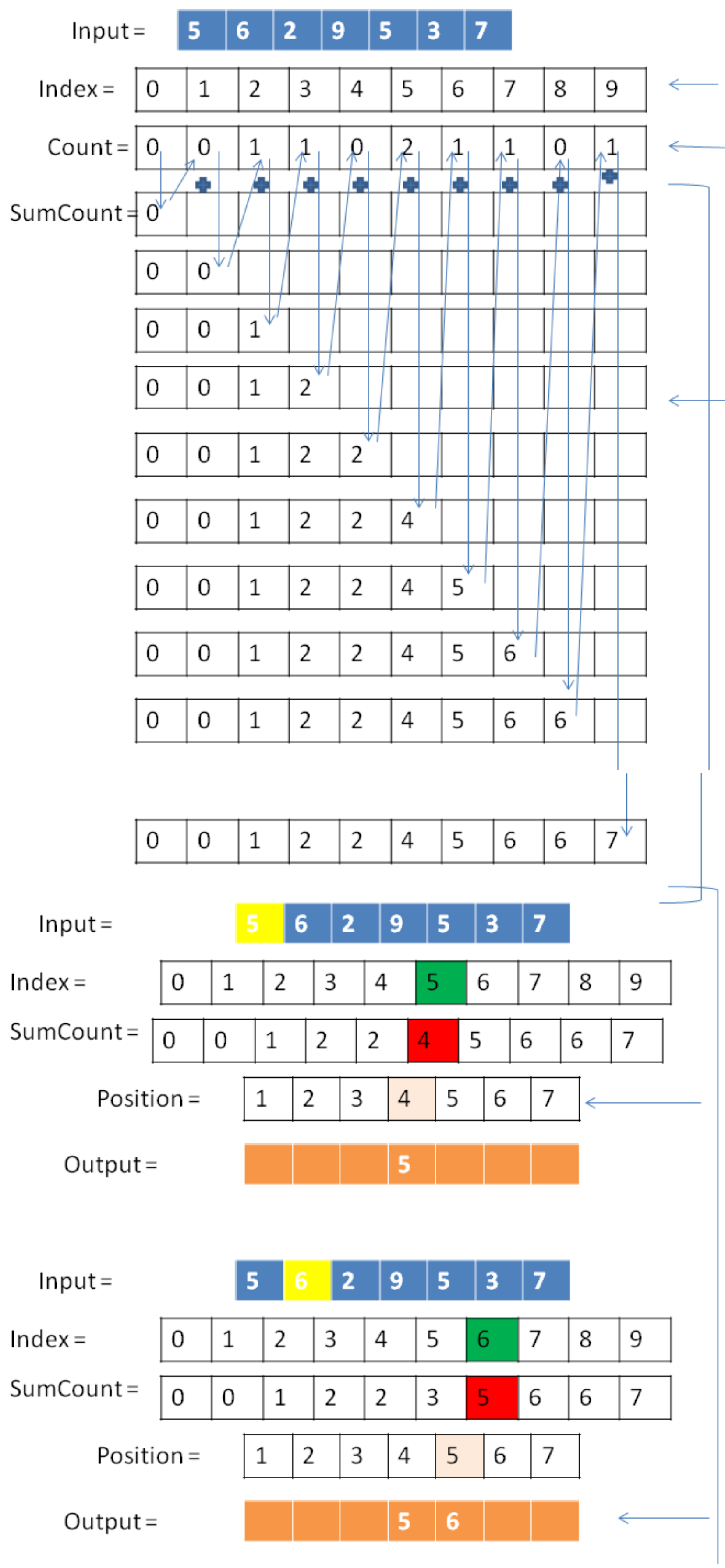
```
## code taken from https://www.geeksforgeeks.org/insertion-sort/
## own comments added.


##Insertion sort Funtion
def insertionSort(arr):

   ##Starts at the 2nd position within the array and iterates through until last position withon the array
    for i in range(1, len(arr)):


        ##Selects the integer within the array at point i intially the 2nd position and sets it as the key
        key = arr[i]


        ##sets j to the position before the key
        j = i-1
        ##while j is greater than or = to 0  and key is less than j
        while j >= 0 and key < arr[j] :
               ## j moves to next position in the array if key is less than j
                arr[j + 1] = arr[j]
                j -= 1
        ## key is now in the next position in the array
        arr[j + 1] = key
```

Count sort

“Counting sort is a sorting technique based on keys between a specific range.” (GeeksforGeeks, 2019). Counting sort has a constant time complexity of n+k where k is the max range of the array and n is the input size. This is the same across best, average and worst cases. Count sort is a stable algorithm. The completion time is linear and should not increase greatly as input size increases. The array must be positive integers and be within the known range of (K). The max range of the algorithm set by the user.

Counting sort works by indexing the input array. Creating an array that goes from 0 to max range of the array (k). It then counts the number of times each integer appears within the input array and adds that number to the count array. Any integers that don't appear in the range are set to 0 in the count array. Once the count array has completed the sumcount array is created it adds the sum of all previous counts including the current count in the array for each integer position and adds them to a new array sumcount. Each position in the sumcount array is related to the count array and the index array for that position.   The algorithm then takes the first integer with the original array and finds its corresponding index and sumcount number. The sumcount number is equal to the position that number will take in the final sorted array. After this the sumcount number for this position is minuses by 1. The algorithm then move on to the next position in the original array and continue this process until all parts of the original array are sorted. It then outputs the sorted array

The diagrams below depict how the Count sort algorithm works.



The index is the range from 0 to the max value within the array

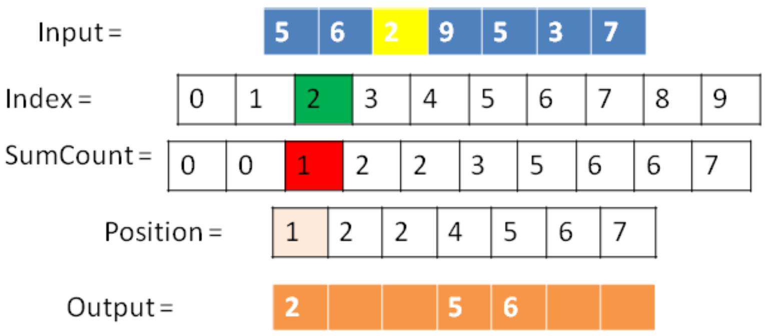
The count is the number of times a given number appears in the array

SumCount adds the sum of the previous positions in the count array and adds it to the current position within the array and places the result in the sumcount array.

Once the input , index Sumcount arrays are created . Two new arrays are created .

The position array which holds the positional information for the output array and is equal the length of the inputs array and arranged by ascending order from 1 to max length.

The output array holds the sorted result of the algorithm



Input =	5	6	2	9	5	3	7			
Index =	0	1	2	3	4	5	6	7	8	9
SumCount =	0	0	1	2	2	3	5	6	6	7
Position =	1	2	2	4	5	6	7			
Output =	2			5	6					

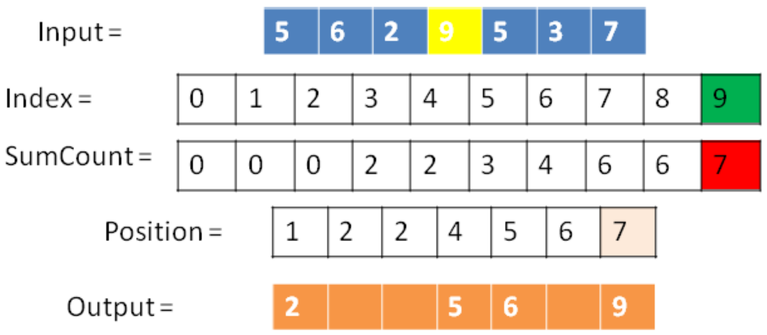
The algorithm selects the first number within the input array.

