Empirical and Theoretical Analysis of Sort Algorithms

Algorithms and Order of growth functions:-
For all the sorting algorithms we consider:-
Best case input: - The best case is when the input array is already sorted. For example input instance is [1, 2, 3, 4, 5].
Average case input: - The average case is when the input array contains a set of numbers that are randomly generated. For example input instance is [3, 1, 5, 4, 2].
Worst case input: - The worst case is when the input array contains a set of numbers that are sorted in reverse order. For example input instance is [5, 4, 3, 2, 1].

Insertion Sort:-

Algorithm for insertion sort is mentioned below:-

Input : Array of n elements such that A[1..n].
Output: Array of n elements A[1..n] sorted in increasing order.

```
INSERSTION_SORT (A)
1.  for i <- 2 to length[A]
2.      key<-A[i]
    // Insert element A[i] into the sorted sub-array A[1..i-1]
3.      j <- i – 1
4.      while j >0 and A[j] > key
5.          A[j+1] <- A[j]
6.          j <- j – 1
7.      A[j+1] <- key
```

To calculate the running time of this algorithm we will have to add up the product of number of times each statement is run and cost of each statement. Considering the cost of a statement to be Ci and say it runs n times then it will contribute to Ci * n to the total time T (n)

For BEST CASE:

Let us calculate the time taken for execution of each step. Let us assume **C1..C7 are constants. Statement Number refers to the line number in the above mentioned algorithm.**

Statement Number	Cost of statement	No. of times executed	Remarks
1	C1	n	
2	C2	n-1	
3	C3	n-1	
4	C4	n-1	

5	C5	0	since A[j] is never>key
6	C6	0	since A[j] is never>key
7	C7	n-1	

T (n) = C1n + C2 (n-1) + C3 (n-1) + C4 (n-1) +C7 (n-1)

= (C1 + C2 + C3 + C4 + C7) n – (C1 + C2 + C3 + C4 + C7)

This can be represented as a linear equation of n as **an + b** where a and b are constants. When the size of n increases the lower order terms and the leading term's constant coefficients like a and b are become relatively insignificant and do not affect the rate of growth for larger values of n and hence can be ignored. That is the **best case running time of insertion sort is O(n).**

For AVERAGE CASE:

Let us calculate the time taken for execution of each step. Let us assume **C1..C7 are constants. Statement Number refers to the line number in the above mentioned algorithm.**

Statement Number	Cost of statement	No. of times executed	Remarks
1	C1	n	
2	C2	n-1	
3	C3	n-1	
4	C4	$\sum^{n}_{i=2} t_i$	Number of times while loop is run for value of i =t_i
5	C5	$\sum^{n}_{i=2} t_i$-1	
6	C6	$\sum^{n}_{i=2} t_i$-1	
7	C7	n-1	

For average case we consider that we that half of the elements in A[1..i-1] are less than A[i] and so t_i =i/2. The total time

$T(n) = C1n + C2(n-1) + C3(n-1) + C4\sum^{n}_{i=2} i/2 + C5\sum^{n}_{i=2}(i/2 -1) + C6\sum^{n}_{i=2}(i/2 -1) + C7(n-1)$

Solving the summations we get

$T(n) = n^2(C4/4 + C5/4 +C6/4) + n(C1 + C2 + C3 + C4/4 -3C5/4 -3C6/4 + C7) – (C2 + C3 + C4 + C7)$

This can be represented as a quadratic equation of n as **an^2+ bn + c** where a, b and c are constants. When the size of n increases the lower order terms and the leading term's constant coefficients like a, b and c are become relatively insignificant and do not affect the rate of growth for larger values of n and hence can be ignored. That is the **average case running time of insertion sort is O(n^2)**.

For WORST CASE:

Let us calculate the time taken for execution of each step. Let us assume **C1..C7 are constants. Statement Number refers to the line number in the above mentioned algorithm.**

Statement Number	Cost of statement	No. of times executed	Remarks

1	C1	n	
2	C2	n-1	
3	C3	n-1	
4	C4	$\sum_{i=2}^{n} t_i$	Number of times while loop is run for value of i =t_i
5	C5	$\sum_{i=2}^{n} t_i$-1	
6	C6	$\sum_{i=2}^{n} t_i$-1	
7	C7	n-1	

For worst case we consider that we that all of the elements in A[1..i-1] are greater than A[i] and so t_i =i . The total time

