Computational Thinking with Algorithms

Higher Diploma in Data Analytics
46887

Sorting Algorithms

Shane Austin
G00318488

April 2021

Contents

Introduction

Sorting

Sorting Algorithms in Computer Science is the design and implementation of reorganizing data into a specific order for later analysis or processing. Many everyday tasks and computations are significantly more efficient and easier to process if the data is organised in a specific way beneficial to the user. An everyday example would be a person having to look up a name in the phone book, if the entries were unordered, it could take the user days to find the specific person checking each entry one by one. If, however, as is the practice, the phonebook was organised alphabetically by surname, a lookup search could be performed in matter of seconds/minutes.

As data processing is a core procedure throughout every layer of a computer system, from the digital logic level all the way through to the user level, structured data systems are a key element of efficient processing. It is believed that up to 25% of all CPU cycles are spent sorting [1], which in turn makes the remaining 75% of processes faster and more efficient. This means that the development of sorting algorithms was a major component of early computer science and is still being researched and developed today.

History

The mathematician John von Neumann, who, in 1945 developed the computer architecture that is still used in modern computers [2], around the same time, he also proposed one of the earliest sorting algorithms “Merge Sort”. Development of this field has since become a major research area in computing with many commonly used variants of sorting algorithms being developed throughout the 1950s and 60s with researchers like Harold H. Seward developer of “Radix” and “Counting Sorts”, C.A.R. Hoare who proposed “Quicksort”, through to more modern times with Tim Peters’ 2002 “Timsort”, and 2014 development of “Cubesort” with many others in between.

Complexity

Sorting algorithms provide the same function, that is, to return an organised set of data from a given input. The process each algorithm implements to perform this task varies from algorithm to algorithm, and so factors such as datatype, current state of the data, data size, hardware capabilities will all play a part in judging the efficiency of an algorithm based on the given operation.

The two major considerations on assessing the efficiency of an algorithm; are the time it takes, and the amount of computational resources it consumes. This is referred to as time and space complexity (space referring to memory), the current standard for assessing this is called “Big O notation”. Keeping in mind all the external factors influencing sort performance, Big O notation assesses performance on how the runtime of a process grows relative to the size of the input [3]. Big O logs the time taken for a process against the size of the input, and classifies the algorithm based on the performance. These complexities are described below:

O(1)	Constant	This is the best case where runtime is constant regardless of input size
O(n)	Linear	Here the time grows with the size of the input
O(log n)	Logarithmic	In this case the time grows linearly where the input grows exponentially
O(n log n)	Linearithmic	Combination of Linear and Logarithmic
$O(n^2)$	Quadratic	The time is proportional to the square of n
$O(n^3)$	Cubic	The time is proportional to the cube of n
$O(2^n)$	Exponential	The time grows exponentially against the input size

Table. 1

These curves are shown in the chart below, where it also shows the efficiency for each class based on performance.

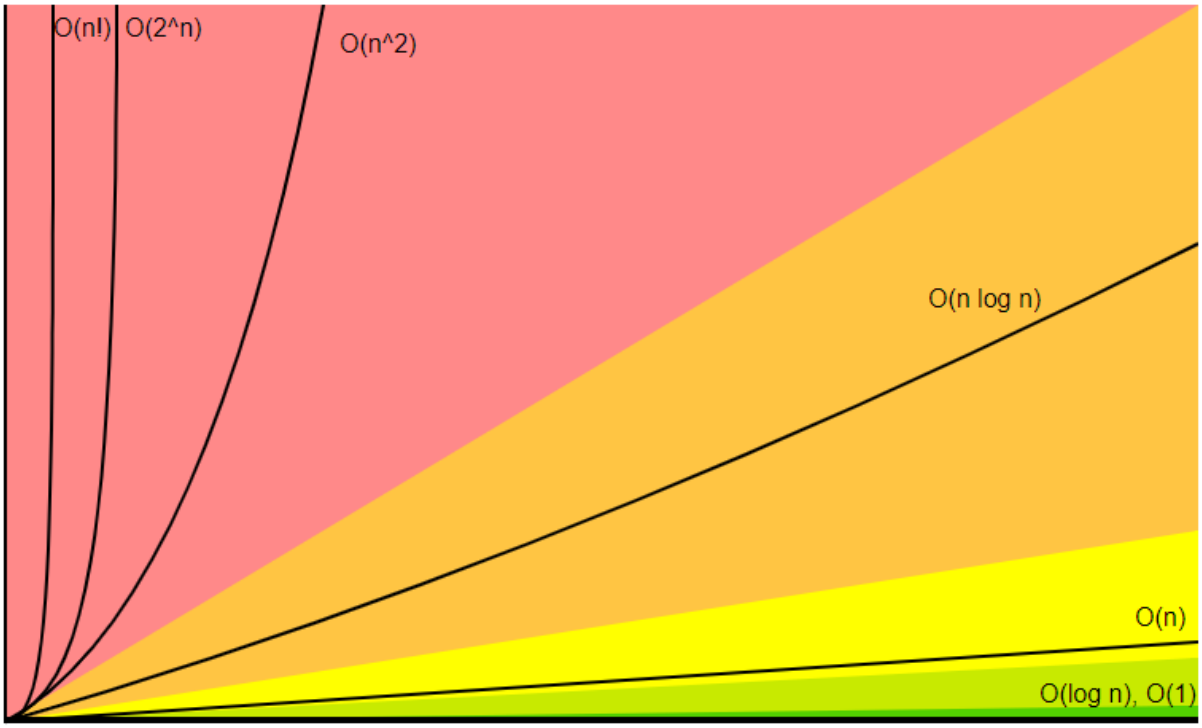

Figure 1 Big-O Complexity Chart [4]

Best, Worst and Average

A development from this is to consider the best, worst and average case of a sort. When developing processes that require the implementation of a sorting algorithm the worst case must be specially considered as to allow for contingency for the longest runtime that may occur.

- Best Case: This is the situation where an algorithm is performing under the optimal environment catered to its strengths.
- Average Case: This is the average runtime of an algorithm based over varied and diverse sets of data. It gives a good basis on general time and memory requirements of running a particular algorithm in most cases. However, designing process based solely on this analysis could prove problematic as is does not properly account for outliers which could significantly slowdown or throttle a process.

- Worst Case: Taking on board the potential issues related to outliers when developing a system based on these processes, it is best practice consider time and space dependencies allowing for potential worst performance situations.

Sort Types

Sorting algorithms can be split into 2 categories based on how they function. Comparison sorts and Non-Comparison Sorts.

Comparison Sorts

Comparison sorts work by systematically comparing two elements in given arrays and assigning an index based on the given parameters, usually an integer size comparison such as; if element a > than element b. With comparison-based algorithms sorting is only performed using the defined comparison definitions. As the definitions are set in the implementation, they can be used in diverse data sets. Due to the iterative process of simple comparison sorts they cannot perform better than O(n log n).

Some examples of simple comparison algorithms are Bubble sort, Selection sort and Insertion sort.

Efficient Comparison Sort algorithms operate in the same manner by comparing two elements, however they are structured in a less iterative way. An example of this would be utilising “Divide and Conquer” techniques to split the data, perform comparisons and merge the data back. Examples of efficient comparison sort algorithms are Merge sort, Quick sort, and Heap sort.

Non-Comparison Sort

Non comparison algorithms do not compare individual elements with each other, they utilise other known information about the data, and so can only be applied in certain cases. Examples of non-comparison sorts include Counting sort, Bucket sort and Radix sort.

Hybrid Sorts

Hybrid sorts can be based on either comparison or non-comparison algorithms and they make use of the advantages of two or more separate algorithms to perform more efficiently. Examples of a comparison-based Hybrid sort is Tim sort that applies both Insertion and Merge sort within it implementation. Introsort is a hybrid non comparison sort based on quick sort and heap sort.

In Place Sorting

In-place sorting is when an algorithm requires very little extra memory resources to perform a sort, and so has an efficient, constant, space complexity growth. Examples of in-place algorithms are Bubble, Selection, Insertion, Quick and Heap sorts

Stability

Stability is also to be considered when sorting, a sorting algorithm is stable if it does not change the order of elements with the same value.[5] An example of this would be with a deck of cards, 5 Clubs, 5 Spades and 5 Hearts all have the same value, but it could be important to keep them in the original order when sorted to a new array of 5,5,5. A stable algorithm ensures this data integrity is retained.

Figure 2 shows the runtime and space complexity of ten of the more common sorting algorithms, including weather they are stable or not.

