**Introduction**

An algorithm can be described as a precise, systematic method for solving a class of problems [7]. At the heart of computer science is Data Structure and sorting is an important issue in Data Structure that generates the sequence of the list of items. Just as sorting is important in everyday life is the same reason that it is important in programming- it is much easier to locate items that are sorted than unsorted [1].

The proliferation of data particularly over the last few years, according to Forbes- 90% of all the data in the world has been generated in the last two years alone [9]. Developing sorting algorithms through improved performance and decreasing complexity has attracted a considerable amount of research and effort [10].

Numerous computations and tasks become simple by properly sorting data in advance. It is worth nothing that an estimated 25% of the total active time of computers is devoted to sorting procedures [2], this makes it an area of high importance for further study and effort to improve the performance of the many electronic devices we now rely on so heavily. Much focus was dedicated on sorting collections of data that were too large for the computers of the day to store in memory [3]. In their article Paira, Chandra and Alam state that the effectiveness of sorting algorithms is to optimise the importance of other sorting algorithms and the optimality of these algorithms is judged when calculating their time and space complexities [4].

**Factors to consider**

When deciding which algorithm to implement for the given problem, several factors must be considered. Firstly, the time complexity, this determines the amount of time that can be taken by and algorithm to run. Time complexity is usually written in form big O(n) notation. Big O notation measures how quickly a function grows or declines, and the value n represents the number of operations performed by the algorithm. The next consideration is stability, meaning that the algorithm keeps elements with equal values in the same relative order in the output as they were in the input. Some algorithms are stable by its nature ad an example of these would be; insertion sort, bubble sort or merge sort while others such as quick sort are not stable. Memory space is the final factor to consider. Recursive algorithms need more copies of sorting data to maintain memory and improve efficiency [10].

Table 1 below highlights the most suitable sorting algorithm depending on the criteria the user deems most relevant.

Table 1.

|  |  |
| --- | --- |
| **Criteria** | **Sorting Algorithm** |
| Small number of items | Insertion Sort |
| Items relatively pre-sorted | Insertion Sort |
| Concerned about worst-case scenarios | Heap Sort |
| Interested in good average-case result | Quicksort |
| Write as little code as possible | Insertion Sort |

**Time Complexity**

Simplicity is an important characteristic of a good algorithm according to Chang,SK. A clear and well documented program is simple to read and easy to explain and maintenance can be made easily long after its conception either by the author or other programmers. Achieving a high efficiency requires some cost in the use of resources that the algorithm needs to reach such efficiency [5]. Time complexity of an algorithm does not equal to the actual time required to execute some code, but the number of times that a statement executes [6].

When considering an algorithm it is essential to identify the worst, average and best case scenarios of the algorithm. At a very high level, the worst case complexity of an algorithm is the greatest number of operations necessary to solve the problem, the best-case complexity of an algorithm is the least number of operations and the average-case complexity of an algorithm is the average number of operations required to solve the problem over all possible inputs of size *n.* In order to compare the efficiencies of competing algorithms for a given problem, it is a requirement to consider the number of operations performed by each algorithm and this is done by classifying and comparing the growth rates of each algorithm’s complexity function[7].

**Sorting Techniques**

Internal and external sorting: When sorting is achieved within the main memory then it is referred to as internal sorting. Datasets that are so big they require secondary memory for the sorting process it is referred to as external sorting.

Stable and Non-Stable sorting: Stable sorting maintains the same sequence of the data with the same values. If after the sorting process the order of the data with the same values has changed then it is referred to as non-stable sorting.

