

COMPUTATIONAL THINKING AND ALGORITHMS PROJECT 2022

Table of Contents

Introduction.....	1
Sorting Algorithms	3
CountingSort.....	3
BubbleSort.....	6
MergeSort.....	9
BogoSort.....	11
InsertionSort.....	14
ImplementationBenchmarking.....	17
Bibliography.....	21

1. INTRODUCTION

The purpose of this assignment is to gain a better understanding of the functionality of different Sorting Algorithms. Prior to writing up my report, I have carried out my own research on the sorting algorithms which I wish to compare and contrast. I have furthermore tested them for myself using Eclipse for performance and run-time.

1.1 Sorting Algorithm Definition

Firstly, to define what the concept sorting means: sorting is a process whereby items or groups of things are arranged in a particular way. ‘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). In other words, algorithms that sort data or some values either in ascending or descending order are known as sorting algorithms. A sorted data set must be systematically reorganised in a way where each value is in sequence.

1.2 When is data regarded as being sorted?

Sorting algorithms can be assessed under the following headings

Suitability:

Whether or not the strengths and weaknesses of the sorting algorithm can hold up to a class instance from user input.

Stability:

If the data set of the algorithm is already slightly pre-sorted, the algorithm should preserve this sorting and not have start from zero.

Performance:

The sorting algorithm should be suitable to work on a specific machine or device i.e. the speed of the computer and its ability to compile the algorithm. According to (Mannion, 2019), algorithms that use this sort of memory usage are called in-place.

External factors:

This refers to algorithms running in external memory.

Efficiency/Time Complexity

Space and Time Complexity are key factors when evaluating the performance of an algorithm which is explained later. Space efficiency examines the computer's ability to run the algorithm including the memory space used for inputs and instructions.

Time efficiency is where the length of time is evaluated in order to compile and run the algorithm.

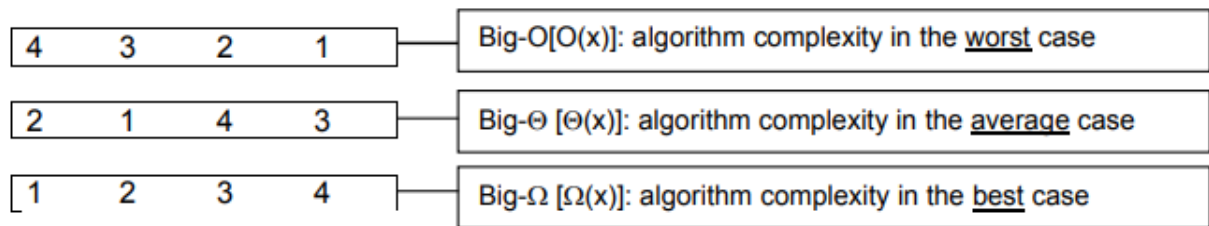
1.3 Asymptotic behavior notation:

For this project, asymptotic notation was used to evaluate the speed and performance of the five sorting algorithms: bubble sort, merge sort, counting sort, bogo sort and insertion sort. .

1.4 Time complexity

Big-O Notation measures how a function grows and declines (Mannion, 2019). It's essentially the standard computational metric for evaluating the running time of an algorithm and identifying the algorithm's behavior as a data input size gets bigger. There are three main

groups of asymptotic notation: Big O Notation Big-Θ notation, and Big-Ω notation.



Best case- this is where the algorithm is completed in the fastest running time. Usually denoted by Ω (Omega) notation and displays a 'linear growth in execution'. (Mannion, 2019).

Average case- the algorithm is compiled several times using different input sizes n . usually denoted by θ (theta) notation e.g. (Mannion, 2019).

Worst case- refers to the slowest runtime possible in order to compile the sorting algorithm

There are different types of time space complexities in Big O Notation.

Name	Big-O Notation
Constant Time	$O(1)$
Linear Time	$O(n)$
Quadratic	$O(n^2)$
Cubic time	$O(n^3)$

Taken from https://en.wikipedia.org/wiki/Big_O_notation

1.5 Comparison and non-comparison sorting algorithms.

Sorting algorithms fall into two main bands: comparison-based and non-comparison algorithms.

Comparison based Sorting algorithm: Comparison based sorting algorithm based on values are compared and values are placed in ascending or descending order.

Non-comparison sorting algorithms: they don't use a comparator for sorting an input data set.

Comparison based sorting algorithm	Non comparison based sorting algorithms
Bubble Sort	Counting Sort

Insertion Sort	
Selection Sort	
Merge Sort	

2. 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 achieve this.

The 5 Sorting Algorithms I have decided to benchmark and implement in this assignment are:

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

2.1 Counting Sort

This algorithm was introduced by H. Seward in 1954. Counting sort algorithm is used for sorting the elements of an array/list by calculating the number of occurrences of each unique component in the array. It then computes the index position of each element in the array/list.

There are some requirements when using the counting sort:

1. Each component in the array list must all be integer numbers
2. A key range k needs to be determined.

2.1.1 Specification Big O Notation

Best Case	Average Case	Worst Case	Auxiliary Space	Stable	Class
-----------	--------------	------------	--------------------	--------	-------

$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$	$O(k)$	Yes	Non-comparison sort
---------------	---------------	----------	--------	-----	---------------------

Table Depicting Counting Sort Specs (GeeksforGeeks, 2022).

2.1.2 Big O Analysis of Counting Sort

One of counting sort algorithm's greatest strengths is that it runs on Linear time ($O(n)$) which makes its running time faster than most comparison-based sorting algorithms i.e. the comparison algorithm 'merge sort' has a running time $n \log n$ (Mannion, 2019).

