Computational Thinking with Algorithms

H.Dip in Computing (Data Analytics)

An Overview and Analysis of Sorting Algorithms

Conor McCaffrey

G00286552

Contents

Introduction

Sorting

Sorting data is one of the most fundamental operations of a computer. However, it is also one of the most power-intensive and time-consuming operations of a computer.

Think of the following simple example: books on a shelf are ordered according to the surname of the author. You are asked to find the book wrote by an author with a surname beginning with 'S'. This is simple, as the books are already sorted. You select the book and move on. Let's now imagine you are asked to select this same book on a shelf where the books are not ordered. This simple request is now much more difficult and time-consuming. Why? This is because the books are not sorted. This is the power of sorting.

It has been estimated that a computer can dedicate as much as 25% of CPU cycles on sorting alone [1]. The process of sorting is generally the first step in a computational program. This demonstrates the significance in being able to sort data in an efficient manner. While sorting data, for example a list of 10 elements and you want to find the minimum value, may appear to be simple, once the number of elements enters into the hundreds or perhaps even thousands, this same process becomes extremely complex [2]. This brings us to the concept of *sorting algorithms*. The essence of sorting algorithms (and therein, it's beauty) is simple: take in an unordered set of elements and arrange these elements in a pre-determined manner, using a comparison operator [2,3]. This comparison operator is the basis upon which this previously unordered data is now to be sorted [3]. This, now ordered, set of data can then be used for further downstream analysis or whatever computational operation you wish to carry out.

Evolution of the Sorting Algorithm

The spark that set of the development of a plethora of sorting algorithms was provided by John von Neumann [4], who in 1945 developed the 'Merge Sort', a comparison-based sorting algorithm that employs a 'divide and conquer' approach to determine the order of a set of elements [5,6]. We will discuss this in greater detail at a later stage. This was closely followed by Harold H Seward, whom in 1954 developed the 'Radix Sort', a non-comparison sorting algorithm that sorts elements into 'buckets' according to their radix [7]. The same mathematician was responsible for developing the 'Counting Sort' in the same year, a technique in where the occurrences of each element is counted and then constructing the input based on these occurrences [8]. Other notable examples include 'Quicksort', developed by C.A.R Hoare in 1962 and also Timsort, developed by Tim Peters in 2002 [4]. The timespan of development of fundamental sorting algorithms demonstrates how important this area of research is, with new implementations constantly being developed.

Complexity of the Sorting Algorithm

Simply defined, the complexity of an algorithm can be determined as the time needed for an algorithm to run [9]. However, this can be broken down into several smaller sections. Prior to delving too deeply into complexity, it would be prudent to briefly touch on *efficiency*. Time efficiency can be considered as the amount of actions/operations carried out by the execution of an algorithm whereas space efficiency can be determined as simply the amount of memory storage that is necessary for an algorithm to run [10]. This second factor can be affected by various parameters such as input size and structure of the data [10].

There are two defined ways in order to analyse the time efficiency of an algorithm:

- A Priori analysis: a purely theoretical efficiency analysis. Efficiency is determined by comparison of the order of growths. This method is a measure of complexity as details such as processer/hardware/language of compiler are independent of analysis. The result returned is an approximate answer [11,12].

- A Posteriori analysis: an empirical evaluation of efficiency. The algorithm to be analysed is implemented on a programming language and executed on a target platform. The efficiency of the algorithm is determined through measurements collated during execution [11,12]. As such, this type of analysis is dependant on system hardware and processer as is deemed to be a measure of performance [12]. The result returned is an exact answer as we have collected data during an algorithms execution.

We can determine the complexity of an algorithm by measuring how long it would take to complete given an input of size *n* [12,13]. Therefore, this can be evaluated mathematically and several 'complexity families' exist for describing the complexity (in this case, *running time complexity*) of an algorithm [13].

Complexity Family Classification	**Notation (Big-O)**	**Description**
Constant	1	Best case, length of input size has no effect
Logarithmic	log(n)	As input size grows, number of operations also grows but at a slow rate [14]
Sublinear	<n	Execution time grows slower than the input size [15]
Linear	n	Execution time grows as input size increases
Linearithmic	n log n	Conjoining of linear and logarithmic
Polynomial	n^2, n^3, n^4..	Running time of polynomial of the size of the input [16]
Exponential	2^n	Execution time grows exponentially as input size increases

Table 1 Overview of Complexity Families with respect to Big-O Notation (adapted from reference 13 and in-table references)

The descriptions for each of the complexity families are relative to their 'Big-O' notation. Big-O notation is the standard approach in assessing algorithmic complexity, as it assumes a *worst-case scenario* for execution time [13]. We will discuss this concept in more detail in the next section.

We can also display graphically the running time of an algorithm with respect to input size, again in relation to Big-O notation [17]:

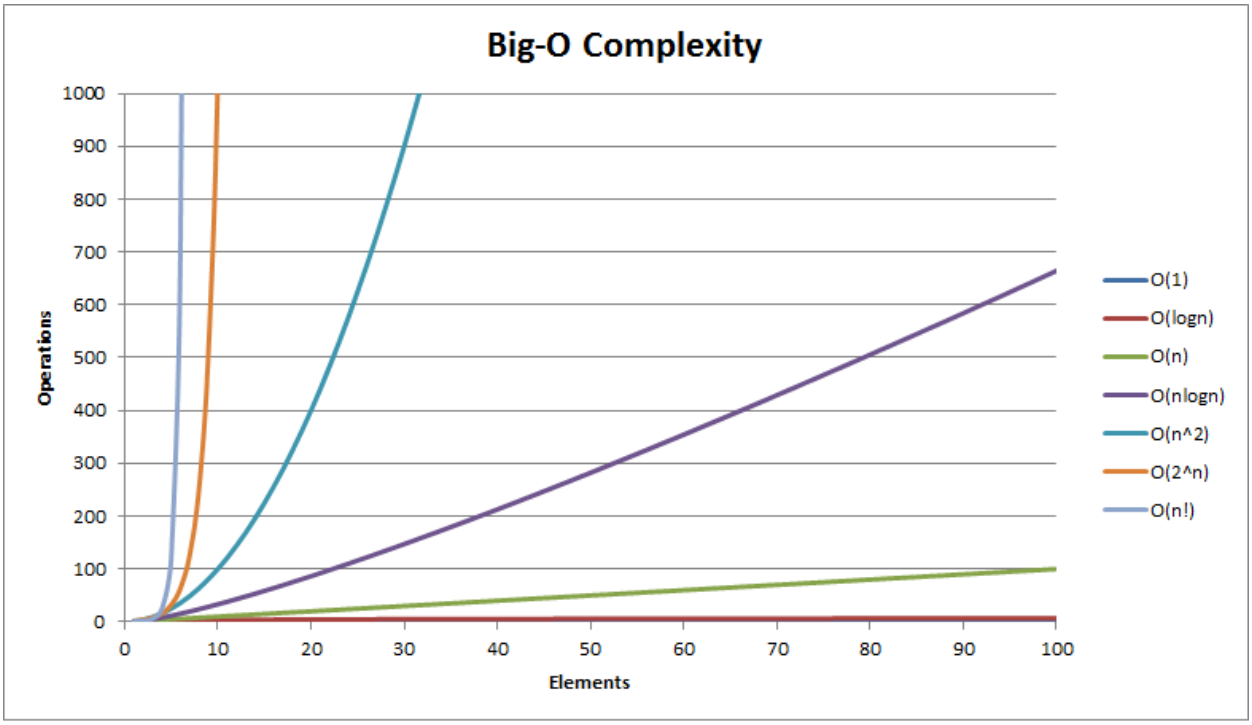

Figure 1 Overview of complexity curves for algorithms with respect to Big-O notation [17]

Best, Average and Worst Cases

While we have just discussed how the running time of a sorting algorithm can increase as the input size *n* increases, we also need to consider the effect of the actual data itself and how this can affect the running time of our algorithm [18,19]. There are various considerations we could make:

- What if the data is not sorted? Or what if it is almost sorted already?
- What if we have duplicate values?
- What if the element we want is at the beginning of our input? Or at the end? Or somewhere in the middle? Or not there at all?

This last point is a good segue to introduce the idea of best (ideal) case, average-case and finally worst-case with respect to running times for an algorithm.

- Best Case: This case provides the lower bound for running time of an algorithm [19]. For a simple example of a Linear Search, our desired element would be at the very beginning in this case. Naturally, this is unlikely to occur in reality and so little information can be gleaned from this.

- Average Case: This would be the behaviour predicted of an algorithm over a range of input instances [18,19]. The obvious query arises of what exactly is an 'average' input instance. In this case, prior knowledge of the mechanics/functionality of an algorithm would be beneficial, which is unhelpful [19].

- Worst Case: Perhaps slightly unintuitively, analysing the worst-case scenario is the most beneficial in regards to algorithmic running times. Here, we are finding out the upper-bound of an algorithm and the number of operations that would be required [19]. Thinking of our Linear Search example again, a worst-case scenario would be that the desired element isn't in the input array/list at all. Time (and memory) has been spent looking for an element that isn't present. Knowing our worst-case performance is useful as we know this level will never be exceeded.

Big-O Notation

It would be remiss to discuss best, average and worst-case complexities without briefly covering Big-O Notation. This notation is used to describe the complexity of an algorithm in the worst-case scenario [18]. We have touched on the notation in Table 1 when describing Complexity Families and again in Figure 1, for Complexity Curves. Big-O Notation is used to describe an algorithms time/space complexity in where, if we state that Algorithm A and Algorithm B have the same Big-O notation, they are considered similar in terms of time/space complexity (running time and memory usage) [18,20]. Referring back to Table 1, we can see that O(1) (Constant Time) is the most efficient runtime while O(n) (Linear Time) describes a run-time that increases as the input size increases.

It would be prudent to highlight also Ω (Omega Notation), which is used to describe an algorithm in the best case, or it's lower-bound [18]. As discussed previously, analysis of best-case complexity is not quite as useful as analysis of worst-case complexity.

