Computational Thinking and Algorithm – Project report

Luc McAuley – G00388057

Table of Contents

[Introduction 1](#_Toc71924340)

[The Sorting Algorithms 4](#_Toc71924341)

[Bubble Sort 4](#_Toc71924342)

[Figure 2-1: Bubble Sort diagram 5](#_Toc71924343)

[Insertion Sort 6](#_Toc71924344)

[Figure 2-2: Insertion Sort Diagram 6](#_Toc71924345)

[Counting Sort 7](#_Toc71924346)

[Figure 2-3: Counting Sort Diagram 7](#_Toc71924347)

[Quick Sort 8](#_Toc71924348)

[Figure 2-4: Quick Sort Diagram 9](#_Toc71924349)

[Merge Sort 10](#_Toc71924350)

[Figure 2-5: Merge Sort Diagram 11](#_Toc71924351)

[Results and Implementation Discussion 12](#_Toc71924352)

[Figure 3-1: Table of Running Times 13](#_Toc71924353)

[Figure 3-2: Graph of Running Times 13](#_Toc71924354)

[Bibliography 15](#_Toc71924355)

# Introduction

In order to analyse and understand sorting algorithms, the concept of an algorithm must first be defined. First appearing in mathematics (Brookshear & Brylow, 2015), algorithms are procedures which processes an input and returns an output (Cormen *et al*., 2009). Algorithms are often used as a tool to solve a well-defined problem with well-defined input and output values (Cormen *et al*., 2009). Thus, with a well-defined input and output, sorting algorithms fall under this category.

As defined by Sedgewick & Wayne (2011), a sorting algorithm is an algorithm which takes an unordered input and rearranges it to provide a pre-defined sorted output. These algorithms are often attributed to playing a role in important developments in science (Harel & Feldman, 2004), especially in early days where it was estimated that 30% of computer cycles were dedicated to processing sorting algorithms (Sedgewick & Wayne, 2011). Sorting algorithms can be found in many disciplines such as linguistics, astrophysics and weather predictions among others (Sedgewick & Wayne, 2011).

Sorting algorithms can be performed using comparator functions. Most sorting algorithms use comparator functions, that is a function that compares two different elements from the input (Harel & Feldman, 2004 and Goodrich & Tamassia, 2004). These functions compare two input values with operators such as greater than, less than or equal (Goodrich & Tamassia, 2004). The result of these comparator functions is a boolean value which the sorting algorithm will use to sort the input. Nonetheless, sorting algorithms can also use non-comparison methods.

Non-comparison sorting requires certain information or characteristics to be set up before the input values can be sorted efficiently. These characteristics can include knowledge of the probability distribution of the input keys such as for Bucket Sort or the range of possible values for Counting Sort (Cormen *et al.,* 2009). Visibly, the prior knowledge of this information may not always be possible or feasible and thus, cannot be used in every situation. Nonetheless, when these algorithms are successfully implemented, they often beat the worst case scenarios of comparison-based functions (Heineman *et al*., 2016). There are multiple ways in which search algorithms can implement both comparison and non-comparison based functions.

Search algorithms can use iterative or recursive structures. An iterative approach consists of going through a list or set of inputs in a looping manner and perform a set of instructions for each input (Brookshear & Brylow, 2015). In contrast, a recursive approach involves the function calling itself (Goodrich & Tamassia, 2004). Recursive algorithms consist of a base case, where the recursion is no longer possible or when a condition is met, ensuring that each step in the recursion addresses a smaller subproblem (Sedgewick & Wayne, 2011). The recursive call can be made at different stages within the function, such as at the end of a set of instructions in the function, called a tail recursion, or multiple recursive calls care made throughout the function, called multiple recursion (Goodrich & Tamassia, 2004). Although more elegant and powerful, not every problem requires recursion (Goodrich & Tamassia, 2004). No matter the way in which the search algorithm is implemented, its run time is of greater importance.

