Computational Thinking and Algorithms Project 2019

Table of Contents

Computational Thinking and Algorithms Project 2019 1
Introduction 2
Sorting Algorithms 8
Bubble Sort 8
Merge Sort 12
Counting Sort 15
Tim Sort 18
Bogo Sort 25
Summary – Specifications of the 5 Chosen Sorting Algorithms 32
Implementation & Benchmarking 28
References 33

Introduction

The aim of this project was to gain a greater knowledge of the capabilities of different Sorting Algorithms that are present and in use today. During the lecture series in the 'Computational Thinking and Algorithms' module we were given information on several sorting algorithms, laying a basic foundation on the topic. However, before I progress with this project I would like to carry out a greater in-depth research on the sorting algorithms that I have chosen to compare. This will allow me to have a better understanding of their functionality and will give me prior knowledge of their existing performance before I progress and test them myself.

Firstly , to address the concept of what sorting is: To sort something refers to 'putting a similar group of things in a particular order or separate them into groups according to a set principle' (Dictionary.cambridge.org, 2019).

Images Depicting Simple Sorting. (Gudpict.pw, 2019), (Gudpict.pw, 2019), (Bandalou, 2019).

As seen with the simple children's sorting toys above, humans have the capability to perform sorting tasks intuitively and with ease. This is not the case in computing. A computer program has to follow a sequence of exact instructions to accomplish the same result. To implement sorting in a computer-based environment we make use of algorithms.

If we apply the same concept to an algorithm as we did previously when defining what the term 'sort' means, ' *A sorting algorithm is a method that has the ability to reorganizing a large number of items into a specific order according to pre-defined ordering rules.' (WhatIs.com, 2019).*

The Purpose of Sorting Algorithms in Computing.

Originally sorting was one of the most fundamental algorithmic problem scientists faced in early computing. Research was centred on finding the most efficient ways to sort a set of data. Close on 25% of all CPU cycles are spent sorting (Cs.cmu.edu, 2019). The reason sorting is such an important problem in computing, is that computational tasks and processes are simplified if the data being analysed is sorted prior to their initiation. If the data is sorted prior to the processing step, it can result in:

- **Easier accessibility** to specific data needed e.g. it would be much harder to find a name in a phonebook if the data set was not ordered alphabetically.
- **Quicker run times** i.e. if the sample data set is ordered it is easier to find the min, max and median of the data set.
- Aids in **time estimate** i.e. it could influence the running times of certain applications, processes.

Types of sorting algorithms.

There are several types of sorting algorithms. The main 3 types include:

1. ***Comparison***
   Comparison sorting algorithms make the use of comparison operators (comparator functions +, -, =, <, > etc.) which have the capability of determining which element of two should appear first in a sequence (Mannion, 2019). These type of algorithms make no assumptions about the data set being analysed and compare all elements with one and another. Some examples of comparison sorting algorithms include Bubble sort, Insertion sort, Merge sort and Quick sort. Any algorithm of this type cannot do better than nlogn performance in best average and worst case.

2. ***Non-Comparison*** (also known as Linear sorting algorithms as they can run at time O(n) time).
   Non- Comparison sorting algorithms make prior assumptions about the data input that is being sorted (e.g. distribution etc), allowing such algorithms to examine bits of keys to sort the input data set being analysed (Sitedownrightnow.com, 2019). As this type of algorithm can avoid comparing each element of input n, it is quicker and is seen to be a huge improvement when sorting larger input sizes. Some examples include Counting sort and Bucket sort.

3. ***Hybrid***
   Hybrid sorting algorithms refer to the combination of two or more sorting algorithms favouring the strengths of each algorithms to achieve a superior hybrid algorithm. Some examples include Timsort and Introsort.

When is the data said to be sorted?

'A collection of items is deemed to be "sorted" if each item in the collection is less than or equal to its successor.' (Mannion, 2019).

From this above statement, I have interpreted it in such way that for a data set that is said to be sorted, according to pre-defined rules, this dataset must be reorganised in a way that displays each value ordered so that they are in a sequence whereby a lesser value is before a value that has a greater magnitude. If there are duplicate values (elements of the same value) within the dataset being sorted, these duplicates must appear in one continues block. It is also very important to note that the data set being sorted must contain the same contents it did before carrying out the sorting algorithm.

An example of this can be seen through the use of a simple random array of elements. In Array, A, its elements consist of **A [b, a, c, c] :**

1. Starting from the left-hand side of the array at index 0, It is seen that element **A[b] > A[a]** as element b > a. The array is rearranged to show this: **A[a, b]**.
2. **A[a, b] < A[c]** is then sorted as element a & b < c. Reordering the array to **A[a, b, c].**
3. Finally, elements at index 2 and 3 are sorted, as **A[c] = A[c]** are the same they are placed one after the next. Leading to the array being "sorted" i.e. Result: **A[a, b, c, c].**

What is involved in sorting?

To sort a given data set we need to focus on what way we want the data to be ordered and also how this is carried out. To achieve this, we need:

- A Sort Key i.e. "*pre-defined ordering rules.' (WhatIs.com, 2019).*
- Inversions.

A *sort key* refers to the information that the algorithm uses to carry out comparisons between the unsorted data. Without the sort key the sorting algorithm would not be able to run, as it would not have its exact instructions needed to carry out the way in which the user wants the data set sorted *(WhatIs.com, 2019)*.. Some examples of sort key types include keys associated with numerical ordering 'comparator functions' and alphanumerical ordering e.g. lexicographical ordering - ordering characters and strings.

An *inversion* is a mathematical term to describe a process used in sorting algorithms to order a sequence of data that is out of order in relation to the sort key. By determining the number of inversions required to sort a data set one can roughly estimate to what degree that specific input instance is from being sorted. Thus, the number of inversions can be a 'measure of the amount of disorder in a permutation' (Leeuwen, 1990).

Criteria for choosing a sorting algorithm i.e. desirable properties and the motivation for benchmarking.

'Sorting algorithms' oftentimes are not suitable for every sorting task they are given i.e. there is not a single sorting algorithm that works ambiguously – 'All algorithms are not created equally' (Mannion, 2019). Instead generally each algorithm is more suited to a specific task, with specific variables and needs e.g. specific input size-range, stability, run-time etc. For this reason, it is very important to know all the strengths and weaknesses associated with your chosen sorting algorithm prior to its use in any given process.

For the reasons above there is a need to benchmark ('posteriori analysis') your chosen algorithm before it is used. This 'empirical method' allows us to identify if it is a suitable sorting algorithm for the sorting task at hand (Mannion, 2019). By evaluating the sorting algorithm beforehand, we have a greater idea of its strengths and weaknesses when it is in use.

Sorting algorithms can be evaluated under the following headings:

Suitability

Determines if the specific strengths and weaknesses of the sorting algorithm are well matched to the expected class of input instances and requirements.

Stability

Refers to a situation where a sorting operation occurs and the relative order of elements that are considered equal upon input is preserved not altered i.e. if the input is already slightly pre-sorted, the algorithm should preserve this sorting and not start from scratch.

Performance:

The sorting algorithm being used should be suitable in for use on the specific computer running the sorting algorithm application e.g. 'speed of the computer and quality of the compiler'. Some algorithms only use a fixed amount of working space regardless of the input size. Algorithms that use this sort of memory usage are called ***in-place.*** (Mannion, 2019)

External factors:

This can refer to many external factors such as algorithms running in external memory or the size of the input n being tested by the sorting algorithm i.e. how many items need to be sorted.

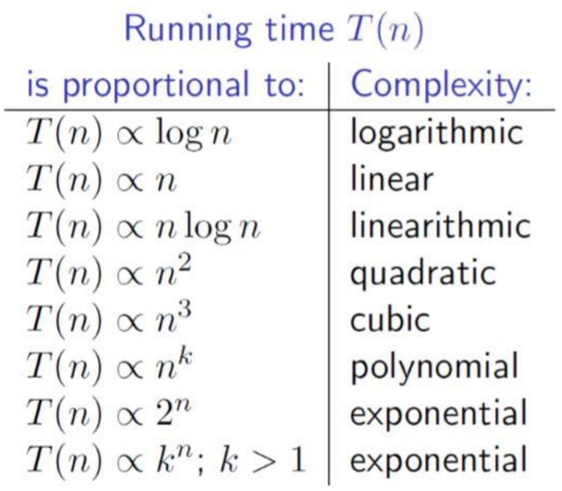
Efficiency/Complexity:

Space and Time efficiency are the key variables looked at when evaluating efficiency of the algorithm.

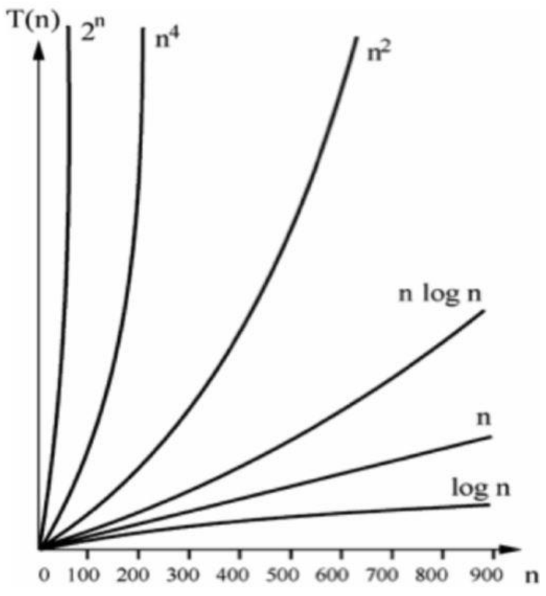
Space efficiency looks at the computer's ability to run the algorithm in terms of the computational memory and space available.

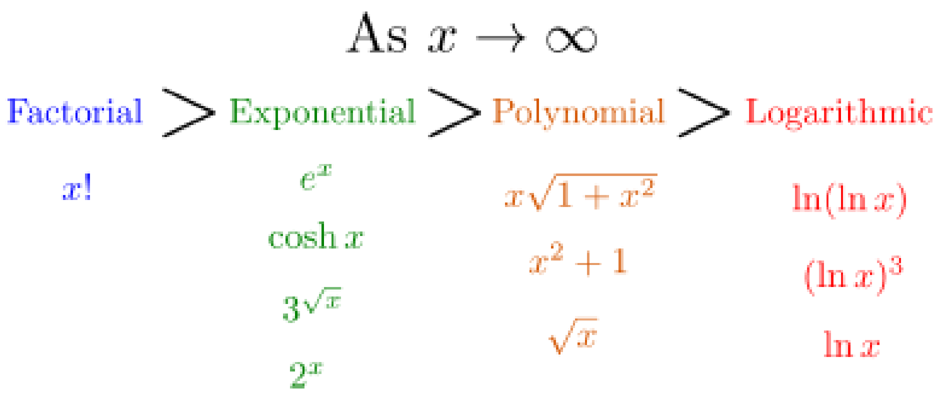
Time efficiency is evaluated by considering the length of time required to run the actual algorithm. Algorithms can be compared to each other by evaluating their running time. By analysing this, one can see if the algorithms scale up well with an increased input size n. Algorithm complexity results in the algorithm falling into a certain order family which is determined by 'the growth in execution time with respect to an increasing input size of n' (Mannion, 2019). To deduce which sorting algorithm has the optimal time frame for a specific input one must identify the most expensive computation within the algorithm to determine its classification i.e. look at the algorithms Worst case run time.

Decreasing in efficiency



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





As $x \to \infty$

Factorial >	Exponential >	Polynomial >	Logarithmic
$x!$	e^x	$x\sqrt{1+x^2}$	$\ln(\ln x)$
	$\cosh x$	x^2+1	$(\ln x)^3$
	$3^{\sqrt{x}}$	$\sqrt{x}$	$\ln x$
	2^x		

Images Depicting Running times in relation to Complexity (Mannion, 2019),(Sitedownrightnow.com, 2019).

We have three cases when analysing the complexity of an algorithm – these all can be described by Big O Notation(also known as asymptotic behaviour notation). Big O notation measures how a function grows and declines *(Mannion, 2019)*.

Best case – refers to completing the algorithm in the fastest runtime. In reality this usually doesn't occur. An example of this includes data that is already sorted (Sciencing, 2019). Usually denoted by Ω (Omega) notation and displays a 'linear growth in execution'. (Mannion, 2019).

Average case – As all possible inputs are equally likely, the average case refers to the arithmetic mean of all these possible scenarios. The algorithm is run several times using many different input sizes n. The total time is computed by adding all individual times together and this total is then divided by the number of trials (Liss, shrivastava and Mihalcin, 2019). Usually denoted by θ (theta) notation e.g. (Mannion, 2019).

Worse case – refers to the worst runtime possible ('max runtime' that would occur when using the selected sorting algorithm) i.e. the slowest time to complete the algorithm (GeeksforGeeks, 2019). This case occurs when the max number of operations are executed. Worst case analysis is usually carried out when comparing algorithms as it is an indication of the worst possible runtime that the algorithm is capable of. Usually denoted by Big O notation. (Mannion, 2019).

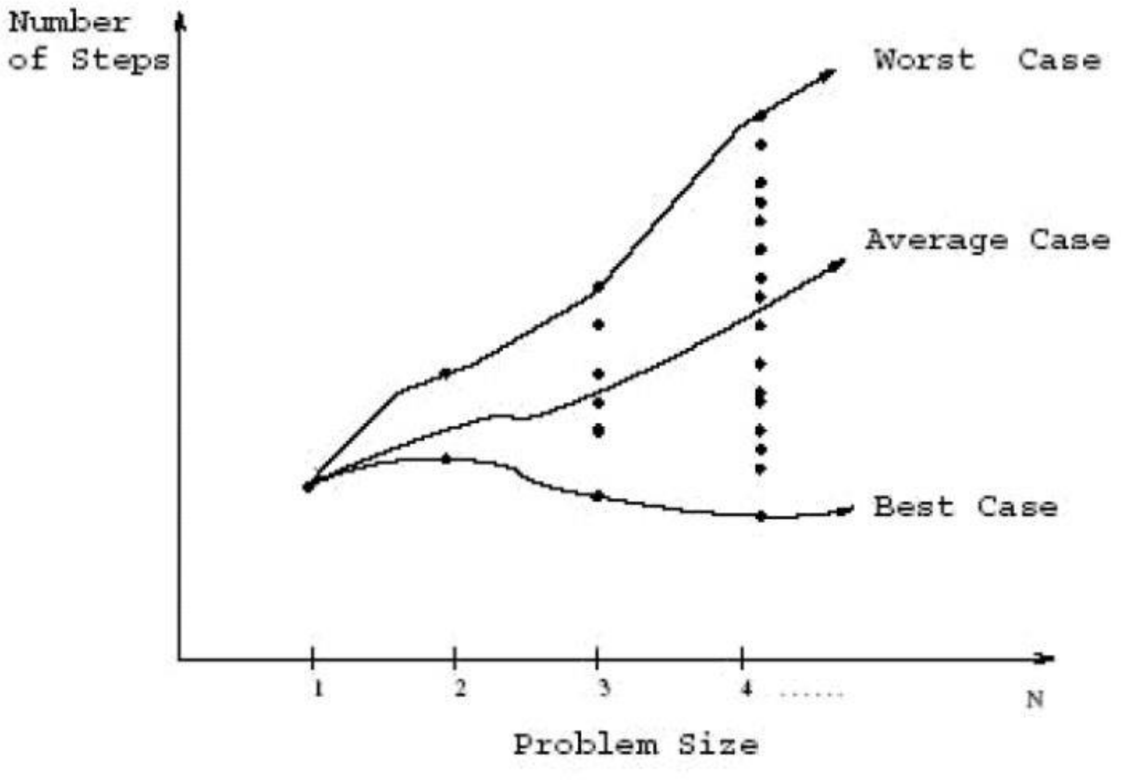


Image Depicting Running times in relation to complexity (Sparrowflights.blogspot.com, 2019).

