A Juxtaposed Study on Sorting Algorithms

Anurag Dutta [1][0000-0002-5787-3860] ,**Veenu Chhabra** [2][0000-0003-3465-9885]

and Pijush Kanti Kumar [*3]

[1] Undergraduate, Computer Science and Engineering, Government College of Engineering and Textile Technology, Serampore-712201, India
[2] Undergraduate, Information Technology, Government College of Engineering and Textile Technology, Serampore-712201, India
[3] Assistant Professor, Government College of Engineering and Textile Technology, Serampore-712201, India

pijush752000@yahoo.com
*Corresponding Author

Abstract. Data is the new fuel. With the expansion of the global technology, the increasing standards of living and with modernization, data values have caught a great height. Now a days, nearly all top MNCs feed on data. Now, to store all this data is a prime concern for all of them, which is relieved by the Data Structures, the systematic way of storing data. Now, once these data are stored and charged in secure vaults, it's time to utilize them in the most efficient way. Now, there are a lot of operations that needs to be performed on these massive chunks of data, like searching, sorting, inserting, deleting, merging and so more. In this paper, we would be comparing all the major sorting algorithms, that have prevailed till date.

Keywords: Sorting, Time Complexity, Recursion, Divide and Conquer.

1 Introduction

Sorting is one of the most important but basic operation that may be enacted on any data structure. It involves arranging the data in increasing order or decreasing order of its magnitude[1]. Sorting algorithms have got a lot to flaunt with, it's well balanced and cunning algorithm, its efficiency, and not only that, even some searching algorithms like binary search[2], interpolation search[3] [4] needs sorting algorithms to drop them in action. In this paper, we have compared all the popularly known and widely used sorting algorithms based on their dimensions of time.

2 Bubble Sort

It is the most basic sorting algorithm, erected upon the pillars of swap till doom, i.e., Swapping consecutives till we touch the core. Let us get the mathematical essence of this Sorting Algorithm.

Let we have been given a set of n unsorted elements in a list (data structure with language independency), such that

$$L(n) = \{\alpha_0, \alpha_1, \alpha_2, \dots, \alpha_{n-1}\}$$

Now, we will start by checking each set of consecutive two elements, (α_{i-1}, α_i), and if $\alpha_{i-1} > \alpha_i$, then swap both of them referentially. Once we have reached $i = n - 1$, the last element that will reach the end of the list will be the largest elements of the list, $max\big(L(n)\big)$, and intuitively, we can say that the last index is its correct place. Now, we will perform this checking consecutives for the newly forming lists for another $n - 2$ times, and each time, we will keep on decrementing the size of the subjected list.

The Pseudocode[5] for this algorithm will be

```
function bubble_sort(number_of_elements, list_of_elements)

    n ← number_of_elements

    for i = 0 to i = n - 1

       for j = 0 to j = n - 1 - i

          if list_of_elements[j] > list_of_elements[j + 1]

             swap(list_of_elements[j], list_of_elements[j + 1])

          end if

       end for

    end for

end
```

In this algorithm, the frequency count[6] of the inner loop will be

$$\sum_{j=0}^{n-1-i} (1) = \big((n-1-i) - (0) + 1\big) \times 1 = n - i$$

the frequency count of the outer loop will be

$$\sum_{i=0}^{n-1} \left(\sum_{j=0}^{n-1-i} (1) \right) = \sum_{i=0}^{n-1} (n-i) = n^2 - \left(\frac{n(n-1)}{2} \right) = \frac{n(n-1)}{2}$$

The Complexity in the dimensions of time for this Sorting Algorithm will be $O\left(\frac{n(n-1)}{2}\right) \approx O(n^2)$.

3 Selection Sort

It is another very basic sorting algorithm. In this algorithm, we simply keep on finding minimums in a selected list and work upon further.
Let we have been given a set of n unsorted elements in a list (data structure with language independency), such that

$$L(n) = \{\alpha_0, \alpha_1, \alpha_2, \dots, \alpha_{n-1}\}$$

Now we will start by finding minimum in a given list indexed between 0 and $n-1$ and will swap the element in initial index with the newly generated minimum referentially. In the next following steps, we will keep on incrementing the initial index until we have only one element left in the list.

