SORTING ALGORITHMS

A COMPARISON

THE ALGORITHMS CHOSEN

- The application itself can demonstrate a wide variety of different algorithms from the simplest type (bubble) so the more complex (gravity) to the memory intensive (counting)
- However, I have chosen to compare the insertion sort and the counting sort directly.

THE PROBLEM?

The problem I am trying to solve is the issue of choosing an algorithm for different situations.

Eg

- Almost sorted data,
- Partitioned data,
- Random data,
- Worst case scenario data,

DATA STRUCTURES

- I have chosen to run my application on a simple 1d array specifically for ease and speed to develop into an application, However, with little to no effort the entire thing could be converted to run on lists of variable sizes.
- This could be accomplished by either simply converting the list to an array beforehand or extending the list class to handle the algorithms.
- Using a 1d array also allows me to do a lot of sorting very quickly and efficiently as I only have to move the pointer and not follow data links to go either way along the collection

THE COMPARISON

- First some obvious statements, the way the algorithms are implemented one fundamentally uses much less memory than the other the counting sort uses the original array as well as 2 more of the same size tripling the amount of memory required to run but the trade off is that it doesn’t use any comparisons whatsoever.
- The insertion sort (for these purposes) sorts by adding each element individually to the correct place in the array (rather than running on insert it runs once the array is full)

THE COMPARISON (AVERAGES OVER 2048 CYCLES)

Counting sort:

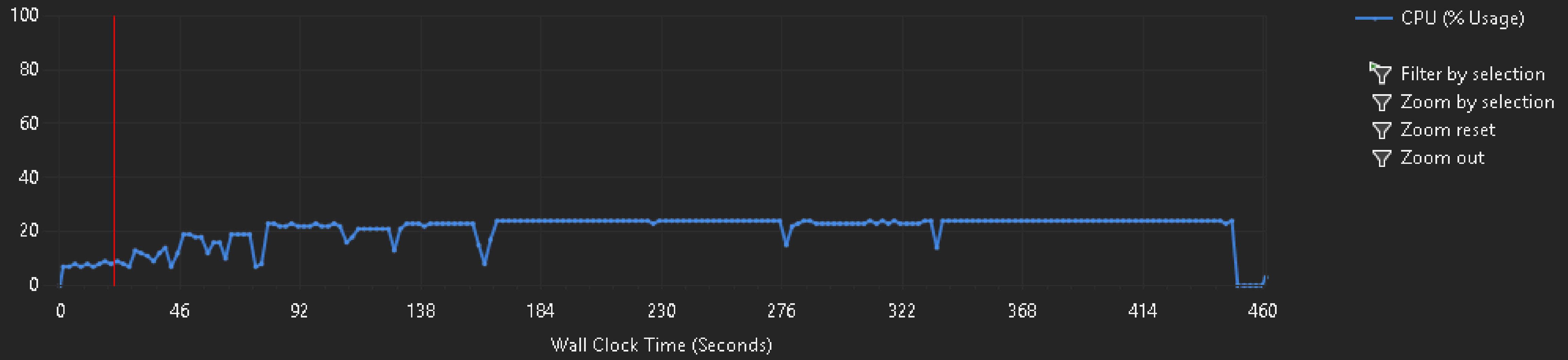
Size	512	1024	2048	4096	8192	16394
Almost Sorted	2.278	2.522	2.505	2.539	3.427	4.028
Partitioned	2.380	2.473	2.480	2.527	3.653	4.062
Random	2.569	2.444	2.593	2.547	3.228	4.086
Worst Case	2.545	2.498	2.509	2.487	3.682	2.795

Insertion sort:

Size	512	1024	2048	4096	8192	16394
Almost Sorted	2.467	2.476	2.491	2.568	2.807	3.485
Partitioned	2.679	3.416.	7.750	19.786	64.221	305.856
Random	2.598	2.934	4.890	11.117	43.757	130.963
Worst Case	2.729	3.431	7.836	18.681	78.529	265.457