Performance

While it would be easy to combine complexity/performance into one single stream of thought when describing algorithms, there are some differences between the two which should be highlighted:

- Performance can be affected by computer-specific parameters such as compilers and speed of the computer whereas Complexity talks to the resource requirements of an algorithm as the input size increases [18]
- The performance of an algorithm is affected by the complexity of the algorithm, however the converse is not true [18].

In-Place Sorting

We have already touched on memory requirements in regards to sorting algorithms in an earlier section. An 'in-place algorithm' is a desirable property of an algorithm as it uses only a fixed amount of additional space for variables, with the output being produced in the same memory as the input [18,21]. In-place algorithms generally overwrite the input with the constructed output in the same memory [22]. Thinking of this concept logically, what if we just wanted the reverse of a person's name? Wouldn't it make sense to just overwrite the string with the reversed string as we have no need any more for the original input? The only extra thing required would be an additional variable to hold elements while the reverse operation is executing. This is the extra additional space we previously described [21]. Examples of 'in-place' sorting algorithms include Bubble Sort, Selection Sort, Insertion Sort and Heapsort [21].

It is worth highlighting that 'out-of-place' sorting also exists [22]. In our previous example, an 'out-of-place' sorting algorithm may create a new array of the same length as the original input array and then delete the original array after reversing [22]. We can deduce immediately that this process is slower as we have discussed the creation and deletion of arrays. Merge Sort is an example of an 'out-of-place' sorting algorithm [23].

Stable Sorting

Another desirable property of a sorting algorithm is stability [4]. A sorting algorithm is considered 'stable' if repeated elements do not swap positions in the final output array i.e., the order is preserved [24]. Unstable algorithms, naturally, do not preserve the order of these repeating elements which can have downstream consequences [24].

Let's consider a simple example of coloured balls with numbers. This unsorted collection of balls has a blue ball followed by a white ball with the number '8' on them. A 'stable' sort would sort these balls in order but maintain the order of the two '8' balls: with the blue ball coming first in the sorted collection followed by the white ball. Stability is important as all sorting algorithms work on the premise of *sort keys* and *satellite data* [4,23]. Satellite data is the data that is associated with a sort key, while a sort key is the information the sorting algorithm acts upon [4,23]. Merge Sort, Timsort and Bubble Sort are examples of common sorting algorithms that are stable by nature [24].

Comparison Sorting

We have already detailed how sorting algorithms can be sub-divided into different groups depending on certain properties i.e., stable/unstable or in-place/out-of-place, and there is one more distinction that is useful when describing sorting algorithms: comparison/non-comparison-based sorting algorithms.

Firstly, we should cover the prevalence of *comparator functions*. These functions can accept two parameters from a collection and return an answer upon a certain condition i.e., if the first element is larger than the second, return '-1', or if they are equal return '0' otherwise return 1 [4]. We can define our own conditions for comparison when using standard sorting algorithms [4].

In comparison-based sorting, pairs of elements in a collection are compared with each other using a single comparison (less than or equal to, generally) in order to determine which element should appear first in the sorted output [4,25]. These types of algorithms don't take into account information such as frequency of occurrences of elements, they work solely on comparison of pairs of elements [25]. Assume we have a collection of numbers ranging from 1-25. A comparison-based sorting algorithm will analyse each pair of numbers and sort them accordingly. An advantage of the primacy of comparison-based sorts is that it is able to handle diverse sets of data [26]. This same advantage also has a downside however in the fact that the execution time is limited due to the number of operations needed (comparisons) [26]. As a result, comparison-based sorts have an average/best-case time complexity of *n logn* [4,27]. Examples of common comparison-based sorting algorithms include Bubble Sort, Quick Sort, Heap Sort and Selection Sort [27].

Non-comparison sorts do not perform primitive comparisons of two elements but rather use information about the sort keys and other operations to determine the final output [27]. These types of sorting algorithms assume that the keys are restricted to a certain data type [5]. As a result of the special input conditions for non-comparison sorts, an average/best-case time complexity of *n logn* no longer applies [27]. Examples of non-comparison sorts include Counting Sort, Radix Sort and Bucket Sort [4].

The following figure can summarise some concepts we have discussed:

Algorithm	Time Complexity (Best)	Time Complexity (Average)	Time Complexity (Worst)	Space Complexity	Stability
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)	Stable
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)	Unstable
Insertion Sort	O(n)	O(n^2)	(n^2)	O(1)	Stable
Merge Sort	O(nlog(n))	O(nlog(n))	O(nlog(n))	O(n)	Stable
Quick Sort	O(nlog(n))	O(nlog(n))	O(n^2)	O(log(n))	Unstable
Radix Sort	O(kn)	O(kn)	O(kn)	O(d + n)	Stable

Figure 2 Sorting Algorithms Comparison Table [28]

Sorting Algorithms

Chosen Algorithms

For this section of the project, I wish to give an overview of the 5 sorting algorithms I have chosen to analyse. I will discuss their respective time/space complexity and explain the functionality of each algorithm with diagrams. The table below contains the chosen algorithms:

Sorting Algorithm	**Description**
Bubble Sort	Simple comparison-based sort
Quick Sort	Efficient comparison-based sort
Counting Sort	Non-comparison-based sort
Selection Sort	Simple comparison-based sort
Insertion Sort	Simple comparison-based sort

Table 2 Chosen Sorting Algorithms

Bubble Sort

Bubble Sort has its origins in a landmark paper by Friend in 1956 which describes an algorithm by 'sorting by exchange' [31,32]. The term 'Bubble Sort' does not gain recognition in print until the release of a book entitled 'A Programming Language' by Iverson in 1962 [33,32]. Bubble Sort can be referred to as a 'push-down' sort in that an element moves one place at a time and cannot 'jump' from beginning to end or to some other index [34]. Bubble Sort has also been referred to a 'Sinking Sort' [36].

Bubble sort is a simple comparison-based sort. As we have previously touched on, comparison-based sorts inspect each element individually with its adjacent element and sorts accordingly [29]. If the element to the right of the inspected element has a value that is less than the inspected element, they will swap positions, otherwise they remain in-place [29]. The idea behind the naming convention is quite ingenious: larger elements in a collection of unordered elements 'bubble-up' to the top at the end of each traversal [10]. This process is repeated until every element in a collection has been sorted.

As you can probably guess from the short description of Bubble Sort, it is not suitable for large data sets due to the vast number of comparisons that would be required [30]. This algorithm does however work well when you only need to sort a small number of elements.

Performance-wise, Bubble Sort is undeniably lacking when it comes to large data sets:

In the **best case,** Bubble Sort works to O(n). This is a case where the array has been input already sorted [29].

In the **worst and average case,** Bubble Sort works to $O(n^2)$. This is a case in where the array is in reverse order so a maximum number of swaps are required [35].

The **space complexity** for Bubble Sort is O(1), as only a small amount of additional memory is required in order the store a temporary variable to hold value before sorting [35]. We have already touched on how Bubble Sort is an 'in-place' sorting algorithm [21].

Bubble Sort is a **stable** sorting algorithm, in where the order of duplicate elements is preserved [24]. This has obvious advantages for satellite data.

The *pseudocode* for Bubble Sort could be as follows:

- An array is passed in that we want to sort in ascending order.
- We inspect the first element at i=0 and its adjacent element.
- If the first element is greater than its adjacent element, we swap them. If not, they remain as is.
- We then inspect element at index i=1 and element at i=2 and swap if element at i=2 is less than element at i=1.
- This process repeats until the final index, as the element at this index is already sorted from the analysis of the second last and final element.
- We now return to the element at i=0 and repeat this process iteratively until every element has been completely sorted and no more swapping is required.

A python example of Bubble Sort could be as follows, adapted slightly with comments from reference 37:

```
def bubbleSort(arr):
    n = len(arr) # length of array
    for i in range(n-1):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]: # if j is greater than j+1
                arr[j], arr[j+1] = arr[j+1], arr[j] # swap them
```

We can display Bubble Sort graphically as follows:

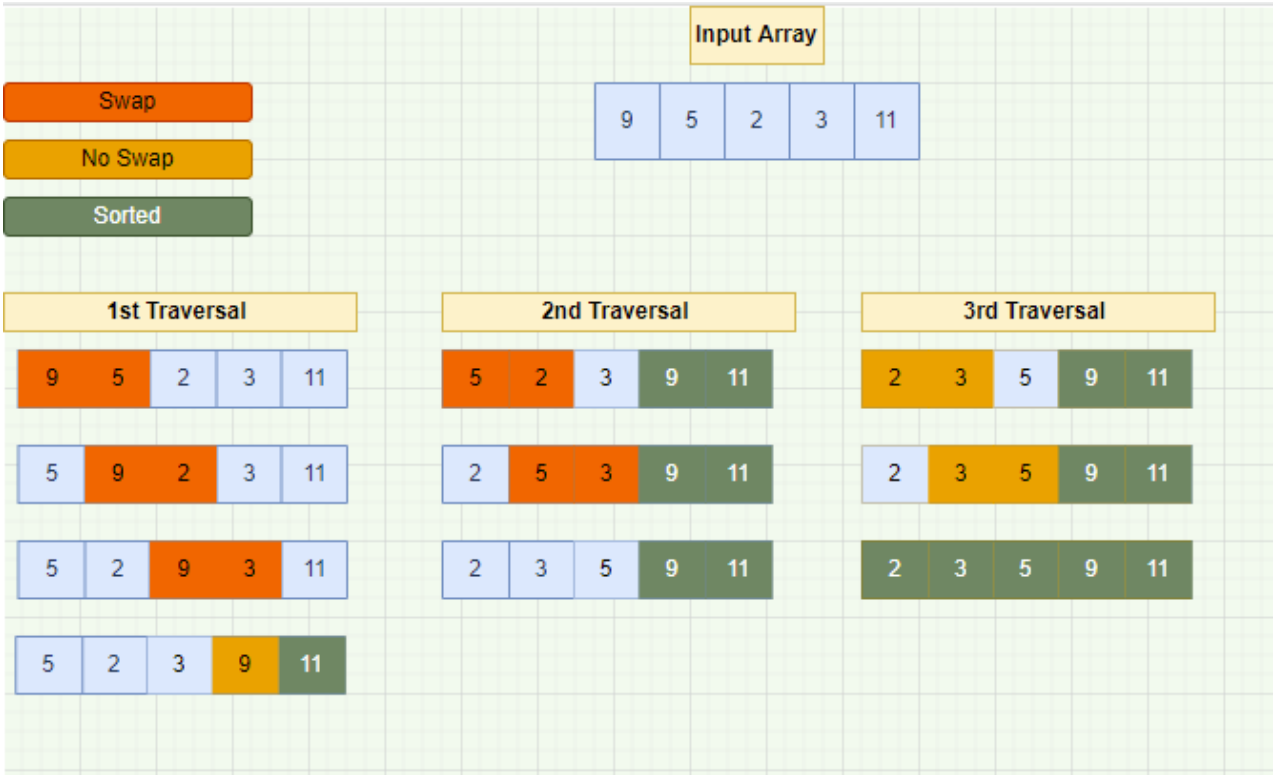

Figure 3 Bubble Sort with input of 5 elements

In Figure 3, we have a simple enough example with 5 integer elements. The process flow is quite simple, we are comparing each pair of elements and determining of they need to be swapped. If the element to the left is greater than the element to the right, they are swapped. Otherwise, they remain in-place. We can already see the limited performance for Bubble Sort as we have required 3 traversals of the collection even though we only have 5 elements.

