2021

Computational Thinking with Algorithms

PROJECT

ARON NUTLEY – G00388055

Contents

Introduction

The main objective of this project was to write an application that will benchmark five (5) different sorting algorithms, along with the production of this report. The aim of the report is to discuss the algorithms that have been chosen as well as the results of the benchmarking process. Before introducing these there are key concepts that need to be discussed. These are as follows:

1. The Concept of Sorting
2. Sorting Algorithms
   a. Requirement
   b. Types
3. Appropriate Algorithm

Concept of Sorting

The general concept of sorting is "to arrange things in groups in a particular order according to their type". (Oxford, 2021) Whether it be shape, colour or size, there are many ways things can be sorted. However, the sorting process always follows a pattern of some sort. Humans have the subconscious mental ability to easily carry out the sorting process, adapting it with little instruction. Computers need to follow an exact set of instructions to carry out a similar task but are nowhere near as adaptable as humans. (Harel, 2004) This project will focus on the latter.

Sorting Algorithms

To implement sorting with computers we use algorithms. An algorithm is "a set of rules that must be followed when solving a particular problem". (Oxford, 2021) Combining both definitions to answer what is a sorting algorithm is somewhat lengthy, and not completely accurate. Simply put, a sorting algorithm is "an algorithm that sorts arrays of data". (Techopedia, 2021)

Requirement

Sorting algorithms have been extensively researched throughout the era of computing, particularly in the early days. Many different approaches have been used and a vast number of sorting algorithms have been developed and analysed. (Mannion, 2021) Sorting is one of the most important algorithmic problems used in applications, to the extent that around 25% of all CPU cycles are spent sorting. (Mannion, 2021) CPU cycles are a valuable resource, therefore there is significant incentive to study the optimisation of sorting algorithms as this increases the efficiency of the CPU. In addition to this, numerous computations and tasks become more efficient by properly sorting information in advance. (Heineman, et al., 2016)

Types

They are many different types of sorting algorithms; this project will look at the following two (2):

1. ***Comparison-Based Sort***
   Comparison-based sorting is a type of sorting algorithm that makes use of comparison operations ***(+, =, -, <, > etc.)*** to determine the order in which elements

should appear in a sorted list. (Mannion, 2021) A sorting algorithm is deemed to be comparison-based if the information about the order is gained by comparing a pair of elements to see if they are ***less than or equal to*** each other. The comparison-based sorting algorithms that are used in this project are Bubble Sort, Merge Sort, Cocktail Sort, and Gnome Sort. (Geeks for Geeks, 2018)

2. **Non-Comparison Sort**
   Also known as Linear sort; non-comparison sorting algorithms make prior assumptions about the data that is being sorted, which allows them to run in linear time ***O(n)***. (Hinklemann, 2019) Unlike comparison-based sorting algorithms, non-comparison sort does not compare every element of input. This makes it quicker and more efficient at sorting larger inputs of data. Examples of a non-comparison sorting algorithm used in this project is Bucket Sort.

Algorithm Selection

An analogy for how to select an appropriate algorithm would be to compare them to cars. All (working) cars will get their user from A to B, and all (working) sorting algorithms will sort data. However, in the same way that you would not expect to take an F1 car on the school run, you need to select an appropriate sorting algorithm depending on the data that is being sorted. (Woltmann, 2020) It is important that you understand the algorithm that you are using to get the most out of it.

Table 1, adapted from Algorithms in a Nutshell by George T. Heineman asks you to choose a sorting algorithm while considering the "qualitative criteria" (Heineman, et al., 2016) in *Table 1*.

Criteria	Sorting Algorithm
Only a few items	Insertion Sort
Items are mostly sorted already	Insertion Sort
Concerned about worst-case scenarios	Heap Sort
Interested in a good average-case behaviour	Quicksort
Items are drawn from a uniform dense universe	Bucket Sort
Desire to write as little code as possible	Insertion Sort
Require stable sort	Merge Sort

Table 1 – Criteria for particular Sorting Algorithms (Heineman, et al., 2016)

In general, when selecting the appropriate algorithm to use, you will need to consider the following characteristics: (Woltmann, 2020)

1. **Complexity (Time & Space)**
2. **Stability**
3. **Comparison Sorts/Non-Comparison Sorts**
4. **Parallelism**

5. **Recursive/Non-Recursive**
6. **Adaptability**

Complexity

Algorithmic complexity is a way of comparing the efficiency of an algorithm. Complexity can be measured by the time it takes for a program to execute (time complexity) or the memory that it will use (space complexity). (Isaac Computer Science, 2021)

Time Complexity

One of the key aspects to consider when selecting an appropriate sorting algorithm is its speed; in particular, how speed changes depending on the amount of data to be sorted. (Woltmann, 2020) "One algorithm can be twice as fast as another at a hundred elements, but at a thousand elements, it can be five times slower". (Woltmann, 2020)

Time complexity is measured by counting the number of fundamental operations an algorithm computes when executed. On the assumption that each operation requires a fixed amount of time to complete, the total number of operations indicates the time that an algorithm requires during its execution. (Isaac Computer Science, 2021)

Running time $T(n)$ is proportional to:	Complexity:
$T(n) \propto \log n$	logarithmic
$T(n) \propto n$	linear
$T(n) \propto n \log n$	linearithmic
$T(n) \propto n^2$	quadratic
$T(n) \propto n^3$	cubic
$T(n) \propto n^k$	polynomial
$T(n) \propto 2^n$	exponential
$T(n) \propto k^n;\ k > 1$	exponential

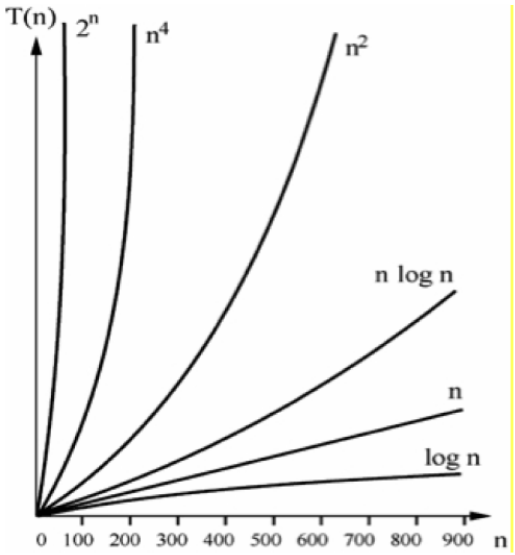

Figure 1 – Growth in Relation to Running Time and Size (Mannion, 2021)

Space Complexity

Space complexity refers to the amount of memory used by the algorithm (including the input values) to execute and produce a result. (Geeks for Geeks, 2021) Space Complexity is measured using Big-O Notation and is determined by the amount of memory that an algorithm requires when it runs. (Isaac Computer Science, 2021)

Big-O Notation

Big-O Notation is “a formal expression of an algorithm’s complexity in relation to the growth of the input size.” (Isaac Computer Science, 2021) It is used to rank algorithms based on their performance when dealing with large input sizes.

Notation	Name
O(1)	Constant
O(log(n))	Logarithmic
$O((\log(n))^c)$	Polylogarithmic
O(n)	Linear
$O(n^2)$	Quadratic
$O(n^c)$	Polynomial
$O(n^n)$	Exponential

Table 2 – Types of Growth and their Big-O Notation (MIT, 2021)

Stability

Stability of a sorting algorithm refers to how the algorithm deals with equal (repeated) data. A sorting algorithm is deemed to be stable when the relative order of equal elements is preserved. An unstable sorting algorithm does not preserve such an order. (Srivastava, 2020)

Comparison Sorts/Non-Comparison Sorts

Comparison Sort is the most used type of sorting algorithm. However, depending on the input data it can be more appropriate to use a non-comparison sort (See *Table 1*).

Parallelism

Parallelism means if and to what extent a sorting algorithm is suitable for parallel process on multiple CPU cores. (Woltmann, 2020)

Recursive/Non-Recursive

Recursive sorting algorithms require additional memory on the stack. If this requirement is too large this can lead to ***StackOverflowExecption***. (Woltmann, 2020)

Adaptability

Adaptive sorting algorithms can adapt their behaviour during execution. They can recognise pre-sorted elements when sorting making them much faster than when dealing with randomly distributed elements. (Woltmann, 2020)

Sorting Algorithms

For this project five (5) sorting algorithms were selected based on the following criteria:

1. A simple comparison-based sort (Bubble Sort, Selection Sort, or Insertion Sort)
2. An efficient comparison-based sort (Merge Sort, Quicksort or Heap Sort)
3. A non-comparison sort (Counting Sort, Bucket Sort or Radix Sort)
4. Any other sorting algorithm of your choice.
5. Any other sorting algorithm of your choice.

The selected algorithms and information relating to them are discussed below.

Bubble Sort

Bubble Sort is a simple comparison-based sorting algorithm. It is used to sort an array of size ***n,*** with a number of elements ***n***. (Study Tonight, 2021) It works by 'bubbling' values to the end of the list over successive passes. During the first pass it evaluates the list while moving the largest value (smallest if sort order dictates) to the end of the array. On the second pass the next appropriate value bubbles to the second last position in the array. This action is repeated until the array is sorted. (Isaac Computer Science, 2021)

Method

Imagine that you have been given the following numbers (*Figure 2)* and are tasked with sorting them in ascending order.

