COMPUTATIONAL THINKING & ALGORITHMS PROJECT 2021

Table of Contents

Computational Thinking & Algorithm: Project

Introduction

It is crucial for data to be arranged in a particular order to be able to access the data efficiently. Sorting can be defined as the arranging of data in a sequence either statistically (ascending or descending order), lexicographically (alphabetical order) or categorically. Sorting is very important as it optimizes data usefulness and the search and retrieval of data can happen much faster. In everyday life for example, finding a word in the dictionary is easy because the words are arranged alphabetically in a sorted form which makes it effortless for us.

According to Cormen, Leiserson, Rivest & Stein (2001) *"An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value or set of values as output"*. A sorting algorithm on the other hand is a procedure which has the ability to rearrange a large amount of items into a precise order according to pre-defined ordering rules.' (w3schools.in, 2021). A large amount of CPU time about 25% is spent on sorting task. This provides an incentive that one has to consider the correct and the most appropriate sorting algorithm for the specific task they wish to accomplish (Mannion, 2021). However, it is not feasible to conclude that one sorting algorithm is superior to another because it's performance depends on what data is being sorted. Sorting algorithms are said to be problem specific that is, they can be useful for a specific problem but not all problems for example some sorting algorithms are useful for a large set of data while some are suitable for data with duplicate values (Mishra & Garg, 2009).

Sorting is such an important aspect in computing due to its simplification of computational task when the data that is being analysed is sorted in advance. This results in the ***easier accessibility*** to the specific data we require, it ***cuts down run times*** and ***supports in time estimation***.

Types of sorting algorithms

Sorting algorithms can be grouped into 3 main types; Comparison sort, Non-comparison sort and Hybrid.

Comparison sort are sorting algorithms that uses comparison operators only to determine which two elements in a set of sorted list or array should occur first in a sequence using a comparator functions such as "≤". These elements are compared with each other to gain information about the total order and have a complexity lower bound of **nlogn** in best average and worst case (Mannion, 2021). Examples of comparison sorting algorithm include Bubble sort, Insertion sort, Merge sort and Quick sort.

Non-comparison sort do not sort by comparison but they rather make assumptions about the data that is being sorted. They use internal characters or keys to reorganise the values of a data set into its accurate sequence. They are also classified as Linear sorting algorithms due to their time complexity of O(n) thus making it much quicker when sorting larger data. Examples of non-comparison sorting algorithm include Bucket sort, Counting sort and Radix sort.

Hybrid sorting algorithm uses a combination of more than one sorting algorithm that solve the same problem. Depending on the type of data, it switches from one sorting algorithm to the other for the

duration of the algorithm. This combination uses the strength of a number of algorithms to achieve better performance that is much better and faster than its individual component. Examples of hybrid sorting algorithm include Timsort and Introsort (techiedelight.com).

Conditions for Sorting Algorithms

A set of items is considered to be sorted if each item that appears in the set or its value is less than or equal to its successor (Mannion, 2021). Based on pre-defined rules e.g. sorting in ascending order, the elements must be rearranged in a way that shows the order of sequence i.e. items of smaller value must appear before items of a greater value. If there is a duplicate item, all elements according to some ***comparator function*** must appear in one continuous block. There can be no value of different elements combined between them. It should also be noted that the contents of the set of items being sorted must also contain the same items before and after sorting. The number of distinct element in X and the frequency of which they occur must be constant before and after the sorting process but their relative ordering is allowed to change.

A Comparator function passes in two elements (x, y) and returns a value that shows the relative order in a sorted output. It compares the values of x & y and determines which comes first of if they're equal e.g.
If x < y return -1
If x==y return 0
If x > y return 1

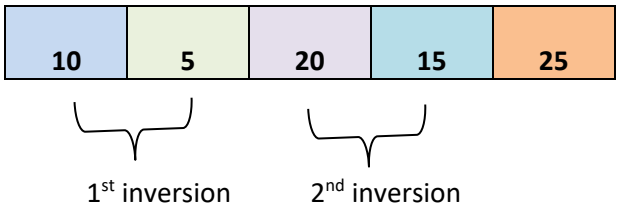
For example using a simple array of elements that consist of **X [2, 1, 3, 3];**

- Starting at index 0 from the left-hand side of the array and working through the elements in the array. The element **X [2]** is greater than **X [1]** as 2 > 1. The array is then reorganised to implement this: **X [1,2]**.
- **X [1,2]** is less than **X [3]** is also sorted as elements 1 & 2 < 3. The array is reorganised to **X [1, 2, 3].**
- The elements at index 2 and 3 which are duplicate items are sorted as **X[3] == X[3].** These elements are placed next to each other in the array. The sorted output becomes **X [1, 2, 3, 3]** which contains the same element before and after sorting.

To be able to sort a collection of item, one has to consider how we want the data to be arranged once it's sorted and the process we would use to achieve this. A ***sort key*** dictates the sequence an algorithm uses to make comparisons between a collection of unsorted data (Mannion, 2021). This is very important in order to get an accurate output as the algorithm requires precise instructions to be able to sort the data as required by the programmer/user. These keys could be based on numerical (using comparator functions) to arrange data in either ascending or descending order or lexicographically ordering.

Another vital concept one has to consider in sorting algorithm is ***inversion***. Inversion is a process whereby a sequence has two elements in a list that are out of their natural order. It indicates

