Module: Computational Thinking with Algorithms

Project Title: Sorting Algorithm Benchmarking

Student: G00267940 - Julian Dunne

Email: g00267940@gmit.ie

Introduction:

Sorting algorithms take a collection of elements and sort in a particular order. Efficient sorting algorithms allow the CPU to sort the data faster, reducing runtime and energy required to complete the task. The larger the collection the greater the number of elements required to be sorted.

As explained on the Real Python website [ref 1], sorting algorithms are useful when:

1. Searching for an element in a list
2. Selecting an item from a list based on its relationship to the other elements in the list
3. Finding duplicates
4. Analysing frequency distribution

Depending on the collection to be sorted, certain algorithms are better suited than others. The concept of sorting and certain sorting algorithms is to be discussed in this report. A comparison of algorithm execution time when sorting a pseudo randomly generated array of elements is to be used to demonstrate how execution times when using each algorithm can differ.

Time and space complexity:

Time complexity can be estimated by counting the number of operations performed by the algorithm where each operation takes a fixed amount of time [ref 2].

This constant time factor can be used to determine the worst, average and best-case time complexities of a given sorting algorithm. Time complexity can be expressed as a function of the size of the input and is generally referred to using Big O notation. Big O notation describes increasing function runtimes as input sizes increase. **Table 1.1** includes the relevant runtime complexities for this project.

Table 1.1. Big O Runtime complexity

Description	Big O
The runtime is constant	O(1)
The runtime grows linearly with size of the input	O(n)
The runtime is a quadratic function of the size of the input	$O(n^2)$
The runtime grows linearly as the size of the input grows exponentially	O(logn)

Table 1.2 includes the relevant sorting algorithm time and space complexities for this project.

Table 1.2. Sorting Algorithm Time and Space Complexity

Algorithm	Best Case	Average Case	Worst Case	Space Complexity
Bubble Sort	n	n^2	n^2	O(1)
Selection Sort	n^2	n^2	n^2	O(1)
Merge Sort	n log n	n log n	n log n	O(n)
Quick Sort	n log n	n log n	n^2	n
Counting Sort	n + k	n + k	n + k	n + k

Time complexity is determined by the input size and the number of steps required. Space complexity refers to the total memory space required by the program for its execution [ref 3]. Space complexity can be increased by assigning variables and creating new data structures within the function body.

Sorting algorithm performance:

The selection of a particular sorting algorithm can depend on the type of data to be sorted and algorithm performance is an important part of data processing especially in situations where real time data is being reported [ref 4].

In-place sorting:

In-place sorting algorithms use a fixed additional amount of space to the array to be sorted and is irrespective of the input size. In-place sorting is useful when dealing with extremely large data sets and extra space is difficult to work with [ref 6]. Understanding how an in-place algorithm executes can be summarized as follows [ref 7]:

"The input is usually overwritten by the output as the algorithm executes. An in-place algorithm updates its input sequence only through replacement or swapping of elements".

Stable sorting:

A sorting algorithm is stable if the order of two objects with equal keys in the unsorted array appear in the same order in the sorted array [ref 5]. Stability is not so much an issue where data is an array of integers or if every element in the array is different. Unstable sorting algorithms can be changed to stable algorithms. This can be done by modifying the key comparison operation to consider position as a factor for objects with equal keys [ref 5].

Comparison based sort:

Comparison based sorting algorithms use comparison operations to determine which element should appear first when the collection is sorted.

Comparison based sorting algorithms can never have a worst-case time better than O(n log n) with n the size of the array to be sorted [ref 8]. Comparison based sorting algorithms are suitable for sorting diverse types of input data.

Non-comparison based sort:

Non-comparison based sorting algorithms use internal characters to rearrange the values of an array into the correct order [ref 9]. This method is not limited to n log n performance as is the case with comparison based sorting. Non comparison based sorting algorithms are also known as Linear sort algorithms where sorting is done in O(n) time. Certain assumptions are made about the data when sorting so that it is not necessary to go through the comparison decision tree as is done for comparison sort [ref 10].

