School of Computer Science and Informatics

Coursework Submission Cover Sheet

Please use Adobe Reader to complete this form. Other applications may cause incompatibility issues.

Student Number	c1646151
Module Code	CM2303
Submission date	06/02/2018
Hours spent on this exercise	18
Special Provision	

(Please place an x is the box above if you have provided appropriate evidence of need to the Disability & Dyslexia Service and have requested this adjustment).

Group Submission

For group submissions, *each member of the group must submit a copy of the coversheet.* Please include the student number of the group member tasked with submitting the assignment.

Student number of submitting group member	

By submitting this cover sheet you are confirming that the submission has been checked, and that the submitted files are final and complete.

Declaration

By submitting this cover sheet you are accepting the terms of the following declaration.

I hereby declare that the attached submission (or my contribution to it in the case of group submissions) is all my own work, that it has not previously been submitted for assessment and that I have not knowingly allowed it to be copied by another student. I understand that deceiving or attempting to deceive examiners by passing off the work of another writer, as one's own is plagiarism. I also understand that plagiarising another's work or knowingly allowing another student to plagiarise from my work is against the University regulations and that doing so will result in loss of marks and possible disciplinary proceedings.

A run-time analysis of insertion sort and counting sort.

C1646151 | Charles Read

1. Introduction

The insertion sort and counting sort are popular sorting algorithms which take an array of integers and outputs those integers in a sorted array. The two algorithms use different approaches to compute and process the data thus leading to different performance and responsiveness outcomes. I will be testing each algorithm with different array inputs and cases to discover how each algorithm responds to the different situations. Average case arrays are arrays where the values are randomly selected, best case arrays are arrays which elements are already sorted, and worst-case arrays are arrays which elements are reversed.

BEST:	[1,2,3,4,5,6,7,8,9]
AVG:	[3, 9,7,5,1,6,2,4,8]
WORST:	[9,8,7,6,5,4,3,2,1]

Below is the best, average and worst case time complexities (big-O notation) for the insertion sort algorithm and counting sort algorithm.

Insertion Sort Time Complexity:

BEST:	O(n)
AVG:	$O(n^2)$
WORST:	$O(n^2)$

Counting Sort Time Complexity:

BEST:	O(n + k)
AVG:	O(n + k)
WORST:	O(n + k)

The Counting sort algorithm is more complex than insertion sort. It tends to be more efficient if the range of elements in the array, which is k, is not much greater than the number of elements, which is n. If the range is significantly larger, then the algorithm would run into performance problems and space complexity problems.

2. Pseudocode and implementation

The two sorting algorithms are contained within 'SortingAlgorithms.java'. Each algorithm's code is contained in their own methods: i**nsertionSort()** and **countingSort()**. In the main method, each algorithm is called from within a for loop. This is to allow me to run each algorithm multiple times to then calculate an average run time.

I used **System.nanoTime()** to retrieve the time, in nanoseconds, before and after each iteration of each algorithm, I then subtracted the times to get the difference. I then divided this difference by the number of iterations each algorithm was ran to give me the average time it took for each algorithm to sort that specific array.

My code uses arrays stored in txt files as an input. I wrote the program, 'ArrayGenerator.java', that generates these array txt files as I thought it would be much easier than writing these txt files manually or having them declared in the algorithm methods. ArrayGenerator.java contains methods BestCase(), AverageCase(), and WorstCase() which each create txt files with the correct configuration of arrays. The first for

loop in the main method of this program can be configured to produce different array quantities, array size (n), and how many elements the arrays increment for each new array.