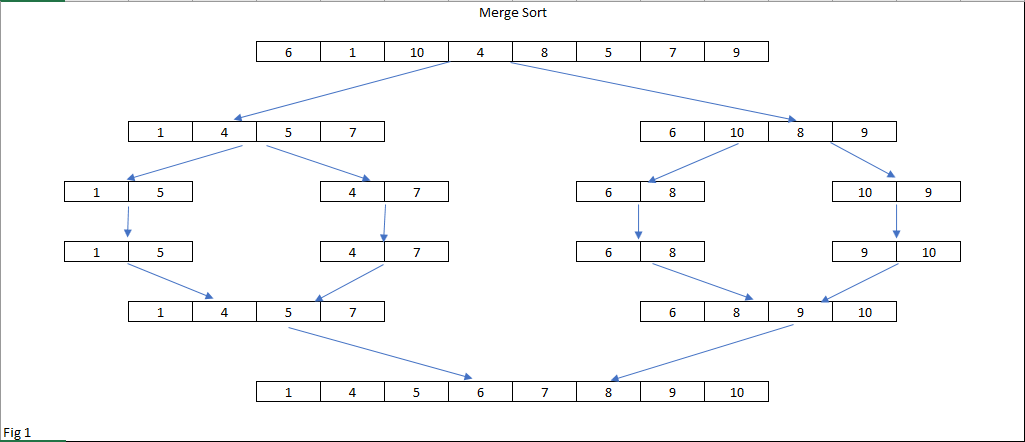
Adaptive and Non-adaptive sorting: When sorting the input, of there are elements already sorted the algorithm does not apply any sorting on them is referred to adaptive sorting. Where the algorithm does not apply this logic, it can be described as non-adaptive.

Comparison-based sorting and Distribution-based sorting: Elements are compared with other elements to find the elements correct place in the ordered list in comparison-based sorting. For distribution-based sorting, all of elements are distributed over memory space and then group the elements to get the sorted list [14].

**Report**

**Merge Sort**

The Merge Sort algorithm can be described as a stable, divide and conquer sorting algorithm and has a time complexity of O (n log n). The array is broken down into smaller, identical tasks known as sub-problems. Next, conquer each sub-problem which is achieved by reducing the problem until it reaches the best case. Finally, combine the solutions to the sub-problems [12]. Fig 1 below highlights the premise of the algorithm.



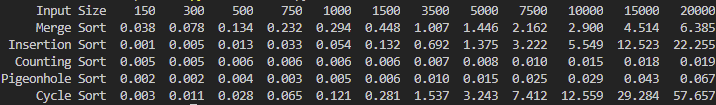
A comparison sort must have a lower bound of Ω(n log n) comparison operations, therefore merge sort is asymptotically optimal in terms of the number of comparisons it must perform for each n (log n). The time complexity for the worst, best and average case are very similar for merge sort.

As we can see below the time taken to sort the input increases roughly in line with the increase in the input size. The advantages of using merge sort include [12]:

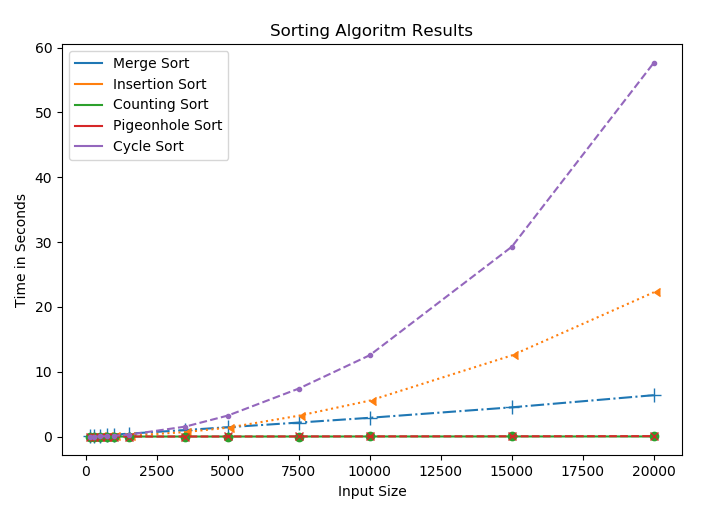
* Time complexity O(n log n)
* Uses both internal and external sorting
* Stable sorting algorithm

And the disadvantages include:

* At a minimum double the memory necessities of the further sorts as it is recursive
* High space complexity is required