Sorting Algorithms

This project focuses on evaluating the time efficiency of 5 chosen sorting algorithms. To carry out this benchmarking we were given the task to design and construct a Java application that would do this.

The 5 Sorting Algorithms I have chosen to benchmark in this Project include:

1. Bubble Sort
2. Merge Sort
3. Counting Sort
4. Insertion Sort
5. Bogo Sort

In this aspect of the project I will discuss each of the Sorting Algorithms separately under these topics:

- Introduction to the chosen Sorting Algorithm
- Its Space & Time Complexity
- How the Sorting Algorithm Works
- Small code example

Bubble Sort

Bubble Sort is a simple comparison sorting algorithm that was created in ‘1956. It was given the name Bubble sort to describe the way ‘larger elements bubble to the top of the list during the sorting process’ (En.wikipedia.org, 2019). It works by comparing adjacent elements to one and another. If the elements are not in order, the sorting algorithm ‘swaps each of the elements’ so that they are in the correct order (www.tutorialspoint.com, 2019). Thus, simply put Bubble Sort is basically based on the idea of repeatedly swapping pairs that have been compared to one and another until the entire input size has been sorted.

Specifications Big O Notation

Best Case	Average Case	Worst Case	Space Complexity (Sorting inplace)	Stable
O(n)	O(n^2)	O(n^2)	1	Yes

Table Depicting Bubble Sort Specs (GeeksforGeeks, 2019).

Best case runtime – Scenario where the input elements are already sorted prior to running the sorting algorithm

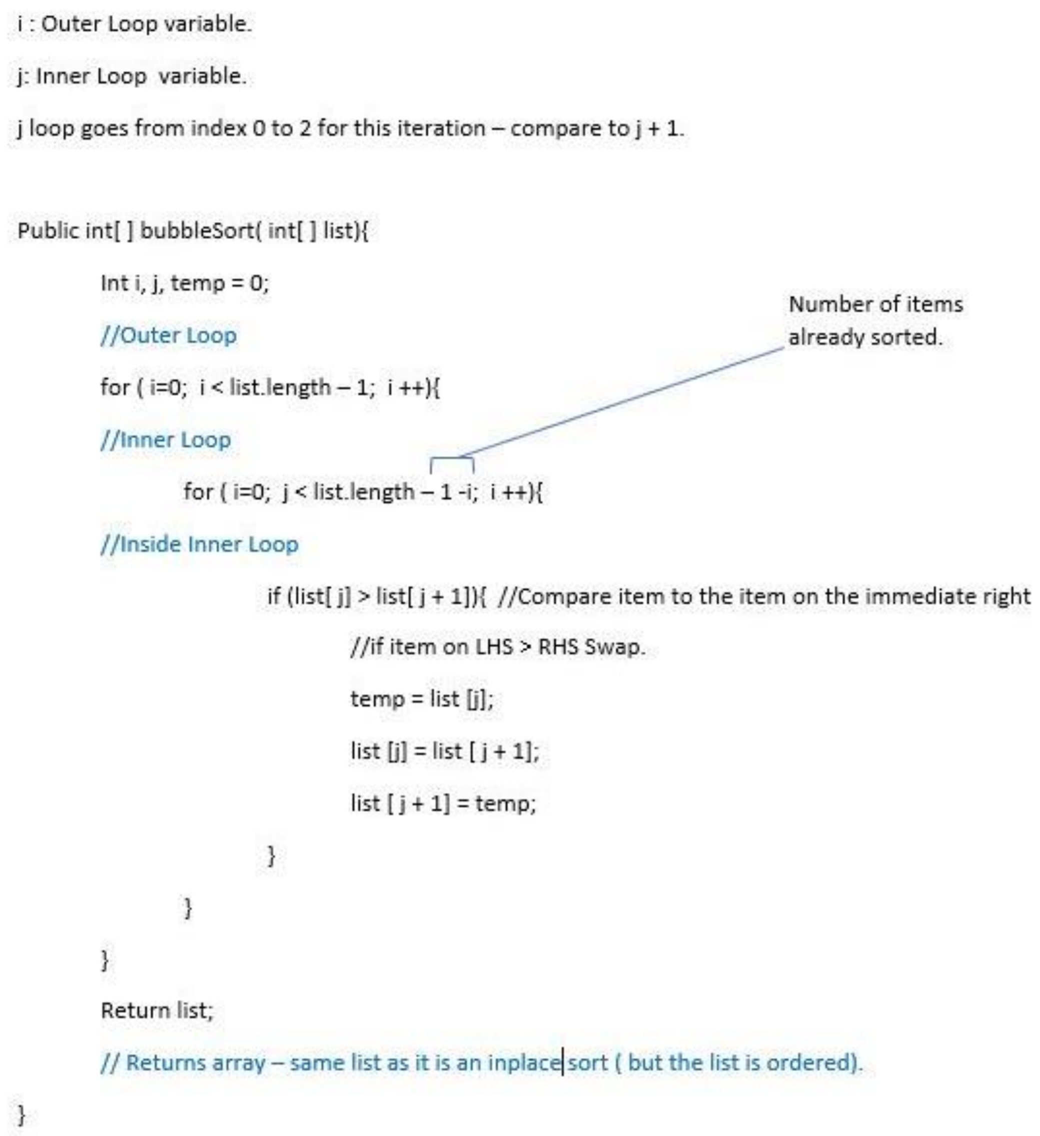
Average/ Worst case runtime - This would occur if the input array was in reverse order prior to sorting the input set.

Big O Analysis of Bubble Sort

Due to this sorting algorithms simplicity it would be a very slow and impractical choice of sorting algorithm for most problems, especially problems with greatly unsorted data and large input sizes (Mannion, 2019). The reason 'Bubble sort' is not as efficient as other sorting algorithms is it uses nested loops during each iteration. However, as seen in some circumstances with the best-case runtime, it would be suited in cases where the input data is nearly sorted or if there is a small input size left to be sorted e.g. It could be used in the end part of wrapping up of a Hybrid sorting algorithm.

Sample Bubble Sort Java Code

Code adapted from an example in: (YouTube - BubbleSort Algorithm, 2019)



i : Outer Loop variable.

j: Inner Loop variable.

j loop goes from index 0 to 2 for this iteration – compare to j + 1.

```
Public int[ ] bubbleSort( int[ ] list){

        Int i, j, temp = 0;

        //Outer Loop

        for ( i=0;  i < list.length – 1;  i ++){

        //Inner Loop

                for ( i=0;  j < list.length – 1 -i;  i ++){

        //Inside Inner Loop

                        if (list[ j] > list[ j + 1]){  //Compare item to the item on the immediate right

                                //if item on LHS > RHS Swap.

                                temp = list [j];

                                list [j] = list [ j + 1];

                                list [ j + 1] = temp;

                        }

                }

        }

        Return list;

        // Returns array – same list as it is an inplace sort ( but the list is ordered).

}
```

Diagram of Bubble Sort Working

Index #:	0	1	2	3	4	5
Input Data Sample:	5	8	1	6	9	2

Iteration 1: Largest element in the input bubbles to the right.

Steps:

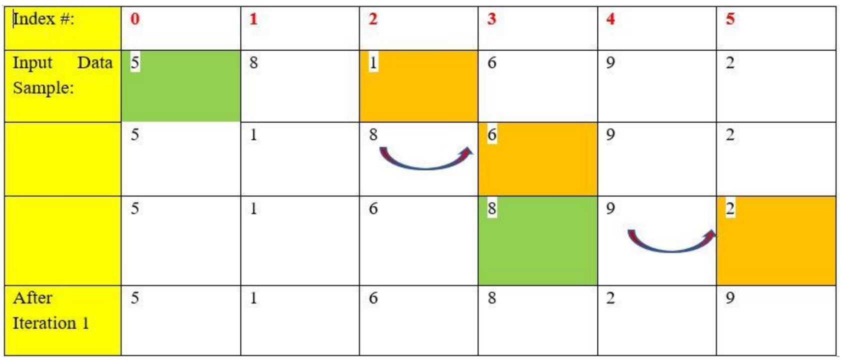
5 is less than 8 – no swap

8 is greater than 1 – swap

8 is greater than 6 - swap

8 is less than 9 - no swap

9 is greater than 2 - swap



Index #:	0	1	2	3	4	5
Input Data Sample:	5	8	1	6	9	2
	5	1	8	6	9	2
	5	1	6	8	9	2
After Iteration 1	5	1	6	8	2	9

Iteration 2: Largest element in the input bubbles to the right.

Steps:

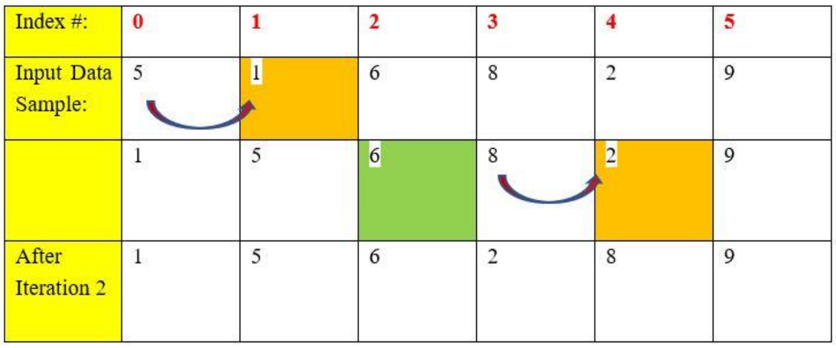
5 is greater than 1 – swap

5 is less than 6 - no swap

6 is less than 8 - no swap

8 is greater than 2 - swap

8 is less than 9 - no swap



Index #:	0	1	2	3	4	5
Input Data Sample:	5	1	6	8	2	9
	1	5	6	8	2	9
After Iteration 2	1	5	6	2	8	9

Iteration 3: Largest element in the input bubbles to the right..

Steps:

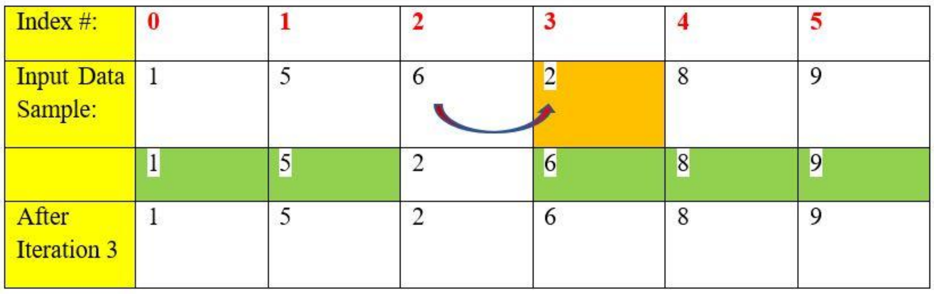
1 is less than 5 – no swap

5 is less than 6 - no swap

6 is greater than 2 – swap

6 is less than 8 - no swap

8 is less than 9 - no swap



Index #:	0	1	2	3	4	5
Input Data Sample:	1	5	6	2	8	9
	1	5	2	6	8	9
After Iteration 3	1	5	2	6	8	9

Iteration 4: Largest element in the input bubbles to the right.

Steps:

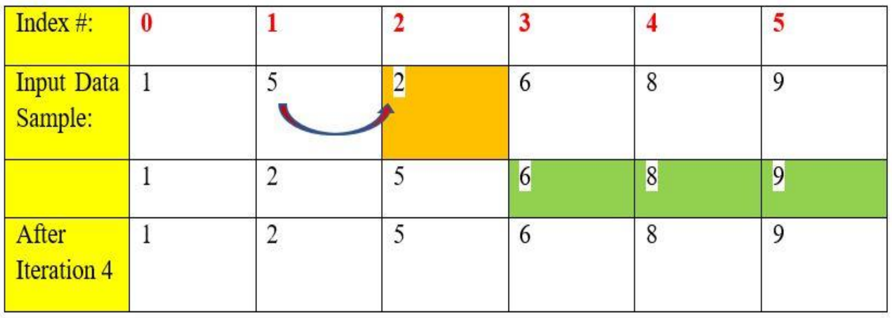
1 is less than 5 - no swap

5 is greater than 2 – swap

5 is less than 6 - no swap

6 is less than 8 - no swap

8 is less than 9 - no swap



Index #:	0	1	2	3	4	5
Input Data Sample:	1	5	2	6	8	9
	1	2	5	6	8	9
After Iteration 4	1	2	5	6	8	9

Iteration 5: Largest element in the input bubbles to the right.

Steps: *Largest element in the input bubbles to the right.*

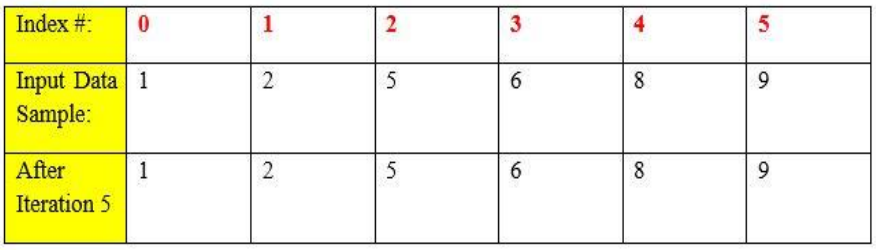
1 is less than 2 – no swap

2 is less than 5 - no swap

5 is less than 6 - no swap

6 is less than 8 - no swap

8 is less than 9 - no swap



Index #:	0	1	2	3	4	5
Input Data Sample:	1	2	5	6	8	9
After Iteration 5	1	2	5	6	8	9

Input data set is sorted: Complete.

Merge Sort

Merge Sort is a more-complex comparison sorting algorithm that was proposed by John von Neumann in '1945 (Mannion, 2019). It gets the name 'Merge' sort as the algorithm results in combining sorted lists together to make a new, sorted final list i.e. merging two sorted lists (Interactivepython.org, 2019) . This sorting algorithm uses a recursive 'Divide and Conquer' paradigm that works by dividing the input list in half into n number of sub-problems of the same related type. It sorts each of the sub-problems, and then recombines these sub lists through a merging process. Each sub-problem is repeatedly merged together until there is only one input list left leading to a completely sorted input (Khan Academy, 2019).

Specifications Big O Notation

Best Case	Average Case	Worst Case	Space Complexity (Sorting inplace)	Stable
n log n	n log n	n log n	0(n)	Yes

Table Depicting Merge Sort Specs (GeeksforGeeks, 2019).

