Algorithmic Time and Space Complexity

Robert Tunn, 2015065

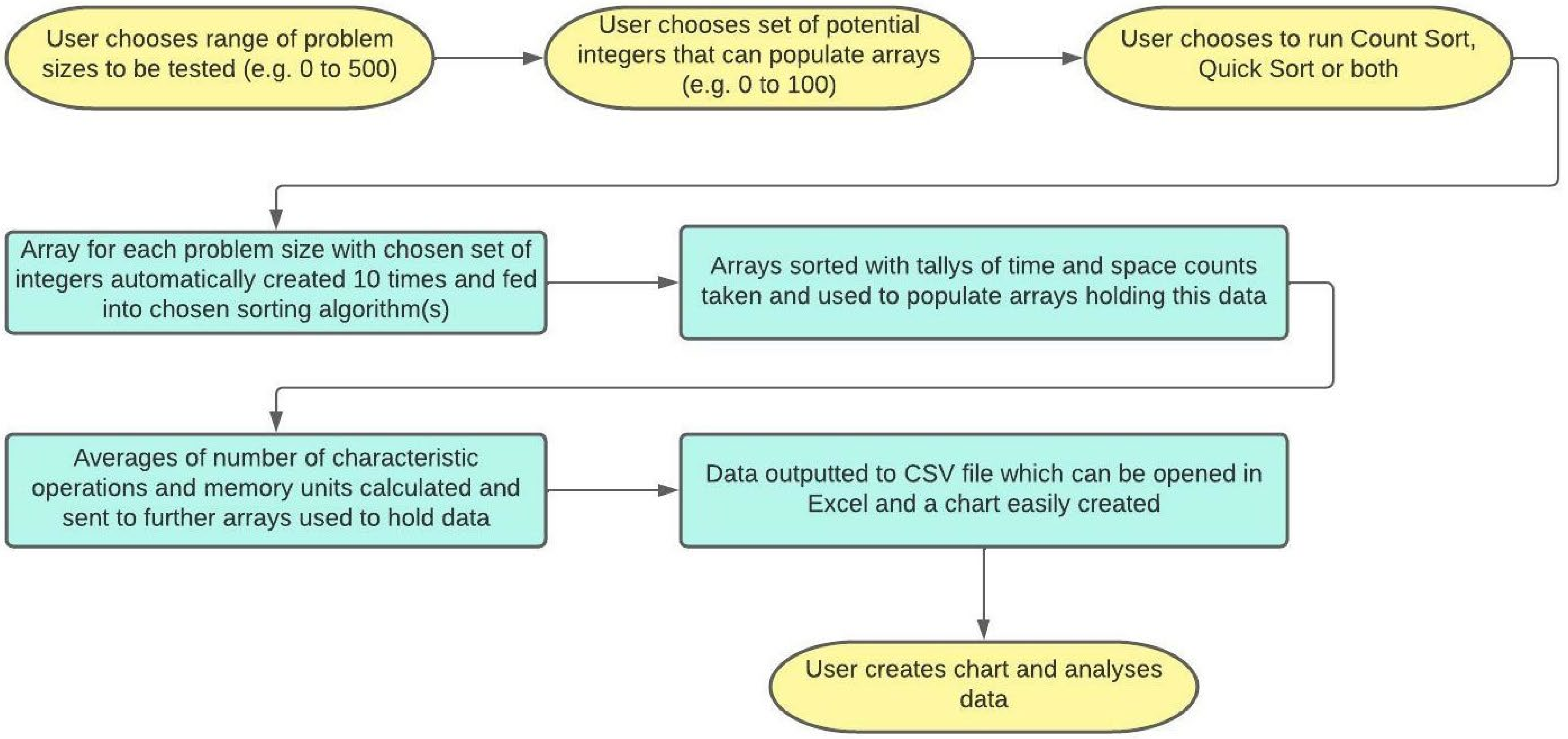
The Experimental Model

Java code

- The code for the Countsort and Quicksort algorithms was cut and pasted into an experimental framework in a Java IDE. Single lines of code were inserted to increment Integer variables which did not interfere with the running of the sorting algorithms rather tallied up the number of characteristic operations for time complexity and the number of memory units for space complexity required for the execution of the sorting algorithms. The placing of these variables is discussed in comments as part of the code.
- The goal was to determine the space and time complexity of the algorithms for the average case (randomly generated integers with little repetition) over varying lengths of problem size (input array length). The best performing algorithm was determined according to the problem size in terms of both time and space complexity. Best (already sorted) and worst case (reverse order) scenarios were not considered.
- In all cases, the first array to be sorted comprises 0 elements and the length of the final array is chosen by the user e.g. 500 elements. The experimental framework can be run on sets of 100 positive integers plus 0 (0 to 100), negative integers plus 0 (-100 to 0) and a mixture of positive and negative integers and 0 (-50 to 50) which are automatically created and sorted. Given that the Court Sort algorithm has two arrays, it is important to keep the size of these sets equal at 101 elements.

The Experimental Model

Flowchart of experimental framework Java code


User chooses range of problem sizes to be tested (e.g. 0 to 500)
User chooses set of potential integers that can populate arrays (e.g. 0 to 100)
User chooses to run Count Sort, Quick Sort or both
Array for each problem size with chosen set of integers automatically created 10 times and fed into chosen sorting algorithm(s)
Arrays sorted with tallys of time and space counts taken and used to populate arrays holding this data
Averages of number of characteristic operations and memory units calculated and sent to further arrays used to hold data
Data outputted to CSV file which can be opened in Excel and a chart easily created
User creates chart and analyses data

The Experimental Model

The collection and presentation of results

- As part of the experimental framework, an output array containing the number of characteristic operations required to sort the input array and another array for the memory space required for each problem size are automatically generated so that the position of the element represents the length of the array e.g. for time complexity {0,1,2…} would mean that a problem size of 0 required 0 operations to sort, a problem size of 1 required 1 operation and problem size of 2 required 2 operations to sort and so on.
- The data is automatically exported from the Java IDE into CSV (comma separated value) files which can be opened in Excel and the results viewed as two columns being problem size against number of characteristic operations/space required. Several Excel files are submitted with this report. Graphs were generated in Excel whereby the x-axis is the problem size and the y-axis is the number of characteristic operations/memory units and these are compared to a chart of different Big-O ratings.