$T(n) = C1n + C2(n - 1) + C3(n - 1) + C4\sum_{i=2}^{n} i + C5\sum_{i=2}^{n} (i - 1) + C6\sum_{i=2}^{n} (i - 1) + C7(n - 1)$

Solving the summations we get

$T(n) = n^2(C4/2 + C5/2 + C6/2) + n(C1 + C2 + C3 + C4/2 - C5/2 - C6/2 + C7) – (C2 + C3 + C4 + C7)$

This can be represented as a quadratic equation of n as **$an^2 + bn + c$** where a, b and c are constants. When the size of n increases the lower order terms and the leading term's constant coefficients like a, b and c are become relatively insignificant and do not affect the rate of growth for larger values of n and hence can be ignored. That is the **worst case running time of insertion sort is $O(n^2)$**.

Selection Sort:-

Algorithm for selection sort is mentioned below:-

Input : Array of n elements such that A[1..n].
Output: Array of n elements A[1..n] sorted in increasing order.

```
SELECTION_SORT (A)
1.  for i <- 1 to length[A]
2.      minIndex <- i
3.      for j <- i + 1 to length[A]
4.          if (A[j] < A[minIndex] )
5.          then minIndex <- j
6.      if minIndex != j then interchange A[i] and A[minIndex]
```

To calculate the running time of this algorithm we will have to add up the product of number of times each statement is run and cost of each statement. Considering the cost of a statement to be Ci and say it runs n times then it will contribute to Ci * n to the total time T(n).

For BEST CASE:

Let us calculate the time taken for execution of each step. Let us assume **C1..C7 are constants. Statement Number refers to the line number in the above mentioned algorithm.**

Statement Number	Cost of statement	No. of times executed	Remarks
1	C1	n+1	
2	C2	n	
3	C3	n(n-1)/2	
4	C4	n(n-1)/2	
5	C5	0	since A[j] is never < A[minIndex]
6	C6	0	since A[j] is never < A[minIndex]

$T(n) = n^2(C3/2 + C4/2) + n(C1 + C2 - C3/2 - C4/2) + C1$

This can be represented as a quadratic equation of n as **$an^2 + bn + c$** where a, b and c are constants. When the size of n increases the lower order terms and the leading term's constant coefficients like a, b and c are become relatively insignificant and do not affect the rate of growth for larger values of n and hence can be ignored. That is the **best case running time of selection sort is $O(n^2)$.**

For AVERAGE CASE:

Let us calculate the time taken for execution of each step. Let us assume **C1..C7 are constants. Statement Number refers to the line number in the above mentioned algorithm.**

Statement Number	Cost of statement	No. of times executed
1	C1	n+1
2	C2	n
3	C3	n(n-1)/2
4	C4	n(n-1)/2
5	C5	n(n-1)/4
6	C6	n/2

$T(n) = n^2(C3/2 + C4/2 + C5/4) + n(C1 + C2 - C3/2 - C4/2 - C5/4 + C6/2) + C1$

This can be represented as a quadratic equation of n as **$an^2 + bn + c$** where a, b and c are constants. When the size of n increases the lower order terms and the leading term's constant coefficients like a, b and c are become relatively insignificant and do not affect the rate of growth for larger values of n and hence can be ignored. That is the **average case running time of selection sort is $O(n^2)$**.

For WORST CASE:

Let us calculate the time taken for execution of each step. Let us **assume C1..C7 are constants. Statement Number refers to the line number in the above mentioned algorithm.**

Statement Number	Cost of statement	No. of times executed
1	C1	n+1
2	C2	n
3	C3	n(n-1)/2
4	C4	n(n-1)/2
5	C5	n(n-1)/2
6	C6	n/2

$T(n) = n^2(C3/2 + C4/2 + C5/2) + n(C1 + C2 - C3/2 - C4/2 - C5/2 + C6/2) + C1$

This can be represented as a quadratic equation of n as **$an^2 + bn + c$** where a, b and c are constants. When the size of n increases the lower order terms and the leading term's constant coefficients like a, b and c are become relatively insignificant and do not affect the rate of growth for larger values of n and hence can be ignored. That is the **worst case running time of selection sort is $O(n^2)$**.