It then finds this integer in the index array

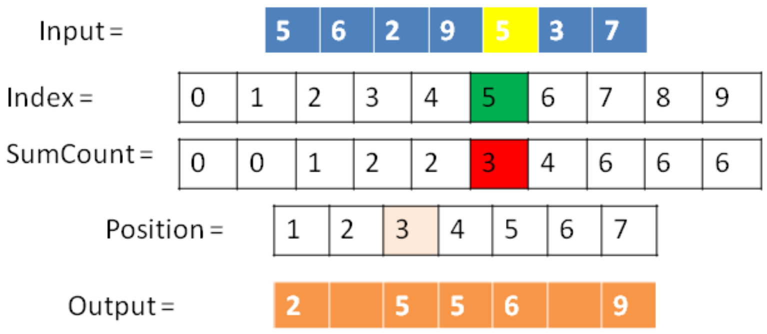
The integer in the index array is compared to its associated number in the SumCount array

This number in the Sumcount array is equal to the position the output array.

The input number is then placed in this location in the output array

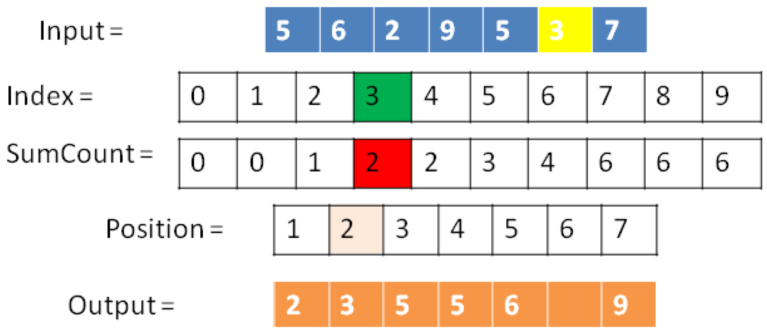


Input =	5	6	2	9	5	3	7			
Index =	0	1	2	3	4	5	6	7	8	9
SumCount =	0	0	0	2	2	3	4	6	6	7
Position =	1	2	2	4	5	6	7			
Output =	2			5	6		9			



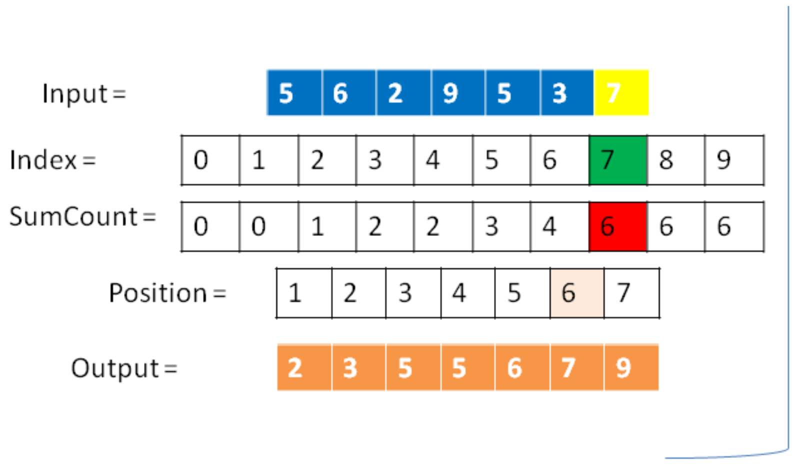
Input =	5	6	2	9	5	3	7			
Index =	0	1	2	3	4	5	6	7	8	9
SumCount =	0	0	1	2	2	3	4	6	6	6
Position =	1	2	3	4	5	6	7			
Output =	2		5	5	6		9			

The number that was corresponding the input number in the sumcount array is then decreased by one.



Input =	5	6	2	9	5	3	7			
Index =	0	1	2	3	4	5	6	7	8	9
SumCount =	0	0	1	2	2	3	4	6	6	6
Position =	1	2	3	4	5	6	7			
Output =	2	3	5	5	6		9			

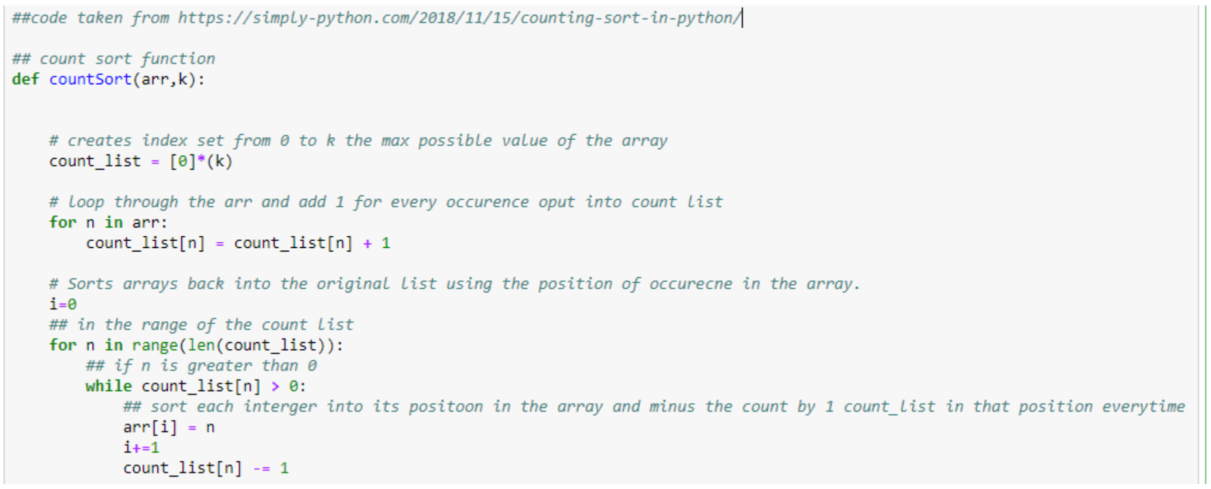
This process is repeated to until the input list has been iterated through



Final Output =
2 3 5 5 6 7 9

Once completed the final sorted array is then outputted

Count Sort Code



```
##code taken from https://simply-python.com/2018/11/15/counting-sort-in-python/

## count sort function
def countSort(arr,k):


    # creates index set from 0 to k the max possible value of the array
    count_list = [0]*(k)

    # loop through the arr and add 1 for every occurence oput into count list
    for n in arr:
        count_list[n] = count_list[n] + 1

    # Sorts arrays back into the original list using the position of occurecne in the array.
    i=0
    ## in the range of the count list
    for n in range(len(count_list)):
        ## if n is greater than 0
        while count_list[n] > 0:
            ## sort each interger into its positoon in the array and minus the count by 1 count_list in that position everytime
            arr[i] = n
            i+=1
            count_list[n] -= 1
```