2.1.3 Steps of Counting Sort working:

Steps adapted from example in: (geeksforgeeks/counting-sort,2022)

Take for instance the data within the index range 0 to 8.

Input data: 1, 3, 1, 4, 2, 7, 8, 6

1. Take a count array to store the frequency of each unique component in the list.

Array Index: 0 1 2 3 4 5 6 7 8

Sum: 1 2 0 1 1 0 1 1 0

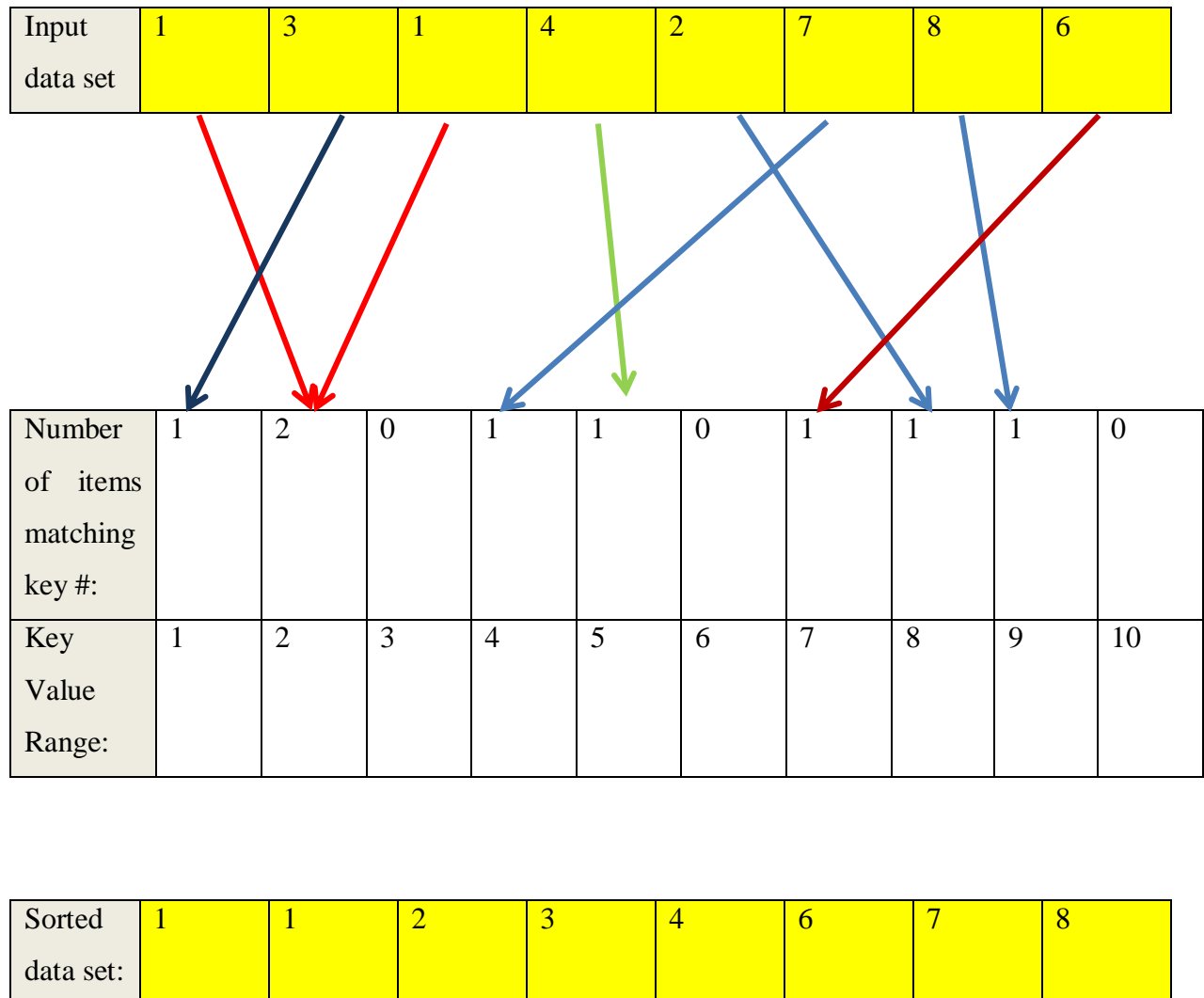
2. Adjust the count array such that each component in the list at each index stores the sum of previous counts.

Index: 0 1 2 3 4 5 6 7 8

Sum: 1 1 2 3 4 6 7 8

The altered count array shows the index position of each object in the user output sequenc

2.1.4 Diagram of Bubble Sort Working



2.1.5 Java Sample implementation of Counting Sort

Code adapted from example in: (Counting Sort - GeeksforGeeks, 2022)

```
package ie.gmit.dip;
```

```
class CountingSort {
```

```
    static void counting_sort(int[] ArrayList) {
```

```
        // variable for largest component of the array list
```

```
        int maximum_number = ArrayList[0];
```

```
        // iterate through the arraylist from index[0]
```

```
        for (int n : ArrayList) {
```

```

        // compare each component with the maximum number in the array list
        if (n > maximum_number)
            maximum_number = n;
    }
    // variable for smallest component of array
    int minimum_number = ArrayList[0];
    // iterate through the array
    for (int n : ArrayList) {
        // compare each component with min_num
        if (n < minimum_number)
            minimum_number = n;
    }
    // find the range for sorting between minimum and maximum number
    int range = maximum_number - minimum_number + 1;
    int count_number[] = new int[range];
    // declare an array named output_arr with size equal to length of array
    int output_arr[] = new int[ArrayList.length];

    // iterate through the array/list and count the number of occurrence of each
number
    for (int i = 0; i < ArrayList.length; i++) {
        count_number[ArrayList[i] - minimum_number]++;
    }

    // change the value of count_num[i] in such a way that count_num[i]
    // now contains actual position of this number in output_arr array
    for (int i = 1; i < count_number.length; i++) {
        count_number[i] += count_number[i - 1];
    }
    // iterate through the array in reverse order in order to make it stable
    // and build the output_arr array
    for (int i = ArrayList.length - 1; i >= 0; i--) {
        // assign the A[i] at proper index of output_arr array