Algorithm	Best case	Worst case	Average case	Space Complexity	Stable?
Bubble Sort	n	n^2	n^2	1	Yes
Selection Sort	n^2	n^2	n^2	1	No
Insertion Sort	n	n^2	n^2	1	Yes
Merge Sort	$n\ log\ n$	$n\ log\ n$	$n\ log\ n$	$O(n)$	Yes
Quicksort	$n\ log\ n$	n^2	$n\ log\ n$	n (worst case)	No*
Heapsort	$n\ log\ n$	$n\ log\ n$	$n\ log\ n$	1	No
Counting Sort	$n + k$	$n + k$	$n + k$	$n + k$	Yes
Bucket Sort	$n + k$	n^2	$n + k$	$n \times k$	Yes
Timsort	n	$n\ log\ n$	$n\ log\ n$	n	Yes
Introsort	$n\ log\ n$	$n\ log\ n$	$n\ log\ n$	$log\ n$	No

Figure 2 Overview of Sorting algorithms complexity [6]

Sorting Algorithms

Based on the assessment requirements and from the analysis above the following sorting algorithms will be discussed and tested:

- Bubble Sort – A good example of a simple comparison-based sort
- Merge Sort – An efficient comparison-based sort
- Counting Sort – A non-comparison sorting algorithm
- Tim Sort – A hybrid sorting algorithm based on Insertion Sort and Merge Sort
- Insertion Sort – This is a simple comparison-based sort but will be assessed as it forms the basis of Tim Sort and so is worth further investigation.

Bubble Sort

The term "Bubble Sort" has been accredited to computer scientist Kenneth Iverson, and was first introduced in 1962, previously it had been referred to as "Exchange Sorting" where it is believed to have been analysed as early as 1956. It is so named because the process involves exchanging positions of items until highest value is placed at the end of an array and so the larger numbers "bubble-up" towards the end of the sequence.

Bubble Sort is a simple comparison-based algorithm, it works by comparing one element in an array with the adjacent element and if the original element is larger than the adjacent element the two will be swapped, and if not, they stay in-place. It then compares the higher of these elements with the new adjacent element and performs the same swap process. This continues up until the last element in the array where it will place the highest number at the end of the array having compared every adjacent element along the way. This is the first pass complete, and so starts the process from the beginning with the second pass where it exchanges adjacent elements up until the second last element in the array. It continues this process up until only the first and second index elements are left, it compares and exchanges these and the list would now be sorted.

Bubble sort performance analysis is shown below:

- **Best Case** – As Bubble Sort performs well on nearly sorted arrays its best-case complexity is O(n) as discussed above represents linear growth
- **Average Case** – has a complexity of $O(n^2)$ quadratic growth
- **Worst Case** - has a complexity of $O(n^2)$ quadratic growth
- **Space Complexity** – as an in-place algorithm it has a constant growth of O (1)
- **Stable?** – Bubble sort maintains the original order of similar value elements and so is considered stable.

The following diagrams show the Bubble sort in action with small sets of theoretical data. Some of the arrays have been manipulated to try to represent best, average, and worst cases.

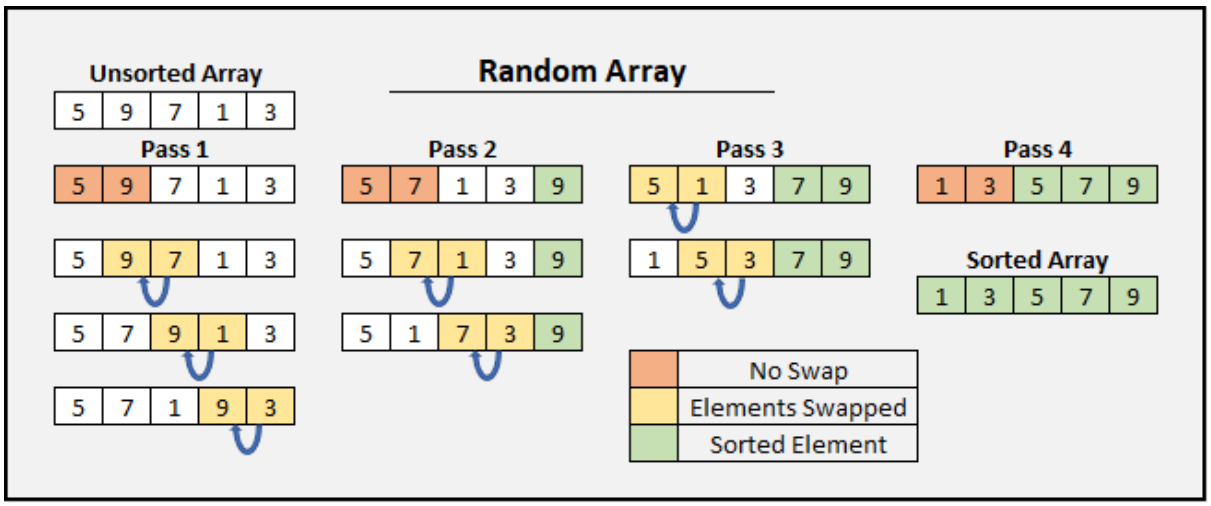

Figure 3 Bubble Sort, Random Array, size 5

Figure 3 shows the basic steps Bubble sort takes to sort a short array of random integers. In the first pass checks are made against the first 2 elements in the array, as they are in the correct order already no swap is made and it moves on to the next two elements, here 9 is larger than 7 so they are swapped, and it moves on to the next element which Is the newly swapped 9 and its adjacent element. It continues this process until the end of the array is reached and hence the last element in the array is the highest value, in this case 9. The process is restarted from the first index up until the second last element and finishes in a state where the last two elements in the array are in the desired order. In this case it repeats this process for 4 passes until the array is sorted. In an array of length 5 the process involves 10 comparisons and 8 swaps to sort the data.

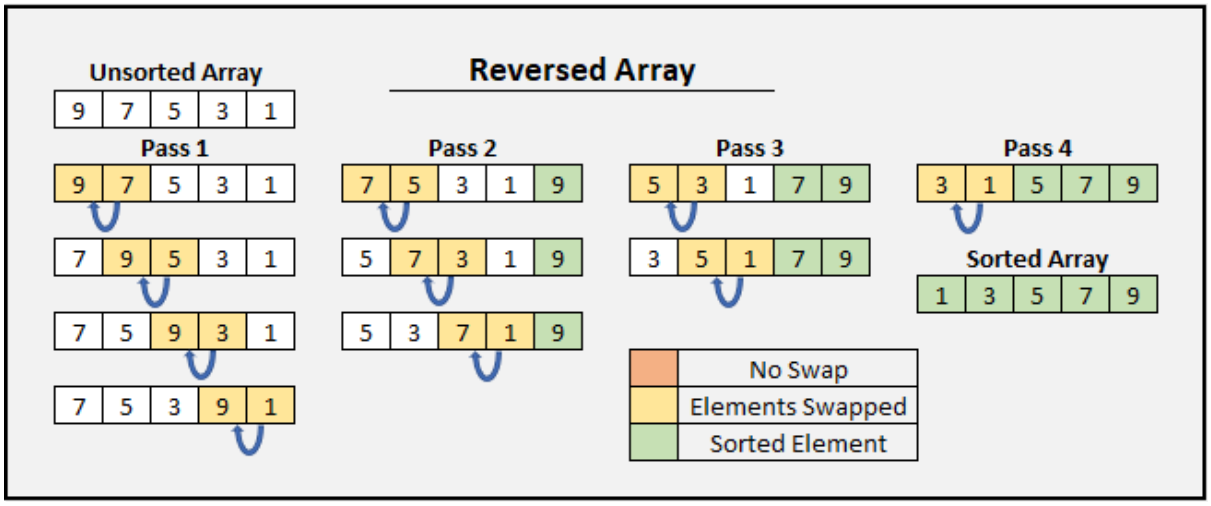

Figure 4 Bubble Sort Reversed Array, size 5

Figure 4 tries to represent a situation where Bubble sort would be expected to perform poorly, here the array is in reverse order. In this run, the process makes 10 comparisons and performs 10 swaps. It is expected that it would take slightly longer to sort this set compared to the random array due to the extra swaps, however as it performs the same number of comparisons it matches the prediction as Bubble sort runs in $0(n^2)$ time for both the worst and average cases.

The next three figures have expanded the array size to 7 with a higher data range to compare Bubble sort's theoretical performance with a random array, reversed array, and a nearly sorted array.