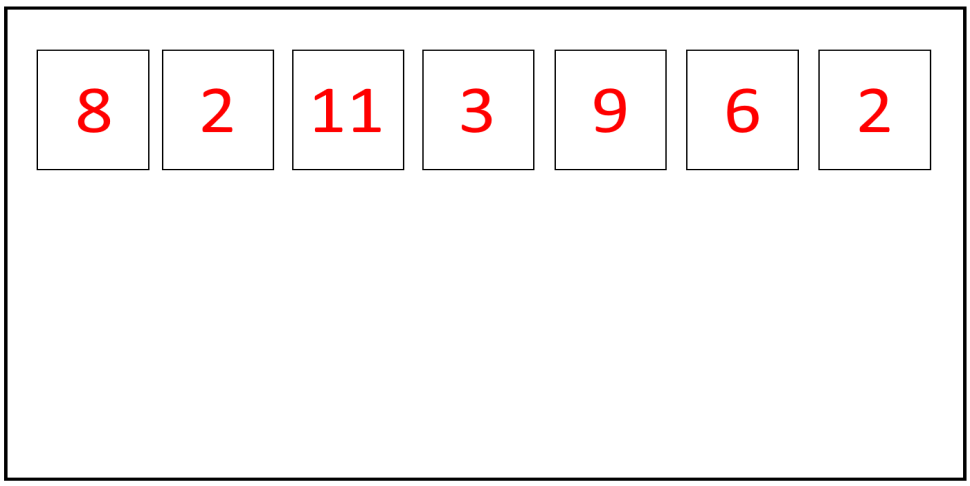

Figure 2 – Unsorted Values 8,2,11,3,9,6,2

A Bubble Sort algorithm looks at the number in the first position of the array. In the example above this is the number 8. It then compares that value to the next value, in this case the number 2. If the first value is greater than the second, they are swapped. In this example the values are swapped as 8 is greater than 2. The value in the second position is then compared with the one in the third position. If necessary, these values are swapped. This process continues, with the algorithm comparing each number to the one that follows. Once they have all been compared, the highest value will be at the end (*Figure 9*). This completes the first pass of the algorithm. *Figures* 3 to 9 shows how the first pass is executed. (Isaac Computer Science, 2021)