One of the main ways in which sorting algorithms are compared is through their time complexity. Measured by looking at worst, average and best case run times, these provide a way of generally comparing the suitability of sorting algorithms for defined tasks (Brookshear & Brylow, 2015). The standard notation is Ω(n) for best case and O(n) for worst case, where n describes the size of the input (Heineman *et al*., 2016). When assessing the run time for any case, a mathematical model of the run time, the largest order of magnitude is then kept, with any constants or factors removed (Sedgewick & Wayne, 2011). Examples of such run times are constant O(n), logarithmic O(log n) and quadratic O(n2) (Sedgewick & Wayne, 2011).

The best case run time is used to describe the expected rate of increase in run time for an algorithm when the input is most favourable, which rarely occurs (Heineman *et al*., 2016). Nonetheless, it is beneficial to know this scenario when evaluating an algorithm in order to assess its suitability for a given implementation (Heineman *et al*., 2016). Average cases on the other hand is used to describe the expected behaviour of an algorithm on random sets of input and can provide some insight into the use of an algorithm for general purposes where the input data may not be well-known (Goodrich & Tamassia, 2004).

However, the worst case run time is the most commonly used when evaluating an algorithm (Goodrich & Tamassia, 2004). By using the worst case as the basis for sorting algorithm comparison, it allows the developer to focus on choosing the best suited algorithm as well as motivate improvements for available algorithms (Goodrich & Tamassia, 2004). Furthermore, as the worst case is the easiest to estimate, it allows programmers to identify the input properties which perform the worst and to compensate for them (Harel & Feldman, 2004). However, the amount of space a sorting algorithm requires can also be important.

Another concept to consider when looking at algorithms is their space complexity. Often, improvements in running time can come at the expense of the amount of storage or memory space required to complete the task (Brookshear & Brylow, 2015). Although the trade-off between space and time complexity may not be of great importance when looking at small or short input types (Sedgewick & Wayne, 2011), especially considering that modern computers often have large amounts of storage and memory (Heineman *et al*., 2016), the trade-offs must be analysed when input sizes get larger.

The space/time complexity can be part of the considerations of implementing an algorithm. Depending on the situation and the implementation, the performance cost can be offset with large amounts of storage with some sorting algorithms may increase their storage requirements at exponential rates to ensure low time complexities (Brookshear and Brylow, 2015; Harel and Feldman, 2004). As such, one of the main reasons algorithms are studied is to improve on either of these complexities in order to increase efficiency which can lead to the ability to complete tasks that may have been previously considered impossible (Sedgewick & Wayne, 2011). Nonetheless, other aspects of sorting algorithms are equally important during the implementation considerations.

One of the influential characteristics when selecting one algorithm over an another is whether they are stable. A sorting algorithm is considered stable if the keys with the same value appear in the same order once sorted (Cormen *et al.,* 2009). Although at first view, this may not seem important as the values are considered sorted, there may be satellite data associated to these values which may relate to their position in the input (Cormen *et al.,* 2009). Satellite data is data that is associated to the key that is being sorted and generally in practice, the satellite data is moved along with the key (Cormen *et al.,* 2009). However, when sorting through large amounts of data with a large storage or memory footprint, data loss may occur due to changing the pointers of the keys as opposed to moving the entire unit of data (Cormen *et al.,* 2009). Thus, keeping the initial ordering may be beneficial in order to keep the key associated to the satellite data (Goodrich & Tamassia, 2004). Furthermore, in-place sorting algorithms should be considered should there be a high value on the space complexity of a sorting algorithm.

In-place search algorithms are important for situations where memory is limited (Sedgewick &Wayne, 2011). An algorithm is considered in-place if it falls under one of two categories, either requiring a small and defined amount of additional memory to store data outside of the input in order to sort or it requires no additional memory other than for a call stack (Cormen *et al.,* 2009; Goodrich & Tamassia, 2004). Evidently, an in-place sorting algorithm is beneficial when storage or memory is limited, however may require some thought and effort to implement in certain cases (Goodrich & Tamassia, 2004).