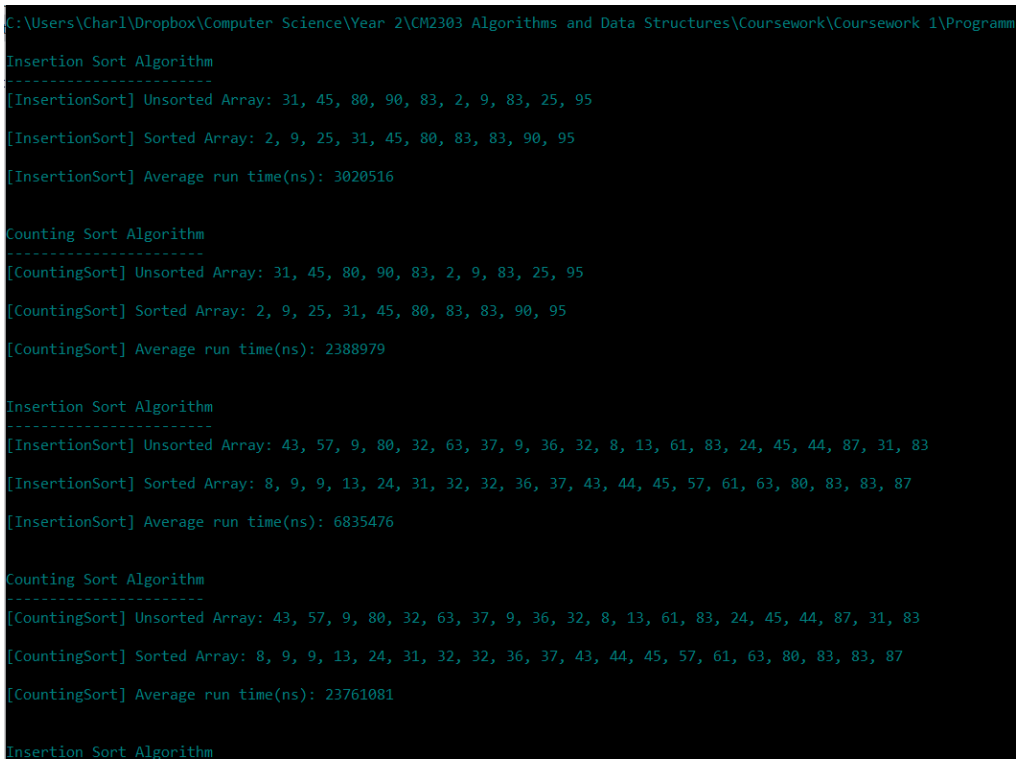
3. Testing

As seen in the results screenshots, I configured my ArrayGenerator.java code to produce much smaller arrays (with only 10 elements, incrementing by 10 each iteration). I did this to make the results clearer, making it easier to see if the code is functioning correctly. From each screenshot, you can see the unsorted array, the outputted sorted array, and the time taken to process for each algorithm (this time is an average time; the 'iterations' variable in the 'main' method of 'SortingAlgorithms.java' determines how many times each algorithm is performed for each array in the txt file).

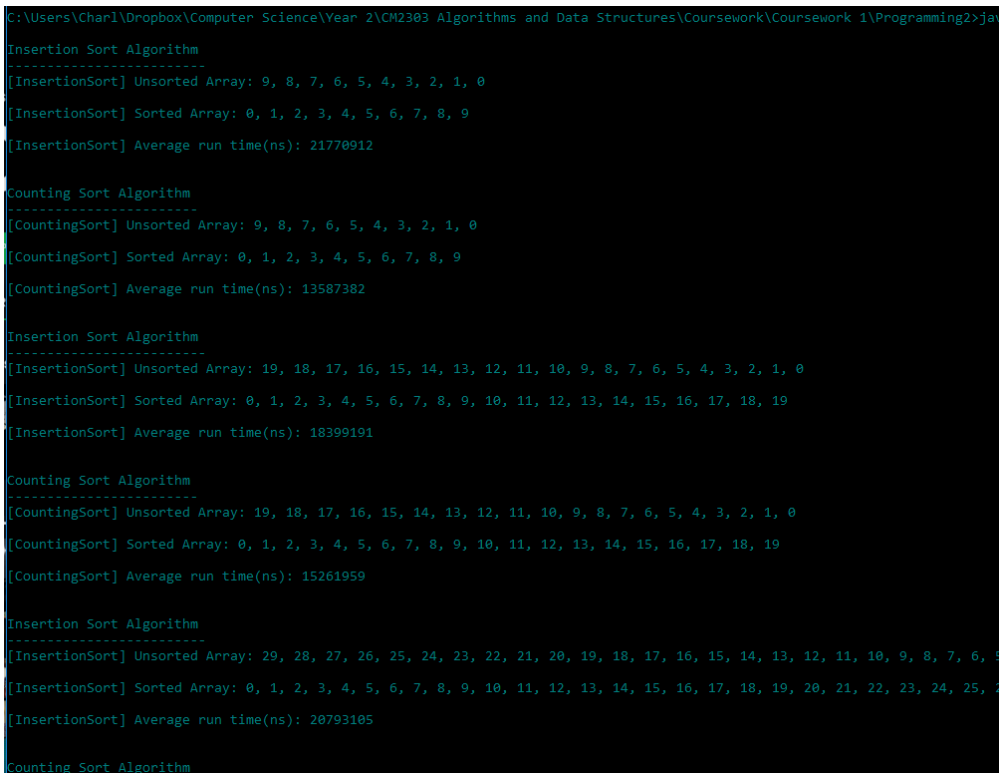
Again, average case arrays have randomly selected elements (not sorted), best case arrays are arrays which elements are already sorted, and worst-case arrays have reversed elements.

