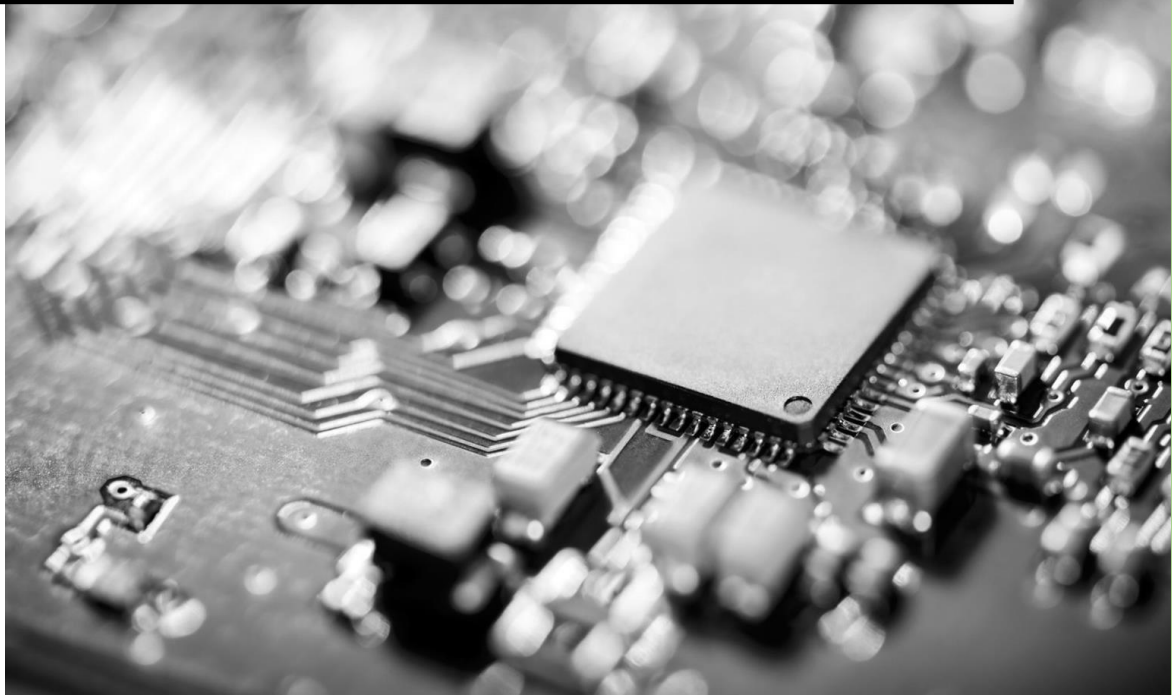


2021

Sorting Algorithms Benchmarking Report



Máire Murphy

G00375722

5/10/2021

Contents

| | | |
|-------|---|----|
| 1 | Introduction | 4 |
| 1.1 | Complexity (Time & Space) | 4 |
| 1.1.1 | Orders of Growth | 5 |
| 1.2 | Performance | 6 |
| 1.3 | In-place Sorting | 6 |
| 1.4 | Stable Sorting | 7 |
| 1.5 | Inversions | 7 |
| 1.6 | Comparator Functions | 7 |
| 1.7 | Comparison-based\Non-Comparison-based Sorts | 7 |
| 1.8 | Sort keys and satellite data | 7 |
| 2 | Sorting Algorithms | 7 |
| 2.1 | Bubble Sort..... | 7 |
| 2.1.1 | Pseudo Code (Non-optimised) | 8 |
| 2.1.2 | Bubble Sort Examples | 8 |
| 2.1.3 | Pseudo Code Optimised | 10 |
| 2.2 | Insertion Sort | 10 |
| 2.2.1 | Pseudo Code | 11 |
| 2.2.2 | Insertion Sort Examples | 11 |
| 2.3 | Merge Sort | 12 |
| 2.3.1 | Pseudo Code | 13 |
| 2.3.2 | Merge Sort Examples | 13 |
| 2.4 | Counting Sort | 14 |
| 2.4.1 | Pseudo Code | 15 |
| 2.4.2 | Counting Sort Examples | 15 |
| 2.5 | Tim Sort..... | 16 |
| 2.5.1 | Pseudo Code | 18 |
| 2.5.2 | Tim Sort Examples..... | 18 |

| | | |
|-------|--|----|
| 3 | Implementation & Benchmarking..... | 19 |
| 3.1 | Sorter Class | 20 |
| 3.2 | Runner Class..... | 21 |
| 3.2.1 | Sample Sizes | 21 |
| 3.3 | Experiment Results | 22 |
| 3.4 | Analysis | 23 |
| 3.4.1 | Analysis by Sample Size..... | 23 |
| 3.4.2 | Analysis by Average Runtime All Samples | 24 |
| 3.5 | Evaluation | 24 |
| 4 | Conclusion..... | 24 |
| 5 | Bibliography | 25 |
| | | |
| | Figure 1: Waiting (www.imgflip.com, n.d.)..... | 4 |
| | Figure 2: Time Complexity of Algorithms (Newton, 2017) | 5 |
| | Figure 3: Summary of Algorithm Efficiencies..... | 6 |
| | Figure 4: Stable Sort Example | 7 |
| | Figure 5: Bubble Sort Random {1,7,2,11,5}..... | 9 |
| | Figure 6: Bubble Sort - Pre-sorted Array {1,2, 5, 7, 11} | 9 |
| | Figure 7: Bubble Sort Reversed Array {11,7,5,2,1}..... | 9 |
| | Figure 8: Optimised Bubble Sort Example | 10 |
| | Figure 9: Insertion Sort - Random Array | 11 |
| | Figure 10: Insertion Sort – Previously sorted Array..... | 12 |
| | Figure 11: Merge Sort Random Order Array..... | 13 |
| | Figure 12: Merge Sort Reversed Array..... | 14 |
| | Figure 13: Counting Sort Random array | 16 |
| | Figure 14: Counting Sort Duplicate Values | 16 |
| | Figure 15: Timsort Galloping Example | 19 |
| | Figure 16: Timsort Runs Example..... | 19 |
| | Figure 17: Benchmark Results (1000 - 100,000 samples) | 22 |
| | Figure 18: Benchmark Results (1 - 750 samples) | 22 |
| | Figure 19: Sorting Algorithm Benchmark Graph..... | 23 |

| | |
|--|----|
| Figure 20: Best Worst Per Sample Size | 23 |
| Figure 21: Mean Runtimes..... | 24 |
| Table 1: Common Complexities | 6 |
| Table 2: Bubble Sort Features..... | 8 |
| Table 3: Insertion Sort Features..... | 11 |
| Table 4: Merge Sort Features..... | 12 |
| Table 5: Counting Sort Features..... | 14 |
| Table 6: Timsort Complexities & Features | 17 |
| Table 7 Platform Information | 20 |
| Table 8: Sorter Class Functions | 20 |
| Table 9: Runner Class Declarations..... | 21 |
| Table 10: Runner Class Functions | 21 |

1 Introduction

Sorting data means placing a collection of data (such as an array) into a premediated ordered sequence. Sorted data can be an important prerequisite to retrieve data in a more efficient manner. For example, the contact list on a mobile phone is usually ordered by name alphabetically to help the user find a number quickly. There are various ways to sort data with well-defined implementable instructions; these are known as Sorting Algorithms. Many sorting algorithms exist; examples of well-known algorithms are: Merge Sort and Bubble Sort. A well-designed sorting algorithm should have well defined inputs and output; it should not end up in an infinite loop and should consistently give the correct solution.