Bubble sort

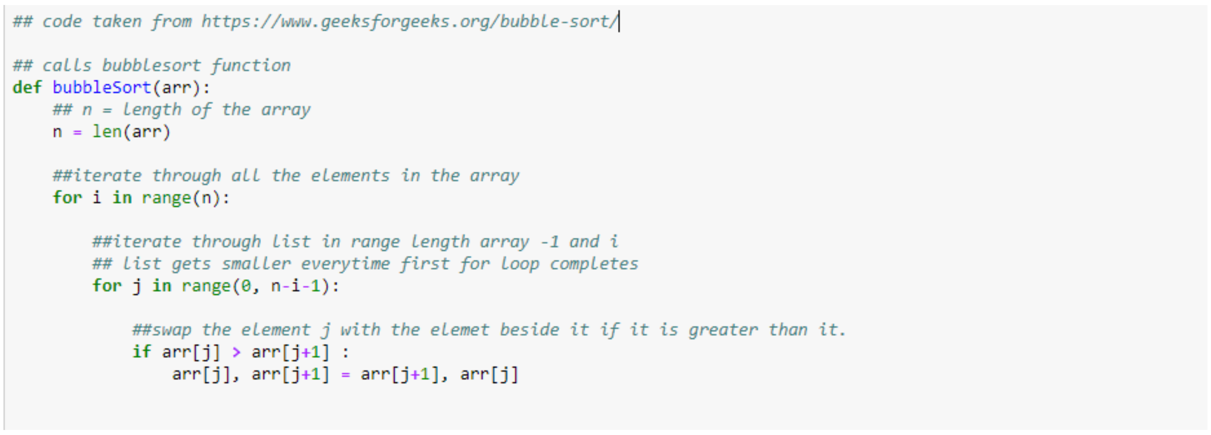
Bubble sort works by selecting the first element in the input array and swapping it with the element next to it if it is greater than that element. It then selects the next element in the array and repeats the action for length of the array -1 –i times. i is the number of times the list has been full passed through. Each time the list is full passed through from one end to another. The length of the list that will be sorted is decreased by 1 as the number in the last position each time it is iterated through is where it will be when it is sorted.

Bubble sort is a stable sort with a best case time complexity of n and a worst case and average case time complexity of n². Bubble sort is a very simple and slow sort. However if the array is already sorted bubble sort will detect this and complete in its best case n time complexity.

The diagrams below depict how the Bubble sort algorithm works.

i= 0	j	0	1	2	3	4	5	6	7
Input	0	3	1	6	7	2	9	4	5
	1	1	3	6	7	2	9	4	5
	2	1	3	6	7	2	9	4	5
	3	1	3	6	7	2	9	4	5
	4	1	3	6	2	7	9	4	5
	5	1	3	6	2	7	9	4	5
	6	1	3	6	2	7	4	9	5
i=1	0	1	3	6	2	7	4	5	**9**
	1	1	3	6	2	7	4	5	
	2	1	3	6	2	7	4	5	
	3	1	3	2	6	7	4	5	
	4	1	3	2	6	7	4	5	
	5	1	3	2	6	4	7	5	
i=2	0	1	3	2	6	4	5	**7**	
	1	1	3	2	6	4	5		
	2	1	2	3	6	4	5		
	3	1	2	3	6	4	5		
	4	1	2	3	4	6	5		
i=3	0	1	2	3	4	5	**6**		
	1	1	2	3	4	5			
	2	1	2	3	4	5			
	3	1	2	3	4	5			
i=4	0	1	2	3	4	**5**			
	1	1	2	3	4				
	2	1	2	3	4				
i=5	0	1	2	3	**4**				
	1	1	2	3					
i=6	0	1	2	**3**					

		1	2						
Output	=	1	2	3	4	5	6	7	9



```python
## code taken from https://www.geeksforgeeks.org/bubble-sort/

## calls bubblesort function
def bubbleSort(arr):
    ## n = length of the array
    n = len(arr)

    ##iterate through all the elements in the array
    for i in range(n):

        ##iterate through list in range length array -1 and i
        ## list gets smaller everytime first for loop completes
        for j in range(0, n-i-1):

            ##swap the element j with the elemet beside it if it is greater than it.
            if arr[j] > arr[j+1] :
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

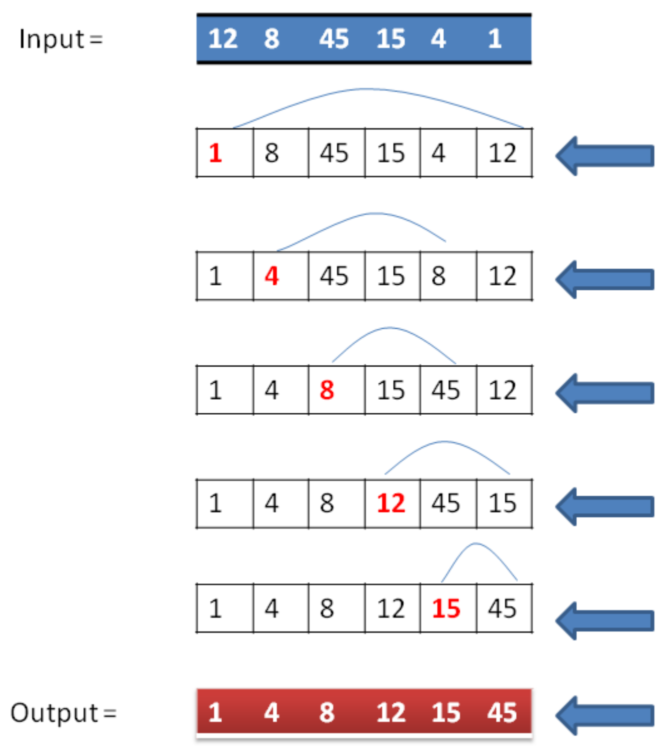
Selection sort

Selection sort is an in place sorting algorithm it works by finding the lowest number in an array and swapping that position with the number in the first position of the array. This will be the lowest numbers final position in the output array. It moves the starting point of i forward one position making the array 1 position shorter and sorts the array again. It continues to do this until it is unable to move forward and then output the sorted array.

The algorithm can work by either finding the max or minimum entry in the array and then swapping the max or minimums position so it is either at the beginning or the end of the array and then working its way up or down the array. Both implementations of the algorithm are equally efficient and give the same result

The time complexity for selection sort is at best, worst and average case n^2. This means it is in efficient at large input sizes. Selection sort is an unstable sorting algorithm.

The diagrams below depict how the Selection sort algorithm works.



Find the lowest number in the array [0→ 5]
And swap the lowest number and place at the beginning of the array.

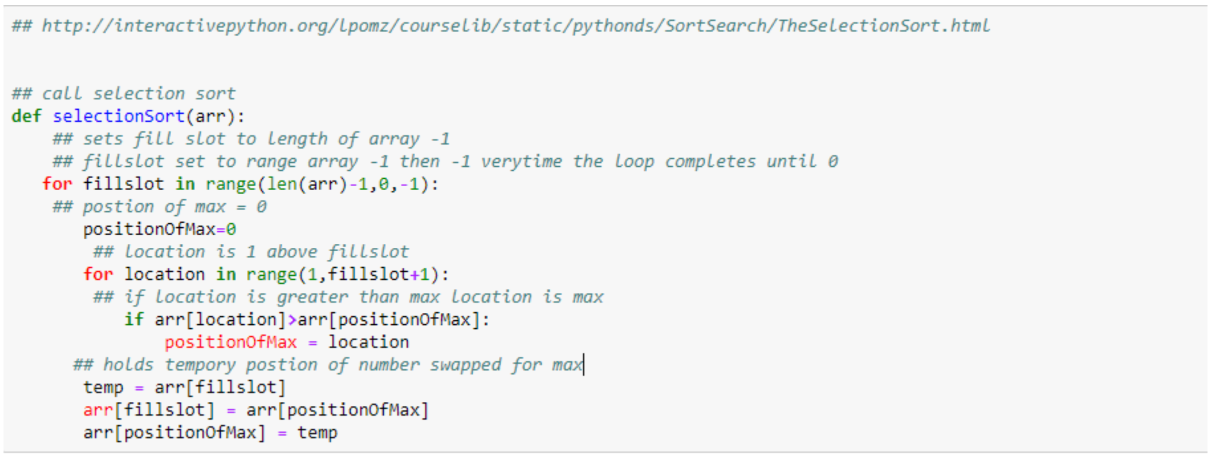
Find the lowest number in the array [1→ 5]
And swap the lowest number and place at the beginning of the array [1→ 5]

Find the lowest number in the array [2→5]
And swap the lowest number and place at the beginning of the array [2→ 5].

Find the lowest number in the array [3→ 5]
And swap the lowest number and place at the beginning of the array [3→ 5].

Find the lowest number in the array [4→ 5]
And swap the lowest number and place at the beginning of the array [4→ 5].

Array completes at and all integers are in order.