THE REPORT

- Once establishing that the numbers were pretty constant I turned down the number of cycles from 2048 to 512 so as I could actually get a detailed report on where the bottleneck's were. (leaving the averages at 2048 results in more than 2 hours worth of waiting on my laptop)



Hot Path

Function Name	Inclusive Samples %	Exclusive Samples %
Sorting algorethims.exe	100.00	0.00
Sorting_algorethims.Program.Main	100.00	0.00
Sorting_algorethims.Program.SortUnit	99.99	0.00
Sorting_algorethims.Sorts.SortArray	94.63	0.00
Sorting_algorethims.Sorts.insersionSort	**94.47**	**94.47**

Related Views: Call Tree Functions

Functions Doing Most Individual Work

Name	Exclusive Samples %
Sorting_algorethims.Sorts.insersionSort	94.47
System.Console.set_BackgroundColor	1.94
System.Console.GetBufferInfo	1.10
System.Console.SetCursorPosition	1.07
System.Console.Write	1.05

THE HOT PATH

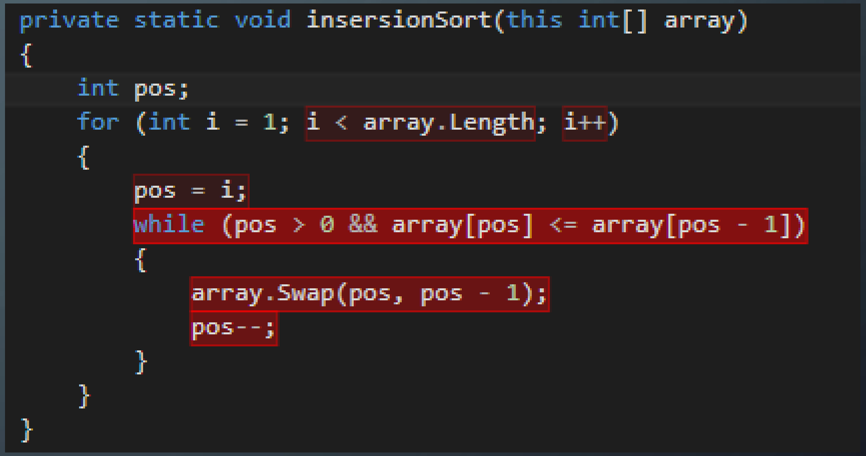
There were some odd discoveries when I checked the hottest lines in the programme eg:

[i < array.Length]

I had not considered what tis property actually did, it would have to pull the length of the array from the stack and not the heap, so technically a slower form of access. To reduce this I could save off the length of the array before hand as in

int length = array.Length

This would save a slight amount of time as it wouldn't have to access the property each time.



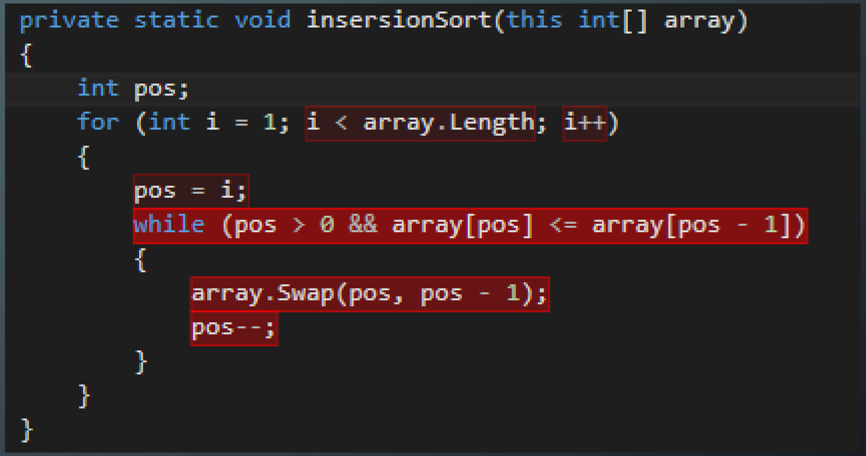
```
private static void insersionSort(this int[] array)
{
    int pos;
    for (int i = 1; i < array.Length; i++)
    {
        pos = i;
        while (pos > 0 && array[pos] <= array[pos - 1])
        {
            array.Swap(pos, pos - 1);
            pos--;
        }
    }
}
```

THE HOT PATH

Setting

pos = i
another one I didn’t except to take up much time unfortunately I cant improve this much farther as both pos and i are needed for the algorethim to work.