The Experimental Model

The collection and presentation of results

- The number of characteristic operations for time and number of memory units for space complexity are calculated 10 times for each length of input array and an average taken.
- The results were compared to determine the benefits of each algorithm and the range of problem sizes under which they operate most efficiently (including allowing negative integers or not) in terms of both time and space.
- In terms of time complexity, it should be noted that some big data problems are limited to natural numbers (all positive integers, here including 0) or limited to only negative integers so I tested these sub-sets of integers separately.
- The reason for the three separate sets of possible element integers is further explained in the slide "When the range of element integers increases" below.

The Experimental Model

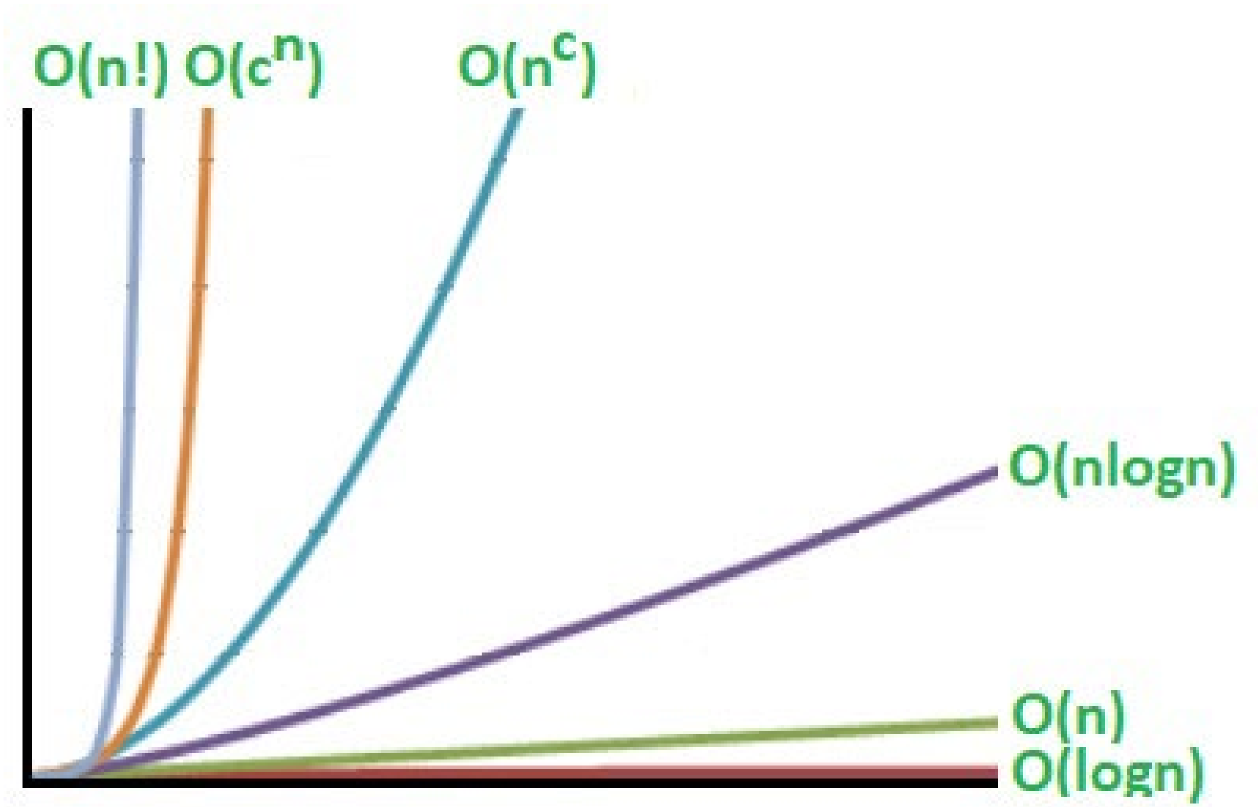
The collection and presentation of results

- The range of possible integers (-100 to 0), (0 to 100) and (-50 to 50) allow arrays of integers to be generated with a high degree of cardinality which is a prerequisite for average case analysis.
- The attached results provided as Excel files represent a broad enough range of input arrays to allow for the nature of the curve to be seen and to be confident that a consistent trend has been established as the problem size tends towards infinity and also to allow for the strengths and drawbacks of each algorithm to be fairly examined.
- Given that the space required for the storage of an array of Integers in Java comprises the header object of 12 bytes + (no of elements x **no of bytes required per Integer**) plus 8 bytes for padding, the number of bytes required to store the integers will become by far the most significant portion of space required as the problem size tends towards infinity. Taking the space required to store one Integer variable as **1 unit (32 bits based on an integer being stored as two's complement),** the space complexity of an array of Integers is dominated by the length of the array. Bytes required for the header and the padding can be disregarded in terms of discerning a Big-O approximation.