Which algorithm to use? The answer is one that best meets the user's requirements/use cases. Use cases could be a combination of the following: the algorithm must be able to handle increasing large amount of data; that it must not reorder same values; be able to function in limited storage\memory execution; take the shortest time taken to execute. Other factors that could influence choosing a particular algorithm are the state of data (nearly sorted) and the size

of data. *'Unfortunately, there is no one best universal Sorting Algorithms for every possible use case'* (Mannion, n.d.). Long execution times can result if the wrong algorithm is used, see *Figure 1: Waiting*. An algorithm's complexity can be analysed mathematically,



Figure 1: Waiting
(www.imgflip.com, n.d.)

independent to a particular platform which is Priori Analysis. Algorithm performance experiments can also be carried out on a targeted platform(s) which is Posteriori Analysis. The analysis and results can indicate which is the best algorithm to pick depending on the use case(s).

This report covers aspects of algorithm analysis that is platform independent such as algorithm complexity and general algorithm categorization. It takes a closer look at these five popular sorting algorithms: bubble sort, Insertion sort, Merge sort, Count sort and Timsort. Then, the five sorting algorithms are tested on a real-world platform and are benchmarked.

1.1 Complexity (Time & Space)

Complexity is a way to measure the efficiency of an algorithm as the data set increases (platform independent). Time complexity analyses the time needed or the number of operations needed to run the algorithm. Mathematical equations are used to calculate the 'best' (Ω Omega notation), 'average' (Θ Theta notation) and 'worst' (Big O notation) cases by counting elementary operations. In the real world the worst-case scenario is usually the most significant. Increasing size of input data can make performance difference more obvious i.e. to execute an algorithm with an input of 10000 elements compared to 10 elements (different orders of magnitude).

Space complexity is the space (memory) that is required by the algorithm to complete. An algorithm that does not require another data structure outside of its own data collection is described in big O as $O(1)$. Other algorithms require extra structures to solve the problem. The extra structures take up more memory which may be an important factor when choosing an algorithm.

1.1.1 Orders of Growth

An algorithm is classified based on its most expensive computation (Mannion, n.d.), which help to classify which family of order of growth it belongs to. For example, if an algorithm has a nested loop then it is described as falling under an order of growth $O(n^2)$. Calculating the order of growth will help to determine which algorithm will scale well to solve problems of a considerable size. The various Orders of Growth classifications are shown in *Figure 2: Time Complexity of Algorithms* and the complexities are explained in *Table 1: Common Complexities*.

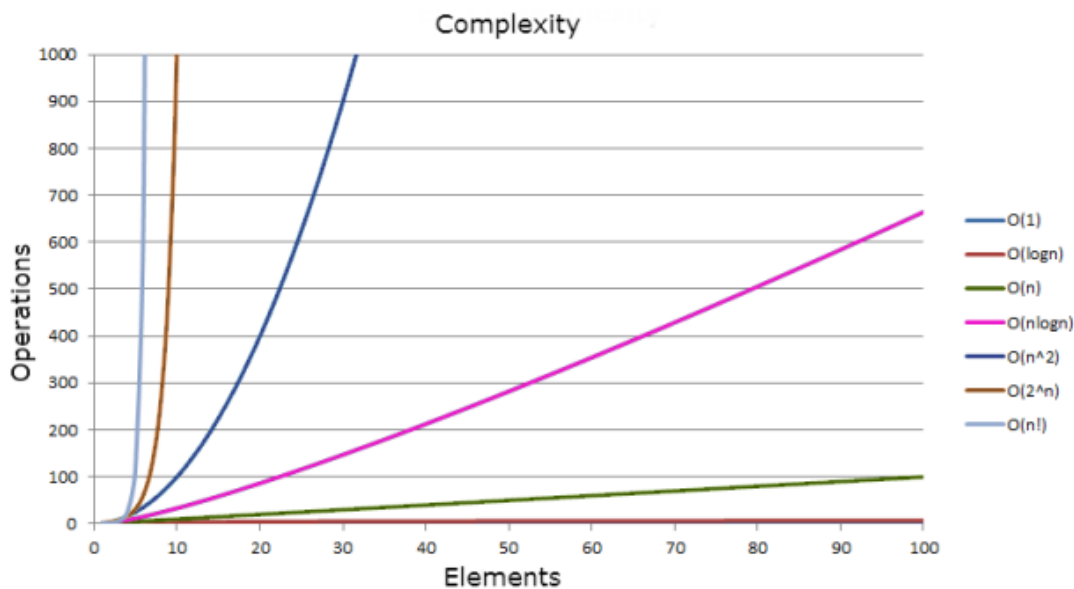


Figure 2: Time Complexity of Algorithms (Newton, 2017)

| Complexity | Explanation | Number Operations $n=8$ | Efficiency |
|-------------|---|-------------------------------|------------|
| $O(1)$ | Constant time – involves only one operation. Sorting an array with only one value or finding the last element in an array are examples. Very fast. $n = 8$ | 1 | Best ↓ |
| $O(\log n)$ | Logarithmic Time. | 3 | |


| | | | |
|---------------|---|-----|---|
| $O(n)$ | Linear time – involves an operation for each element in the collection. | 8 |  Worst |
| $O(n \log n)$ | Linearithmic | 24 | |
| $O(n^2)$ | Quadratic time. For example, the algorithm uses a nested loop. | 64 | |
| $O(2^n)$ | Exponential | 256 | |
| $O(n!)$ | n Factorial. Algorithms are worst in terms of complexity. A little increase to the size of data will have a huge knock on effect. | | |

Table 1: Common Complexities

Figure 3: Summary of Algorithm Efficiencies gives an overview of the best, average and worst time complexities for the 5 sorting algorithms examined in this report. It also shows which algorithms uses in-place sorting and auxiliary space required (if any) plus which are stable.

| Algorithm | Worst | Best | Average | In-place | Stable | Space Complexity |
|-----------|------------|------------|------------|----------|--------|------------------|
| Bubble | n^2 | n | n^2 | Yes | yes | 1 |
| Insertion | n^2 | n | n^2 | Yes | yes | 1 |
| Merge | $n \log n$ | $n \log n$ | $n \log n$ | No | yes | $O(n)$ |
| Counting | $n + k$ | $n + k$ | $n + k$ | No | yes | $n + k$ |
| Timsort | n | $n \log n$ | $n \log n$ | No | yes | n |

Figure 3: Summary of Algorithm Efficiencies

1.2 Performance

Performance is the time and resources the used when a program\sorting application is run on a platform. The performance is affected by complexity but not the other way around. Performance can be measured by experimenting on a targeted platform.

1.3 In-place Sorting

An In-place Sorting algorithm does not require an additional structure such as a temporary array to fulfil the sorting process. It uses the original array only and can be described as ‘using $O(1)$ auxiliary memory’ (Stack Overflow, n.d.). In-place sorting maybe a use case where memory availability is low. Examples of In-place sorting algorithms are: ‘Bubble sort, Selection Sort, Insertion Sort and Heapsort’ (geeksforgeeks, 2019).