```
## http://interactivepython.org/lpomz/courselib/static/pythonds/SortSearch/TheSelectionSort.html


## call selection sort
def selectionSort(arr):
    ## sets fill slot to length of array -1
    ## fillslot set to range array -1 then -1 verytime the loop completes until 0
   for fillslot in range(len(arr)-1,0,-1):
    ## postion of max = 0
       positionOfMax=0
        ## location is 1 above fillslot
       for location in range(1,fillslot+1):
        ## if location is greater than max location is max
           if arr[location]>arr[positionOfMax]:
               positionOfMax = location
      ## holds tempory postion of number swapped for max
       temp = arr[fillslot]
       arr[fillslot] = arr[positionOfMax]
       arr[positionOfMax] = temp
```

Complexity table

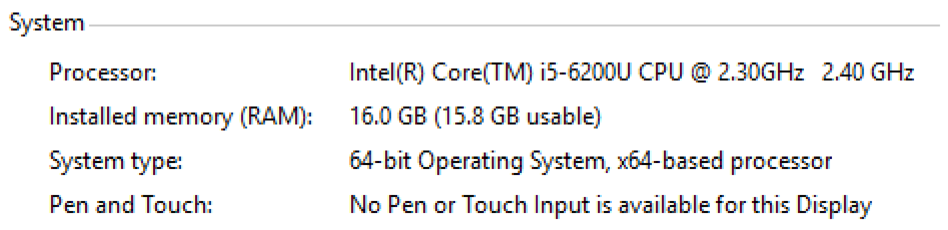
Name	Best	Average	Worst	Stable
Merge sort	nlogn	nlogn	nlogn	yes
Insertion sort	n	n^2	n^2	yes
Count sort	-	n+r	n+r	yes
Bubble Sort	n	n^2	n^2	yes
Selection sort	n^2	n^2	n^2	no

Implementation and Benchmarking

Below are graphs for two separate rounds results from the same code. I have decided to run the code twice to account for outliers in the material

The code was run on jupyter notebooks

On a machine of specs:



System

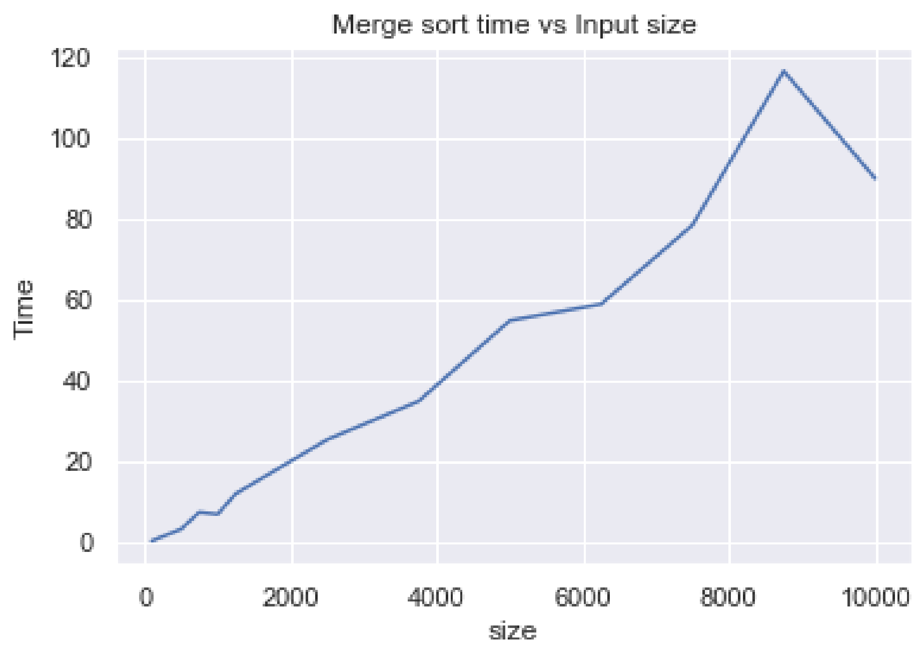
Processor:	Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz 2.40 GHz
Installed memory (RAM):	16.0 GB (15.8 GB usable)
System type:	64-bit Operating System, x64-based processor
Pen and Touch:	No Pen or Touch Input is available for this Display

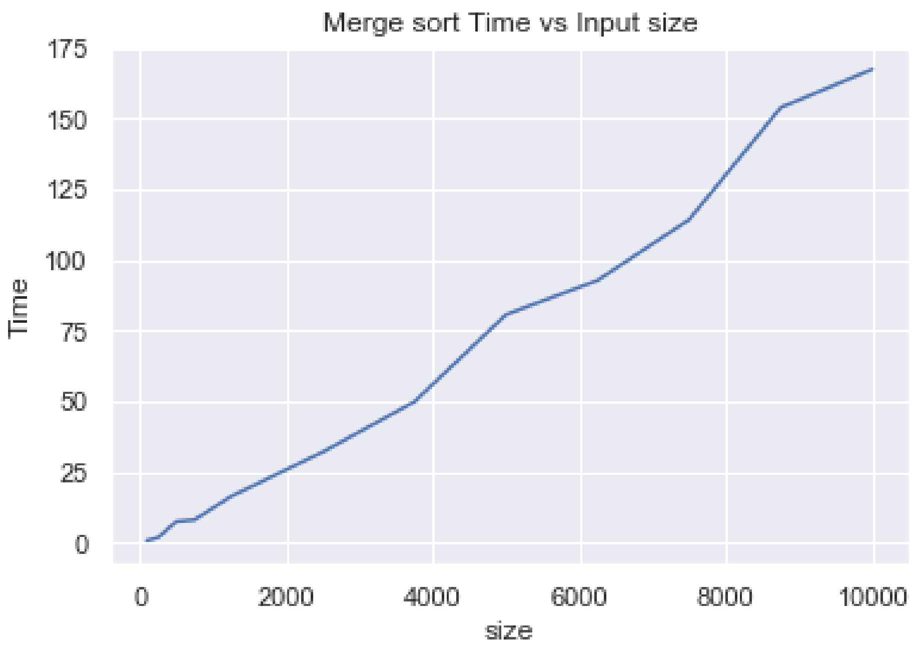
Merge Sort

My expected result or merge sort algorithm whose average times sort complexity is nlogn. Would be the time for the completion to increase at a steady rate from each input point to another.

The graph below shows the results of the average of 10 completions at each n input size of my merge sort algorithm. The graph shows the algorithm increases as expected until it drops at the 10000 n mark. I believe this to be an outlier so I have run the program again from the beginning to see if this graph is consistent.

The second graph confirms that this was an outlier and runs exactly as expected.

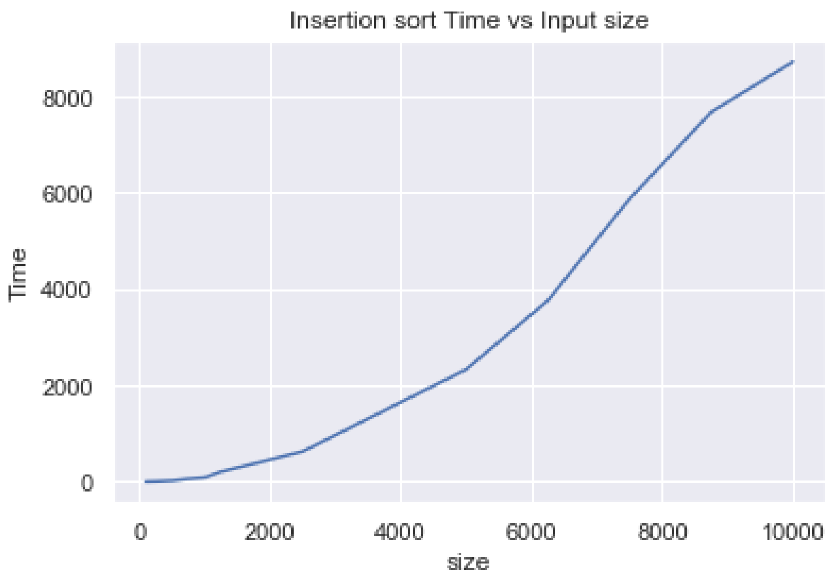


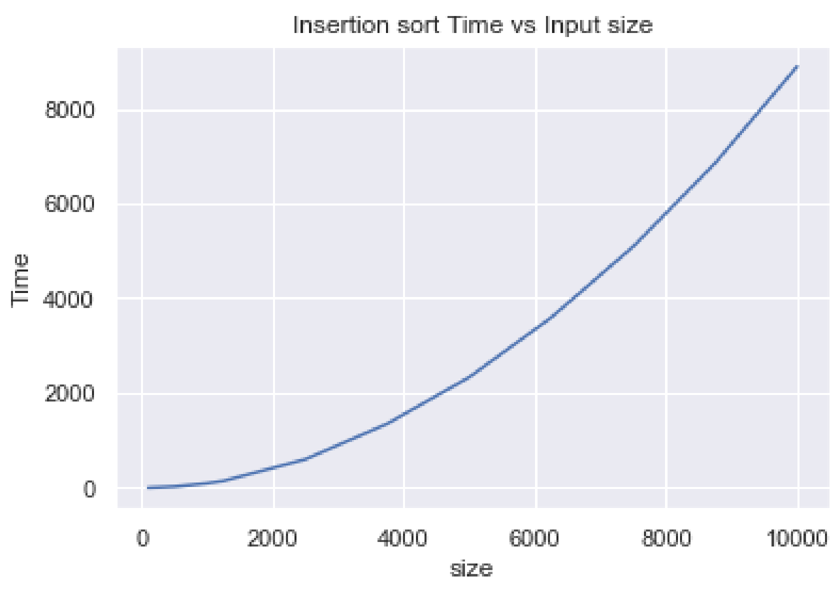


Insertion Sort

The insertion sort algorithm has an n^2 average case time complexity so I would expect the time of completion to graph to remain low initially and then get exponentially longer as the size of n increases.