To begin, we compared the first two elements, 9 and 5. As 9 is greater than 5, they swap indexes. We now compare the element at index i=1 and element i=2, the newly placed 9 and 2. As 9 is greater than 2, these elements swap position. We now move on to 9 and 3. As 9 is greater than 3, they also swap. We finally are at the final 2 elements, 9 and 11. As 9 is not greater than 11, these elements can remain in place. We now iterate back to the beginning and repeat this process again. 5 is greater than 2 so they swap positions and we continue through the collection. This process is continued until we have finally sorted the whole collection with [2,3,5,9,11]. We have completed numerous swaps and a high quantity of comparisons for a collection of only 5 elements. We have already discussed how Bubble Sort has a worst/average case of $O(n^2)$ [35].

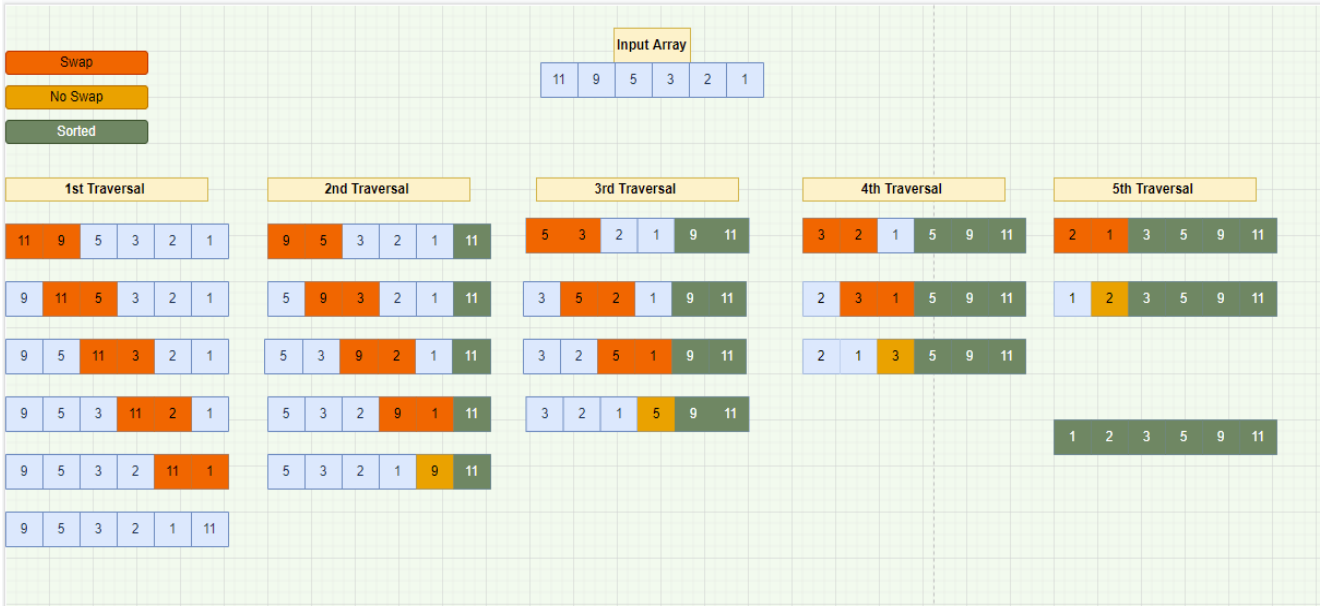

Figure 4 Bubble Sort with a Reversed Array

In Figure 4, we have the proof we need to display that Bubble Sort is quite a slow sorting algorithm. We have only 6 elements in our input however they are in reverse order. We can see how 5 traversals are required and this is an example of Bubble Sort running at a time complexity of $O(n^2)$. We have required almost 20 individual comparisons to sort this short collection of elements.

Quick Sort

Quick Sort is an efficient, widely used, comparison-based sort developed by British scientist C.A.R Hoare in 1959 [27,38]. Quick sort is known as a 'divide-and-conquer' algorithm, in where a 'pivot' is selected from the input array and the array is sub-divided on the basis of being less than or greater than this pivot value [39].

Some versions of Quicksort pick the first element as the pivot element, some pick the final element, some pick the median element and finally some pick any random element [40]. This idea is known as 'partitioning the data' [40]. The sub-arrays that have now been constructed are then sorted recursively (calling upon itself) and in-place, therefore requiring a small amount of additional memory, as we have already detailed [21].

Performance-wise, Quick Sort is quite efficient for larger data sets.

In the **best case,** Quick Sort works to $O(n\ logn)$. This is a case where the partition process selects the middle element to act as the pivot [40].

In the **average case,** Quick Sort works to $O(n\ logn)$. This is a case in where the pivot element is chosen at random [40].

In the **worst-case**, Quick Sort works to $O(n^2)$. This is a case where the partition always picks the greatest or the smallest element as the pivot [40].

The **space complexity** for Quick Sort is O(n), although variants exist which have a space complexity of O(*n logn)* [42].

Quick Sort is **not a stable** sorting algorithm as the sorting is performed according to the pivot element and therefore the original locations of the elements is not considered [24,41].

The *pseudocode* for Quick Sort could be as follows, adapted slightly from reference 39:

- Base case (as we have recursion) is that there is only 1 element, or no element.
- Pick your pivot element
- Re-arrange elements smaller than pivot to left of pivot.
- Re-arrange elements greater than pivot element to the right of the pivot.
- Pivot is now in its final position
- Use recursion to repeat these steps, picking a new pivot element, until the array is sorted.

This process can be displayed algorithmically as follows, adapted from reference 41:

```
def QuickSort(arr):
    if len(arr) < 2: #  Base Case
        return arr
    less, equal, higher = [], [], []   # Define empty arrays to hold our elements
    # Select our pivot element and store this in a pivot variable
    pivot = arr[random.randint(0, len(arr) - 1)]
    # Traverse through each element in our array
    for element in arr:
        # Comparison of our element with the pivot element.
        # We will append our element to a previously defined list depending on its value.
        # Elements that have a lower value to our pivot element will be appended to the 'less' list
        # Elements that have an equal value to our pivot element will be appended to the 'equal' list
        # Elements that have a higher value to our pivot element will be appended to the 'higher' list
        if element < pivot:
            less.append(element)
```

```
        elif element == pivot:
            equal.append(element)
        elif element > pivot:
            higher.append(element)
    # Let's now 'append' our three lists for the output
    return QuickSort(less) + equal + QuickSort(higher)
```

Quick Sort can be displayed graphically as follows:

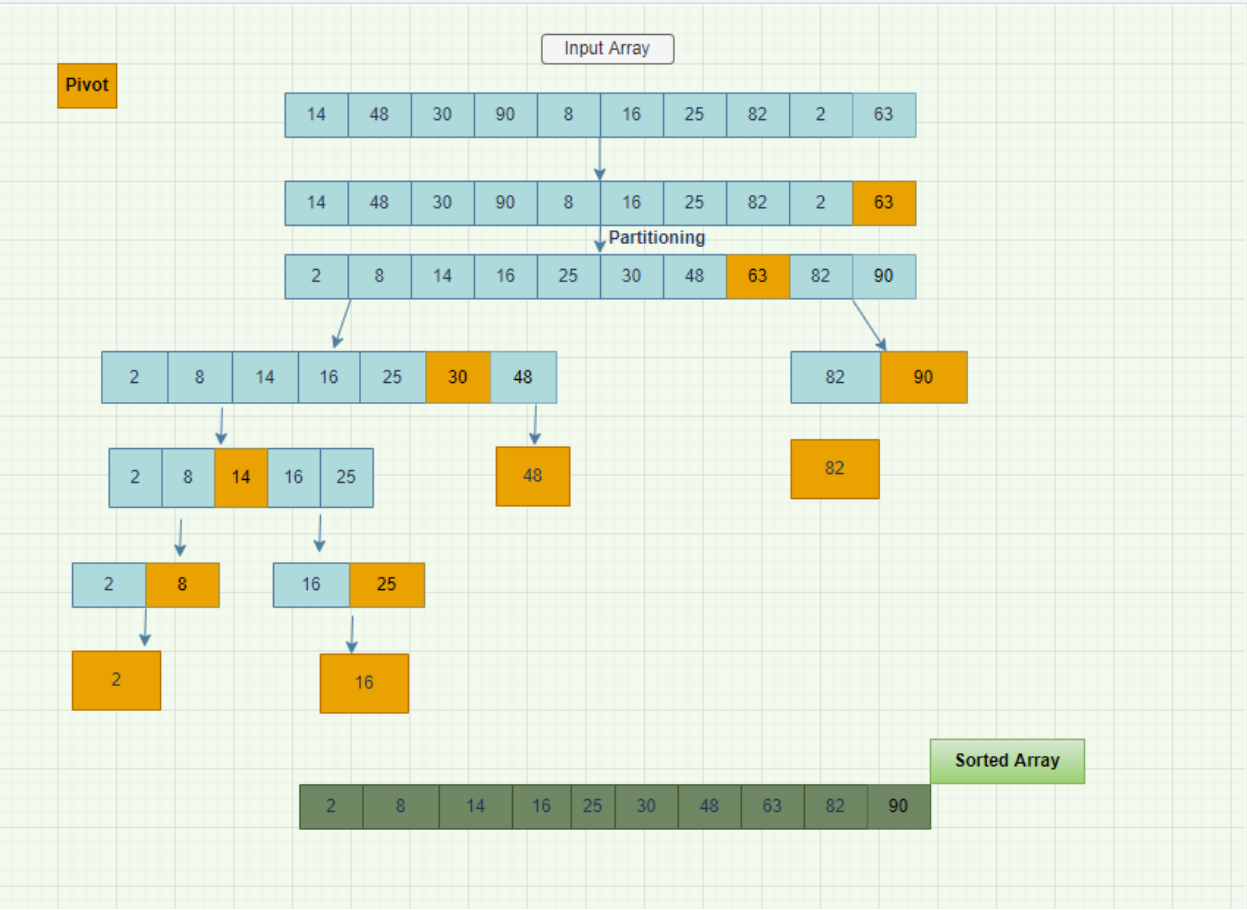

Figure 5 Overview of Quick Sort

In Figure 5, we have a simple overview of Quick Sort. We can see the recursive calls at each step from this collection of 10 elements. We started by picking the last element as the pivot point and through partitioning, moved all elements less than the pivot point to the left and all elements greater than the pivot point to the right. We then selected a new pivot point each time and repeated this same recursive call

until all elements were sorted. We demonstrated various options in regards to selecting pivot points: picking the last element, picking a random element and picking the median element.

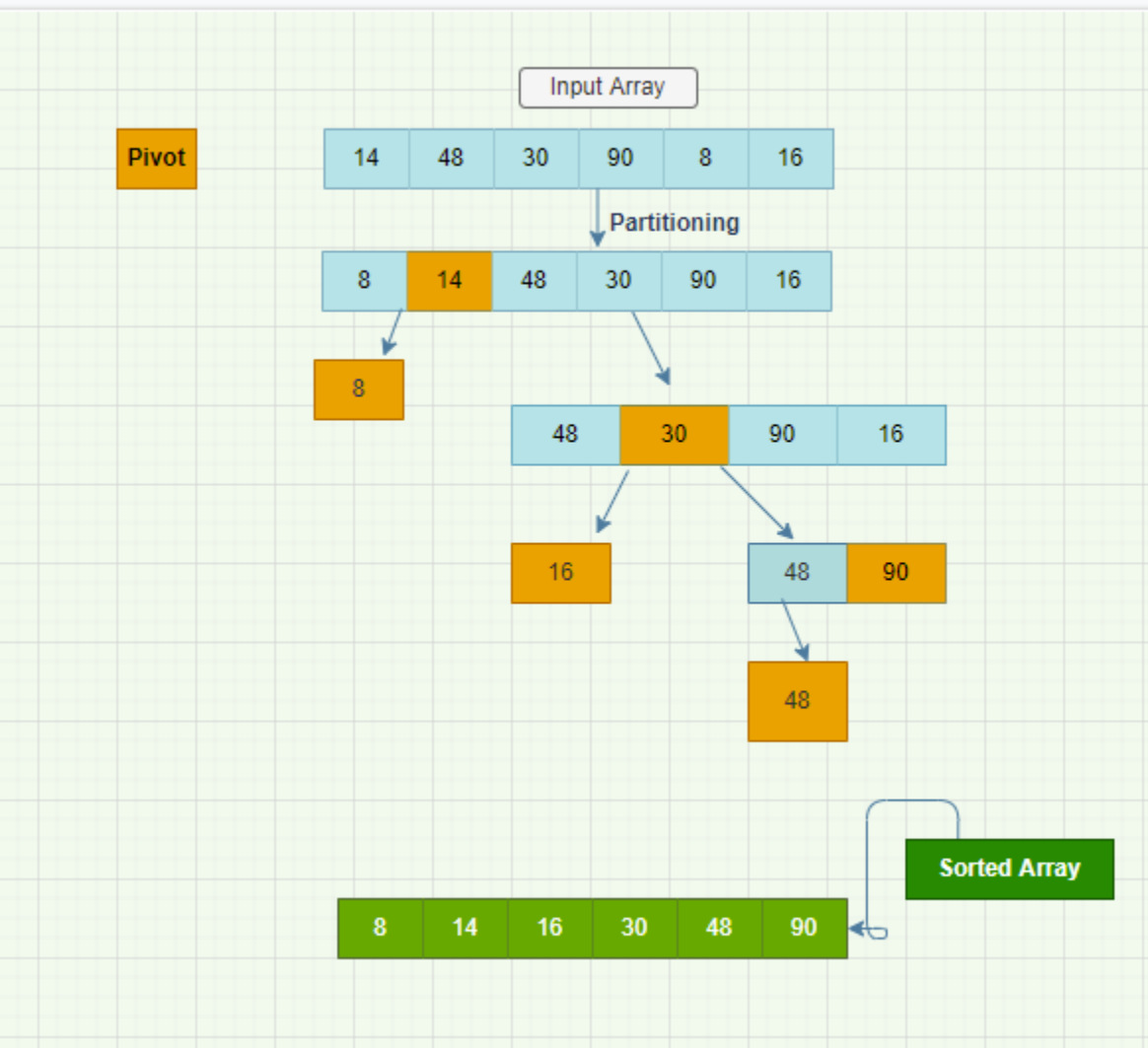

Figure 6 Simple Example of Quick Sort

In this example, we have displayed how Quick Sort can work quite rapidly with lower input instances. The same principles apply here as in the previous example. We pick the first element as our initial pivot (14).

Counting Sort

Counting sort is a non-comparison, integer sorting algorithm developed by Harold H. Seward in 1954 [43]. Objects in this sorting algorithm are sorted through counting of objects with a specific key value [44]. A count is performed of the occurrence of each unique object which then stored in a newly initialised array designed to hold these occurrence values [42]. These count values are then used to index each element in a newly initialised array [42].

There are several important assumptions to be made when incorporating the use of Counting Sort: it will assume each value will be a non-negative integer and will also assume the range of these values is known in advance [45]. These assumptions have several consequences for its use and performance: it can run in near Linear Time complexity (O(n)) however space complexity is quite costly if the range of potential values is quite wide [45]. Counting Sort is often called as part of Radix Sort, which is outside the scope of this current report [45].

Performance-wise, Counting Sort is efficient however it is quite inflexible with regard to input data types.

In the **best/worst/average case**, Counting Sort works to O(n+k) Linear time. This best case is a scenario where each of the elements are of the same range [46]. The worst case would be a situation where the data is skewed and the input range is very wide [46].

The **space complexity** for Counting Sort is O(n+k), the space complexity is quite poor if we have a diverse range of input data [46].

Counting Sort is a **stable** sorting algorithm which preserves the order of elements that are of similar value [42].

Counting Sort is a **not-in-place** sorting algorithm, as we have discussed the creation of input and output arrays and we also need to initialise a count in order to tally the occurrences of distinct key values [42].

The pseudocode for Counting Sort could be as follows, adapted from reference 42 and 47:

- Take in an input array that we desire to be sorted
- Construct a new array which will have a length of n +1, where *n* is the length if the input. Initialise each element to 0 in this new count array.
- Iterate through the input and tally each occurrence in our newly generated array at the corresponding index.
- Initialise a second new result array, which will store our sorted output.
- Based on the frequencies of distinct keys stored in the count array, populated the sorted array.

Counting Sort could be displayed algorithmically as follows, adapted from reference 48:

```
def countingSort(arr):

    n = len(arr)

    output = [0] * n

    count = [0] * 10     # initialise our count array
```

```
    for j in range(0, size): # we will now store our tallies of each element in input
        count[arr[j]] += 1
    for j in range(1, 10): # storing the sum count
        count[m] += count[j - 1]
    j = size – 1     # populate output array after finding the index of each element of original array in count array
    while j >= 0:
        output[count[arr[j]] - 1] = arr[j]
        count[arr[j]] -= 1
        j -= 1
    for m in range(0, size):
        arr[m] = output[m]
```

The following is a graphical representation of Counting Sort:

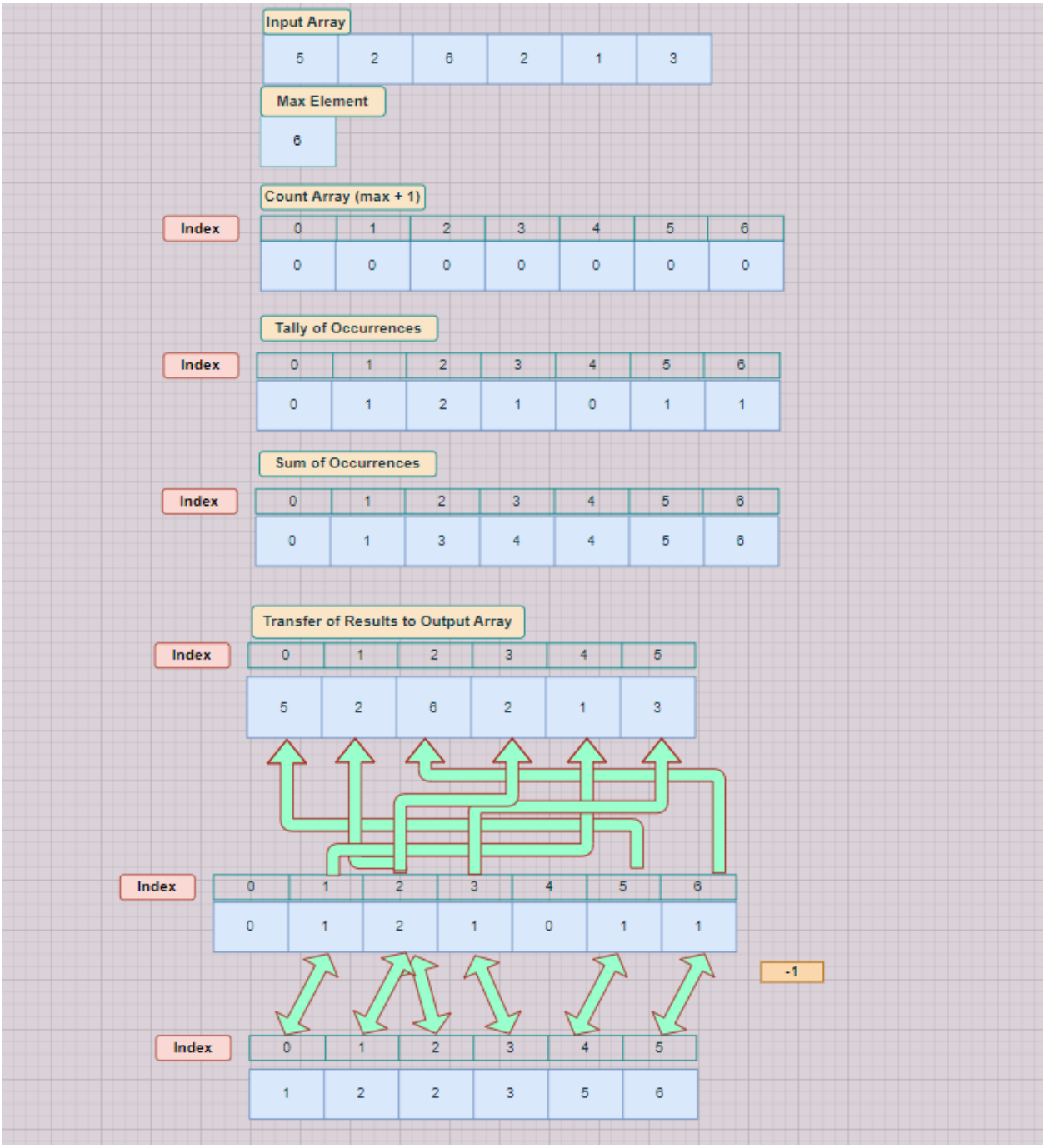

Figure 7 Overview of Counting Sort

In Figure 7, we have the basic overview of the Counting Sort algorithm. We have an input array of 6 elements ranging from 1-6, with 6 being the max element. As such, we initialise our count array with a length of 'max + 1'. We take a tally of the occurrences of each key value and then we will modify the previously defined count array with a cumulative sum of the appearance of each key value. We initialise a result array which will store the sorted resulting output of the Counting Sort algorithm. We transfer the

results of our cumulative count array to the resulting sorted output array, decrementing the count each time.

Selection Sort

Selection Sort is a simple comparison-based sorting algorithm, and one of the easiest to implement [49]. A basic example you could use in everyday life is playing cards: if all of the cards were placed in front of you, face up, and you iteratively placed the card with the lowest value to the left. Eventually there would be no more cards to sort and you would have a sorted deck of cards. When we visualise this, we would notice we have two different sub-arrays here: one with sorted cards and the other sub-array which remains unsorted until the end.

The concept is quite simple: an input array is sorted by iteratively selecting the minimum element in an unsorted array and moving this element to the left until there are no elements remaining unsorted [49]. We can see that the process would be more efficient than Bubble Sort, however still quite inefficient overall as we are inspecting each element in the unsorted array at each iteration [49].

Performance-wise, Selection Sort is efficient however it is quite inflexible with large data sets, similar to Bubble Sort.

In the **best/worst/average case**, Selection Sort works to $O(n^2)$. This is due to the fact there are two nested loops, as we have already discussed in our example with the playing cards [50].

The **space complexity** for Selection Sort is O(1). This is because no new arrays are initiated, with all operations carried out in the original array [50].

Selection Sort is an **unstable** sorting algorithm as the order of elements are not preserved [51]. While Selection Sort may appear stable, when elements are transferred to the sorted sub-array the order may not be kept.

Selection Sort is an **in-place** sorting algorithm.

The pseudocode for Selection Sort could be as follows, adapted from references 49-50:

- Take in an input array
- Search through each element in the array and select the small element. Place this element at index 0 and swap the element at position zero with the position of the minimum element.
- Search through each element of the array beginning at index 1 (as we have already sorted the element at index 0). Find the minimum element in the array and place it at index 1.
- Swap the element at index 1 with the element that was the minimum.
- Repeat this process until the whole array is sorted.

Selection Sort could be displayed algorithmically as follows, adapted from reference 52:

```
for i in range(len(arr)): # arr is our input array
   min = i # set the min variable to i to initialise
    for j in range(i+1, len(arr)):
        if arr[min] > arr[j]: # if inspected variable is less than the min element currently
            min = j
     arr[i], arr[min] = arr[min], arr[i] # swap the elements
```

Selection Sort can be displayed graphically as follows:

Figure 8 Overview of Selection Sort

As we can see from Figure 8, the process flow of Selection Sort is quite straight-forward. We have an input array of 6 elements. At each pass, setting 'i' equals to the pass number, we are searching iteratively for the minimum element. So in the first pass, we see that the minimum element is 3 so we place this element at index 0 and swap the element at index 0 to the position '3' was occupying. The element at index 0 is now considered sorted. We repeat this process, starting at index 1 and searching the array for

the minimum element. Now, it is 8, so we swap the value 8 to index 1 and move the value 55 (which was present at index 1 in our array) to index 3. We repeat this process until our array is fully sorted.

Insertion Sort

Insertion sort is another simple comparison-based sorting algorithm, first discussed by John Mauchly in 1946 in the first discussion on computer program sorting [43].

Similar to our example for Selection Sort in where we discussed a deck of cards, we can use the same example to explain Insertion Sort to a high-level. Whereas in Selection Sort we 'selected' the minimum card from the whole deck and moved it away from the unsorted cards, in Insertion Sort we can move each card individually by comparing it with the already-sorted elements [53].

We make the assumption the first element is already sorted and start our comparison with the element in index 1. We compare this element to the element at index 0 and move it to the left if it is less than the element at index 0. We then move (iterate) to index 2 and make the same comparisons with the previous two elements and move it accordingly.

It can surely be deduced by now that this sorting algorithm would not work very efficiently for large data sets, due to the number of operations required [53]. This sorting algorithm does work more efficiently than Bubble Sort and Selection Sort [54].

Performance-wise, this sorting algorithm has similar limitations than other simple comparison-based sorting algorithms.

In the **best case**, Insertion Sort can work to O(N), in where the input array is already sorted [55]. Naturally, this wouldn't happen very often in practice.

In the **worst/average** case, Insertion Sort works to $O(n^2)$. This would be a case in where the input array is in reverse order so the maximum number of swaps are required [55].

The **space complexity** for Insertion Sort is O(1). This is because no new arrays are initiated, with the creation of sub-lists [49,55].

Insertion Sort is **stable** sorting algorithm where the order of elements are preserved [55].

Insertion Sort is an **in-place** sorting algorithm [55].

The *pseudocode* for Insertion Sort could be as follows, adapted from references 49 and 55:

- Take in an input array
- Starting with i=1 as the key, iterate through the array, determining if the elements predecessor is greater than the current element.
- If it is, swap them. Then compare this newly-swapped element to its predecessor, and make the same deduction. If not, make no swap.
- Repeat this step for the length of the array.

Insertion Sort could be described algorithmically as follows, again adapted form reference 55:

```
def insertionSort(arr):
    for i in range(1, len(arr)): # starting at index=1 up to end of input array
        key = arr[i] # set our key value to the first element initially
        j = i-1
        while j >= 0 and key < arr[j] :
            arr[j + 1] = arr[j] # swap the keys
            j -= 1
        arr[j + 1] = key # new key is set to new element
```

Insertion Sort can be displayed graphically as follows:

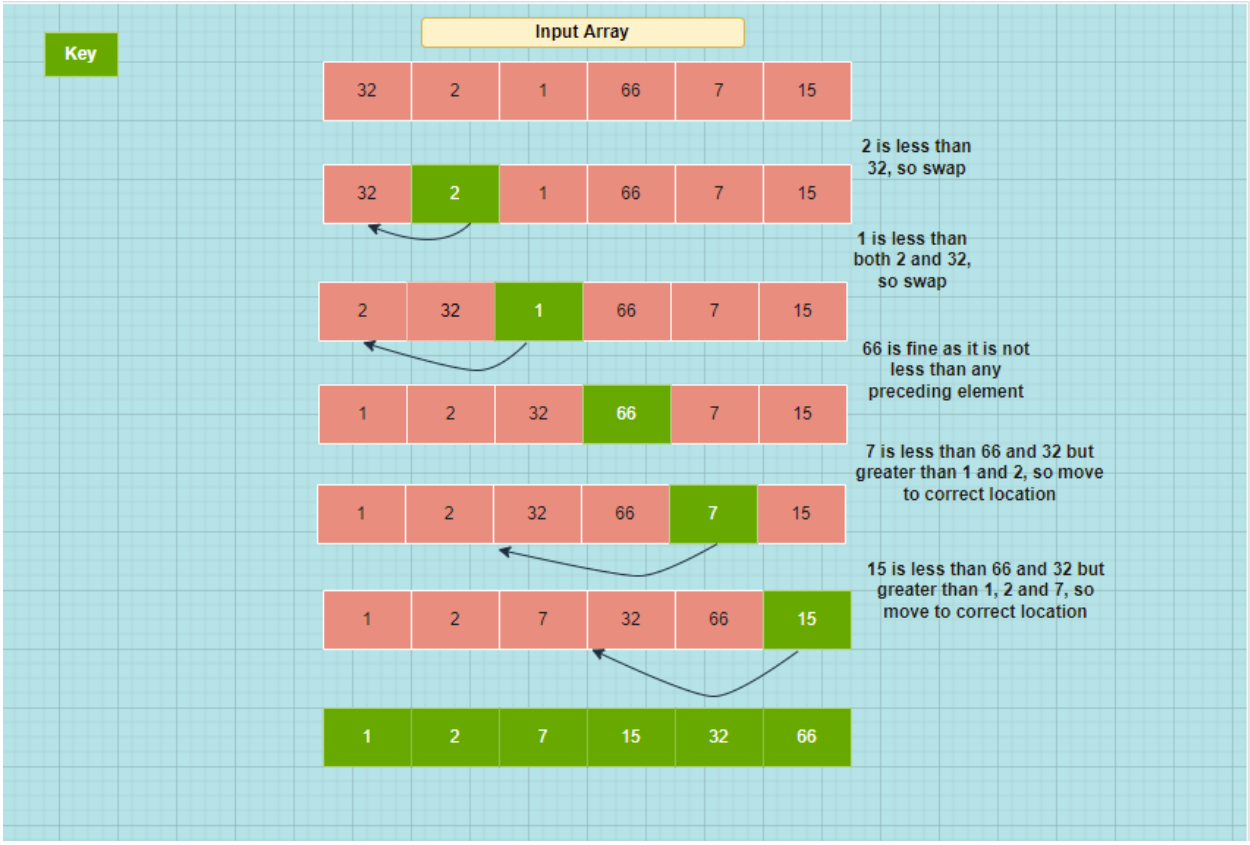

Figure 9 Overview of Insertion Sort

In Figure 9, we have the basic overview of Insertion Sort. We have an input array of 6 elements in random order. We select 2 as our initial key value, assuming the first element to be already sorted. We compare our key value to its predecessors and move position accordingly. This is repeated until the whole

array in sorted. In some cases, no swap at all is required. This is seen when 66 is the key value. In this case, we just keep iterating.

