Authors: Mpendulo Bungane (216810108) // Deregistered
Khanya Gqirana (216057361)
Sitembiso Caleni (216277361)

COMPARISON OF SORTING ALGORITHMS

1. INTRODUCTION

In the ever growing technological era, data manipulation is extensively used. The need for a comprehensive display of data is thus significant. In computer science a sorting algorithm, is a sequence of steps used to put certain elements of a list in a predefined way (chronological order, either ascending or descending). (Rouse, n.d.) The variety of lists to be sorted induces an in-depth study of sorting algorithms, with the focal point of the study being the comparison of the algorithms. Six sorting algorithms are explored in this document. An in-depth description and implementation of each algorithm is provided. The criteria in use for the comparison of the sorting algorithms is: stability, memory allocation, instruction code, execution time etc. The implementation of each algorithm is expanded to sort lists of varying sizes to highlight the strengths, weaknesses and efficiency of each algorithm. Relevant data will be tabulated and graphs displayed. Lists will be sorted in descending order. The assumption is that the lists to be sorted will be 'unsorted', random and the number of items to be sorted is unknown. The limitations of the comparison of the eight algorithms is that items of the lists to be sorted are integers only (maximum of 32 bit in value). This limits the ability to suss the full extent of the strengths or weaknesses each algorithm has with sorting certain data properties (e.g. strings). With the duplications of integers in the list being only 30%, the strength of algorithms that thrive in performance with high duplication of items will be minimized. Moreover, the random function might cluster integer of the same value together, which might impede the observation of efficiency. The methods of the Stop-Watch class in c# will be used to measure the execution time of each algorithm. The Stop-Watch class utilises the built in timers offered by the operating system. Furthermore, the class has a property called *'IsHighResolution',* which determines the configuration of the Stop-Watch. If this property is disabled, the Stop-Watch proves to be inefficient. (Ellen, n.d.) Another limitation with using Stop-Watch class is that its efficiency varies with the systems processing power. (MicroSoft, n.d.)

2. CRITERIA

- **Stability of the algorithm:** A stable sorting algorithm preserves the sequence of similar elements after sorting according to the parameter of sorting (Tutorial, n.d.). An unstable sorting algorithm changes the sequence of similar elements after sorting. (Tutorial, n.d.)
- **Is the sorting method in-place or not:** When sorting algorithms do not require any extra memory, where the sorting is executed in-place this is called 'in-place'. Examples of this would be bubble sort, selection sort, insertion sort (Tutorial, n.d.). When sorting algorithms require memory which is equal to or far greater than the elements being sorted, this is called 'not-in-place' sorting. An example of this would be Merge sort.
- **Whether the sorting method is adaptive or non-adaptive**: A sorting algorithm is adaptive if it can recognize elements that are already sorted in the list and take advantage of this by not trying to sort elements that are already in order (Tutorial, n.d.). A sorting algorithm is non-adaptive if it cannot recognize elements that are already sorted and goes through sorting every single element regardless of whether they were in order already or not. (Tutorial, n.d.)
- **Whether the sorting method is a comparison or an integer sorting method**: In a comparison-based sorting algorithm, the elements are compared to each other according to a certain parameter to find the sorted array (Neelam Yadav, 2016). In an integer sorting method does not make use of comparisons between the elements of the list.
- The number of **instructions used to code the sorting method** (a measure of programmer effort): This is a measure of programmer effort by the amount of lines of code that make up the algorithm.
- **Execution time:** Execution time is a measure of the time cost of the algorithm, the computers clock is used to measure the run time of the algorithm for different sets of data of different sizes (Stephans, 2013).
- **Number of comparisons** made during sorting: The number of comparisons made refers to the amount of times the algorithm compares an element to another before deciding whether the element needs to be swapped with another according to the sorting parameters.

- **Number of swops** made during sorting: The number of swaps refers to the refers to the number of times an element is swapped with another during the process of sorting (Neelam Yadav, 2016).

3. BASIC SORTING ALGORITHM

The **Iterative Insertion (IS)** algorithm works in the following way. The starting index, the length of the array list and the array list itself are passed through the algorithm. On the ith pass through the list, where i ranges from 1 to the length of the array list minus one, the ith item is removed from the list and inserted into its proper place among the first i items in the list after according to which item is greater or lesser. After the ith pass, the first i items are sorted. Insertion sort is a stable sorting algorithm that preserves the order of items that are already sorted within the list. The sorting algorithm is an in-place algorithm. The algorithm is adaptive and does not attempt to sort items that are already sorted. The algorithm is a comparison-based algorithm and performs best with smaller sets of data that contain items that are already in order.

