

The ComparISon of Sorting AlgoRithms in multiple data sizes

Jordan Coskey



April 30, 2017

s216163110

Bcom (Computer Science & Info Systems)

# Introduction

Sorting of data is a vitality important technique used in Computer Science that significantly reduces the complexity of problems in Programs, more specifically the reasons for sorting data is that it can reduce the searching complexity of data. There are numerous algorithms that perform the sorting of data however since efficiency is key, the best sorting algorithm needs to be utilized under the current circumstance (data size). This report tests and analyses the performance of 4 specific sorting algorithms namely Insertion sort, Recursive insertion sort, Double insertion sort and Bucket sort. The testing and determination of the best sorting algorithm in its respective environment will be done with 5 different array sizes ranging from small to large all with the assumption that all data types will be of type Integer and only descending order is desired. However, the limitations of this report will be that different testing computers will vary in the CPU computational ability, resulting in the clock ticks it takes to perform the algorithm to vary.

# Related Work

Each sorting algorithm has its own technique of providing the desired outcome of which is a sorted list, as well as its own performance measured in Edmund Landau’s Big O notation along with their associated benefits and shortcomings. These attributes will be elaborated on in this related work section.

## Insertion Sort (Basic)

Find Appendix 1.1 for implemented code

This is an elementary sorting algorithm that uses the “Iterative sorting” approach. The Algorithm initially assumes the list is unsorted and subsequently splits the array into a sorted and unsorted portion. In the initial pass, the first element in the unsorted portion is placed into the sorted portion. During the consecutive passes the next element in the unsorted portion is compared to each element in the sorted portion until it is larger than the element behind it and smaller than the element in front of it, this is where it is now placed, reducing the unsorted portion until all elements lay in the sorted portion. In terms of performance the basic version of Insertion Sort has the same performance through all of its cases; worst, best and Average. This is a Performance of O(n2) (Chhatwani & Somani, 2013). Using this theoretical performance measurement the expected algorithm will be quadratic (polynomial) (Moshe J. Augenstein, 2007) as demonstrated by its Big O Notation performance. This is beneficial under the conditions where the data input is kept small (Chhatwani & Somani, 2013). This algorithm utilises an in-place sorting approach. Therefore the resources, in terms of the space it consumes during the sort, is kept to a minimal (Chhatwani & Somani, 2013). However the shortcomings are when the input data is large as the time taken to sort will be exponentially bigger as the input size increases (Answers.com, 2015).

## Insertion Sort (Recursive)

Find Appendix 1.2 for implemented code

This algorithm uses itself to recursively sort data using the same comparing and placing logic as the basic Insertion sort, more specifically it utilises head recursion. The recursive method receives the array list and a position in the list as its parameter. The method is recursively called till the position sent through reaches the terminating condition of this position being the bottom of the unsorted list. The base case is to then to compare and place this position into its correct place in the sorted portion. When observing performance the recursive version of Insertion Sort has the same performance as its basic counterpart with a performance of O(n2) with an average performance of O(n2) and its best case Performance of O(n2) (Chhatwani & Somani, 2013). The use of recursion is advantages solely for the programmer as it can be much simpler to program compared to its iterative version, the solution length can be shorter and it can be much easier to understand the implemented solution. However the disadvantages arise when taking the consumption of computer resources into consideration as each recursive call gets added to the stack, the more recursive calls the bigger the stack, the more CPU, Memory and time it consumes. This is until the stack cannot accommodate any more recursive calls and a Stack Overflow error arises. (Nakov & Kolev, 2013)