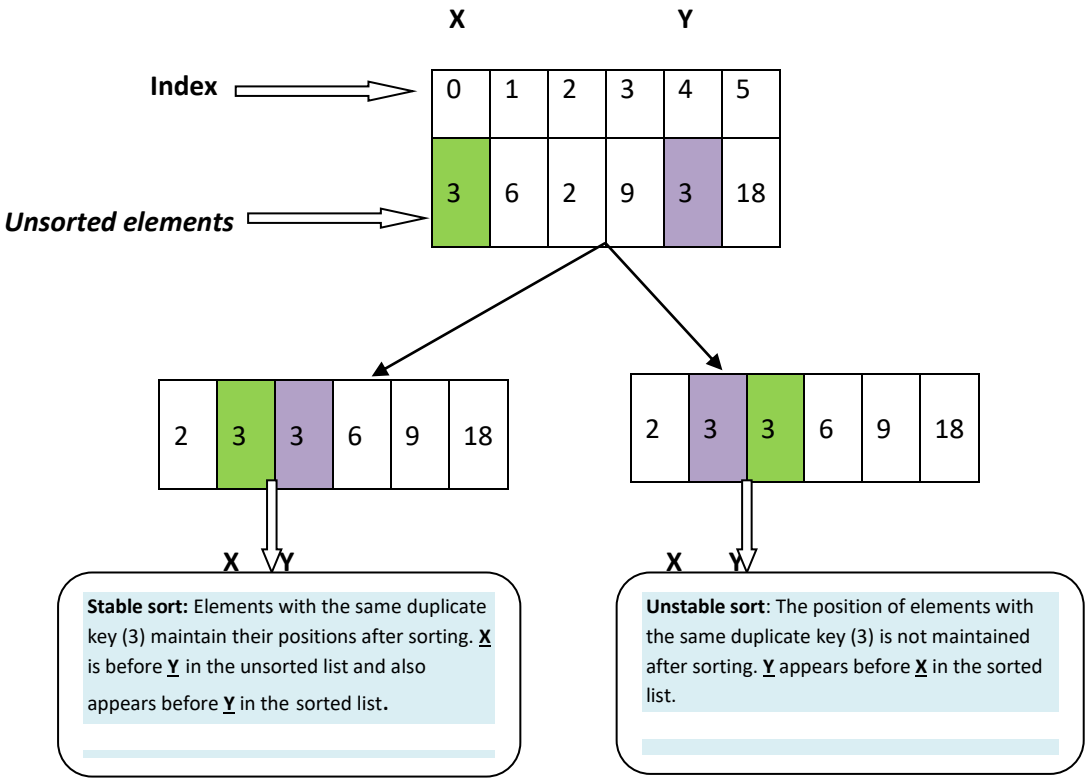
how far an item in a collection is from being sorted (Bakhrani, 2019). If an element on the left-hand side according to a comparator function is greater than the element on the right we can say both elements are out of order. From the example below, there are two inversion [10, 5] and [20, 15].



Classification.

Each sorting algorithm has its varying efficiency as all algorithms are not created equally. Some sorting algorithms are more suitable to a particular task, with specific *needs e.g. specific input size-range, stability, run-time etc.* Hence, it's imperative that the most efficient sorting algorithm is applied to solve a problem. Sorting algorithms can be classified based on the following:

Stability: A sorting algorithm is said to be stable if the order of elements or similar elements maintain their relative positions after sorting. i.e. objects that have the same form of duplicate keys would appear in the same order before and after its sorted (Chauhan & Duggal, 2020).



Adaptivity: A sorting algorithm is said to be adaptive, when it takes into account of elements that are nearly sorted. It considers elements that are already sorted in a list and doesn't change their order (Chauhan & Duggal, 2020).

Performance: Determines the amount of computing time and storage and algorithm needs. e.g. 'the computer speed and quality of the compiler'. Some sorting algorithms would use a set amount of memory despite of the size of input. These types of algorithms are said to be "**in-place".** This means that they only utilise a constant amount of working space to produce or sort an output by only changing the sequence of elements in an array or list.

Efficiency/Complexity

There is a relationship between the complexity of an algorithm and its comparative effectiveness. Time and space complexity serve as a scale of measurement for algorithms (Dehal, 2019).

Space complexity measures the amount of memory or storage an algorithm takes in relation its input size to execute its task and this also depends on the algorithm that is being used (Healy, 2020).

Time complexity denotes the amount of time required by the program to run the algorithm till its completion and this also depends on the algorithm that is being used. By evaluating the running time of an algorithm through comparison, one can deduce an algorithm's growth rate when the input size *n* is increased. The complexity makes the algorithm fall into a particular order which depends on the growth in execution time relating to an increase in input size of *n* (Mannion, 2021). To examine which sorting algorithm has the optimal execution time we must identify its worse case run time to establish its classification.

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

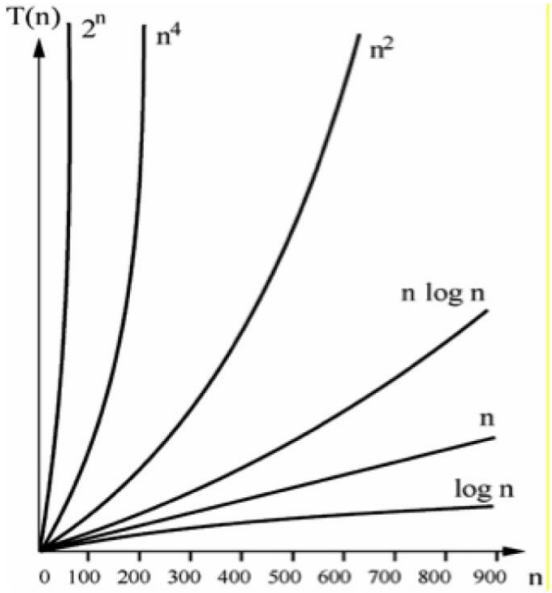


Image *depicting orders of growth (Mannion, 2021)*

Understanding how time complexity increases as a function of input size, provides an objective way of measuring efficiency. The **Big-O** notation (**O** ⇨ Order of magnitude) is the standard terminology and notation for describing the relationship between running time and input size. When the running time of an algorithm, depends on the actual input rather than the size of input, there are three cases to analyse this;

Worse case – refers to the worst running time or slowest time for an algorithm to complete given any input of size *n*, for example searching for a key at last index. If the time doubles as input size *n* doubles, the growth rate is described as *linear.* We say that the growth rate has an order of *n* and a running time of **O**(*n*). The **Big-O** notation looks at worst-case performance (Healy, 2020). In order words the algorithm will not take longer than the worst-case time. It could perform in much quicker time for some input, but it never takes longer than the worst-case time, no matter what the input is.

Best case – refers to the quickest runtime it will take for an algorithm to complete given any input size *n,* for example searching key element present at first index. This doesn't usually happen in reality and it has a constant running time of **O(1)**. The best case is usually represented by Omega notation Ω. (Mannion, 2021).

Average case – refers to an estimate of running time for an "average" input or the expected behaviour when executing the algorithm on random input instances. It looks at all possible cases and the time taken in each possible case divided by the number of cases. It has a running time of ***O(n/2).*** The average case is usually represented by Theta notation θ. (Mannion, 2021).

In practice the worst-case complexity proves to be most useful of these three measures in practice.