Big O Analysis of Merge Sort

This sorting algorithm is a good 'all-round' recursive algorithm in terms of performance and runtime. In the above table, we can see that the best, average and worst case all have similar time complexity. This shows us the size n of the input data to be tested doesn't cause as much of a strain on this sorting algorithm i.e. Even with large data sets n log n (Mannion, 2019). It is important to note that it is highly favourable to have a time complexity of this scale in the worst-case runtime. It allows the user to easily predict max run times regardless of whether it is best, average or worst-case scenarios. This stable sorting algorithm ensures the same producibility each time it is run, however in terms of space complexity the algorithm takes up a lot of space which can lead to slower operations during runtime (Medium, 2019).

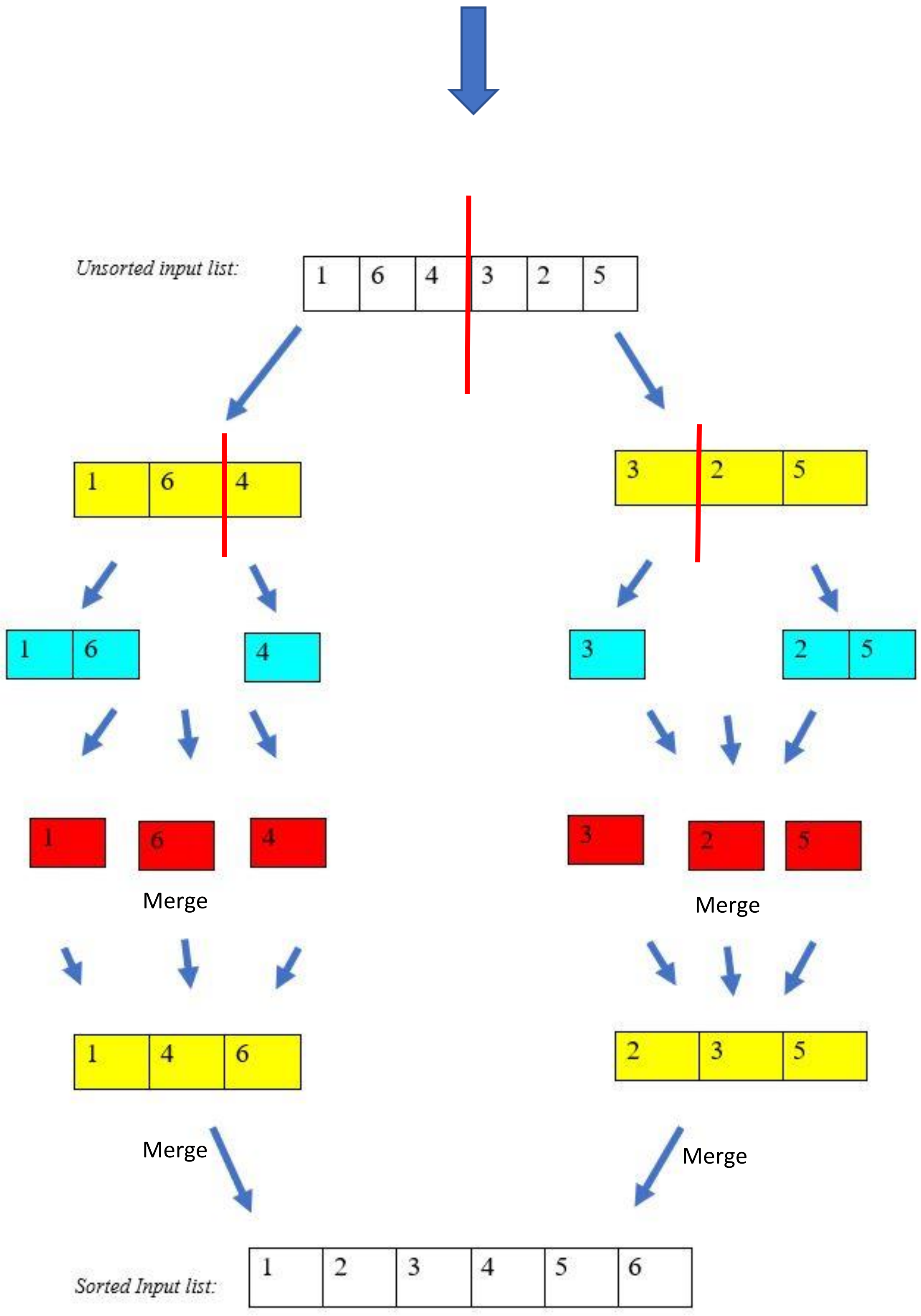
Diagram of Merge Sort Working

Steps of Merge Sort working:

1. The Input problem list is split in half i.e. Into two separate smaller lists.
2. The two separate lists are further simplified into smaller more sorted lists.
3. Elements are sorted and rearranged in order. The smaller lists begin to merge back to bigger lists.
4. The final two lists are merged to form one sorted list.

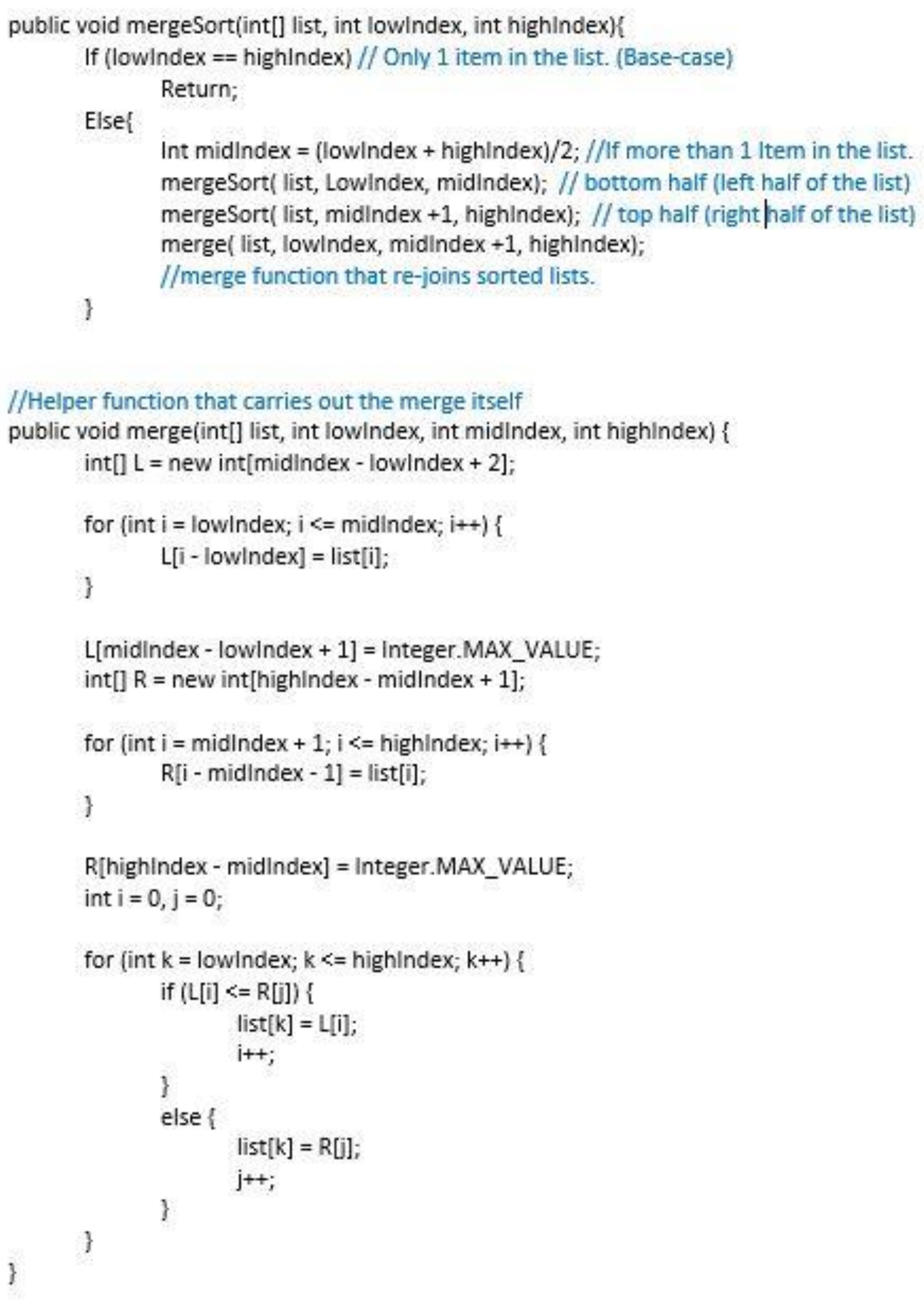
Input List of Integers:

1	6	4	3	2	5



Sample Merge Sort Java Code

Code adapted from example in: (YouTube - MergeSort Algorithm, 2019)



```
public void mergeSort(int[] list, int lowIndex, int highIndex){
        If (lowIndex == highIndex) // Only 1 item in the list. (Base-case)
                Return;
        Else{
                Int midIndex = (lowIndex + highIndex)/2; //If more than 1 Item in the list.
                mergeSort( list, LowIndex, midIndex);  // bottom half (left half of the list)
                mergeSort( list, midIndex +1, highIndex);  // top half (right half of the list)
                merge( list, lowIndex, midIndex +1, highIndex);
                //merge function that re-joins sorted lists.
        }


//Helper function that carries out the merge itself
public void merge(int[] list, int lowIndex, int midIndex, int highIndex) {
        int[] L = new int[midIndex - lowIndex + 2];

        for (int i = lowIndex; i <= midIndex; i++) {
                L[i - lowIndex] = list[i];
        }

        L[midIndex - lowIndex + 1] = Integer.MAX_VALUE;
        int[] R = new int[highIndex - midIndex + 1];

        for (int i = midIndex + 1; i <= highIndex; i++) {
                R[i - midIndex - 1] = list[i];
        }

        R[highIndex - midIndex] = Integer.MAX_VALUE;
        int i = 0, j = 0;

        for (int k = lowIndex; k <= highIndex; k++) {
                if (L[i] <= R[j]) {
                        list[k] = L[i];
                        i++;
                }
                else {
                        list[k] = R[j];
                        j++;
                }
        }
}
```

Counting Sort

Counting sort was invented by Harold H.Seward in 1954. Counting sort is a non-comparison sorting algorithm that is based on pre-defined assumptions i.e. input data falling into a certain range (Mannion, 2019). Due to these assumptions, input elements do not need to be compared with one and another in a manner similar to comparison sorting. The assumptions made with counting sort include the assumption that each element in the input has a non-negative int key with a range of k. Before the algorithm is run a key range k needs to be determined for the input data(GeeksforGeeks, 2019).

Specifications Big O Notation

Best Case	Average Case	Worst Case	Space Complexity (Sorting inplace)	Stable
n + k	n + k	n + k	n + k	Yes

Table Depicting Counting Sort Specs (GeeksforGeeks, 2019).

Big O Analysis of Counting Sort

Counting Sort's greatest strength is that it runs on Linear time ('*O(n)*') making it faster than most of the comparison-based sorting algorithms e.g. the comparison algorithm 'merge sort' has n log n (Mannion, 2019). This major strength in time efficiency is unfortunately shadowed by its a few of its other weaknesses. 'Counting sort' depends on pre-made assumptions, one being the potential value range of the input data that will be used in the algorithm (Counting Sort Algorithm, 2019). If this key range is not known the algorithm would not be suitable i.e. data may be out of range if it was run / it would not run as it would return an incorrect 'sorted' dataset . Also, If the range was very large for the input set being tested, using the counting sort algorithm would require too much space (possibly even larger than *O(n)*). These two features add a restriction with the use of Counting sort (GeeksforGeeks, 2019).

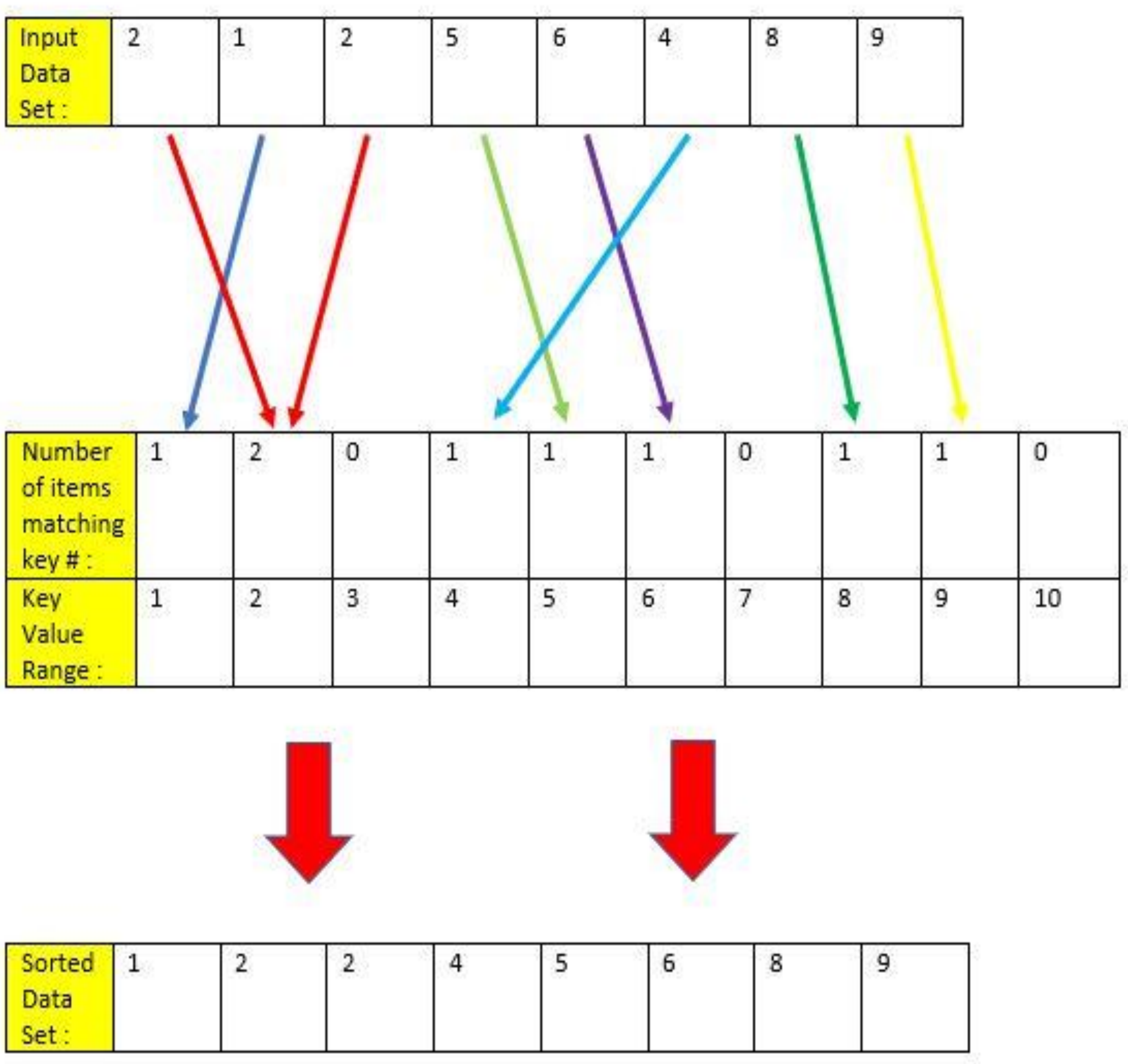
Diagram of Counting Sort Working

Steps of Counting Sort working:

1. Prior to running the 'Counting sort' algorithm, the key range is determined for the Input data set (Determined by the maximum value in the input array). In this case the key range is 1-10. Counting sort creates an empty bucket equivalent to each of the values

1-10 in range k. These empty buckets are used to count each element and organise the elements value in order.