The **Recursive Insertion Sort(RIS)** algorithm's basic operation is described by the following process:
The method continuously calls itself while with the parameters passed to it being an arraylist(AR) and the number of items (n) in the AR still to be processed, with the items to be processed decremented on each subsequent call. The terminating condition of the method is when (n <=1). An integer (last) is assigned to the last item in the list with a variable (j) to keep track of its positon. This integer is inserted in the correct position in the list. The algorithm continues to sort the remaining i^{th} items, it moves elements of AR[0…i-1] that are less than AR[j] one position down of their current position (Stephans, 2013, pp. 132-134). The RIS does not alter the position of items which are of the same value, thus making it stable. No extra memory is required to solve the list using RIS, it's an in-place sorting algorithm. RIS is an adaptive sorting algorithm as it makes use of 'sorted' portion of the list and finds correct position to insert i^{th} element . The i^{th} element to be inserted in sorted portion of the list is compared to the existing items in this list to be inserted in its correct positon. (Tutorial, n.d.) RIS is comparative sorting based algorithm. $(\text{Sonal Tuteja, n. d.})$ The algorithms performance increases with smaller lists and lists with items that are mostly sorted. The algorithm is very slow when sorting a list with a lot of items. The worst case time complexity of the algorithm is $O(N^2)$, with an average of $O(N^2)$ and when the algorithm exhibits its greatest strengths it has a time complexity of $O(N)$. $(\text{Sonal Tuteja, n. d.})$

4. OPTIMISED SORTING ALGORITHMS

The iterative and recursive IS are most effective for small lists with minimal repetitions. What happens when the list has many repetitions? The basic sorting algorithms described above can be further advanced to cater for repetitions in lists also manipulating the 'pre-sorted-ness' of the list. The benefit of this advancement is an improved time complexity.

4.2. ENHANCED INSERTION SORT

The Enhanced Insertion Sort (EIS) is an enhancement to the IS. Inserting a new element at desired place in an already sorted part of an array and decreasing the number of comparisons, is how (IS) is improved (adaptive). The difference in sorting process in the aforementioned algorithms being that the first element in the array is considered the first of an already sorted portion. (Tarundeep Singh Sodhi, 2013, February) Like the IS, the EIS is spilt into 2 portions 'sorted' and 'unsorted'. Upon each iteration, the i^{th} element is compared to the first element List[0]th, which in the case of a list to be sorted in descending order, the List[0]th element is the max of the list. In the case that the i^{th} element is greater than the first element (List[0]th) position, the i^{th} element will become List[0]th element and thus traversing the remaining elements one position to the right. If the i^{th} element is less than List[0]th element, there is no swapping(insertion), then the i^{th} element is compared with the $(i-1)^{th}$ element (i.e. the last in the sorted portion), if the i^{th} element < $(i-1)^{th}$, element, the sorted portion of the list is expanded to include the i^{th} element. The i^{th} element is used as a divisor of the two portions of the list. With none of the aforementioned conditions being met; a binary search is used to find the position of where the i^{th} element should be inserted in the correct portion of the list. Upon sorting elements of the same value, the order is preserved thus EIS is stable. The EIS uses no additional memory to sort lists, thus it's in place. The EIS is a comparison based algorithm. The EIS is advantageous in that it has minimal comparisons.

The increase in repetitions of elements in a list advances its average complexity to $O(n^{1.585})$. Furthermore, the number of comparisons made in EIS plays a pivotal role in this average time complexity. In the best case scenario i.e. elements in descending order, the complexity of the EIS is O(n). In the case that the list is sorted in ascending order, there are 1+(n-2) comparisons where n is the size of the list (Tarundeep Singh Sodhi, 2013, February, p. 4). Moreover, the complexity is $O(N)$, which is significantly less as compared to IS. The number of swaps (IS) is the same with EIS.