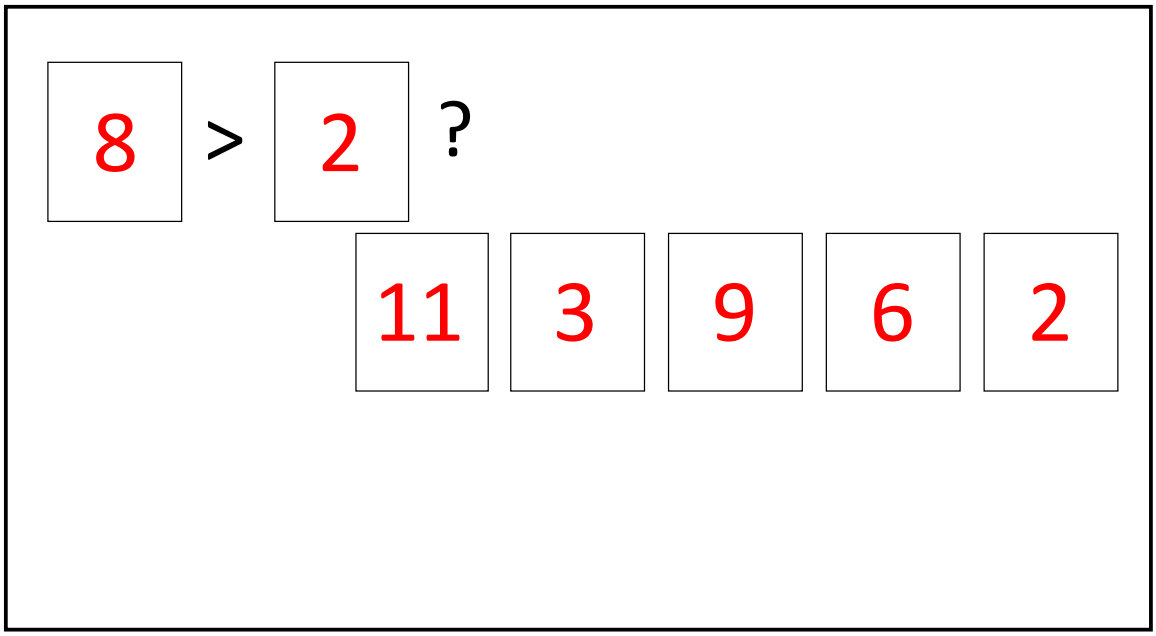

Figure 3 – First Pass Step 01

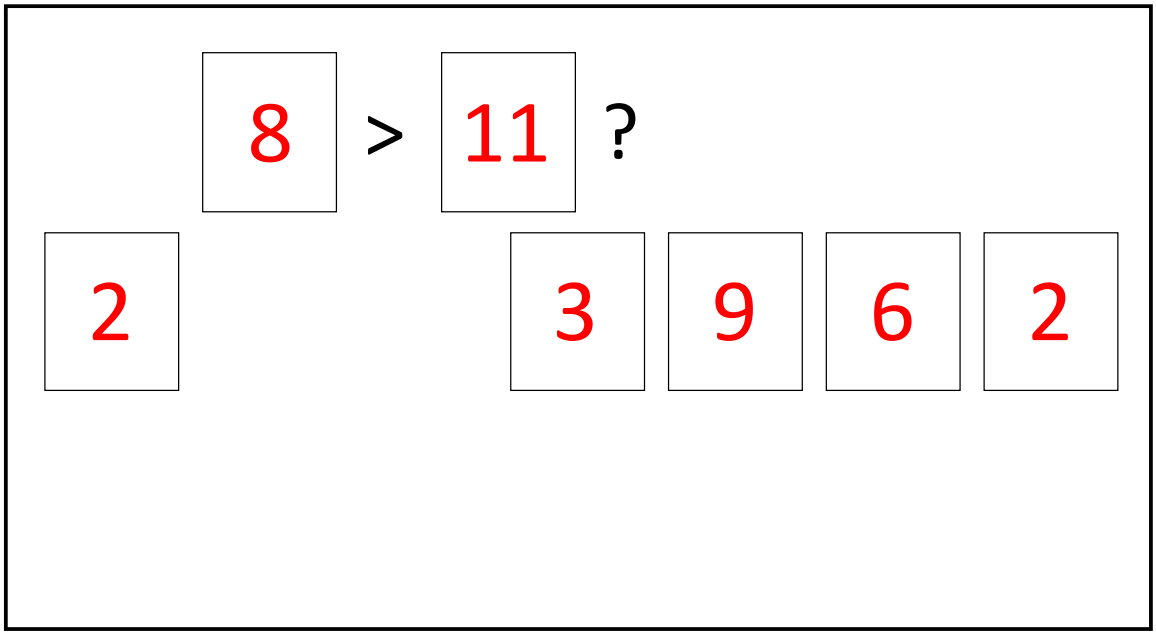

Figure 4 – First Pass Step 02

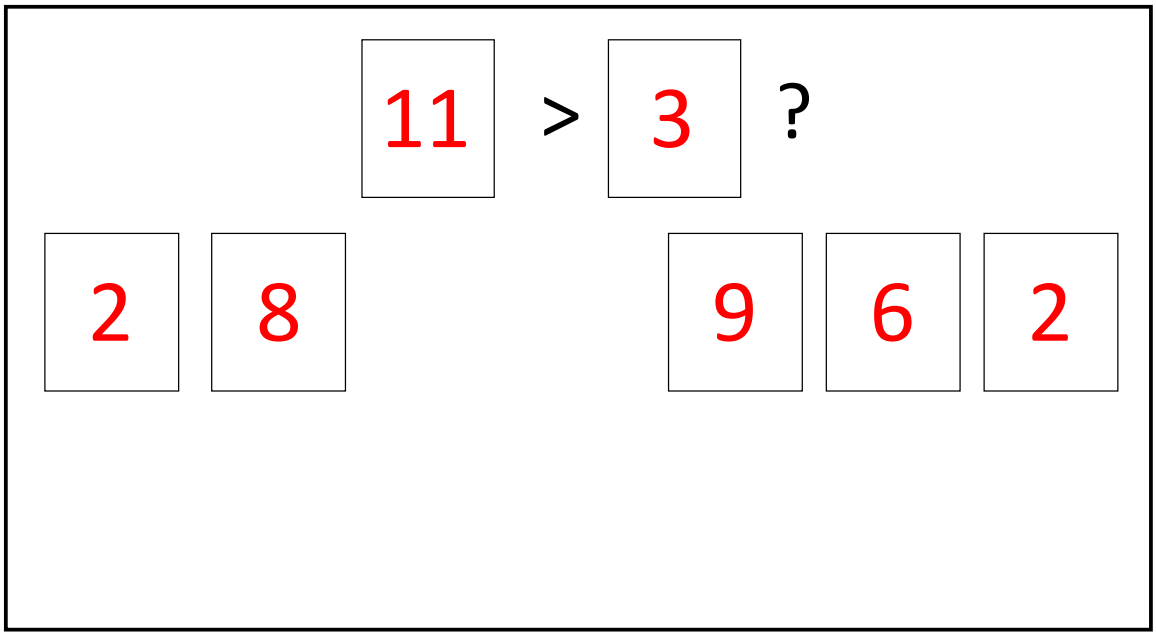

Figure 5 – First Pass Step 03

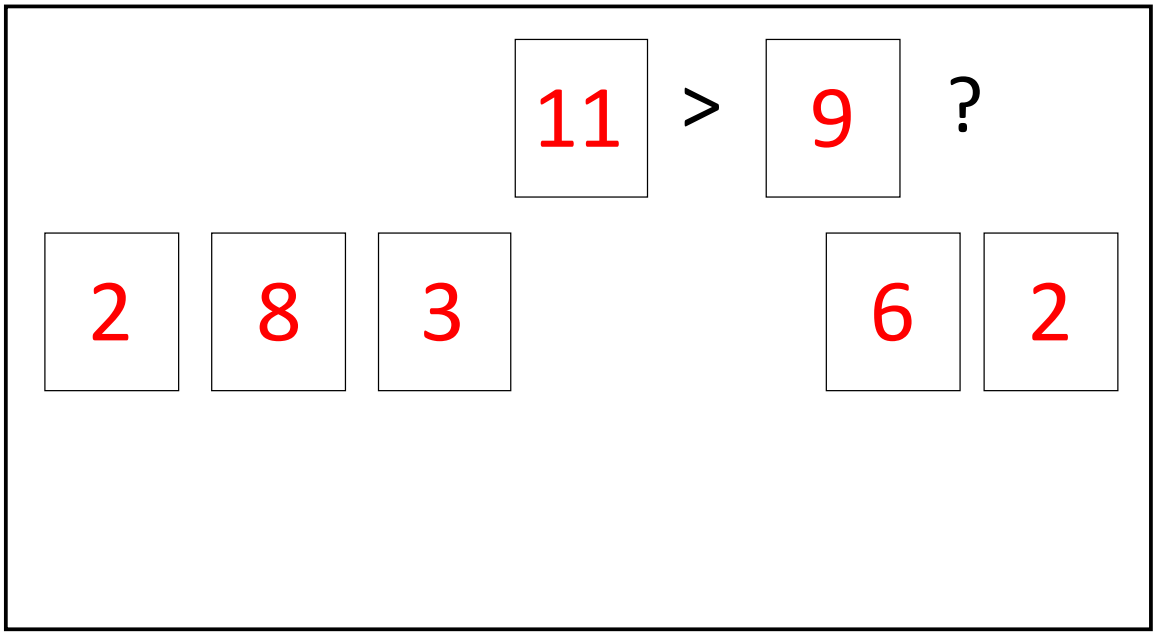

Figure 6 – First Pass Step 04

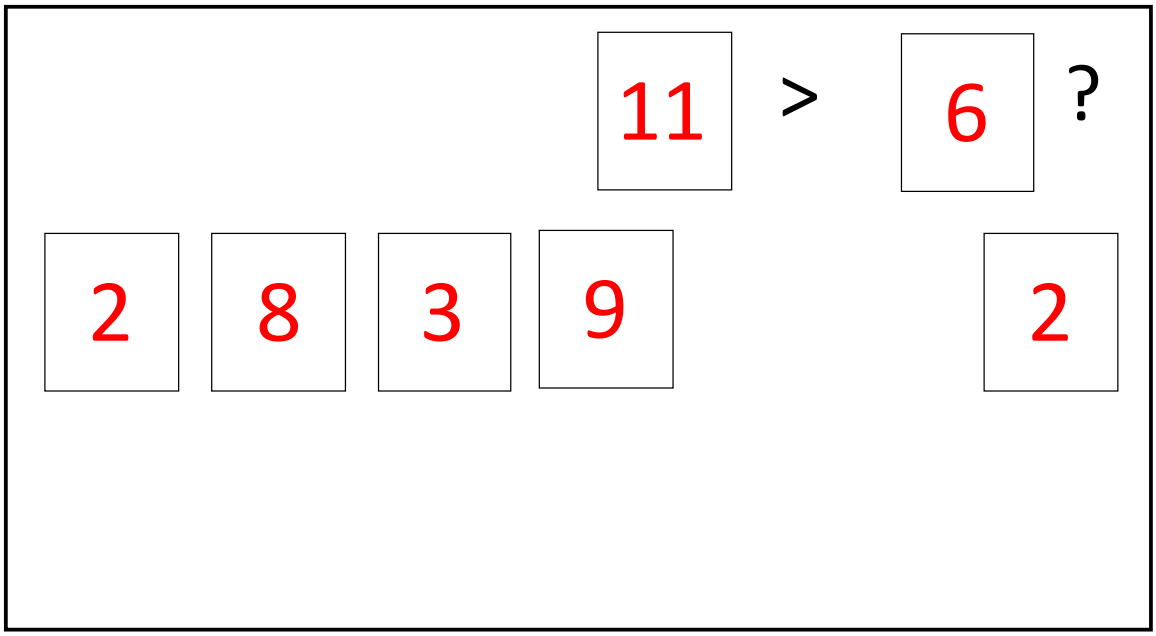

Figure 7 – First Pass Step 05

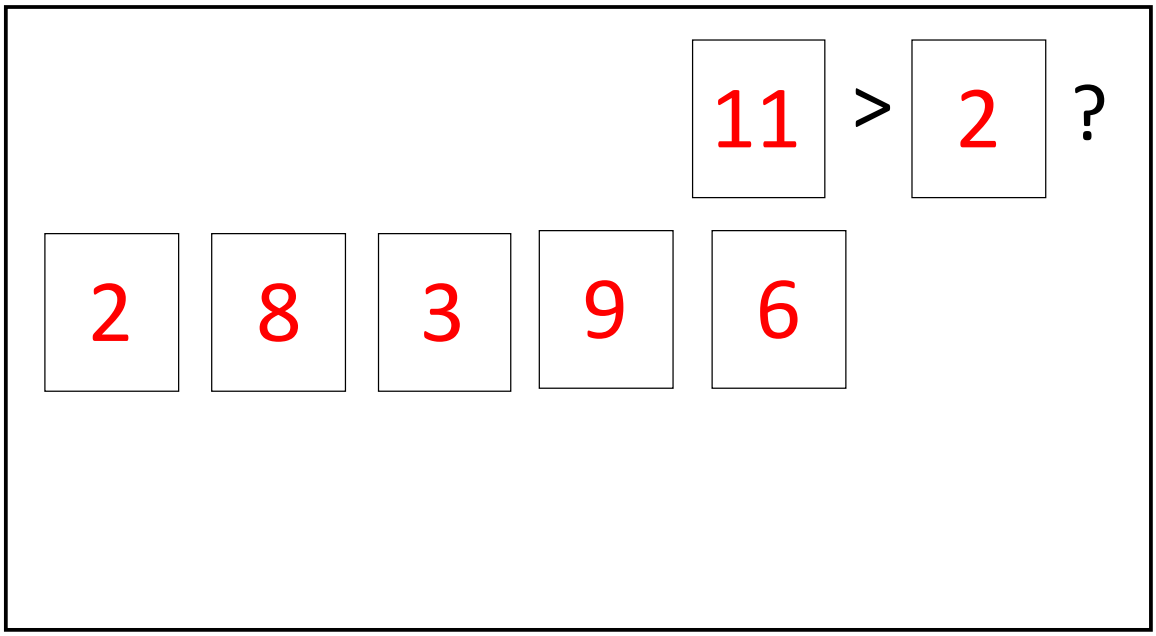

Figure 8 – First Pass Step 06

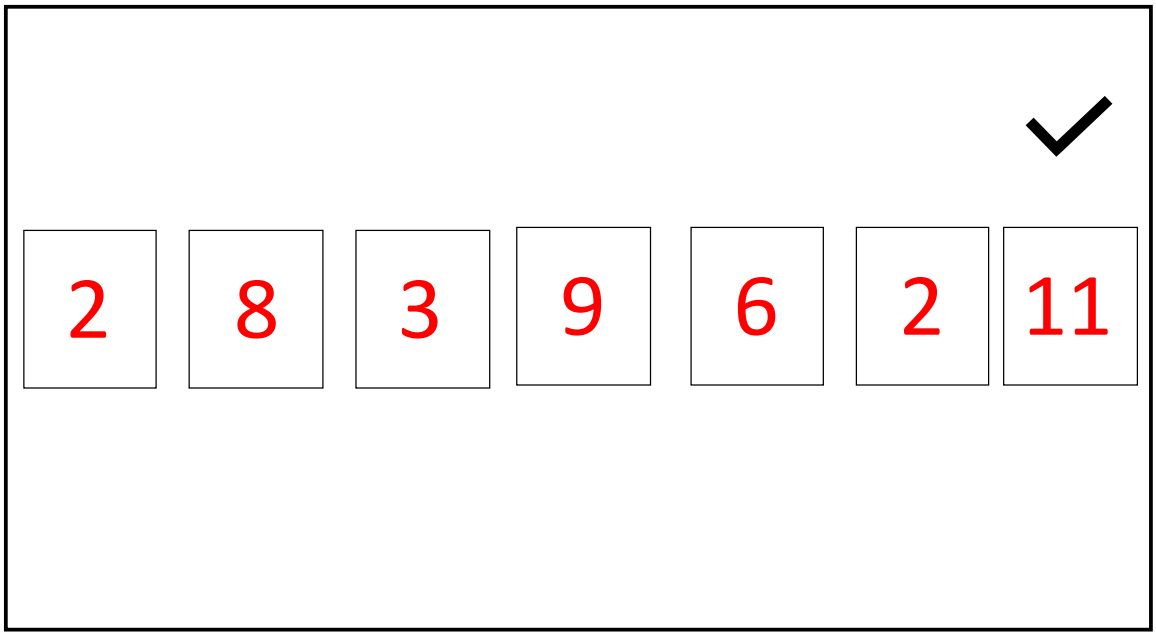

Figure 9 – First Pass Complete

Once the first pass has been completed the application will carry out the same process again. This time it will place the second highest number in the second last position in the array. In the case of the example above, the resulting order of the second pass will be [2,8,3,6,2,9,11]. The algorithm will continue to repeat this process until the numbers are sorted into order. (Isaac Computer Science, 2021)

Time Space Complexity – Bubble Sort

Worst Case Time Complexity	Best Case Time Complexity	Average Time Complexity	Space Complexity
$O(n^2)$	O(n)	$O(n^2)$	O(1)

Table 3 – Time Space Complexity Bubble Sort (Study Tonight, 2021)

Bubble Sort's simplicity means that it is suited for problems where the input data is nearly sorted or if there is a small input size to be sorted. It does not perform well with problems where data is mostly unsorted and large. (Study Tonight, 2021) In terms of memory requirement, it is very efficient as the only item that needs to be copied is the value that is being swapped.