```

```

        output_arr[count_number[ArrayList[i] - minimum_number] - 1] =
ArrayList[i];

        // decrement the number count_num
        count_number[ArrayList[i] - minimum_number]--;
    }
    // now copy all the elements of output_arr array in original array 'A'
    for (int i = 0; i < ArrayList.length; i++) {
        ArrayList[i] = output_arr[i];
    }
}

// Driver code
public static void main(String[] args) {
    // input array
    int A[] = { -5, -12, 5, 12, 9, 0, -23, 98, 47, 32, 34 };
    // print the message
    System.out.println("The unsorted array is: ");
    // print the array before sorting it
    for (int n : A) {
        System.out.print(n + " ");
    }

    // call the counting_sort() and sort the array
    counting_sort(A);

    // print the message after sorting
    System.out.println("\n\nThe sorted array is: ");
    // print the array before sorting it
    for (int n : A) {
        System.out.print(n + " ");
    }
}
}

```


2.2 Bubble sort

Bubble sort also sometimes referred to sinking sort is a simplest sorting algorithm which was created in 1956. The algorithm is named bubble as to describe how smaller or larger elements 'bubble' to the top of an array list (wiki/bubble-sort, 2022). This sorting algorithm is a comparison-based algorithm whereby each pair of adjacent elements in the array is compared to one another (tutorialspoint, 2022). In other words, it works repeatedly on adjacent elements in the array by swapping them if they are not in the right order. The iterations through the list are repeated until the array is sorted.

2.2.1 Specification Big O Notation

Best Case	Average Case	Worst Case	Auxiliary Space	Stable	Class
$O(n)$	$O(n*n)$.	$O(n^2)$	$O(1)$	Yes	Sorting algorithm

Table Depicting Bubble Sort Specifications (GeeksforGeeks, 2022).

Best Case Time Complexity: $O(n)$. The best case scenario is when the elements in the list are already sorted.

Average/ worst case runtime: This would occur if the element in the array list is in reverse order prior to sorting the array set.

2.2.2 Big O Analysis of Bubble Sort

Due to the simplicity of this sorting algorithm it would be an impractical choice of sorting algorithm to use for most problems, especially problems with greatly unsorted data and large input sizes (Mannion, 2019). The algorithm would be suited where the user input list is nearly sorted or if there is an input array of small size left to be sorted or categorized.

The following steps are required in the bubble sort algorithm:

1. Start from index 0 of the array/list and compare all the elements in the list right up until the end.
2. If the element at index (0) is bigger than the current element then swap them.

3. Increment the index value to 1 and repeat the step 2 for value at index 1.
4. Again increment the value until n-1 and repeat the same process.

Finally, all the elements will be sorted in ascending order.

2.2.3 Sample Bubble Sort Java Code

Code adapted from example in: (Bubble Sort - GeeksforGeeks, 2022)

```

public static int[] BubbleSortMethod(int[] arraylist) { // Define method BubbleSort()
to implement Bubble sort of an

    // array of
    // integers.
    int temp;
    long startTime = System.nanoTime();

    for (int i = 0; i < arraylist.length - 1; i++) { // Outer Loop variable
        for (int k = 1; k < arraylist.length - i - 1; k++) { // Inner Loop variable
            if (arraylist[k - 1] > arraylist[k]) {
                temp = arraylist[k - 1];
                arraylist[k - 1] = arraylist[k];
                arraylist[k] = temp;
            }
        }
    }

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

    return arraylist; // Returns array- same array list but is ordered

```

}
}

2.2.4 Diagram of Bubble Sort Working

Index number	0	1	2	3	4
Input sample data	1	6	5	3	9

First pass: As we can see, $6 > 5$, the algorithm compares the first two elements and swaps them over.

Index	0	1	2	3	4
Input sample data	1	6	5	3	9
	1	5	6	3	9
	1	5	3	6	9
After iteration 1	1	5	3	6	9

Steps:

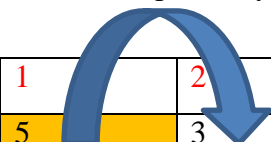
1 6 5 3 9 **Swap** because $6 > 5$

1 5 6 3 9 **Swap**, because $6 > 5$

1 5 3 6 9 **Swap**, because $6 > 3$

1 5 3 6 9 **No Swap** because $6 < 9$

Second Pass: Largest element in the input array bubbles to the right.



Index	0	1	2	3	4
Input sample data	1	5	3	6	9
	1	3	5	6	9
After	1	3	5	6	9

Steps:

1 5 3 6 9 **No Swap** because $1 < 5$

1 3 5 6 9 **Swap**, because $5 > 3$

1 3 5 6 9 **No swap**, the array is already sorted now.

Input data set is sorted: Complete.