4.3. IMPROVED INSERTION SORT

The Improved Insertion Sort(ISS) algorithm makes use of an array b of randomly placed integers with duplicate integer values. To optimize the efficiency of the algorithm a sentinel is used (Nevalainen, O. Raita, T. & Thimbleby, H. , n.d.). It's strengths are: it seeks the highest value in the list with the smallest index, which will be termed the sentinel integer (Nevalainen, O. Raita, T. & Thimbleby, H. , n.d.) . The sentinel is thus placed inside b[0] (Nevalainen, O. Raita, T. & Thimbleby, H. , n.d.) , which takes constant time O(1) when done, using an unstable sort method (Nevalainen, O. Raita, T. & Thimbleby, H. , n.d.) . The sentinel also does not use any extra space in the list, thus in-place sorting is applied. The largest element in the array from index: 1 to the end of the list is found by using linear time in it's worst case performance O(N) . The best case performance would have been O(1) if it was in the position next to the sentinel integer. Each element between the boundary of the largest element in the list and the sentinel integer b[0] is shifted one position up towards the largest elements position. Then the largest element is placed at b[0]. This is done to ensure stable sorting is maintained within this boundary (Nevalainen, O. Raita, T. & Thimbleby, H. , n.d.) . Since b[0] is the new sentinel integer , we note that b[1] is automatically sorted in the preferred position, since the list is sorted in descending order (Nevalainen, O. Raita, T. & Thimbleby, H. , n.d.) . The last step is done by using stable sorting from index x=2 of array b until the last index of the array (Nevalainen, O. Raita, T. & Thimbleby, H. , n.d.) . An efficient insertion sort algorithm will be used. The outer loop will point to each element in the list, with the worst case time complexity of O(N) , while the inner loop will use a comparison method to compare the two elements , hence its worst case performance will be O(N). Thus the outer and the inner loop provide the algorithm with a worst case performance of O(N^2). Inside the inner loop is the swap method. It sorts the elements in the list in descending order. It also allows for adaptive sorting because when comparing two elements, using the comparison method: prevOne.CompareTo(b[x]) < 0 where prevOne is the b[x-1] element in the list. It will not swap the elements if the comparison is not true.

5. ADVANCED SORTING ALGORITHMS

Consider a list with non-uniform distribution of integers, what happens when the list has a million integers? Now consider a list with many repetitions which are mostly relatively sorted, with a few unsorted items lingering in between. Would an O(N) algorithm be efficient to sort the list? The aforementioned algorithms fall short in satisfying the preceding questions. The development of 'Advanced sorting algorithms' is necessary to overcome the shortcomings of the aforementioned algorithms.

5.2. QUICKSORT ALGORITHM

The Quick Sort(QS) works on a divide and conquer method. A list is divided into two portions, with the divider being the point of the split. The divider is chosen to be the first term of an unsorted portion. The QS recursively calls itself to sort the portions of the lists independently. When sorting a portion, the QS makes use of two pointers, 'hi' and 'lo'. These preceding pointers are used to point to the first(lo) and last(hi) term. On the first iteration, lo and the divider are on the same point with hi being on the last term. The divider is compared to hi, if hi is greater than the divider, hi and divider will be swapped; now the pointer hi will move towards the divider and lo (decrement hi). If the divider is greater than hi, lo pointer will move one position up. This process continues until the divider is in the correct position. The QS will continue to call itself until list is sorted. A list is considered sorted when it has one item. In some instances, the sequence of the same elements is not retained hence QS is not stable. The space used in QS is for storing the calls of the recursive function of the stack, and not items of the list thus making QS in place. QS is comparison based since it compares elements within list with each other. Moreover, QS is adaptive, the divider is chosen as the first term of unsorted portion in the list. When a list is evenly distributed, this is when the QS exhibits its best time complexity. The best case performance is $O(N \log N)$. In the case that the divider is greater than all the items in the unsorted portion of the list, the performance is $O(N^2)$. The QS also has a worst case performance ($O(N^2)$) when all the items in the list have the same value. QS is particularly useful for lists of non-uniform distribution and lists with many items. The average performance leans towards $O(N \log N)$, it must be noted that the choice of the divider plays a role on the performance of the algorithm. (Stephans, 2013, pp. 145-153)

5.3. BUCK SORT ALGORITHM

A defined array b [] has a uniform distribution of integer values from the range 0 to (int). Max values of size V. A group of buckets is defined by a static array c [] of size S. Each element in the static array is chained to a dynamic data structure known as a Singly linked list, where its size can expand to a non-constant size. To understand the range of distinct values that can fit in each bucket, the algorithm makes use of a ceiling, which is the upper bound of the max values in b [] divided by the number of buckets (S). To decide which bucket to place the values of b [] in, the algorithm makes use of an integer sorting method by finding the floor (lower bound) of the list value b [] divided by the ceiling value. This floor value acts as an index (bucket number) to the bucket c []. To be able to decide the bucket number each value in b [] will be placed in, it will take G(V/S) where G is the execution time function to complete (Stephans, 2013, pp. 158-159). The total execution time for all the buckets will be O (S * G(V/S)) (Stephans, 2013, pp. 158-159). Once the bucket number is known. A comparison sort for the elements of the buckets is used. A cell pointer is created to compare the value in b [] with the current cargo value of the dynamic data structure. Not in-place sorting is a result, since a new data structure is used to sort the values in b []. If the cargo value is greater than the value in b [], the value b is added before the cargo value, it is noted that a new cell pointer is defined to complete the adding. Or else, if the cargo value is less, the initial cell pointer traverses to the next cell in the list. If it happens the pointer reaches null reference, the value b is chained and added to the list to become the last cell, using a new cell pointer in memory. This process allows for adaptive sorting to occur. In the situation where the value b is identical to the cargo value, the cell pointer will continue to traverse the list without changing the order or sequence of the cells. Thus, stable sorting is maintained. Once all the values in b [] are added in the buckets, they override the original array b [] starting with the elements in the first bucket c [0] to the last one c[S-1]. This will take a time complexity of O(V) and an additional O(S) when disregarding the empty buckets (Stephans, 2013, pp. 158-159). Thus, the total time complexity of the algorithm is O (V + S) (Stephans, 2013, pp. 158-159). If it happens that S<V then the total time complexity is O(V) (Stephans, 2013, pp. 158-159). A great strength about the algorithm is it will operate well for data structures with uniform distributed data within a range. A weakness is that extra memory is used in the creation of a pair of cell pointers for the traversing and adding in the dynamic data structure of each bucket. This weakness can be solved by reducing the number of buckets. This will reduce the pair of pointers found in each bucket.