1.4 Stable Sorting

A sorting algorithm is stable if equal elements appear in the same order after sorting. *Figure 4: Stable Sort Example* shows an example where the same values are kept in the original order. Examples of a stable sorting algorithm are: ‘Bubble Sort, Insertion Sort, Merge Sort and Count Sort’ (geeksforgeeks, 2019).

| | | | | | | | | | | | | | |
|---|--|---|---|---|----|---|----|---|---|---|---|---|---|
| Unsorted Array | <table><tr><td>9</td><td>2</td><td>5</td><td>1</td><td>2</td><td>11</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table> | 9 | 2 | 5 | 1 | 2 | 11 | 0 | 1 | 2 | 3 | 4 | 5 |
| 9 | 2 | 5 | 1 | 2 | 11 | | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | | | | | | | | |
| Array after sorting algorithm using a stable sort. Element at index 1 was not changed. | <table><tr><td>1</td><td>2</td><td>2</td><td>5</td><td>9</td><td>11</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table> | 1 | 2 | 2 | 5 | 9 | 11 | 0 | 1 | 2 | 3 | 4 | 5 |
| 1 | 2 | 2 | 5 | 9 | 11 | | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | | | | | | | | |

Figure 4: Stable Sort Example

1.5 Inversions

Inversions describe the number of swaps of elements required to sort the data. It is useful to have an idea of the state of data as it may affect the choice of algorithm .

1.6 Comparator Functions

To sort a collection in order of ascending the elements must be less than or equal to its neighbouring element. To compare elements in an integer of arrays we can using less than (<), greater than (>) or equals(=).

1.7 Comparison-based\Non-Comparison-based Sorts

Sorting algorithms fall under one of two main categories which are algorithms that use comparison-based methods using a comparator function or ones that use non-comparison-based methods. Counting sort is an example that sorts using a key-value instead of a comparator operator. Bubble sort uses a comparator operator.

1.8 Sort keys and satellite data

Sort key is the current value the algorithm uses to sort data. Satellite data is other data stored in the structure that moves with the element as it is sorted .

2 Sorting Algorithms

Sorting algorithms can be categorized as ‘comparison-based’ (simple/efficient) or ‘non-comparison-based’. A Comparison is when an element is compared against another element using an operator such as ‘less than’ or ‘greater than’. This report will analyse an algorithm for each of these strategies.

2.1 Bubble Sort

Bubble sort (also known as a Sinking sort) works by getting the larger value to ‘bubble’ to the end by using a series of comparisons and swaps. It is a simple comparison-based approach to the problem. *Table 2: Bubble Sort* displays the complexities for this algorithm. The bubble sort uses a nested loop

to solve the problem, this gives it a complexity of best, worst and average complexity of n^2 . Unfortunately for an end-user who uses an application that processed large real-world data with a bubble sort would experience long wait times. On the plus side, the algorithm does not require additional space to solve the problem and it keeps same value in their original order.

Advantage(s): Bubble sort is easy to understand and ok for a small data collection.

Disadvantage(s): Efficiency is poor as the collection size increases.

Table 2: Bubble Sort Features

| Attribute | | Comment |
|------------------|---------------|---|
| Type | Comparison | Simple comparison sort |
| Stable | Yes | |
| Worst Case | $O(n^2)$ | Algorithm has a nested loop |
| Best Case | $\Omega(n)$ | previously sorted list (optimized bubble sort – if no) |
| Average | $\Theta(n^2)$ | Algorithm has a nested loop |
| In-place | Yes | |
| Space Complexity | 1 | original array used for sorting |

2.1.1 Pseudo Code (Non-optimised)

Below is the pseudo code for the bubble sort. This algorithm uses a nested loop which gives it a complexity classification of $O(n^2)$. It is not optimised to consider if the input is pre-sorted.

```

for all elements of array
    for all elements of array minus 1
        if array[i] > array[i+1]
            swap(array[i], array[i+1])
        end if
    end for
end for

```

2.1.2 Bubble Sort Examples

The starting array is randomly populated with these 5 values: {1,7,2,11,5}. *Figure 5: Bubble Sort Random {1,7,2,11,5}* shows that 10 operations were needed to complete the sort.

| | 0 | 1 | 2 | 3 |
|---|--|---|---|---|
| 0 | Input: [1, 7 , 2, 11, 5] Output: [1, 7, 2, 11, 5] | Input: [1, 2 , 7, 5, 11] Output: [1, 2, 7, 5, 11] | Input: [1, 2 , 5, 7, 11] Output: [1, 2, 5, 7, 11] | Input: [1, 2 , 5, 7, 11] Output: [1, 2, 5, 7, 11] |
| 1 | Input: [1, 7 , 2, 11, 5] Output: [1, 2, 7, 11, 5] 7↔2 | Input: [1, 2 , 7, 5, 11] Output: [1, 2, 7, 5, 11] | Input: [1, 2 , 5, 7, 11] Output: [1, 2, 5, 7, 11] | |
| 2 | Input: [1, 2, 7 , 11, 5] Output: [1, 2, 7, 11, 5] | Input: [1, 2, 7 , 5, 11] Output: [1, 2, 5, 7, 11] 7↔5 | | |
| 3 | Input: [1, 2, 7, 11 , 5] Output: [1, 2, 7, 5, 11] 11↔5 | | | |

Key
 Ignored
 Compared
 Swapped

Figure 5: Bubble Sort Random {1,7,2,11,5}

The example in Figure 6: Bubble Sort - Pre-sorted Array {1,2, 5, 7, 11} shows the steps needed to sort through a pre-ordered array as the pseudo code is not optimised.

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | Input: [1, 2 , 5, 7, 11] Output: [1, 2, 5, 7, 11] | Input: [1, 2 , 5, 7, 11] Output: [1, 2, 5, 7, 11] | Input: [1, 2, 5, 7, 11] Output: [1, 2, 5, 7, 11] | Input: [1, 2, 5, 7, 11] Output: [1, 2, 5, 7, 11] |
| 1 | Input: [1, 2 , 5, 7, 11] Output: [1, 2, 5, 7, 11] | Input: [1, 2 , 5, 7, 11] Output: [1, 2, 5, 7, 11] | Input: [1, 2, 5, 7, 11] Output: [1, 2, 5, 7, 11] | |
| 2 | Input: [1, 2, 5 , 7, 11] Output: [1, 2, 5, 7, 11] | Input: [1, 2, 5, 7, 11] Output: [1, 2, 5, 7, 11] | | |
| 3 | Input: [1, 2, 5, 7 , 11] Output: [1, 2, 5, 7, 11] | | | |

Key
 Ignored
 Compared
 Swapped

Figure 6: Bubble Sort - Pre-sorted Array {1,2, 5, 7, 11}

The example in Figure 7: Bubble Sort Reversed Array {11,7,5,2,1} shows the steps and swaps necessary for an array that is reversed.