Big-O Notiation

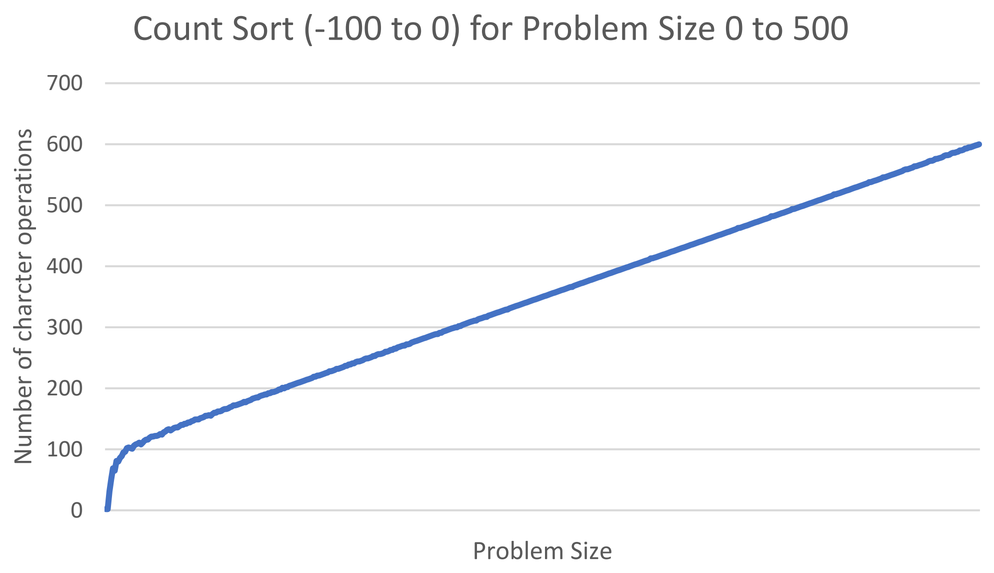
Examples in a chart

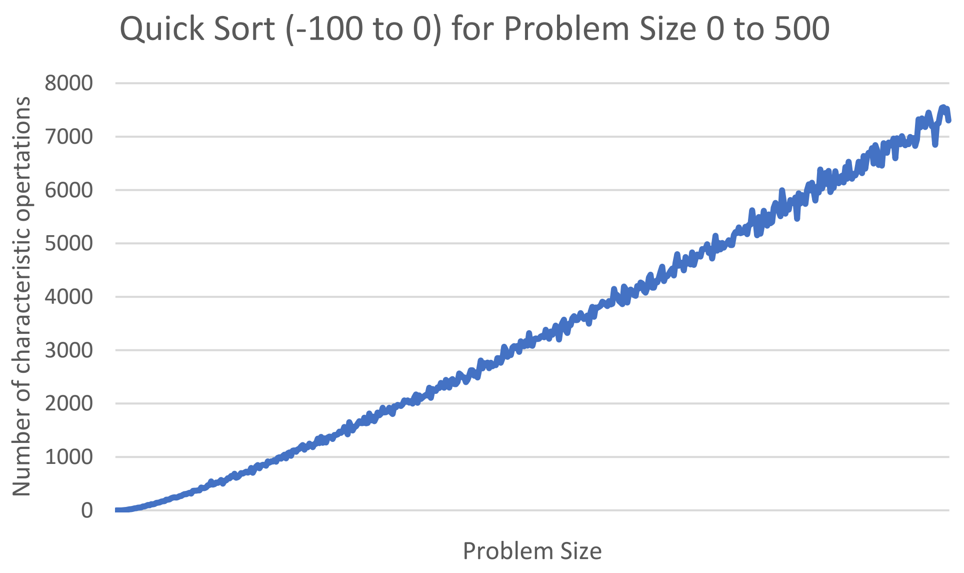
The output charts of the results will be compared to this chart:



Results – time complexity

Integers -100 to 0 for problem size 0 to 500





We can see that **O(n) or linear** time complexity for Count Sort over this range of input integers is a good fit.

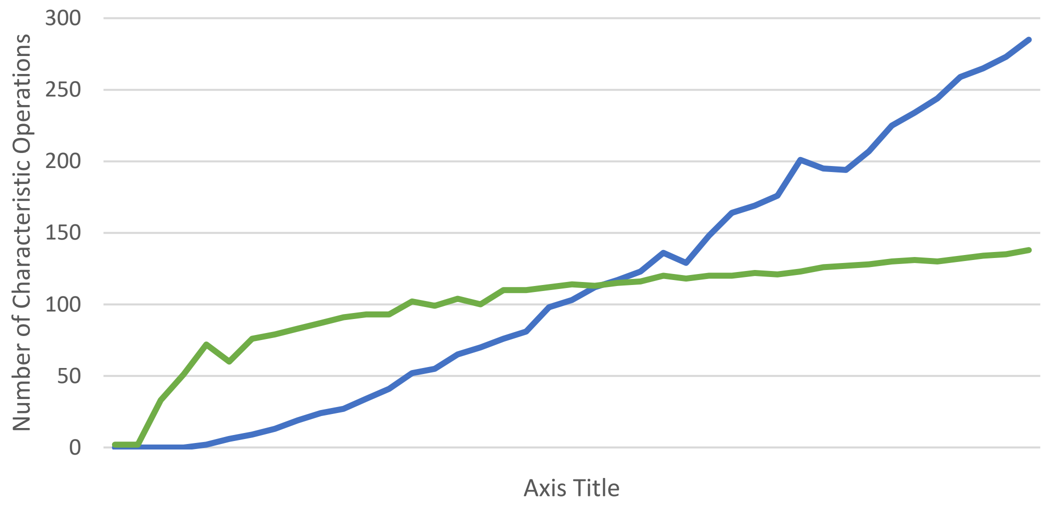
We can see that **O(n log n) or log linear** time complexity for Quick Sort over this range of problem size is a good fit.