2.3 Merge Sort

Merge Sort is another comparison sorting algorithm developed by J. von Neumann in '1945. Merge is a recursive sorting algorithm that uses a divide and conquers approach (Mannion, 2019). It divides the array into two halves. Each half is then further divided into two halves. The division keeps going until the size of the array becomes 1.

Merge sort has three main steps.

1. Initially, array is divided in two halves of the array
2. The two smaller array lists are further divided into smaller sorted lists.
3. The smaller lists merge back into each other resulting in a single sorted array.
4. The final two lists are merged to form one complete sorted array.

2.3.1 Big O Analysis of Merge Sort

In the below table for the Complexity we can see that in all 3 cases (best case, average and worst case scenarios) have similar time complexities ($n \log n$). This shows us that regardless

of the size of the user input data, it doesn't put much strain on the algorithm even with large data sets of $n \log n$ time (Mannion, 2019).

2.3.2 Complexity Big O Notation

Best Case	Average Case	Worst Case	Space	Stable	Class
$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$	Yes	Comparison sort

Table Depicting Merge Sort Specifications (GeeksforGeeks, 2022).

2.3.3 Java Sample implementation of Merge Sort

Code adapted from example in: (Merge Sort - GeeksforGeeks, 2022)

```

public void mergeSort(int[] list, int lowIndex, int highIndex) {
    if (lowIndex == highIndex)
        return;
    else{

        int midIndex=(lowIndex+highIndex)/2;
        //sorts the left side of the array
        mergeSort(list, lowIndex, midIndex); //sorts the right side of the array
        mergeSort(list, midIndex + 1, highIndex);
        merge(list, lowIndex, midIndex + 1, highIndex);
        //merge function that re=joins sorted lists
    }
}

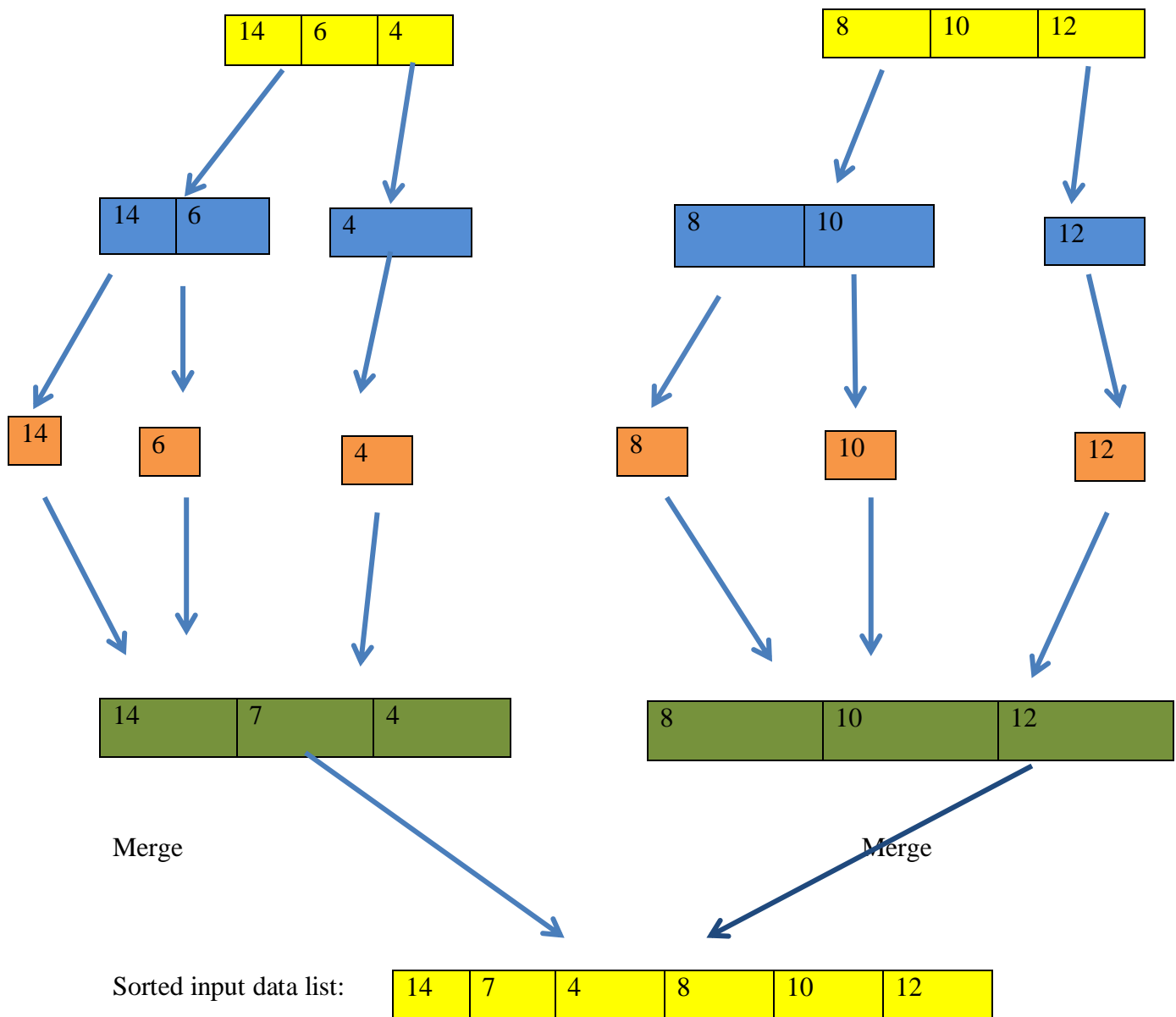
```

2.3.4 Diagram of Merge Sort Working

Input list of integers:

14	6	4	8	10	12
----	---	---	---	----	----





2.3.5 Explanation of the above diagram for Merge Sort.

1. Firstly, the array is split into two halves of the array.
2. First 2- way splitting is followed till we have a single sorted element and after that 2- way merging is done.
3. In 2-way merging the elements of the array/list are merged in such a way that all the elements get sorted in a single array.