Implementation and Benchmarking

This section describes the implementation of each of our chosen sorting algorithms and the benchmarking results when running the algorithms in Visual Studio Code.

Implementation of Bubble Sort

```python
# A simple comparison-based sort: Bubble Sort
# References:
# https://runestone.academy/ns/books/published/pythonds/SortSearch/TheBubbleSort.html (adapted)
# https://www.geeksforgeeks.org/python-program-for-bubble-sort/  (adapted)
# https://www.guru99.com/bubble-sort.html (adapted)
# https://learnonline.gmit.ie/pluginfile.php/579167/mod_resource/content/0/Sorting.py (adapted)

# take in an input array and run BubbleSort function
def bubbleSort(arr):
    # Looping from size of array from final index to first index, decreasing each iteration by 1
    for outer in range(len(arr)-1,0,-1):
        # iterate through the array
        for inner in range(outer):
            # conditional to determine if first element is greater than second element
            if arr[inner]>arr[inner+1]:
                # define temp variable to hold our original arr[inner] element
                temp = arr[inner]
                # swap our two elements as arr[inner] is greater than arr[inner+1]
                arr[inner] = arr[inner+1]
                # complete the swap.
                arr[inner+1] = temp
```

Figure 10 Bubble Sort from VS Code

Figure 10 represents the implementation of Bubble Code. As referenced within the code, my implementation was adapted from several stated sources.

We first define our function and set it to take in an input array (arr variable). The outer loop will loop from the final index to the first index for the required amount of iterations, decreasing by 1 on each iteration. The next nested inner loop iterates through the array and progresses to a conditional ‘if’ statement which asks: “is the first element greater than the second element?”. If the answer is ‘yes’, then this value will be transferred to a ‘temp’ variable while are two inspected elements are swapped.

Implementation of QuickSort

```python
# An effeicient comparison-based sort: QuickSort
# References:
# https://runestone.academy/ns/books/published/pythonds/SortSearch/TheQuickSort.html (adapted)
# https://realpython.com/sorting-algorithms-python/#the-quicksort-algorithm-in-python (adapted)
# https://www.geeksforgeeks.org/quick-sort/ (adapted)


# Take in an input array and run QuickSort function
def QuickSort(arr):
    # This is a recursive function, so if the length of the array is less than two,
    # then just return it as the result of the function.
    # This is considered the 'Base Case'
    if len(arr) < 2:
        return arr
    # Define empty arrays to hold our elements
    less, equal, higher = [], [], []

    # Select our pivot element and store this in a pivot variable
    pivot = arr[random.randint(0, len(arr) - 1)]

    # Traverse through each element in our array
    for element in arr:
        # Comparison of our element with the pivot element.
        # We will append our element to a previously defined list depending on it's value.
        # Elements that have a lower value to our pivot element will be appended to the 'less' list
        # Elements that have an equal value to our pivot element will be appended to the 'equal' list
        # Elements that have a higher value to our pivot element will be appended to the 'higher' list
        if element < pivot:
            less.append(element)
        elif element == pivot:
            equal.append(element)

        elif element > pivot:
            higher.append(element)

    # Let's now 'append' our three lists for the output
    return QuickSort(less) + equal + QuickSort(higher)
```

Figure 11 Overview of QuickSort from VS Code

Figure 11 displays my implementation of QuickSort, again adapted from several stated references at the beginning of the code. We begin by defining the function and again accepting 1 parameter as input. We then define our Base Case (which we have already covered the theory of) which is if the array is empty or only contains one element, then return the array as it can be considered sorted already. We then define three empty variables which will hold values based on their relationship to the pivot variable. This pivot variable is then defined next, picking a random element. We then initialise a for loop which will traverse each element in the input array. A conditional ‘if’ statement is initialised which asks if the element is less-than, equal or greater-than the pivot element. The inspected element is then assorted into either of our three previously defined variables based on the answer to this conditional statement. Finally, our three defined variables are appended together for the final sorted output.

Implementation of Counting Sort

```python
# A non-comparison sort: CountingSort
# References:
# https://www.geeksforgeeks.org/counting-sort/ (adapted)
# https://python.plainenglish.io/sorting-algorithms-explained-using-python-counting-sort-32b87a0f3011 (adapted)

# Take in an input array and run CountingSort function
def count_sort(arr):
    # let's find the max and min values of our input array
    max_a = int(max(arr))
    min_a = int(min(arr))

    # Intialise an array count which will be used to store
    # the count of number of times each value appears in input
    # initialise it's elements to 0 to begin with.
    count_arr = [0 for i in range(max_a - min_a +1)]
    # We will store our output here
    output_arr = [0 for i in range(len(arr))]

    # Store count of each character after its occurrence
    for j in range(0, len(arr)):
        count_arr[arr[j]-min_a] += 1

    # Count of sum of occurrences
    for j in range(1, len(count_arr)):
        count_arr[j] += count_arr[j-1]

    # Construction of defined output array
    for j in range(len(arr)-1, -1, -1):
        output_arr[count_arr[arr[j] - min_a] - 1] = arr[j]
        count_arr[arr[j] - min_a] -= 1

    # Simple copy of our output array defined earlier into our array so that it
    # now contains our sorted elements
    for i in range(0, len(arr)):
        arr[i] = output_arr[i]

    return arr
```

Figure 12 Overview of Counting Sort from VS Code

In Figure 12, we have the implementation of Counting Sort, adapted from the references stated at the beginning of the code. We first define our function, accepting 1 parameter as input. We define two variables which will respectively hold the maximum and minimum values of our input. We then initialise a count array which will store the occurrences of each key and also an output array which will later store our sorted output. We initialise a for loop which will store the count of each character/value after each appearance and then tally the sum of occurrences. Based on the frequencies of distinct keys stored in the count array, we then construct and populate the output array.

Implementation of Selection Sort

```
# References:
# https://runestone.academy/ns/books/published/pythonds/SortSearch/TheSelectionSort.html (adapted)
# https://learnonline.gmit.ie/pluginfile.php/579165/mod_resource/content/0/08%20Sorting%20Algorithms%20Part%202.pdf

# Take in an input array and run SelectionSort function
def selection_sort(arr):
    n = len(arr)
    # for loop for range of length of input array, incrementing 1 each time
    for outer in range(0,n,1):
        # set a default min value to do the comparisons
        min = outer

        # for loop in range of all elements to right of where we currently are
        for inner in range(outer+1,n,1):
            # if selected element is greater than the min value
            if arr[inner] < arr[min]:
                # assign arr[inner] as the new min value
                min = inner;
        # define temp variable to hold arr[outer] element
        temp = arr[outer]
        # swap our two elements
        arr[outer] = arr[min]
        # complete the swap
        arr[min] = temp
```

Figure 13 Overview of Selection Sort from VS Code

Figure 13 displays the implementation of Selection Sort, with code adapted from the stated references in code. We define our array, accepting 1 parameter as input. We define a variable ‘n’, which will be the length of the input. We initialise an ‘outer’ for loop for the range of our input and set a default minimum value to begin with. We initialise an ‘inner’ for loop and use an ‘if’ conditional statement to ask: ‘is the inspected element less than our currently defined minimum value?’. If the answer is yes, we assign our inspected value as the new minimum value and begin the process of swapping them, using a temp variable similar to as described to with Bubble Sort.

Implementation of Insertion Sort

```
# Any other sorting algorithm: Insertion Sort
# References: https://runestone.academy/ns/books/published/pythonds/SortSearch/TheInsertionSort.html (adapted)
# https://learnonline.gmit.ie/pluginfile.php/579165/mod_resource/content/0/08%20Sorting%20Algorithms%20Part%202.pdf

# Take in input array and run InsertionSort function
def insertionSort(arr):

    # for loop in range of index  1 to to length of array
    # the element at index 0 is initially placed in 'sorted' list as no item to it's left to compare with
    for i in range(1,len(arr)):
    # set the 'key' to the element at current index
        currentsortvalue = arr[i]
        position = i

    # if value to left is higher than the value we are currently trying to sort
    # ('position > 0' section is there to prevent negative indexing)
        while position>0 and arr[position-1]>currentsortvalue:
            # then swap them
            arr[position]=arr[position-1]
            position = position-1

        arr[position]=currentsortvalue
```

Figure 14 Overview of Insertion Sort from VS Code

Figure 14 displays the implementation of Insertion Sort, adapted from the stated references in the code. We begin by defining the function, which can accept 1 parameter as input. We initialise a for loop beginning at the second element (i=1, remember we ignore the first element) for the length of the input. We define a variable called ‘currentsortvalue’ which will act as the key. We initialise a while loop that prevents negative indexing and also only works while the position to the left of the key is less than the key value. The key value is then placed in its correct location in relation to the sorted output and the while loop iterates again. This process is repeated until every element has been sorted.