6. EXPERIMENTAL PROCESS

Experiments for our algorithms will be conducted on an acer, with installed RAM of 6GB (5.68 usable) with a Processor of the following specifications: Intel(R) Core™ i5 CPU M580 @ 2.67Ghz. The system type is 64-bit operating system (Windows 10 Enterprise). The the input for each algorithm consists of four different lists of integer values. Each list has different sizes, that being small, medium, large and very large and they are populated with following sizes of integer values respectively: 348, 3800, 11375, 23000. The lists are populated with random integer values, with 30% of them being duplicated at least 5 times. The following measures will be considered: The number of instruction used to execute the algorithm, this will give an indication of the complexities and effort required to execute the algorithm. The number of comparisons made within the algorithm, this highlights the efficiency of the algorithm. Execution time of the algorithm, the ideal situation is that the algorithm sorts the list as quick as possible. Number of swaps made; show how the algorithms treats a list with items that are repeated (potentially).
Data will be collected in various ways. The number of instructions used to code the algorithm will merely be found on c# as the program has a tab displaying them. To Keep track of the comparisons and swaps, a counter variable will be used in each algorithm, and where objects are compared/swapped the variables will be incremented. The comparisons and swops for each algorithm will be tabulated. In attempts to confirm the theoretical time complexity and which algorithm is the most efficient, elapsed ticks vs list size will be graphed. The programmer effort will be compared with time complexity. On assumption that number of instruction lines are inversely proportional to time complexity, deductions will be made about an algorithm (efficiency, and which algorithms are preferred over it if any). The number of comparisons and swops made within each algorithm will be compared against other algorithms. Based on the different strengths and weaknesses of the algorithms, the criterion for 'best algorithm' needs to be defined as the

algorithms will outperform each other under different circumstances. The best algorithm is one that has the following characteristics: The least number of comparisons, least number of swaps Generally, the fastest execution time irrespective of the size of the list. An average time complexity of O(N) or less. An algorithm with the least amount programmer input but also yielding the highest results. An algorithm that is in-place. Based on how much an algorithm satisfies the 'best algorithms' characteristics, the one possessing the most character traits will be said to outperform the one possessing the least characteristic

7.RESULTS AND DISCUSSION

	Small		medium		large		very large	
	swops	comps	swops	comps	swops	comps	swops	comps
RIS	37047	383	3546121	3799	n/a	n/a	n/a	n/a
EIS	38650	767	3602183	7599	35761779	30348	168157041	76347
QS	1166	7094	15479	167264	16645	174358	118766	2218902
BS	0	4389	0	338706	0	3574011	0	18554993
ISS	6972400	7050282	72421406	72506888	274672677	274780909	765534429	765688661
IS	37047	37427	4286075	4290637	61481499	61501618	378617900	378676575

Table 1: Comparison and swops of each algorithm

A relatively 'low' programmer's effort is required in the IS and the RIS of the algorithms. With the assumption noted in the experimental method, the low number of instructions yields a time complexity of $O(N^2)$ and this is depicted by the graph below. The inverse is true with the advanced sorting algorithms. Admittedly, these results may be obscured in that there are efficient ways to reach a solution which would yield 'lower' programmer effort. However, as is with most things, the greater the effort the better the results.