2.4 Bogosort

Bogosort also known as permutation sort is a sorting algorithm. It gets its name where the algorithm keeps on making permutations and checks if that permutation gives a sorted array/list (wiki, year). If bogosort were like a deck of cards, one would shuffle the entire deck, choose cards at random, and repeat the process until the entire deck was sorted. Bogosort is an example of a non-stable sorting algorithm, as there is no guarantee that randomly generated sorted permutations have the same relative order of elements of the same value as in the input or not.

There are some requirements when using the bogo sort:

1. Each element in the array can work with positive integer numbers however it can't work with negative integers.
2. It's a non-comparison based algorithm therefore it only works with permutations.
3. It is not considered useful for sorting although it may be used for educational purposes to compare it to other efficient algorithms.

2.4.1 Big O Analysis of Bogo Sort

Best Case	Average Case	Worst Case	Auxiliary Space	Stable	Class
$O(n)$	$O(n*n!)$	$O(\infty)$	$O(1)$	Yes	Comparison sort

Table Depicting Bogo Sort Specifications (GeeksforGeeks, 2022).

The average Big O complexity of this sorting algorithm is $O(n-n)$ which means to sort an array/list of 100 elements, it would take approximately 10^{140} millennia. The worst case scenario Big O complexity for this algorithm may be infinite because the algorithm itself is not finite and does not end in finite time in every case which means that the sorting algorithm may never be able to return a given array list. One of the advantages of bogosort is it is not guaranteed that we get the sorted permutation at some point after shuffling the array/list a certain number of times. Although its probability is infinitesimally low, there's a chance that we do not get the sorted order even after numerous times of shuffling.

2.4.2 Diagram of Bogo Sort Working-

Given Array:

2	3	1	2	4
---	---	---	---	---

1st random permutation:

2	2	3	1	4
---	---	---	---	---

2nd random permutation:

4	1	2	3	2
---	---	---	---	---

Sorted Array:

1	2	2	3	4
---	---	---	---	---

2.4.3 Java Sample implementation of Bogo Sort

```

package ie.gmit.dip;

import java.util.Random;

//Java class for the execution of Permutation/Bogo Sort

public class PermutationSort {
    public static void main(String[] args) {

        // Random array to be sorted here
        Random array = new Random();

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

        System.out.print("Random Numbers Generated:");
        for (int d = 0; d < figure.length; d++) {
            int RandomNumbers = array.nextInt(figure.length) + 1;
            figure[d] = RandomNumbers;
            System.out.print(" " + RandomNumbers);
        }
    }
}

```



```

    long startTime = System.nanoTime();

    PermutationSort now = new PermutationSort();
    System.out.println(" ");
    now.display1D(figure);

    now.permutationsort(figure);

    System.out.println(" ");
    System.out.print("Sorted Array: ");
    now.display1D(figure);

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

void permutationsort(int[] figure) {
    // Keep a track of the number of shuffles
    int shuffle = 1;
    for (; !isSorted(figure); shuffle++)
        mix(figure);
    // Boast
    System.out.println("This took " + shuffle + " shuffles of components in the
array list.");
}

void mix(int[] number) {
    // To generate permutation of the array
    int i = number.length - 1;

```

```

        while (i > 0)
            exchange(number, i--, (int) (Math.random() * i));
    }

    void exchange(int[] number, int i, int j) {
        int temp = number[i];
        number[i] = number[j];
        number[j] = temp;
    }

    boolean isSorted(int[] number) {

        for (int i = 1; i < number.length; i++)
            if (number[i] < number[i - 1])
                return false;

        return true;
    }

    void display1D(int[] number) {
        for (int i = 0; i < number.length; i++)
            System.out.print(number[i] + " ");
        System.out.println(" ");
    }
}

```

2.5 Insertion sort

Insertion sort is a type of sorting algorithm used to arrange elements in an array/list in either ascending or descending order. Insertion sort is an example of an in-place sorting algorithm. In other words the algorithm doesn't require any extra space or memory to sort the data set. As insertion sort is an in-place sorting algorithm, it maintains a sub array within itself.

2.5.1 Step-by-Step Implementation *Steps adapted from example in: (javatpoint/insertion-sort, 2022)*

1. The key is the element at arr (1). This element is first compared with the element at arr (0) or x.

a. If the current element (key) at (arr (1)) is smaller than x, then the key is inserted before x.

b. If the key element (index (1)) is greater than its predecessor, then the key is inserted after x.

2. In this way, the first two elements are sorted. In the next step, the element at index (2) is now considered as the new key. Step 1 is repeated with the new key, and this key is inserted in its correct position.

3. This process is repeated until all elements in the array are sorted.

2.5.2 Big O Analysis of Insertion Sort

Best Case	Average Case	Worst Case	Auxiliary Space	Stable	Class
$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Comparison sort

Table Depicting Insertion Sort Specifications (Javatpoint, 2022).