This report has defined algorithms and sorting algorithms defined, along with discussing the ways they are implemented and the considerations when deciding which sorting algorithm to implement. The upcoming section will now introduce the five sorting algorithms chosen, namely Bubble Sort, Insertion Sort, Counting Sort, Quick Sort and Merge Sort.

# The Sorting Algorithms

## Bubble Sort

Bubble sort is considered a bad sorting algorithm, to the point it is often not discussed in computer science courses (Harel & Feldman, 2004). This algorithm sorts the input by repeatedly running through each element of the input array, swapping two neighbouring elements to be in the correct order (Brookshear & Brylow, 2015). This leaves the highest element at the end of the array, thus the second run does not need to extend to the last position of the array (Harel & Feldman, 2004). With each traversal, Bubble Sort no longer needs to run the whole array as the last sorted element is in the right position, essentially bubbling to the top (Harel & Feldman, 2004).

Having to run through the array n times, doing n number of swaps at worst, the time complexity of Bubble sort is O(n2), meaning the run time is quadratic (Harel & Feldman, 2004), it has one of the largest run times of the selection of sorting algorithms discussed in this report. With regards to space complexity, it is considered in-place as the sorting can be performed within the input array, thus has a space complexity of O(n) (Harel & Feldman, 2004), meaning the amount of space it requires grows linearly with the input size.

Diagram

Description automatically generated

### Figure 2-1: Bubble Sort diagram

## Insertion Sort

Best suited when the input size is small or nearly sorted, or when the size of the code needs to be small (Heineman *et al*., 2016), this algorithm sorts within itself, similarly to Bubble sort (Brookshear & Brylow, 2015). This sorting algorithm runs through the input array, selecting a pivot and compares the elements in the array which precede it (Brookshear & Brylow, 2015). Any element larger than the current pivot is moved to the right hand side of the pivot (Brookshear & Brylow, 2015).

Using mathematical analysis, the worst case run time complexity of Insertion sort is , giving us a worst case run time of O(n2) (Brookshear & Brylow, 2015). The space complexity of Insertion sort is the same as for Bubble sort, namely O(n) as it can sort in-place (Brookshear & Brylow, 2015). This low space complexity comes at the cost of speed, as each element lower than the pivot needs to be moved one at a time (Brookshear & Brylow, 2015). Implementations of Insertion sort which can move multiple elements at a time will help the speed of the algorithms, however it will come at the cost of the space complexity (Heineman *et al*., 2016)

Diagram

Description automatically generated

### Figure 2-2: Insertion Sort Diagram

## Counting Sort

Counting sort is a non-comparison based sorting algorithm, requiring prior knowledge of the range of values to be sorted (Cormen *et al.,* 2009). Using a separate array of the size of the expected elements, the Counting algorithm traverses the input values and counts the number of times a value appears (Cormen *et al.,* 2009). The position of the counting array corresponds to a value, and the value in that position corresponds to the number of instances it appears (Cormen *et al.,* 2009). At the end of the counting, the counting array is traversed, and the value of each position is then inserted into the output array, with an occurrence of the value in that position (Cormen *et al*., 2009). Most notably, this is a stable sorting array, where the values appear in the output array in the same order as the input array (Cormen *et al*., 2009).

As there are two array traversals, one for counting the input array and one for completing the output array, the time complexity of this sorting algorithm is O(n), that is a linear run time (Cormen *et al*., 2009). Notably, the best, worst and average case all have the same time complexity (Heineman *et al*., 2016). As for the space complexity, there are two arrays of size n, with an additional value for the comparison so there is “2n” space required. Thus, the space complexity is linear, namely O(n).