2. The algorithm iterates through the input once. The first item is a 2, so one count of two is added to counts[2]. The next item is a 1, so one count of one is added to counts[1] and so on until this process has been applied to all elements of the input data.
3. When the end of the input data is reached, the total counts for each number in the input set is achieved. As the key range is in order the sorted list is then filled in.



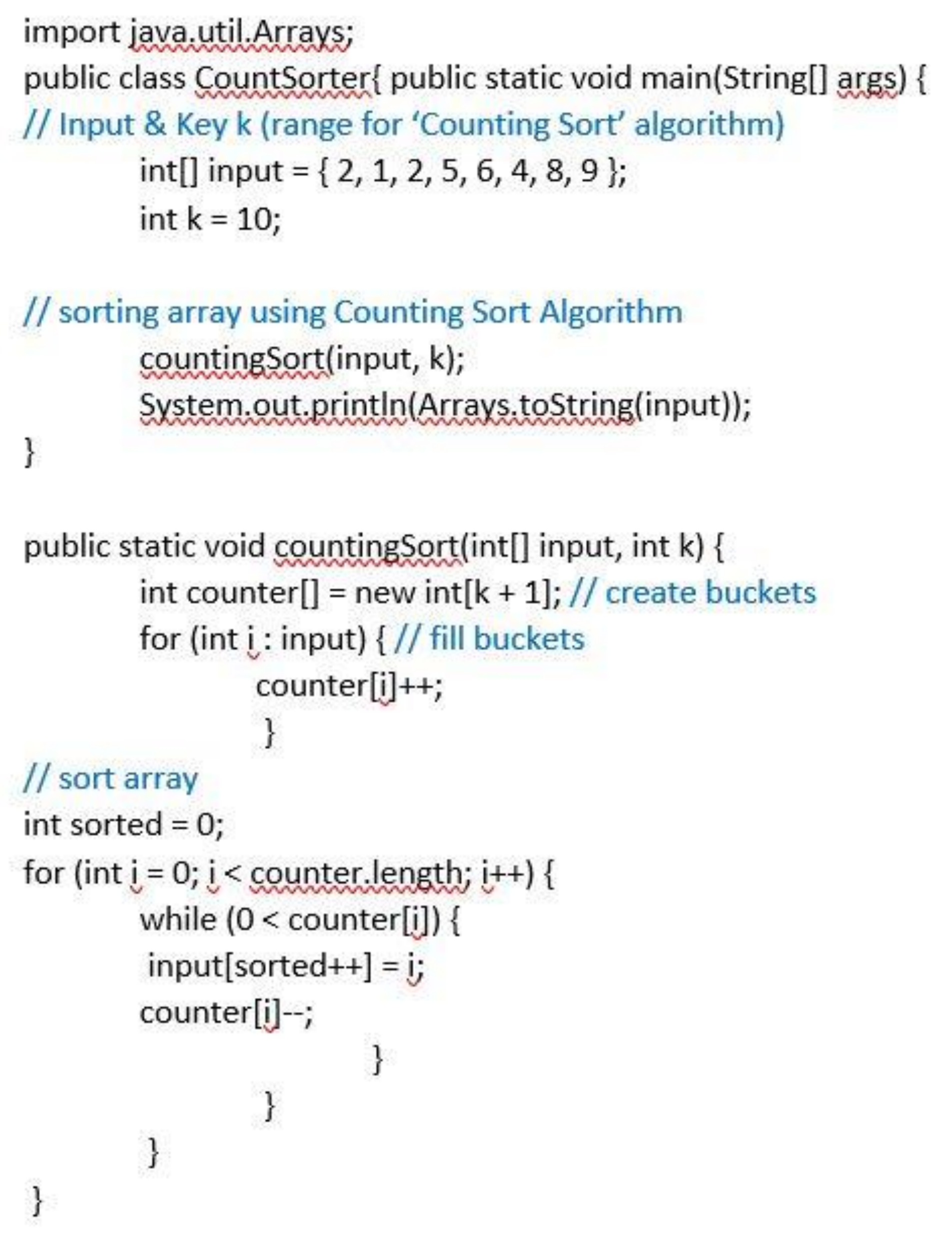
Input Data Set :	2	1	2	5	6	4	8	9

Number of items matching key # :	1	2	0	1	1	1	0	1	1	0
Key Value Range :	1	2	3	4	5	6	7	8	9	10

Sorted Data Set :	1	2	2	4	5	6	8	9

Sample Counting Sort Java Code

Code adapted from example in: (Javin, 2019)



```
import java.util.Arrays;
public class CountSorter{ public static void main(String[] args) {
// Input & Key k (range for 'Counting Sort' algorithm)
          int[] input = { 2, 1, 2, 5, 6, 4, 8, 9 };
          int k = 10;

// sorting array using Counting Sort Algorithm
          countingSort(input, k);
          System.out.println(Arrays.toString(input));
}

public static void countingSort(int[] input, int k) {
          int counter[] = new int[k + 1]; // create buckets
          for (int i : input) { // fill buckets
                    counter[i]++;
                    }
// sort array
int sorted = 0;
for (int i = 0; i < counter.length; i++) {
          while (0 < counter[i]) {
           input[sorted++] = i;
          counter[i]--;
                              }
                    }
          }
 }
```

Tim Sort

I had originally planned to discuss Tim Sort however I ended up having trouble with my java code and decided to include insertion sort instead as a replacement. I thought I would include this discussion on Tim Sort however as I enjoyed the research. Tim Sort is a hybrid sorting algorithm based on the combination of 'Insertion sort and Merge sort' (Mannion, 2019). It was created by Tim Peters in 2002 and is based off the idea that in real-world data sorting, some of the data is already sorted. This is an adaptive sorting algorithm as it identifies already-sorted data from unsorted data by using both sorting algorithms and returns a fully sorted version of the input set. The unsorted input data is divided into many parts of size between 32-64 items (as this is the recommended size for best case runtime in insertion sort) called 'Runs' which are then sorted by the insertion sorting algorithm. Following this these sorted runs are then merged to form a completely sorted version of the previously unsorted input (TimSort - GeeksforGeeks, 2019).

Specifications Big O Notation

Best Case	Average Case	Worst Case	Space Complexity (Sorting inplace)	Stable
n	n log n	n log n	n	Yes

Table Depicting Tim Sort Specs (GeeksforGeeks, 2019).

Big O Analysis of Tim Sort

The data set is firstly analysed to view what items are sorted and unsorted regardless of if it is in increasing or decreasing order. This cuts out unnecessary sorting that would be time consuming. Once this is established these sorted values are swapped around so as they are in order and all that is left is unsorted values (TimSort - GeeksforGeeks, 2019). These unsorted items are then broken into smaller chunks called 'Runs' that are sorted using insertion sort which runs very well on smaller numbered input sets (between 32-64 items has a time complexity of O(n^2)). This is quicker than n log n that is seen in merge sort. Once the smaller chunks are sorted, they are merged together with the already sorted data that was initially established, and the input data is said to be sorted.

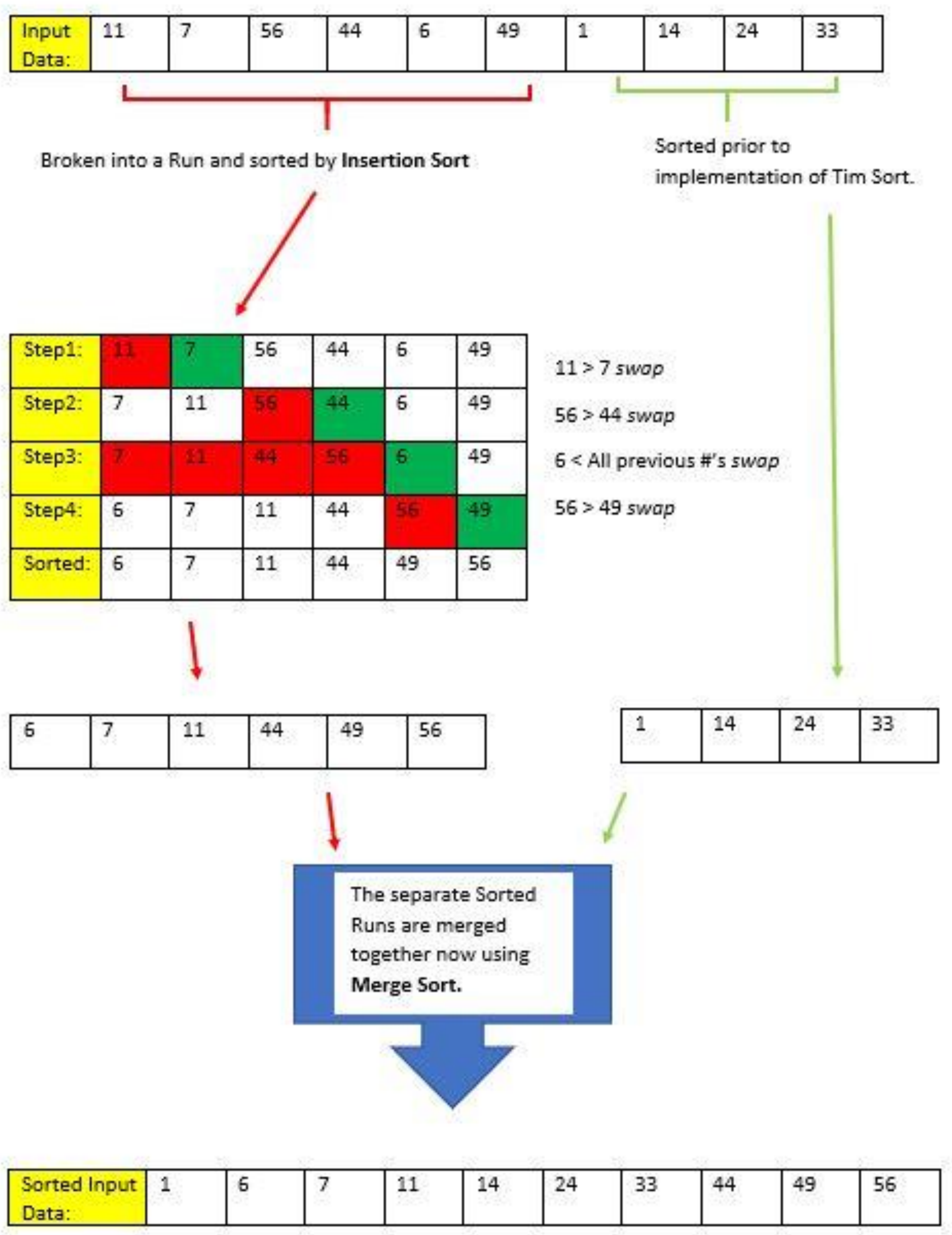
Diagram of Tim Sort Working

Steps of Merge Sort working:

1. Items in the input set are viewed as to whether they are sorted or not sorted.

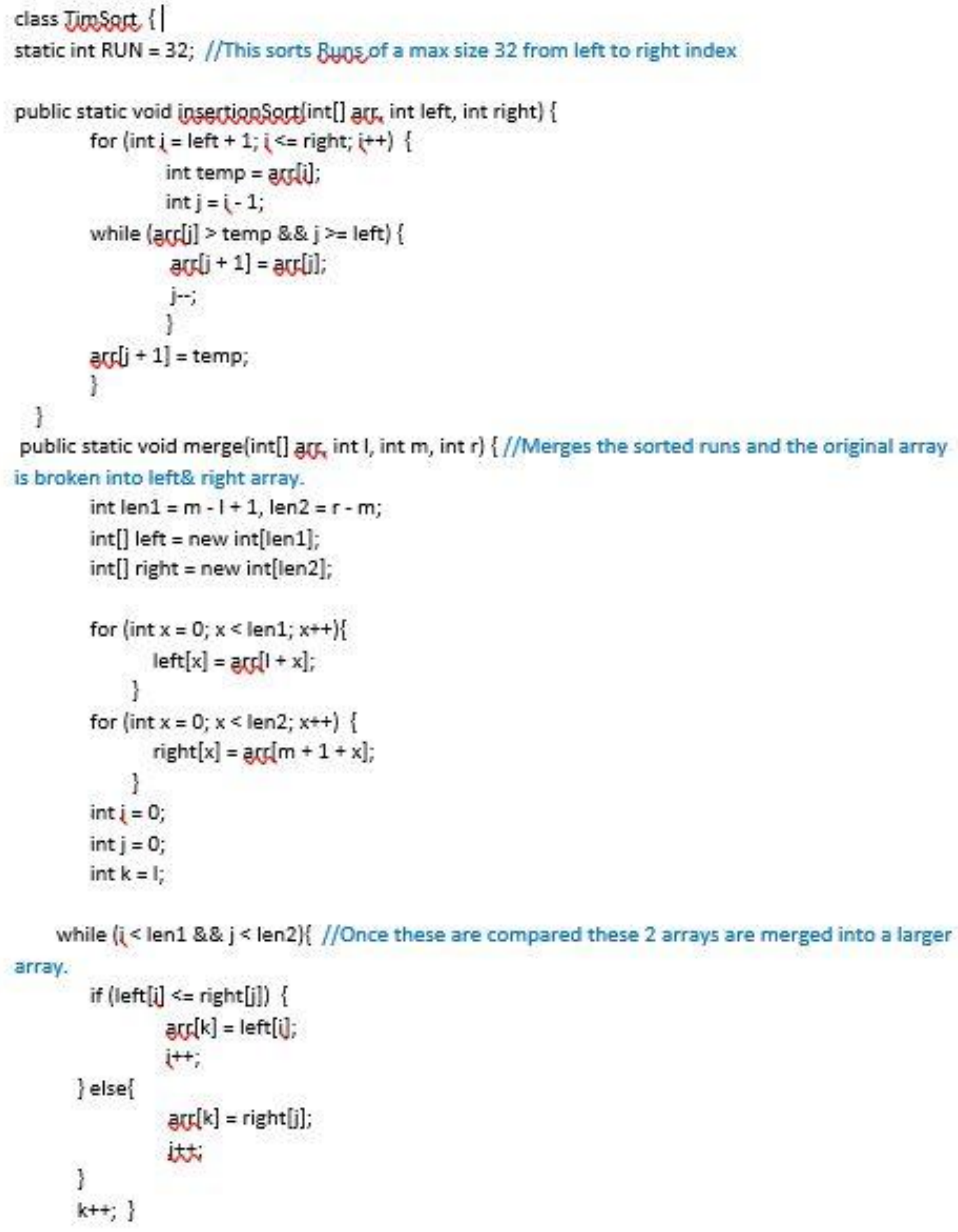
2. Remaining items that are not sorted are broken into Runs and are sorted using insertion sort which rearranges the items into the correct order.
3. When all of the Runs are sorted and rearranged in order. The smaller lists begin to merge back to a bigger list, finally forming one sorted list.