2.5.3 Diagram of Insertion Sort Working-

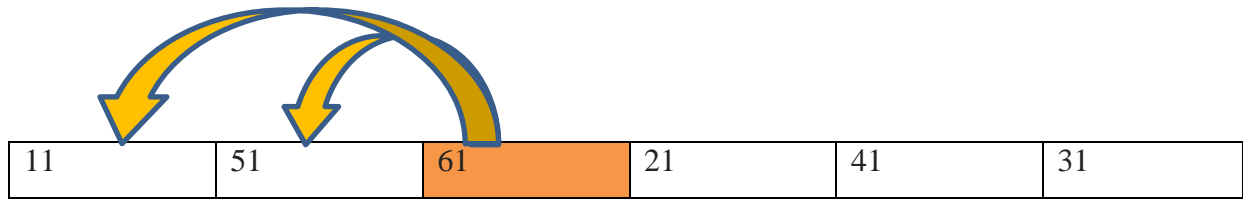
A sample with the input data set 51, 11, 61, 21, 41, and 31

51	11	61	21	41
----	----	----	----	----

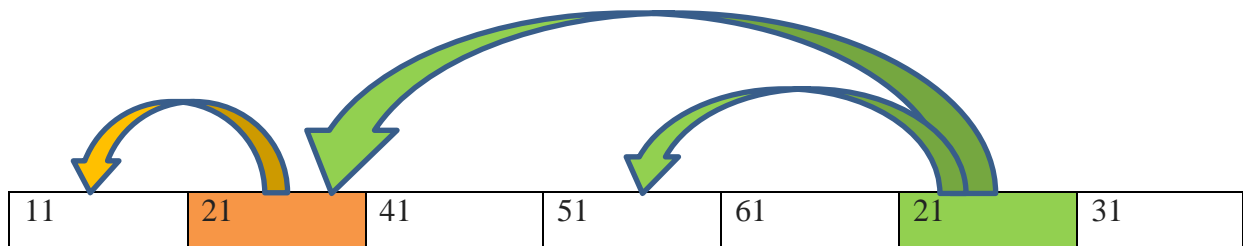
$11 < 51$

51	11	61	21	41	31
----	----	----	----	----	----

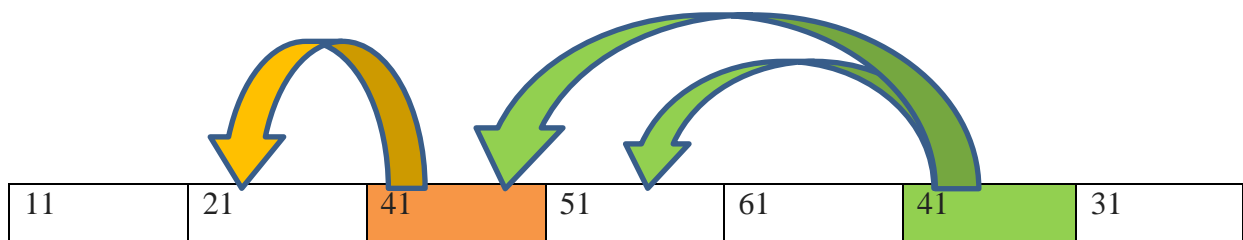
Inserting 11 before 51 $61 > 11$ $61 > 51$



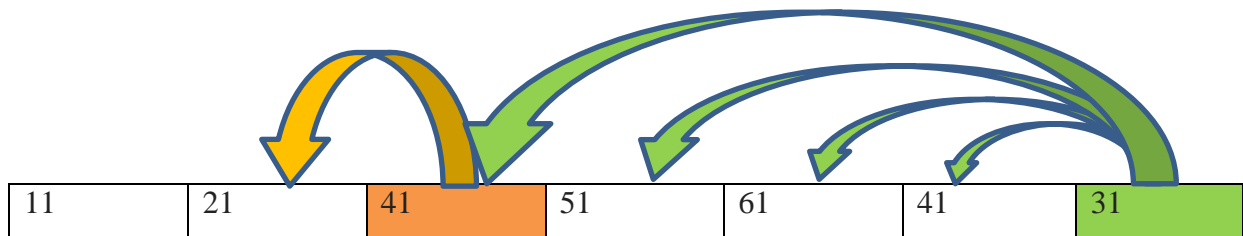
No change required $21 > 11$ $21 < 51$ $21 < 61$



Inserting 21 after 11 and before 51 $41 > 21$ $41 < 51$ $41 < 61$



Inserting 41 after 21 and before 51 $31 > 21$ $31 < 41$ $31 < 51$ $31 < 6$



Inserting 31 after 21 and before 41

11	21	31	41	51	61
----	----	----	----	----	----

Final sorted array list

2.5.5 Java Sample implementation of Insertion Sort

Pseudo Code adapted from example in: (Insertion Sort - GeeksforGeeks, 2022)

```
void insertionSortFunction (int arrSample[], int n)

/*Here arr Sample is the array to be sorted, and n is the number of elements in the array*/

{

    int i, key, x;

    for (i = 1; i < n; i++)

    {

        key = arrSample [i]; //Here the element at index(1) is the key at first.

        x = i - 1; // The variable j will always hold the value prior to the key

        while (x >= 0 && arrSample [x] > key) //comparison between key & x done

        {

            arrSample [x + 1] = arrSample [x];

            x = x - 1;

        }    arrSample [x + 1] = key;

    }

}
```