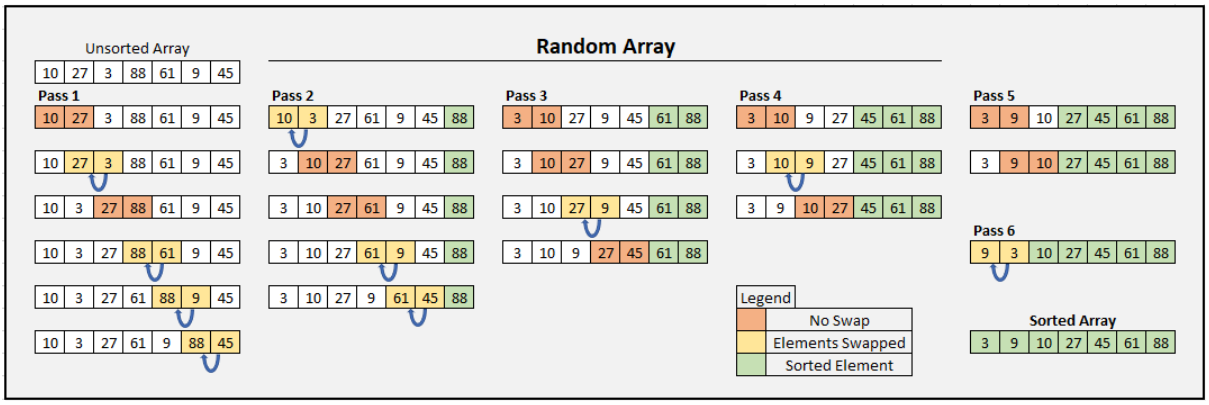

Figure 5 Bubble Sort Random Array, size 7

To sort this data, 6 passes are required with 21 comparisons and 10 swaps, compared the previous example the array has grown by two elements and the number of comparisons required has doubled.

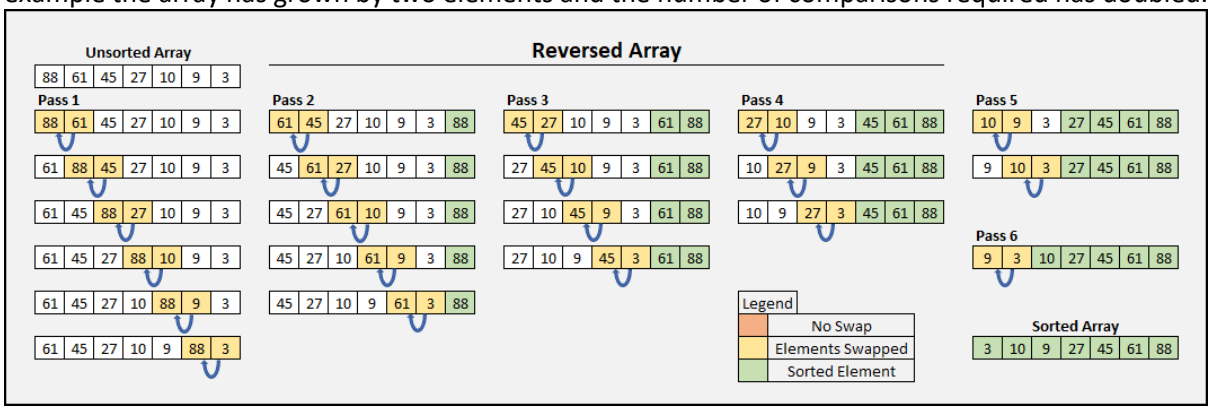

Figure 6 Bubble Sort Reversed Array, size 7

With the reversed array detailed in figure 6, it performs as expected with 21 comparisons and 21 swaps and 6 passes.

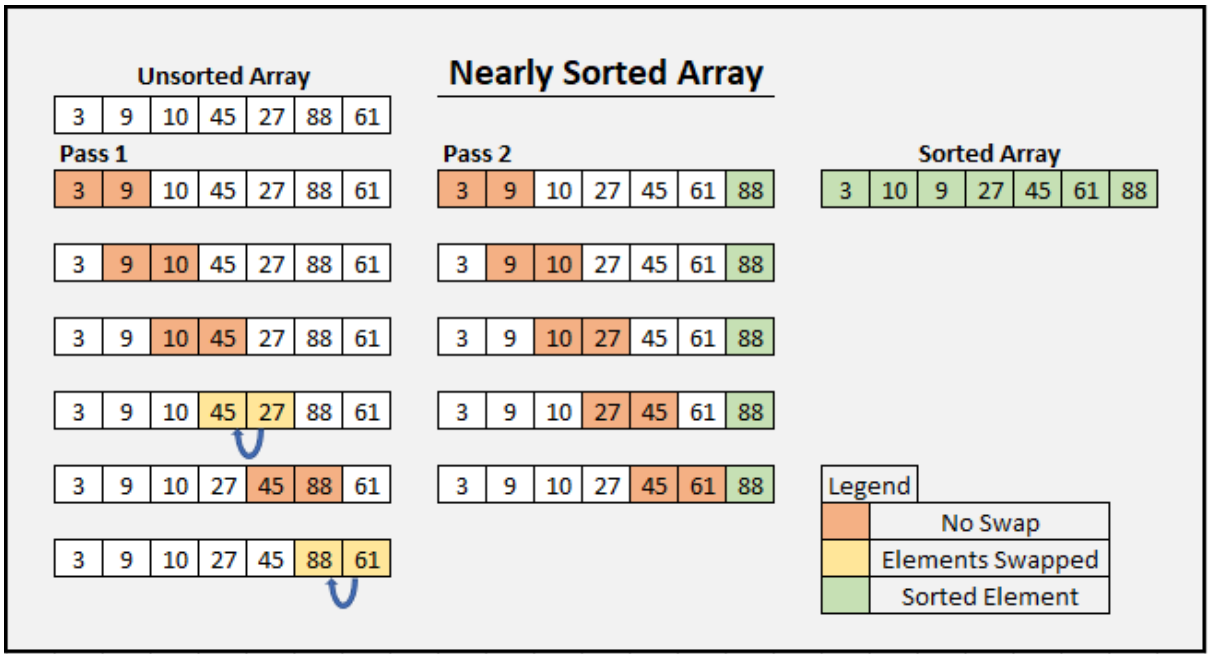

Figure 7 Bubble Sort Nearly Sorted Array, size 7

Figure 7 represents a nearly sorted array, a simulation of best case, or more accurately a more favourable configuration. As discussed, best case runs at 0(n) time, and here it is represented with the

same data set, that is almost sorted. Bubble sort performs the sort with just 2 passes, 11 comparisons and 2 swaps. This is significantly faster than the other two examples.

Algorithm

The step by step process the algorithms take to sort an array as detailed in the above figure is outlined below:

1. Take an array to be sorted numerically in ascending order
2. Start from the first element i = 0 and compare with the next element at i+1 with a loop.
3. If i > i+1 then swap the elements in the array. If it is not no swap takes place.
4. Next i is incremented in the loop and the process in 3 repeats.
5. Continue steps 3 and 4 until the comparison of the second last element and the last element has completed and the largest element is now at the end of the array.
6. Set i = 0 again and length of list is decremented by 1
7. Steps 3 to 6 are repeated until the first and second elements are processed, and the list is sorted.

Pseudocode:

Pseudocode for the steps above could be represented as follows:

```
Bubble Sort(arr)
        n = length arr
        For i from n -1 to 0, decrement by 1
            For j from 0 to i, increment by 1
                If arr[j]>[j+1]
                    swap arr[j] and arr[j+1]
```

Merge Sort

Merge sort is an example of an efficient comparison-based algorithm, developed by John von Neumann in 1945 it is one of the earliest sorting algorithms. It is still considered one of the most efficient algorithms and utilises the "Divide and Conquer" principle. It works by separating the elements of an array into several sub-arrays until each sub-array contains only one element, it then merges those subarrays into larger arrays sorting the elements as it goes. It finishes when all the elements are returned into a single sorted array.

Merge sort performance analysis is shown below:

- **Best Case** – n log (n)
- **Average Case** – n log (n)
- **Worst Case** – In all three case Merge sort has a run time of 0(n log (n)) which is logarithmic growth. As the array is always divided in the same way the data distribution will not affect performance.
- **Space Complexity** – O(n) it is linear as it requires as many new arrays as elements in the original array
- **Stable?** – Yes, the indexes of equal value elements will not be switched

As best, average and worst cases with Merge Sort are all n log(n), extensive analysis of different array types is not necessary, a quick comparison of a random and reversed array of size 5, and a random array size 7.

Figure 8 shows the process of a random and a reversed array for merge sort. It keeps dividing the arrays until all elements are in their own sub-array. Then it begins to merge them in order until the full array is reconstructed and in order.