Results – comparison

Integers -100 to 0 for problem size 0 to 40

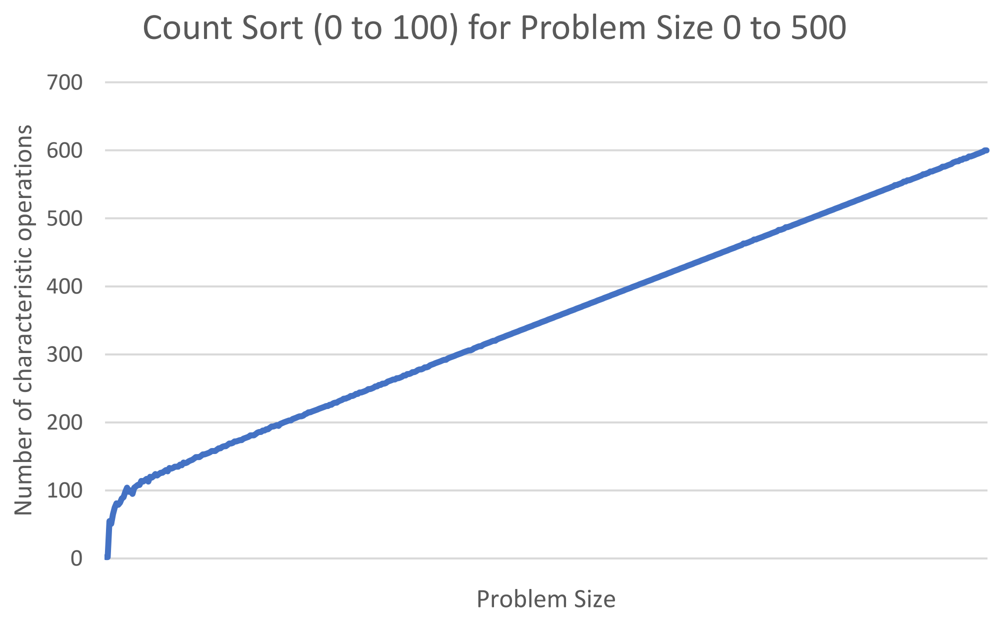
The following chart shows the time complexity results for Count Sort (**green**) and Quick Sort (**blue**) together over this range of potential input integers and it can readily be seen that Quick Sort is more efficient than Count Sort until the problem size reaches about 22 after which Count Sort becomes more efficient.

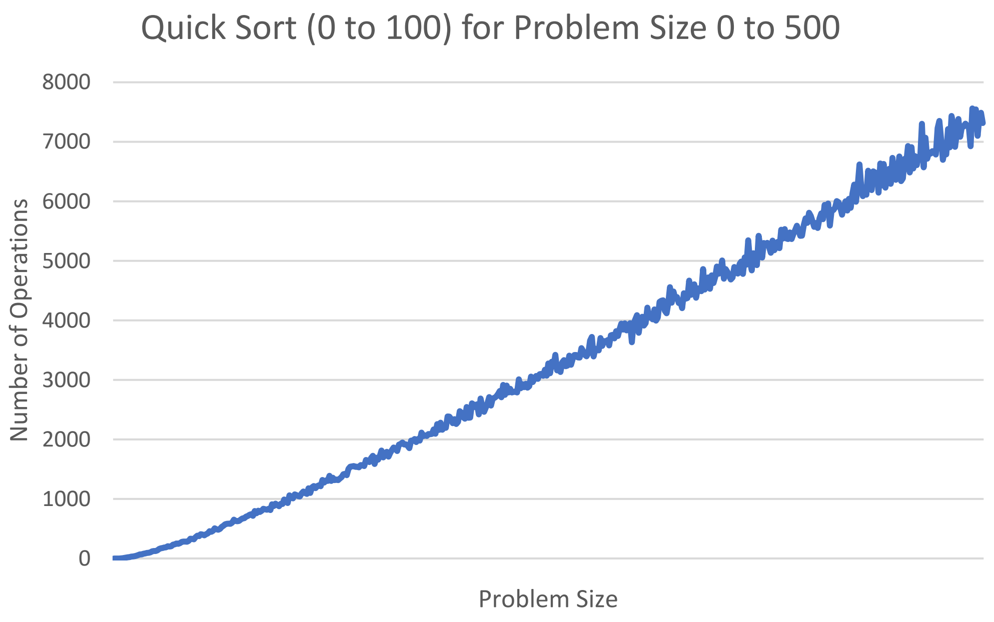
CS v. QS (-100 to 0) for problem size 0 to 40



Results – time complexity

Integers 0 to 100 (the natural numbers including 0) for problem size 0 to 500





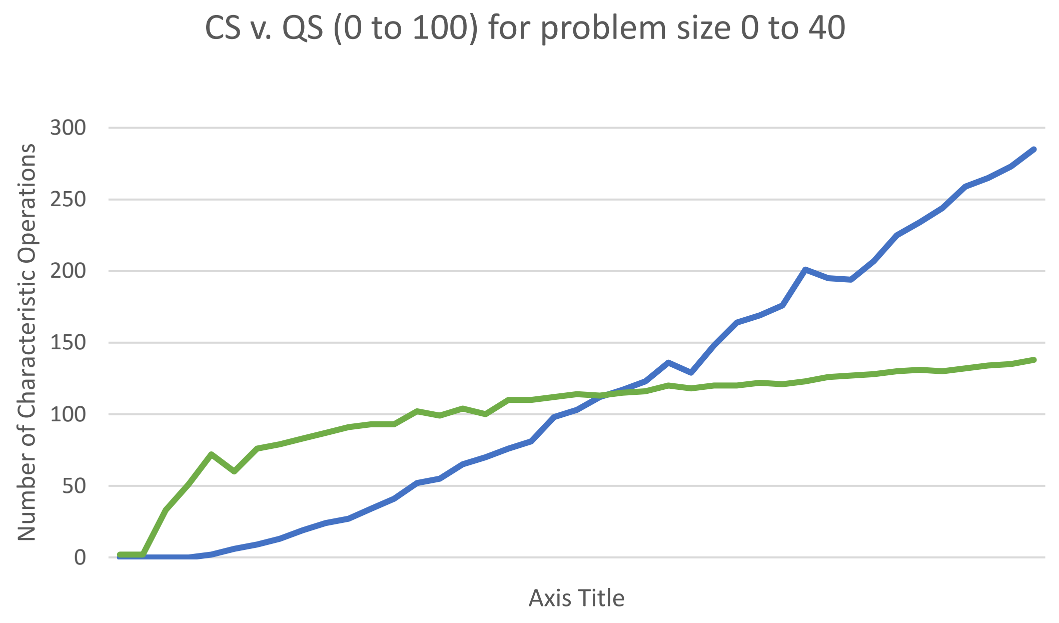
We can see that **O(n) or linear** time complexity for Count Sort over this range of input integers is a good fit. This chart is almost identical to the last chart for Court Sort potential input integers -100 to 0.