3. IMPLEMENTATION AND BENCHMARKING

As previously mentioned, all code for this project was written using the Eclipse Che package. I've chosen to generate the input arrays for each of each of the five sorting algorithms of differing input sizes. Arrays of 5 different sizes (0, 10, 100, 1000, 10000) were generated. One issue I struggled with was automating 10 runs for each algorithm as described in the final lecture slides for this module. 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

```

1 // number of times to test the method
2 int numRuns = 10;
3
4 // array to store time elapsed for each run
5 double[] results = new double[numRuns];
6
7 // benchmark the method as many times as specified
8 for(int run=0; run<numRuns; run++) {
9     long startTime = System.nanoTime();
10    methodToTest(input);
11    long endTime = System.nanoTime();
12    long elapsed = endTime-startTime;
13    double timeMillis = elapsed/1000000.0;
14
15    // store the time elapsed for this run
16    results[run] = timeMillis;
17 }

```



Dr Patrick Mannion, Dept of Computer Science & Applied Physics

6

Source taken from (Mannion, 2022)

To solve this issue, 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 in excel.

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

Size	10	sum:	0.024	100	sum:		1000	sum:		10000	sum:	
Bubble sort	0.0037	average	0.0048	0.3544	average		12.0491	average		240.069	average	
	0.0039			0.4968			12.8972			270.235		
	0.0055			0.7729			11.3329			261.98		
	0.0058			0.2424			11.2736			272.762		
	0.0051			0.2647			9.223			230.322		

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

Size	10	sum:	3.3124	100	sum:	30.9828	1000	sum:	137.863	10000	sum:	496.989
Merge Sort	0.4721	average	0.66248	4.8621	average	6.19656	13.6841	average	27.5726	131.199	average	99.3977
	0.6693			11.1144			16.5773			86.4542		
	0.811			4.5759			50.6149			101.452		
	0.6894			5.3434			42.0131			106.001		
	0.6706			5.087			14.9736			71.8824		

Results table3 – Count Sort time output values in milliseconds with differing array input sizes

n.

Size	10	sum:	1.7423	100	sum:	3.4401	1000	sum:	5.8719	10000	sum:	176.595
Count Sort	0.2349	average	0.34846	0.6661	average	0.68802	1.8089	average	1.17438	30.1623	average	35.3189
	0.3711			0.6845			1.0871			12.0415		
	0.6294			0.8179			1.0669			19.0005		
	0.2127			0.5913			0.7994			12.2627		
	0.2942			0.6803			1.1096			103.128		

Results table4 – Count Sort time output values in milliseconds with differing array input sizes

n.

Size	10	sum:	0.021	100	sum:	0.5649	1000	sum:	45.9102	10000	sum:	595.085
Insertion Sor	0.0038	average	0.0042	0.1025	average	0.11298	7.625	average	9.18204	82.9248	average	119.017
	0.0046			0.0778			12.0757			158.685		
	0.0042			0.1676			8.1312			138.148		
	0.0041			0.108			10.0331			111.382		
	0.0043			0.109			8.0452			103.945		

Results table5 – Permutation sort time output values in milliseconds with differing array

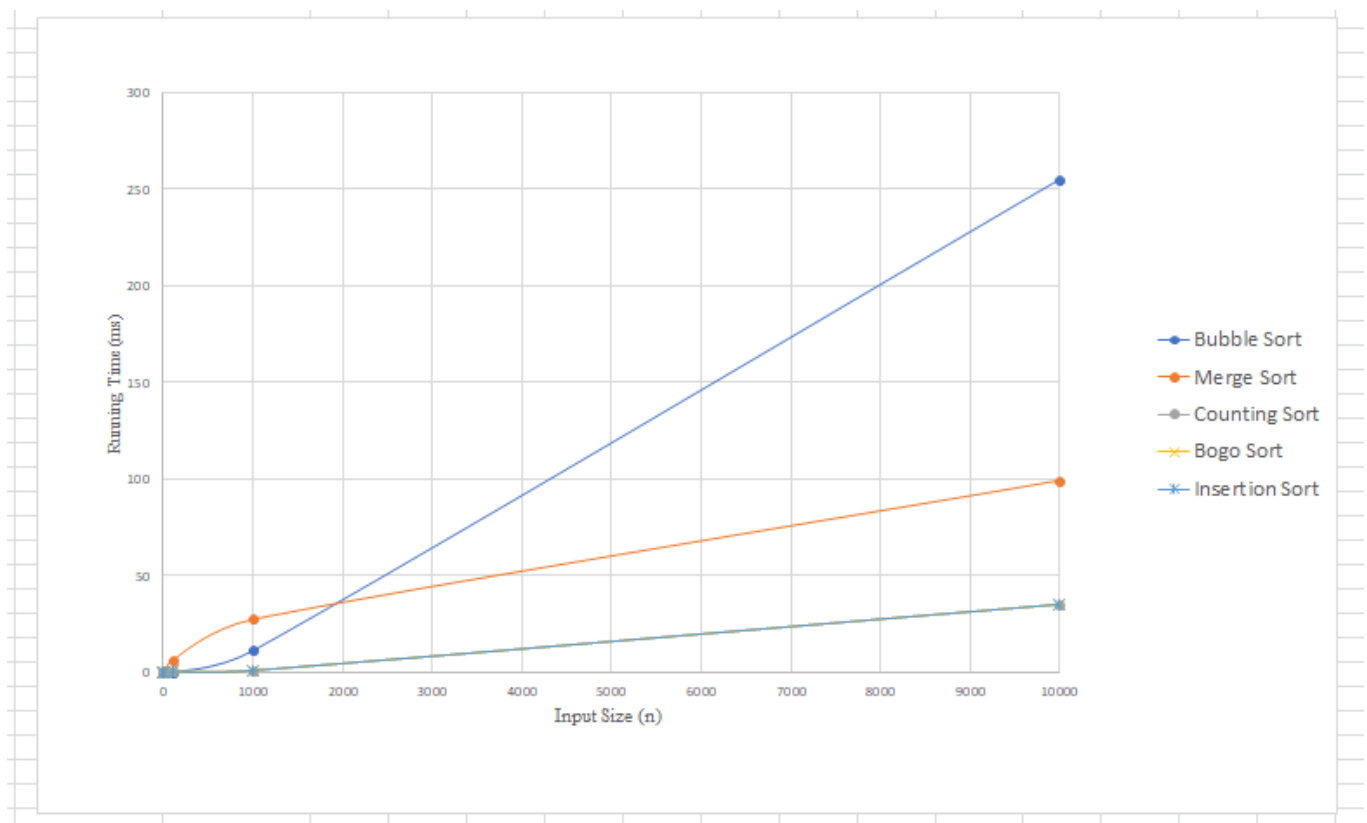
input sizes n.