## Double Insertion Sort (Optimised)

Find Appendix 1.3 for implemented code

This algorithm uses a divide a conquer approach by sorting the list from the middle out. It first starts by selecting the middle element in the list, regarding it as sorted and the rest of the list unsorted. It then selects two elements, adjacent to either side of the sorted portion, these two elements are swapped if the selected element on the left is smaller than the selected element on the right. The two elements are placed into the sorted portion and then compared till they are positioned correctly in the sorted portion. This logic is continued until all elements lay inside the sorted portion. In terms of performance this optimised algorithm has a best case performance of O(n) with its Average performance being less than O(n1.585) and its worst performance being less than O(n1.585) as well. This performance demonstrates that as the array gets bigger the efficiency of this algorithm gets better (Sodhi, et al., 2013). The advantages of this optimised Insertion Sort Algorithm lays in its ability to reduce the time complexities its basic version faces. It becomes more efficient the bigger the list is, demonstrated in its theoretical Big O performance. The disadvantages arise when list sizes are small as compared to other algorithms, it does an unnecessary amount of passes. (Paira, et al., 2015)

## Bucket Sort (Advanced)

Find Appendix 1.4 for implemented code

Similarly to the Double Insertion Sort this algorithm uses a divide a conquer approach. By partitioning the original data set into a set number of buckets (groups), internally sorting those buckets then inserting the sorted buckets back into the original data set.

It first starts out by either; receiving a finite value for the range of numbers, needing to be stored in each bucket or having the number of buckets to be used. If the latter is the case, the original data structure needs to be traversed to find the maximum value contained within, once this numerical value is found it is divided by the given number of buckets with an incrimination of one, this computed value is the value for the range of numbers needing to be stored in each bucket. The original data structure values are committed into their respective bucket (according to which numerical value falls within the bucket range). The values in each bucket are then sorted (using any sorting method preferred). The buckets are then stored back into the original data structure. This advanced algorithm has performance that can vary depending on a few factors. The closer the number of buckets used is to the number of elements in the original data set, the faster the algorithm performs. The type of sorting method chosen to sort the buckets internally will affect the performance as well, worse or better is entirely dependant on which is chosen. Its best case, average, worse case performance is O(n), this is with its optimal sorting method utilised(buckets used correlates to the number of elements in the original data structure). The shortcomings arise when resources in terms of memory is

### Benefits & Shortcomings

This optimised Insertion Sort Algorithm advantages lay in its ability to reduce the time complexities its basic version faces. It becomes more efficient the bigger the list is, demonstrated in its theoretical Big O performance. The disadvantages come along when list sizes are small as compared to other algorithms, it does an unnecessary amount of passes. (Paira, et al., 2015)

# Experimental Design

The Experimental Environment in which the analysis and comparison of the Insertion variations (Insertion Sort, Recursive Insertion Sort, and Double Insertion Sort) as well as the advanced Bucket Sort, will consist of Integer Elements representing web pages, more specifically the number of times they are accessed represented in the data structure of an Integer Array. In order to grasp to the overall performance of each Sorting Algorithm, a variety of Array sizes will be tested ranging from; small, medium, large and very large. In order to keep test conditions as constant as possible and the results as accurate as possible, the same test computer will be used as well as no additional applications concurrently running. The Array size in each size category will be used throughout however the integers contained in these Arrays will be Pseudo-random values using C#’s “Random” Class the method “Next” in C#’s “Random” class will be used to minimize repetition numbers being generated (Skeet, 2013). These Pseudo-random values will be in the range of 0-109. The comparisons will be based upon to two methods; The Comparison of efficiency using Big O notation of performances in worst, best and average cases as well as the specific Clock Ticks using C#’s “Stopwatch” Class, and the method “Start”, “Stop” and Attribute Elapsed Ticks. Appendix 2.1 can be found for implementation of Pseudo-random values in Arrays. The former method will give a Theoretical Performance indication and the latter a Practical Performance Indication. I will collect the Tick data of each sorting algorithm, in each array size ten times and then average the results to give a more accurate indication of the practical performance in terms of clock ticks. These tick results will be automatically sorted in a text file during the experimental design. Graphing of each Sorting Algorithms performance (using the average of ten clock tick results) with each array size will be done to convert the raw data results into visually interpretable findings. Each sorting algorithm will then be compared to see which performs the best under each array size to determine which is best in each data input size situation.

# Results

Array size of 11:

It was found, demonstrated graphically in Appendix 3.1 that Insertion Sort (Basic) performed the best by having the lowest amount of Average Ticks whereas Double Insertion Sort performed the worst having the highest amount of average Ticks.