We can see that **O(n log n) or log linear** time complexity for Quick Sort over this range of problem size is a good fit. This chart is almost identical the last chart for Quick Sort potential input integers -100 to 0.

Results – comparison

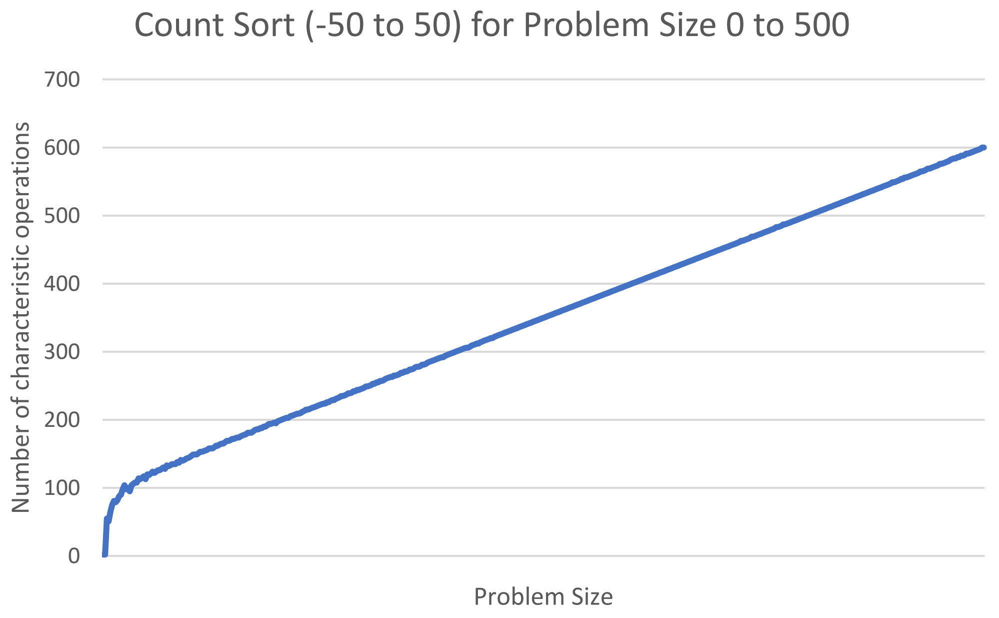
Integers 0 to 100 for problem size 0 to 40

The following chart shows the time complexity results for Count Sort (**green**) and Quick Sort (**blue**) together over this range of potential input integers and again it can readily be seen that Quick Sort is more efficient than Count Sort until the problem size reaches about 22 after which Count Sort becomes more efficient.

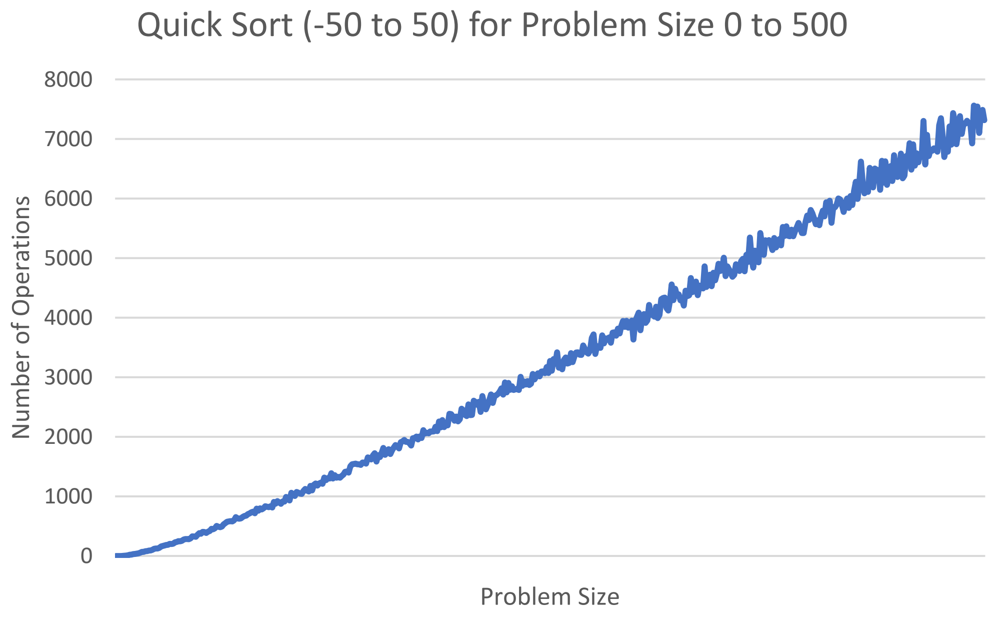


Results – time complexity

Integers -50 to 50 for problem size 500



We can see that **O(n) or linear** time complexity for Count Sort over this range of input integers is a good fit. This chart is almost identical to the last chart for Court Sort potential input integers 0 to 100.