Bubble Sort:

Bubble Sort consists of two nested for loops where in the algorithm performs element comparisons throughout the entire collection. It is one of the simpler and easier sorting algorithms to understand however it is slow to sort as multiple passes must be made through an unsorted list comparing each individual element and swapping elements that are out of order. After each iteration the largest element "bubbles up" to its correct position. Each iteration reduces the number of elements to be sorted and the number of steps in that iteration.

Applying the Bubble Sort algorithm to a collection that is already sorted would give a runtime complexity of O(n) as each element is already sorted so only a single iteration is required. However

for a large unordered collection the average to worst case complexity of $O(n^2)$ means the number of iterations required to sort the collection grows rapidly making this algorithm unsuitable for sorting large collections efficiently [ref 11].

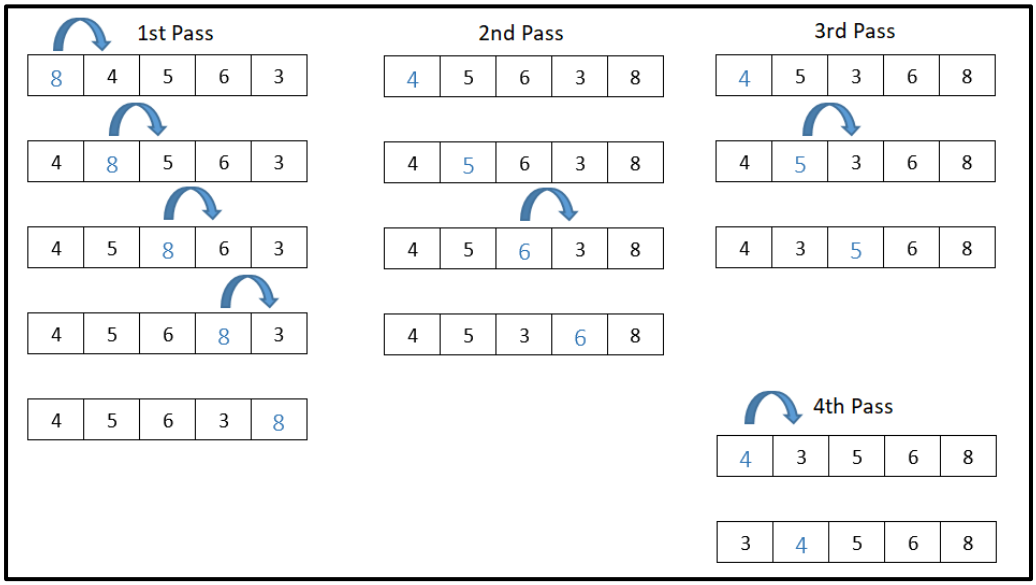

Figure 1: Bubble Sort

Selection Sort:

Selection Sort is an in-place sorting algorithm that separates the unordered collection into a left and a right side. The left side is the sorted side and the right side the unsorted. The left side starts with the index[0] element in the collection and it is compared with all other elements to see which element meets the sorting condition (e.g. smallest value when sorting integers in ascending order). If the reference element is not the smallest value then it is swapped with the element that meets this condition and this element is then sorted in position in the left side collection. After each pass the left side boundary increases by one until all elements have been moved in from the right side.

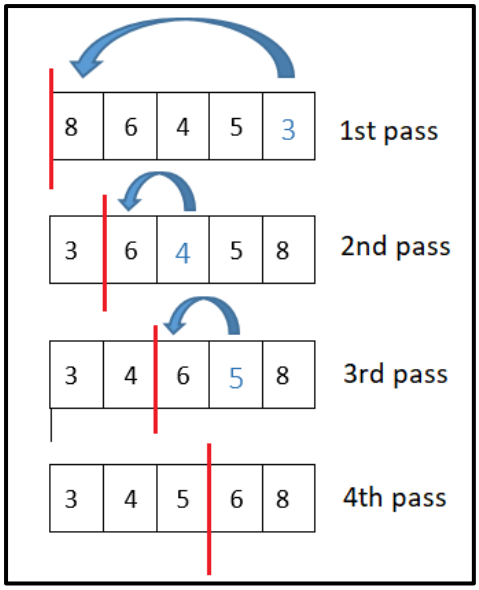

Figure 2: Selection Sort

Merge Sort:

Merge Sort is based on the divide and conquer technique. Using recursion the collection of unordered elements are broken down into smaller "ordered" collections. The break condition for the recursive loop is a collection of less than 2 elements. Each element is then merged back together producing the single sorted collection.