Benchmarking

This section describes the process followed in benchmarking each of our algorithms. We initially defined different array sizes, based on the example given in the project description (Figure 15).

```
# define array sizes as per CTA Project file example
arraylength = (100,250,500,750,1000,1250,2500,3750,5000,6250,7500,8750,10000)
```

Figure 15 Defined array lengths as per project description

The next step was to benchmark our sorting algorithms.

```
#Define running_time function to calculate running times for each sorting algorithm
def running_times(algo):
    # define empty list to be populated with average running times
    run = []
    #for loop for length of input array
    for sz in arraylength:
        lengthofRun = timerRun(algo, sz)
        run.append(lengthofRun)

    return(run)

# following code will run each sorting algorithm using above running_time function
bubbleSort_RunTime = running_times(bubbleSort)
quickSort_RunTime = running_times(QuickSort)
countingSort_RunTime = running_times(count_sort)
selectionSort_RunTime = running_times(selection_sort)
insertionSort_RunTime = running_times(insertionSort)
```

Figure 16 Overview of benchmarking each sorting algorithm

We defined a program called ‘running_times’ and accepted a parameter as input. We defined an empty list called ‘run’ which will be used downstream in order to store our running times in milliseconds. We then construct a for loop which will run for each instance of array length we have previously defined (Figure 15). This for loop calls our defined ‘timerRun’ function which executes our sorting algorithms with the defined input array length. The empty ‘run’ variable is then populated with average running times.

```python
#Run the timerRun to get the average run time with the defined array size
def timerRun(algo, sz):

    # empty array to contain generated arrays from below loop
    array = []

    # For loop to run 10 times, one for each instance as we need 10 runs as per project decscription
    for arr in range(10):
        #generate random arrays populated with integers ranging from 1-99 for the given array size
        randintArray =[random.randint(0, 100) for i in range(sz)]
        # store the vlaues of random array into array which will be used later on
        array.append(randintArray)

    #start timer as per project file
    start_time = time.time()

    # For loop to initiate sorting algorithm for each stored array in array variable
    for randintArray in array:
        algo(randintArray)

    #stop the timer as per project file
    end_time = time.time()

    #convert our running times to milliseconds and divide by 10 to get the average (each algo ran 10 times)
    time_taken = ((end_time - start_time)/10)*1000
    #return running time to 3 decimal places
    return round(time_taken,3)
```

Figure 17 Overview of timerRun function

We defined the timerRun function to benchmark our algorithms. The function accepts 2 input parameters, the array length to test and also the algorithm. This function is based heavily upon the example given in the project description in where we generate an array of random integers up to 99 and store these values in a previously defined empty 'array' variable. We then being our timer program (based on project description) which will record start and end times for the execution of our algorithm. Finally, we perform some basic arithmetic in order to convert our times to milliseconds and return the final result rounded to 3 decimal places as requested.

Results of Benchmarking

The following figure displays, in tabulated format, the results in milliseconds of running our sorting algorithms 10 times with arrays of differing sizes.

	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
Bubble Sort	0.998	8.168	34.924	74.425	130.485	213.594	881.741	2052.145	3289.087	5310.245	7806.493	10370.837	13940.853
Quick Sort	0.203	0.399	0.814	1.013	1.311	1.726	4.186	5.461	10.595	11.555	9.312	10.744	12.474
Counting Sort	0.100	0.199	0.299	0.499	0.701	0.798	3.015	4.137	4.320	4.152	5.067	5.872	7.097
Selection Sort	0.514	2.736	13.245	27.499	49.044	89.954	320.851	797.621	1412.299	2137.212	3120.978	4256.484	5567.375
Insertion Sort	0.613	6.093	18.739	43.384	82.762	126.970	530.379	1223.901	2164.644	3359.541	4813.880	6667.733	8827.596

Figure 18 Benchmarking Results

Let's refresh ourselves with the expected average and worst-case time complexities for each of our chosen algorithms and then compare with our results, as displayed in Figure 25. I have chosen to not display best case as since we have 10000 inputs in one instance, we are rarely going to be achieving best-

case scenario. Also, we have seen in the literature, as described above, that the best-case time complexity is of limited information value.

Sorting Algorithm	Average Case	Worst Case
Bubble Sort	$O(n^2)$	$O(n^2)$
Quick Sort	$O(n\ logn)$	$O(n^2)$
Counting Sort	O(n+k)	O(n+k)
Selection Sort	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n^2)$	$O(n^2)$

Table 3 Summary of Time Complexities

It should probably be noted that my laptop is a Lenovo X260 with an Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz 2.40 GHz processor. This will obviously have a bearing on overall run times. Looking at the results, it may be time to upgrade.

It appears our results tie in quite similarly to the information detailed in Table 3. We consistently see that Bubble Sort, Selection Sort and Insertion Sort are much slower than either Quick Sort or Counting Sort over the course of array input lengths. Each of these sorting algorithms have an average-case time complexity of $n^{2.}$ Within the three simple comparison-based sorting algorithms, Bubble Sort is much slower. This ties in to what we have seen in the literature [30,49]. Selection Sort is the quickest of the three simple sorting algorithms, with Bubble Sort approaching 3x slower execution time. Insertion Sort sits in the middle of the two simple comparison-based sorting algorithms. It is interesting that there is such a difference in execution times when all three of the sorting algorithms have the same average and worst-case time complexities. I didn't expect to see such vast differences however I was seeing in the literature that there were large differences between the simple comparison-based algorithms so it makes sense.

For Quick Sort and Counting Sort, I was very surprised at just how quick Counting Sort was. There was a big jump in execution time between input array length of 1250 and input length 2500. This does seem to suggest that an increase of elements dramatically increases execution time for this algorithm. The time complexity of Counting Sort is O(n+k) which is faster than Quick Sorts time complexities and our results appear to agree with this. While Quick Sort was obviously much faster than the three simple comparison-based sorting algorithms, it was over 50% slower than Counting Sort. Similar to Counting Sort, there was a vast increase in execution time when the input array length increased from 1250 to 2500. This warrants further investigation.

The following graph displays the running times of our 5 chosen Sorting Algorithms:

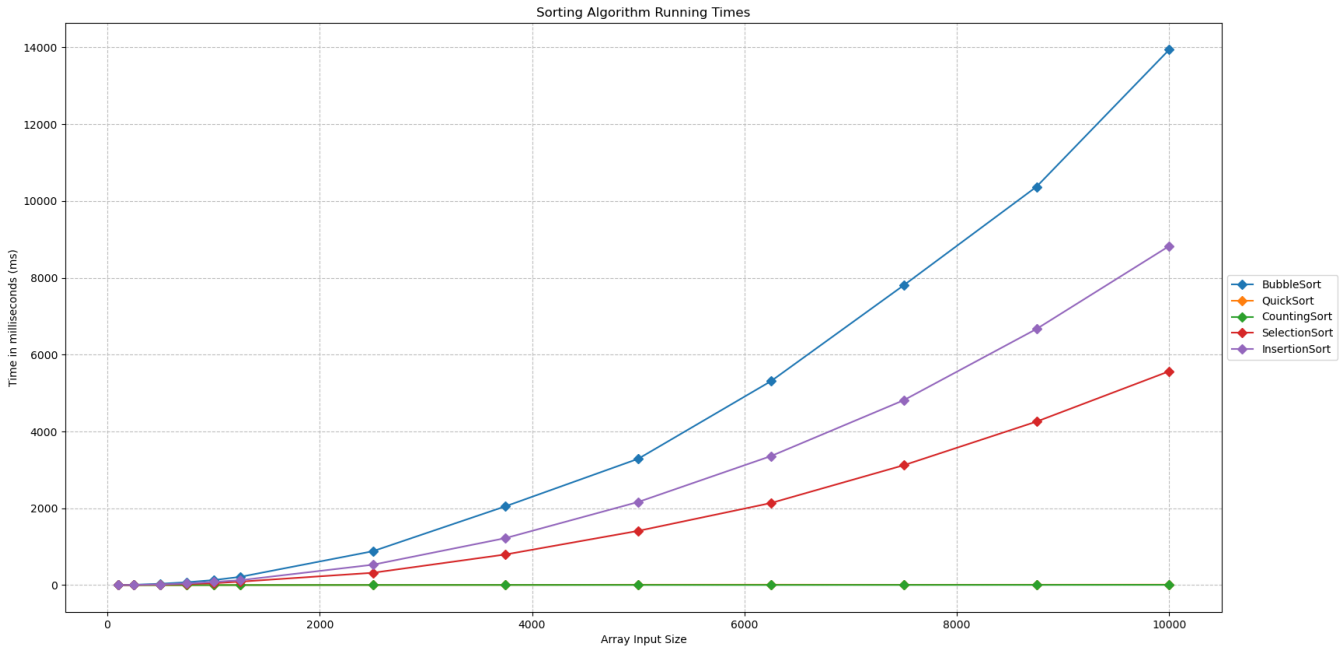

Figure 19 Running Times of each Sorting Algorithm

The graph above (Figure 19) merely confirms the results we have seen in our tabulated data, as well as what we have seen in the literature. We see the 3 simple comparison-based sorting algorithms are order of magnitudes slower than Counting Sort and Quick Sort. Due to the scale of the three simple-comparison based sorting algorithms, Quick Sort and Counting Sort appear the be close to 0.

That concludes my analysis of 5 sorting algorithms and benchmarking.

Thank you.