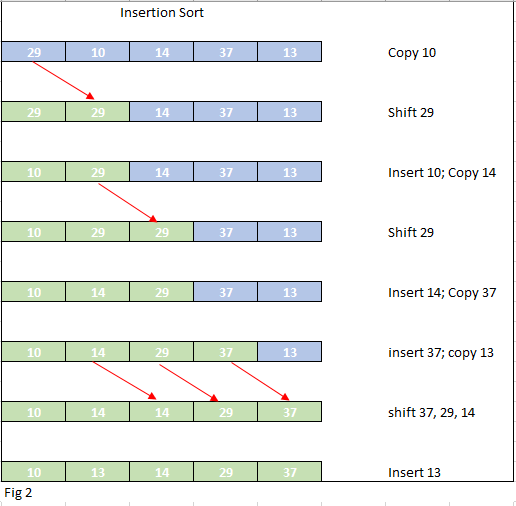


Out of the chosen sorting algorithms merge sort takes considerably longer than the quickest algorithms, but it certainly does not produce the worst run time. It is worth noting that merge sort starts off taking a lot longer than any of the other algorithms. Up to the array with 1500 elements it produces the worst time. Below is a graph highlighting the time the algorithm takes to execute for each of the input sizes. Merge sort is a Linearithmic algorithm, meaning it is asymptotically optimal in terms of the number of comparisons it must perform for each n (log) n. As stated previously divide and conquer methods are leveraged to recursively separate the problem into one or more smaller subproblems, until these become simple enough to solve. The solutions are then merged together. This method is crucial in attaining its O(n log n) time complexity and therefor proficient in scaling to large datasets [18]. 

**Insertion Sort**

Insertion sort is a comparison-based, stable sorting algorithm. It is a simple sorting algorithm and is efficient for small datasets but is inefficient when it comes to large datasets as can be seen from the graph below.

Just as its name suggests, insertion sort inserts each item into its correct place in the final list. The first iteration of the algorithm starts by comparing the 1st element with the 0th element, if the element 0 is less than element 1, then the sub array is considered sorted. If there are only 2 elements in the array, then the task is complete. If not, then the second iteration the next element is compared with the 0th and 1st element. In each iteration an element is compared with all elements and this process is repeated for all elements in the array [15]. The diagram below visualises how the algorithm sorts a given array.



As we can see from the table below, the time the algorithm takes to complete quickly start to outstrip the rate of the input increase. As highlighted earlier, the insertion performs better than all other algorithms for the smaller arrays, but as the inputs increase the slower the performance of the algorithm



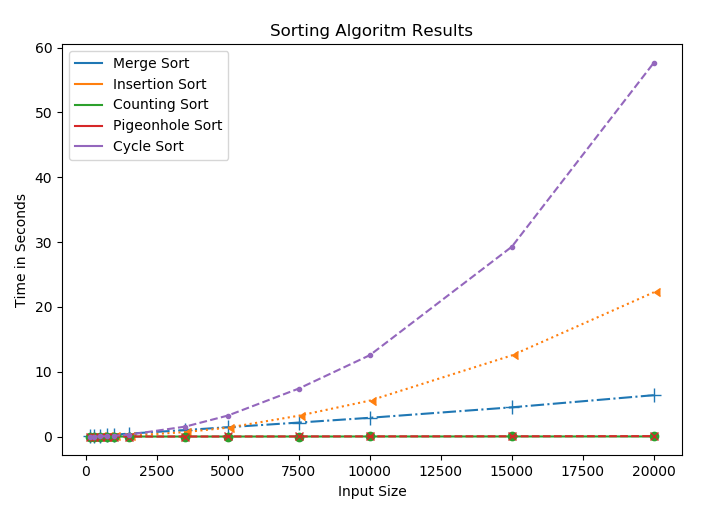
The advantages of insertion sort include:

* Exhibits good performance for small datasets
* Minimal space requirements as it is an in-place sorting algorithm
* Comparatively simple and easy to implement

The disadvantages include:

* Does not display good performance when dealing with large datasets
* The algorithm repetitively scans the list of elements each time, inserting the elements in the unordered sequence into its correct location [15].