NOTE: Usually Runs are between the range of 32-64 items in length but for the purposes of this diagram I have simplified this to 10.

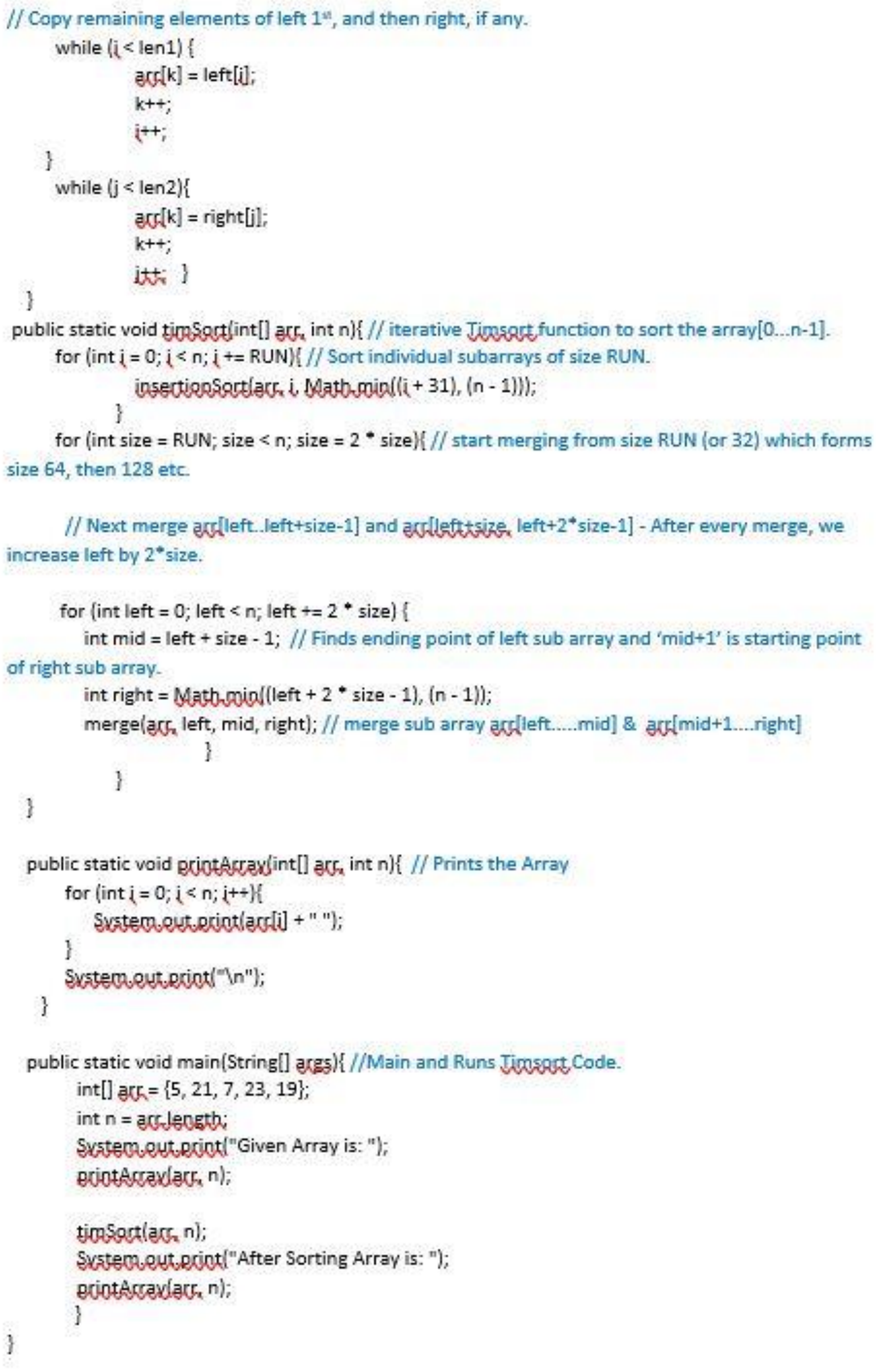

Input Data: 11 7 56 44 6 49 1 14 24 33
Broken into a Run and sorted by **Insertion Sort**
Sorted prior to implementation of Tim Sort.
Step1: 11 7 56 44 6 49
Step2: 7 11 56 44 6 49
Step3: 7 11 44 56 6 49
Step4: 6 7 11 44 56 49
Sorted: 6 7 11 44 49 56
11 > 7 *swap*
56 > 44 *swap*
6 < All previous #'s *swap*
56 > 49 *swap*
6 7 11 44 49 56
1 14 24 33
The separate Sorted Runs are merged together now using **Merge Sort.**
Sorted Input Data: 1 6 7 11 14 24 33 44 49 56

Sample Tim Sort Java Code

Code adapted from example in: (TimSort - GeeksforGeeks, 2019)



```
class TimSort {
static int RUN = 32;  //This sorts Runs of a max size 32 from left to right index

public static void insertionSort(int[] arr, int left, int right) {
            for (int i = left + 1; i <= right; i++)  {
                        int temp = arr[i];
                        int j = i - 1;
            while (arr[j] > temp && j >= left) {
                         arr[j + 1] = arr[j];
                        j--;
                        }
            arr[j + 1] = temp;
            }
   }
 public static void merge(int[] arr, int l, int m, int r) { //Merges the sorted runs and the original array
is broken into left& right array.
            int len1 = m - l + 1, len2 = r - m;
            int[] left = new int[len1];
            int[] right = new int[len2];

            for (int x = 0; x < len1; x++){
                      left[x] = arr[l + x];
                 }
            for (int x = 0; x < len2; x++)  {
                      right[x] = arr[m + 1 + x];
                 }
            int i = 0;
            int j = 0;
            int k = l;

        while (i < len1 && j < len2){  //Once these are compared these 2 arrays are merged into a larger
array.
            if (left[i] <= right[j])  {
                        arr[k] = left[i];
                        i++;
          } else{
                        arr[k] = right[j];
                        j++;
          }
          k++;  }
```



```
// Copy remaining elements of left 1st, and then right, if any.
        while (i < len1) {
                        arr[k] = left[i];
                        k++;
                        i++;
      }
        while (j < len2){
                        arr[k] = right[j];
                        k++;
                        j++;   }
  }
 public static void timSort(int[] arr, int n){ // iterative Timsort function to sort the array[0...n-1].
        for (int i = 0; i < n; i += RUN){ // Sort individual subarrays of size RUN.
                        insertionSort(arr, i, Math.min((i + 31), (n - 1)));
                    }
        for (int size = RUN; size < n; size = 2 * size){ // start merging from size RUN (or 32) which forms
size 64, then 128 etc.

           // Next merge arr[left..left+size-1] and arr[left+size, left+2*size-1] - After every merge, we
increase left by 2*size.

         for (int left = 0; left < n; left += 2 * size) {
             int mid = left + size - 1;  // Finds ending point of left sub array and 'mid+1' is starting point
of right sub array.
             int right = Math.min((left + 2 * size - 1), (n - 1));
             merge(arr, left, mid, right); // merge sub array arr[left.....mid] &  arr[mid+1....right]
                           }
                    }
  }

   public static void printArray(int[] arr, int n){  // Prints the Array
          for (int i = 0; i < n; i++){
               System.out.print(arr[i] + " ");
          }
          System.out.print("\n");
     }

   public static void main(String[] args){ //Main and Runs Timsort Code.
            int[] arr = {5, 21, 7, 23, 19};
            int n = arr.length;
            System.out.print("Given Array is: ");
            printArray(arr, n);

            timSort(arr, n);
            System.out.print("After Sorting Array is: ");
            printArray(arr, n);
            }
}
```

Insertion Sort

Insertion sort is also a simple comparison sorting algorithm.. It was given the name Insertion sort to describe the way it shuffles larger elements to the left in a ‘manner similar to shuffling cards’(Mannion, 2019). It manages to do this It works by comparing adjacent elements to one and another. If the elements are not in order, the sorting algorithm shuffles to the left, so that they are in the correct order (www.tutorialspoint.com, 2019). Thus, simply put Insertion Sort is basically based on the idea of repeatedly moving higher ranked input integers to the end of the input list.

Specifications Big O Notation

Best Case	Average Case	Worst Case	Space Complexity (Sorting inplace)	Stable
n	n^2	n^2	1	Yes

Table Depicting Insertion Sort Specs (GeeksforGeeks, 2019).

Big O Analysis of Insertion Sort

Very useful for small datasets as best case is n time. However, as input size n increases this sorting algorithm slows down due to its use of nested loops to shift items into place. Usually its average and worst case run time would be $O(n^2)$.

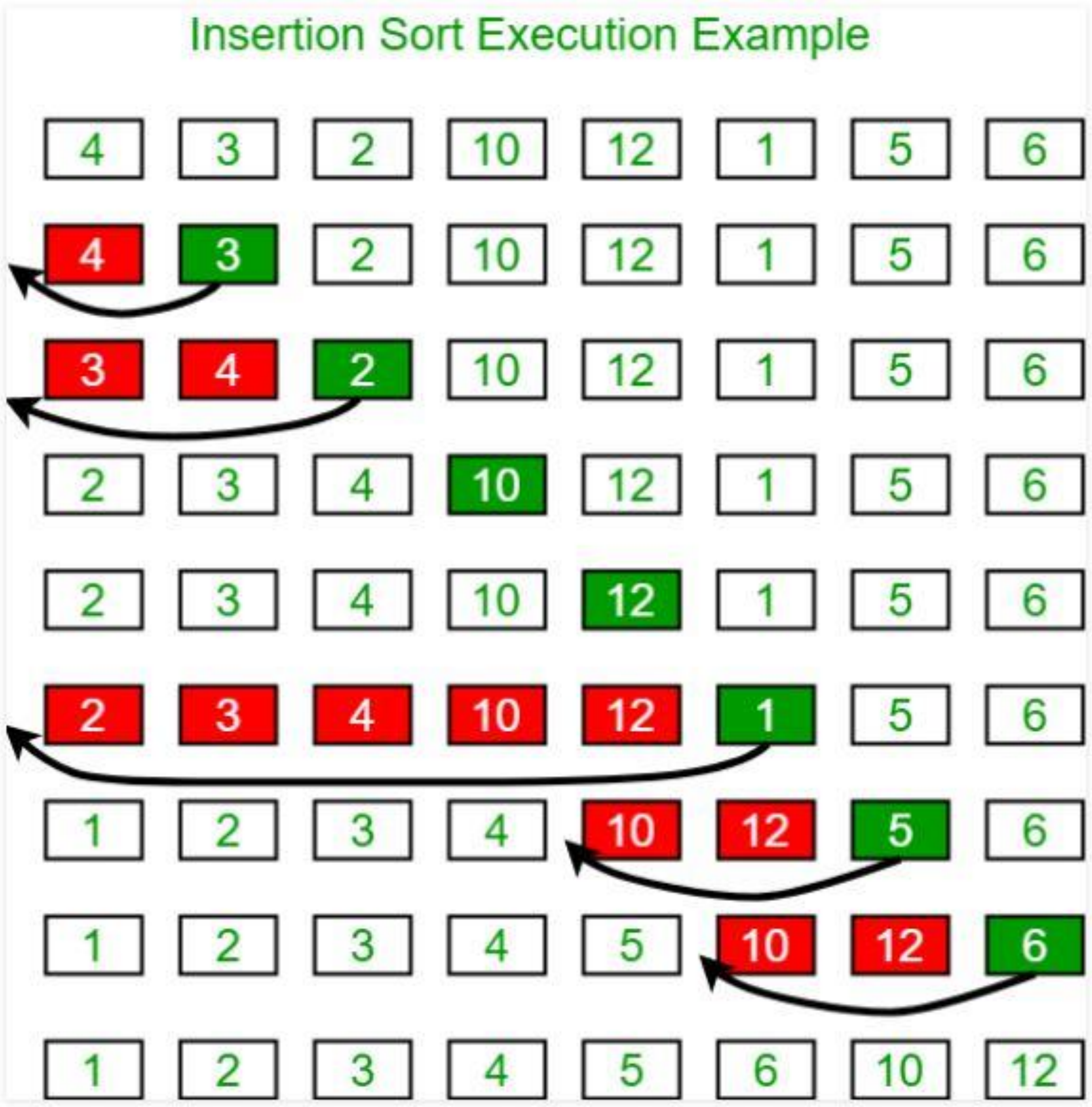
Insertion sort is usually very helpful on datasets < 32 and for this reason is oftentimes paired with another algorithm to finish off the smaller dataset when the input is nearly entirely sorted, this was discussed previously in its use in the hybrid algorithm Tim sort (TimSort - GeeksforGeeks, 2019).

Diagram of Insertion Sort Working in Diagram

Image taken from (Insertion Sort- GeeksforGeeks, 2019)

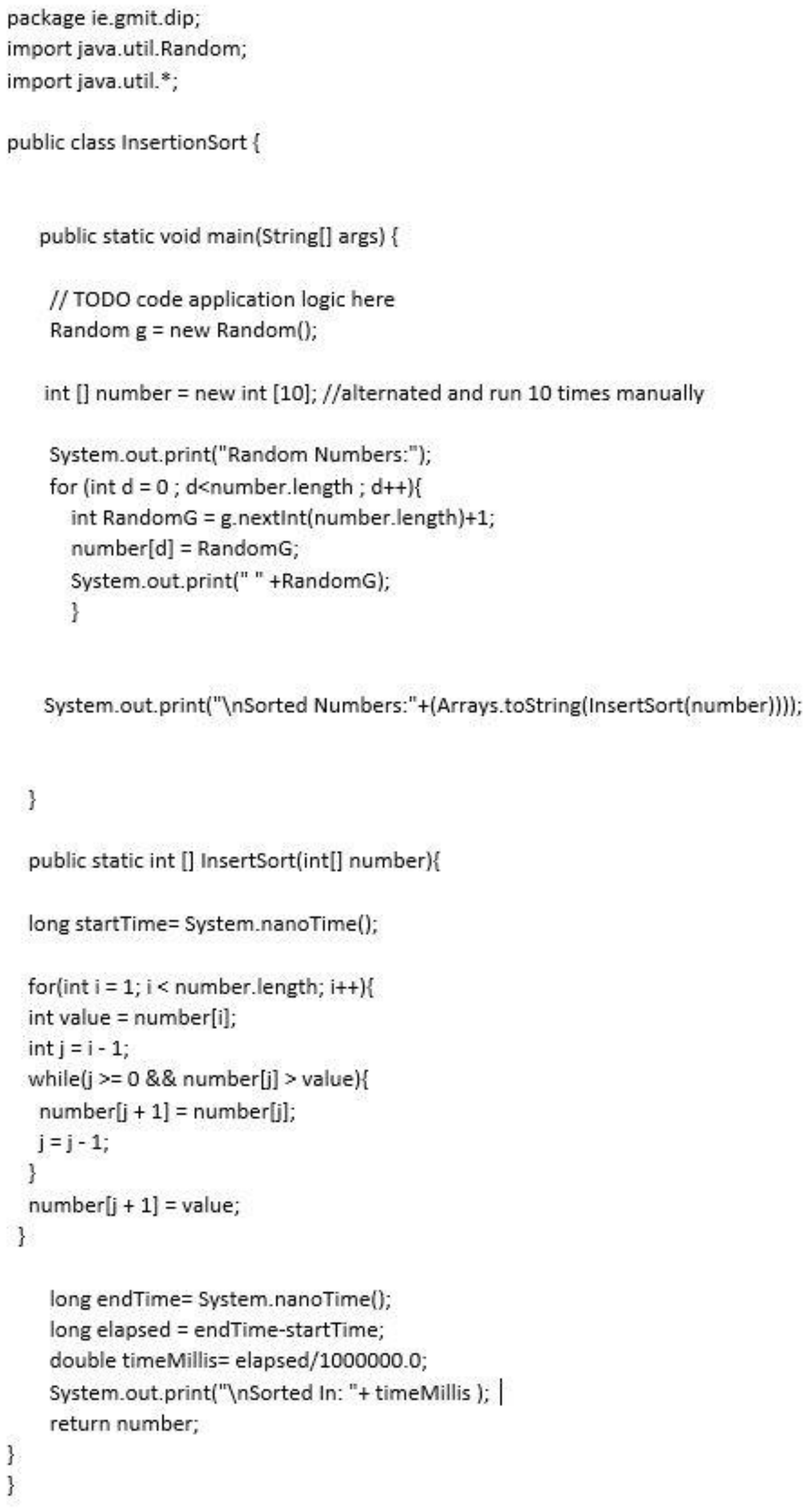
Steps of Insertion Sort working:

1. Items in the input set are viewed as to whether they are sorted or not sorted.
2. Using insertion sort data is shuffled to the left if smaller than its comparator to the right. This rearranges the items into the correct order.