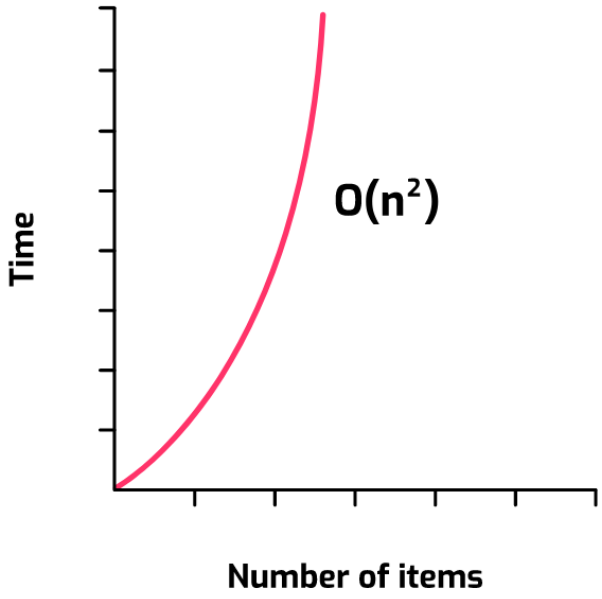

Figure 10 – Bubble Sort Time Complexity (Isaac Computer Science, 2021)

Pseudocode of Bubble Sort

```
procedure bubbleSort( list : array of items )

   loop = list.count;

   for i = 0 to loop-1 do:
      swapped = false

      for j = 0 to loop-1 do:

         /* compare the adjacent elements */
         if list[j] > list[j+1] then
            /* swap them */
            swap( list[j], list[j+1] )
            swapped = true
         end if

      end for

      /*if no number was swapped that means
      array is sorted now, break the loop.*/

      if(not swapped) then
         break
      end if

   end for

end procedure return list
```

Figure 11 – Pseudocode of Bubble Sort (Study Tonight, 2021)

A point to note about the pseudocode is that after every iteration the highest value settles down at the end of the array. The efficiency of the code can be improved if this element is disregarded during the next iteration of the code as it is already sorted. (Study Tonight, 2021)

Merge Sort

Merge Sort is an efficient comparison-based sorting algorithm. The algorithm uses a "divide-and-conquer" (Study Tonight, 2021) method to sort an array of data. It divides the array into smaller sub-arrays and then merges these sub-arrays to form a sorted array of data.

Method

Step 01: Divide the array by finding the first and last number in an array. In the case of the example below (*Figure 12*) ***p*** and ***r*** represent the start and end points of array, while ***q*** represents the midpoint.

Step 02: Recursively sort the subarrays into two further subarrays.

Step 03: Sort and combine the subarrays by merging them back into a single sorted subarray.

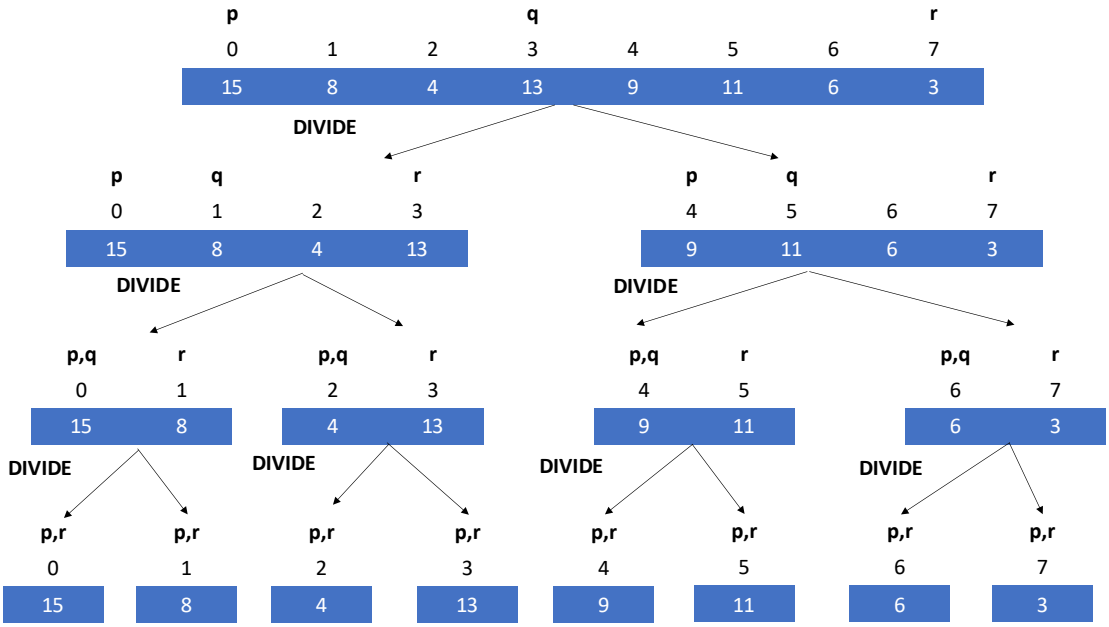

Figure 12 – Depicts an Array being divided into a series of subarrays

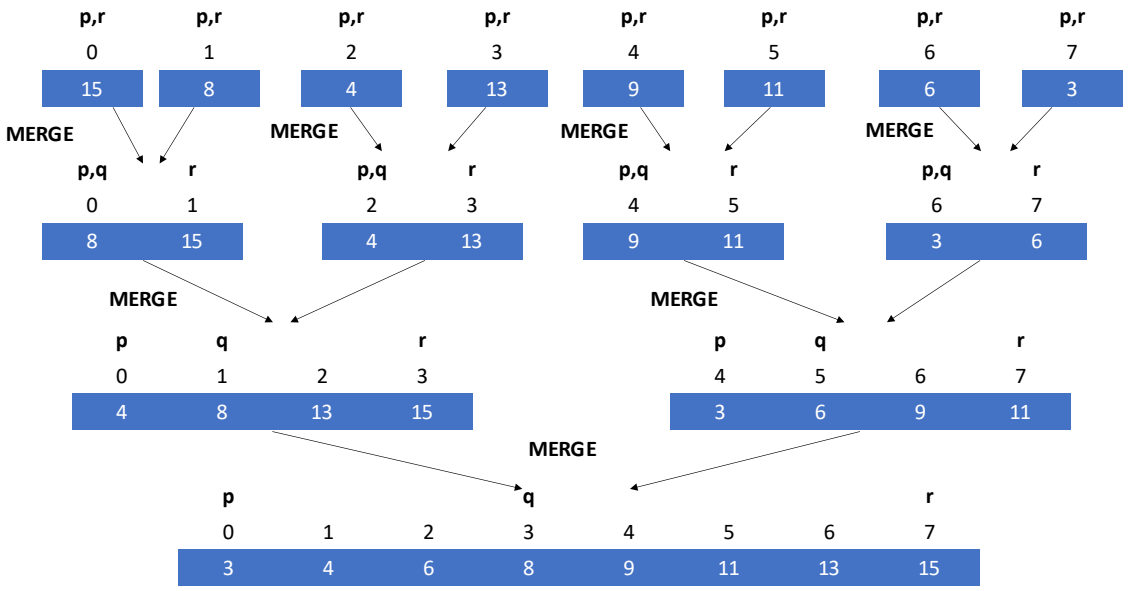

Figure 13 – Subarrays being merged into a single sorted array

Time Space Complexity – Merge Sort

Worst Case Time Complexity	Best Case Time Complexity	Average Time Complexity	Space Complexity
O(log(n))	O(log(n))	O(log(n))	O(n)

Table 4 – Time Space Complexity Merge Sort (Study Tonight, 2021)

The time complexity of the Merge Sort algorithm is O(log(n)). When calculating the time complexity, you must consider that the two main operations of the algorithm are dividing and merging. (Study Tonight, 2021) When splitting an array of size ***n***, the array is continuously divided until only sub-arrays remain that have single values ***n***. If we keep dividing an array of size ***n*** by 2, after $\log_2 n$ steps we will have ***n*** sub-arrays with 1 item. Therefore, the time complexity of the dividing is O(log(n)), as the list is divided log(n) times. (Isaac Computer Science, 2021)

When merging the sub-arrays of size ***n***, the subarrays are combined by examining every item and placing them into an ordered merged list. Due to this, the worst-case scenario is that there will be ***n*** comparisons to order all ***n*** values for each merge. (Isaac Computer Science, 2021)

There are $\log_2 n$ cases of division for every division that takes place and ***n*** values are compared and merged for every instance of merging. The division and merging process together involves nlog2n operations which means the time complexity of Merge Sort is O(log(n)). (Isaac Computer Science, 2021)