The graph below shows the results of the average of 10 completions at each n input size of my insertion sort algorithm. The graph below confirms my theory based on time complexity of insertion sort. As n increases the time taken for the algorithm to complete takes much longer as n^2 means the time complexity will increase at a squared speed of the n input.

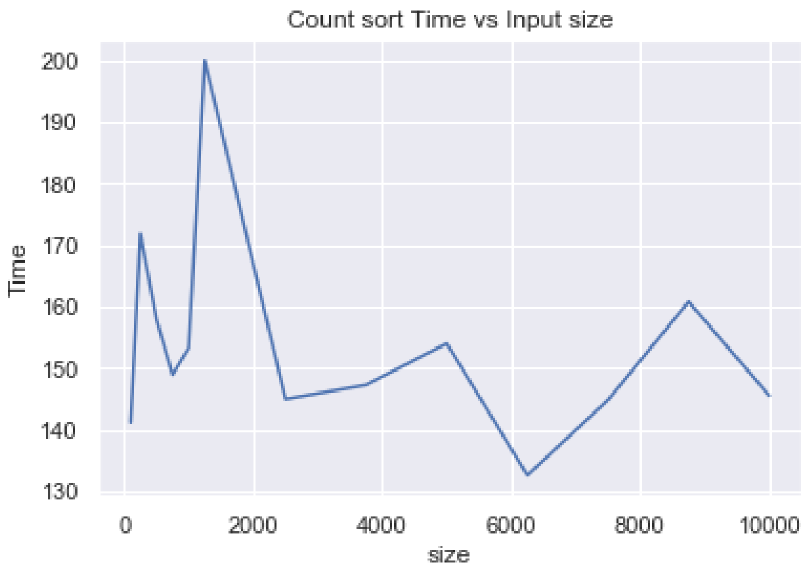


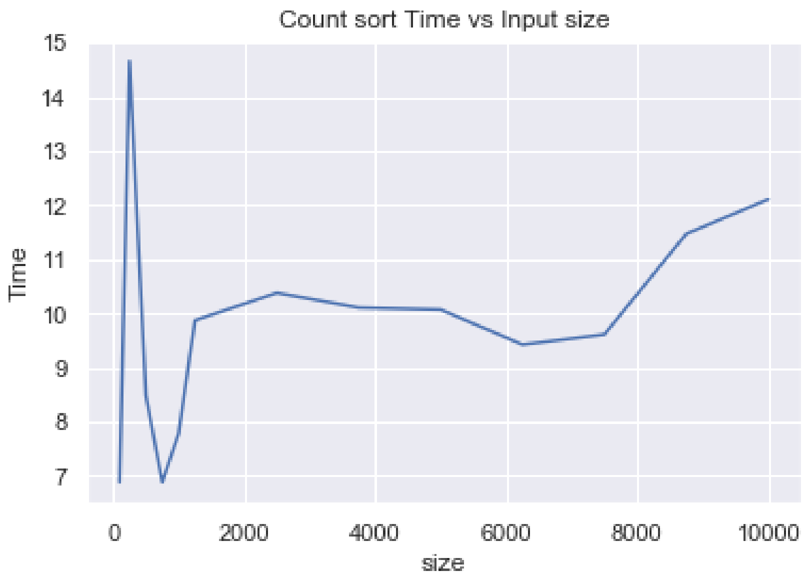


Count Sort

Count sort has a time complexity of n+r where n is the input size of the array and r is the range in the case of this project r was set to 50000. Because range is so high and the inputs much lower I would expect the count sort array to have similarly low completion times.

The graph below shows the results of the average of 10 completions at each n input size of my count sort algorithm. The graph shows low numbers throughout. Some of the smaller n inputs have longer time completion than the larger ones. This is because the size of k is much larger than n in this project and has a much more domineering presence on the result.

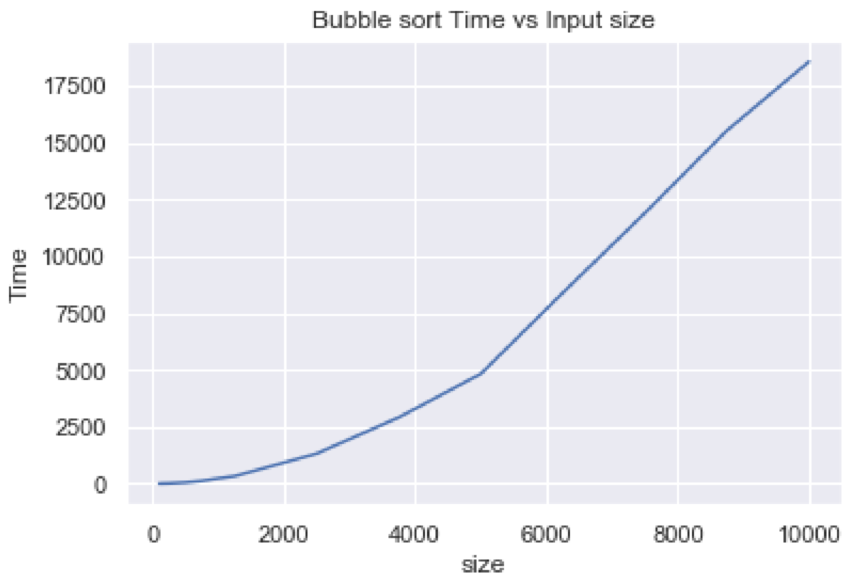


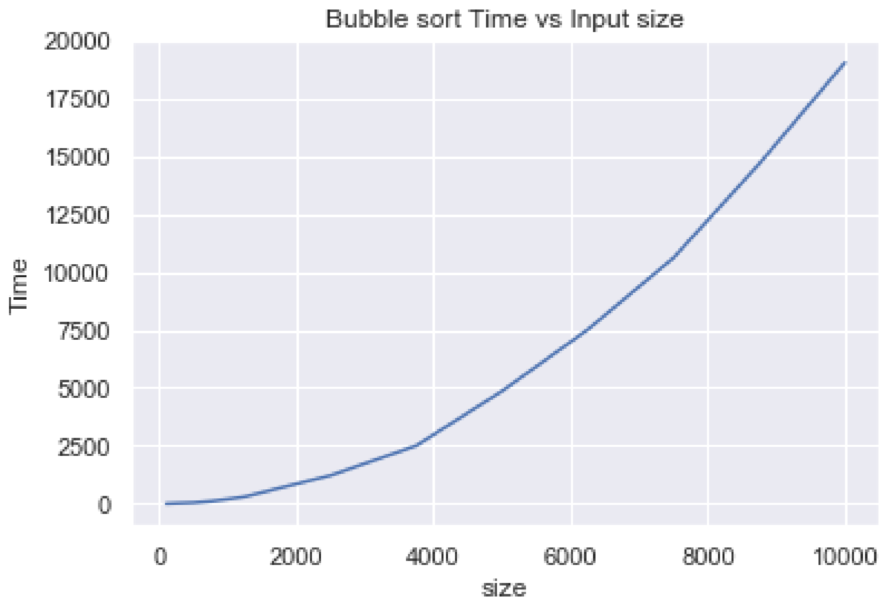


Bubble Sort

The Bubble sort algorithm has an n^2 average case time complexity so I would expect the time of completion to graph to remain low initially and then get exponentially longer as the size of n increases. Similar to insertion sort and selection sort

The graph below shows the results of the average of 10 completions at each n input size of my Bubble sort algorithm. The graph below confirms my theory based on time complexity of insertion sort. As n increases the time taken for the algorithm to complete takes much longer as n^2 means the time complexity will increase at a squared speed of the n input. The algorithm does work as expected and looks similar on a graph however it takes much longer to complete than insertion sort or selection sort

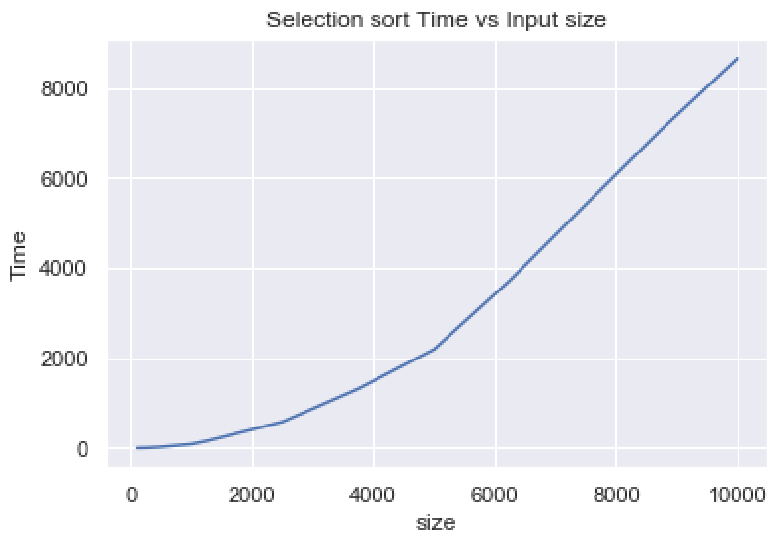


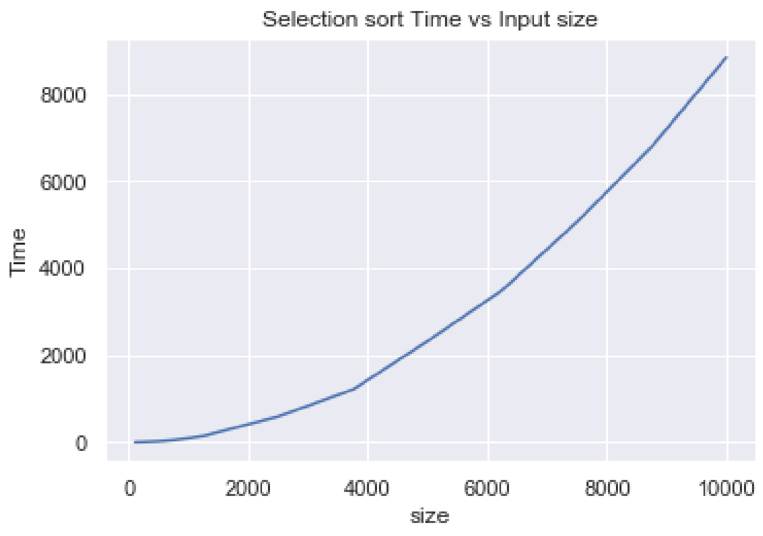


Selection Sort

Selection sort has the same average time complexity as bubble sort and insertion sort o I would expect them to graph similarly with an average time complexity of n^2. The time of completion should get longer to the squared power as n increases starting off small and getting larger much more quickly