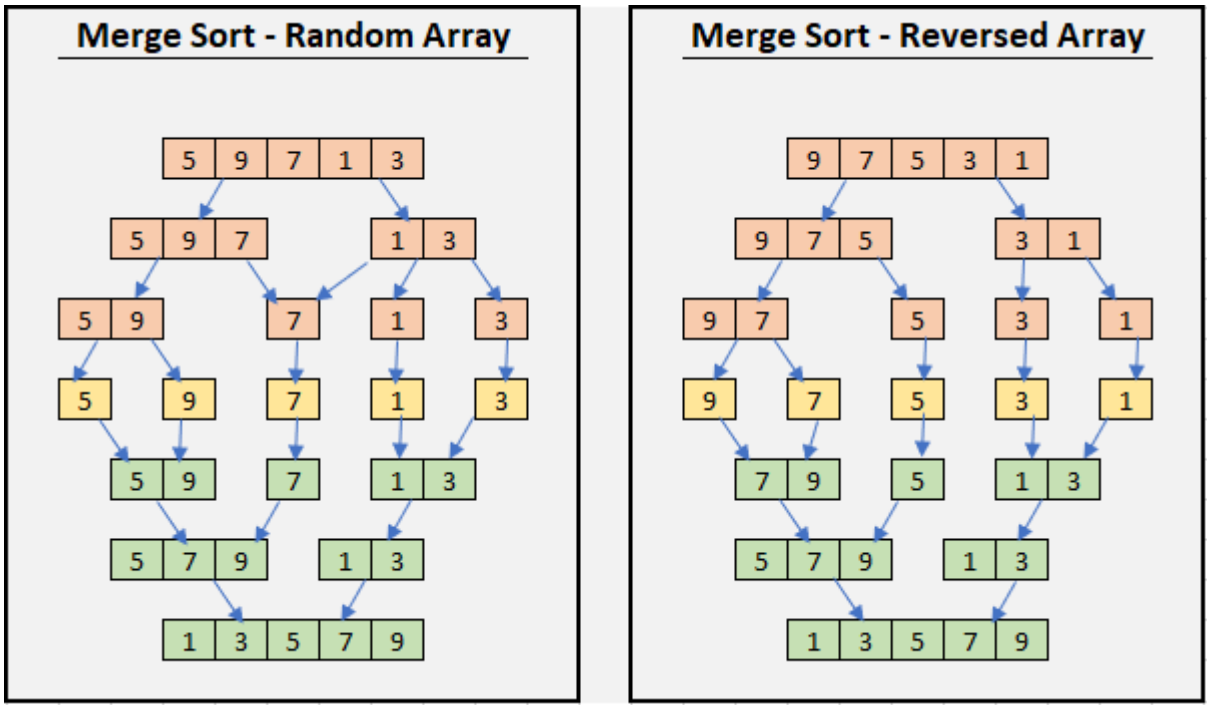

Figure 8 Merge Sort Random & Reversed Array, size 5

As seen in figure 8 and as expected there is no difference between the time and space performance between the two-array distributions.

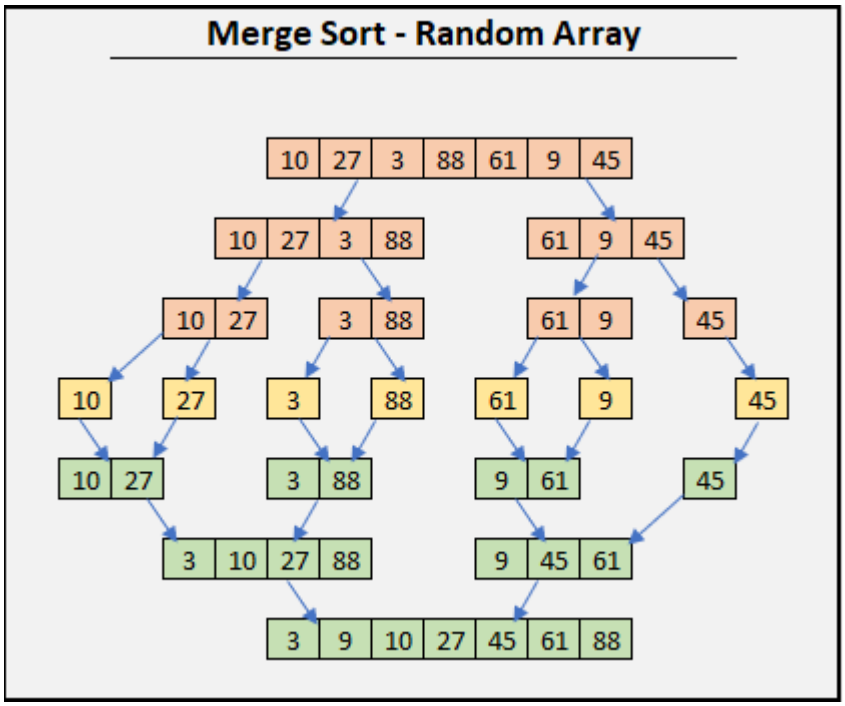

Figure 9 Merge Sort Random Array, size 7

In figure 9 the extra memory requirement increased as the array size increased, however the number of comparisons and runs has not increased significantly unlike what was displayed with Bubble sort.

Algorithm

The step by step process the algorithms take to sort an array as detailed in the above figure is outlined below:

1. Take an array to be sorted numerically in ascending order
2. Find the len(array) and the mid-point len(array)//2
3. Divide the array at the mid-point
4. While the array lengths > 1, recursively split the arrays into a left and a right array
5. Compare first single element array with the adjacent and merge in order
6. Continue merging like this as the arrays get larger and return to a full sorted array

Pseudocode

```
MergeSort(arr)
        If length array > 1 (base case)
                Mid = length array /2
                Left = first sub array
                Right = second sub array
        mergeSort (left,  right) recusively

        left index = 0
        right index = 0
        output = 0
```

while each index < each array
    if i < j append i to k array
    increment i
    else append j to k array
    increment j

append i to k
    increment i and k
append j to k
    increment j and k

Counting Sort

Counting sort was developed by Harold Seward 1954, it is an example of a non-comparison sorting algorithm. It works by iterating through the array and counts the number of instances of each distinct value, it uses those values to index the element in the new sorted array. [7]

First the highest value of the array needs to be established, then an array, of length based on this highest value is created. So, if the array contains numbers from 1-10, an array of length 10 will be created as the count array. The index of the count array is then appended to the new sorted array based on the count of the index value. This results in a very efficient sorting algorithm with a run time of 0(n) across the board, however it is limited in its implementation to integers with an established range.

Counting sort performance analysis is shown below where k equals the highest value of the array values:

- **Best Case** – 0(n) + k
- **Average Case** – 0(n) + k
- **Worst Case** - 0(n) + k
- **Space Complexity** – 0(n) + k
- **Stable?** – Yes

This algorithm is very efficient in all cases with a base run time and space complexity of 0(n) which is linear growth, with an influence of the k value of the array.

Figure 10 details the process of counting sort using an array of 10 integers where k =9. The process in the diagram works from left to right with a top-down direction. In the first stage the array to be sorted is taken and a counting index of length k is generated. It then iterates through the array tallying each distinct value. The result of that is shown in the next stage with the tally count at the top and the cumulative value of the count indexed at the bottom.

The next stage is the beginning of the sorting process, an empty output is generated, and the first element of the original array is referenced, in this example the element “5” is referenced to the count array index and the value of the count array is then used to reference the index of the new sorted array. In the example the element “5” refers to the count value of 6 and so “5” is appended to index 6 in the new sorted array.

In the next stage the value taken form the count array is decremented by 1 and the process is repeated for the next element in the original array, being “9”. This process is carried out ten times for each element in the original array and outputs a sorted array.

The figure shows that the distribution of the data in the original array will not affect the runtime, and the steps taken will always equal the length of the array. The only factor that affects performance other than the expected linear growth is the size of k.