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | Input: [11 , 7, 5, 2, 1] Output: [7, 11, 5, 2, 1] 11↔7 | Input: [7, 5 , 2, 1, 11] Output: [5, 7, 2, 1, 11] 7↔5 | Input: [5, 2 , 1, 7, 11] Output: [2, 5, 1, 7, 11] 5↔2 | Input: [2, 1 , 5, 7, 11] Output: [1, 2, 5, 7, 11] 2↔1 |
| 1 | Input: [7, 11 , 5, 2, 1] Output: [7, 5, 11, 2, 1] 11↔5 | Input: [5, 7 , 2, 1, 11] Output: [5, 2, 7, 1, 11] 7↔2 | Input: [2, 5 , 1, 7, 11] Output: [2, 1, 5, 7, 11] 5↔1 | |
| 2 | Input: [7, 5, 11 , 2, 1] Output: [7, 5, 2, 11, 1] 11↔2 | Input: [5, 2, 7 , 1, 11] Output: [5, 2, 1, 7, 11] 7↔1 | | |
| 3 | Input: [7, 5, 2, 11 , 1] Output: [7, 5, 2, 1, 11] 11↔1 | | | |

Key
 Ignored
 Compared
 Swapped

Figure 7: Bubble Sort Reversed Array {11,7,5,2,1}

2.1.3 Pseudo Code Optimised

```

for all elements of array
    flag = false
    for all elements of array minus 1
        if array[i] > array[i+1]
            swap(array[i], array[i+1])
            flag = true
        end if
    end for
    if flag equals false
        end program
end for

```

Example in *Figure 8: Optimised Bubble Sort Example* shows that by adding an optimization to the algorithm such as checking for a pre-sorted collection it can reduce the time complexity. For a non-trivial size collection this could save a lot of time.

| | |
|----------|---|
| 0 | Input: 1, 2 , 5, 7, 11] |
| | Output: [1, 2, 5, 7, 11] |
| 1 | Input: [1, 2, 5 , 7, 11] |
| | Output: [1, 2, 5, 7, 11] |
| 2 | Input:[1, 2, 5 , 7 , 11] |
| | Output: [1, 2, 5, 7, 11] |
| 3 | Input: [1, 2, 5, 7 , 11] |
| | Output: [1, 2, 5, 7, 11] |

Key

Ignored

Compared

Swapped

Figure 8: Optimised Bubble Sort Example

2.2 Insertion Sort

Insertion sort is a simple comparison sort. It uses nested loops to check if the key (current element being sorted) is less than the elements before it. The algorithm will then slot the key into the correct sorted position. This algorithm would not be suitable for applications that process large real-world data as the end-user would experience long wait times. The algorithm uses a nested loop that gives a worst and average case complexity of n^2 . The efficiency of insertion sort reduces as the data size grows. On the plus side, it is fast for sorting small data collections (<65). It does not require additional storage to solve the problem and it keeps same values in their original order. *Table 3: Insertion Sort Features* gives an overview of Insertion sort complexities.

Advantage(s): Simple code. It is optimised to cut down operations for pre-sorted\nearly sorted arrays and is more efficient than the bubble sort.

Disadvantage(s): Uses nested loops $O(n^2)$, so when large volumes of data are involved, it will be slower than other algorithms.

| Attribute | | Comment |
|------------------|---------------|---------------------------------|
| Type | Comparison | Simple comparison sort |
| Stable | Yes | |
| Worst Case | $O(n^2)$ | Algorithm has a nested loop |
| Best Case | $\Omega(n)$ | previously sorted list |
| Average | $\Theta(n^2)$ | Algorithm has a nested loop |
| In-place | Yes | |
| Space Complexity | 1 | original array used for sorting |

Table 3: Insertion Sort Features

2.2.1 Pseudo Code

```

for each element in the array (i = 2nd element)
    Key = array[i]
    j = i - 1
    while (array[j - 1] > key && j > 0)
        array[j + 1] = arr[j];
    End While Loop
    array[j+1] = key
End For Loop

```

2.2.2 Insertion Sort Examples

The insertion algorithm uses seven operations in the example shown in *Figure 9: Insertion Sort - Random Array*.

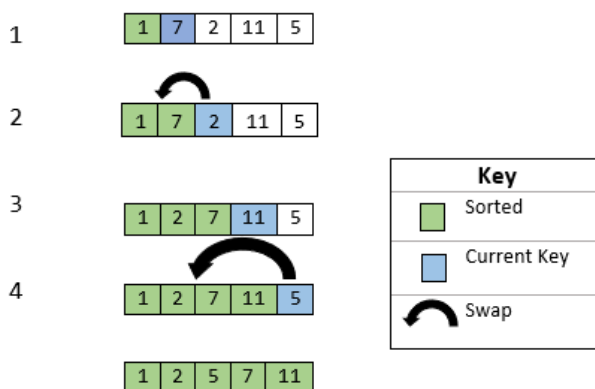


Figure 9: Insertion Sort - Random Array

Four operations are required in a pre-sorted array see example, *Figure 10: Insertion Sort – Previously sorted Array*.

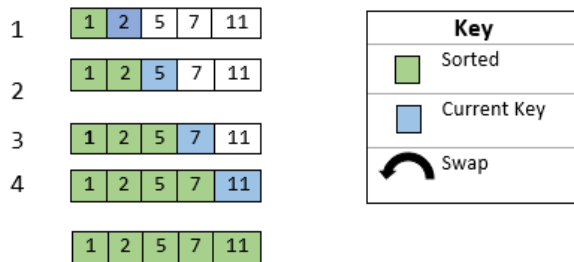


Figure 10: Insertion Sort – Previously sorted Array

2.3 Merge Sort

The merge sort algorithm operates by recursively splitting the input into two sub arrays (left and right) until each contains one element; merge the subarrays in sorted order back to the array. Merge sort was developed in 1945 by John Von Neumann. Compared to bubble sort and insertion sort is more efficient as sorting large data collections as its complexity for best, worst and average cases is 'n log n'. Instead of nested loops this algorithm uses single loops and recursion making it less complex. On the downside, it does require extra space to solve the problem as it uses supplementary arrays. See *Table 4: Merge Sort Features* for an overview of its complexities and features.

Advantage(s): Fast at sorting large collections of data with a consistent run time.

Disadvantage(s): Uses more space than some other algorithms and slower at sorting small data collection than others.

Table 4: Merge Sort Features

| Attribute | | Comment |
|------------------|--------------------------|--|
| Type | Comparison | Efficient comparison sort |
| Stable | Yes | |
| Worst Case | $O(n \cdot \log n)$ | Merge sort always divides the array in two halves and takes linear time to merge two halves (studytonight.com, n.d.) |
| Best Case | $\Omega(n \cdot \log n)$ | |
| Average | $\Theta(n \cdot \log n)$ | |
| In-place | No | Auxiliary arrays are used. |
| Space Complexity | $O(n)$ | Uses auxiliary arrays to solve the problem |

2.3.1 Pseudo Code

```

Function sort(array[], start index l, end index r):
while (there are values)
    Find the middle point of array[] m.
    Call sort(array, l, m) for first half.
    Call sort(array, m+1, r) for second half.
    Call merge(array, l, m, r):
        Create temp left and right arrays from array[].
        Copy values back to array[] in sorted order.
        Copy any other remaining values.

```

2.3.2 Merge Sort Examples

The examples shown below in *Figure 11: Merge Sort Random Order Array* show the steps for a random ordered array {1, 7, 2, 11, 5}.