Of course the major bottleneck is the while loop. Which was interesting because even when the data was almost sorted we came across a situation where it would still be more efficient on the processor to use the counting sort method rather than loop though the array.

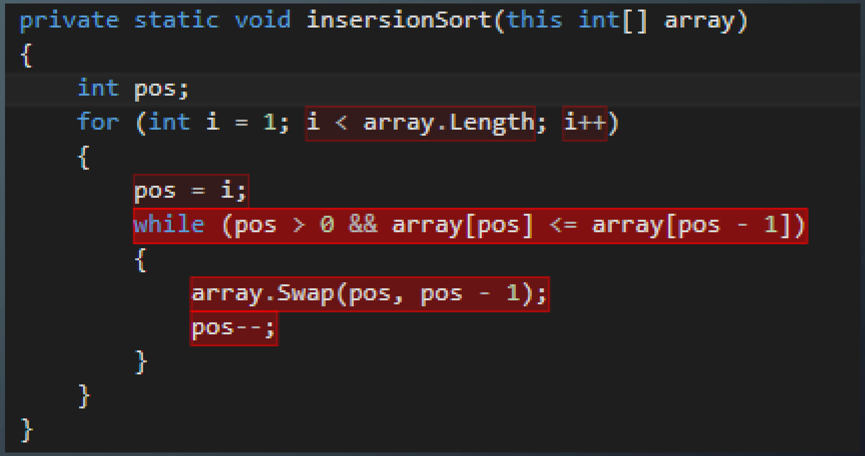


```
private static void insersionSort(this int[] array)
{
    int pos;
    for (int i = 1; i < array.Length; i++)
    {
        pos = i;
        while (pos > 0 && array[pos] <= array[pos - 1])
        {
            array.Swap(pos, pos - 1);
            pos--;
        }
    }
}
```