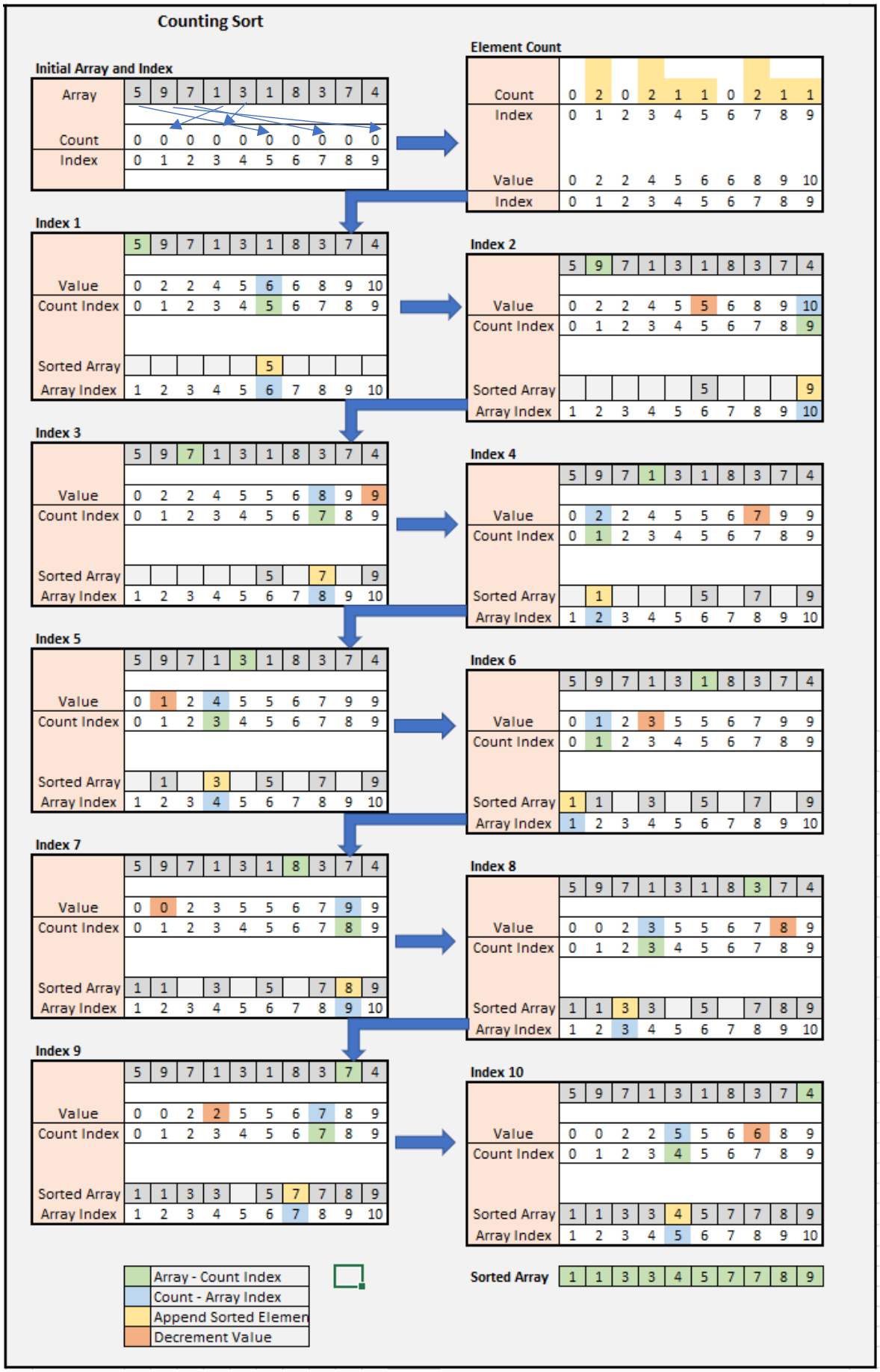

Figure 10 Counting Sort Random Array, size 10

Algorithm

The step by step process the algorithms take to sort an array as detailed in the above figure is outlined below:

1. Take an array to be sorted numerically in ascending order
2. Create a new array of a length equal to the highest value in the original array
3. Tally the instances of the elements of the original array, appending the counts to the related index in the new array.
4. Update the tally to make the values of the new array a cumulative count.
5. Iterate through the original array referencing the value to the index of the new array and appending the index of the new array to the sorted array.
6. Decrement the count value
7. Repeat with all elements of the original array
8. Output the sorted array

Pseudocode

```
CountingSort(arr)
    Max = largest element in array
    Min = smallest element in array
    countArr = 0 for j in range (max-min +1)
    sortedArr = 0 for j in range length of array

    for i in range length of arr
        count arr[i]
    for i in range length of countArr
        countArr[i] += countArr[i-1]
    for i in range length or array -1
        append count to sorted
        decrement count by 1
    return arr
```

Insertion Sort

Insertion sort works in a similar way to Bubble sort where it iterates through the array and compares each element with the adjacent element. It, compares the given element with all other elements to the left in the array and places that element in the correct position. The method utilised is often considered the most approachable, and the way humans carryout sorting processes. It is often compared to a card player sorting their hand. By picking a card scanning for its location and inserting it into the desired place, then repeating the process with the next card until the hand is in the correct order.

Due to the similar iterative design of Insertion sort to the previously discussed Bubble sort, it has the same complexity in best, average and worst-case scenarios.

- **Best Case** –O(n), Insertion sort works best with sorted and nearly sorted arrays and has linear growth.
- **Average Case** – Runtime of $O(n^2)$ representing quadratic growth
- **Worst Case** – Runtime of $O(n^2)$ representing quadratic growth
- **Space Complexity** – as an in-place algorithm it has a constant growth of O (1)
- **Stability** – Insertion sort maintains the original order of similar value elements and so is considered stable.

Figure 11 shows a simulation of Insertion sort in practice, sorting an array of 7 elements, with an estimation of average, worst and best case by using a random, reversed and nearly sorted array.

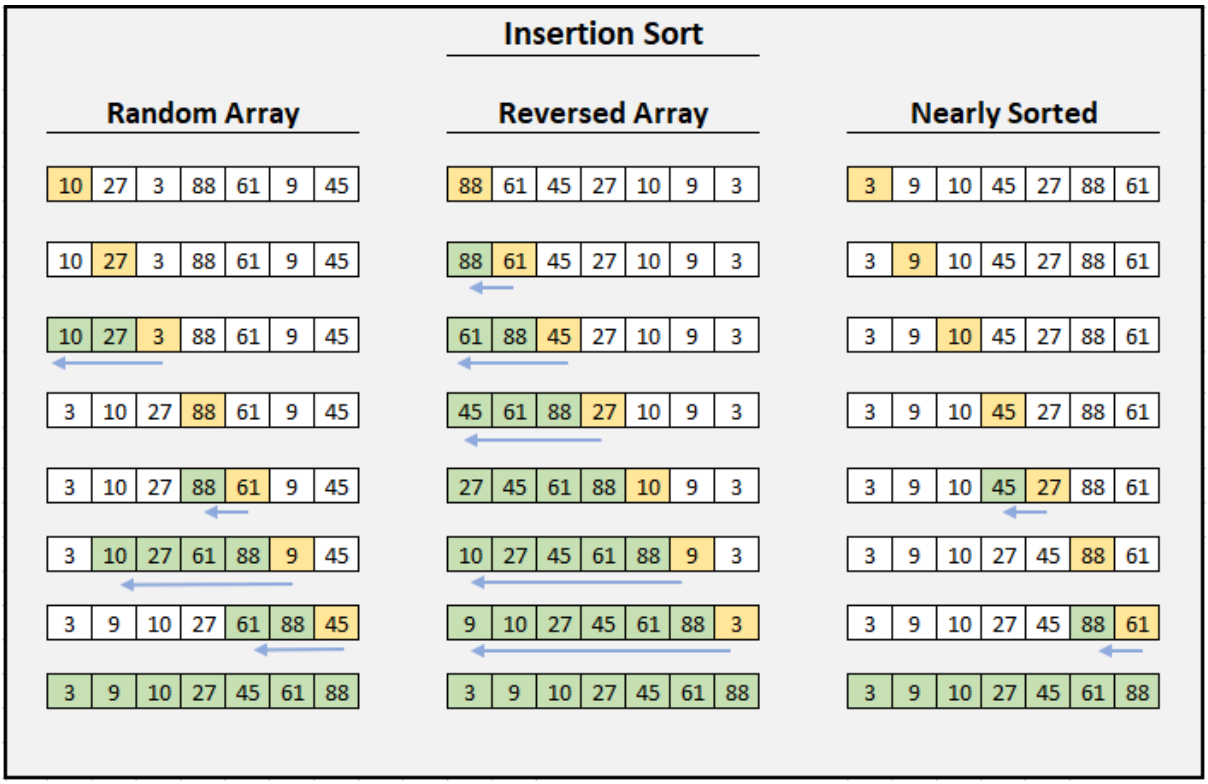

Figure 11 Insertion Sort Random, Reversed and Nearly Sorted Array, size 7

It starts by taking the second element in the array and assigning it as the key, it compares it to the element on the left, and swaps position if the key is smaller than the element to its left. It then moves on the next element in the list and keeps comparing to see if its smaller than all the elements to its

left. It stops checking and inserts the element when it reaches a number that does not instigate a swap. It iterates through the array until the last element has been processed and the array is sorted.