The Pseudocode for this algorithm will be

```
function selection_sort(number_of_elements, list_of_elements)

    n ← number_of_elements

    for i = 0 to i = n - 1

       min_index ← i

       for j = i to j = n - 1

          if list_of_elements[min_index] > list_of_elements[j]

             min_index ← j

          end if

       end for

    swap(list_of_elements[min_index], list_of_elements[i])

    end for

end
```

In this algorithm, the frequency count of the inner loop will be

$$\sum_{j=i}^{n-1}(1) = \big((n-1)-(i)+1\big) \times 1 = n - i$$

the frequency count of the outer loop will be

$$\sum_{i=0}^{n-1}\left(\sum_{j=i}^{n-1}(1)\right) = \sum_{i=0}^{n-1}(n-i) = n^2 - \left(\frac{n(n-1)}{2}\right) = \frac{n(n-1)}{2}$$

The Complexity in the dimensions of time for this Sorting Algorithm will be $O\left(\frac{n(n-1)}{2}\right) \approx O(n^2)$.

4 Insertion Sort

This Sorting algorithm is quite similar in terms of it's attire with the Selection Sort, but the face of this algorithm is holding a great difference concealed within it.
Let we have been given a set of n unsorted elements in a list (data structure with language independency), such that

$$L(n) = \{\alpha_0, \alpha_1, \alpha_2, \dots, \alpha_{n-1}\}$$

In this Sorting technique, we select a reference indexed element (say α_i) and start comparing that with the elements preceding it till we find the appropriate position for the reference indexed element. The affirmative comparison can be defined mathematically as $\alpha_j > \alpha_i$, $\forall j < i$ i.e., we keep on going back in index until we find an element to be smaller than the reference indexed elements, α_i. Meanwhile, in this process, we store the reference indexed element in secure vault and will shift each element to the right which follows the affirmation condition. The main merit of this algorithm is that the elements in left indices of the referential index are properly sorted.

The Pseudocode for this algorithm will be

```
function insertion_sort(number_of_elements, list_of_elements)

    n ← number_of_elements

    for i = 1 to i = n

       reference ← list_of_elements[i]

       j ← i - 1

       while list_of_elements[j] > reference and j ≥ 0
```

```
            list_of_elements[j + 1] = list_of_elements[j]

            decrement j

        end while

        list_of_elements[j + 1] = reference

    end for

end
```

In this algorithm, the frequency count of the inner loop will be i
the frequency count of the outer loop will be

$$\sum_{i=1}^{n-1}(i) = \frac{n(n-1)}{2}$$

The Complexity in the dimensions of time for this Sorting Algorithm will be $O\left(\frac{n(n-1)}{2}\right) \approx O(n^2)$.

5 Merge Sort

Merge sort is a very good sorting technique as it follows the divide and conquer[7] algorithm.
Let we have been given a set of n unsorted elements in a list (data structure with language independency), such that

$$L(n) = \{\alpha_0, \alpha_1, \alpha_2, \dots, \alpha_{n-1}\}$$

Under this algorithm the list is divided into equally sized sub parts and merged step by step in a recursive[8] manner to bring it to sorted format. It is often referred to as the best sorting technique when we are required to sort a linked list.