Array size of 101:

It was found, demonstrated graphically in Appendix 3.2 that Insertion Sort (Basic) performed the best again, having the lowest amount of Average Ticks whereas Recursive Insertion Sort incrementally performed the worst over Double Insertion Sort, having the highest amount of average Ticks.

Array size of 1001:

It was found, demonstrated graphically in Appendix 3.3 that Double Insertion Sort performed the best having the lowest amount of Average Ticks whereas Insertion Sort (Basic) performed the worst having the highest amount of average Ticks.

Array size of 10 001:

It was found, demonstrated graphically in Appendix 3.3 that Double Insertion Sort performed the best again, having the lowest amount of average Ticks whereas Insertion Sort (Basic) performed the worst having the highest amount of average Ticks

# Conclusion

During the testing, the timing mechanism used could fluctuate depending on the background processes that were running parallel to the programming IDE that was used for testing. I recommend using a test computer that has limited background processes such as a Raspberry PI or a bare OS such as Linux as well as running the timing mechanism over 100 arrays to grasp a more accurate tick average.

Based on the results obtained from conducting these experimental trials on various Algorithms using different array sizes, it was concluded that Insertion Sort (Basic) was most efficient for small to medium array sizes, specifically in this case Array sizes of 11 to 101. There is a steady increase of average elapsed ticks compared to the other algorithms when looking from graph to graph as the array size increases, this demonstrates that as the array size increases the efficiency of this algorithm decreases. These results were accepted in part with the Related Work section of this Experiment Report, where the efficiency of this algorithm was discussed.

Insertion Sort (Recursive) was not found to be the most efficient in any case, however, this was expected as found in the Related Work section that any recursive method uses more resources than its iterative counterpart, by its use of the Stack to accomplish its solution.

Double Insertion Sort was found to be the most efficient Algorithm for the sorting of large to very large arrays, specifically in this experimental report the array sizes of 1001 to 10 001. There is a steady decrease of average elapsed ticks compared to the other algorithms when looking from graph to graph as the array size increases, this demonstrates that as the array size decreases the efficiency of this algorithm increases. This concurs with the Related Work section when Double Insertion Sort’s benefits & shorting comings were discussed.

A recommendation based on this Experimental report can be given that Basic Insertion Sort should be used for small to medium array sizes and Double insertion Sort should be used for large to very large array sizes. All in terms of using the most efficient Algorithm for sorting arrays of type integer.

# References

Answers.com, 2015. *What are the advantages and disadvantages of insertion sort?.* [Online]   
Available at: http://www.answers.com/Q/What\_are\_the\_advantages\_and\_disadvantages\_of\_insertion\_sort  
[Accessed 10 April 2017].

Chhatwani, P. K. & Somani, J. S., 2013. Comparative Analysis & Performance of Different Sorting Algorithm in Data Structure. *International Journal of Advanced Research in Computer Science and Software Engineering,* 3(11), pp. 503-506.

Moshe J. Augenstein, A. M. T. Y. L., 2007. Data structures using c and c++. In: P. India, ed. *Data structures using c and c++.* New York: New Age international (P) Limited, p. 9.

Nakov, S. & Kolev, V., 2013. *Fundamentals of Computer Programming with C#.* Bulgaria: Faber Publishing.

Paira, S., Agarwal, A. & Alam, S. S., 2015. *Emerging Research in Computing, Information, Communication and Applications.* 1 ed. India: Springer.

Skeet, J., 2013. *C sharp in Depth.* [Online]   
Available at: http://csharpindepth.com/Articles/Chapter12/Random.aspx  
[Accessed 10 April 2017].

Sodhi, T. S., Kaur, S. & Kaur, S., 2013. Enhanced Insertion Sort Algorithm. *International Journal of Computer Applications,* 64(21), pp. 35-39.