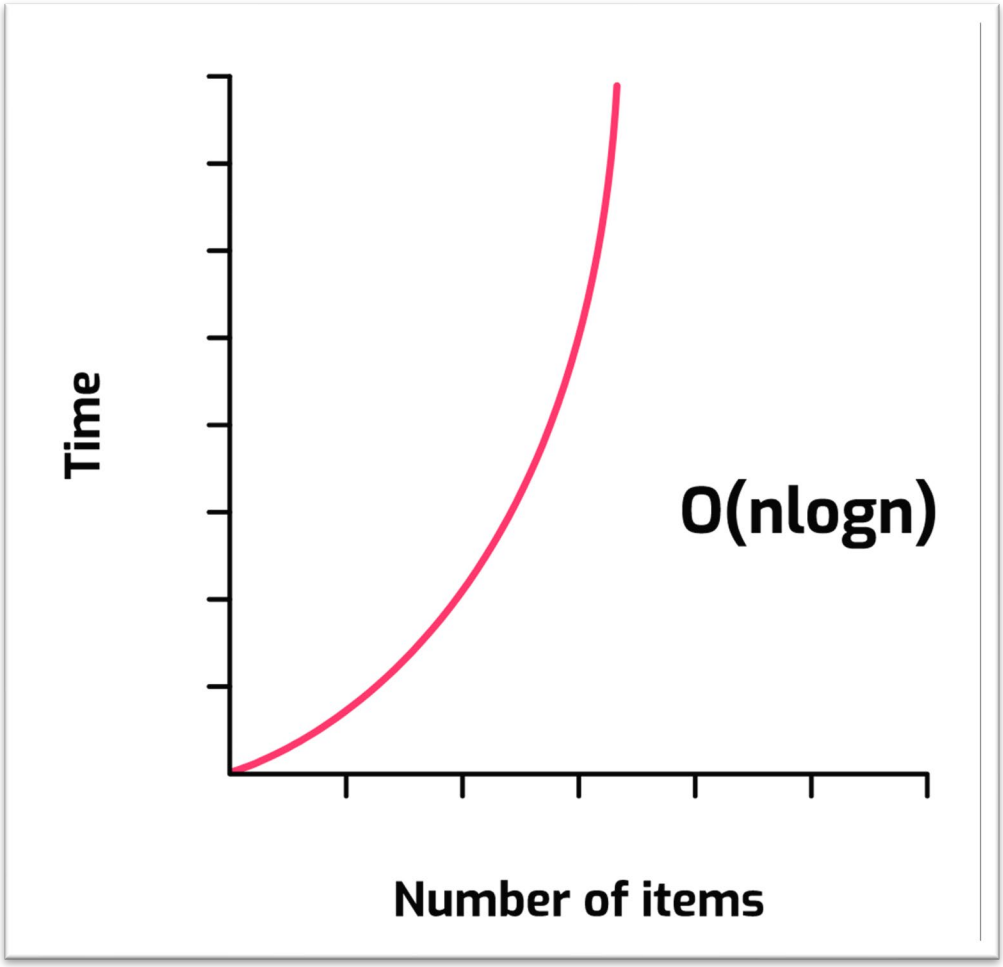

Figure 14 – Time Complexity Graph of Merge Sort (Isaac Computer Science, 2021)

Pseudocode of Merge Sort

In the example in *Figure 15* we can see how an array is divided and then merged. This is done with recursion.

```
procedure mergesort( var a as array )
   if ( n == 1 ) return a

   var l1 as array = a[0] ... a[n/2]
   var l2 as array = a[n/2+1] ... a[n]

   l1 = mergesort( l1 )
   l2 = mergesort( l2 )

   return merge( l1, l2 )
end procedure

procedure merge( var a as array, var b as array )

   var c as array
   while ( a and b have elements )
      if ( a[0] > b[0] )
         add b[0] to the end of c
         remove b[0] from b
      else
         add a[0] to the end of c
         remove a[0] from a
      end if
   end while

   while ( a has elements )
      add a[0] to the end of c
      remove a[0] from a
   end while

   while ( b has elements )
      add b[0] to the end of c
      remove b[0] from b
   end while

   return c

end procedure
```

Figure 15 – Pseudocode Merge Sort (Point Tutorials, 2021)

Counting Sort

Counting Sort is a non-comparison linear sorting algorithm based on keys that are between a specific range. It works by counting the number of objects that have distinct key values. These values are small integers and lie between a specific range. It counts the number of objects that have such a key and then uses a calculation to place each key value into a sorted sequence. (Geeks For Geeks, 2021)

Method

Step 01: Determine the key range. In the example below the data being sorted only numbers from 0 to 9. This is based on the maximum value in the array. (Cake Labs, 2021)

4	6	3	5	5	2	0	2	8	7	7	9	6	2	8

Figure 16 – Array to be sorted.

Step 02: An auxiliary array is created to store the count of the elements in the array. (Programiz, 2021) It is important to note that the first value of an array is zero (0) and this must be taken into consideration when choosing the appropriate size of the Auxiliary Array.

0	0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10

Figure 17 – Auxiliary Array

Step 03: Store the count of each element of the array into the respective count in the Auxiliary Array. (Programiz, 2021) In *Figure 18,* 4 is the first element of the array. We increase the value in the Auxiliary Array at position 4 by 1. The same is done with all other elements. For example, the element at array index 2 increases the auxiliary array index 6 by one. This process is repeated until all elements of the array are placed in their relative position.

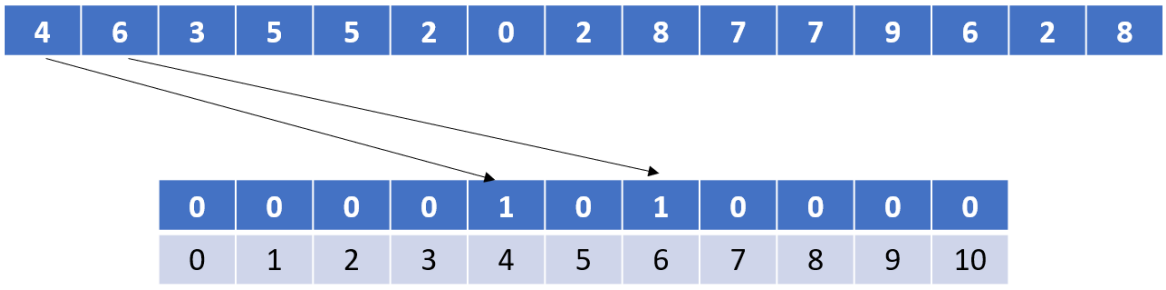

Figure 18 – Counting Process for first 2 elements

1	0	3	1	1	2	2	2	2	1	0
0	1	2	3	4	5	6	7	8	9	10

Figure 19 – Auxiliary Array after Count has completed.

Step 04: Using the Auxiliary Array the sorted output is constructed. In *Figure 19* we can see that the position of an element in the array directly corresponds to the number of times that value occurs. In *Figure 19* we have 1 x 0, 0 x 1, 3 x 2, and so forth. The sorted result is seen in *Figure 20*.

0	2	2	2	3	4	5	5	6	6	7	7	8	8	9

Figure 20 – Sorted Array

Time Space Complexity – Count Sort

Worst Case Time Complexity	Best Case Time Complexity	Average Time Complexity	Space Complexity
O(n+k)	O(n+k)	O(n+k)	O(n+k)

Table 5 – Time Space Complexity Count Sort

The time and space complexity for Count Sort is O(n+k) where ***n*** is the number of elements in the input array and ***k*** is the range of the output. (Geeks For Geeks, 2021) When dealing with the algorithm it must be noted that there are four (4) main loops. This does not include finding the greatest value, which can be done outside of the function. (Programiz, 2021)

For-Loop	Time of Counting
1st	O(max)
2nd	O(size)
3rd	O(max)
4th	O(size)

Table 6 – Loops Relating to Big-O Notation (Programiz, 2021)

Taking *Table 6* into account, the Overall Complexity of Count Sort is O(max) + O(size) + O(max) + O(size) which can be derived into O(max + size). This equates to O(n + k) which is the case in all of the above for worst, best and average case complexity. (Programiz, 2021)

Pseudocode of Count Sort

```
1  CountingSort (A,B,k)
2  Let C[0......k} be a new array
3  for i=0 to k
4      C[i]=0;
5  for j=1 to A.length or n
6      C[A[j]] = C[A[j]] + 1;
7  for i=1 to k
8      C[i] = C[i] + C[i-1];
9  for j=n or A.length down to 1
10     B[ C[A[j]]] = A[j];
11     C[ A[j]] = C[A[j]] -1;
```

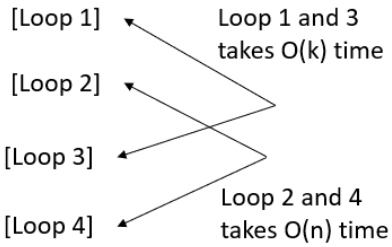

Figure 21 – Pseudocode of Counting Sort (Elkahlout & Maghari, 2017)

In *Figure 21* we can see the four (4) for loops being used in the pseudocode. The first loop initiates the array to sort, the second counts each element, the third positions them in the Auxiliary Array then the final one builds the output array. (Mangal, 2017) It must also be noted that these for loops are from which the time complexity is calculated.

Cocktail Sort

Cocktail Sort is a Comparison Based Sorting algorithm that is a variation of Bubble Sort. As already discussed, the Bubble Sort algorithm works by traversing the array and moving the largest (or smallest) element to the end of the array. Cocktail Sort works in a similar manner except that it traverses through the array in both directions alternatively. (Geeks For Geeks, 2020) Despite doing double the work of Bubble Sort in a single iteration, Cocktail Sort does not typically perform twice as fast. (Lukic, n.d.)

Method

How Cocktail Sort works is best demonstrated by a practical example. Let us consider the following example array [6,2,5,2,9,0,3]. (Geeks For Geeks, 2020)

Step 01: First Forward Pass

The first two values in the array (6,2) are compared to each other. As 6 > 2 they are swapped.

[**6**,**2**,5,2,9,0,3] – 6 > 2 therefore resulting array is [**2**,**6**,5,2,9,0,3]

The second two values in the array (6,5) are compared to each other. As 6 > 5 they are swapped.

[2,**6**,**5**,2,9,0,3] – 6 > 5 therefore resulting array is [2,**5**,**6**,2,9,0,3]

This process is repeated until the algorithm completes its first pass.

[2,5,**6**,**2**,9,0,3] – 6 > 2 therefore resulting array is [2,5,**2**,**6**,9,0,3]

[2,5,2,**6**,**9**,0,3] – 6 < 9 therefore resulting array is [2,5,2,**6**,**9**,0,3]

[2,5,2,6,**9**,**0**,3] – 9 > 0 therefore resulting array is [2,5,2,6,**0**,**9**,3]

[2,5,2,6,0,**9**,**3**] – 9 > 3 therefore resulting array is [2,5,2,6,0,**3**,**9**]

Once the first pass has been completed the greatest element in the array will be positioned at the last index of the array. (Geeks For Geeks, 2020)

Step 02: First Backward Pass:

This process works similarly to the forward pass. Starting on the last unsorted index in the array the algorithm works backwards swapping values using the comparison operation. At the end of this process the smallest element in the array will be positioned at the first index of the array. (Lukic, n.d.)