Looking at figure 14 in more detail it is shown that with the worst- and best-case, Insertion sort performs in a similar way to bubble sort with 21 and 8 comparisons, respectively. It does however perform a lot less comparison in the average case as it stops early when the correct position has been found. Based on this assessment it would be expected that Insertion sort will perform quicker than Bubble sort but within the same complexity band.

Algorithm

1. Take an array to be sorted numerically in ascending order
2. Assign the second element, i, in the array to be the key and compare with the sub array to the left.
3. If the key < the element to the left swap the elements
4. Take the next key or i +1 and compare with the sub array to the left now containing 2 elements.
5. Keep checking the key with the elements of the sub array and insert it when the key !< element.
6. This will iterate assigning new keys for every element in the array until the last value in the original array has been processed and the array is sorted.

Pseudocode

Pseudocode for the steps above could be represented as follows:

```
InsertionSort(arr)
        for i from 1 to length of array
            Key = i
            j = i-1
            while j >=0 and key < j
                swap j and key
            assign new key to i + 1
```

Tim Sort

Timsort was developed by the Software Engineer Tim Peters in 2002 and was designed to be the standard sorting algorithm for use in Python. Designed for use with real world datasets where it is more often already partially sorted. It utilizes a combination of Insertion sort and Merge Sort, it first creates and sorts small sub arrays or "runs" using insertion sort which is suited for smaller sets of data, then merges these subarrays using merge sort.

On an average or worst case run Timsort complexity is O(n log n) which puts it in the same category as Merge sort. However, Timsort improves upon Merge sort when using nearly sorted arrays and performs to O(n) which exhibits very efficient linear growth.

- **Best Case** – O(n) designed for use with nearly sorted arrays, equates to very efficient best-case performance.
- **Average Case** – Runtime of O(n log n) or logarithmic growth
- **Worst Case** – Runtime of O(n log n) or logarithmic growth
- **Space Complexity** – O(n) like Merge Sort space complexity grows linearly
- **Stability** – Based on two stable algorithms, Tim Sort is also considered stable

In more detail Tim sort works by taking advantage of pre-sorted chunks of data, which can be either ascending or descending. It starts by looking for the longest run for concurrent sorted data within the full array, in the defined range. It is known that Insertion sort works best on smaller data sets so Run range is often set to equal 32. If no Run of the minimum range is found Insertion sort is used to append more elements to the run until it is at least 32 elements. These sorted sub arrays (runs) are set aside for merging later. The algorithm reforms the data using the latter part of Merge sort to restructure the sub arrays into one sorted array. As discussed above merging data of sub arrays into a new array is extremely efficient when the sub arrays are sorted. [8]

As Timsort is designed to work with real world data, which is often partially sorted, it may not perform to it's potential in the benchmarking tests. This is because it will be working with randomly generated arrays and so n log n performance is to be expected.

The following figures show the three steps of the Tim sort algorithm for an average case and a nearly sorted case with a min Run = 3. It can be seen, that much of the space complexity of merge sort has been reduced. Although the insertion sort section is not described here, it can be seen from the numbers that with the random array a few comparisons and insertions must be made to create the two sub-arrays. So, in this case, with the process of insertion sort and merge sort it runs at a similar complexity to merge sort. However, with the nearly sorted array, the left sub array can be appended in place without any sorting and so Insertion sort is not required in the right array only one swap is required so it has performed much more efficiently than the standard Merge sort.

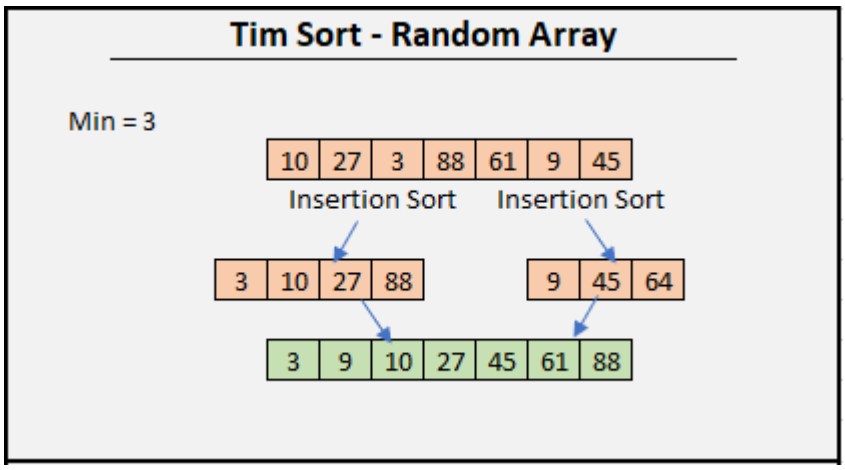

Figure 12 Tim Sort Random Array, size 7

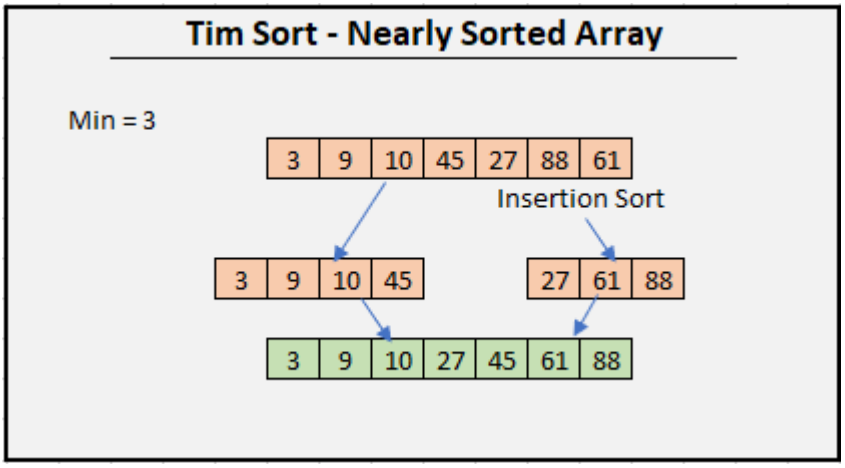

Figure 13 Tim Sort Nearly Sorted Array, size 7

Algorithm

1. Calculate the min Run, using a binary search method to implement Insertion Sort
2. Iterate over the full array to create the subarrays based of the size of min Run and sort each subarray using Insertion sort.
3. Utilize Merge sort to join the sorted subarrays into one sorted array

Pseudocode

Pseudocode for the steps above could be represented as follows:

```
timSort(arr)

    n = length array
    minRun = 32
    for i in range (0,n,minRun)
            insertionSort()
    size = minRun
    while size < n
    merge(sub-arrays)
    size = size *2
```

Implementation

To perform a benchmarking test on these algorithms they need to be adapted to run in Python. The python code used in the program for the algorithms is described below, followed by the timing function, generation of random arrays and the run function.

Bubble Sort:

The code sample below is the python implementation of Bubble Sort adapted, from lecture notes [9].

```
def bubbleSort(arr):

    for i in range(len(arr)-1, 0, -1):
        for j in range(0,i,1):

            if arr[j] > arr[j+1]:
                swap = arr[j]
                arr[j] = arr[j+1]
                arr[j+1] = swap
```

The first line sets the function with the given array. The first for loop sets up the amount of passes it will need to perform, which equals array length -1, decrementing by 1 for every pass. Then a for loop to compare the indexes is begun. If element in question is greater than element + 1 it will create a temporary placeholder for the element, make j = j+1 then overwrite j+1 with the placeholder j, effectively switching the two elements.

Figure 14 Bubble sort python code

Insertion Sort:

The code sample below is the implementation of Insertion Sort, adapted from GeeksforGeeks [10].

```
def insertionSort(arr):

    for i in range(1, len(arr)):

        key = arr[i]

        j = i-1
        while j >= 0 and key < arr[j] :
                arr[j + 1] = arr[j]
                j -= 1
        arr[j + 1] = key
```

The first line sets the function with the given array, it then iterates through the array setting the current test element as the “key” (key = arr[i]).

The comparison element is set to the adjacent integer to the left, and while the key is less than the comparison element in sub array, it moves on to the next element in the subarray to compare with the key. When the key reaches an element that it is larger or equal to it gets inserted into that position arr [j + 1] = key. It then loops back to set the next key.