# Appendices

Appendix 1.1

public void InsertionSort(ref int[] list)

{

for (int j = 1; j < list.Length; j++)

{

int cmpr = list[j];

int i = j - 1;

while (i >=0 && list[i] < cmpr)

{

list[i + 1] = list[i];

i = i - 1;

}

list[i + 1] = cmpr;

}

}

Appendix 1.2

public void RecursiveSort(int[] list)

{

RecurInsertionSort(list, 1);

}

private void RecurInsertionSort(int[] list, int i)

{

if (i < list.Length)

{

int temp = list[i];

doInsrtInCorctPlce(i, temp, list);

RecurInsertionSort(list, i + 1);

}

}

private void doInsrtInCorctPlce(int a, int current, int[] list)

{

if (a==0)

{

list[a] = current;

}

else

{

int prev = list[a - 1];

if (prev.CompareTo(current) < 0) //< for decending //for ascending >=

{

list[a] = prev;

doInsrtInCorctPlce(a - 1, current, list);

}

else

{

list[a] = current;

}}}

Appendix 1.3

public int[] dbleInsertionSort(int[] list)

{

int smllrItem, lrgrItem;

int listSize = list.Length;

int toMove, midle = (listSize / 2) ;

for (int fringe = 0; fringe < listSize / 2; fringe++)

{

if (list[midle - fringe - 1] < list[midle + fringe + 1])

{

smllrItem = list[midle + fringe + 1];

lrgrItem = list[midle - fringe - 1];

}

else

{

lrgrItem = list[midle + fringe + 1];

smllrItem = list[midle - fringe - 1];

}

for (toMove = midle + fringe; list[toMove] < lrgrItem; toMove--)

list[toMove + 1] = list[toMove];

list[toMove + 1] = lrgrItem;

for (toMove = midle - fringe; (list[toMove] > smllrItem); toMove++)

list[toMove - 1] = list[toMove];

list[toMove - 1] = smllrItem;

}

return list;

}

Appendix 1.4

public void BucketSort(int[] list)

{

int max = getMax(list); int numbrBckts1 = 3;

int numbrBckts = fndBcktRnge(max, numbrBckts1);

ArrayList buckets = new ArrayList();

for (int a = 1; a <= numbrBckts; a++)

{

buckets.Add(new ArrayList());

}

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

{

int Bucket = (int)list[i] / numbrBckts;

ArrayList currentBucket = (ArrayList)buckets[Bucket];

int bucketindex = 0;

bool isAdded = false;

while ((bucketindex < currentBucket.Count) && (!isAdded))

{

int val = (int)currentBucket[bucketindex];

if (val > (int)list[i])

{

addBeforeInArrayList(currentBucket, (int)list[i], (int)currentBucket[bucketindex]); //insertion sort

isAdded = true;

break;

}

bucketindex++;

}

if (!isAdded)

currentBucket.Add(list[i]);

}

int Index = list.Length - 1; //Index = 0 for ascending order

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

{

ArrayList curbucket = (ArrayList)buckets[i];

int bucketindex = 0;

while (bucketindex < curbucket.Count)

{

list[Index] = (int)curbucket[bucketindex];

bucketindex++;

Index--; //Index++ for ascending order

}

}

}

private void addBeforeInArrayList(ArrayList bucket, int value, int valuebefore)

{

int curnum;

for (int a = 0; a < bucket.Count; a++)

{

curnum = (int)bucket[a];

if (curnum == valuebefore)

{

bucket.Insert(a, value);

return;

}

}

bucket.Add(value);

}

private int getMax(int[] Numbers)

{

int maximum = int.MinValue;

foreach (int current in Numbers)

{

if (current > maximum)

maximum = current;

}return maximum; }

private int fndBcktRnge(int maximum, int numbuckets)

{

return maximum / numbuckets + 1;

}

Appendix 2.1

static int[] genRandomArray()

{

Random rnd = new Random();

int count = /\*11\*/ /\*101\*/ /\*1001\*/ 10001;

int[] Arry = new int[count];

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

{

Arry[i]= rnd.Next();

}

return Arry;}

Appendix 3.1

Appendix 3.2

Appendix 3.3

Appendix 3.4