Size	10	sum:	469.023	100	sum:	∞	1000	sum:	∞	10000	sum:	∞
Permutation	53.9747	average	93.8045	∞	average	∞	∞	average	∞	∞	average	∞
	179.911			∞			∞			∞		
	138.174			∞			∞			∞		
	34.3446			∞			∞			∞		
	62.6191			∞			∞			∞		

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

Input Size Array n	0	10	100	1000	10000
Bubble Sort	0	0.0048	0.42624	11.3552	255.073
Merge Sort	0	0.66248	6.19656	27.5726	99.3977
Counting Sort	0	0.34846	0.68802	1.17438	35.3189
Bogo Sort	0	93.8045	∞	∞	∞
Insertion Sort	0	0.0042	0.11298	9.18204	119.017

Results table – Graph representation of the Big O Notation for each sorting Algorithm Tested



By analysing the time complexity of each of the five chosen algorithms, one was able to calculate the average time in milliseconds in relation to an increased input size of the array. It is clearly evident that each of the sorting algorithms follows a unique pattern similar to their Big O time complexity.

3.1 Summary of expected time and space complexity of the chosen sorting algorithms

Time Complexity	Bubble Sort	Merge Sort	Counting Sort	Insertion Sort	Permutation Sort
Best Case	$\Omega(n \log(n))$	$\Omega(n \log(n))$	$\Omega(n+k)$	$O(n)$	$O(\infty)$

Worst Case	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n+k)$	$O(\infty)$	$O(\infty)$
Average Case	$\Theta(n \log(n))$	$\Theta(n \log(n))$	$\Theta(n+k)$	$O(n*n!)$	$O(n*n!)$
Space Complexity	$O(1)$	$O(1)$	$O(k)$	$O(1)$	$O(1)$

Table Depicting expected time complexity of chosen sorting algorithms (GeeksforGeeks, 2022).

By comparing the time complexity for sorting algorithms with an input array size of 10, Insertion Sort was the fastest function, followed by bubble sort, counting sort, and Merge sort. The slowest function, as expected was Bogo sort. Insertion sort performs better with a smaller user input array. Bubble sort was the second fastest sorting algorithm tested in this assignment. It is regarded as an efficient algorithm with a good space complexity. However on the contrary it is not a stable algorithm and its running time depends on the size of the input array. Bubble sort jumps considerably from 100 to 1000 in running time.

4. BIBIOGRAPHY

Big O Notation. [online] Available at: https://en.wikipedia.org/wiki/Big_O_notation [Accessed 14 May 2022].

Bubble sort. [online] Available at: https://en.wikipedia.org/wiki/Bubble_sort [Accessed 14 May 2022].

Bogo sort. [online] Available at: <https://en.wikipedia.org/wiki/Bogosort> [Accessed 14 May 2022].

GeeksforGeeks. (2022). Analysis of Algorithms | Set 2 (Worst, Average and Best Cases) - GeeksforGeeks. [online] Available at: <https://www.geeksforgeeks.org/analysis-of-algorithmsset-2-asymptotic-analysis/> [Accessed 14 May 2022].

GeeksforGeeks. (2022). Bubble Sort - GeeksforGeeks. [online] Available at: <https://www.geeksforgeeks.org/bubble-sort/> [Accessed 4 May 2019].

GeeksforGeeks. (2022). Counting Sort - GeeksforGeeks. [online] Available at: <https://www.geeksforgeeks.org/counting-sort/> [Accessed 14 May 2022].

GeeksforGeeks. (2022). Merge Sort - GeeksforGeeks. [online] Available at: <https://www.geeksforgeeks.org/merge-sort/> [Accessed 14 May 2022].

Time complexity of sorting algorithms- Javatpoint (2022). [online] Available at: <https://www.javatpoint.com/time-complexity-of-sorting-algorithms> [Accessed 14 May 2022]

Insertion Sort- Javatpoint (2022). [online] Available at: <https://www.javatpoint.com/insertion-sort> [Accessed 14 May 2022]

Mannion, P. (2022). Sorting Algorithms Part 1. Galway: Galway-Mayo Institute of Technology., p.Lecture 2.

Mannion, P. (2022). Sorting Algorithms Part 2 Galway: Galway-Mayo Institute of Technology., p.Lecture 3.

www.tutorialspoint.com. (2022). Data Structures and Algorithms - Bubble Sort. [online] Available at:

https://www.tutorialspoint.com/data_structures_algorithms/bubble_sort_algorithm.htm

[Accessed 14 May 2022].