Note: for these tests I changed the 'iterations' variable to 1 so the average run time is just the value for running each algorithm once per array. I did this so multiple arrays can be seen being sorted in the same screenshot.

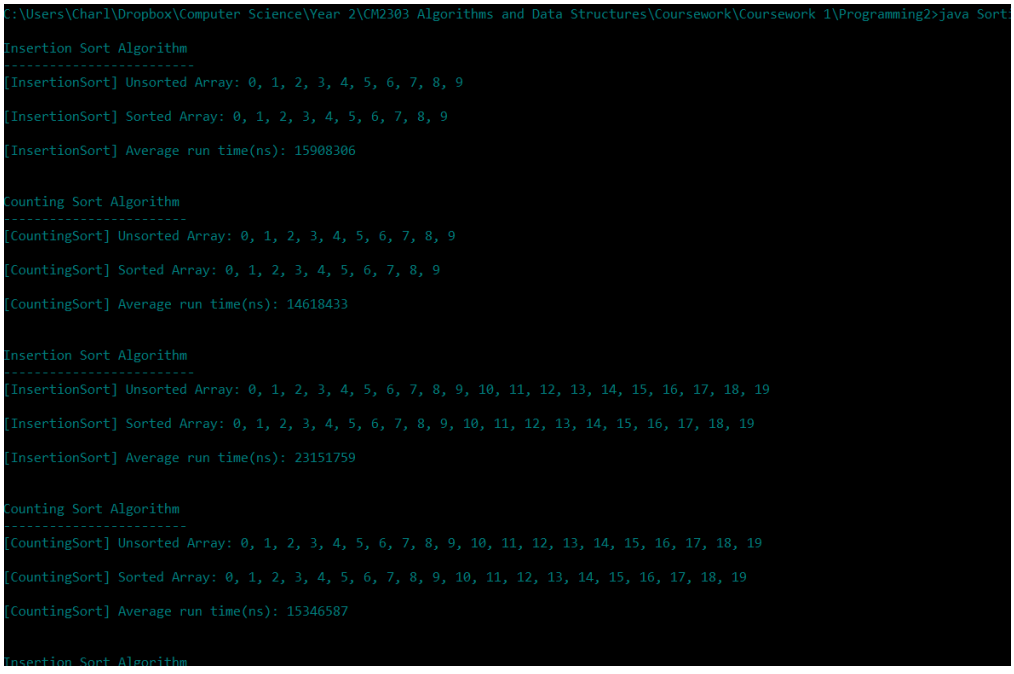
Average-Case Results:



Worst-Case Testing Results:



Best-Case Testing Results:



4. Experimental Setup and Results

To get the most accurate results, I used much larger arrays to produce the data to plot the graphs. I used an array size of 1000 and incremented 1000 for each array. When I started my experiments, I was incrementing by 10 elements for each array. An incrementation 10 definitely wasn't enough of an increase in proportion to my array sizes so I changed it to 1000 to really show the linear pattern of each of the algorithms. I decided to use 10 arrays (10 plots of data) as I thought that would be enough to show the correlation. E.g. my first array had 1000 elements, second had 2000, third had 3000, etc.