We can see from the results below that the insertion sort is a suitable algorithm for small datasets, but quite quickly becomes less efficient. Belleza, J, states that for Linear or ‘brute force’ sorting techniques, performance is dependent on the input size [16], and this bears out in the results of the tests I completed.

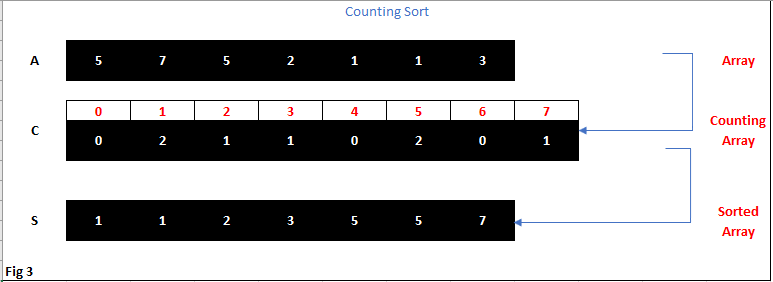


**Counting Sort**

Counting sort is a non-comparison based stable algorithm with linear running time. It utilises out of place sorting meaning the data is sorted using another data structure, therefore it requires more system memory than an in-place sorting algorithm. As counting sort is an integer-based algorithm (values are assumed to be integers) it is among the fastest sorting algorithms and this can be seen in the rest results achieved by this algorithm.

Counting sort, sorts each element over a specific range. It counts the number of occurrences of each value and calculates the number of elements less than each value. Next, it places the values in sorted order based on the count of the values. As the algorithm uses simple for loops without recursion or calls to subroutines, it is a relatively straightforward algorithm [18]. The time complexity of counting sort is O(n+k) in the best, worst and average case

Fig 3 is a visualisation of the counting sort algorithm.



Advantages of counting sort:

* Linear time. Runs in O(n) time, making it asymptotically quicker than comparison-based sorting algorithms.

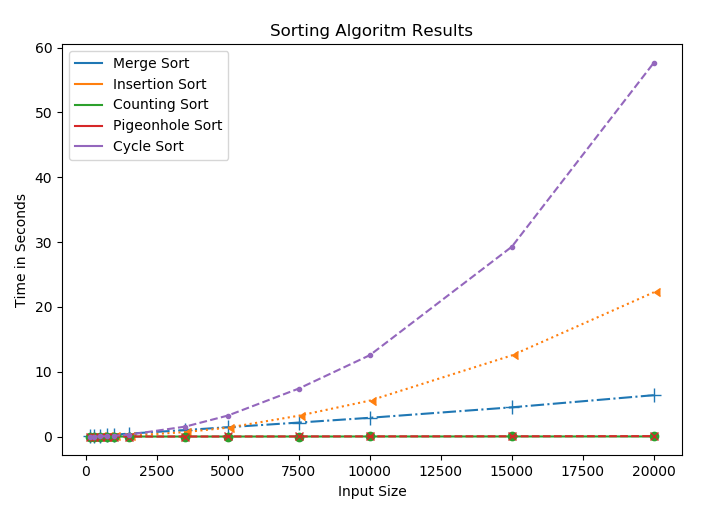
Disadvantages of counting sort:

* Only works when the range of potential items in the input is known ahead of time
* Inefficient for string data
* If the range of the input is large, counting sort will require a lot of space [16].

As evident from the table below, the running time suffers little or no impact as the input increases, this would make this an efficient algorithm for all type of integer-based datasets. It is worth noting that memory resources will be impacted by this type of sorting algorithm.



As we can see from the graph below, counting sort remains steady as the input size increases.

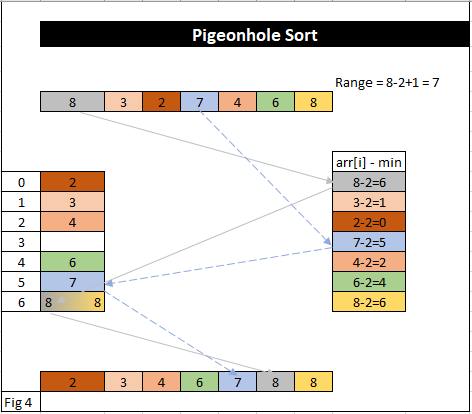


**Pigeonhole Sort**

Pigeonhole sort is a non-comparison based, stable sorting algorithm. Like the counting sort algorithm, pigeonhole sort is suitable for sorting lists of elements where the number of elements (n) and the length of possible key values (N) are roughly the same. It differs from counting sort in that it moves elements two times, firstly to the bucket array and secondly to the final destination [21].