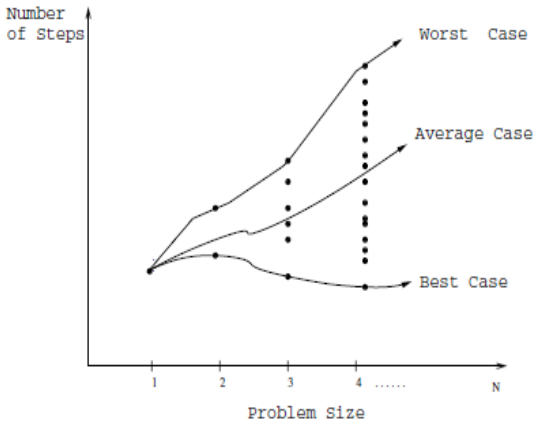


Image representing running time orders of the three cases (Sparrowflights.blogspot.com, 2021)

Sorting Algorithms

For this project I will be focusing on 5 sorting algorithms, discussing and evaluating their time efficiency separately. The chosen algorithms include;

1. Bubble Sort
2. Merge Sort
3. Counting Sort
4. Insertion Sort
5. Selection sort

Bubble Sort

The bubble sort is the oldest and simplest comparison based sorting algorithm. It also happens to be the slowest amongst sorting algorithm. It compares each element in the data set with the element next to it, and swaps its position if they are not in order. The algorithm repeats these swaps until it passes all the way through the list without having to swap any element (Verma & Kumar, 2013). This leads to larger elements moving to the bottom of the list and smaller elements moving to the top of the list. The bubble sort is usually considered to be the most inefficient algorithm in comparison to others. Due of its simple and less complex nature bubble sort can be efficient when the element to be sorted is quite small or already sorted. When the list is already sorted, the performance of bubble sort in best case is **O(n)** and its worst case and average case complexity both are **$O(n^2)$.**

Space and time complexity

Worse Case	Average Case	Best Case	Method	Space Complexity (In-place sorting)	Stable
$O(n^2)$	$O(n^2)$	O(n)	Exchange	1	Yes

Table showing complexity for Bubble Sorts (GeeksforGeeks, 2021).

Best case runtime – in cases where the input elements are already in a sorted order before running the sorting algorithm

Average/ Worst case runtime – in cases where the input element are in a random or reverse order before running the sorting algorithm.

Sample Bubble Sort Code – Java

Code adapted from programiz.com, 2021

```
public class bubbleSample {
    public int[] bubbleSample(int[] array) {
    int size = array.length;

    // run loops two times
    // first loop access each element of the array
    for (int i = 0; i < size - 1; i++){

      // second loop performs the comparison in each iteration
      for (int j = 0; j < size - i - 1; j++){

        // sort the array in ascending order

          // compares the adjacent element
          if (array[j] > array[j + 1]) {

            // swap if left element is greater than right
            int temp = array[j];
            array[j] = array[j + 1];
            array[j + 1] = temp;
          }
        }
    }
     return array; //Return ordered array
}
```

Diagram illustrating how Bubble Sort Works

0	1	2	3	4	5	Index
3	7	2	9	18	1	Unsorted Elements
1st Iteration						
3 >	7	2	9	18	1	No Swap
3	7 >	2	9	18	1	Swap
3	2	7 >	9	18	1	No Swap
3	2	7	9 >	18	1	No Swap
3	2	7	9	18 >	1	Swap
3	2	7	9	1	18	After 1st Iterarion

0	**1**	**2**	**3**	**4**	**5**	Index
3	2	7	9	1	18	Unsorted Elements
2nd Iteration						
3 >	2	7	9	1	18	Swap
2	3 >	7	9	1	18	No Swap
2	3	7 >	9	1	18	No Swap
2	3	7	9 >	1	18	Swap
2	3	7	1	9 >	18	No Swap
2	3	7	1	9	18	After 2nd Iterarion

0	**1**	**2**	**3**	**4**	**5**	Index
2	3	7	1	9	18	Unsorted Elements
3rd Iteration						
2 >	3	7	1	9	18	No Swap
2	3 >	7	1	9	18	No Swap
2	3	7 >	1	9	18	Swap
2	3	1	7 >	9	18	No Swap
2	3	1	7	9 >	18	No Swap
2	3	1	7	9	18	After 3rd Iteration

0	**1**	**2**	**3**	**4**	**5**	Index
2	3	1	7	9	18	Unsorted Elements
4th Iteration						
2 >	3	1	7	9	18	No Swap
2	3 >	1	7	9	18	Swap
2	1	3 >	7	9	18	No Swap
2	1	3	7 >	9	18	No Swap
2	1	3	7	9 >	18	No Swap
2	1	3	7	9	18	After 4th Iteration

0	**1**	**2**	**3**	**4**	**5**	Index
2	1	3	7	9	18	Unsorted Elements
5th Iteration						
2 >	1	3	7	9	18	Swap
1	2 >	3	7	9	18	No Swap
1	2	3 >	7	9	18	No Swap
1	2	3	7 >	9	18	No Swap
1	2	3	7	9 >	18	No Swap
1	2	3	7	9	18	After 5th Iteration

Merge Sort

In comparison to bubble sort, the merge sort is a more-complex comparison sorting algorithm. It is as a result of an algorithm breaking the list to be sorted into halves, placing them in separate arrays. Each array is then sorted out recursively and then put back together to obtain the final sorted output (Verma & Kumar, 2013). The merge sort uses a recursive '*Divide and Conquer*' concept with two parts (partition & merge). In the partition part, if the unsorted list has more than one element, the list breaks in half and the merge sort algorithm is implemented on each half of the list. By the end of the partition phase, the algorithm would have two halves that are sorted. To merge both halves together, the smallest element in one half is compared with that of the smallest element in the other half. The smaller of the two element from the list is moved to the end of the merged list. This process is repeated until both lists are empty (*programiz.com, 2021*).