[2,5,2,6,**0**,**3**,9] - [2,5,2,6,**0**,**3**,9] Remain in same position as 3 > 0

[2,5,2,**6**,**0**,3,9] - [2,5,2,**0**,**6**,3,9] Swapped as 6 > 0

[2,5,**2**,**0**,6,3,9] - [2,5,**0**,**2**,6,3,9] Swapped as 2 > 0

[2,**5**,**0**,2,6,3,9] - [2,**0**,**5**,2,6,3,9] Swapped as 5 > 0

[**2**,**0**,5,2,6,3,9] - [**0**,**2**,5,2,6,3,9] Swapped as 2 > 0 First Backward Pass completed.

Step 03: Repeat Passes

Once the first backward pass has completed the algorithm will start a new forward pass. Starting with the value in array index one (1) the application will compare values and make the necessary swaps until the second highest value is in the second last position of the array. Once this has been completed a new backward pass will commence. This process is completed until the array is sorted.

[0,**2**,**5**,2,6,3,9] - [0,**2**,**5**,2,6,3,9]

[0,2,**5**,**2**,6,3,9] - [0,2,**2**,**5**,6,3,9]

[0,2,2,**5**,**6**,3,9] - [0,2,2,**5**,**6**,3,9]

[0,2,2,5,**6**,**3**,9] - [0,2,2,5,**3**,**6**,9] Forward Pass Completed

[0,2,2,**5**,**3**,6,9] - [0,2,2,**3**,**5**,6,9]

[0,2,**2**,**3**,5,6,9] - [0,2,**2**,**3**,5,6,9]

[0,**2**,**2**,3,5,6,9] - [0,**2**,**2**,3,5,6,9] 2 = 2 therefore pass is complete, and array is sorted.

Time Space Complexity – Cocktail Sort

Worst Case Time Complexity	Best Case Time Complexity	Average Time Complexity	Space Complexity
$O(n^2)$	O(n)	$O(n^2)$	O(1)

Table 7- Time Space Complexity Cocktail Sort (Growing With The Web, 2016)

The best time complexity for Cocktail Sort occurs when all the elements in the array are already or nearly sorted. Like Bubble Sort, the worst case occurs when the sequence is in reverse order. (Hurna, 2021)

Cocktail Sort does not use buffers or make use of recursion therefore it requires O(1) space in all cases. (Hurna, 2021)

Pseudocode of Cocktail Sort

In the pseudocode in Figure 22 you can see similarities with Bubble Sort. The additional for loop at the end is the key difference between the algorithms.

```
procedure cocktailShakerSort( A : list of sortable items ) defined as: do
    swapped ← false
     if A[ i ] > A[ i + 1 ] then // test whether the two elements are in the wrong order
        swap( A[ i ], A[ i + 1 ] ) // let the two elements change places
        swapped ← true
      end if
    end for
    if swapped = false then
      // we can exit the outer loop here if no swaps occurred.
      break do-while loop
    end if
    swapped ← false
    for each i in length( A ) - 1 to 0 do:
      if A[ i ] > A[ i + 1 ] then
        swap( A[ i ], A[ i + 1 ] )
        swapped ← true
      end if
    end for
  while swapped // if no elements have been swapped, then the list is sorted
end procedure
```

Figure 22 – Pseudocode Cocktail Sort (Elkahlout & Maghari, 2017)

Gnome Sort

Gnome Sort is a simple comparison-based sorting algorithm that works by adjusting adjacent elements in an array if they are not in order. (Open Genus, 2021)It is based on the concept of a Garden Gnome sorting its flowerpots and this is where it gets its name. The gnome looks at the post next to them and the one preceding it. If they are in the correct position, he steps forward onto the next pot. If not, the Gnome will swap the pots stepping the pot back to the previous one's position. If there is no previous pot (starting position) the gnome steps forward. If there is no pot next to the Gnome, it is at the finish point and done. (Geeks For Geeks, 2021)

Method

Step 01: If positioned at the start of the array then go to the next element on the right (arr[0] to arr[1]).

Step 02: Compare the element at arr[1] to arr[0]. If arr[1] is ***greater than or equal to*** arr[0] then proceed onto the next array index. If the current array element is smaller than the previous one swap them and go one step backwards. This process is repeated until the array is sorted. (Geeks For Geeks, 2021)This is shown in *Table 8*.

Current Array	Position	Condition In Effect	Action To Take
[**6**,4,3,5]	0	Array Index = 0	Increase Position
[6,**4**,3,5]	1	arr[1] < arr[0]	Swap, Decrease Position
[**4**,6,3,5]	0	Array Index = 0	Increase Position
[4,**6**,3,5]	1	arr[1] > arr[0]	Increase Position
[4,6,**3**,5]	2	arr[2] < arr[1]	Swap, Decrease Position
[4,**3**,6,5]	1	arr[1] < arr[0]	Swap, Decrease Position
[**3**,4,6,5]	0	Array Index = 0	Increase Position
[3,**4**,6,5]	1	arr[1] > arr[0]	Increase Position
[3,4,**6**,5]	2	arr[2] > arr[1]	Increase Position
[3,4,6,**5**]	3	arr[3] < arr[2]	Swap, Decrease Position
[3,4,**5**,6]	2	arr[2] > arr[1]	Increase Position
[3,4,5,**6**]	3	arr[3] > arr[2]	Increase Position
[3,4,5,6]	4	Position = Length (arr)	completed

Table 8 – Gnome Sort Example (Open Genus, 2021)

Space Time Complexity – Gnome Sort

Worst Case Time Complexity	Best Case Time Complexity	Average Time Complexity	Space Complexity
$O(n^2)$	$O(n^2)$	O(n)	O(1)

Table 9 – Time Space Complexity Gnome Sort (Geeks For Geeks, 2021)

Gnome Sort is not an effective sorting algorithm. As there is no nested loop there would be reason to assume that the time complexity is linear. However, this is not the case. This is because the variable in the index does not always get increased and in some cases is decreased. Like Bubble Sort and Cocktail Sort, Gnome Sort works best if the array is already or partially sorted. (Geeks For Geeks, 2021)

Gnome Sort is an in-place algorithm, so the space complexity is O(1). (Geeks For Geeks, 2021)

Pseudocode of Gnome Sort

```
1 procedure gnomeSort(a[]):
2 pos := 0
3 while pos < length(a):
4    if (pos == 0 or a[pos] >= a[pos-1]):
5       pos := pos + 1
6    else:
7       swap a[pos] and a[pos-1]
8       pos := pos - 1
9
```

Figure 23 – Pseudocode for Gnome Sort (Geeks For Geeks, 2021)

In *Figure 23* we can see the pseudocode for Gnome Sort using a zero-based array. It is a simple algorithm, where the if statement progresses the position and the else enables the swap and the decrease in the index position. (Geeks For Geeks, 2021)

Implementation & Benchmarking

The application was developed in Java using Eclipse IDE for Java Developers. The initial objective of the project was to develop the code for the algorithms. Once this had been achieved the next objective was to develop a method that would allow for benchmarking. The goal was to have the application produce a table that gave the average sort time of each algorithm in respective to the size of the array that was to be sorted (See *Figure 24).*

Size	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
Bubble Sort	0.102	0.149	0.403	0.480	0.793	1.180	4.242	11.394	21.123	37.690	47.000	63.786	95.543
Selection Sort	0.012	0.059	0.201	0.406	0.672	0.998	4.518	12.229	20.606	35.831	54.218	67.590	93.471
Insertion Sort	0.012	0.060	0.200	0.408	1.270	1.178	4.618	8.946	19.700	36.823	49.433	64.449	91.628
Counting Sort	0.011	0.031	0.056	0.070	0.018	0.021	0.025	0.023	0.030	0.037	0.062	0.031	0.032
Merge Sort	0.027	0.029	0.061	0.097	0.098	0.089	0.184	0.285	0.381	0.486	0.596	0.701	1.568

Figure 24 – Desired Output Style (Carr, 2021)

Once the data had been gathered the next intention was to analyse the data and present the results of the benchmarking.

Java

The code that was used in the application is a result of using pseudocode and online resources. Any references used are included in the code and in an attached references text file. Unfortunately, I was unable to programme the application to run to the exact specification that was originally intended. Instead of producing a console output like in *Figure 24*, my application was run ten (10) times using an array of a certain size. These results were manually inputted into an Excel Spreadsheet and the average was determined. The array size was then increased, and the process repeated until all data was gathered. The excel file in which the data was entered will be included in the submission.