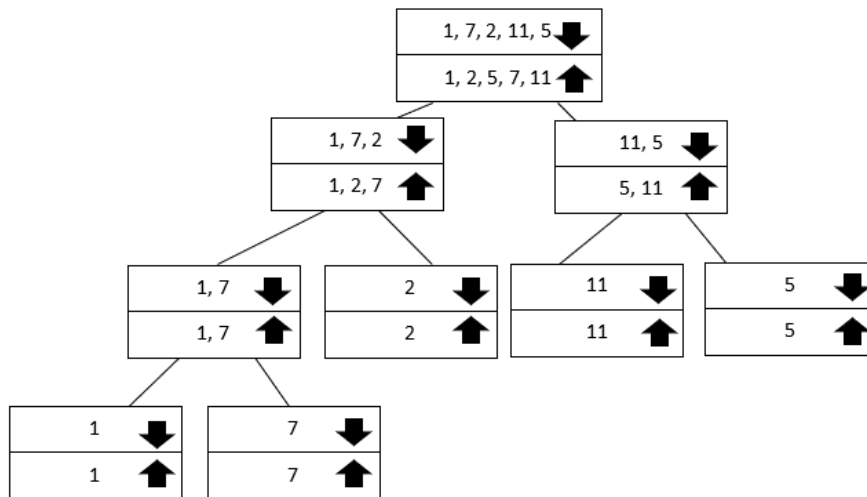


Figure 11: Merge Sort Random Order Array

Figure 12: Merge Sort Reversed Array show the steps to sort a completely reversed array {11, 7, 5, 2, 1}.

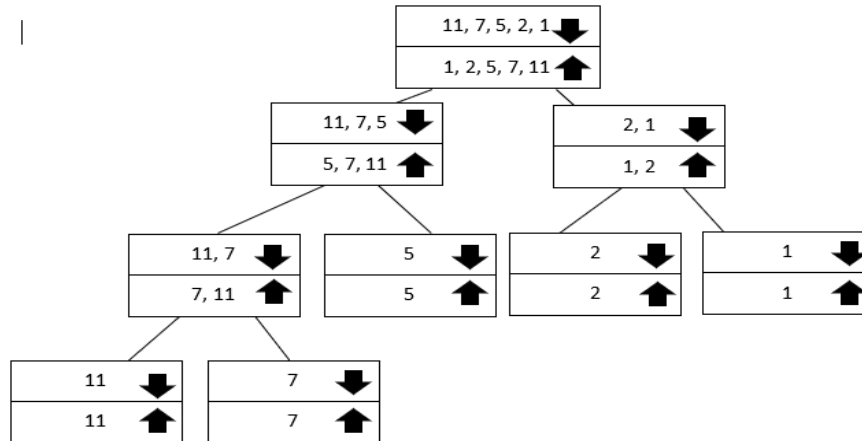


Figure 12: Merge Sort Reversed Array

2.4 Counting Sort

Counting sort is a non-comparison-based algorithm created in 1954 by Harold H. Seward. It enumerates occurrences of stored values and performs arithmetic to calculate the position of each element in the output array\collection. The counting sort algorithm complexity is 'n+k' for best, worst and average cases. It solves the problem very quickly in use cases where discrete data is used, and the data value range allows the indexes required to be kept to a minimum. On top of having limited use cases this algorithm also requires additional storage to solve the problem. *Table 5: Counting Sort Features* gives an overview of this algorithm's complexities and features.

Advantage(s): Very fast in suitable use cases.

Disadvantage(s): Major disadvantages are its limitation of use. It is described in Wikipedia as 'only suitable for direct use in situations where the variation in keys is not significantly greater than the number of items' (Wikipedia, 2021). Secondly, It can be 'used only to sort discrete values (for example integers), because otherwise the array of frequencies cannot be constructed' (Programming-Algorithms.net, n.d.).

Table 5: Counting Sort Features

| Attribute | | Comment |
|------------|------------------|---|
| Type | Non - Comparison | Efficient non-comparison sort |
| Stable | Yes | Equal elements appear in the same order after sort. |
| Worst Case | $O(n + k)$ | n is the size k represents the range of values |
| Best Case | $\Omega(n + k)$ | |
| Average | $\Theta(n + k)$ | |

| | | |
|-------------------------|------------|--|
| In-place | No | Auxiliary arrays are used. |
| Space Complexity | $O(n + k)$ | Uses auxiliary arrays to solve the problem |

2.4.1 Pseudo Code

```

min = calculate min
max = calculate max

//store count of each value
For each element in array
    count[arr[i] - min]++
end for

//shift index positions one place to the right
for each element in the array
    count[i] += count[i - 1]
end for

//place values in output array in the index
// specified by count array
//update count value
for each element working from end of array
    output[count[arr[i] - min] - 1] = arr[i];
    count[arr[i] - min]--;
}

//copy new sorted array back to the original array
for each element in array
    arr[i] = output[i];
end for

```

2.4.2 Counting Sort Examples

Figure 13: Counting Sort Random array shows the step needed to sort a random array: $A\{1, 7, 2, 11, 5\}$.

| Unsorted array : A | | | |
|---------------------------|--|--|---|
| | | | 1 7 2 11 5 |
| 1 | <div> <div>1 2 3 4 5 6 7 8 9 10 11</div> <div>W 1 1 0 0 1 0 1 0 0 0 1</div> </div> | Create working array W with index values spanning range of A. Populate W with no. of occurrences of each number in range. | |
| 2 | <div> <div>1 2 3 4 5 6 7 8 9 10 11</div> <div>W 1 2 2 2 3 3 4 4 4 4 5</div> </div> | Sum each value into its neighbour in W | |
| 3 | <div> <div>1 2 3 4 5 6 7 8 9 10 11</div> <div>W 0 1 2 2 2 3 3 4 4 4 4</div> </div> | Shift values in W to right by 1. 0 is inserted in the first element. | |
| 4 | <div> <div> <div>1 2 3 4 5 6 7 8 9 10 11</div> <div>W 1 2 2 2 3 3 4 4 4 4 5</div> </div> <div> <div>1 7 2 11 5</div> <div>A</div> </div> <div> <div>1 2 5 7 11</div> <div>B</div> <div>0 1 2 3 4</div> </div> </div> | For each element in A (original unsorted array): <ol style="list-style-type: none"> find the index in W that matches its element value – the value at the position in W represents its new location in array B Once value is inserted in B then increment the value in array W by 1. | |

Figure 13: Counting Sort Random array

Figure 14: Counting Sort Duplicate Values show the steps required to sort an array that contains same values: A {3, 5, 3, 1, 5}.

| Unsorted array : A | | | |
|---------------------------|--|--|--|
| | | | 3 5 3 1 5 |
| 1 | <div> <div>1 2 3 4 5</div> <div>W 1 0 2 0 2</div> </div> | Create working array W with index values spanning range of A. Populate W with no. of occurrences of each number in range. | |
| 2 | <div> <div>1 2 3 4 5</div> <div>W 1 1 3 3 5</div> </div> | Sum each value into its neighbour in W | |
| 3 | <div> <div>1 2 3 4 5</div> <div>W 0 1 1 3 3</div> </div> | Shift values in W to right by 1. 0 is inserted in the first element. | |
| 4 | <div> <div> <div>1 2 3 4 5</div> <div>W 1 1 3 3 5</div> </div> <div> <div>3 5 3 1 5</div> <div>A</div> </div> <div> <div>1 3 3 5 5</div> <div>B</div> <div>0 1 2 3 4</div> </div> </div> | For each element in A (original unsorted array): <ol style="list-style-type: none"> find the index in W that matches its element value – the value at the position in W represents its new location in array B Once value is inserted in B then increment the value in array W by 1. | |

Figure 14: Counting Sort Duplicate Values

2.5 Tim Sort

Timsort was created by Tim Peters in 2002. It is a hybrid algorithm as it uses both components of insertion sort and merge sort. It is used for sorting in Python, Java, Android platform and GNU Octave (AWDESH, 2018). Tim Sort uses the 'Runs', 'Min Run' and 'Galloping' to cut down on comparisons and be more efficient.