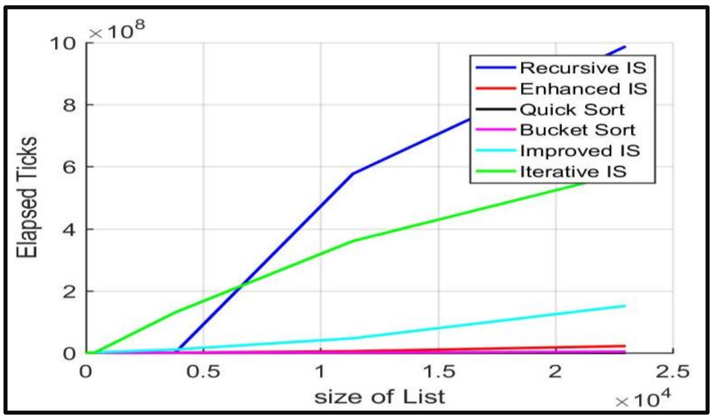

Figure 1: Size of List versus Elapsed Ticks (Time Complexity)

With 'small to medium' sized lists, both the IS and RIS insertion sort time complexities are directly proportional to the size of the list. As the list size is varied to 'large' and' very large', the elapsed ticks exhibit an exponential behaviour. The strengths of both algorithms are highlighted with small sized lists and weaknesses shown with very large lists. The performance of the IS and RIS is $O(N^2)$, as depicted by figure 1, with RIS more exponential in nature due to recursion. The elapsed ticks for EIS and ISS are proportional to the size of the list, irrespective of the size of the list. The BS and QS lie almost on the x-axis of the graph, this is owing to their performance and the size of the lists. From Table 1, The IS has the most number of swops and comparisons. The RIS crashes for large lists, hence swops and comps can't be determined.

8. CONCLUSION

The IS and RIS have similar characteristics. The theory is confirmed by practical results in this regard. The ISS and EIS recorded performance, show how they are an improvement to IS for large lists ($O(N)$). The number of swops for EIS and (IS), is more or less the same as is defined in theory. The number of comps for EIS is significantly less than IS. Oddly ISS has a greater swops and comps than IS which could be due to programmer effort. The advanced algorithms performance for the graphs confirm the theory with them being practically on the axis. Table one further solidifies the theory with the swops and comps of QS and BS being low. It can thus be concluded that the basic sorting algorithms are suitable for lists with small sizes(items) and advanced algorithms for lists with large sizes.

9. BIBLIOGAPHY

REFERENCES

Ellen, S. (n.d.). *dotnetperls*. Retrieved from dotnetperls: http://www.dotnetperls.com/stopwatch

MicroSoft. (n.d.). *Microsoft* . Retrieved from Microsoft Docs: https://docs.microsoft.com/en-us/dotnet/desktop/winforms/controls/limitations-of-the-timer-component-interval-property?view=netframeworkdesktop-4.8

Neelam Yadav, S. K. (2016). SORTING ALGORITHMS. *International Research Journal of Engineering and Technology (IRJET)*, pp. 538-531.

Rouse, M. (n.d.). *whatis?, tech target*. Retrieved from what is? : https://whatis.techtarget.com/definition/sorting-algorithm

Sonal Tuteja, A. G. (n.d.). *Greeks for Greeks*. Retrieved from Greeks for Greeks: https://www.geeksforgeeks.org/analysis-of-different-sorting-techniques/

Stephans, R. (2013). *Essential ALgorithms, A practical approach to Computer Algorithms.* (R. Elliot, Ed.) United States of America: John Wiley.

Tarundeep Singh Sodhi, S. K. (2013, February). Enhanced Insertion Sort Algorithm. *International Journal of Computer Applications*.

Tripathi, P. (2017, Novemebr 30). *C# Corner*. Retrieved from C3 Corner: https://www.c-sharpcorner.com/blogs/binary-search-implementation-using-c-sharp1

Tutorial. (n.d.). *Tutorials Point*. Retrieved from https://www.tutorialspoint.com/data_structures_algorithms/sorting_algorithms.htm

APPENDIX A: ALGORITHM TO GENERATE CASE STUDY INPUT LISTS

```
using System.Threading.Tasks;

using System.Collections;

using System.IO;

namespace PopulateList

{

    class PopList

    {

        ArrayList List;

        ArrayList DesVal;

        public PopList()

        {

            List = new ArrayList();

            DesVal = new ArrayList();

        }


        public void Fixit()

        {

            Console.Write("Enter the size of the array\nMake the size larger than 16:  ");

            int size = int.Parse(Console.ReadLine());

            do

            {

                Console.WriteLine("Enter the size of the array\nMake the size larger than 16:  ");

            } while (size <= 16);


            Random values = new Random();


            for (int x = 1; x <= size; x++)

            {

                DesVal.Add(values.Next(0, int.MaxValue));
            }

                int Dup = Convert.ToInt32(DesVal.Count * (0.3));


            for (int i = 0; i < Dup; i++)

            {

                    for (int y = 0; y < Dup; y++)

                    {

                        DesVal.Add(DesVal[i]);

                    }

            }

            Shuffle(DesVal);

        }

       private void Shuffle(ArrayList point)

        {

            Random temp = new Random();

            for (int i = 0; i < point.Count; i++)

            {

                Object obj = point[i];

                int val = temp.Next(point.Count - i) + i;

                point[i] = point[val];

                point[val] = obj;

            }

        }


        public void WriteData(string FileName)

        {

            StreamWriter SW = new StreamWriter(FileName);


            for (int x = 0; x <= DesVal.Count - 1; x++)

            {

                SW.WriteLine(DesVal[x]);
            }

            SW.Close();
        }

        public void PopulateList(string FileName)
        {
            StreamReader Input = new StreamReader(FileName);
            while (!Input.EndOfStream)
            {
                string[] data = Input.ReadLine().Split(';');
                List.Add(int.Parse(data[0]));
            }
            Input.Close();
        }

        public void DisplayList()
        {
            for (int x = 0; x <= List.Count-1; x++)
            {
                Console.WriteLine(List[x]);
            }
            Console.WriteLine();
            Console.WriteLine();
            Console.WriteLine();
            newCount();
        }

        private void newCount()
        {
            int count = 0;
            for (int i = 0; i <= List.Count - 1; i++)

            {

                count = count + 1;

            }

            Console.WriteLine("New list size is: {0}", count);

        }

    }

}
```