Console Output

The data below was manually inputted into Excel. This data was then used to generate the graph in *Figure 30.*

Date:	03/05/2021												
Bubble Sort													
Size	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
Test 1 (ms)	0.146	0.875	3.917	5.396	9.091	12.084	18.400	26.638	47.335	57.557	95.178	107.698	139.313
Test 2 (ms)	0.145	0.893	2.843	4.250	7.679	9.715	17.950	28.415	41.603	59.434	90.894	123.204	158.934
Test 3 (ms)	0.242	1.279	2.855	4.782	10.662	10.256	18.816	27.407	31.296	58.969	93.125	108.414	158.100
Test 4 (ms)	0.145	3.905	2.903	4.955	8.279	9.493	18.023	22.487	36.480	62.924	67.947	104.217	159.727
Test 5 (ms)	0.144	0.918	7.054	4.938	7.029	7.830	24.314	21.008	32.246	68.406	79.544	115.444	145.678
Test 6 (ms)	0.142	1.433	3.674	4.268	6.534	8.887	20.926	22.204	42.659	65.760	78.904	113.116	160.427
Test 7 (ms)	0.224	3.981	3.220	5.682	11.689	8.274	19.311	26.909	32.809	61.604	69.503	110.913	143.484
Test 8 (ms)	0.288	1.247	7.683	4.962	8.176	8.305	17.721	29.274	32.668	56.555	91.211	109.194	152.270
Test 9 (ms)	0.143	0.869	2.896	4.340	6.870	9.227	24.270	19.089	43.311	57.231	66.820	110.586	154.792
Test 10 (ms)	0.146	1.432	2.859	4.390	6.314	13.498	22.426	31.092	43.640	46.684	75.203	111.470	153.879
Avg (ms)	0.177	1.683	3.990	4.796	8.232	9.757	20.216	25.452	38.405	59.512	80.833	111.426	152.660

Figure 25 – Results for Bubble Sort

Date:	03/05/2021												
Merge Sort													
Size	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
Test 1 (ms)	1.348	2.875	5.987	4.806	8.664	6.100	11.506	12.210	15.545	16.762	23.734	35.496	34.317
Test 2 (ms)	1.302	2.801	5.589	4.571	4.541	5.591	7.242	12.760	15.003	21.499	29.234	33.370	37.113
Test 3 (ms)	1.361	5.559	5.645	4.710	6.880	8.820	7.341	14.557	13.968	18.428	28.881	33.422	38.821
Test 4 (ms)	1.355	2.596	5.127	4.970	5.872	6.996	7.304	14.613	15.106	18.884	22.671	27.253	49.308
Test 5 (ms)	3.196	3.019	6.709	7.049	6.621	8.299	9.822	16.336	15.647	20.924	24.595	29.827	32.249
Test 6 (ms)	1.311	2.821	4.229	9.135	5.821	7.823	10.044	10.123	16.801	19.554	33.010	32.309	37.422
Test 7 (ms)	1.559	5.528	4.716	5.982	4.747	5.278	7.703	10.054	15.160	19.907	29.840	34.899	34.024
Test 8 (ms)	2.525	3.340	3.416	4.835	6.789	5.541	7.478	10.753	19.515	21.194	31.603	34.399	39.135
Test 9 (ms)	1.624	2.773	6.384	6.850	6.039	7.326	7.274	16.964	17.836	19.220	25.790	30.839	30.478
Test 10 (ms)	1.540	2.821	4.365	4.755	5.380	7.751	7.250	9.887	16.017	20.884	30.659	35.706	34.261
Avg (ms)	1.712	3.413	5.217	5.766	6.135	6.952	8.296	12.826	16.060	19.726	28.002	32.752	36.713

Figure 26 – Results for Merge Sort

Date:	03/05/2021												
Counting Sort													
Size	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
Test 1 (ms)	0.128	0.263	0.281	0.656	0.317	0.388	0.782	1.808	1.509	3.494	4.332	5.068	3.982
Test 2 (ms)	0.134	0.548	0.374	0.267	0.629	0.694	0.768	1.083	1.389	1.702	4.637	4.184	5.865
Test 3 (ms)	0.142	0.792	0.312	0.622	0.435	0.372	1.181	1.348	1.543	1.835	3.332	5.386	5.339
Test 4 (ms)	0.133	0.243	0.319	0.368	0.377	0.395	1.059	1.507	1.312	1.882	4.350	6.572	4.895
Test 5 (ms)	0.167	0.309	0.285	0.385	0.322	0.734	0.835	1.051	1.814	1.474	4.777	5.280	4.928
Test 6 (ms)	0.366	0.332	0.531	1.171	0.472	0.726	0.740	1.060	1.352	2.419	4.500	4.725	5.030
Test 7 (ms)	0.164	0.342	0.310	0.364	0.318	0.439	1.245	1.117	1.814	1.582	3.428	5.660	6.880
Test 8 (ms)	0.142	0.250	0.215	0.290	0.527	0.387	0.780	1.952	1.395	1.475	3.146	4.601	5.310
Test 9 (ms)	0.160	0.418	0.285	0.667	0.384	0.396	1.007	1.062	2.467	4.040	2.786	2.004	4.508
Test 10 (ms)	0.134	0.819	0.270	0.366	0.696	0.397	0.778	1.420	2.405	4.064	4.425	7.857	3.771
Avg (ms)	0.167	0.432	0.318	0.516	0.448	0.493	0.917	1.341	1.700	2.396	3.971	5.134	5.051

Figure 27 – Results for Counting Sort

Date:	05/05/2021												
Cocktail Sort													
Size	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
Test 1 (ms)	0.470	0.720	2.855	4.996	5.954	8.854	18.614	29.432	45.409	47.058	74.523	98.406	128.641
Test 2 (ms)	0.121	0.738	2.795	7.097	5.919	11.608	21.478	22.963	34.056	47.359	69.929	87.639	145.909
Test 3 (ms)	0.173	1.160	3.450	6.312	8.759	8.230	23.542	32.730	35.625	56.132	68.724	83.644	129.715
Test 4 (ms)	0.109	0.685	2.661	4.214	8.134	11.125	19.067	27.500	40.899	50.482	78.404	111.686	117.377
Test 5 (ms)	0.201	1.221	2.704	5.665	5.773	10.512	20.223	26.038	39.342	46.259	79.662	93.004	123.651
Test 6 (ms)	0.198	0.730	3.581	6.222	10.595	9.681	27.989	29.283	41.390	46.733	77.757	86.328	140.297
Test 7 (ms)	0.188	1.063	2.830	4.789	6.517	11.150	17.303	30.768	37.381	56.157	77.135	89.878	124.269
Test 8 (ms)	0.356	0.699	2.720	3.723	6.025	9.640	23.654	22.894	41.384	45.587	76.102	88.724	104.972
Test 9 (ms)	0.126	0.726	2.753	4.102	7.878	11.136	24.936	30.038	40.740	45.225	76.475	86.201	111.034
Test 10 (ms)	0.112	1.059	2.722	5.697	7.237	9.035	27.836	36.557	40.626	46.384	66.944	97.190	121.091
Avg (ms)	0.205	0.880	2.907	5.282	7.279	10.097	22.464	28.820	39.685	48.738	74.565	92.270	124.696

Figure 28 – Results for Cocktail Sort

Date:	05/05/2021												
Gnome Sort													
Size	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
Test 1 (ms)	0.654	0.797	2.560	3.930	6.999	7.593	7.539	14.079	24.083	30.385	36.503	52.791	65.007
Test 2 (ms)	0.144	0.881	2.857	4.064	5.064	5.589	9.614	17.079	23.560	25.878	40.711	53.301	62.642
Test 3 (ms)	0.182	2.221	4.028	4.924	6.496	5.751	11.416	13.683	25.221	30.673	38.490	45.959	70.691
Test 4 (ms)	0.148	0.836	4.670	7.478	9.421	5.569	7.717	13.670	19.558	28.569	37.481	51.291	78.715
Test 5 (ms)	0.158	0.815	2.413	5.574	9.806	7.989	10.028	19.622	22.227	25.754	36.288	44.943	100.393
Test 6 (ms)	0.177	0.887	2.411	5.260	5.208	6.242	12.059	12.793	21.966	28.791	36.583	48.372	77.912
Test 7 (ms)	0.168	0.875	2.958	6.472	5.128	8.132	13.301	13.097	24.827	31.586	37.444	47.722	84.957
Test 8 (ms)	0.170	1.517	2.448	5.665	6.605	8.508	7.532	15.121	22.115	26.941	36.380	48.867	85.782
Test 9 (ms)	0.185	0.728	2.453	4.231	5.764	8.960	9.433	18.329	21.975	29.789	37.868	51.366	62.051
Test 10 (ms)	0.156	0.785	2.833	4.968	7.290	8.165	11.450	13.865	21.315	29.368	53.433	48.023	72.077
Avg (ms)	0.214	1.034	2.963	5.257	6.778	7.250	10.009	15.134	22.685	28.773	39.118	49.263	76.022

Figure 29 – Results for Gnome Sort

Analysis

The data in *Figure 29* shows the time complexity for each of the algorithms used in the project. When you compare these curves against the Cheat Sheet in *Figure 31* you can determine what the Big-O Notation was for the data generated by the sorting algorithms. These results are shown in *Table 10*.