Sample Insertion Sort Java Code

Code adapted from example in: (Insertion Sort- GeeksforGeeks, 2019)



```
package ie.gmit.dip;
import java.util.Random;
import java.util.*;

public class InsertionSort {


    public static void main(String[] args) {

      // TODO code application logic here
      Random g = new Random();

     int [] number = new int [10]; //alternated and run 10 times manually

      System.out.print("Random Numbers:");
      for (int d = 0 ; d<number.length ; d++){
         int RandomG = g.nextInt(number.length)+1;
         number[d] = RandomG;
         System.out.print(" " +RandomG);
         }


     System.out.print("\nSorted Numbers:"+(Arrays.toString(InsertSort(number))));


   }

   public static int [] InsertSort(int[] number){

   long startTime= System.nanoTime();

   for(int i = 1; i < number.length; i++){
   int value = number[i];
   int j = i - 1;
   while(j >= 0 && number[j] > value){
     number[j + 1] = number[j];
     j = j - 1;
   }
   number[j + 1] = value;
  }

      long endTime= System.nanoTime();
      long elapsed = endTime-startTime;
      double timeMillis= elapsed/1000000.0;
      System.out.print("\nSorted In: "+ timeMillis );
      return number;
}
}
```

Bogo Sort

Bogo sort (also known as slow sort, monkey sort, permutations sort etc.) works on a ‘generate and test paradigm’ (BogoSort| Data Structures, 2019). This means that it generates permutations of the input data until it finds the correct sorted sequence. A good analogy of this would be if you had a ‘deck of cards that was shuffled and in need of sorting’. If this analogy was to be sorted using the Bogo Sort algorithm, the deck of cards could be sorted by throwing them on the floor, picking up the cards and checking if they were sorted – and repeat this process until the algorithm finds the correct permutation (Bogosort | wikipedia, 2019).

Specifications Big O Notation

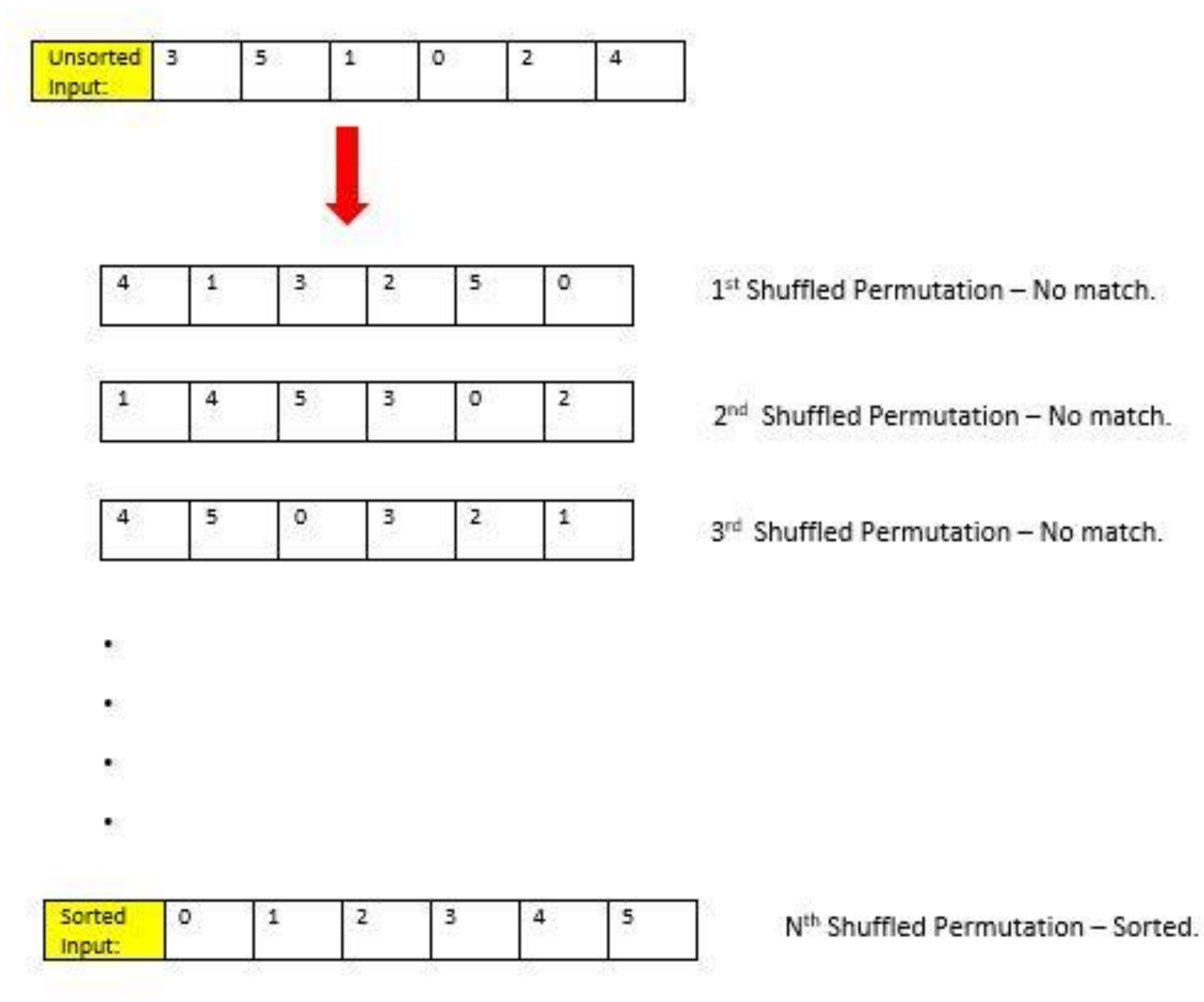
Best Case	Average Case	Worst Case	Space Complexity (Sorting inplace)	Stable
n	(n*n)!	∞ Infinite time	1	No

Table Depicting Bogo Sort Specs (GeeksforGeeks, 2019).

Big O Analysis of Bogo Sort

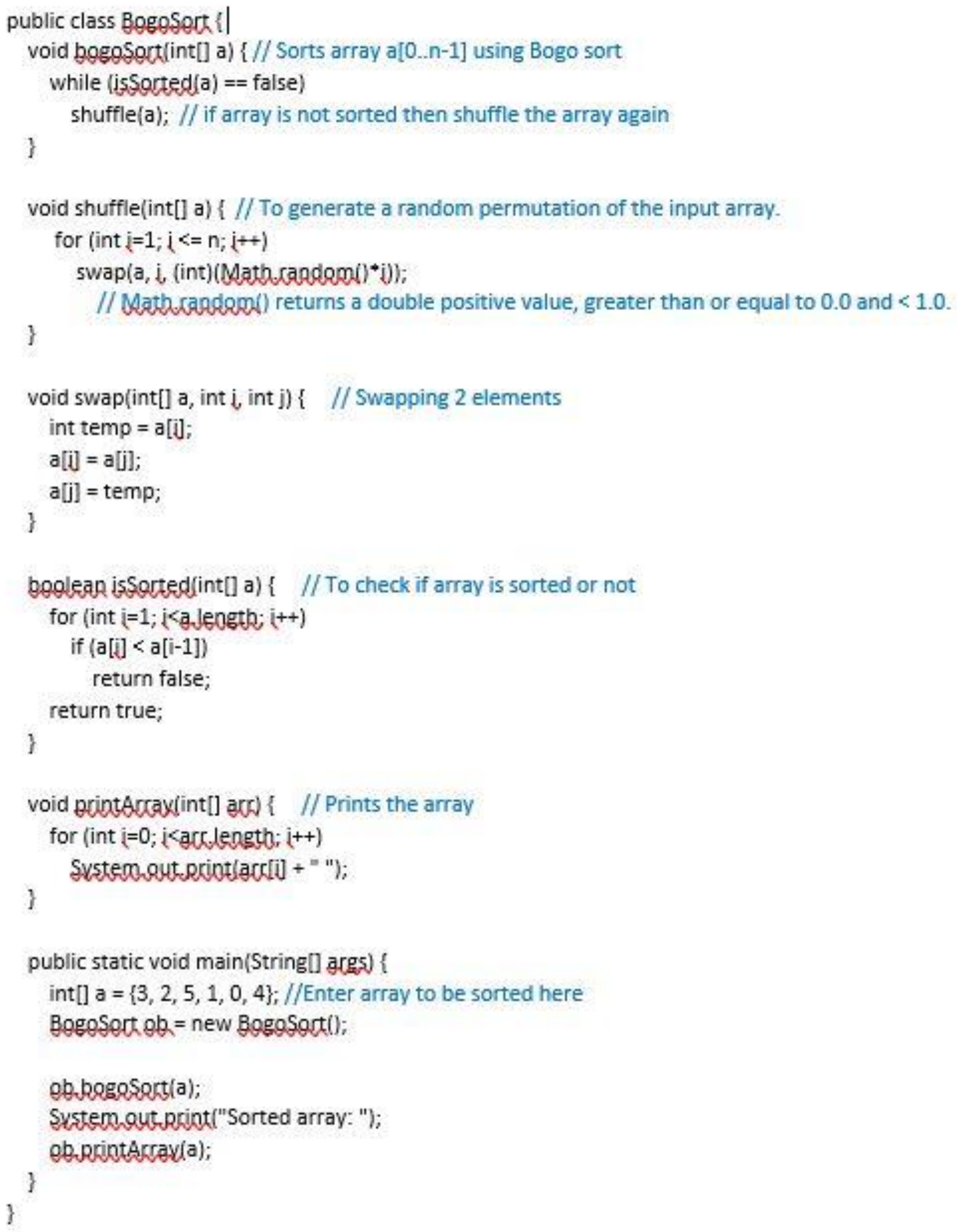
As one would assume, this is a very slow algorithm as it bases its paradigm of randomly finding the correctly sorted version of the inputted data sequence, which could take an infinite amount of time and would be very dependent on the input size n.

Diagram of Bogo Sort Working


Unsorted Input: 3 5 1 0 2 4
4 1 3 2 5 0
1st Shuffled Permutation – No match.
1 4 5 3 0 2
2nd Shuffled Permutation – No match.
4 5 0 3 2 1
3rd Shuffled Permutation – No match.
Sorted Input: 0 1 2 3 4 5
Nth Shuffled Permutation – Sorted.

Sample Bogo Sort Java Code

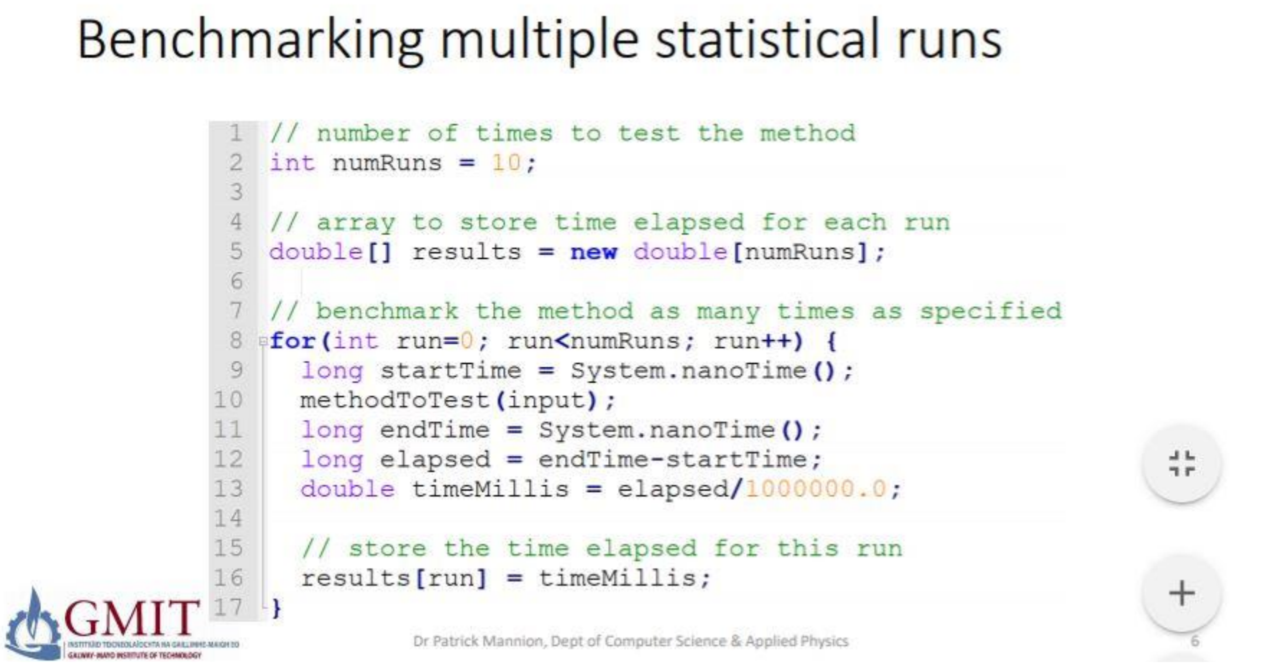
Code adapted from example in: (BogoSort - GeeksforGeeks, 2019)



```
public class BogoSort {
  void bogoSort(int[] a) { // Sorts array a[0..n-1] using Bogo sort
    while (isSorted(a) == false)
      shuffle(a);  // if array is not sorted then shuffle the array again
  }

  void shuffle(int[] a) {  // To generate a random permutation of the input array.
     for (int i=1; i <= n; i++)
       swap(a, i, (int)(Math.random()*i));
         // Math.random() returns a double positive value, greater than or equal to 0.0 and < 1.0.
  }

  void swap(int[] a, int i, int j) {    // Swapping 2 elements
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
  }

  boolean isSorted(int[] a) {    // To check if array is sorted or not
    for (int i=1; i<a.length; i++)
      if (a[i] < a[i-1])
        return false;
    return true;
  }

  void printArray(int[] arr) {    // Prints the array
    for (int i=0; i<arr.length; i++)
      System.out.print(arr[i] + " ");
  }

  public static void main(String[] args) {
    int[] a = {3, 2, 5, 1, 0, 4}; //Enter array to be sorted here
    BogoSort ob = new BogoSort();

    ob.bogoSort(a);
    System.out.print("Sorted array: ");
    ob.printArray(a);
  }
}
```

Implementation & Benchmarking - Results

All code for this project was written using the Codenvy Eclipse Che package used in the 'Introduction to Java' module for this course (Codenvy.io, 2019). The code implemented in my App was derived from website GeeksforGeeks (GeeksforGeeks, 2019) and interpreted as well as possible. I unfortunately had issues understanding certain aspects of the algorithms, but I tried my best to carry out the tasks at hand for the project. One issue I struggled with was automating10 runs for each algorithm as described in the final lecture slides for this module (seen below). This was needed to find the average time for each of the sorting algorithms, so instead I carried out this task out manually.