APPENDIX B: BASIC SORTING ALGORITHM (ITERATIVE VERSION)

```
internal class InsertionSort
    {
            Stopwatch time = new Stopwatch();
            int comparison = 0;
            int swaps = 0;

    public void Insertion(ArrayList A)
            {
                for (int x = 1; x <= A.Count - 1; x++)
                    Insert((int)A[x], x - 1, A);
                display_List(A);
            }
            private void Insert(int value, int Last, ArrayList A)
            {
            time.Start();
            for (int x = Last; x >= 0; x--)
            {
                int cur = (int)A[x];

                comparison++;
                    if ((value > cur))
                    {
                        A[x + 1] = A[x]; swaps++;
                    }
                    else
                    {
                        A[x + 1] = value;
                        return;
                    }
            }
            A[0] = value;
            time.Stop();
            }

            public void display_List(ArrayList A)
            {
                for (int i = 0; i < A.Count; i++)
                {
                    Console.WriteLine(A[i]);
                }
                Console.WriteLine();
                Console.WriteLine("Number of swaps : {0}", swaps);
                Console.WriteLine();
                Console.WriteLine("Number of comparisons : {0}", comparison);
                Console.WriteLine();
                Console.WriteLine("Nmuber of Elapsed ticks : {0}", time.ElapsedTicks);
            }
        }
```

APPENDIX C: BASIC SORTING ALGORITHM (RECURSIVE VERSION)

```
    class Recursive_IS
    {
        Stopwatch time = new Stopwatch();
        int swops = 0, comparisons = 0;
            public void doInsertion(ArrayList Integers)
        {
            IS_Recursive(Integers, Integers.Count);
            display_List(Integers);
        }
        private void IS_Recursive(ArrayList Integers, int N)
        {
            time.Start();
            if (N <= 1)
                return;

            IS_Recursive(Integers, N - 1);
            int last = (int)Integers[N - 1];
            int j = N - 2;
            comparisons++;
            while (j >= 0 && (int)Integers[j] < last)
            {
                Integers[j + 1] = Integers[j];
                j--;
                swops++;
            }
            Integers[j + 1] = last;
            time.Stop();
        }


        public void display_List(ArrayList List)
        {
            for (int i = 0; i < List.Count; i++)
            {
                Console.WriteLine(List[i]);
            }
            Console.WriteLine();
            Console.WriteLine("Number of swops : {0}",swops);
            Console.WriteLine();
            Console.WriteLine("Number of comparisons : {0}", comparisons);
            Console.WriteLine();
            Console.WriteLine("Nmuber of Elapsed ticks : {0}", time.ElapsedTicks);
        }
}
```

APPENDIX E: ENHANCED INSERTION SORT

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Collections;
using System.Diagnostics;
```

Authors:	Mpendulo Bungane (216810108) // Deregistered
	Khanya Gqirana (216057361)
	Sitembiso Caleni (216277361)

```
namespace Recursive_IS
{
    class Enhanced_IS
    {
        Stopwatch time = new Stopwatch();

        public int swop= 0, comparison =0;
            public void doEIS(ArrayList Integers)
        {
            EIS(Integers);
            display_List(Integers);
        }
            private void EIS(ArrayList Integers)
        {
            time.Start();
            int i = 1, current = 0, n = Integers.Count;
            // while the entire list hasn't been processed
            while (i < n)
            {
                int j = i; // variable to compare elements within list
                comparison++;
                // compares element at i with first element in the list
                if ((int)Integers[i] >= (int)Integers[0])
                {
                    // inserting the current element i into its correct position in the sorted
portion of the list
                    while (j > 0)
                    {
                        swap(j, j - 1, Integers);
                        j--;