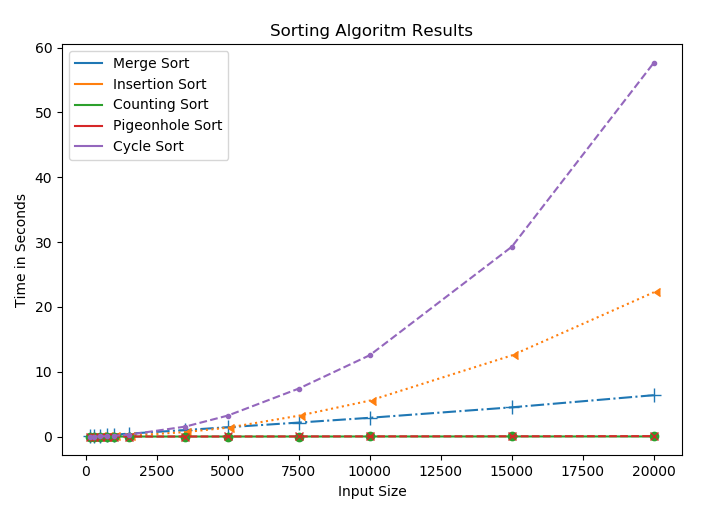
This algorithm requires O (n + Range) time where n is the number of elements in the input array and “Range” is the number of possible values in the array. It starts by finding the minimum and maximum values of the array. Allow minimum and maximum values be “min” and “max” values respectively and find range as “max-min-1”. Create an array of initially empty “pigeonholes” the same size as the range. Iterate through the array and place each element in its pigeonhole. An element arr[i] – min. Start the loop all over the pigeonhole array in order and put the elements from non-empty holes back into the original array [22].

Fig 4 highlights the method the algorithm applies to achieve the pigeonhole sort.



As we can see the max value in the range is 8 and the min is 2 adding one shows there is 7 elements in the range. Each element in the array is subtracted by the lowes value in the range to find its relevant pigeonhole, when this has completed then each element can be placed in its final position in the sorted array. The worst and average case performance of the pigeonhole sort is O (N + n).

As is evident from the graph below this algorithm reports very quick sorting time, along with counting sort it is the consistently one of the quickest algorithms I tested. For the first few tests it is quickest but is then overtaken marginally by the counting sort.

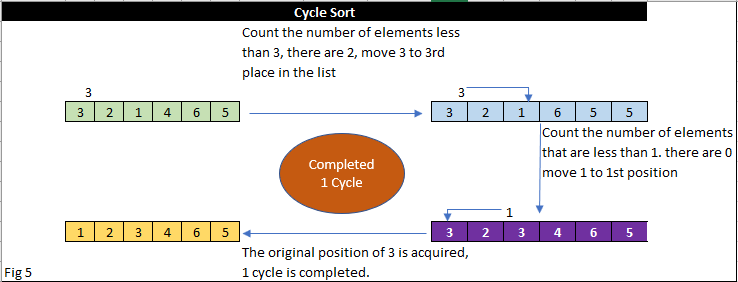


Even as the number of inputs increase the time taken to sort the elements does not increase in line with the increase in elements. The quick time performace is however at the expense of memory, as the non-comparison algorithms require storage to create temporary arrays.