Merge Sort combines both input and output arrays by looking at each element at most once. The collection is split recursively into smaller sub-collections. As the function is called for each half it provides a best possible worst case runtime of O(nlogn). This is the best possible performance for a comparison based sorting algorithm.

An advantage of Merge Sort is the speed at which sorting can be achieved however a limiting factor is the additional memory required when sorting. Required memory can be a limiting factor when sorting large collections.

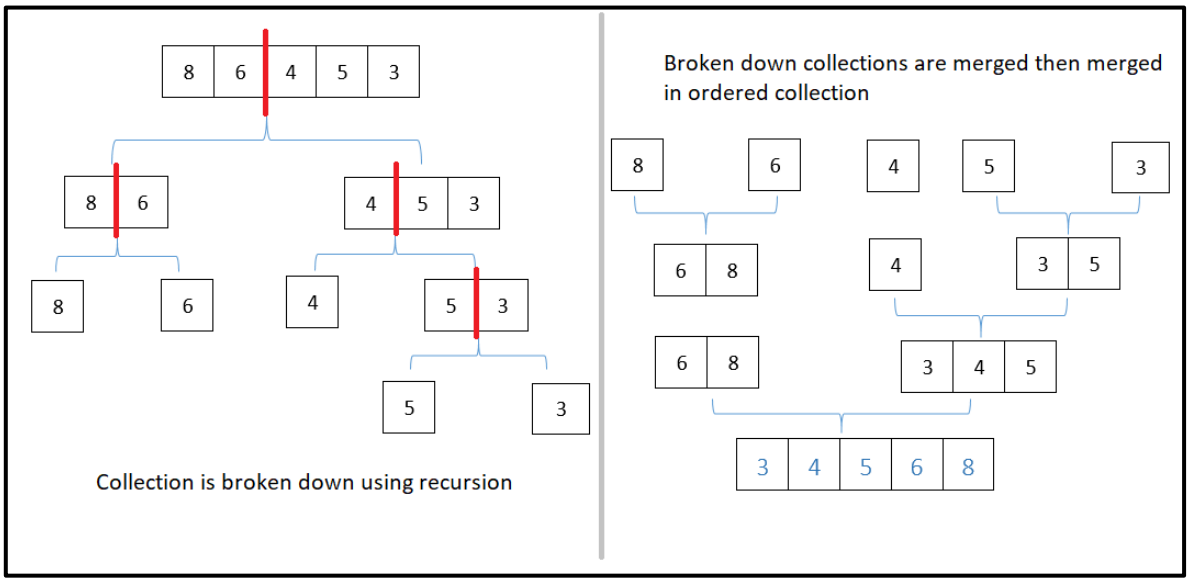

Figure 3: Merge Sort

Quick Sort:

Quick Sort also utilises the divide and conquer technique and divides the unsorted collection. This is done by selecting an element in the collection and setting it as the "pivot". Elements in the collection are then compared with the pivot and separated into "lower than" or "greater than" sub-collections accordingly. Using recursion the same procedure is then applied to the sub-collections with the pivot positioned correctly for the final sorted list. The break condition for the recursive loop is a when both the sub-collections have less than 2 elements each.

The efficiency of the Quick Sort algorithm depends on the pivot selected and the order of the collection. The algorithm performs best when the pivot value selected is the median value of the collection for each partition. This best case scenario provides a complexity of O(nlogn). However if the pivot value is the smallest or largest of the elements then this would be the worst case as the number of partitions required grows $O(n^2)$. Forcing the algorithm to choose the median value would therefore optimise performance however it is seldom implemented as a randomly selected pivot in general meets the required performance level [ref 11].