Diagram

Description automatically generated

### Figure 2-3: Counting Sort Diagram

## Quick Sort

Used when interested in average case run times (Heineman *et al*., 2016), it is the first recursive algorithm that is analysed in this report and is considered in high regard (Harel & Feldman, 2004; Heineman *et al*., 2016). Using a divide and conquer approach (Brookshear & Brylow, 2015), the Quick Sort algorithm selects an element in the array as the pivot. The input array is then divided into two arrays, one containing values larger than the pivot and one containing values less than the pivot. This procedure is performed until the arrays contain two or less values, where they are sorted and merged. This merge and sorting is done for every array until all sub arrays have merged back into the sorted array (Heineman *et al*., 2016).

As the worst case would require quick sort to run through each element in the array in linear time, up to n-1 times, the run time complexity of O(n2) (Heineman *et al*., 2016). Nonetheless, it outperforms the execution times of Insertion sort for larger arrays (Heineman *et al*., 2016). With regards to the space complexity, the sorting array will require to store multiple arrays outside of the input array for each recursive call and thus, the space complexity will grow linearly, namely O(n) (Sedgewick & Wayne, 2011).

Diagram

Description automatically generated

### Figure 2-4: Quick Sort Diagram

## Merge Sort

Noted as one of the fastest sorting algorithms (Brookshear & Brylow, 2015), Merge sort is used when the sorting algorithm needs to be stable (Heineman *et al*., 2016). Similarly to the above Quick sort algorithm, Merge sort also uses the divide and conquer approach of recursion. It involves a similar approach to Quick sort, dividing the arrays into two, however in this case it is done by dividing the array in the middle (Harel & Feldman, 2004). Once the array is length two or shorter, it is sorted and merged with the other array and sorted again. This sort and merge is performed until the output array is returned (Brookshear & Brylow, 2015).

In the worst case, the time complexity of the algorithm will be O(n log(n)), which is considered the best possible worst-case run time for comparison-based sorting algorithms (Sedgewick & Wayne, 2011). However, due to the recursive nature of Merge sort, there will be a large number of arrays with up to n merges required (Heineman *et al*., 2016). Thus, the space complexity of Merge Sort is O(n) in the current implementation.

Diagram

Description automatically generated

### Figure 2-5: Merge Sort Diagram

# Results and Implementation Discussion

Apart from the first two sorting algorithms, namely Bubble Sort and Insertion Sort, the sorting algorithms were implemented through the descriptions and material provided in the Computational Thinking and Algorithms lecture notes.

The implementation process involved ensuring that each sorting algorithm was working correctly one by one, with randomly generated arrays using code provided in the project requirements. Using logs printed to the console, the algorithms’ functions were analysed and confirmed to be working as expected. Quick Sort and Merge Sort were the more difficult to implement due to the recursion. Finding base cases as well as handling fringe cases was difficult, appearing with larger sample input sizes. One notable example is the handling of the median values in Quick Sort and the handling of the base case for Merge Sort for arrays of size 2 and 1 separately.

The next step was to ensure the same randomly generated array was used for each of the sorting algorithms. At first using the same instance of the array, the run times did not seem in line with expectations. Upon further investigation, the instance of the randomly generated array was sorted, resulting in the already sorted array being sent to the algorithms. The inelegant solution was to create individual arrays which were copies of the original randomly generated array. As this error was detected close to the deadline after everything else has been implemented, a more elegant solution was not found in time as work had to be done to ensure that the sorting arrays were in fact working correctly.

Next, the way the times were recorded was implemented. Again, using code provided in the project description, each run of the sorting algorithm was added to an array. Each sorting algorithm was run 10 times for each sample size, with each run time added to the array. Each value was then divided by the number of runs in a separate method.

With the sorting algorithms created and run times recorded, it was now time to look at the output table. First removing all of the debugging elements, a two-dimensional array was created as well as methods to populate the names of the sorting arrays and the number of runs. After the 10 runs for each size, the respective column is populated with the average runs. Once every column is populated, the table is displayed. After some tinkering with formatting, the use of the /t tag after each element in the array ensured that the display looks legible.