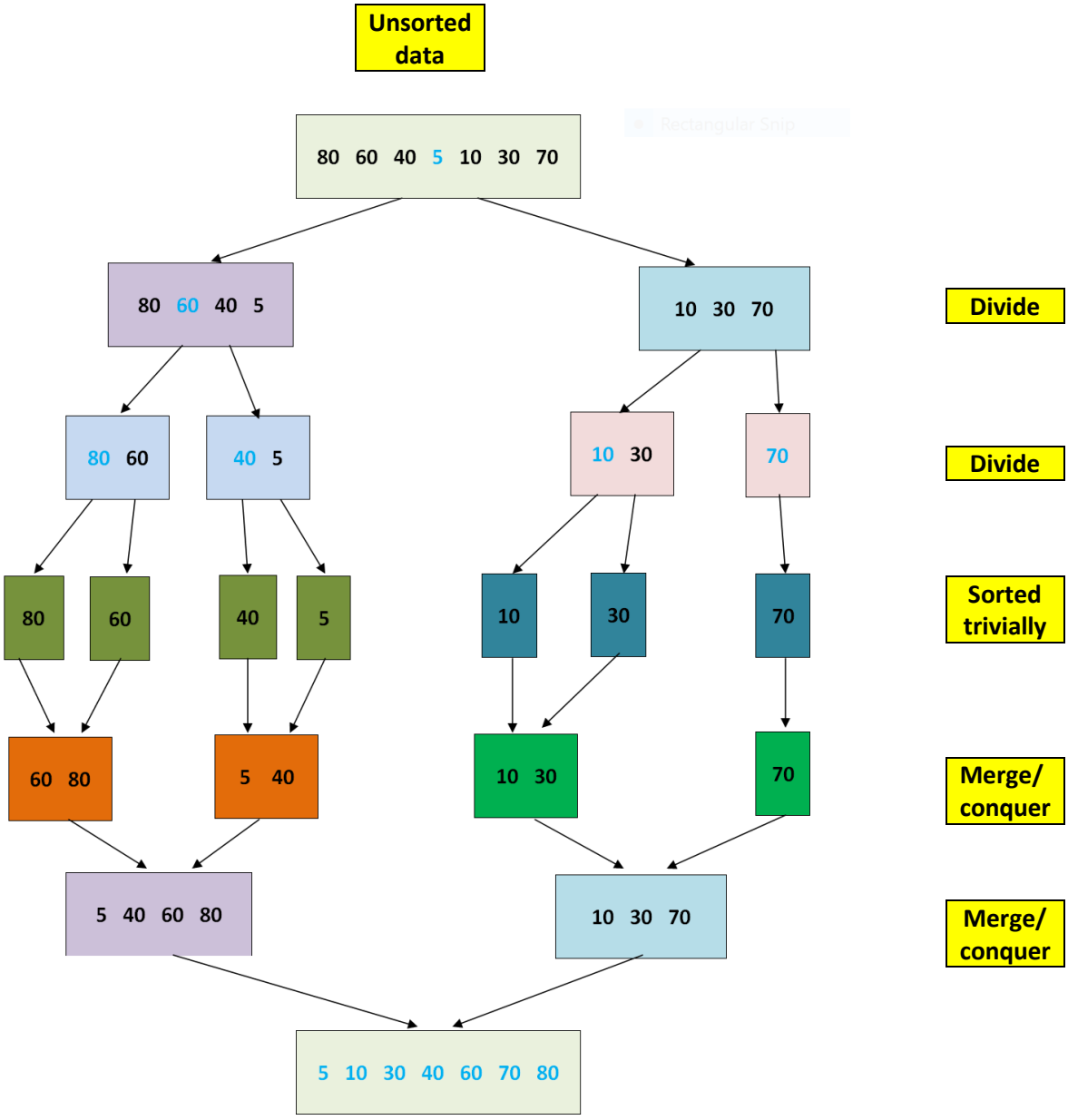
Like most recursive sorting algorithm, it has a time complexity of O(*n* log *n*) across all three cases. This indicates that the size *n* of the input data to be sorted doesn't put much strain on the sorting algorithm (Mannion, 2021). It also makes it easier to predict the algorithm run times in spite of the cases.

Worse Case	**Average Case**	**Best Case**	**Method**	**Space Complexity (In-place sorting)**	**Stable**
O(*n* log *n*)	O(*n* log *n*)	O(*n* log *n*)	Merge	1	Yes

Table showing Time complexity for Merge Sorts

The merge sort is slightly faster for a large set of data, however it requires more memory space due to the second array. It also goes through the whole process even when the list is already sorted.

Diagram illustrating Merge Sort



Steps for Merge Sort:

1. The unsorted data is split in half i.e. into two separate smaller lists.

2. The two separate lists are further split into smaller parts.

3. The elements are broken again into single parts and sorted from smallest to the largest.

4. The split sorted list are merged together to form the final sorted list.

Sample Merge Sort Code- Java

Code adapted from (programiz.com, 2021)

```
public class MergeSort {

    // Merge two sub arrays L and M into array
  void merge(int array[], int p, int q, int r) {

    int n1 = q - p + 1;
    int n2 = r - q;

    int L[] = new int[n1];
    int M[] = new int[n2];

    // fill the left and right array
    for (int i = 0; i < n1; i++)
      L[i] = array[p + i];
    for (int j = 0; j < n2; j++)
      M[j] = array[q + 1 + j];

    // Maintain current index of sub-arrays and main array
    int i, j, k;
    i = 0;
    j = 0;
    k = p;

    // Until we reach either end of either L or M, pick larger among
    // elements L and M and place them in the correct position at A[p..r]
    // for sorting in descending
    // use if(L[i] >= <[j])
    while (i < n1 && j < n2) {
      if (L[i] <= M[j]) {
        array[k] = L[i];
        i++;
      } else {
        array[k] = M[j];
        j++;
      }
      k++;
    }
```

```
    // When we run out of elements in either L or M,
    // pick up the remaining elements and put in A[p..r]
    while (i < n1) {
      array[k] = L[i];
      i++;
      k++;
    }

    while (j < n2) {
      array[k] = M[j];
      j++;
      k++;
    }
  }

  // Divide the array into two sub arrays, sort them and merge them
  void mergeSort(int array[], int left, int right) {
    if (left < right) {

      // m is the point where the array is divided into two sub arrays
      int mid = (left + right) / 2;

      // recursive call to each sub arrays
      mergeSort(array, left, mid);
      mergeSort(array, mid + 1, right);

      // Merge the sorted sub arrays
      merge(array, left, mid, right);
    }
  }
```

Counting Sort

Counting sort is a non-comparison sorting algorithm that is based on pre-defined assumptions i.e. it knows the range of numbers in the array to be sorted. This sorting technique is based on the frequency or count of each element (*consisting of positive integers only*) to be sorted and storing its key values. A new array (*where max is biggest positive integer*) is then created by adding previous key elements and assigning to objects (Srivastava, Jaiswal & Mall, 2018). Based on these assumption there is no need to compare the input elements with each other. For example given a range of numbers in array X (input array), it applies this range to form another array Y (counting array) of this length. Each index *i* in the array is used to check the frequency of elements in array X that have the value *i*. The counts stored in array Y is then applied to sort the elements in array X into its correct order.

Worse Case	Average Case	Best Case	Method	Space Complexity (In-place sorting)	Stable
O(n+K)	O(n+K)	O(n+K)	Counting	O(n+K)	Yes

Table showing Time complexity for Counting Sorts

Counting sort runs on linear time ('O(n)') and can only be applied to positive integers. It has a time complexity of O (*n* + K) where *n* is the number of elements (length of input array) and K is the maximum value that may be present in *n*. Thus, it is suitable for instances where the difference in keys is not significantly greater than the amount of elements in the dataset (Kumar & Singla, 2019). Despite the counting sort being simple to implement however, it is not suitable for large dataset as an equal amount of the output array is needed ***(programiz.com, 2021)***.