Similar to Merge Sort, Quick Sort is fast to sort however the required memory to sort is again a limiting factor when sorting large collections.

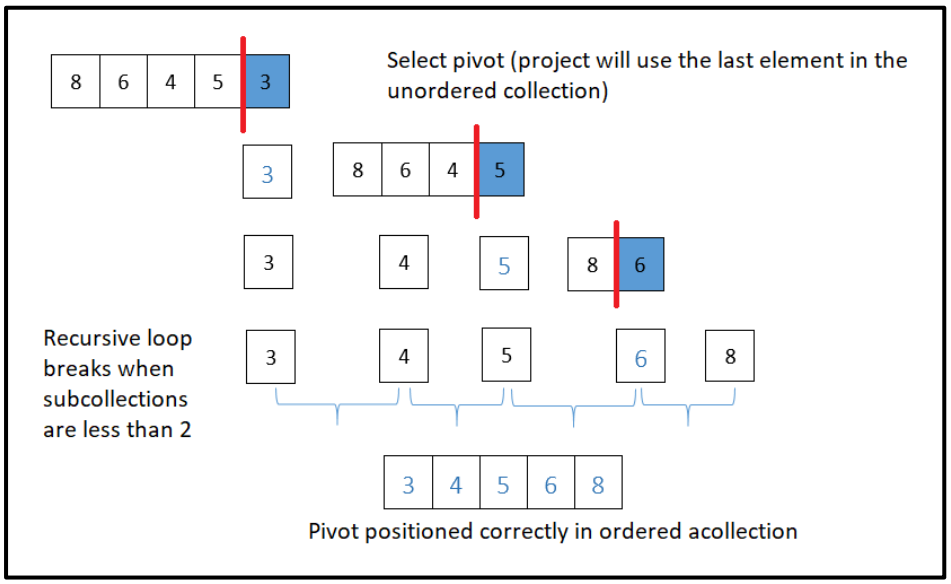

Figure 4: Quick Sort

Counting Sort:

The algorithms discussed until now are all comparison based sorting algorithms and compare all elements in a collection against one another. Comparing each element can result in a greater number of steps to be performed by the algorithm.

Counting Sort is a non-comparison based sorting algorithm that can sort a collection of elements in close to linear time. This is done by counting the frequency in which each element occurs in a collection. The count is stored in an auxiliary array and sorting is done by mapping the count as an index of the auxiliary array [ref 12].

Counting sort is most efficient when used on collections consisting of elements of smaller integers with multiple counts. When sorting collections containing extremely large performance levels are reduced as an auxiliary array of the size range is also required.

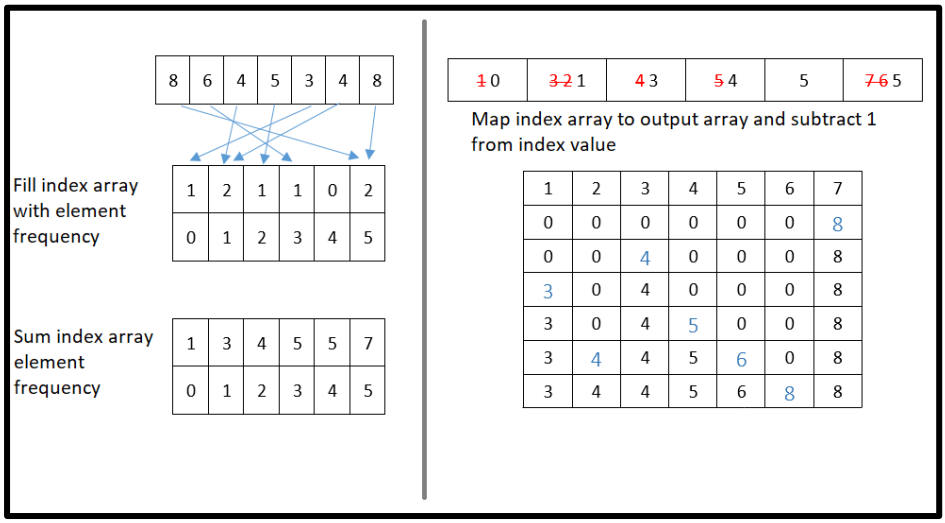

Figure 5: Counting Sort

Implementation and Benchmarking:

The sorting algorithms discussed above have been implemented and run multiple times so as to calculate the benchmark average execution time when sorting a pseudo randomly generated array of integers. Each algorithm was run ten times for each input size with the average algorithm execution time calculated. **Table 1.3** and **Figure 6** below show the average sorting times for arrays of given input size. The values in each collection sorted is in the range of one to one hundred.

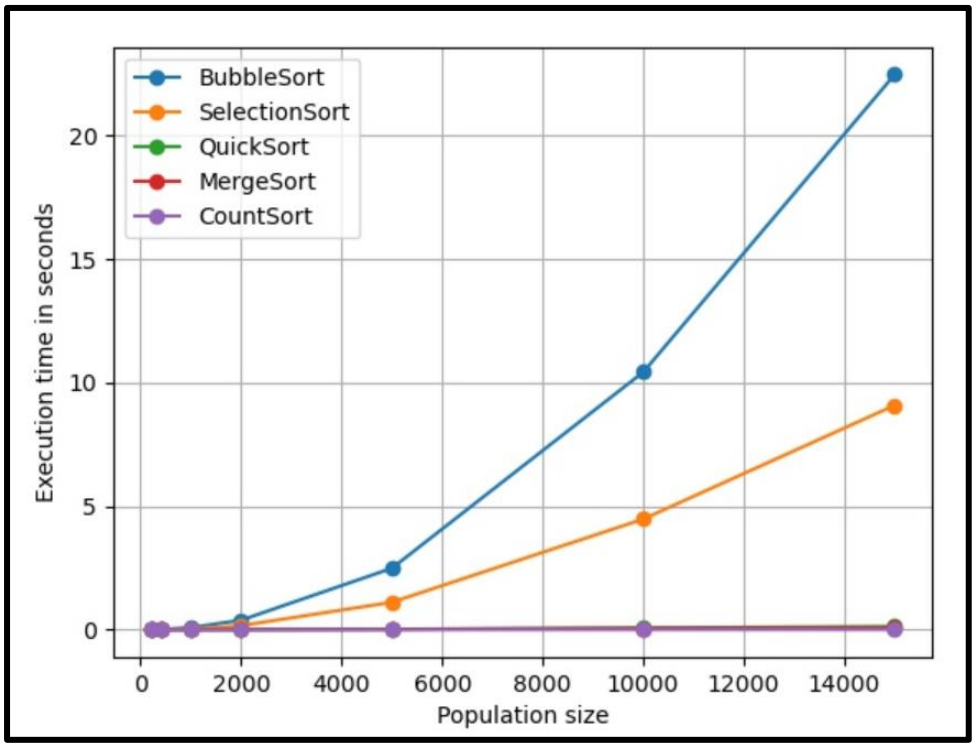

Figure 6: Graph of sorting algorithm average benchmark time when sorting integers in range 1 – 100

Size	BubbleSort	SelectionSort	QuickSort	MergeSort	CountSort
200	0.003	0.001	0	0	0
400	0.011	0.005	0.001	0.001	0
1000	0.086	0.035	0.002	0.004	0
2000	0.385	0.165	0.004	0.009	0.001
5000	2.489	1.111	0.017	0.02	0.003
10000	10.434	4.48	0.073	0.058	0.007
15000	22.481	9.088	0.131	0.076	0.008

Table 1.3: Table of sorting algorithm average benchmark time when sorting integers in range 1 - 100

As expected the Bubble Sort algorithm is the slowest to sort with Selection Sort next. This is due to the number of inversions required to sort the array. Quick Sort and Merge Sort are significantly faster with Merge Sort being the fastest of the comparison based sorting algorithms to execute completing in O(nlogn) time.