Benchmarking multiple statistical runs

```
// number of times to test the method
int numRuns = 10;

// array to store time elapsed for each run
double[] results = new double[numRuns];

// benchmark the method as many times as specified
for(int run=0; run<numRuns; run++) {
  long startTime = System.nanoTime();
  methodToTest(input);
  long endTime = System.nanoTime();
  long elapsed = endTime-startTime;
  double timeMillis = elapsed/1000000.0;

  // store the time elapsed for this run
  results[run] = timeMillis;
}
```

GMIT

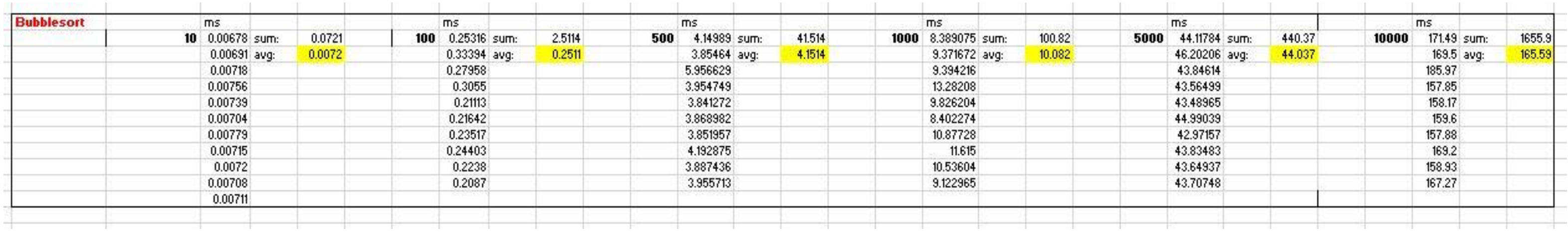
Dr Patrick Mannion, Dept of Computer Science & Applied Physics

6

Image taken from (Mannion, 2019)

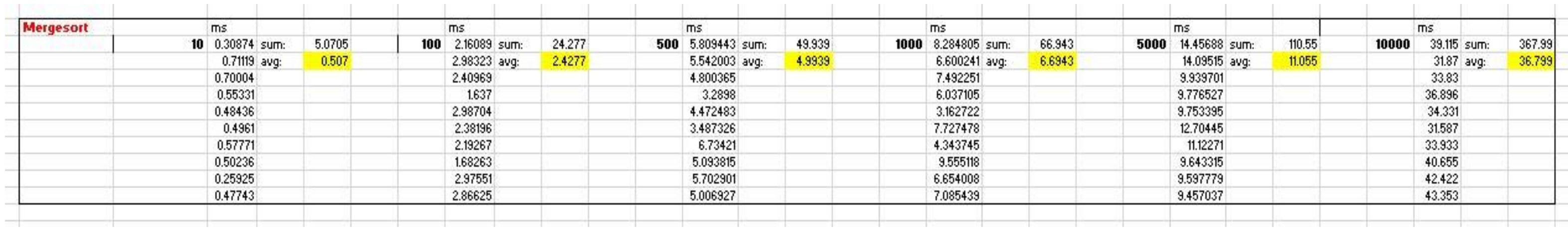
For this reason, I decided to include each of the generated files below in the results section of this report document and also the attached excel file used to generate the graph.

Results table1 – Bubble sort time output values in milliseconds with differing inputs sizes n.



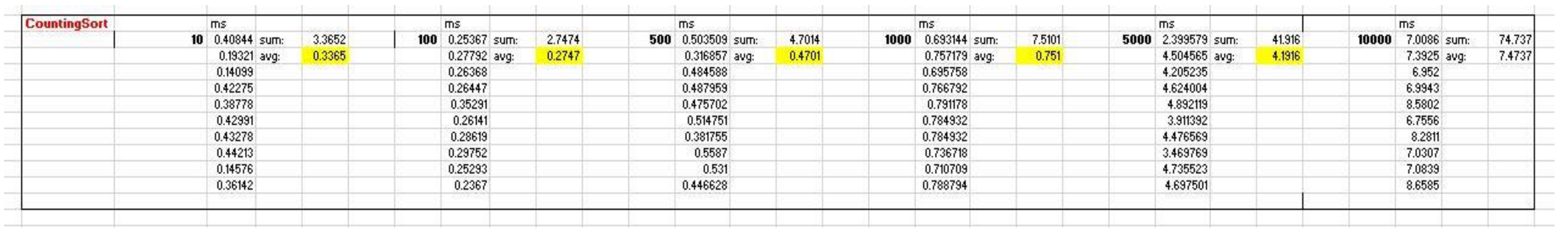
Bubblesort		ms				ms				ms				ms				ms				ms		
	10	0.00678	sum:	0.0721	100	0.25316	sum:	2.5114	500	4.14989	sum:	41.514	1000	8.389075	sum:	100.82	5000	44.11784	sum:	440.37	10000	171.49	sum:	1655.9
		0.00691	avg:	0.0072		0.33394	avg:	0.2511		3.85464	avg:	4.1514		9.371672	avg:	10.082		46.20206	avg:	44.037		169.5	avg:	165.59
		0.00718				0.27958				5.956629				9.394216				43.84614				185.97		
		0.00756				0.3055				3.954749				13.28208				43.56499				157.85		
		0.00739				0.21113				3.841272				9.826204				43.48965				158.17		
		0.00704				0.21642				3.868982				8.402274				44.99039				159.6		
		0.00779				0.23517				3.851957				10.87728				42.97157				157.88		
		0.00715				0.24403				4.192875				11.615				43.83483				169.2		
		0.0072				0.2238				3.887436				10.53604				43.64937				158.93		
		0.00708				0.2087				3.955713				9.122965				43.70748				167.27		
		0.00711																						

Results table2 – Merge sort time output values in milliseconds with differing inputs sizes n.



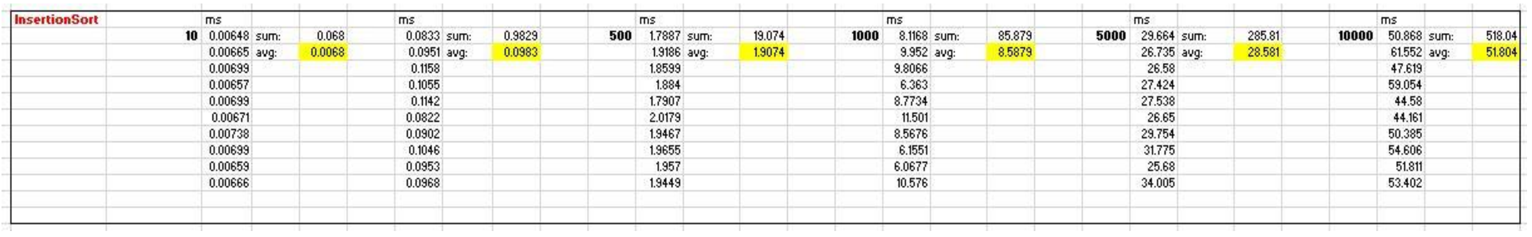
Mergesort		ms				ms				ms				ms				ms				ms		
	10	0.30874	sum:	5.0705	100	2.16089	sum:	24.277	500	5.809443	sum:	49.939	1000	8.284805	sum:	66.943	5000	14.45688	sum:	110.55	10000	39.115	sum:	367.99
		0.71119	avg:	0.507		2.98323	avg:	2.4277		5.542003	avg:	4.9939		6.600241	avg:	6.6943		14.09515	avg:	11.055		31.87	avg:	36.799
		0.70004				2.40969				4.800365				7.492251				9.939701				33.83		
		0.55331				1.637				3.2898				6.037105				9.776527				36.896		
		0.48436				2.98704				4.472483				3.162722				9.753395				34.331		
		0.4961				2.38196				3.487326				7.727478				12.70445				31.587		
		0.57771				2.19267				6.73421				4.343745				11.12271				33.933		
		0.50236				1.68263				5.093815				9.555118				9.643315				40.655		
		0.25925				2.97551				5.702901				6.654008				9.597779				42.422		
		0.47743				2.86625				5.006927				7.085439				9.457037				43.353		

Results table 3 – Counting sort time output values in milliseconds with differing inputs sizes n.



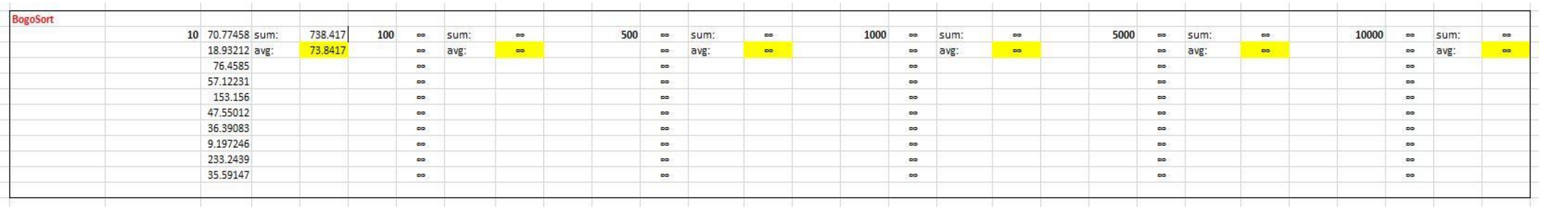
CountingSort		ms				ms				ms				ms				ms				ms		
	10	0.40844	sum:	3.3652	100	0.25367	sum:	2.7474	500	0.503509	sum:	4.7014	1000	0.693144	sum:	7.5101	5000	2.399579	sum:	41.916	10000	7.0086	sum:	74.737
		0.19321	avg:	0.3365		0.27792	avg:	0.2747		0.316857	avg:	0.4701		0.757179	avg:	0.751		4.504565	avg:	4.1916		7.3925	avg:	7.4737
		0.14099				0.26368				0.484588				0.695758				4.205235				6.952		
		0.42275				0.26447				0.487959				0.766792				4.624004				6.9943		
		0.38778				0.35291				0.475702				0.791178				4.892119				8.5802		
		0.42991				0.26141				0.514751				0.784932				3.911392				6.7556		
		0.43278				0.28619				0.381755				0.784932				4.476569				8.2811		
		0.44213				0.29752				0.5587				0.736718				3.469769				7.0307		
		0.14576				0.25293				0.531				0.710709				4.735523				7.0839		
		0.36142				0.2367				0.446628				0.788794				4.697501				8.6585		

Results table 4 – Insertion sort time output values in milliseconds with differing inputs sizes n.



InsertionSort		ms				ms				ms				ms				ms				ms		
	10	0.00648	sum:	0.068		0.0833	sum:	0.9829	500	1.7887	sum:	19.074	1000	8.1168	sum:	85.879	5000	29.664	sum:	285.81	10000	50.868	sum:	518.04
		0.00665	avg:	0.0068		0.0951	avg:	0.0983		1.9186	avg:	1.9074		9.952	avg:	8.5879		26.735	avg:	28.581		61.552	avg:	51.804
		0.00699				0.1158				1.8599				9.8066				26.58				47.619		
		0.00657				0.1055				1.884				6.363				27.424				59.054		
		0.00699				0.1142				1.7907				8.7734				27.538				44.58		
		0.00671				0.0822				2.0179				11.501				26.65				44.161		
		0.00738				0.0902				1.9467				8.5676				29.754				50.385		
		0.00699				0.1046				1.9655				6.1551				31.775				54.606		
		0.00659				0.0953				1.957				6.0677				25.68				51.811		
		0.00666				0.0968				1.9449				10.576				34.005				53.402		

Results table 5 – Bogo sort time output values in milliseconds with differing inputs sizes n.

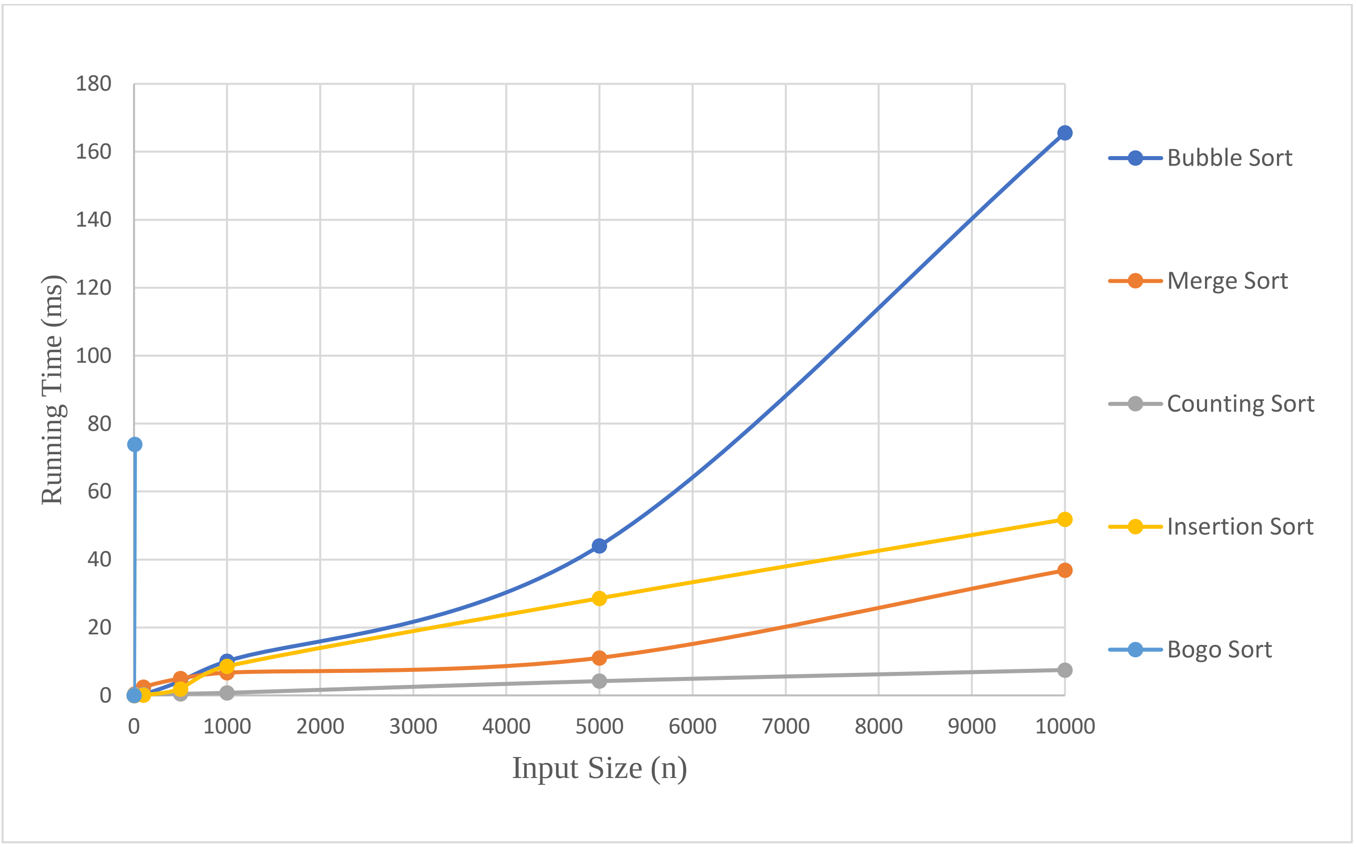


BogoSort																								
	10	70.77458	sum:	738.417	100	∞	sum:	∞	500	∞	sum:	∞	1000	∞	sum:	∞	5000	∞	sum:	∞	10000	∞	sum:	∞
		18.93212	avg:	73.8417		∞	avg:	∞		∞	avg:	∞		∞	avg:	∞		∞	avg:	∞		∞	avg:	∞
		76.4585				∞				∞				∞				∞				∞		
		57.12231				∞				∞				∞				∞				∞		
		153.156				∞				∞				∞				∞				∞		
		47.55012				∞				∞				∞				∞				∞		
		36.39083				∞				∞				∞				∞				∞		
		9.197246				∞				∞				∞				∞				∞		
		233.2439				∞				∞				∞				∞				∞		
		35.59147				∞				∞				∞				∞				∞		