Each algorithm ran each array 1000 times as I wanted to get accurate results to ensure I had a linear line in my graphs. This value can be changed by editing the 'iterations' variable in my main method of SortingAlgorithms.java. At first, I had tried 10 and 100 but my results were too inconsistent, and a pattern wasn't clearly visible within my results.

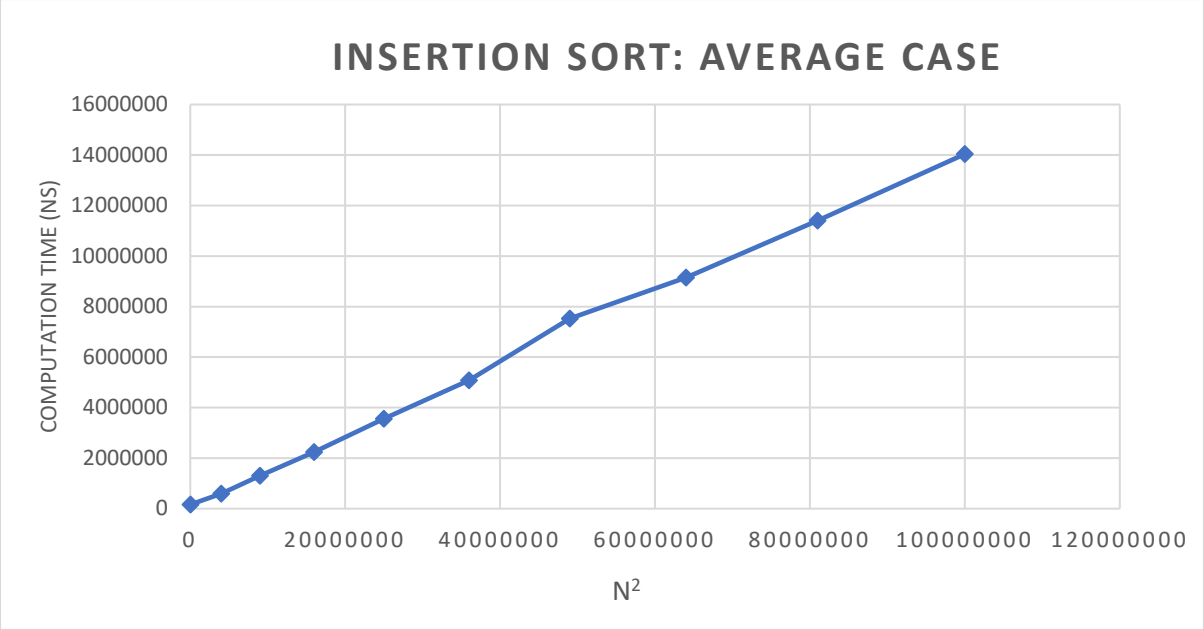
For my x axis scale, I used the time complexities that I had presented in my introductions. I found that using these gave me the most linear distribution of plots and allowed me to draw my conclusions.

Insertion Sort Time Complexity:		Counting Sort Time Complexity:	
BEST:	O(n)	BEST:	O(n + k)
AVG:	$O(n^2)$	AVG:	O(n + k)
WORST:	$O(n^2)$	WORST:	O(n + k)

Insertion Sort:

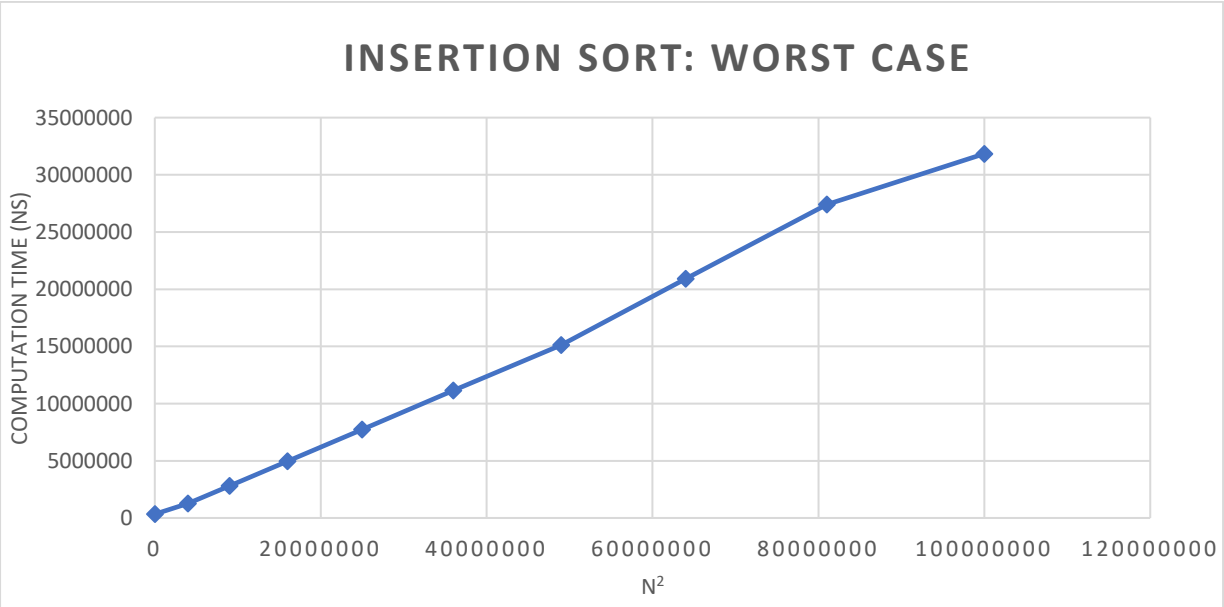
Insertion Sort: Average Case

n^2	Computation Time (ns)
10000	161562
4000000	588697
9000000	1302093
16000000	2244494
25000000	3563071
36000000	5074112
49000000	7524465
64000000	9147372
81000000	11409023
100000000	14038718