The Pseudocode for this algorithm will be

```
function merge(list_of_elements, low_index, mid_index, high_index)

    size_vault1 ← mid_index – low_index + 1

    size_vault2 ← high_index - (mid_index + 1) + 1

    vault1[size_vault1]
```

```
    vault2[size_vault2]
    for i = 0 to i = size_vault1 - 1
        vault1[i] = list_of_elements[low_index + i]
    end for
    for i = 0 to i = size_vault2 - 1
        vault2[i] = list_of_elements[mid_index + 1 + i];
    end for
    i, j ← 0, 0
    k ← low_index
    while i < size_vault1 and j < size_vault2
        if vault1[i] > vault2[j]
            list_of_elements[k] = vault2[j]
            increment j, k
        else
            list_of_elements[k] = vault1[i];
            increment i, k
        end if
    end while
    while j < size_vault2
        list_of_elements[k] = vault2[j];
        increment j, k
    end while
    while i < size_vault1
```

```
        list_of_elements[k] = vault1[i];
        increment i, k
    end while
end
function merge_sort(list_of_elements, low_index, high_index)
    if low_index < high_index
       mid_index ← low_index + (high_index – low_index) / 2
       merge_sort(list_of_elements, low_index, mid_index)
       merge_sort(list_of_elements, mid_index + 1, high_index)
       merge (list_of_elements, low_index, mid_index, high_in-
dex)
    end if
end
```

In this algorithm, we will try to find the recurrence relation, that is

$$\psi(n) = 2\psi\left(\frac{n}{2}\right) + \theta(n)$$

This can be solved using Master's Theorem of Divide and Conquer

$$\psi(n) = n\,log_2\,n$$

The Complexity in the dimensions of time for this Sorting Algorithm will be $O(n\,log_2\,n)$.

6 Conclusion

We have studied BISM (Bubble, Insertion, Selection, Merge) Sorting Algorithms in our paper, and we can conclude that Merge Sort will be the fastest in all aspects. Further, we can conclude that IS (Insertion, Selection) sorts took nearly the same time. We plot the time taken by each of them in its worst run, i.e., the data elements were arranged in descending order and we were to reorder them in increasing order of it's magnitude.

We also plotted bar chart and pie of pie chart taking the time taken in terms of $10^{-5} seconds$ by them as a whole of 100%. and the result was quite promising.