Runs

Tim Sorts exploits the concept of 'Runs' in the array. Usually, in real world data, some of the data will already be in order (ascending or descending). The chunks of sorted data are known as runs. Runs that are in descending order are blindly reversed.

Min Run

Tim Sort finds the minimum length of each run that gives the most efficient time later in the sorting and merging process. Using the 'min run' the algorithm creates runs in ascending or descending order. If the 'min run' is met but the next element follows a pattern, then keep adding elements to the run. Ideally, if $(N \setminus \text{'min run'})$ was a power of 2, it would make the merge process more efficient.

Gallop

During the merging process a 'galloping' technique can be used if the algorithm identifies a series of number in one array which can be taken as a chunk and slotted into the merged array which cuts down on comparisons.

The time complexity of Timsort in the best case is linear time, making it very fast. Its time complexity is better than Merge sort due to it including extra functionality such as 'Gallop' and calculating efficient minimum run sizes. Its efficiency is on par with Merge sort in both worst and average cases 'n log n'. However, due to its extra features and using insertion sort for small data collections it beats Merge sort in terms of efficiency. Timsort does require extra space to solve the problem as auxiliary arrays are used. See, *Table 6: Timsort Complexities & Features* for an overview of its complexities and features.

Advantage(s): An efficient, fast and stable algorithm that is good 'all-rounder' as it is good at sorting both small and large data collections. It uses binary insertion sort which is quick at sorting small data sets. It can use the merge sort which works well with larger data. 'Gallop' speeds it up too.

Disadvantage(s): It is a complex algorithm (difficult to diagram as insertion sort sorts arrays as minimum run sizes are usually set to 32 or 64)

Table 6: Timsort Complexities & Features

| Attribute | | Comment |
|------------|---------------------|---------------------------|
| Type | Comparison | Efficient comparison sort |
| Stable | Yes | |
| Worst Case | $O(n \cdot \log n)$ | |

| | | |
|-------------------------|--------------------|--|
| Best Case | $\Omega(n)$ | Best case if data collection is already sorted it runs on linear time. |
| Average | $\Theta(n \log n)$ | |
| In-place | No | Auxiliary arrays are used. |
| Space Complexity | $O(n)$ | Uses auxiliary arrays to solve the problem |

2.5.1 Pseudo Code

```

If array length <= 64 (or 32)
    minimum_run_size = N
    Call insertion sort and return sorted array

Else {
    Calculate the minimum_run_size
    Divide into Runs {
        If current_run equals min_run_size
            If next element follows the run pattern
                Add element to current_run
    }
    Sort each run using insertion sort
    Descending runs are blindly reversed.
    Merge sorted_runs
    If > 7 elements in series are inserted from one subarray
        trigger a 'Galloping' process
    return sorted array
}

```

2.5.2 Tim Sort Examples

The example of 'Galloping' is shown in *Figure 15: Timsort Galloping Example*. A chunk of ordered numbers can be inserted into the merged array from one array in one go without the need for comparisons.

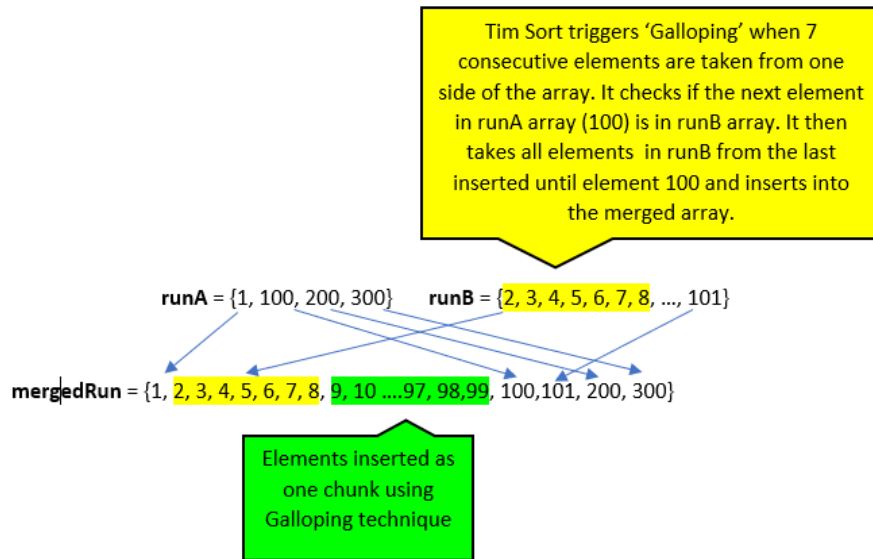


Figure 15: Timsort Galloping Example

In the example shown in *Figure 16: Timsort Runs Example*, it shows that elements will be included in a run if it follows the same pattern. It shows that Timsort groups Runs in ascending and descending orders.

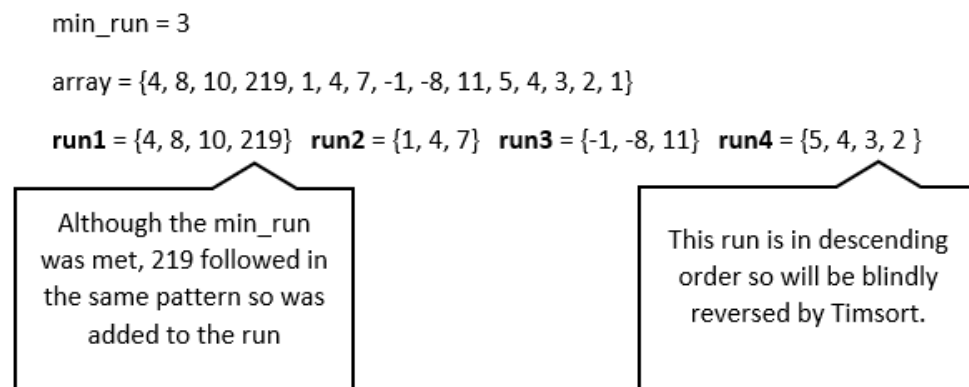


Figure 16: Timsort Runs Example

3 Implementation & Benchmarking

A real-world experiment to benchmark the five algorithms above (Bubble, Insertion, Merge, Counting and Timsort) was performed. The experiment is platform dependent. *Table 7 Platform Information* shows a selection of the platform information this experiment was executed on. As each sorting algorithm was tested using this platform it can give a good overall indication of the algorithm's

performance in comparison with the others. Also, each algorithm was executed ten times for each sample size to give a fairer result.