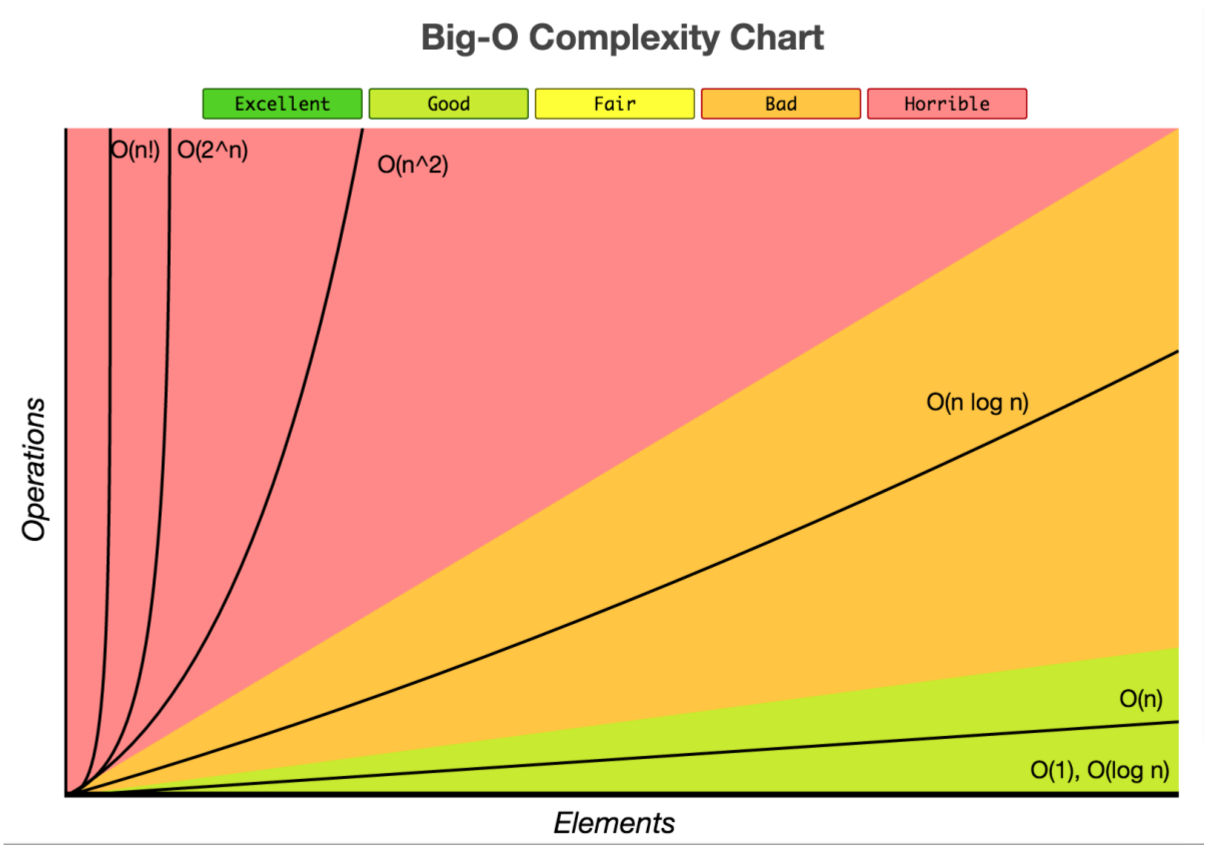
Results table 6 – values are in milliseconds and are an average of 10 manually repeated runs seen in the excel file above.

Input Size Array n	0	10	100	500	1000	5000	10000
Bubble Sort	0	0.00721	0.251145	4.151414	10.0817	44.0374	165.586
Merge Sort	0	0.507048	2.42769	4.99393	6.69429	11.0547	36.7992
Counting Sort	0	0.336516	0.27474	0.47014	0.75101	4.19163	7.47372
Insertion Sort	0	0.006799	0.09829	1.90738	8.58786	28.5806	51.8036
BogoSort	0	73.8417	∞	∞	∞	∞	∞

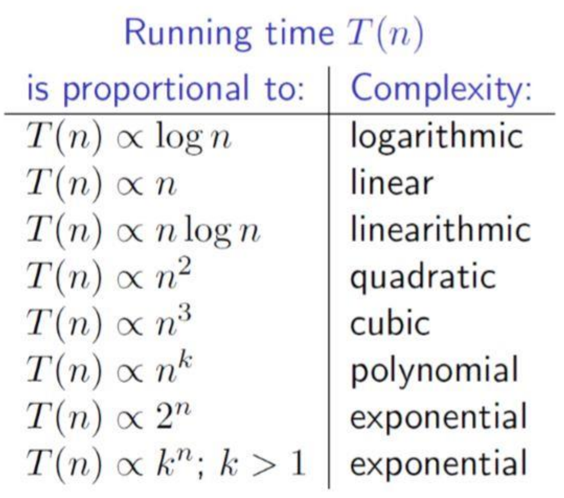
Results table – Graph representation of the Time Complexity of each sorting Algorithm Tested



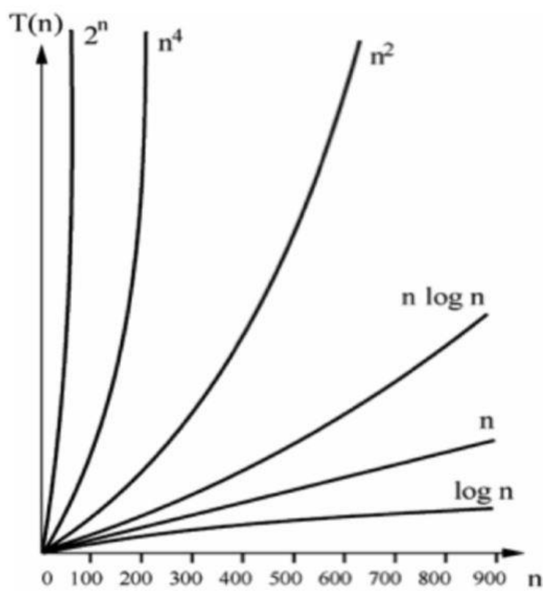
By analysing the time complexity of each of the five chosen algorithms, one was able to determine the scalability in relation to an increased input size. This was then compared with the previously discussed diagram (which I will reintroduce below) seen in the introduction. It can clearly be seen that each of the algorithms follow a distinct pattern similar to their Big O time complexity



Decreasing in efficiency



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



Images Depicting Running times in relation to Complexity (Mannion, 2019) (Fox, 2019)

Summary – Specifications of the 5 Chosen Sorting Algorithms.

Finally, to conclude this this project focused on evaluating the time efficiency of 5 chosen sorting algorithms. These included Bubble Sort, Merge Sort, Counting Sort, Insertion Sort and Bogo Sort. An inclusive algorithm application was developed to benchmark each to evaluate each of their time complexity efficiencies.

After the research carried out for this project, I have developed an interest in the different way's algorithms are analysed and I hope to pursue this further in the future.

Sorting Algorithm	Best Case	Average Case	Worst Case	Space Complexity (Sorting inplace)	Stable
Bubble Sort	O(n)	O(n2)	O(n2)	1	Yes
Merge Sort	n log n	n log n	n log n	0(n)	Yes
Counting Sort	n + k	n + k	n + k	n + k	Yes
Tim Sort	n	n log n	n log n	n	Yes
Insertion Sort	n	n^2	n^2	1	Yes
Bogo Sort	n	(n*n)!	∞ (Infinite)	1	No

References

Bandalou. (2019). *Plan Toys Shape and Sort It Out*. [online] Available at: https://bandaloubaby.com/products/65736-plan-toys-shape-and-sort-it-out [Accessed 6 May 2019].

Bogosort | wikipedia. (2019). Bogosort. [online] Available at: https://en.wikipedia.org/wiki/Bogosort [Accessed 6 May 2019].

BogoSort| Data Structures. (2019). BogoSort in Kotlin | Data Structures. [online] Available at: https://chercher.tech/kotlin/bogosort-kotlin [Accessed 6 May 2019].

Codenvy.io. (2019). *Codenvy*. [online] Available at: https://codenvy.io/ [Accessed 12 May 2019].

Calculus.seas.upenn.edu. (2019). PennCalc | Main / OrdersOfGrowth. [online] Available at: http://calculus.seas.upenn.edu/?n=Main.OrdersOfGrowth [Accessed 5 May 2019].

Counting Sort Algorithm. (2019). Interview Cake: Programming Interview Questions and Tips. [online] Available at: https://www.interviewcake.com/concept/python3/counting-sort [Accessed 3 May 2019].

Cs.cmu.edu. (2019). Introduction to Sorting. [online] Available at: http://www.cs.cmu.edu/~clo/www/CMU/DataStructures/Lessons/lesson8_1.htm [Accessed 3 May 2019].

Dictionary.cambridge.org. (2019). SORT | meaning in the Cambridge English Dictionary. [online] Available at: https://dictionary.cambridge.org/dictionary/english/sort [Accessed 4 May 2019].

En.wikipedia.org. (2019). Bubble sort. [online] Available at: https://en.wikipedia.org/wiki/Bubble_sort [Accessed 4 May 2019].

Fox, P. (2019). *Intro to Algorithms*. [online] Teaching-materials.org. Available at: https://www.teaching-materials.org/algorithms/#/ [Accessed 11 May 2019].

GeeksforGeeks. (2019). Analysis of Algorithms | Set 2 (Worst, Average and Best Cases) - GeeksforGeeks. [online] Available at: https://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/ [Accessed 5 May 2019].

GeeksforGeeks. (2019). Bubble Sort - GeeksforGeeks. [online] Available at: https://www.geeksforgeeks.org/bubble-sort/ [Accessed 4 May 2019].

GeeksforGeeks. (2019). Counting Sort - GeeksforGeeks. [online] Available at: https://www.geeksforgeeks.org/counting-sort/ [Accessed 7 May 2019].

Gudpict.pw. (2019). Colour sorting toys. [online] Available at: http://gudpict.pw/colour-sorting-toys.html [Accessed 6 May 2019].

Insertion Sort- GeeksforGeeks. (2019). Insertion Sort - GeeksforGeeks. [online] Available at: https://www.geeksforgeeks.org/timsort/ [Accessed 10 May 2019]

Interactivepython.org. (2019). The Merge Sort — Problem Solving with Algorithms and Data Structures. [online] Available at: http://interactivepython.org/lpomz/courselib/static/pythonds/SortSearch/TheMergeSort.html [Accessed 5 May 2019].

Javin, P. (2019). Counting Sort. [online] Counting Sort in Java - Example. Available at: http://www.java67.com/2017/06/counting-sort-in-java-example.html [Accessed 6 May 2019].

Khan Academy. (2019). Overview of merge sort. [online] Available at: https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/overview-of-merge-sort [Accessed 4 May 2019].

Leeuwen, J. (1990). Handbook of theoretical computer science. — 3.1 Inversions. Amsterdam New York Cambridge, Mass: Elsevier MIT Press, p.p. 459.

Liss, A., shrivastava, s. and Mihalcin, A. (2019). Best, Worst and Average case Running Times. [online] Stack Overflow. Available at: https://stackoverflow.com/questions/9561242/best-worst-and-average-case-running-times [Accessed 4 May 2019].

Mannion, P. (2019). Sorting Algorithms Part 1. Galway: Galway-Mayo Institute of Technology., p.Lecture 2.

Medium. (2019). A Simplified Explanation of Merge Sort. [online] Available at: https://medium.com/karuna-sehgal/a-simplified-explanation-of-merge-sort-77089fe03bb2 [Accessed 5 May 2019].

Sciencing. (2019). The Advantages & Disadvantages of Sorting Algorithms. [online] Available at: https://sciencing.com/the-advantages-disadvantages-of-sorting-algorithms-12749529.html [Accessed 4 May 2019].

Sitedownrightnow.com. (2019). upenn seas | About - Summer Engineering Academy. [online] Available at: http://www.sitedownrightnow.com/search/upenn-seas [Accessed 7 May 2019].

Sparrowflights.blogspot.com. (2019). Big Oh notation explained. [online] Available at: http://sparrowflights.blogspot.com/2011/01/big-oh-notation-explained.html [Accessed 3 May 2019].

TimSort - GeeksforGeeks. (2019). TimSort - GeeksforGeeks. [online] Available at: https://www.geeksforgeeks.org/timsort/ [Accessed 7 May 2019].

WhatIs.com. (2019). What is sorting algorithm? - Definition from WhatIs.com. [online] Available at: https://whatis.techtarget.com/definition/sorting-algorithm [Accessed 6 May 2019].

www.tutorialspoint.com. (2019). Data Structures and Algorithms - Bubble Sort. [online] Available at: https://www.tutorialspoint.com/data_structures_algorithms/bubble_sort_algorithm.htm [Accessed 3 May 2019].

YouTube - BubbleSort Algorithm. (2019). Java: BubbleSort Algorithm Explained. [online] Available at: https://www.youtube.com/watch?v=F13_wsHDIG4 [Accessed 30 Apr. 2019].

YouTube - MergeSort Algorithm. (2019). Java: MergeSort Algorithm Explained. [online] Available at: https://www.youtube.com/watch?v=iMT7gTPpaqw [Accessed 28 Apr. 2019].