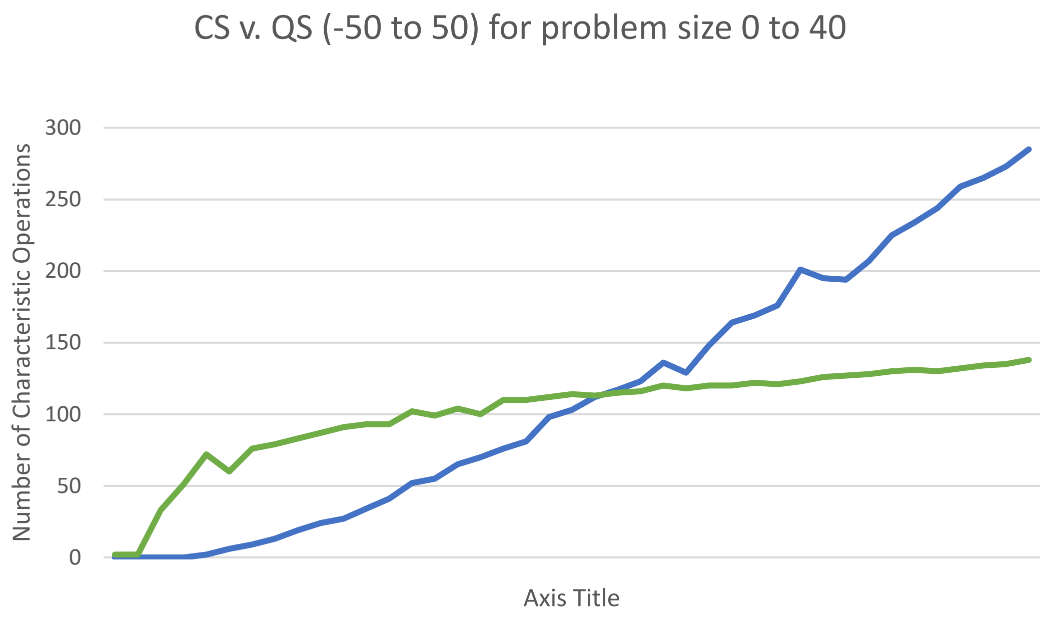


We can see that **O(n log n) or log linear** time complexity for Quick Sort over this range of problem size is a good fit. This chart is almost identical to the last chart for Quick Sort potential input integers 0 to 100.

Results – comparison

Integers -50 to 50 for problem size 0 to 30

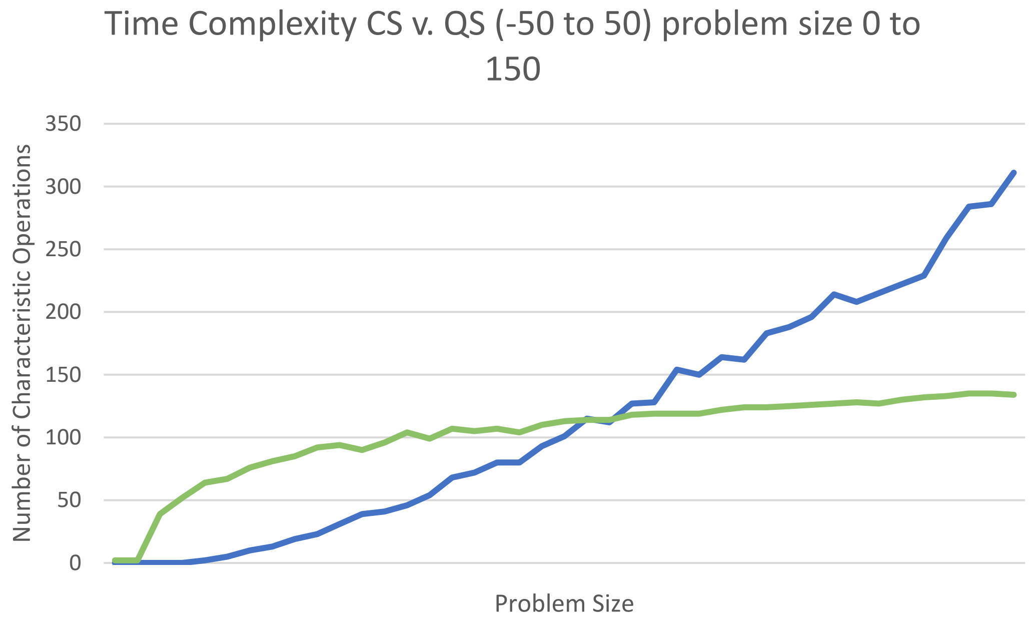
The following chart shows the time complexity results for Count Sort (**green**) and Quick Sort (**blue**) together and it can readily be seen that Quick Sort is more efficient than Count Sort until the problem size reaches about 22 after which Count Sort becomes more efficient.