                    }
                    i++; //move to the next element in the list
                }
                // if the element i is less the its precceding element, no swapping is done i is in
the 'correct position'
                else if ((int)Integers[i] <= (int)Integers[i - 1])
                {
                    comparison++;
                    i++;
                    continue;
                }
                // the binary search is used to find where i should be inserted in 'partly' sorted
portion of the list
                else
                {
                    current = BinarySearch(Integers, (int)Integers[i], i);
                    if (current == 0)
                    {
                        comparison++;
                        while (j > current + 1)
                        {
                            swap(j, j - 1, Integers);
                            j--;
                        }
                    }
                    else
                    {
                        comparison++;
                        while (j > current)
                        {
                            swap(j, j - 1, Integers);
                            j--;
                        }
                    }
                    i++;
                }
            }
            time.Stop();
        }

        private int BinarySearch(ArrayList Integers, int i, int j)
        {
            int f = 0; // first
            int l = j - 1; // last
            int m; // middle
            while (f <= l)
            {
                m = (f + l) / 2;
                int cur = (int)Integers[m];
                if (m == 0)
                    return 0;
                else if (cur == i)
                    return m;
                else if (i > (int)Integers[m + 1] && i < (int)Integers[m])
                    return m + 1;
                else if (i > (int)Integers[m] && i < (int)Integers[m - 1])
                    return m;
                else if (i < cur)
                    f = m + 1;
                else
                    l = m - 1;

            }
            return -1;
        }

        private void swap(int a, int b, ArrayList Integers) // 35
        {
            int temp = (int)Integers[a];
            Integers[a] = Integers[b];
            Integers[b] = temp;
            ++swop;

        }
        public void display_List(ArrayList List)
        {
            for (int i = 0; i < List.Count; i++)
            {
                Console.WriteLine(List[i]);
            }
            Console.WriteLine();
            Console.WriteLine("Number of swops : {0}", swop);
            Console.WriteLine();
            Console.WriteLine("Number of comparisons : {0}", comparison);
            Console.WriteLine();
            Console.WriteLine("Nmuber of Elapsed ticks : {0}", time.ElapsedTicks);
        }
    }
}
```

APPENDIX F: IMPROVED INSERTION SORT

```
class InsertSortt

    {

        Stopwatch timer = new Stopwatch();

        int swops = 0, comparisons = 0;


        public void Insert(ArrayList List)

        {

            timer.Start();


            int Pos = 0;

            int val = (int)List[Pos];


            for (int x = 1; x <= List.Count - 1; x++)

            {

                if (((int)List[x]).CompareTo(val) > 0)

                {

                    Pos = x;

                    val = (int)List[Pos];

                    comparisons++;

                }

            }

            for (int x = Pos; x > 0; x--)

            {

                List[x] = List[x - 1];

            }

            List[0] = val;


            for (int x = 2; x <= List.Count - 1; x++)

            {

                int y = x;

                int value = (int)List[y];
                int prevOne = (int)List[y - 1];


                    while (prevOne.CompareTo(value) < 0)

                    {

                    comparisons++;

                    Object temp = List[y];

                    List[y] = List[y-1];

                    List[y-1] = temp;

                    swops++;


                    y = y - 1;

                        value = (int)List[y];

                        prevOne = (int)List[y - 1];

                    }


            }

            timer.Stop();

        }



        public void DisplayList(ArrayList List)

        {

            for (int i = 0; i <= List.Count - 1; i++)

            {

                Console.WriteLine(List[i]);


            }


            Console.WriteLine();

            Console.WriteLine();

            Console.WriteLine();
```

```
            Console.WriteLine("Number of comparisons : {0}", comparisons);

            Console.WriteLine();

            Console.WriteLine("Number of swops : {0}", swops);

            Console.WriteLine();

            Console.WriteLine("Number of Elapsed ticks : {0}", timer.ElapsedTicks);



        }

    }
```

APPENDIX H: QUICK SORT

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Diagnostics;
using System.Collections;

namespace Recursive_IS
{
    class Quick_Sort
    {
        Stopwatch time = new Stopwatch();
        int swops = 0, comparisons = 0;
        public void doQS(ArrayList List)
        {
            QS(List, 0, List.Count - 1);
            display_List(List);
        }
        private void QS(ArrayList Integers, int start, int end)
        {
            time.Start();
            comparisons++;
            if (start >= end)
                return;
            int piv = (int)Integers[start];
            int l = start;
            int h = end;
            while (true)
            {
                while ((int)Integers[h] <= piv)
                {
                    h--;
                    comparisons++;
                    if (h <= l)
                        break;
                }
                comparisons++;
                if (h <= l)
                {
                    swops++;
                    Integers[l] = piv;
                    break;
                }
                swops++;
                Integers[l] = Integers[h];
                l++;
                while ((int)Integers[l] > piv)
                {
                    l++;
                    comparisons++;
                    if (l >= h)
                        break;
                }
                comparisons++;
                if (l >= h)
                {
                    l = h;
                    swops++;
                    Integers[h] = piv;
                    break;
                }
                swops++;
                Integers[h] = Integers[l];
            }
            QS(Integers, start, l - 1);
            QS(Integers, l + 1, end);
            time.Stop();
        }
        public void display_List(ArrayList List)
        {
            for (int i = 0; i < List.Count; i++)
            {
                Console.WriteLine(List[i]);
            }
            Console.WriteLine();
            Console.WriteLine("Number of swops : {0}", swops);
            Console.WriteLine();
            Console.WriteLine("Number of comparisons : {0}", comparisons);
            Console.WriteLine();
            Console.WriteLine("Nmuber of Elapsed ticks : {0}", time.ElapsedTicks);
        }
        }
}
```