Upon checking the code and verifying the functionality, the aforementioned issue with the randomly generated arrays required amendments to be made. Correction work had to be done on Quick Merge with regards to handing the median which caused issues due to an original misunderstanding of the algorithm. Originally believing that the subarrays would be of the same size, this was working as the array being passed through was already sorted. However, upon further inspection and further analysis of the description, the algorithm was changed to create arrays of different sizes to ensure the values below and above the pivot would be in separate arrays. Further handling of the pivot values was implemented. As there was no description as to the way in which it should be handled was described should there be multiple instances and was separated between both sub arrays.

Below is a table of the results from one of the runs with the final code, along with the corresponding graph.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Size** | **100** | **250** | **500** | **750** | **1000** | **1250** | **2500** | **3750** | **5000** | **6250** | **75000** | **8750** | **10000** |
| **Bubble Sort** | 0.300 | 0.464 | 1.375 | 2.752 | 0.652 | 0.902 | 2.848 | 5.953 | 10.202 | 15.576 | 22.493 | 31.202 | 54.092 |
| **Insertion Sort** | 0.032 | 0.010 | 0.025 | 0.044 | 0.077 | 0.114 | 0.429 | 0.945 | 1.667 | 2.584 | 3.710 | 5.049 | 6.581 |
| **Merge Sort** | 0.220 | 0.065 | 0.086 | 0.154 | 0.207 | 0.416 | 1.278 | 1.710 | 2.985 | 4.336 | 6.257 | 8.374 | 10.555 |
| **Count Sort** | 0.379 | 0.145 | 0.104 | 0.120 | 0.032 | 0.039 | 0.076 | 0.104 | 0.145 | 0.173 | 0.208 | 0.245 | 0.280 |
| **Quick Sort** | 0.111 | 0.047 | 0.097 | 0.354 | 0.191 | 0.191 | 0.238 | 0.426 | 0.481 | 0.660 | 1.087 | 1.319 | 1.163 |

### Figure 3-1: Table of Running Times

**Chart

Description automatically generated with medium confidence**

### Figure 3-2: Graph of Running Times

When looking at the results of the runs, the first point of discussion is the very long run times for the first sample size of 100. It is much greater than it should be and for Count Sort, it is greater than for the sample size of 10000. Furthermore, Count Sort when looked at individually, is U shaped which seems somewhat off, however due to the rapid run times of less than a millisecond, this could be attributed to some randomness.

However, especially with Bubble sort, the other run times seem to be in line with the available literature and analysis from the second section of this report. With the worst run time and apparent quadratic run time, it is understandable why Bubble Sort is not used in certain computer science courses. Insertion sort has quite a good run time. Similarly with Count Sort, which was pretty much flat for the whole run, there is an visible benefit to non-comparison based sorting algorithms.

Surprisingly, Merge sort was not closer to Insertion sort. Due to the fact that Merge Sort provides the optimal situation for Insertion Sort, namely short or nearly sorted input values, the recursive aspect of dividing and conquering did not have the added benefit expected from a run time point of view.

Although outside of the scope of this assignment, it would be interesting to compare the space complexities of these sorting algorithms for the same run to analyse the relationship – or cost – of these run times with the amount of space they use.

# Bibliography

Brookshear G.J. & Brylow D. (2015) *Computer Science: An Overview*, 12th edn. Pearson Education Limited, England.

Cormen T.H., Leiserson C.E., Rivest R.L. & Stein C. (2009) *Introduction to Algorithms*, 3rd edn. Massachusetts Institute of Technology, Massachusetts.

Goodrich M.T. & Tamassia R. (2004) *Data Structures & Algorithms in Java*, 4th edn. John Wiley & Sons, Inc., Crawfordsville, Indiana.

Harel D. & Fieldman Y. (2004) *Algorithmics: The Spirit of Computing*, 3rd edn. Pearson Education Limited, England.

Heineman G.T., Pollice G. & Selkow S. (2016) *Algorithms in a Nutshell*, 2nd edn. O’Reilly Media Inc., United States of America.

Sedgewick R. & Wayne K. (2011) *Algorithms*, 4th edn. Pearson Education, Boston, Massachusetts.