THE HOT PATH

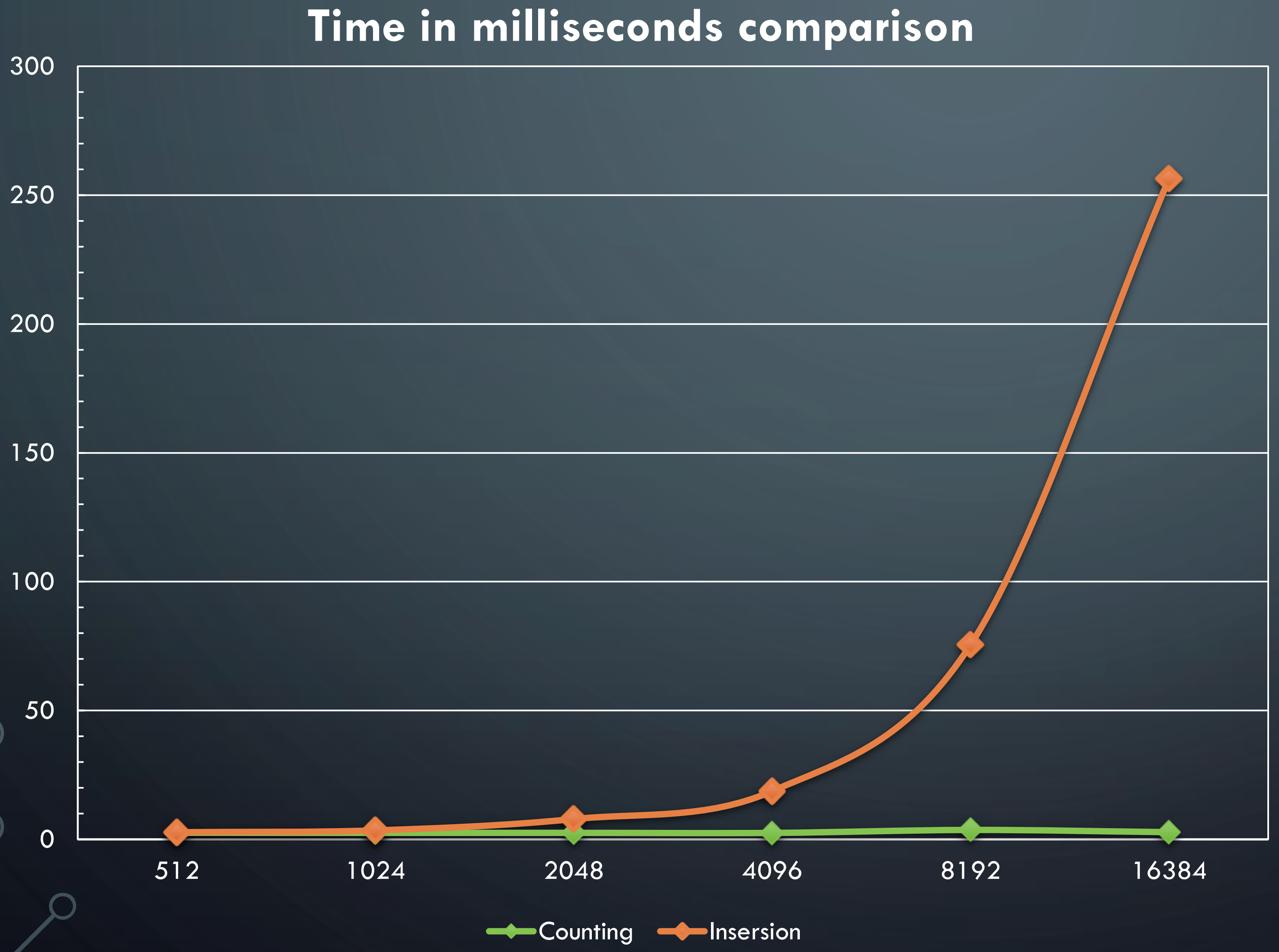
I would have thought that sorting 1 value into an already sorted list would be quicker even though it loops through the entire thing just to check.

The swapping command was going to take up time and I knew that already since it copies a value and does a 3 point swap. In order to accomplish it's namesake.



```
private static void insersionSort(this int[] array)
{
    int pos;
    for (int i = 1; i < array.Length; i++)
    {
        pos = i;
        while (pos > 0 && array[pos] <= array[pos - 1])
        {
            array.Swap(pos, pos - 1);
            pos--;
        }
    }
}
```