Merge Sort:-

Merge sort uses divide and conquer approach to sort given list of numbers. The algorithm for merge sort is mentioned below:-

Input : Array of n elements such that A[1..n].
Output: Array of n elements A[1..n] sorted in increasing order.

```
MERGE_SORT (A, startIndex, endIndex)
1.  if startIndex < endIndex
2.      midIndex <- (startIndex + endIndex)/2
3.      MERGE_SORT(A, startIndex, midIndex)
4.      MERGE_SORT(A, midIndex, endIndex)
5.      MERGE (A, startIndex, midIndex, endIndex)

MERGE (A, startIndex, midIndex, endIndex)
1.  n1 <- midIndex - startIndex + 1
2.  n2 <- endIndex - midIndex
3.  for i <- 1 to n1
4.      L[i] =A[startIndex + i – 1]
5.  for j <- 1 to n2
6.      R[j] =A[midIndex + j]
7.  i <- 1
8.  j <- 1
9.  for k <- startIndex to endIndex
10.     if L[i] <= R[j]
11.         A[k] <- L[i]
12.         i++
13.     else
14.         A[k] <- R[j]
15.         j++
```

Let D(n) be the time taken to divide the array at the middle. Therefore **D(n) = O(1)**.
Since we recursively divide the problem into 2 sub problems the size of the array becomes n/2 and hence the running time is 2T(n/2).
Calculating the time required for Merge :-
Statements 1,2,7 and 8 take constant time which are lower order terms and can be ignored. Calculating the time of loops we get
$T(n) = C3(n1+1) + C4n1 + C5(n2+1) + C6n2 + C9(n+1) + 3n$ Since n1+n2 =n

T(n) is approximately equal to n+3n i.e. 4n
Where C3 is cost of statement 3, C4 is cost of statement 4 and so on
Thus T(n) for the **merge algorithm is O(n).**
Since each level of merge tree computes to time cn and number of levels of the tree (lg n + 1) the time T(n) for merge sort is cn(lg n + 1) which after ignoring the lower order terms comes to **O(n log(n))**
Since this algorithm is independent of input the order of growth function is the same for all the cases. **That is the Best case, Average case and Worst case running time of Merge sort is O(n log(n)).**

Bubble Sort :-

Bubble sort compares each pair of adjacent elements, if they are not in order they are swapped. This is continued until the list is sorted. This way after every iteration the largest element bubbles up to its final position eventually sorting the list.

Input : Array of n elements such that A[1..n].
Output: Array of n elements A[1..n] sorted in increasing order.

```
BUBBLE_SORT (A)
    1.  for i <- 1 to length[A]
    2.      for j <- 1 to length[A]-i
    3.          if A[j] > A[j+1]
    4.              Interchange A[j] and A[j+1]
```

To calculate the running time of this algorithm we will have to add up the product of number of times each statement is run and cost of each statement. Considering the cost of a statement to be Ci and say it runs n times then it will contribute to Ci * n to the total time T(n).

For BEST CASE:

Let us calculate the time taken for execution of each step. Let us assume **C1..C7 are constants. Statement Number refers to the line number in the above mentioned algorithm.**

Statement Number	Cost of statement	No. of times executed
1	C1	n+1
2	C2	n
3	C3	n(n-1)/2
4	C4	0

$T(n) = n^2(C3/2) + n(C1 + C2 - C3/2) + C1$

This can be represented as a quadratic equation of n as $\mathbf{an^2 + bn + c}$ where a, b and c are constants. When the size of n increases the lower order terms and the leading term's constant coefficients like a, b and c are become relatively insignificant and do not affect the rate of growth for larger values of n and hence can be ignored. That is the **best case running time of bubble sort is $O(n^2)$.**

For AVERAGE CASE:

Let us calculate the time taken for execution of each step. Let us assume **C1..C7 are constants. Statement Number refers to the line number in the above mentioned algorithm.**

Statement Number	Cost of statement	No. of times executed
1	C1	n+1
2	C2	n
3	C3	n(n-1)/2
4	C4	n(n-1)/4

$T(n) = n^2(C3/2 + C4/4) + n(C1 + C2 - C3/2 - C4/4) + C1$

This can be represented as a quadratic equation of n as **$an^2 + bn + c$** where a, b and c are constants. When the size of n increases the lower order terms and the leading term's constant coefficients like a, b and c are become relatively insignificant and do not affect the rate of growth for larger values of n and hence can be ignored. That is the **average case running time of bubble sort is $O(n^2)$**.