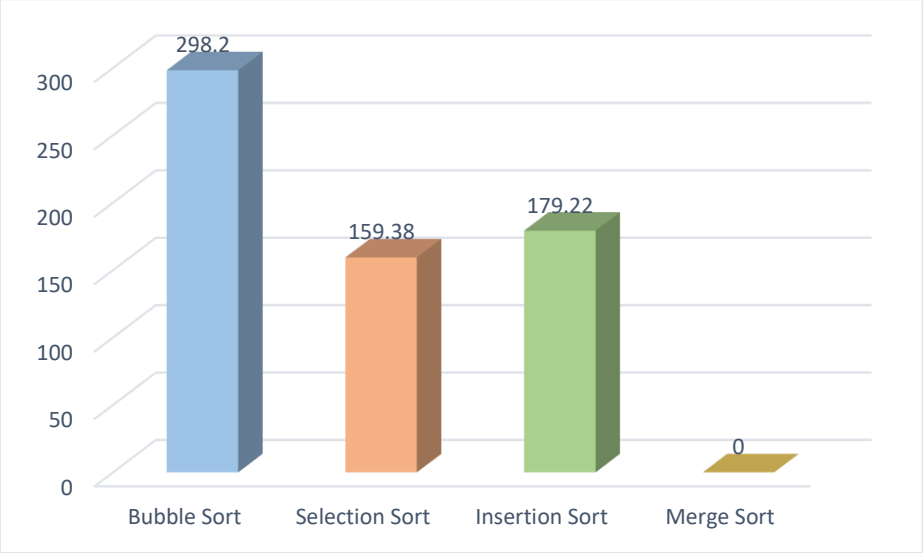

Fig. 1.1. For a Data set of 1000 entries

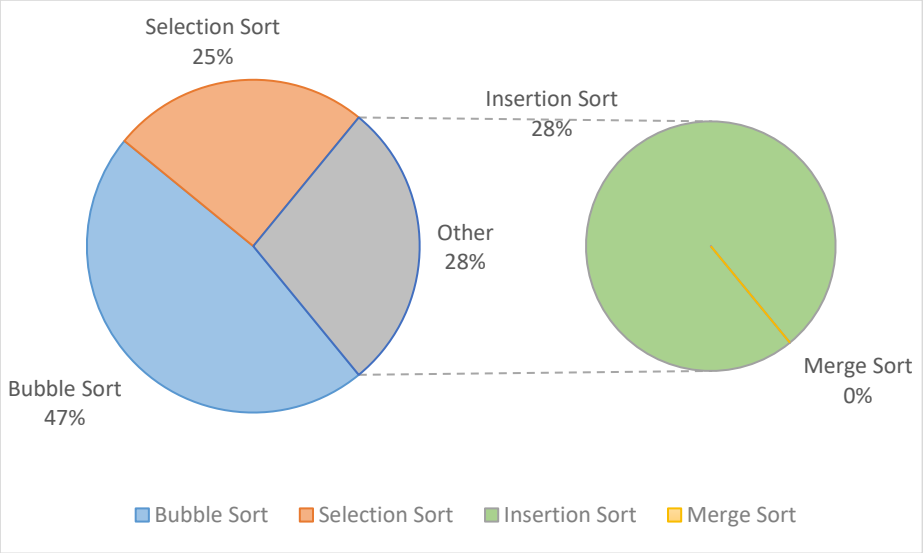

Fig. 2.2. For a Data set of 1000 entries

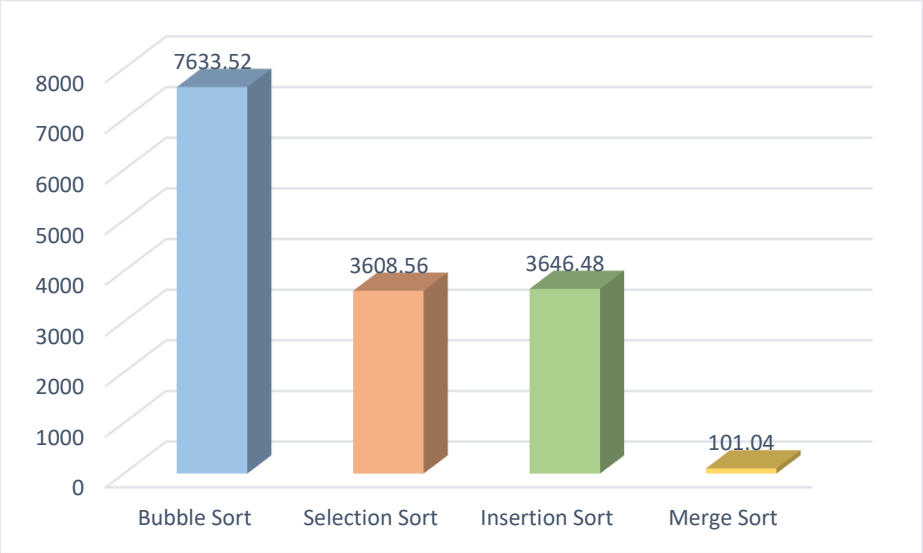

Fig. 2.1. For a Data set of 5000 entries

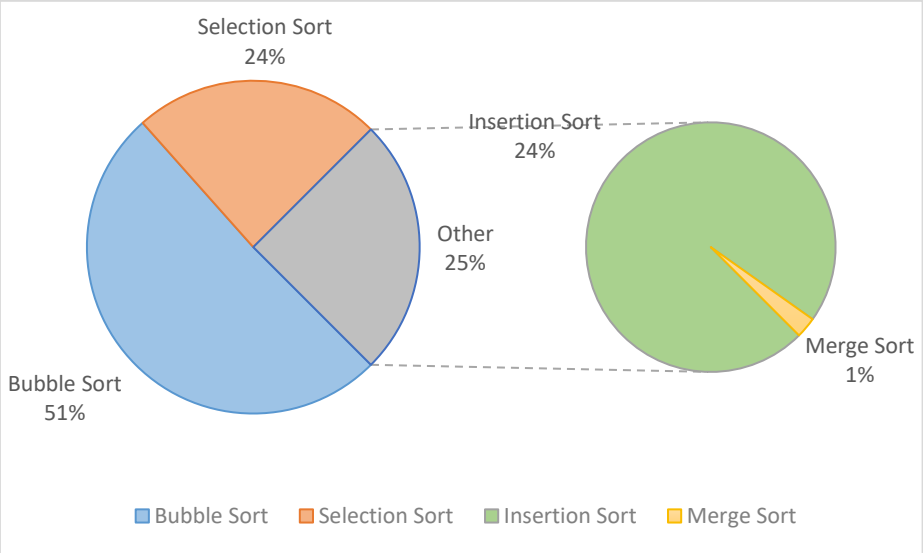

Fig. 3.2. For a Data set of 5000 entries

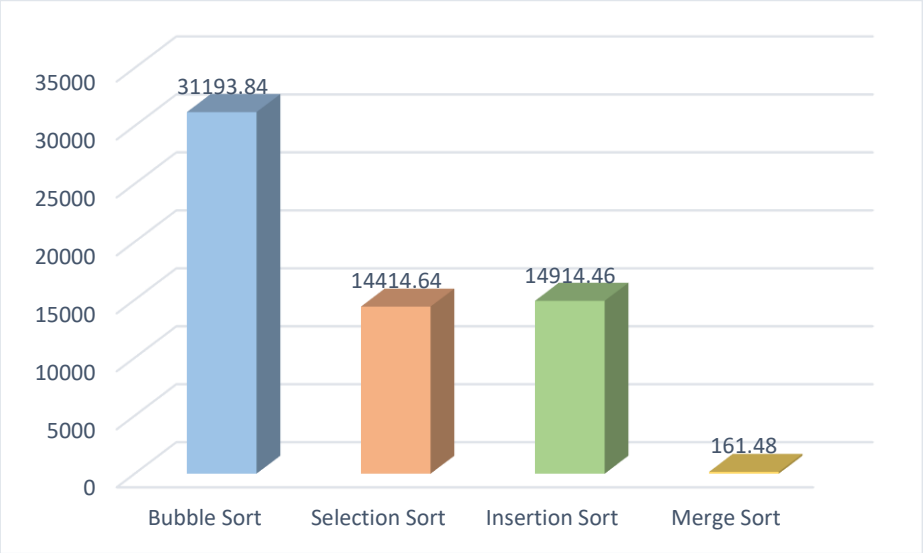

Fig. 3.1. For a Data set of 10000 entries