Figure 15 Insertion sort python code

Merge Sort:

The code sample below is the implementation of Merge Sort, adapted from GeeksforGeeks [11].

```
def mergeSort(arr):
    if len(arr) > 1:

        mid = len(arr)//2
        left = arr[:mid]
        right = arr[mid:]

        mergeSort(left)
        mergeSort(right)

        i = j = k = 0

        while i < len(left) and j < len(right):
            if left[i] < right[j]:
                arr[k] = left[i]
                i += 1
            else:
                arr[k] = right[j]
                j += 1
            k += 1

        while i < len(left):
            arr[k] = left[i]
            i += 1
            k += 1

        while j < len(right):
            arr[k] = right[j]
            j += 1
            k += 1
```

Figure 16 Merge sort python code

The first line sets up the function with the given array. Then the base case for the recursive part of the function, to keep splitting the arrays until the length = 1.

Find the mid-point of the array and create 2 subarrays either side of the midpoint. Then recursively run the sort on the left and right sub arrays.

Compare lowest left value with lowest right value and append to array k. To begin merging the sub-arrays.

If left array is remaining with no corresponding right array append all left array to array k, and visa versa.

Counting Sort:

The code sample below is the implementation of Counting Sort, adapted from GeeksforGeeks [12].

```
def countingSort(arr):
    max_k = int(max(arr))
    min_k = int(min(arr))

    countArr = [0 for j in range(max_k - min_k +1)]
    sortedArr = [0 for j in range(len(arr))]

    for i in range(0, len(arr)):
        countArr[arr[i]-min_k] += 1

    for i in range(1, len(countArr)):
        countArr[i] += countArr[i-1]

    for i in range(len(arr)-1, -1, -1):
        sortedArr[countArr[arr[i] - min_k] - 1] = arr[i]
        countArr[arr[i] - min_k] -= 1

    for i in range(0, len(arr)):
        arr[i] = sortedArr[i]

    return arr
```

Figure 17 Counting Sort python code

The first line sets up the function with the given array.

Then find and define max and min values for k.

Generate count array and output array.

Iterate through the array tallying occurrences to each value.

Cumulatively add the tally.

Append the count value indexes to reference value to append to the sorted array.

Tim Sort:

The code sample below is the implementation of Tim Sort, adapted from GeeksforGeeks [13].

```python
MIN_MERGE = 32

def calcMinRun(n):
    r = 0
    while n >= MIN_MERGE:
        r |= n & 1
        n >>= 1
    return n + r

def insertion(arr, left, right):
    for i in range(left + 1, right + 1):
        j = i
        while j > left and arr[j] < arr[j - 1]:
            arr[j], arr[j - 1] = arr[j - 1], arr[j]
            j -= 1
```

This is the implementation of Tim Sort. The min Run size is set to 32.

Def calcMinRun processes the minimum bounds and sets minRun to be less than or equal to a power of 2.

This is an adapted insertion sort to be carried out on the sub arrays where required to reach minRun size.

Figure 18 Tim sort (calcMinRun & insertion) python code

```python
def merge(arr, l, m, r):

    len1, len2 = m - l + 1, r - m
    left, right = [], []
    for i in range(0, len1):
        left.append(arr[l + i])
    for i in range(0, len2):
        right.append(arr[m + 1 + i])
```

Figure 19 shows the start of the adapted merge sort algorithm for use within Tim Sort. It sets the length of the left and right array and creates to empty array left and right. To append the original array into two parts.

Figure 19 Tim sort (merge 1) python code

```
116
117         i, j, k = 0, 0, l
118
119         while i < len1 and j < len2:
120             if left[i] <= right[j]:
121                 arr[k] = left[i]
122                 i += 1
123
124             else:
125                 arr[k] = right[j]
126                 j += 1
127
128             k += 1
129
130         while i < len1:
131             arr[k] = left[i]
132             k += 1
133             i += 1
134
135         while j < len2:
136             arr[k] = right[j]
137             k += 1
138             j += 1
```

Figure 20 Tim sort (merge 2) python code

It then uses the latter end of the original Merge sort algorithm discussed in above, to combine the the two arrays into a larger sorted sub array.

Figure 21 shows the implementation of Tim Sort and how it utilises Insertion Sort and Merge Sort.

It calls in the array sets a variable for its length and defines the minRun.

It then run insertion sort iterating from 0 to the end of the array in increments of minRun.

```
def timSort(arr):
    n = len(arr)
    minRun = calcMinRun(n)


    for start in range(0, n, minRun):
        end = min(start + minRun - 1, n - 1)
        insertion(arr, start, end)

    size = minRun
    while size < n:

        for left in range(0, n, 2 * size):

            mid = min(n - 1, left + size - 1)
            right = min((left + 2 * size - 1), (n - 1))

            if mid < right:
                merge(arr, left, mid, right)

        size = 2 * size
```

Figure 21 Tim sort python code

Insertion sort is used to sort and append elements from the rest of the main array into the sub array until it reaches the minRun length.

Once all the sorted subarrays have been set up, it starts the merge part. It starts to merge the sub arrays of size minRun into an array of minRun size *2. It then doubles the new larger sub array size and does the same again until all the sub arrays have been merged into on sorted array.

Benchmark Code

Once the algorithms have been coded and tested to run in python, the code to test the performance of each can be implemented.

```
167  ##############################################################
168  arraySizes = (100,250,500,750,1000,1250,2500,3750,5000, 6250, 7500,8750,10000)
```

Figure 22 array sizes list

Figure 22 shows the test cases of 13 different array sizes to produce random arrays of increasing size to sort and record the time taken.

```
191
192  def sortArray(algorithm):
193      sortTimes = []
194
195      for size in arraySizes:
196
197          times = timeLog(algorithm, size)
198          sortTimes.append(times)
199
200      return (sortTimes)
201
202  ###########################################################################
203
204  bubbleTime = sortArray(bubbleSort)
205  mergeTime = sortArray(mergeSort)
206  countingTime = sortArray(countingSort)
207  insertionTime = sortArray(insertionSort)
208  timTime = sortArray(timSort)
209
210  ###########################################################################
```

Figure 23 Python Benchmark program (sortArray)

Figure 23 Shows how the sort array runs by calling the specific algorithm to test, it creates an empty array “sortTimes” where it will append the average run times. It then iterates through the “arraySizes” list testing the time taken for each array size by calling the “timeLog” function. It then appends the results from each test to “sortTimes” and returns the array. This is implemented five times one for each sorting algorithm.

The “timeLog” function is shown in figure 24, it gets called with the size of array to generate and the algorithm to test. It then generates an array of arrays of random integers for the given array size. The timer is started, and sort process is carried out for the ten arrays and once complete the timer is stopped. The average run time is calculated and converted to milliseconds.

Lastly a data frame is created to output the results into a tabular format and a chart is plotted to graphically display the results in a way that can be compared to the complexity plots in figure 1.

```
167  ############################################################
168  arraySizes = (100,250,500,750,1000,1250,2500,3750,5000, 6250, 7500,8750,10000)
169  sortAlgorithm = ["Bubble Sort", "Merge Sort", "Counting Sort", "Insertion Sort","Tim Sort"]
170  ############################################################
171
172  def timeLog(algorithm, size):
173
174      testArrays = []
175
176      for _i in range(10):
177          randomArray = [random.randint(0, 100) for i in range(size)]
178          testArrays.append(randomArray)
179
180      start_time = time.time()
181
182      for randomArray in testArrays:
183
184          algorithm(randomArray)
185
186      end_time = time.time()
187      time_elapsed = ((end_time - start_time)/10) *1000
188      return round(time_elapsed,3)
189
```

Figure 24 Python Benchmark program (timeLog)

Benchmarking

Having run the program, the average of the times for each algorithm and array test sizes were output into a data table, this is shown in figure 25.