Procedure for Counting Sort:

1. Before running the 'Counting sort' algorithm, the key range is determined for the Input data list (determined by the max value in the input array). In the example below the key range is between 1-8. An empty index (*count*) array of the size of range obtained previously is also created. This empty count array is filled with the number of occurrences/frequency of each element.

2. The modified count array (sum of predecessor values) indicates the position of each element in the output. Place each element from the input sequence and decrease its count by 1 and store this value in temp array. Example, Position of 3 is 3. Put the data 3 at index 3 in temp array. Decrease the count by 1 to place next element 3 at an index 2 less than this index.

3. The process is repeated until the end of the input data and the values are arranged.

Diagram of Counting Sort

Input number list:

3	8	2	3	6	5

Number range:

Empty array size:

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

Number range:

Frequency/ count:

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

+ + + + + + +

Sum up count values with predecessor values.

Number range:

Sum of predecessor values:

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

Decrement index array value by 1

Index:

Sorted Output:

1	2	3	4	5	6
2	3	3	5	6	8

Sample Count Sort Code- Java

Code adapted from (programiz.com, 2021)

```java
package ie.gmit.dip;
import java.util.Arrays;

public class CountSortExample {

    void countSort(int array[], int k) {
    int[] output = new int[k + 1];
    // Find the largest element of the array
    int max = array[1];
    for (int i = 1; i < k; i++) {
      if (array[i] > max)
        max = array[i];
    }
    int[] count = new int[max + 1];

    // Initialize count array with all zeros.
    for (int i = 0; i < max; ++i) {
      count[i] = 0;
    }
    // Store the count of each element
    for (int i = 0; i < k; i++) {
      count[array[i]]++;
    }
    // Store the cummulative count of each array
    for (int i = 1; i <= max; i++) {
      count[i] += count[i - 1];
    }
    // Find the index of each element of the original array in count array, and
    // place the elements in output array
    for (int i = k - 1; i >= 0; i--) {
      output[count[array[i]] - 1] = array[i];
      count[array[i]]--;
    }
    // Copy the sorted elements into original array
    for (int i = 0; i < k; i++) {
      array[i] = output[i];
    }
  }
```

Insertion Sort

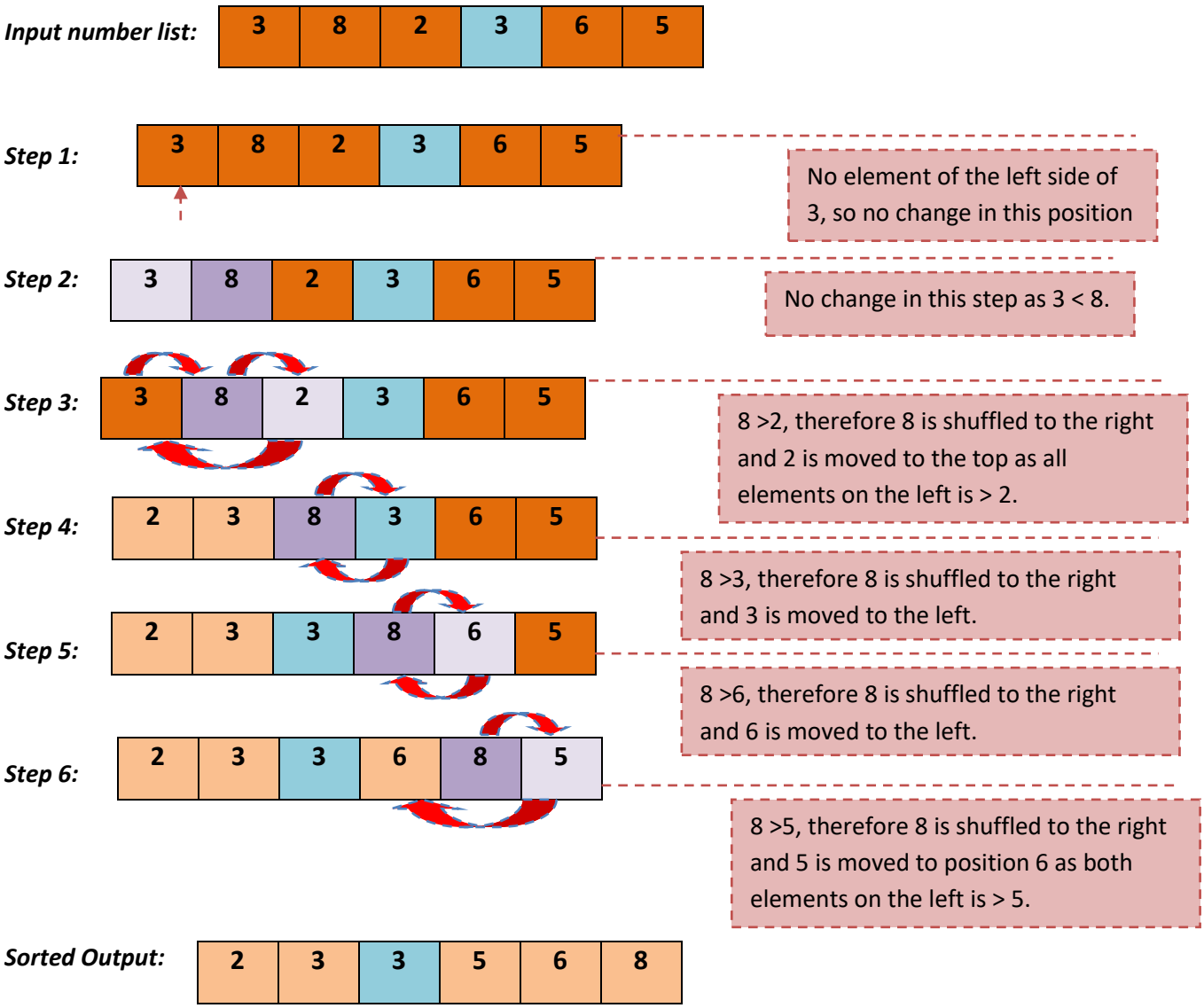
Insertion sort is a simple sorting algorithm which involves inserting every element individually into its correct place in the final sorted output. In insertion sort, the first element [A] is assumed to be sorted, it then moves to the next element. If the next element [B] is greater than the previous element it is placed on the right hand side otherwise it is shuffled to the left. This process is continued until the n-th element is inserted into its position. The total number of steps in insertion sort is reduced if it given a partially sorted list, thus increasing efficiency (Chauhan & Duggal, 2020).