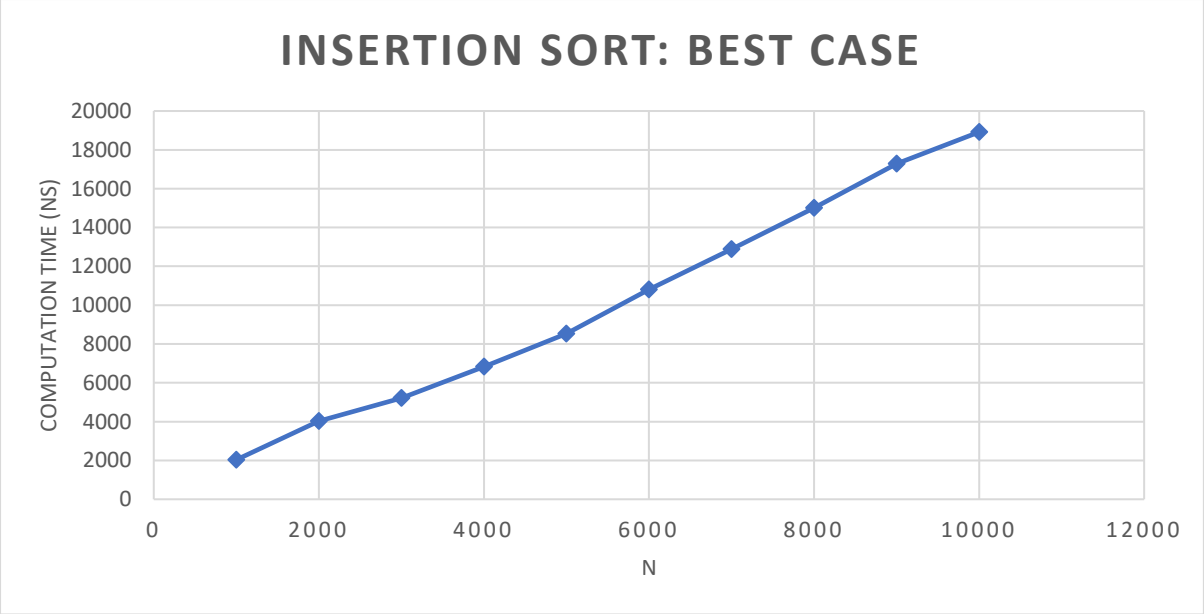
Insertion Sort: Worst Case

n^2	Computation Time (ns)
10000	337017
4000000	1268349
9000000	2796189
16000000	4960546
25000000	7732246
36000000	11139749
49000000	15130866
64000000	20924585
81000000	27393912
100000000	31839586



Insertion Sort: Best Case

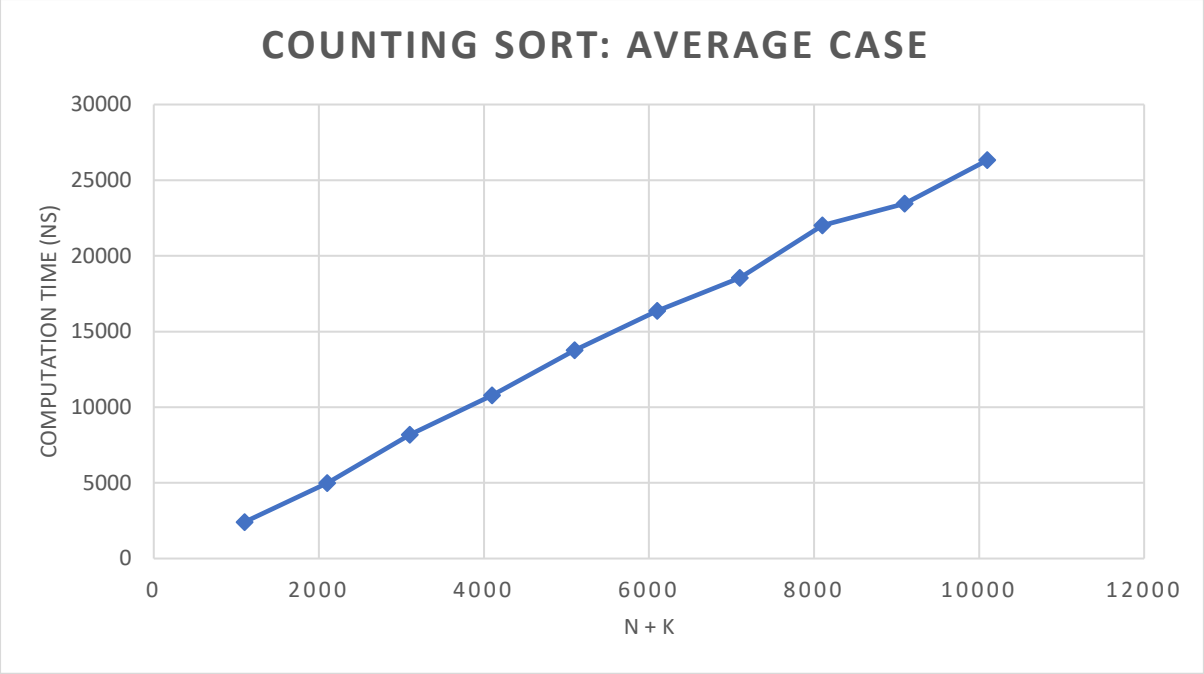
n	Computation Time (ns)
1000	2034
2000	4026
3000	5221
4000	6830
5000	8522
6000	10808
7000	12887
8000	15014
9000	17288
10000	18915



Counting Sort:

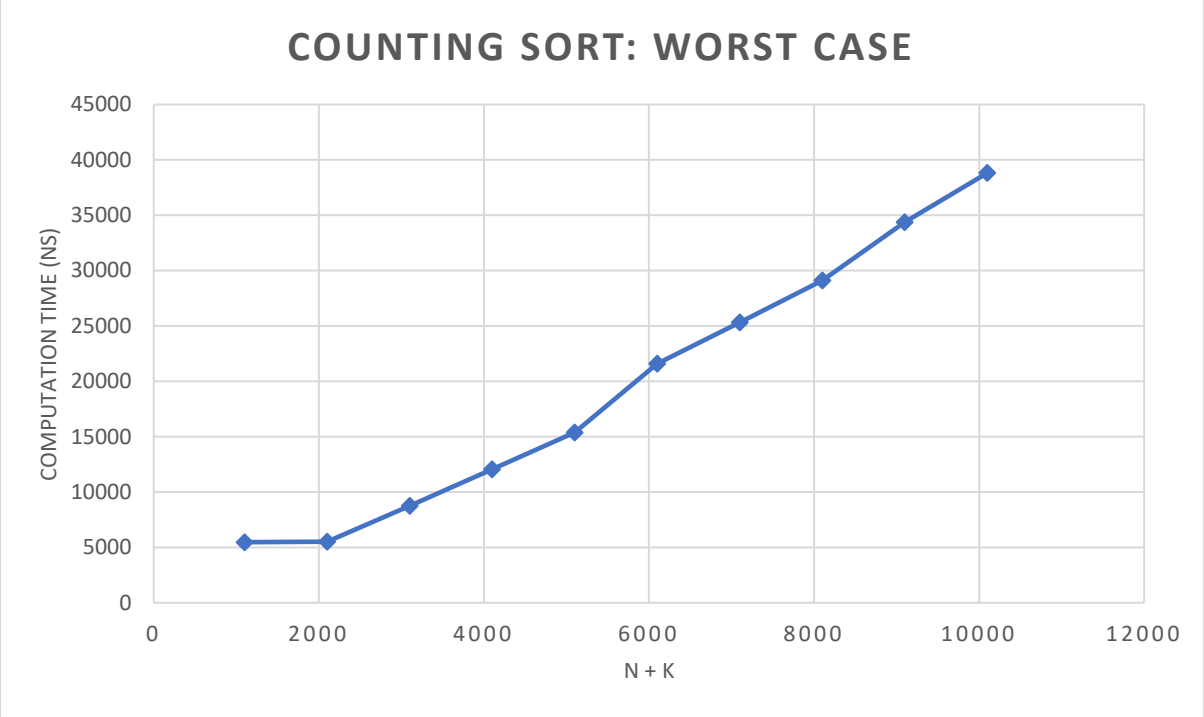
Counting Sort: Average Case

n + k	Computation Time (ns)
1100	2412
2100	4988
3100	8172
4100	10791
5100	13766
6100	16366
7100	18544
8100	22016
9100	23446
10100	26319