The graph below shows the results of the average of 10 completions at each n input size of my selection sort algorithm. The graph below confirms my theory based on time complexity of selection sort. The graph has almost identical completion as insertion sort and graphs similarly. It looks similar on this graph to bubble sort however bubble sort takes much longer to complete on average.

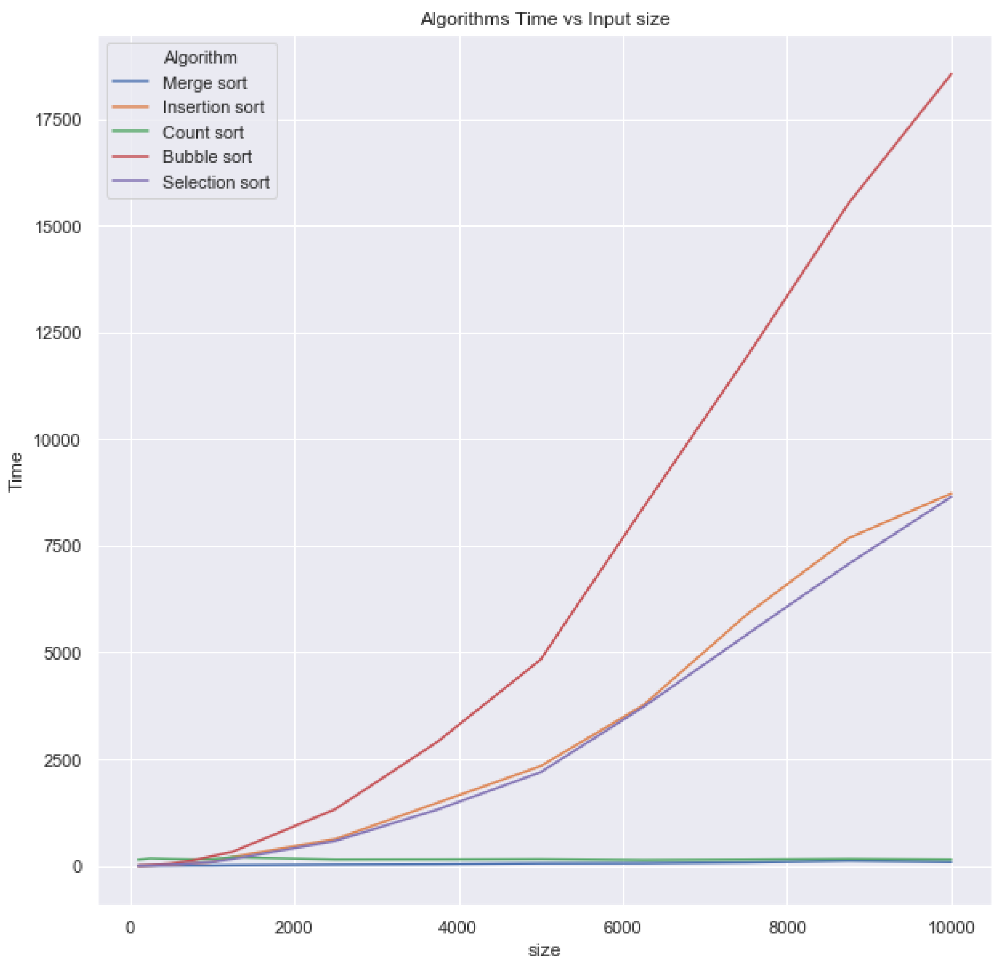




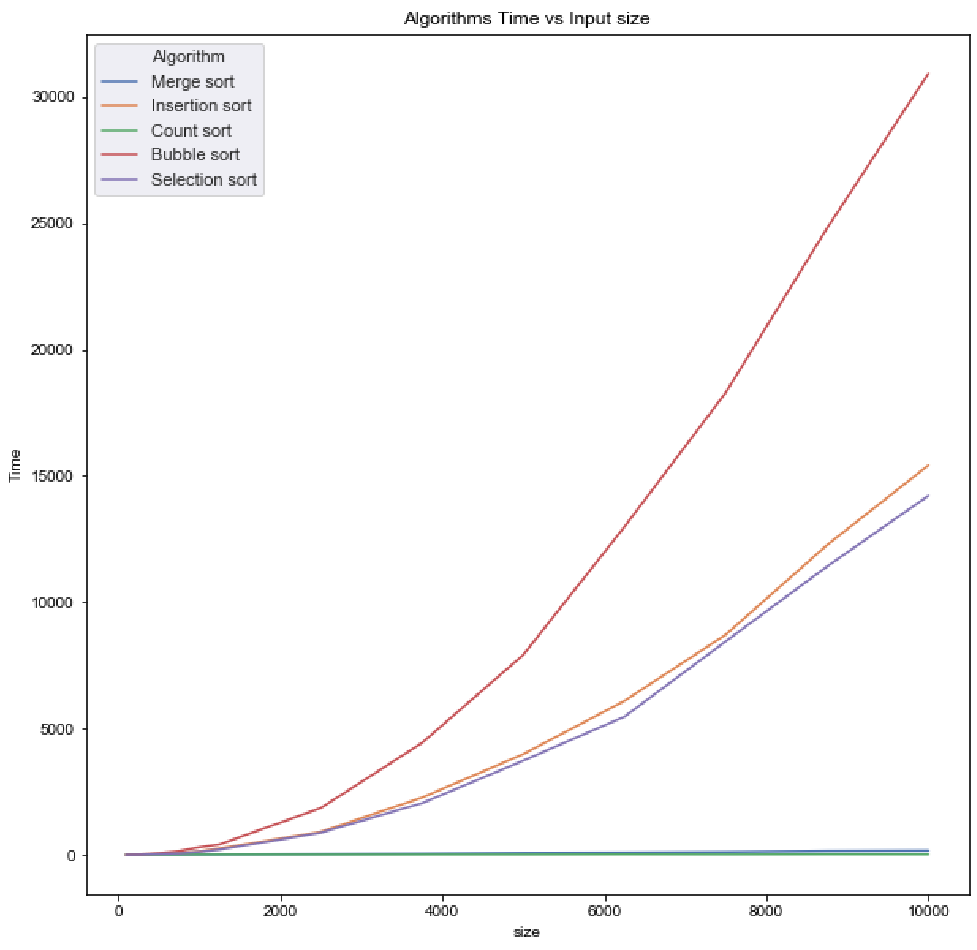
Algorithm time comparisons

On the below graphs we are able to see all of the algorithms time of completions on the same scale. Merge sort and count sort appear as flat lines on this graph as there increases/ decrees from point to point are so minimal in comparison to the other 3 sorts. I expected merge sort, insertion sort and bubble sort to have very similar completion times. While this is true for merge sort and insertion sort having almost identical times of completion. Bubble sort takes much longer to complete on average this may be down to the efficiency of the algorithm.

Overall results were similar to what was expected other than bubble sort taking longer. However all the algorithms graph in accordance to the Big o notation.

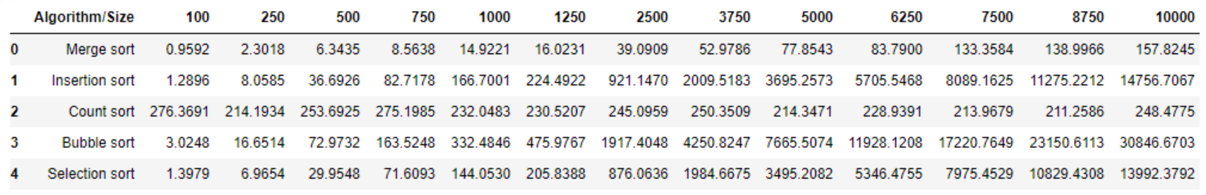


Algorithms Time vs Input size
Algorithm
Merge sort
Insertion sort
Count sort
Bubble sort
Selection sort
Time
size
0
2500
5000
7500
10000
12500
15000
17500
0
2000
4000
6000
8000
10000

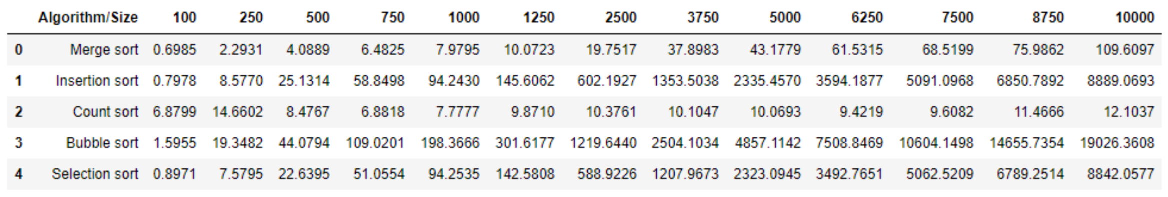


Algorithm Benchmark Chart.

Below are the two output tables



	Algorithm/Size	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
0	Merge sort	0.9592	2.3018	6.3435	8.5638	14.9221	16.0231	39.0909	52.9786	77.8543	83.7900	133.3584	138.9966	157.8245
1	Insertion sort	1.2896	8.0585	36.6926	82.7178	166.7001	224.4922	921.1470	2009.5183	3695.2573	5705.5468	8089.1625	11275.2212	14756.7067
2	Count sort	276.3691	214.1934	253.6925	275.1985	232.0483	230.5207	245.0959	250.3509	214.3471	228.9391	213.9679	211.2586	248.4775
3	Bubble sort	3.0248	16.6514	72.9732	163.5248	332.4846	475.9767	1917.4048	4250.8247	7665.5074	11928.1208	17220.7649	23150.6113	30846.6703
4	Selection sort	1.3979	6.9654	29.9548	71.6093	144.0530	205.8388	876.0636	1984.6675	3495.2082	5346.4755	7975.4529	10829.4308	13992.3792



	Algorithm/Size	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