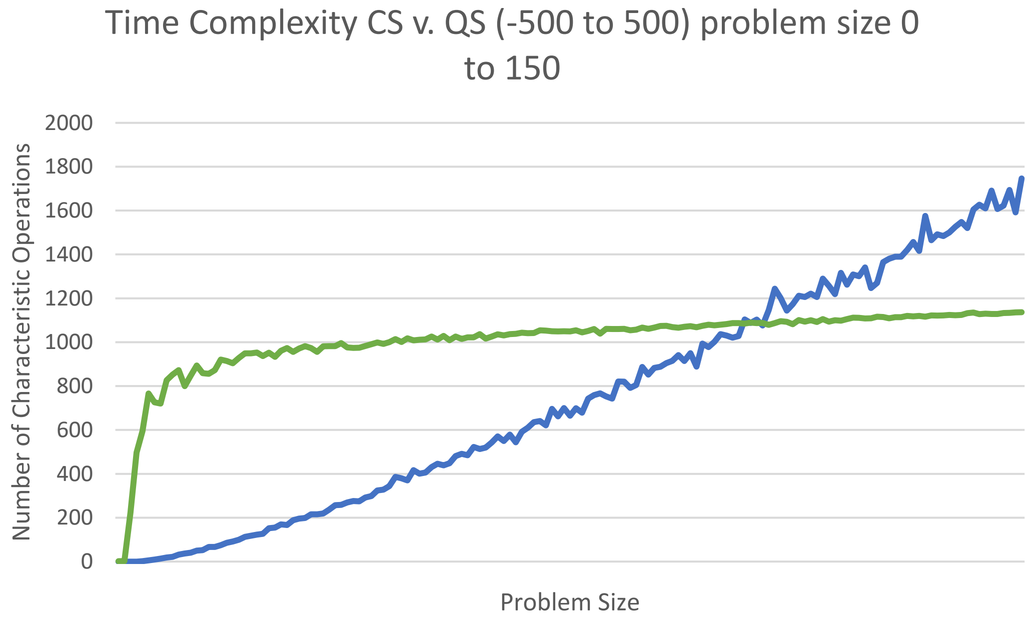


Comparison – Count Sort v Quick Sort

How increase in size of set of element integers affects comparison

In terms of time complexity, the point at which the Count Sort algorithm (**green**) becomes more efficient than Quick Sort (**blue**) is raised as the range of potential element integers broadens although a problem size will always be reached at which Count Sort becomes more efficient.

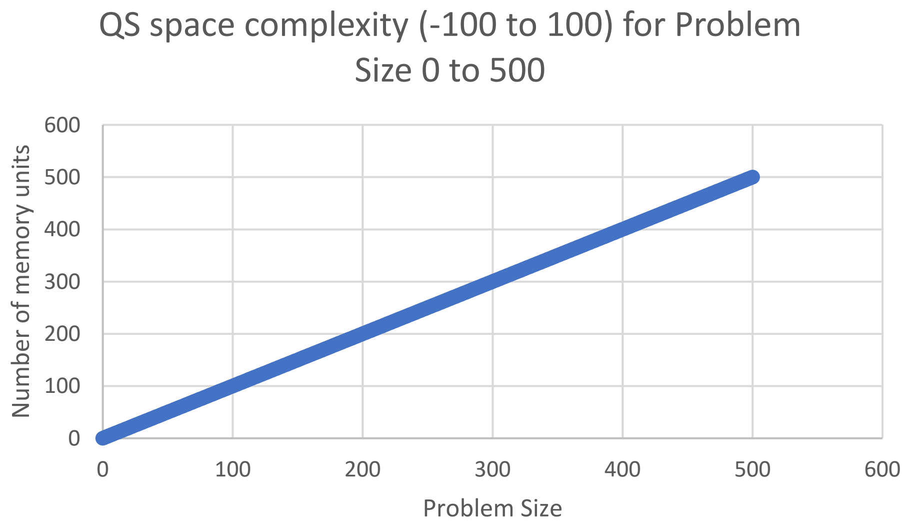


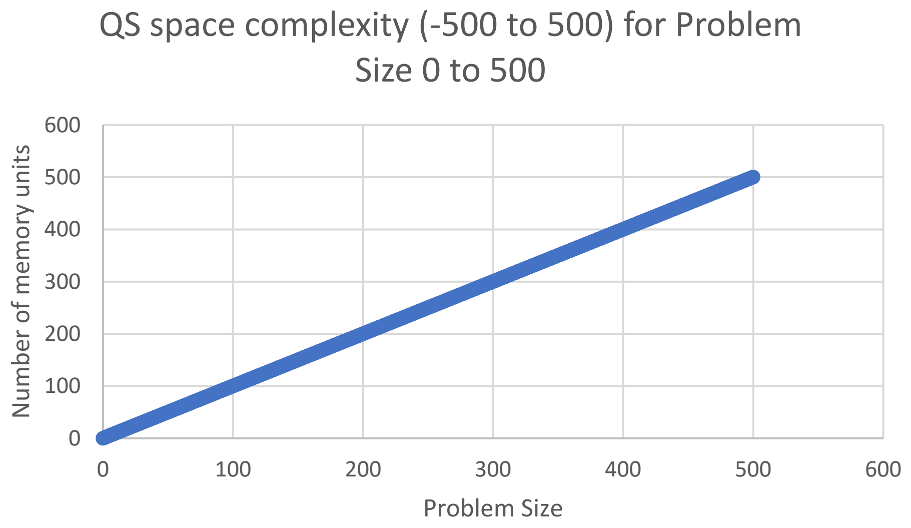


Results – Quick Sort space complexity

Integers -100 to 0, 0 to 100 and -50 to 50

Starting with the Quick Sort this time, the algorithm in question involves only one array which is sorted in-place. The number of operations involved in the assignment and reassignment of Integer variables becomes increasingly insignificant as the problem size tends towards infinity. The range of possible element integers plays no part in the length of this array rather it is equal to the problem size therefore Quick Sort has a space complexity **Big-O rating of O(n)**. In the second chart I increased the set for element integers to (-500 to 500) and it made no difference for memory space.