Counting Sort which is a non-comparison based sorting algorithm and can run in near linear O(n) time the algorithm requires the least number of steps to execute and is the fastest at sorting of all of the chosen algorithms. For comparison purposes an additional test was run using collections

containing integers in the range of one to one hundred thousand to see if there was a noticeable difference on algorithm execution times.

Size	BubbleSort	SelectionSort	QuickSort	MergeSort	CountSort
200	0.004	0.002	0	0.001	0.022
400	0.019	0.009	0.001	0.002	0.022
1000	0.079	0.034	0.001	0.003	0.017
2000	0.345	0.139	0.003	0.008	0.017
5000	2.311	0.834	0.007	0.019	0.019
10000	9.345	3.96	0.017	0.046	0.023
15000	21.617	9.074	0.055	0.075	0.026

Table 1.4: Table of sorting algorithm average benchmark time when sorting integers in range 1 - 100000

Interestingly the Quick Sort comparison based sorting algorithm execution time noticeably reduced. An explanation for this might be that the Quick Sort algorithm using the divide and conquer technique had a wider range of numbers which allowed for better comparison pivot values which reduced the number of steps required to break down the unordered array.

The primary intention of running the additional benchmark test was to measure the difference in execution time of the Counting Sort algorithm. As the index range grew significantly it was expected that the sorting time would also increase which is what has happened. Although Counting Sort still executes in the least amount of time, performance and efficiency falls as the size of the values increase. **Table 1.4** execution times show reduced execution times for Quick Sort and increased execution times for Counting Sort. Demonstrating that depending on the collection of elements to be sorted, certain sorting algorithms can be better suited than others.

References:

[ref 1] Real Python sorting algorithms webpage: https://realpython.com/sorting-algorithms-python/#the-importance-of-sorting-algorithms-in-python

[ref 2] Wikipedia Time Complexity webpage: https://en.wikipedia.org/wiki/Time_complexity

[ref 3] Pythonguides sorting algorithms webpage: https://pythonguides.com/sorting-algorithms-in-python/#:~:text=Python%20sorting%20algorithms%20time%20complexity%20The%20efficiency%20of,space%20required%20by%20the%20program%20for%20its%20execution.

[ref 4] Online pdf on sorint algorithms: https://www.ijltemas.in/DigitalLibrary/Vol.6Issue6/39-41.pdf

[ref 5] Geeksforgeeks stability sorting algorithms webpage: https://www.geeksforgeeks.org/stability-in-sorting-algorithms/

[ref 6] Planetmath in-place sorting algorithms webpage: https://planetmath.org/inplacesortingalgorithm#:~:text=A%20sorting%20algorithm%20is%20said%20to%20be%20in-placeif,allow%20any%20polynomialin%20log%E2%81%A1nin%20order%20for%20to%20qualify.

[ref 7] Wikipedia in-place sorting algorithms webpage: https://en.wikipedia.org/wiki/In-place_algorithm

[ref 8] Online non-comparison sorting algorithm webpage: http://pages.cs.wisc.edu/~paton/readings/Old/fall08/LINEAR-SORTS.html

[ref 9] Big-O non-comparison sorting algorithm definition webpage: https://big-o.io/algorithms/non-comparison/#:~:text=Non-Comparison%20sorting%20algorithms%20are%20algorithms%20that%20use%20the,values%20of%20an%20array%20into%20the%20correct%20order

[ref 10] Youtube video on non-comparison based sorting algorithms: https://www.youtube.com/watch?v=4NzZMHlZTEo

[ref 11] Real Python sorting algorithm comparison webpage: https://realpython.com/sorting-algorithms-python/

[ref 12] Programiz webpage on the Counting Sort algorithm : https://www.programiz.com/dsa/counting-sort#:~:text=Counting%20sort%20is%20a%20sorting%20algorithm%20that%20sorts,of%20the%20auxiliary%20array.%20How%20Counting%20Sort%20Works%3F