Table 7 Platform Information

| Component | Version |
|-----------------------------------|------------------------------------|
| Eclipse Execution Environment JRE | JavaSE-15 |
| Application Language | Java |
| Installed RAM | 4GB |
| Operating System | Windows 10 |
| Processor | Intel® Celeron® CPU N3550@ 1.10GHz |

Eclipse was the tool used to create a new Java project which contained two classes called Sorter and Runner. Sorter contains the five sorting algorithms, which are implemented as a mix of public and private functions. The Runner class contains the main() function to start up the application along with other functions to execute the benchmarking.

3.1 Sorter Class

The code for the sorting algorithms has been sourced from the course material and from online resources. The sources have been acknowledged in the code comments in the application. *Table 8: Sorter Class Functions* displays the function signatures associated with each sorting algorithm.

Table 8: Sorter Class Functions

| Algorithm | Functions |
|-----------|---|
| Bubble | <code>public void bubbleSort(int[] arr)</code> |
| Insertion | <code>public void insertionSortArr(int arr[])</code> |
| Counting | <code>public void countingSort(int[] arr)</code> |
| Merge | <code>public void sort(int arr[], int l, int r)</code> <code>public void merge(int arr[], int l, int m, int r)</code> |
| Timsort | <code>public void timSort(int[] arr, int n)</code> <code>private void mergeTS(int[] arr, int l, int m, int r)</code> <code>private void insertionSort(int[] arr, int left, int right)</code> <code>private static int minRunLength(int n)</code> |

3.2 Runner Class

Table 9: Runner Class Declarations and *Table 10: Runner Class Functions* provide a summary of the implementation performed in the Runner class. The application was run several times to check if the results were roughly the same each time (consistent results).

3.2.1 Sample Sizes

A selection of sample sizes ranging from small (1) to large (100,000) were chosen to test how the algorithms handle different sized arrays. The sample size of 1 – is considered a sorted array. This should give an indication of the constant time taken to complete the algorithm without a need to sort the data.

Table 9: Runner Class Declarations

| Properties | Description |
|---|--|
| <code>private static int[] arrSampleSizes = {1, 5, 10, 25, 50, 75, 100, 250, 500, 750, 1000, 2500, 5000, 7500, 10000, 25000, 50000, 75000, 100000 };</code> | These are the sample sizes chosen for the experiment. |
| <code>private static final int runs = 10;</code> | Application will run each algorithm 10 times for each sample. |
| <code>private static String[] titles = {"Bubble Sort", "Insertion Sort", "Merge Sort", "Counting Sort", "TimSort"};</code> | Titles of the algorithms used for the benchmarking – this are used in the results display. |
| <code>private static double[][] sortResults</code> | hold the average of total milliseconds elapsed for each sample size for each algorithm |

Table 10: Runner Class Functions

| Functions | Description |
|--|---|
| <code>public static void main(String[] args):</code> | Repeats for each sample size, calls runSort() for each algorithm, stores the results and then calls printResults(). |

| | |
|---|--|
| <pre>public static double runSort(int cnt, double totalElapsedMillis, Sorter test, int[] arr, String sortType):</pre> | Runs 10 times via self-calls; for each run it calls the appropriate sorting algorithm and times how long it takes to execute in milliseconds. The results are totalled. The average of run time is returned to main(). |
| <pre>static void printResults()</pre> | Displays the results to screen in a table format with headings. |

3.3 Experiment Results

The results from a typical run are shown in *Figure 18: Benchmark Results (1 - 750 samples)* and *Figure 17: Benchmark Results (1000 - 100,000 samples)*. These results were then plotted into a line graph to give a visual representation of the benchmarking results shown in *Figure 19: Sorting Algorithm Benchmark Graph*.

| Size n | 1 | 5 | 10 | 25 | 50 | 75 | 100 | 250 | 500 | 750 |
|----------------|-------|--------|-------|-------|-------|-------|-------|-------|-------|-------|
| Bubble Sort | 0.001 | 0.002 | 0.007 | 0.038 | 0.207 | 0.497 | 3.190 | 0.185 | 0.407 | 2.164 |
| Insertion Sort | 0.001 | 0.001 | 0.004 | 0.012 | 0.055 | 0.119 | 4.271 | 0.032 | 0.252 | 0.219 |
| Merge Sort | 0.001 | 0.008 | 0.092 | 0.145 | 0.023 | 0.789 | 0.032 | 0.062 | 0.114 | 0.179 |
| Counting Sort | 5.205 | 10.992 | 0.831 | 2.086 | 1.022 | 3.097 | 1.895 | 1.500 | 1.055 | 1.137 |
| Timsort | 0.004 | 0.003 | 0.005 | 0.025 | 0.077 | 0.153 | 0.162 | 0.102 | 0.475 | 1.622 |

Figure 18: Benchmark Results (1 - 750 samples)

| Size n | 1000 | 2500 | 5000 | 7500 | 10000 | 25000 | 50000 | 75000 | 100000 |
|----------------|-------|--------|--------|---------|---------|----------|----------|-----------|-----------|
| Bubble Sort | 2.539 | 13.869 | 69.313 | 116.137 | 212.720 | 1483.541 | 5380.320 | 12149.700 | 21647.690 |
| Insertion Sort | 0.717 | 1.221 | 4.769 | 10.937 | 21.736 | 152.443 | 645.963 | 1478.192 | 2643.359 |
| Merge Sort | 0.223 | 0.579 | 1.037 | 1.400 | 2.024 | 5.057 | 10.635 | 17.127 | 23.634 |
| Counting Sort | 1.344 | 0.603 | 1.583 | 1.347 | 0.864 | 0.955 | 1.726 | 2.408 | 3.457 |
| Timsort | 0.761 | 1.801 | 2.296 | 1.462 | 1.451 | 4.029 | 9.243 | 14.639 | 19.817 |

Figure 17: Benchmark Results (1000 - 100,000 samples)

The benchmark graph shows that the counting sort is initially the poorest performing. The lines in general appear fairly parallel until there is a drastic spike after sample size 1000 by the Bubble sort. Insertion sort runtime dramatically increases from sample size 5000. Both Merge sort and Timsort

steadily increase after the 10000. Counting sort although having the poorest beginning has the best performance with large sample sizes.