	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
bubbleSort	0.898	5.485	20.348	48.966	84.484	133.047	544.688	1341.294	2268.636	3422.521	4920.021	6761.944	8880.954
mergeSort	0.299	0.796	1.695	2.596	3.690	4.789	10.572	15.555	23.637	29.469	35.515	42.583	49.519
countingSort	0.000	0.100	0.299	0.499	0.598	0.798	1.496	1.995	2.493	3.189	4.391	4.890	5.685
insertionSort	0.402	2.590	11.173	25.131	43.580	70.412	286.287	597.374	1096.442	1737.159	2389.822	3266.245	4238.929
timSort	0.199	0.801	1.795	2.693	4.092	4.588	10.073	18.354	22.341	30.469	39.096	40.991	48.570

Figure 25 Benchmark output table

From initial inspection, Counting Sort was by far the quickest in every array size, Merge Sort and Tim Sort had similar performances throughout, with slight variation in some array sizes, for example with size 3750 Tim Sort was noticeably slower than Merge Sort which is most likely to do with the random arrays they tested against. Insertion sort was almost 10x slower than Merge and Timsort which could be expected based on the established runtime complexity. It was however over 2x quicker than Bubble sort, which is an example of the variation within the defined complexity bands. Referring to figure 2 the results match the expected run times based of the average case complexity

- Bubble sort – $O(n^2)$
- Insertion sort - $O(n^2)$
- Merge Sort – O(n log n)
- Tim Sort - O(n log n)
- Counting Sort – O(n + k)

Figure 26 shows the plot of the 5 algorithms based on array size and the average time taken to

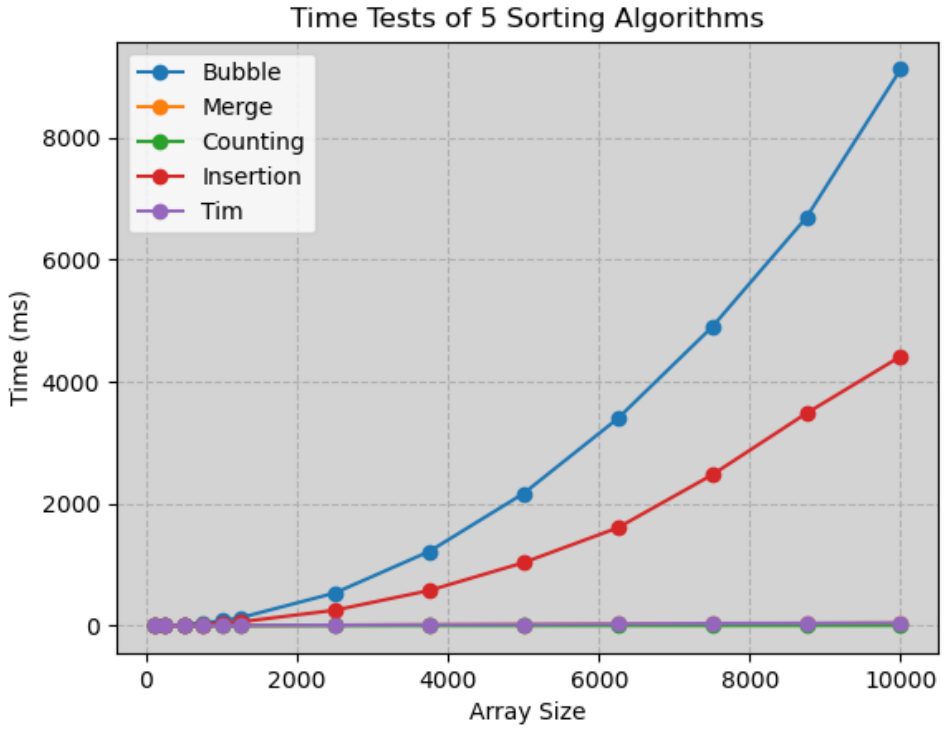

Figure 26 Plot of test times

sort the given array. This shows the quadratic growth expected from Bubble and Insertion sort however due to the inefficiency of these algorithms it is hard to see the performance of the faster 3 algorithms. Figure 27 plots the same data on a logarithmic scale.

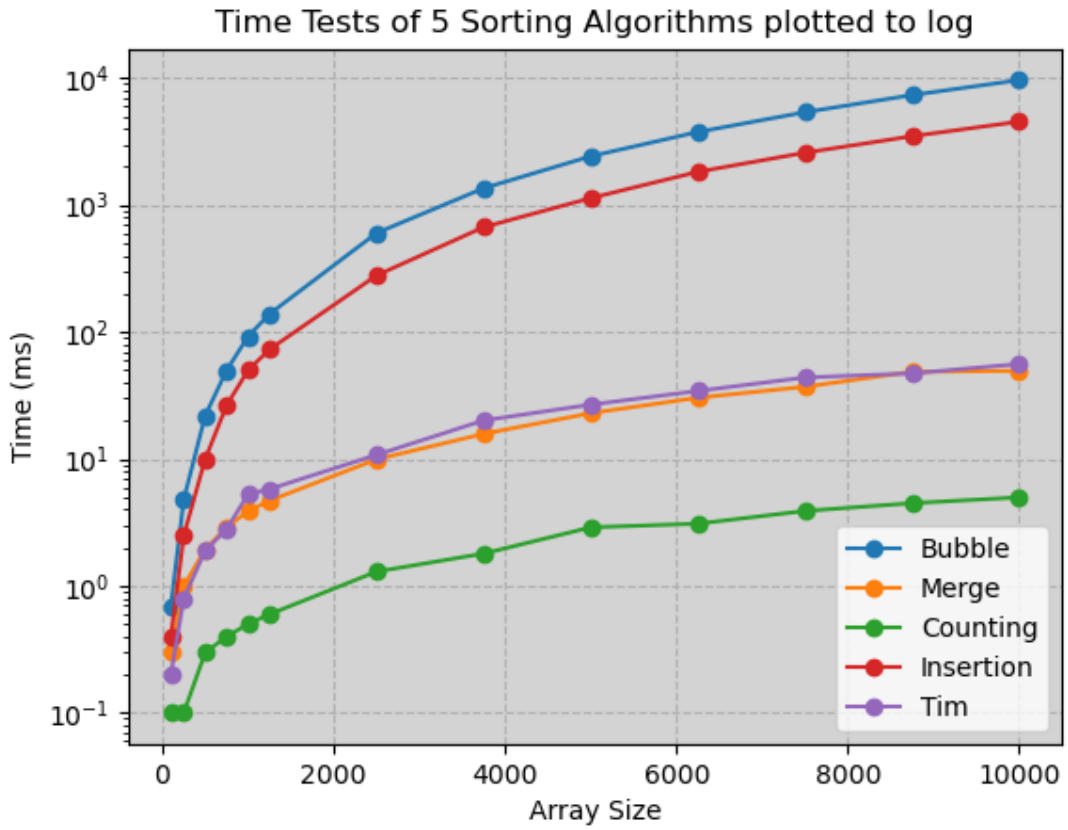

Figure 27 Plot of test times on log scale

Here the separation of the 5 algorithms over the 3 complexities can be seen with Bubble and Insertion running at $O(n^2)$, Merge and Tim Sort running at $O(n \log n)$, and Counting sort running close to $O(n)$. This is in line with the expectations from the theoretical diagrams above.

Conclusion

With these algorithms analysed and tested strengths and weaknesses of each algorithm can be discussed. As there is no perfect algorithm for every situation.

Bubble Sort

Bubble sort cannot really be considered as a viable solution in any real-life situations, although it is simple to implement and runs in-place, requiring little memory overheads, it is very slow when dealing with larger arrays and there are more efficient, simple in-place comparison algorithms to utilise.

Insertion Sort:

Insertion sort has the same benefits of Bubble sort, being simple to implement, sorts in-place, and is very efficient with short arrays. Although it does not scale as well as more efficient algorithms, it has been utilised very successfully into other hybrid algorithms such as Tim Sort.

Merge Sort:

Merge sort is efficient, and it scales very well as the input grows, with a complexity of $O(n \log n)$ across all cases. This however means it can be very inefficient sorting shorter arrays and has much larger memory requirements than in-place solutions

Counting Sort:

Counting sort runs with times close to linear growth, however this comes at the cost of having to have established information of the dataset and so it is not as widely applicable as other alternative algorithms.

Tim Sort:

Tim Sort is much more complex than the other algorithms discussed, this makes it harder to implement and requires much more coding. Timsort is adaptable as it can run as a single Insertion sort algorithm when running short arrays and addresses the scalability issues of merge sorts space complexity. Even though it has not been shown in this analysis, it has been shown to perform very well with partially sorted real world data.

In short and in the words of Barack Obama:

“I think the bubble sort would be the wrong way to go."

References

1. [https://learnonline.gmit.ie/pluginfile.php/297433/mod_resource/content/0/07%20Sorting%20Algorithms%20Part%201.pdf] (slide 2)
2. https://en.wikipedia.org/wiki/Von_Neumann_architecture
3. https://realpython.com/sorting-algorithms-python/
4. https://www.bigocheatsheet.com/
5. https://afteracademy.com/blog/comparison-of-sorting-algorithms
6. https://learnonline.gmit.ie/pluginfile.php/297433/mod_resource/content/0/07%20Sorting%20Algorithms%20Part%201.pdf
7. https://www.interviewcake.com/concept/java/counting-sort
8. https://medium.com/@rylanbauermeister/understanding-timsort-191c758a42f3
9. https://learnonline.gmit.ie/pluginfile.php/313257/mod_resource/content/0/Sorting.py
10. https://www.geeksforgeeks.org/insertion-sort/
11. https://www.geeksforgeeks.org/merge-sort/
12. https://www.geeksforgeeks.org/counting-sort/
13. https://www.geeksforgeeks.org/timsort/