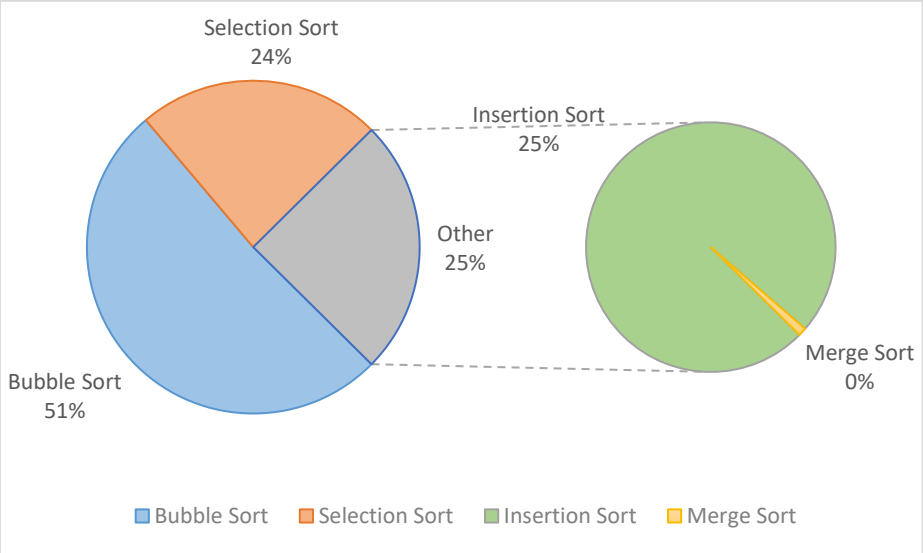

Fig. 4.2. For a Data set of 10000 entries

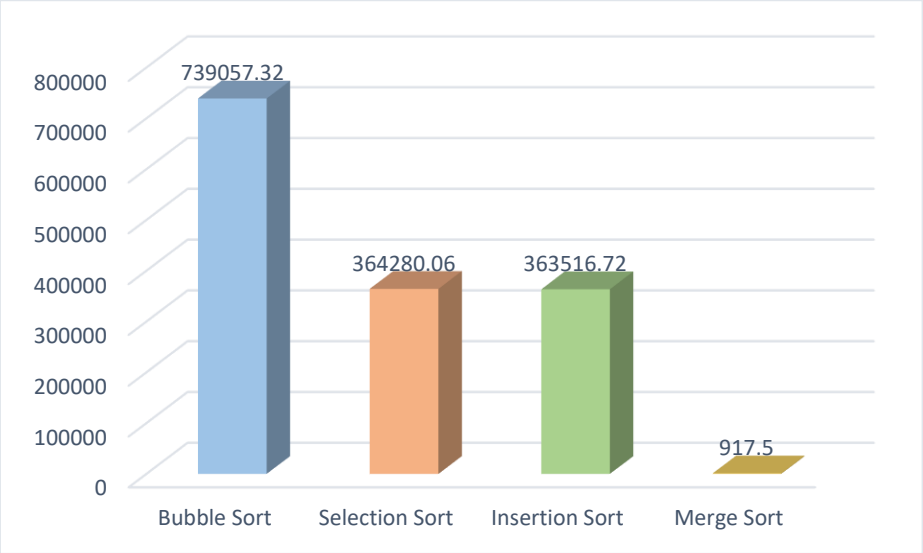

Fig. 4.1. For a Data set of 50000 entries

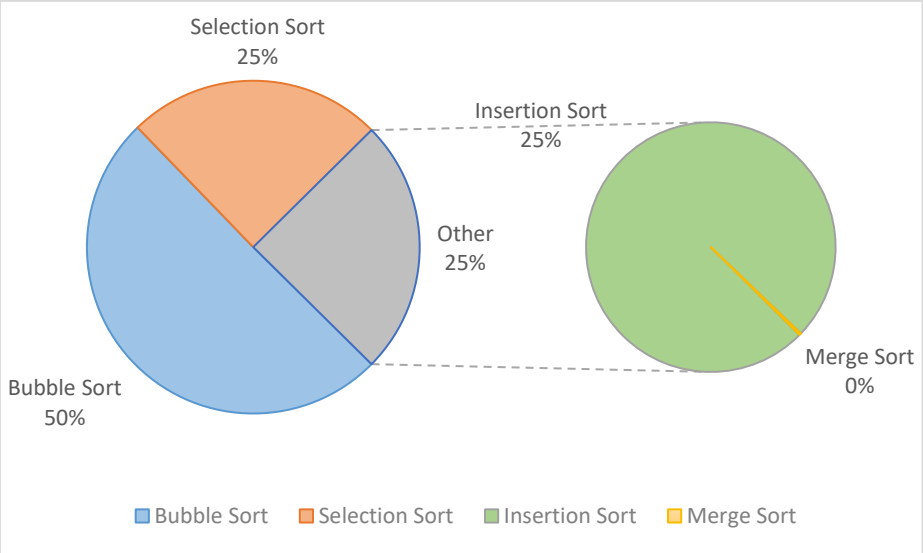

Fig. 5.2. For a Data set of 50000 entries

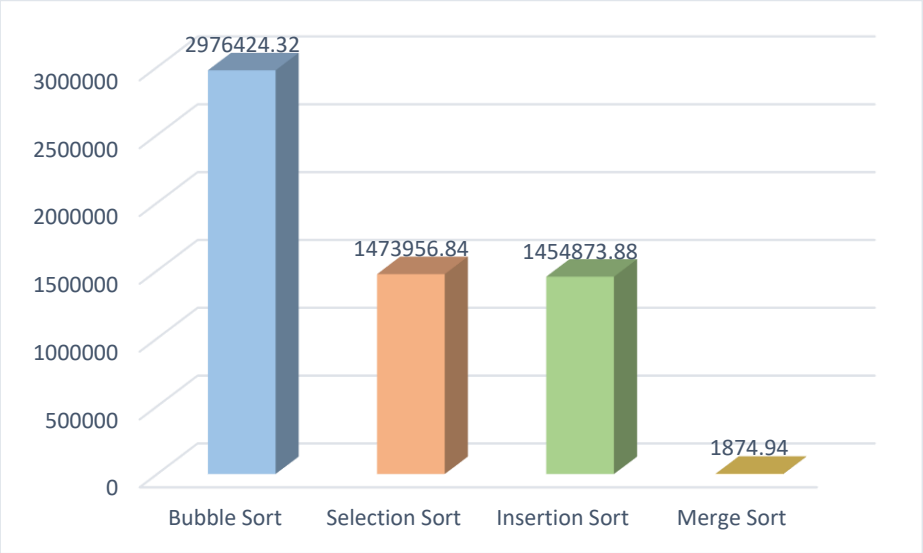

Fig. 5.1. For a Data set of 100000 entries

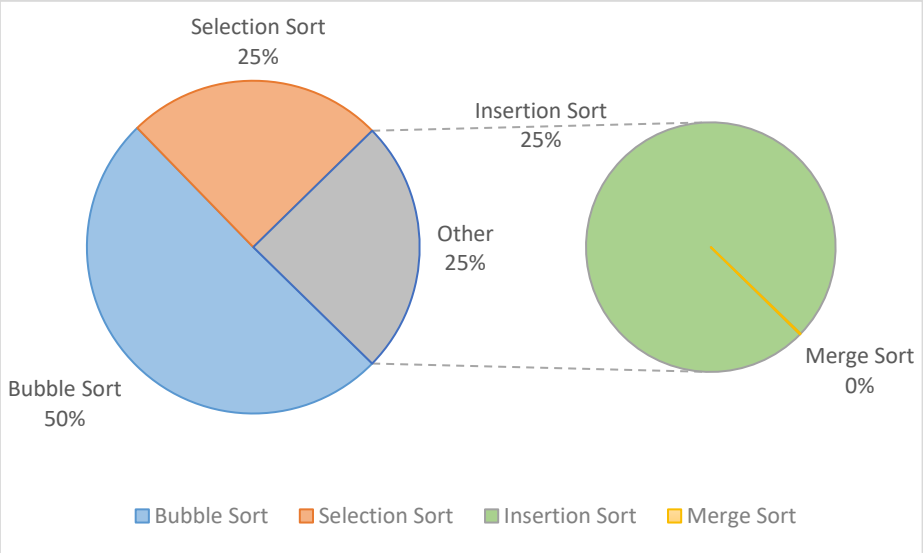

Fig. 6.2. For a Data set of 100000 entries

7 References

1. Thomas H. Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein, “Introduction To Algorithms”, MIT Press, Cambridge, Massachusetts London, England, 2001.
2. Donald E. Knuth, “The Art of Computer Programming”, Addison-Wesley Professional, 1968.

3. Gonnet G., Rogers L. and George J. (1980), “An algorithmic and complexity analysis of interpolation search”, Acta Informatica, Springer, 13, pp 39 – 52.
4. Reingold E. and Perl Y. (1977), “Understanding the complexity of interpolation search”, Information Processing Letters, Vol. 6, Issue 6, pp 219 – 222.
5. Mark Allen Weiss, “Data Structures and Algorithm Analysis in C++”, Pearson Education India, 1991
6. David Harel, “Algorithmics: The Spirit of Computing”, Springer, 1987.
7. Jon Kleinberg, Éva Tardos, “Algorithm Design”, Pearson, 2005
8. Herbert Wilf, “Algorithms and Complexity”, Tailor and Francis, 2002.