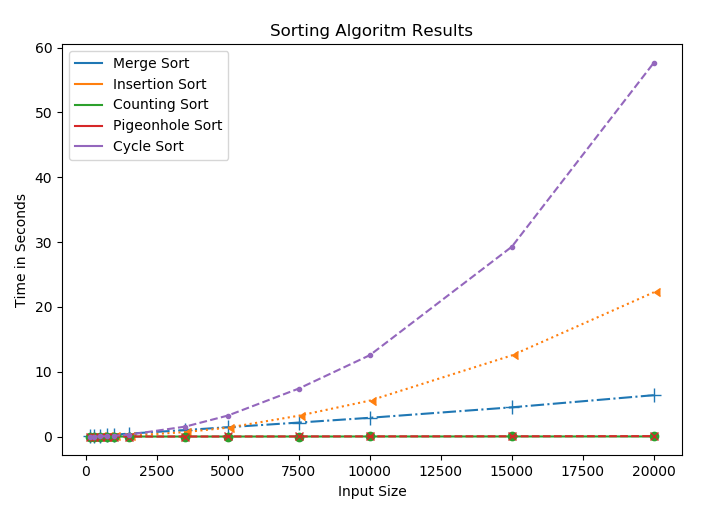
**Cycle Sort**

Cycle sort is an in-place , unstable, linear and comparison based sorting algorithm. Given an element *a,* we can locate the index at which it will occur in the sorted array by simply counting the number of elements in the entire list that are smaller than *a.* If the element is already at the correct position, do nothing. If it’s not, write it to its intended position. That position is inhabited by a different lement *b*, which we then have to move to its correct position. This process continues until an element is moved to its original position of *a*, thus completing the cycle. Repeating this process for each element sorts the list, with a single writing operation only if an element is not already in its corrct position. Computing the correct position takes O(n) time for each element, resulting in quadratic time complexity [24].

Fig 5 demostrates 1 cycle of a cycle sort.



As we can see from the below graph, the cycle sort was by some distance the least effiient of the sorting algorithms tested. On average this algorithm took 57 seconds to sort the array of 20,000 elements, in comparison to the quickest algorithm which only took, 0.019 seconds. The performance of the cycle sort was in line with all of the other sorting algorithms for the smaller datasets but at the 3,500 elements it started to become increasingly innefficient and pretty much doubled in time taken to complete for each of the susequent datasets.



Cycle sort techinques are not universally applicable, but where they can be used, it offers the advantage of a linear time complexity for little or no additional storage [23].

**Conclusion**

As evident from the research, there is no single algorithm that is suitable for all input instances.

Before deciding on an appropriate sorting algorithm to implement, the user needs to know some properties regarding the input data. For example, if the array is a random string, then MSD string sort would be an option [25]. Other factors to consider are:

**Computational Complexity:** this factor considers the performance of each algorithm in which some have worst case performance with large datasets, where other algorithms have best case performance with optimal conditions.

**Space Complexity:** here we consider the memory requirements of an algorithm, in-place sorting where this is an issue and out of place sorting where it is not an issue.

**Stability:** a stable algorithm perserves the relative order of elements with equal values [26].

Table 2 below is a good guide in helping to decide which sorting algorithm might be best for the given problem:

Table 2

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Algorithm** | **Best-Case** | **Worst-Case** | **Average-Case** | **Space Complexity** | **Stable?** |
| Merge Sort | O(n log n) | O(n log n) | O(n log n) | O(n) | Yes |
| Insertion Sort | O(n) | O(n2) | O(n2) | O(1) | Yes |
| Counting Sort | O(k + n) | O(k + n) | O(k + n) | O(k + n) | Yes |
| Pigeonhole Sort | O(n + N) | O(n+2 ^k) | O(n+2 ^k) | O(2^k) | Yes |
| Cycle Sort | O(n2) | O(n2) | O(n2) | O(n) | No |

Insertion sort is efficient when you have a small number of elements to sort or the elements in the array are nearly sorted, also, the type of element to be sorted is a factor to be considered. As much real-world data is already partially sorted this mught be a good candidate. However this algorithm is inefficent for value based data due to the amount of memory that must be shifted to make space for a new value.

When analysing sorting algorithms it is essential to understand its best-case, worst-case and average-case performance. The average-case is typically the most difficult to accurately quantify. Even when an algorithm has been shown to have a desirable average-case cost, its implemntation may be impractical. It is agreed I computer science that no algorithm that sorts by comparing elements can do better than O(n log n) performance in the average or worst case [8].