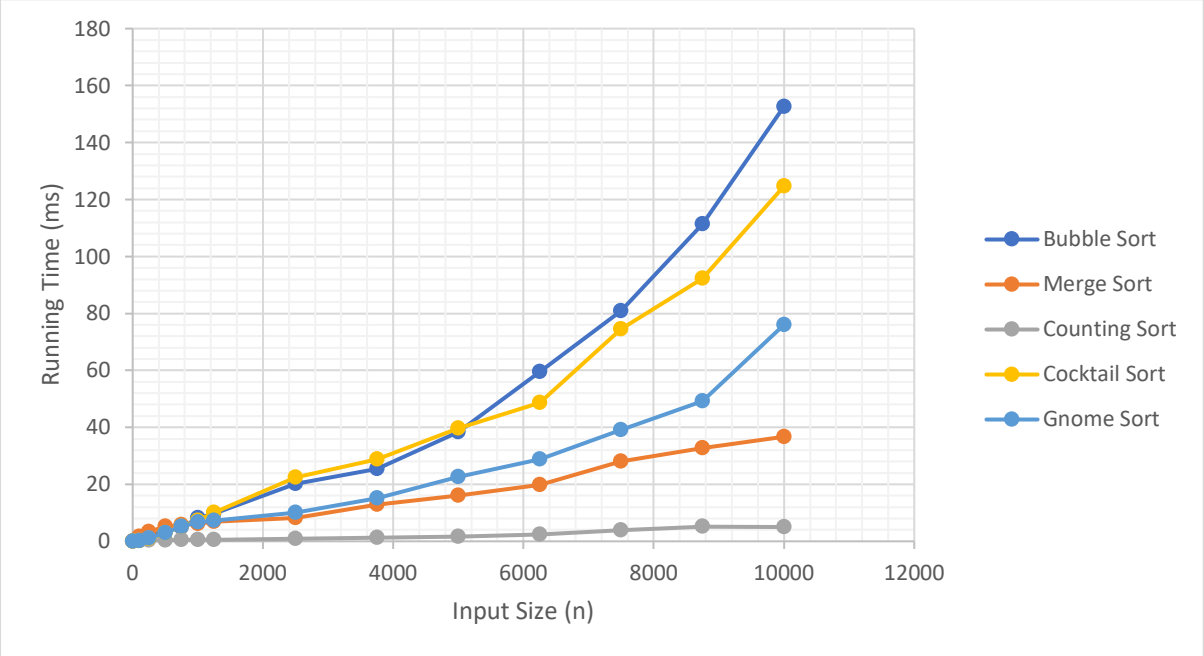

Figure 30 – Time Complexity of Each Sorting Algorithm

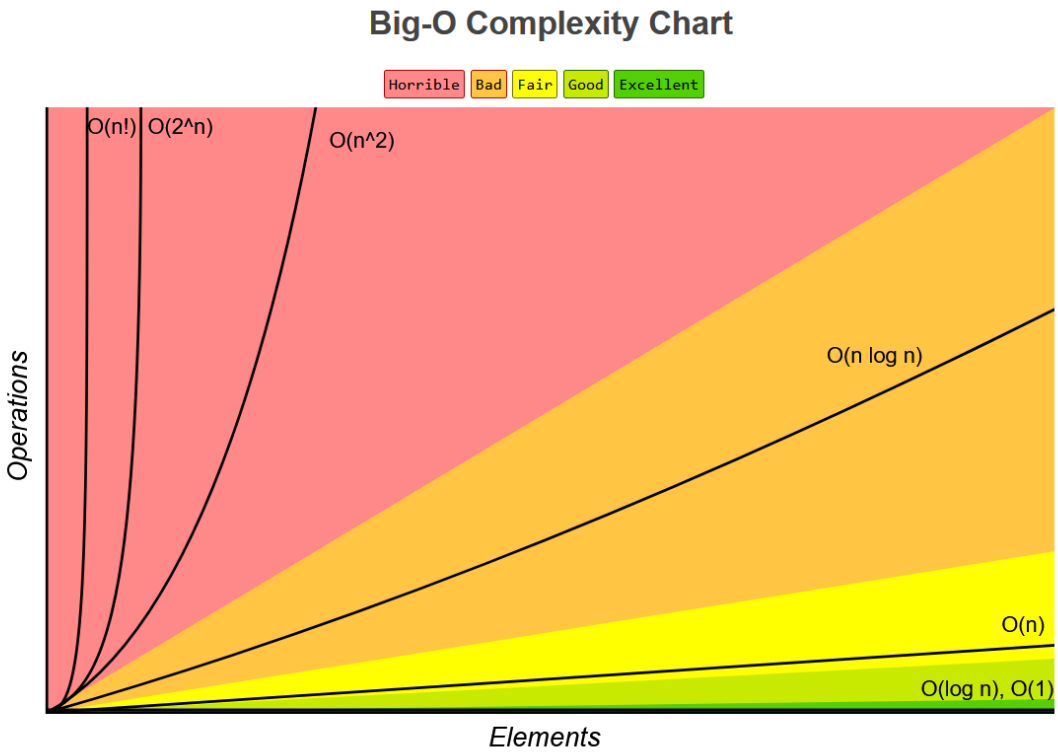

Figure 31 – Big-O Cheat Sheet (Big-O Cheet Sheet, n.d.)

Sorting Algorithm	Worst Case Time Complexity	Best Case Time Complexity	Average Time Complexity	Big-O According to Results and Cheat Sheet	Space Complexity
Bubble Sort	$O(n^2)$	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Merge Sort	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Count Sort	O(n+k)	O(n+k)	O(n+k)	O(n)	O(n+k)
Cocktail Sort	$O(n^2)$	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Gnome Sort	$O(n^2)$	$O(n^2)$	O(n)	O(n)	O(1)

Table 10 – Combined Big O Table

Conclusion

This project looks at five (5) different sorting algorithms with the objective of using them in an application and analysing the results. The algorithms used were Bubble Sort, Merge Sort, Counting Sort, Cocktail Sort and Gnome Sort. Project results indicate that Count Sort was the best performing and Bubble Sort was the worst. The collection of data is in line with convention, and the trends produced are what was to be expected. Although the application was not to the specification that I originally intended, I do feel that the data generated was beneficial in improving one's knowledge of sorting algorithms.

Table of Tables

Table of Figures

References

Big-O Cheet Sheet, n.d. *Big-O Cheat Sheet.* [Online]
Available at: https://www.bigocheatsheet.com/
[Accessed 12 May 2021].

Cake Labs, 2021. *Counting Sort.* [Online]
Available at: https://www.interviewcake.com/concept/java/counting-sort
[Accessed 10 May 2021].

Carr, D., 2021. *CTA Project.* s.l.:s.n.

Elkahlout, A. H. & Maghari, A. Y., 2017. A Comparative Study of Sorting Algorithms Comb, Cocktail and Counting Sort. *International Research Journal of Engineering and Technology,* 04(01), pp. 1387 - 1390.

Geeks for Geeks, 2018. *Sorting Algorithms.* [Online]
Available at: https://www.geeksforgeeks.org/sorting-algorithms/
[Accessed May 2021].

Geeks For Geeks, 2020. *Cocktail Sort.* [Online]
Available at: https://www.geeksforgeeks.org/cocktail-sort/
[Accessed 09 May 2021].

Geeks For Geeks, 2021. *Counting Sort.* [Online]
Available at: https://www.geeksforgeeks.org/counting-sort/
[Accessed 10 May 2021].

Geeks For Geeks, 2021. *Gnome Sort.* [Online]
Available at: https://www.geeksforgeeks.org/gnome-sort-a-stupid-one/
[Accessed 10 May 2021].

Geeks for Geeks, 2021. *Space Complexity.* [Online]
Available at: https://www.geeksforgeeks.org/g-fact-86/
[Accessed 08 May 2021].

Growing With The Web, 2016. *Cocktail Sort.* [Online]
Available at: https://www.growingwiththeweb.com/2016/04/cocktail-sort.html
[Accessed 10 May 2021].

Harel, D., 2004. *Algorithmics: The Spirit of Computing.* s.l.:Addison-Wesley.

Heineman, G. T., Pollice, G. & Selkow, S., 2016. *Algorithms in a Nutshell.* s.l.:O'Reilly.

Hinklemann, F., 2019. *Medium.* [Online]
Available at: https://medium.com/free-code-camp/my-favorite-linear-time-sorting-

algorithm-f82f88b5daa1
[Accessed May 2021].

Hurna, 2021. *Cocktail Sort.* [Online]
Available at: https://hurna.io/academy/algorithms/sort/cocktail.html
[Accessed 11 May 2021].

Isaac Computer Science, 2021. *Complexity.* [Online]
Available at: https://isaaccomputerscience.org/topics/complexity
[Accessed 08 May 2021].

Isaac Computer Science, 2021. *Isaac Computer Science.* [Online]
Available at: https://isaaccomputerscience.org/concepts/dsa_search_bubble
[Accessed 07 May 2021].

Isaac Computer Science, 2021. *Merge Sort.* [Online]
Available at: https://isaaccomputerscience.org/concepts/dsa_search_merge
[Accessed 9 May 2021].

Lukic, M., n.d. *Stack Abuse.* [Online]
Available at: https://stackabuse.com/bubble-sort-and-cocktail-shaker-sort-in-javascript/
[Accessed 10 May 2021].

Mangal, P., 2017. *CodingGeek.* [Online]
Available at: https://www.codingeek.com/algorithms/counting-sort-explanation-pseudocode-and-implementation/
[Accessed 10 May 2021].

Mannion, P., 2021. *GMIT.* [Online]
Available at:
https://learnonline.gmit.ie/pluginfile.php/295818/mod_resource/content/0/03%20Analysing%20Algorithms%20Part%201.pdf
[Accessed May 2021].

Mannion, P., 2021. *GMIT.* [Online]
Available at:
https://learnonline.gmit.ie/pluginfile.php/297433/mod_resource/content/0/07%20Sorting%20Algorithms%20Part%201.pdf

MIT, 2021. *Big O Notation.* [Online]
Available at: https://web.mit.edu/16.070/www/lecture/big_o.pdf

Open Genus, 2021. *Gnome Sort.* [Online]
Available at: https://iq.opengenus.org/gnome-sort/
[Accessed 10 May 2021].

Oxford, 2021. *Dictionaries.* [Online]
Available at:

https://www.oxfordlearnersdictionaries.com/definition/english/sort_2?q=sorting
[Accessed 6 May 2021].

Oxford, 2021. *Oxford Learner's Dictionaries.* [Online]
Available at:
https://www.oxfordlearnersdictionaries.com/definition/english/algorithm?q=algorithm
[Accessed 06 May 2021].

Point Tutorials, 2021. *Merge Sort.* [Online]
Available at:
https://www.tutorialspoint.com/data_structures_algorithms/merge_sort_algorithm.htm
[Accessed 11 May 2021].

Programiz, 2021. *Counting Sort.* [Online]
Available at: https://www.programiz.com/dsa/counting-sort
[Accessed 10 May 2021].

Srivastava, P., 2020. *Baeldung.* [Online]
Available at: https://www.baeldung.com/cs/stable-sorting-algorithms
[Accessed 07 May 2021].

Study Tonight, 2021. *Merge Sort Algorithm.* [Online]
Available at: https://www.studytonight.com/data-structures/merge-sort
[Accessed 9 May 2021].

Study Tonight, 2021. *Study Tonight.* [Online]
Available at: https://www.studytonight.com/data-structures/bubble-sort
[Accessed 09 May 2021].

Techopedia, 2021. *Sorting Algorithm.* [Online]
Available at: https://www.techopedia.com/definition/33090/sorting-algorithm
[Accessed 06 May 2021].

Woltmann, S., 2020. *HappyCoders.* [Online]
Available at: https://www.happycoders.eu/algorithms/sorting-algorithms/
[Accessed May 2021].