APPENDIX I: BUCKET SORT

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.IO;

using System.Collections;

using System.Diagnostics;
```

```
namespace Recursive_IS

{

   class bucket_Alg

   {

      Stopwatch Time = new Stopwatch();




      private int Max(ArrayList Arr)

      {

         int maxVal = -999;


         for (int x = 0; x <= Arr.Count - 1; x++)

         {

            if (maxVal < (int)Arr[x])

            {

               maxVal = (int)Arr[x];

            }

         }

         return maxVal;

      }


      public void BucketSort(int numOfBuckets, ArrayList Arr, ref int swops, ref int comparisons)

      {

         Time.Start();


         SinglyLinkedList[] buckets = new SinglyLinkedList[numOfBuckets + 1];


         for (int i = 0; i <= buckets.Length - 1; i++)
         {

            buckets[i] = new SinglyLinkedList();

         }



         double ceil = Math.Ceiling(Max(Arr) * 1.0 / numOfBuckets);



         int sameCeil = Convert.ToInt32(ceil);



         /* Allocate each of the unsorted list items into the correct bucket.

          * Consider:  i) The bucket for a value is given by the floor of the value on which the sorting is being done

          *             divided by the range of a bucket.

          *          ii) Make sure that each bucket remains sorted as you add a new value (refer to the post-condition

          *             above for clarity).  */



         for (int x = 0; x <= Arr.Count - 1; x++)

         {

            int va = (int)Arr[x];



            double index = Math.Floor((int)Arr[x] * 1.0 / sameCeil);



            int sameIndex = Convert.ToInt32(index);



            SinglyLinkedList Temp = buckets[sameIndex];    // same as Temp = new SinglyLinkedList



            Cell start = Temp.getFirst();



            bool added = false;
```

```
    while ((start != null) && (!added))

    {

        int val = (int)start.value();

        comparisons++;

        if (val > (int)Arr[x])

        {

            Temp.addBefore((int)Arr[x], start);

            added = true;

            break;

        }

        start = start.next();

    }


    if (!added)

    {

        Temp.addLast(Arr[x]);

    }


}


/* Combine each bucket in sequence, overwriting the values in the original list.

 * Consider:  i) That buckets are not all the same length.

 *          ii) That you need to keep track of where the next slot in the original list is for a value

 *              to be overwritten. */


int nrEl = Arr.Count - 1;

for (int i = 0; i <= buckets.Length - 1; i++)

{
```

```
            //Temp = buckets[i];


            //Cell cur = Temp.getFirst();


            Cell cur = buckets[i].getFirst();


            while (cur != null)

            {

                Arr[nrEl] = (int)cur.value();

                cur = cur.next();

                nrEl--;


            }


        }

        Time.Stop();


        long time = Time.ElapsedTicks;

        Display(Arr, time, ref swops, ref comparisons);


    }


    private void Display(ArrayList Arr, long Time, ref int swops, ref int comparisons)

    {

        for (int x = 0; x <= Arr.Count - 1; x++)

        {

            Console.WriteLine("{0}", (int)Arr[x]);

        }
```

```
            Console.WriteLine();

            Console.WriteLine();

            Console.WriteLine("The number of elapsed ticks are: {0}", Time);

            Console.WriteLine();

            Console.WriteLine("The number of swops are: {0}", swops);

            Console.WriteLine();

            Console.WriteLine("The number of comparisons are: {0}", comparisons);

        }

    }

}
```

APPENDIX J: DECLARATION BY WRA(V)201 PEER PROOFREADER (NOT ONE OF THE GROUP MEMBERS)

I, Ushveer Ramasir 216078733(name and student number) declare that I have proofread this report on 5/27/2020(date).

Below the proofreader must provide a summary of the recommendations made to the authors to improve the report:

- Adjust the coherency between the progression of new points.
- Add more citations from the given references.

Signature: [illegible]