Worse Case	Average Case	Best Case	Method	Space Complexity (In-place sorting)	Stable
O(n)	$O(n^2)$	$O(n^2)$	Insertion	O(1)	Yes

Table showing Time complexity for Insertion Sorts

The insertion sort is basically more efficient than selection sort and a bubble sort, even though all of them have O(n*n) worst case complexity. It is useful for small data sets but very incompetent for larger lists i.e. as input size n increases the algorithm slows down due to its implementation of nested loops to shuffle elements into place.

Diagram of Insertion Sort



Sample Insertion Sort Code- Java

Code adapted from (programiz.com, 2021)

```java
package ie.gmit.dip;

import java.util.Arrays;

public class InsertionSort {

    void insertionSort(int[] array) {
    int size = array.length;

    for (int step = 1; step < size; step++) {
      int key = array[step];
      int j = step - 1;

      // Compare key with each element on the left of it until an element
      // smaller than it is found.

      while (j >= 0 && key < array[j]) {
        array[j + 1] = array[j];
        --j;
      }

      // Place key at after the element just smaller than it.
      array[j + 1] = key;
    }
  }
```

Selection Sort

The selection sort is also a simple algorithm which involves selecting the smallest or largest element in an unsorted list and putting it in its correct order in the final sorted list. The algorithm does this by separating the input list into two parts: [X] a left sorted part of elements and [Y] a right unsorted part. To begin with, the sorted part is empty and the unsorted part in the whole input list. It then selects the smallest element, from an unsorted list [Y], in each iteration, and puts that element at the beginning of the sorted part [X]. The number of passes for a given list, is equivalent to the amount of elements in that list (Chauhan & Duggal, 2020).

Despite it being simple and easy to implement, it is quite inefficient for a large list or array. However it is efficient when The time complexity for selection sort, is the same in all three cases $O(n^2)$, that means regardless of whether the list is sorted or not, it will perform at the same time.

Worse Case	Average Case	Best Case	Method	Space Complexity (In-place sorting)	Stable
$O(n^2)$	$O(n^2)$	$O(n^2)$	Selection	O(1)	No

Table showing Time complexity for Selection Sorts.

Sample Selection Sort Code- Java

Code adapted from (programiz.com, 2021)

```java
package ie.gmit.dip;

public class SelectionSort {

    void sort(int[]num)
    {
        int n = num.length;

        // One by one move boundary of unsorted subarray
        for (int i = 0; i < n-1; i++)
        {
            // Find the minimum element in unsorted array
            int minimum = i;
            for (int j = i+1; j < n; j++)
                if (num[j] < num[minimum])
                    minimum = j;

            // Swap the found minimum element with the first
            // element
            int swap = num[minimum];
            num[minimum] = num[i];
            num[i] = swap;
        }
    }
```

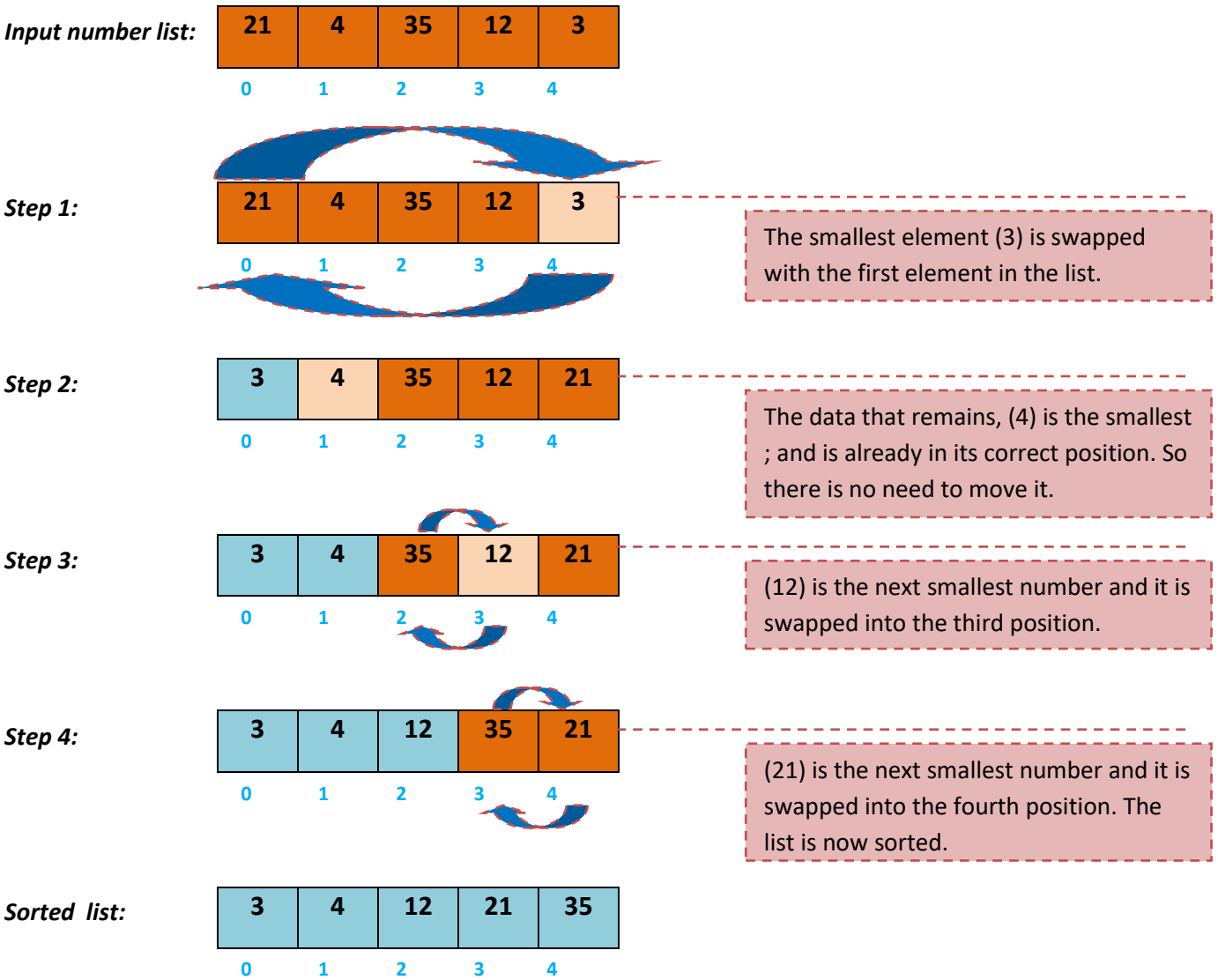
To sort a list of numbers into ascending order,

a) Look through the whole list to locate the smallest element (selecting the smallest element).

b) Swap that element with the element in the first position of the list.

c) Consider the remaining element (from second position to the end) as a new list and the first two steps are repeated until the list is complete.