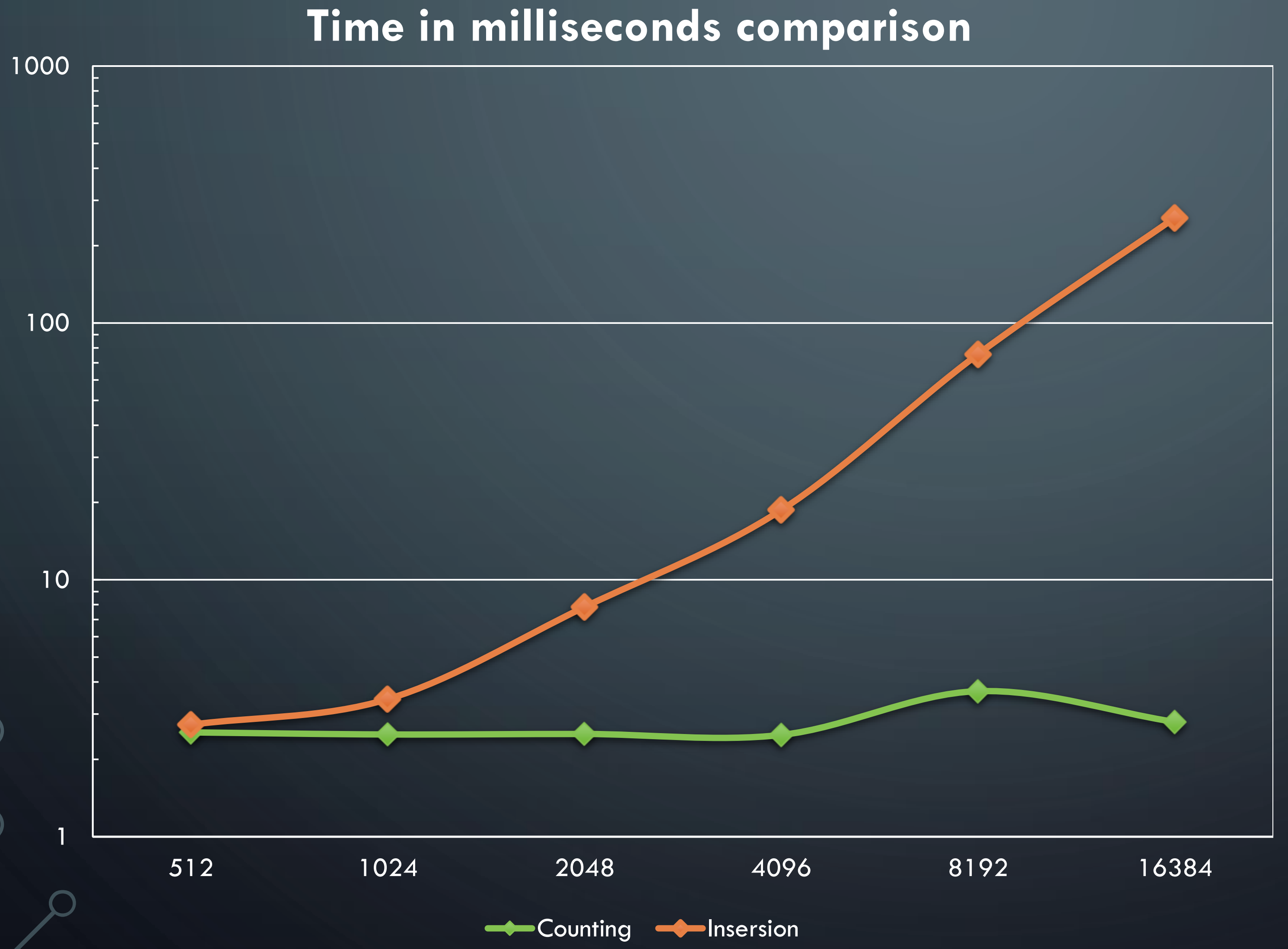
THE EFFICIENCY / COMPLEXITY



Here is a graph of time vs items in array (for worst possible outcome),

And from this we can deduce the algorithms complexities.

THE EFFICIENCY / COMPLEXITY



Logarithmic representation of the same data.

THE EFFICIENCY / COMPLEXITY

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)

This gives us these values for predicting how long the sort will take with each with varying sizes of data.

Using this we can predict that running an insertion and counting sort of size 65,536

This will be roughly 65,536+k and 4,294,967,296 units
we can scale these to the time axes using the previous graphs to get around 3.8ms and 3440ms respectively. (actual results were 3.845 and 3455.527)

CONCLUSION AND FINAL POINTS

In order to make this project I had to rip apart an old programme I made that sorted many different ways and set it to simply do the two in question. So I changed which bit's of the code were called. Creating a new method that takes in all the information needed in order to do the sorting, each can be changed individually and set accordingly.

CONCLUSION AND FINAL POINTS

In conclusion the unit was relatively painless to construct and if I were given the chance to do it again, id choose a more interesting algorithm than just the insertion sort.

FINAL NOTES

- The programme has the option to save every result into a file on the desktop for easier comparisons this is disabled by default but can be enabled within the code