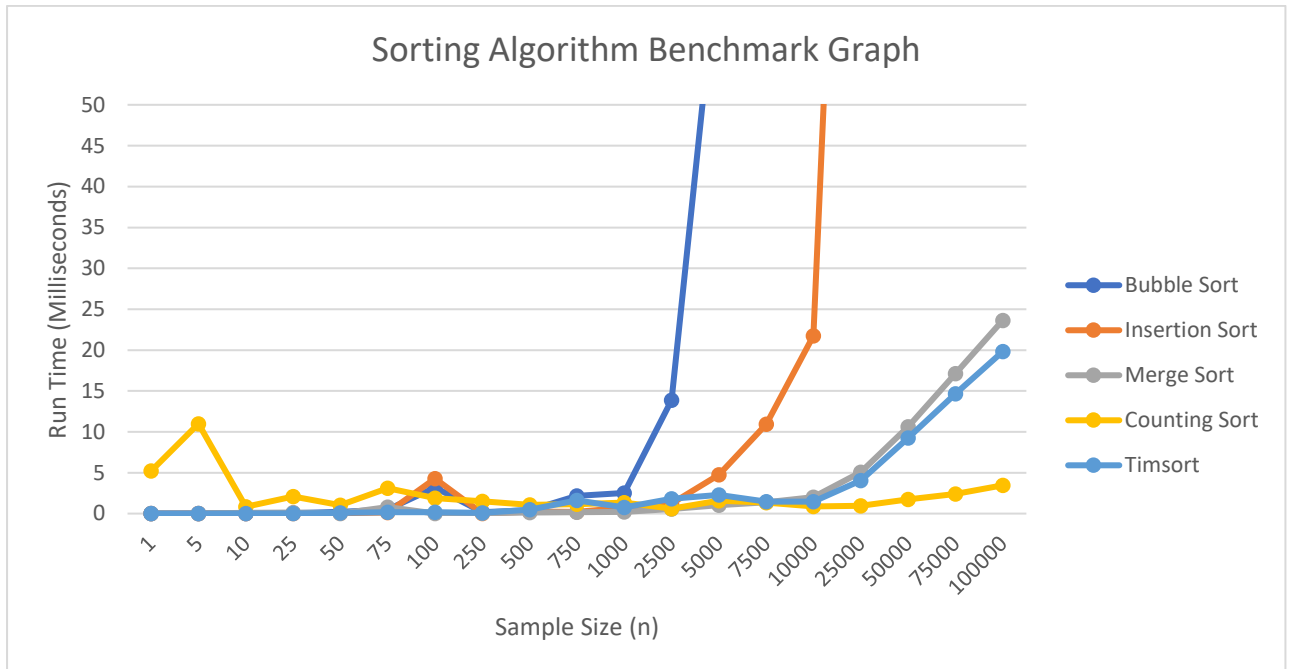


Figure 19: Sorting Algorithm Benchmark Graph

3.4 Analysis

3.4.1 Analysis by Sample Size

Figure 20: Best Worst Per Sample Size displays Best and Worst performers for each sample size based on the results from the experiment shown in Figure 18: Benchmark Results (1 - 750 samples) and Figure 17: Benchmark Results (1000 - 100,000 samples).

Best Performing

- Small sample size (1 - 50): In general, it is the Insertion sort. Insertion sort code requires less operations than the algorithms which benefit it initially.
- Mid sample size (100 - 1000): In general, it is Merge.
- Large sample size (50k, 100k): In general, it is the Counting Sort.

| Size n | Best | Worst |
|--------|--------------------------|-----------|
| 1 | Bubble, Insertion, Merge | Counting |
| 5 | Insertion | Counting |
| 10 | Insertion | Counting |
| 25 | Insertion | Counting |
| 50 | Merge | Counting |
| 75 | Insertion | Counting |
| 100 | Merge | Insertion |
| 250 | Insertion | Counting |
| 500 | Merge | Counting |
| 750 | Merge | Bubble |
| 1000 | Merge | Bubble |
| 2500 | Merge | Bubble |
| 5000 | Merge | Bubble |
| 7500 | Merge | Bubble |
| 10000 | Counting | Bubble |
| 25000 | Counting | Bubble |
| 50000 | Counting | Bubble |
| 75000 | Counting | Bubble |
| 100000 | Counting | Bubble |

Figure 20: Best Worst Per Sample Size

Poorest Performing

- Small sample size (1 - 50): Counting sort. Range spread compared to data size makes this algorithm slow for smaller sample sizes. Requires the most operations where $n = 1$.
- Mid sample size (100 - 1000): Mix of counting, bubble, Timsort & Insertion.
- Large sample size (50k, 100k): Bubble sort is the poorest performer followed by Insertion sort. The 3 other algorithms are much faster in comparison.

3.4.2 Analysis by Average Runtime All Samples

The mean runtime was calculated for each sorting algorithm for all sample sizes, shown in *Figure 21: Mean Runtimes*. Counting has the best average time. Timsort algorithm is shown to have a slightly better performance overall than the merge sort. It shows that there is a big difference between the two poorest performing algorithms Insertion and Bubble.

| Algorithm | Mean |
|----------------|----------|
| Bubble Sort | 2162.238 |
| Insertion Sort | 261.279 |
| Merge Sort | 3.324 |
| Counting Sort | 2.269 |
| Timsort | 3.059 |

Figure 21: Mean Runtimes

3.5 Evaluation

Timsort has proved to be the best general all-rounder. It has good performance for both large and small data collections. It slightly outperforms the Merge sort. Counting sort had the worst performance with smaller data sets and has limited use cases as discussed in Counting Sort. Bubble sort and Insertion sort are ok for small data sets and teaching the sorting concepts. However, for real world applications with large data collections they should be avoided.

4 Conclusion

Understanding how an algorithm works; its efficiency and performance can guide the programmer to choose the right one for their use case(s). As shown above there are vast differences between algorithms in terms of efficiency especially when large volumes of data are concerned. A tweak to slightly optimise an algorithm can have a large knock on effect (when large volumes of data are involved). Users want immediate results and can be easily frustrated with any waiting periods no matter how small. Having the best algorithm both in terms of efficiency and performance may result in one business winning out compared to another.

5 Bibliography

- AWDESH, 2018. *Timsort: Fastest sorting algorithm for real world problems*. [Online] Available at: <https://awdesh.medium.com/timsort-fastest-sorting-algorithm-for-real-world-problems-1d194f36170e> [Accessed 08 05 2021].
- geeksforgeeks, 2019. *Stability in sorting algorithms*. [Online] Available at: <https://www.geeksforgeeks.org/stability-in-sorting-algorithms/> [Accessed 13 04 2021].
- GreenTeaPress, n.d. *Chapter 3 Analysis of algorithms*. [Online] Available at: <https://greenteapress.com/complexity/html/thinkcomplexity004.html#:~:text=An%20order%20of%20growth%20is,set%20grows%20linearly%20with%20n.> [Accessed 18 04 2021].
- Mannion, P., n.d. s.l.:s.n.
- Mannion, P., n.d. *Search Algorithms*, Galway: GMIT.
- Newton, D., 2017. *Learning Big O Notation With $O(n)$ Complexity*. [Online] Available at: <https://dzone.com/articles/learning-big-o-notation-with-on-complexity> [Accessed 13 04 2021].
- Programming-Algorithms.net, n.d. *Counting Sort*. [Online] Available at: <http://www.programming-algorithms.net/article/40549/Counting-sort> [Accessed 18 04 2021].
- Stack Overflow, n.d. *Algorithms; Notes for Professionals*. [Online] Available at: <https://goalkicker.com/AlgorithmsBook/> [Accessed 13 04 2021].
- studytonight.com, n.d. *Merge Sort Algorithm*. [Online] Available at: <https://www.studytonight.com/data-structures/merge-sort> [Accessed 15 04 2021].
- Wikipedia, 2021. *Counting Sort*. [Online] Available at: https://en.wikipedia.org/wiki/Counting_sort [Accessed 18 04 2021].

www.imgflip.com, n.d. *Skeleton Waiting Computer*, s.l.: s.n.