Diagram of Selection Sort



Implementation & Benchmarking

There are several reasons to analyse the efficiency of an algorithm, the first is to understand how the running time increases based on the size of elements and the second is to compare how well several sorting algorithms will perform when carrying out the same task. Based on this, I adapted sorting algorithm code from *www.geeksforgeeks* and arrays of randomly generated integers (using *Math.random()* method) with different input sizes were implemented using Java to benchmark my chosen algorithms. Bubble Sort, Selection Sort, Insertion Sort, Counting sort and Merge Sort were tested with the random sequence input of size 100, 250, 500, 750, 1000, 1250, 2500, 3750, 5000, 6250, 7500, 8750 and 10000 to check its performance. All the five sorting algorithms were executed on a machine with 64-bit Operating System having Intel(R) Core (TM) m3-6Y30 processor @ 0.90 GHz, 1.51 GHz and installed memory (RAM) 4.00 GB. The running time (in milliseconds) for each algorithm was measured 10 times and the average of the 10 runs for each algorithm and each input size can be seen in *Table 1* below. The plot of input size and running time (in milliseconds) is shown in *figure 1* below.

Size	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
Bubble sort	0.396	0.540	0.564	1.059	1.575	2.326	8.616	20.729	38.952	63.699	93.673	133.418	179.697
Selection sort	0.146	0.466	0.546	0.537	0.534	0.787	2.809	6.253	11.608	16.579	23.379	33.050	41.254
Insertion sort	0.194	0.379	1.436	0.147	0.161	0.244	0.922	2.000	3.756	5.184	7.531	10.758	13.295
Counting sort	0.066	0.077	0.098	0.188	0.042	0.035	0.071	0.021	0.029	0.049	0.034	0.030	0.034
Merge sort	0.169	0.087	0.116	0.170	0.148	0.193	0.370	0.566	1.065	2.362	1.299	1.248	2.484

Table 1: Running Time (in ms) of sorting algorithms on random data averaged 10 runs

From the table above, looking at simple comparison based sort, as expected, Insertion Sort was the fastest amongst the three and Selection sort worked efficiently. However, the relative performance of Selection Sort and Insertion sort varied on different size of input. For example, Selection Sort performed slightly faster with input size of 100 and 500 in comparison to the other two simple comparison algorithms. Insertion Sort was also slightly faster than Merge sort when the input size was 750. Nevertheless, Merge Sort proved to be an efficient comparison based sort when compared to Bubble Sort, Selection Sort and Insertion Sort.

Result also shows that for small input size the performance across all algorithms is all most same, with the exception of Bubble sort which was much slower. I wasn't expecting such difference with a small input size and expected Bubble Sort to have a similar average running time like the other algorithms.

The results also show that non-comparison based sorting algorithm (Counting sort) is faster than comparison based sorting algorithm (Bubble sort, Sort and insertion Sort, Selection Sort and Merge

Sort). It also shows that Counting Sort performed slightly faster with a larger input size than a smaller input size. Despite this, it can be concluded that Counting Sort algorithm performs well for all sizes of input elements as it takes less running time in comparison to Insertion Sort, Bubble Sort, Selection Sort, and Merge Sort for both small inputs and larger inputs.

Based on these result, I plotted a graph for the algorithms to analyze it's performance across various input element sizes. The graph below shows the performance of sorting algorithms for randomly generated integers.

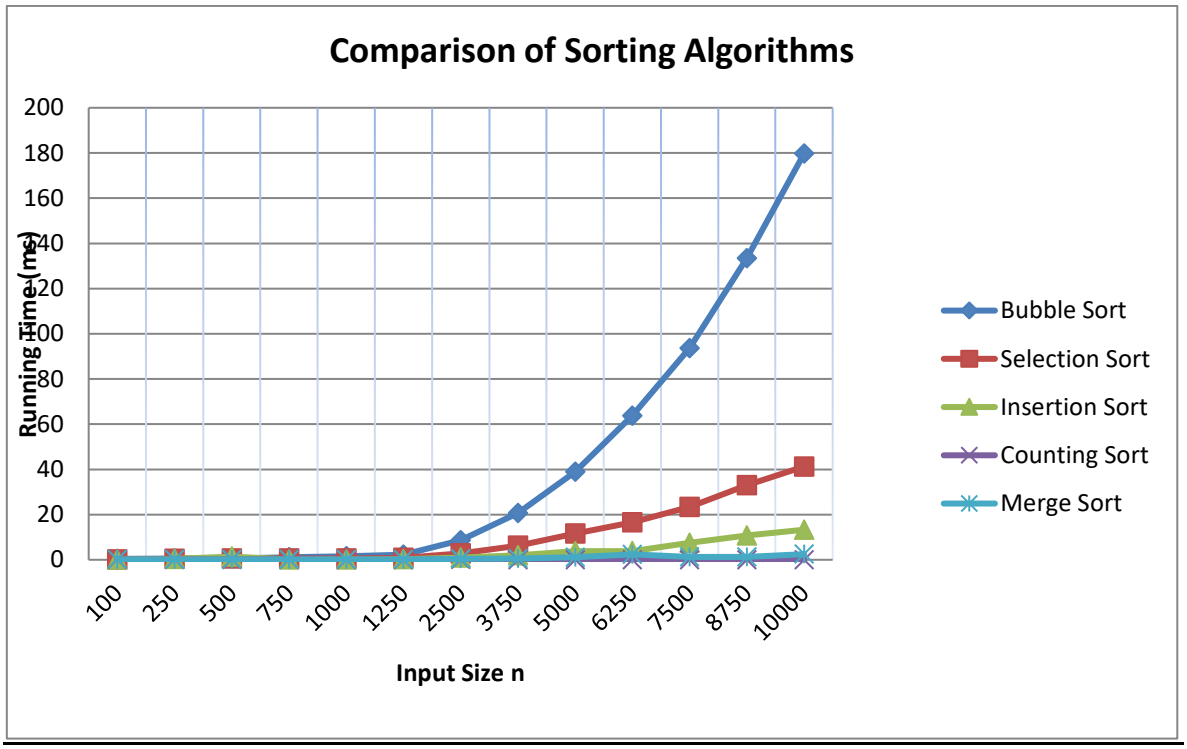


Figure 1: Graph showing the Running Time (in ms) of sorting algorithms on random data averaged 10 runs

Even though bubble sort, selection sort and insertion sort increase asymptotically at the same rate, there is considerable a constant factor between bubble sort and the other two simple comparison based algorithm. However, Insertion sort performs well in comparison to other $O(n^2)$ sorting algorithms tested in this project. This results supports other previous findings that insertion sort is more efficient than bubble sort and selection sort in practice. The graph also show the efficiency of O(n log n) cases (Merge Sort) and O(n+k) cases (Counting Sort) being highly efficient. Although in practice, Counting Sort is proves to be slightly faster than Merge Sort.