References:

1. www.cs.cmu.edu. Introduction to Sorting. [online] Available at: https://www.cs.cmu.edu/~clo/www/CMU/DataStructures/Lessons/lesson8_1.htm.
2. Bunse, Christian & Höpfner, Hagen & Mansour, Essam & Roychoudhury, Suman. (2009). Exploring the Energy Consumption of Data Sorting Algorithms in Embedded and Mobile Environments. Proceedings - IEEE International Conference on Mobile Data Management. 600-607. 10.1109/MDM.2009.103.
3. GeeksforGeeks. (2017). Sorting Algorithms - GeeksforGeeks. [online] Available at: https://www.geeksforgeeks.org/sorting-algorithms/.
4. https://learnonline.gmit.ie/pluginfile.php/579161/mod_resource/content/0/07%20Sorting%20Algorithms%20Part%201.pdf
5. Estivill-Castro, Vladimir , "Sorting and Order Statistics" , in Algorithms and Theory of Computation Handbook ed. Mikhail J. Atallah and Marina Blanton (Boca Raton: CRC Press, 20 Nov 2009), accessed 05 May 2022 , Routledge Handbooks Online.
6. Knuth, Donald (1998). "Section 5.2.4: Sorting by Merging". Sorting and Searching. The Art of Computer Programming. Vol. 3 (2nd ed.). Addison-Wesley. pp. 158–168. ISBN 0-201-89685-0.
7. GeeksforGeeks. (2013). Radix Sort. [online] Available at: https://www.geeksforgeeks.org/radix-sort/
8. George Heineman, Gary Pollice, and Stanley Selkow. 2008. Algorithms in a Nutshell. O'Reilly Media, Inc.
9. https://learnonline.gmit.ie/pluginfile.php/579141/mod_resource/content/0/03%20Analysing%20Algorithms%20Part%201.pdf
10. David and Feldman Yishai Harel (2014). Algorithmics: The Spirit of Computing. Berlin: Springer.
11. GeeksforGeeks. (2019). Difference between Posteriori and Priori analysis. [online] Available at: https://www.geeksforgeeks.org/difference-between-posteriori-and-priori-analysis/ [Accessed 5 May 2022].
12. https://learnonline.gmit.ie/pluginfile.php/579141/mod_resource/content/0/03%20Analysing%20Algorithms%20Part%201.pdf
13. Algorithmic Complexity. (2017) [online] Devopedia. Available at: https://devopedia.org/algorithmic-complexity#:~:text=Algorithmiccomplexityisameasure.
14. Alraja, H.A. (2021). Logarithms & Exponents in Complexity Analysis. [online] Medium. Available at: https://towardsdatascience.com/logarithms-exponents-in-complexity-analysis-b8071979e847 [Accessed 6 May 2022].
15. xlinux.nist.gov. (n.d.). sublinear time algorithm. [online] Available at: https://xlinux.nist.gov/dads/HTML/sublinearTimeAlgo.html [Accessed 6 May 2022].
16. www.seas.upenn.edu. (n.d.). CSC 4170 Polynomial-Time Algorithms. [online] Available at: https://www.seas.upenn.edu/~cit596/notes/dave/p-and-np2.html#:~:text=ApolynomialDtimealgorithmis [Accessed 6 May 2022].

17. Stricklin, J. (2019). Computer Science Fundamentals: Big O Notation. [online] Medium. Available at: https://medium.com/@jstricklin89/computer-science-fundamentals-big-o-notation-e5b930ac36f4 [Accessed 6 May 2022].
18. https://learnonline.gmit.ie/pluginfile.php/579144/mod_resource/content/0/04%20Analysing%20Algorithms%20Part%202.pdf
19. Algorithm Tutor. Best, Average and Worst case Analysis of Algorithms. [online] Available at: https://algorithmtutor.com/Analysis-of-Algorithm/Best-Average-Worst-case-analysis-of-Algorithms/.
20. Lovett, P. (2020). Big-O Notation: A Simple Explanation with Examples. [online] Medium. Available at: https://betterprogramming.pub/big-o-notation-a-simple-explanation-with-examples-a56347d1daca.
21. GeeksforGeeks. (2018). In-Place Algorithm - GeeksforGeeks. [online] Available at: https://www.geeksforgeeks.org/in-place-algorithm/.
22. Baeldung (2019). How an In-Place Sorting Algorithm Works. [online] Baeldung. Available at: https://www.baeldung.com/java-in-place-sorting.
23. Technology-assistant.com. Is merge sort in-place or out-of-place? [online] Available at: https://technology-assistant.com/article/is-merge-sort-in-place-or-out-of-place [Accessed 6 May 2022]
24. Srivastava, P. (2020). Stable Sorting Algorithms | Baeldung on Computer Science. [online] www.baeldung.com. Available at: https://www.baeldung.com/cs/stable-sorting-algorithms#:~:text=Stablesortingalgorithmspreservethe [Accessed 6 May 2022].
25. GeeksforGeeks. (2018). Analysis of different sorting techniques - GeeksforGeeks. [online] Available at: https://www.geeksforgeeks.org/analysis-of-different-sorting-techniques/.
26. ilyasergey.net. 5.2. Best-Worst Case for Comparison-Based Sorting — YSC2229 2021. [online] Available at: https://ilyasergey.net/YSC2229/week-04-sorting-best-worst.html [Accessed 6 May 2022].
27. afteracademy.com. Comparison of Sorting Algorithms. [online] Available at: https://afteracademy.com/blog/comparison-of-sorting-algorithms.
28. Adams, D. (2021). Searching and Sorting Algorithms in JavaScript | The Ultimate Guide. [online] www.doabledanny.com. Available at: https://www.doabledanny.com/searching-and-sorting-algorithms-in-javascript#15 [Accessed 6 May 2022].
29. GeeksforGeeks. (2014). Bubble Sort - GeeksforGeeks. [online] Available at: https://www.geeksforgeeks.org/bubble-sort/.
30. Tutorialspoint.com. (2019). Data Structure - Bubble Sort Algorithm - Tutorialspoint. [online] Available at: https://www.tutorialspoint.com/data_structures_algorithms/bubble_sort_algorithm.htm.
31. Edward H. Friend. 1956. Sorting on Electronic Computer Systems. J. ACM 3, 3 (July 1956), 134–168. https://doi.org/10.1145/320831.320833
32. The Craft of Coding. (2021). The origins of Bubble sort. [online] Available at: https://craftofcoding.wordpress.com/2021/09/22/the-origins-of-bubble-sort/ [Accessed 7 May 2022].
33. Iverson, K. E., A Programming Language, John Wiley & Sons (1962)
34. users.cs.duke.edu. (n.d.). Bubble Sort: An Archaeological Algorithmic Analysis. [online] Available at: https://users.cs.duke.edu/~ola/bubble/bubble.html.

35. Studytonight.com. (2019). Bubble Sort Algorithm | Studytonight. [online] Available at: https://www.studytonight.com/data-structures/bubble-sort.
36. Simplilearn.com. (2022). What is Bubble Sort Algorithm? Time Complexity & Pseudocode | Simplilearn. [online] Available at: https://www.simplilearn.com/tutorials/data-structure-tutorial/bubble-sort-algorithm#:~:text=The%20bubble%20sort%20algorithm%20is.
37. Parvez, F. (2020). What is Bubble Sort Algorithm Using C,C++, Java and Python. [online] GreatLearning Blog: Free Resources what Matters to shape your Career! Available at: https://www.mygreatlearning.com/blog/bubble-sort/#sh11.
38. web.archive.org. (2015). Sir Antony Hoare | Computer History Museum. [online] Available at: https://web.archive.org/web/20150403184558/http://www.computerhistory.org/fellowawards/hall/bios/Antony%2CHoare/
39. Tutorialspoint.com. (2020). Data Structure and Algorithms - Quick Sort - Tutorialspoint. [online] Available at: https://www.tutorialspoint.com/data_structures_algorithms/quick_sort_algorithm.htm.
40. GeeksforGeeks (2014). QuickSort - GeeksforGeeks. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/quick-sort/.
41. InterviewBit (n.d.). Quicksort Algorithm. [online] InterviewBit. Available at: https://www.interviewbit.com/tutorial/quicksort-algorithm/#:~:text=SpaceComplexityofQuicksort&text=Theworstcasespaceused [Accessed 7 May 2022].
42. https://learnonline.gmit.ie/pluginfile.php/579172/mod_resource/content/0/09%20Sorting%20Algorithms%20Part%203.pdf
43. Donald E. Knuth. 1998. The art of computer programming, volume 3: (2nd ed.) sorting and searching. Addison Wesley Longman Publishing Co., Inc., USA.
44. Tutorialspoint.Dev. (n.d.). Counting Sort - TutorialsPoint.dev. [online] Available at: https://tutorialspoint.dev/algorithm/sorting-algorithms/counting-sort [Accessed 7 May 2022].
45. GeeksforGeeks. (2013). Counting Sort. [online] Available at: https://www.geeksforgeeks.org/counting-sort/.
46. Interview Cake: Programming Interview Questions and Tips. (n.d.). Counting Sort Algorithm | Interview Cake. [online] Available at: https://www.interviewcake.com/concept/java/counting-sort.
47. Simplilearn.com. (2021). Counting Sort Algorithm: Overview, Time Complexity & More | Simplilearn. [online] Available at: https://www.simplilearn.com/tutorials/data-structure-tutorial/counting-sort-algorithm?source=sl_frs_nav_playlist_video_clicked.
48. FavTutor. 2022. Counting Sort in Python (Code with Example) | FavTutor. [online] Available at: https://favtutor.com/blogs/counting-sort-python [Accessed 7 May 2022].
49. https://learnonline.gmit.ie/pluginfile.php/579165/mod_resource/content/0/08%20Sorting%20Algorithms%20Part%202.pdf
50. GeeksforGeeks. (2014). Selection Sort - GeeksforGeeks. [online] Available at: https://www.geeksforgeeks.org/selection-sort/.
51. https://www.happycoders.eu/. (2020). Selection Sort – Algorithm, Source Code, Time Complexity. [online] Available at: https://www.happycoders.eu/algorithms/selection-sort/#Selection_Sort_Time_Complexity [Accessed 8 May 2022].

52. GeeksforGeeks. (2014). Python Program for Selection Sort. [online] Available at: https://www.geeksforgeeks.org/python-program-for-selection-sort/.
53. Programiz.com. (2020). Insertion Sort Algorithm. [online] Available at: https://www.programiz.com/dsa/insertion-sort.
54. GeeksforGeeks. (2019). Comparison among Bubble Sort, Selection Sort and Insertion Sort. [online] Available at: https://www.geeksforgeeks.org/comparison-among-bubble-sort-selection-sort-and-insertion-sort/?ref=rp.
55. GeeksforGeeks (2013). Insertion Sort - GeeksforGeeks. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/insertion-sort/.