0	Merge sort	0.6985	2.2931	4.0889	6.4825	7.9795	10.0723	19.7517	37.8983	43.1779	61.5315	68.5199	75.9862	109.6097
1	Insertion sort	0.7978	8.5770	25.1314	58.8498	94.2430	145.6062	602.1927	1353.5038	2335.4570	3594.1877	5091.0968	6850.7892	8889.0693
2	Count sort	6.8799	14.6602	8.4767	6.8818	7.7777	9.8710	10.3761	10.1047	10.0693	9.4219	9.6082	11.4666	12.1037
3	Bubble sort	1.5955	19.3482	44.0794	109.0201	198.3666	301.6177	1219.6440	2504.1034	4857.1142	7508.8469	10604.1498	14655.7354	19026.3608
4	Selection sort	0.8971	7.5795	22.6395	51.0554	94.2535	142.5808	588.9226	1207.9673	2323.0945	3492.7651	5062.5209	6789.2514	8842.0577

References

Blelloch, G.E., Leiserson, C.E., Maggs, B.M., Plaxton, C.G., Smith, S.J. and Zagha, M., 1996. A comparison of sorting algorithms for the connection machine CM-2. *Commun. ACM*, *39*(12es), pp.273-297.

Hawking, S. 2000. *Professor Stephen Hawking's website*. [Online]. [Accessed 9 February 2009]. Available from: http://www.hawking.org.uk/

Includehelp.com. (2019). *Python - cmp() function with Example*. [online] Available at: https://www.includehelp.com/python/cmp-function-with-example.aspx [Accessed 25 Apr. 2019].

GeeksforGeeks. (2019). Stability in sorting algorithms - GeeksforGeeks. [online] Available at: https://www.geeksforgeeks.org/stability-in-sorting-algorithms/ [Accessed 25 Apr. 2019].

GeeksforGeeks. (2019). Lower bound for comparison based sorting algorithms - GeeksforGeeks. [online] Available at: https://www.geeksforgeeks.org/lower-bound-on-comparison-based-sorting-algorithms/ [Accessed 25 Apr. 2019].

GeeksforGeeks. (2019). Merge Sort - GeeksforGeeks. [online] Available at: https://www.geeksforgeeks.org/merge-sort/ [Accessed 26 Apr. 2019].

Python, R, and Linux Tips. (2019). *How to Compute Executing Time in Python?*. [online] Available at: https://cmdlinetips.com/2018/01/two-ways-to-compute-executing-time-in-python/ [Accessed 8 May 2019].

GeeksforGeeks. (2019). *Insertion Sort - GeeksforGeeks*. [online] Available at: https://www.geeksforgeeks.org/insertion-sort/ [Accessed 8 May 2019].

GeeksforGeeks. (2019). *Counting Sort - GeeksforGeeks*. [online] Available at: https://www.geeksforgeeks.org/counting-sort/ [Accessed 8 May 2019].

GeeksforGeeks. (2019). *Python Program for Bubble Sort - GeeksforGeeks*. [online] Available at: https://www.geeksforgeeks.org/python-program-for-bubble-sort/ [Accessed 8 May 2019].