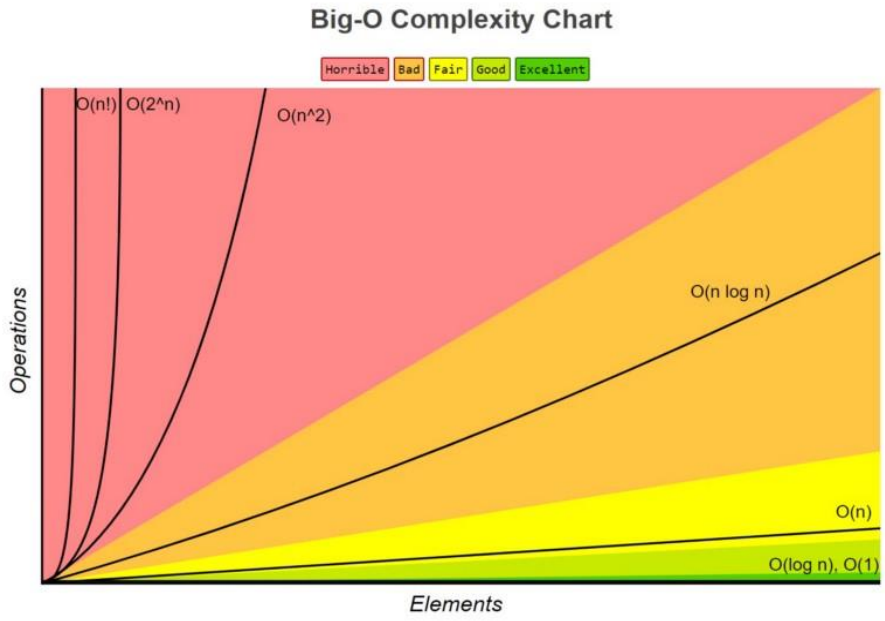


Figure 2: Graph representing Time Complexity.

Conclusion

In conclusion, this project presented the comparison of five sorting algorithms along with the results of its benchmarking. The results proves that an algorithms behaviour with change according to the size of input elements to be sorted.

The running time of simple comparison based sort $O(n^2)$, efficient comparison based sort $O\ (n\ log\ n)$ and non comparison based sort was investigated. Each sorting algorithm, surpassed the other sorting algorithms in certain situations for instance non-comparison based sorting algorithms surpassed comparison based sorting algorithm in large data inputs and $O(n\ log\ n)$ sorting algorithm surpassed the $O(n^2)$ sorting algorithms in large data inputs. This concludes that selecting an efficient sorting algorithm depends on the type of problem and sorting algorithm is problem specific.

References

Bakhrani, H.J (2019) Count Inversions: algorithm for Counting Iversions. Available at: https://medium.com/the-andela-way/count-inversions-5fe3288f11fb [Accessed 20 April 2021].

Cormen, T.H., Leiserson, C.E., & Rivest, R.L. Introduction to Algorithms (2nd ed.). Prentice Hall of India private limited, New Delhi-110001 (2001).

Chauhan, Y. & Duggal, A. (2020) Different Sorting Algorithms comparison based upon the Time Complexity. International Journal of Research and Analytical Reviews (IJRAR). Volume 7, Issue 3. Available at: https://www.researchgate.net/publication/344280789_Different_Sorting_Algorithms_comparison_based_upon_the_Time_Complexity

Dehal, A. (2019) An Introduction to the Time Complexity of Algorithms. Available at: https://www.freecodecamp.org/news/time-complexity-of-algorithms/ [Accessed 20 April 2021].

GeeksforGeeks. (2021). Bubble Sort - GeeksforGeeks. [online] Available at: https://www.geeksforgeeks.org/bubble-sort/ [Accessed 23 April 2021].

GeeksforGeeks. (2021). Counting Sort - GeeksforGeeks. [online] Available at: https://www.geeksforgeeks.org/counting-sort/ [Accessed 23 April 2021].

Healy, J (2020) Big-O Notation. Advanced OOP Galway: Galway-Mayo Institute of Technology., p.Lecture Wk 3.

H. Roy, M. Shafiuzzaman and M. Samsuddoha, "SRCS: A New Proposed Counting Sort Algorithm based on Square Root Method," *2019 22nd International Conference on Computer and Information Technology (ICCIT)*, Dhaka, Bangladesh, 2019, pp. 1-6, doi: 10.1109/ICCIT48885.2019.9038601.

Kristina Popovic (2021) Selection Sort in Java : Available at: https://stackabuse.com/selection-sort-in-java/ [Accessed 23 April 2021].

Kumar, S. & Singla P. (2019) Sorting using a combination of Bubble Sort, Selection Sort & Counting Sort. *International Journal of Mathematical Sciences and Computing*, Vol 2, pp.30-43

Mannion, P. (2021). Sorting Algorithms Part 1. Galway: Galway-Mayo Institute of Technology., p.Lecture 2.

Mannion, P. (2021). Sorting Algorithms Part 2. Galway: Galway-Mayo Institute of Technology., p.Lecture 2.

Sorting Techniques (2021) Available at: https://www.w3schools.in/data-structures-tutorial/sorting-techniques/ [Accessed 20 April 2021].

Sparrowflights.blogspot.com. (2021). Big Oh notation explained. [online] Available at: http://sparrowflights.blogspot.com/2011/01/big-oh-notation-explained.html [Accessed 25 April 2021].

Srivastava, S., Jaiswal, U. C., & Mall, S., (2018) An innovative Counting Sort Algorithm for Negative Numbers. *International Journal of Applied Engineering Research.* ISSN 0973-4562 Volume 13, Number 1 pp. 195-201.

https://www.techiedelight.com/inversion-count-array/ [Accessed 25 April 2021].

Verma, A. K. & Kumar, P. (2013). List Sort: A New Approach for Sorting List to Reduce Execution Time.https://www.researchgate.net/publication/258114251_List_Sort_A_New_Approach_for_Sorting_List_to_Reduce_Execution_Time

https://www.programiz.com/dsa/bubble-sort [Accessed 20 April 2021].
https://www.programiz.com/dsa/insertion-sort [Accessed 20 April 2021]
https://www.programiz.com/dsa/selection-sort [Accessed 20 April 2021]
https://www.programiz.com/dsa/counting-sort [Accessed 20 April 2021]
https://www.programiz.com/dsa/merge-sort [Accessed 20 April 2021]