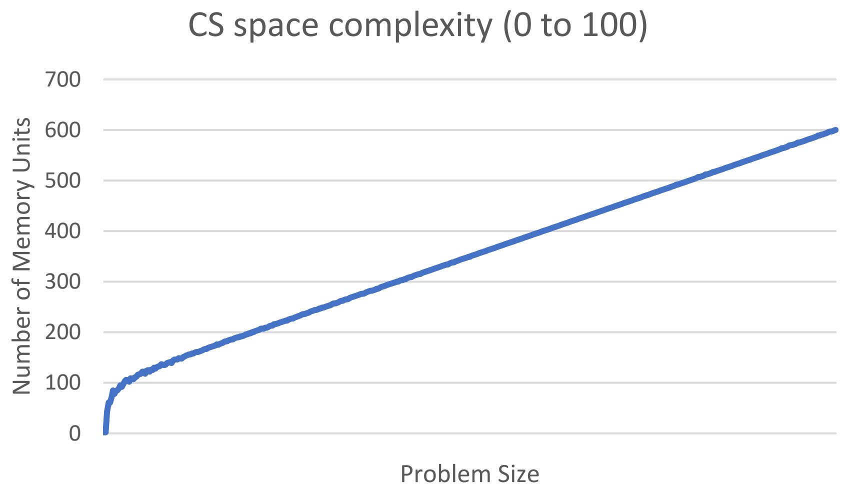


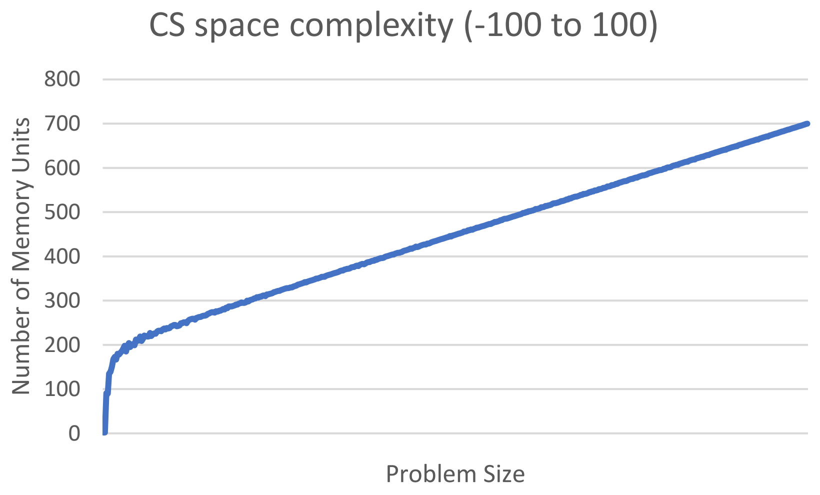


Results – Count Sort space complexity

Integers -100 to 0, 0 to 100 and -100 to 100

The Count Sort algorithm has two arrays one of which is directly proportional to the problem size and the second depends on the range of potential inputs. If we take a max problem size of 500 and run the experimental method to generate all random integers between -100 and 100, the max potential size of the input array is 500 and the max potential size of the frequencies array is 201 giving a max required memory space of 701 units. When one maintains the max problem size but restricts the range of potential integers to (-100 to 0) or (0 to 100) giving a max frequencies array size of 101 the max size of the frequencies array is reduced to 600 units. Regardless of the size of the range of integers the **Big-O space complexity of Count Sort is O(n)**. These figures are reflected in the two following output graphs:

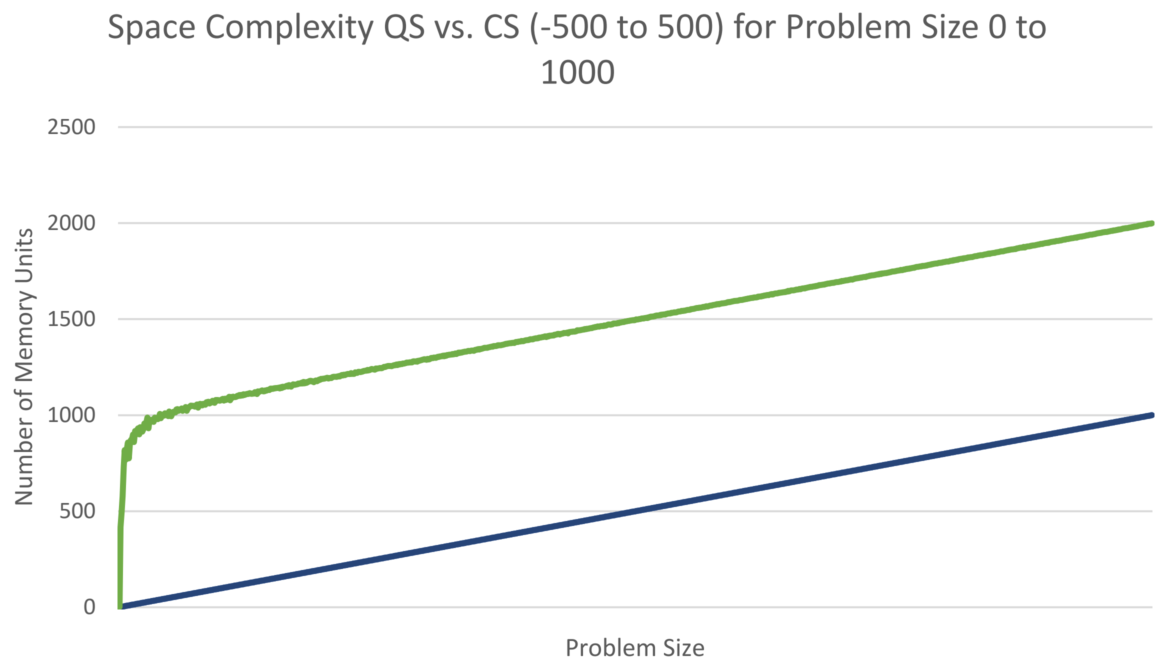




Comparison – Count Sort v Quick Sort

How increase in set of element integers affects comparison

Quick Sort will always be more efficient than Count Sort in terms of space complexity, as discussed already this results from the use of two arrays in the Count Sort algorithm. The two lines develop the same gradient and this continues towards infinity, please see file “Proof of gradient” in results.


Space Complexity QS vs. CS (-500 to 500) for Problem Size 0 to 1000
Number of Memory Units
Problem Size