**References**

[1] http://www.ijsrd.com/articles/IJSRDV5I70074.pdf

[2] https://www.sciencedirect.com/science/article/pii/S0360131507000838?

[3] https://www.oreilly.com/library/view/algorithms-in-a/9781491912973/ch04.html

[4] Enhanced Merge Sort- a new approach to the merging process. Paira,S, Chandra,S, Alam,S. September 2016

[5] Data Structures and Algorithms. Chang,SK. 2003

[6] https://www.geeksforgeeks.org/understanding-time-complexity-simple-examples/

[7] The History of Algorithmic complexity. Nasar, Y. The Mathematics Enthusiast. Volume 13, Number 3.

[8] Table 1-Algorithms in a Nutshell. Pollice,G. Selkow,S. Heineman, G.

[9] https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/#5cd40cd360ba

[10] Review on Sorting Algorithms. A comparitive Study. Al-Kharabsheh,K. AlTurani, I. AlTurani, A. Zanoon,N. International Journal of

Science and Security (IJCSS), Vol (7): Issue (3): 2013.

file:///C:/Users/Ger/Downloads/IJCSS-877.pdf

[11] Merge Sort Diagram Fig1https://www.researchgate.net/publication/306067883\_Enhanced\_Merge\_Sort-\_A\_New\_Approach\_to\_the\_Merging\_Process

[12] https://www.thecodingdelight.com/merge-sort-algorithm-tutorial/

[13] Performance Comparison between merge and Quick Sort Algorithms in Data Structure. A, Irfan. Nawaz, H. Ameen,M. Rind, M.

IJACSA, Vol. 9 No 11, 2018

[14] Performance Comparison of Different Sorting Algorithms. Prajapati,P. Bhatt, N. Bhatt,N.

International Journal of Latest Technology in Engineering, Management & Applied Science (IJLTEMAS) Volume VI, Issue VI, June 2017

[15] Insertion Sort with its Enhancement. Chhatwani, P. IJCSMC, Vol 3, Issue 3, March 2014, pg. 801-806

Fig 2- https://topoint.in/c-programs/insertion-sort-program-c/

Insertsion Sort Code video- 6vClXM\_iHsKr1fAPwduCuAk&q=insertion+sort+python&oq=insertsion+sort+&gs\_l=psy-ab.1.1.0i13l10.982.12202..14389...0.0..0.114.714.16j1......0....1..gws-wiz.....0..35i39j0i131j0j0i10.KOTq2FvjNxs#kpvalbx=1

[16] https://www.interviewcake.com/concept/java/counting-sort

[17] Performance Analysis of Counting Sort Algorithm using various Parallel Programming Models . Babu, m. Khalid, M. Soni, S. Chowdari,S

(IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 2 (5) , 2011, 2284-2287

[18] https://en.wikipedia.org/wiki/Counting\_sort

Fig 3- https://www.researchgate.net/figure/Example-of-counting-sort\_fig4\_220686480

[19] Quadratic and Linearithmic Comparison-based Sorting Algorithms. Deringer, J.

https://johnderinger.wordpress.com/2012/12/28/quadratic-and-linearithmic-comparison-based-sorting-algorithms/

[20] http://interactivepython.org/lpomz/courselib/static/pythonds/SortSearch/TheShellSort.html

[21] https://www.revolvy.com/page/Pigeonhole-sort

[22] https://www.geeksforgeeks.org/pigeonhole-sort/

Fig 4- https://www.youtube.com/watch?v=nVQz0kZNC64

[23] Cycle-Sort: A linear Sorting Method. Haddon, B.

[24] https://en.wikipedia.org/wiki/Cycle\_sort

fig 5- https://www.javatpoint.com/cycle-sort

[25] http://www.informit.com/articles/article.aspx?p=2180073&seqNum=5

[26] https://pdfs.semanticscholar.org/d010/950f6b3c9521eb437334fa69c0b2b9353010.pdf

Table 2- https://brilliant.org/wiki/sorting-algorithms/