Interactivepython.org. (2019). *5.8. The Selection Sort — Problem Solving with Algorithms and Data Structures*. [online] Available at: http://interactivepython.org/runestone/static/pythonds/SortSearch/TheSelectionSort.html [Accessed 8 May 2019].

WhatIs.com. (2019). *What is sorting algorithm? - Definition from WhatIs.com*. [online] Available at: https://whatis.techtarget.com/definition/sorting-algorithm [Accessed 8 May 2019].

GeeksforGeeks. (2019). *Selection Sort - GeeksforGeeks*. [online] Available at: https://www.geeksforgeeks.org/selection-sort/ [Accessed 8 May 2019].

Anon, *Python - cmp() function with Example*. Available at: https://www.includehelp.com/python/cmp-function-with-example.aspx [Accessed May 8, 2019].

important?, W., Adams, J., Murphy, B., Pierce, C. and Perry, J. (2019). *What is stability in sorting algorithms and why is it important?*. [online] Stack Overflow. Available at: https://stackoverflow.com/questions/1517793/what-is-stability-in-sorting-algorithms-and-why-is-it-important [Accessed 8 May 2019].

GeeksforGeeks. (2019). *Stability in sorting algorithms - GeeksforGeeks*. [online] Available at: https://www.geeksforgeeks.org/stability-in-sorting-algorithms/ [Accessed 8 May 2019].

GeeksforGeeks. (2019). *In-Place Algorithm - GeeksforGeeks*. [online] Available at: https://www.geeksforgeeks.org/in-place-algorithm/ [Accessed 8 May 2019].

What are 'in place' and 'out of place' sorting algorithms? - Quora. 2019. What are 'in place' and 'out of place' sorting algorithms? - Quora. [ONLINE] Available at: https://www.quora.com/What-are-in-place-and-out-of-place-sorting-algorithms. [Accessed 08 May 2019].

GeeksforGeeks. 2019. Lower bound for comparison based sorting algorithms - GeeksforGeeks. [ONLINE] Available at: https://www.geeksforgeeks.org/lower-bound-on-comparison-based-sorting-algorithms/. [Accessed 08 May 2019]

Stack Overflow. 2019. algorithm - Definition of non comparison sort? - Stack Overflow. [ONLINE] Available at: https://stackoverflow.com/questions/25788781/definition-of-non-comparison-sort. [Accessed 08 May 2019].

GeeksforGeeks. 2019. Bubble Sort - GeeksforGeeks. [ONLINE] Available at: https://www.geeksforgeeks.org/bubble-sort/. [Accessed 08 May 2019].

Simply Python. 2019. Counting Sort in Python | Simply Python. [ONLINE] Available at: https://simply-python.com/2018/11/15/counting-sort-in-python/. [Accessed 08 May 2019].