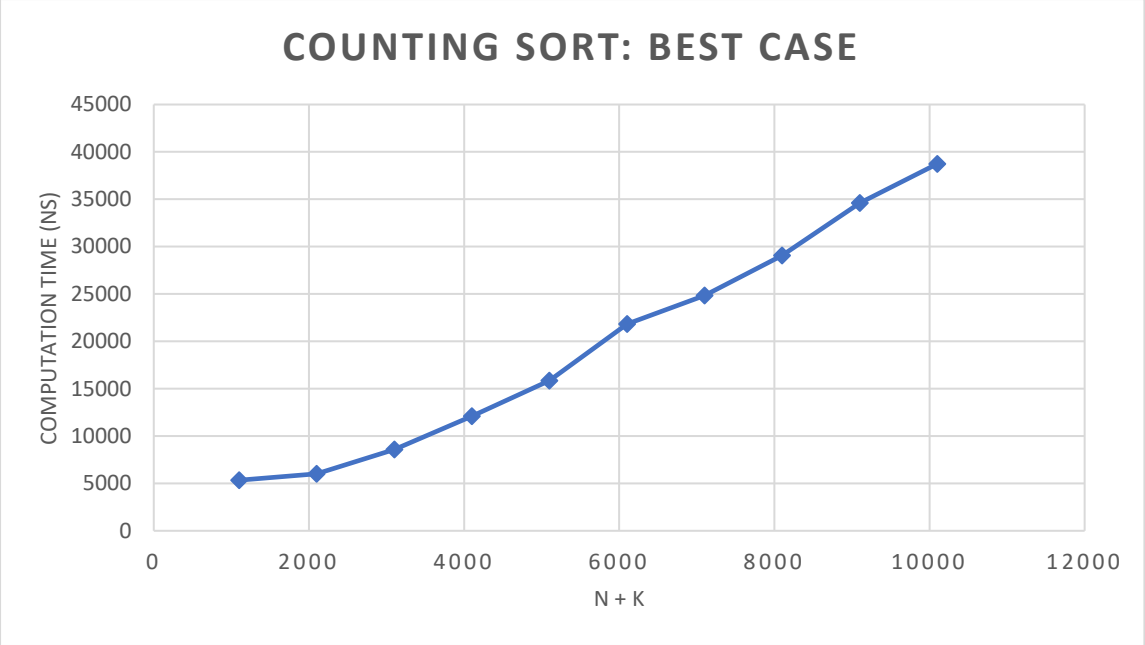
Counting Sort: Worst Case

n + k	Computation Time (ns)
1100	5476
2100	5525
3100	8766
4100	12043
5100	15366
6100	21597
7100	25315
8100	29106
9100	34364
10100	38824



Counting Sort: Best Case

n + k	Computation Time (ns)
1100	5341
2100	6003
3100	8593
4100	12099
5100	15855
6100	21826
7100	24851
8100	29062
9100	34615
10100	38737



5. Conclusions and Discussion

My results from my experiments show that counting sort is much quicker than insertion sort in all cases, whether that be best case, worst case or average case. The difference between counting sort cases (average, worst, best) didn't really differ that much in comparison to how the insertion sort cases did.

From my insertion sort results, a clear positive correlation between n^2 and its time complexity can be seen. The linear line for each case, shows that the algorithm takes progressively more time to sort the arrays the larger the arrays get. The best case is very fast compared to the average case and worst case due to the algorithm not having to perform any functions if it comes across an element that is already sorted. This can't be said for counting sort as that algorithm consistently runs at O(n + k) for all cases as each element is manipulated by the C array when the elements are counted. The counting sort values for each case are very similar with surprisingly the average case having slightly quicker run-times, this of course could potentially be an anomaly caused by factors that are discussed below.

Background programs and processing could have caused problems/anomalies while the algorithms were running as the run time averages could have been influenced by processing power spent on these other resources. To make this an even fairer test, I could have made sure all programs were closed (apart from the essentials), stopped any unrequired services like updates or anti-virus, as well as made sure I was running all the tests on the same machine and OS.

Additionality, to improve the accuracy of my data, I could run each algorithm a few times before starting to record the run-times, like a warm up run that would not be recorded in the results. This would remove any potential setup processing spikes from declaring variables or other functions within each algorithm methods.

The data recorded is from arrays that start with 1000 elements. In programs that require sorting smaller arrays, insertion sort could still be a viable option. If the array is known to be almost sorted or just smaller, then the insertion sort would be the preferred algorithm as it is very simple to implement and the performance is justifiable. Additionally, if performance isn't a factor you are concerned about, then insertion would also be recommended due to its simplicity and its space complexity compared to counting sort.

It is preferred to use counting sort when you know the size of the array/s as when the range of the arrays are much greater than the size of the arrays, then the algorithm isn't effective and using a simpler algorithm like insertion sort would be more efficient. If the range is large, the counting sort algorithm may also suffer poor space complexity.

In conclusion, my results show that both algorithms have a positive and progressive correlation. Counting sort algorithm is much quicker, but it doesn't excel in a certain case (best, worst, average). Counting sort is a slightly more complicated algorithm that, with larger arrays, may cause space complexity problems. Insertion sort is slower, it can perform fast depending on its input case but still not recommended for very large array sizes.