For WORST CASE:

Let us calculate the time taken for execution of each step. Let us assume **C1..C7** are constants. **Statement Number refers to the line number in the above mentioned algorithm.**

Statement Number	Cost of statement	No. of times executed
1	C1	n+1
2	C2	n
3	C3	n(n-1)/2
4	C4	n(n-1)/2

$T(n) = n^2(C3/2 + C4/2) + n(C1 + C2 - C3/2 - C4/2) + C1$

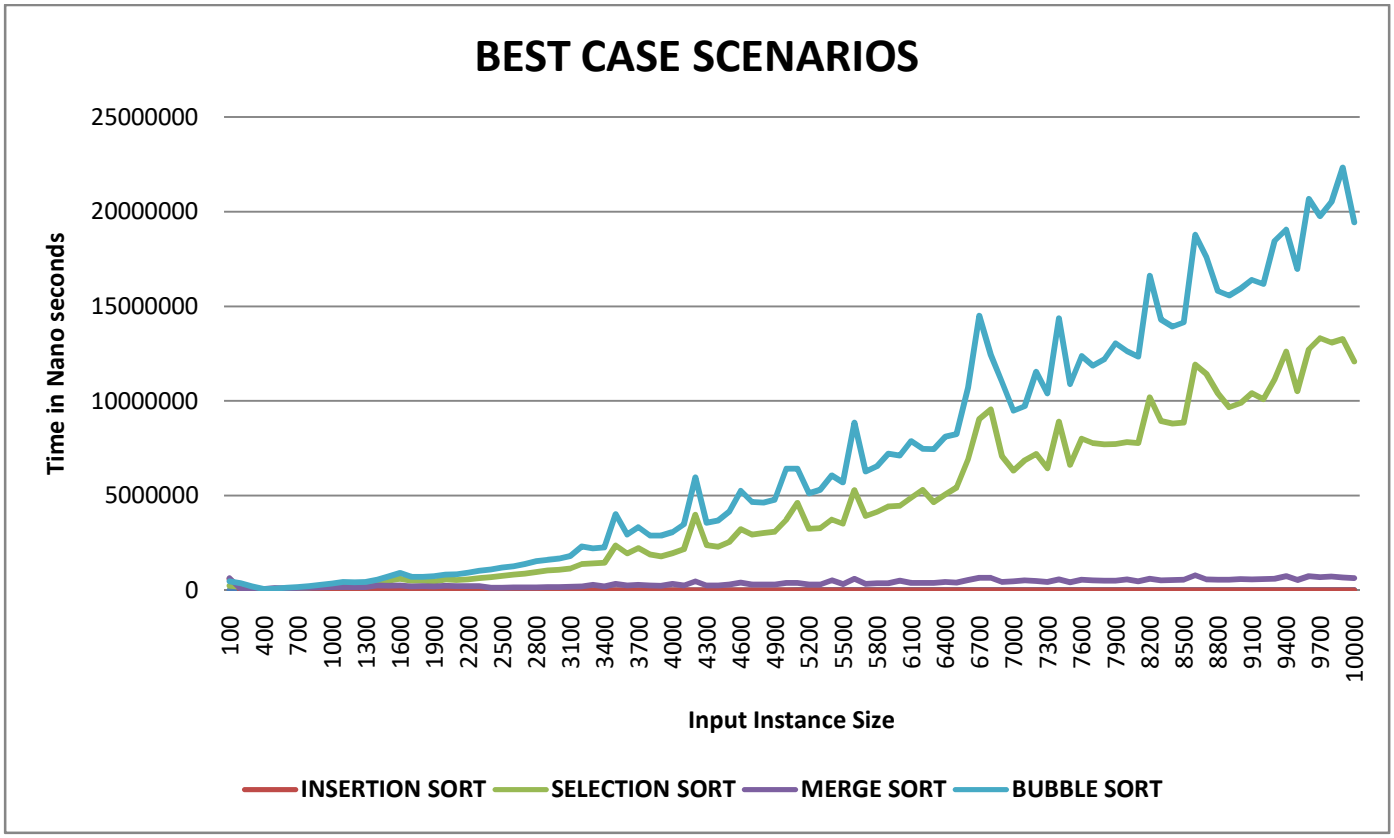
This can be represented as a quadratic equation of n as **$an^2 + bn + c$** where a, b and c are constants. When the size of n increases the lower order terms and the leading term's constant coefficients like a, b and c are become relatively insignificant and do not affect the rate of growth for larger values of n and hence can be ignored. That is the **worst case running time of bubble sort is $O(n^2)$**.

List of Order of Growth Function for each Sort Algorithm :-

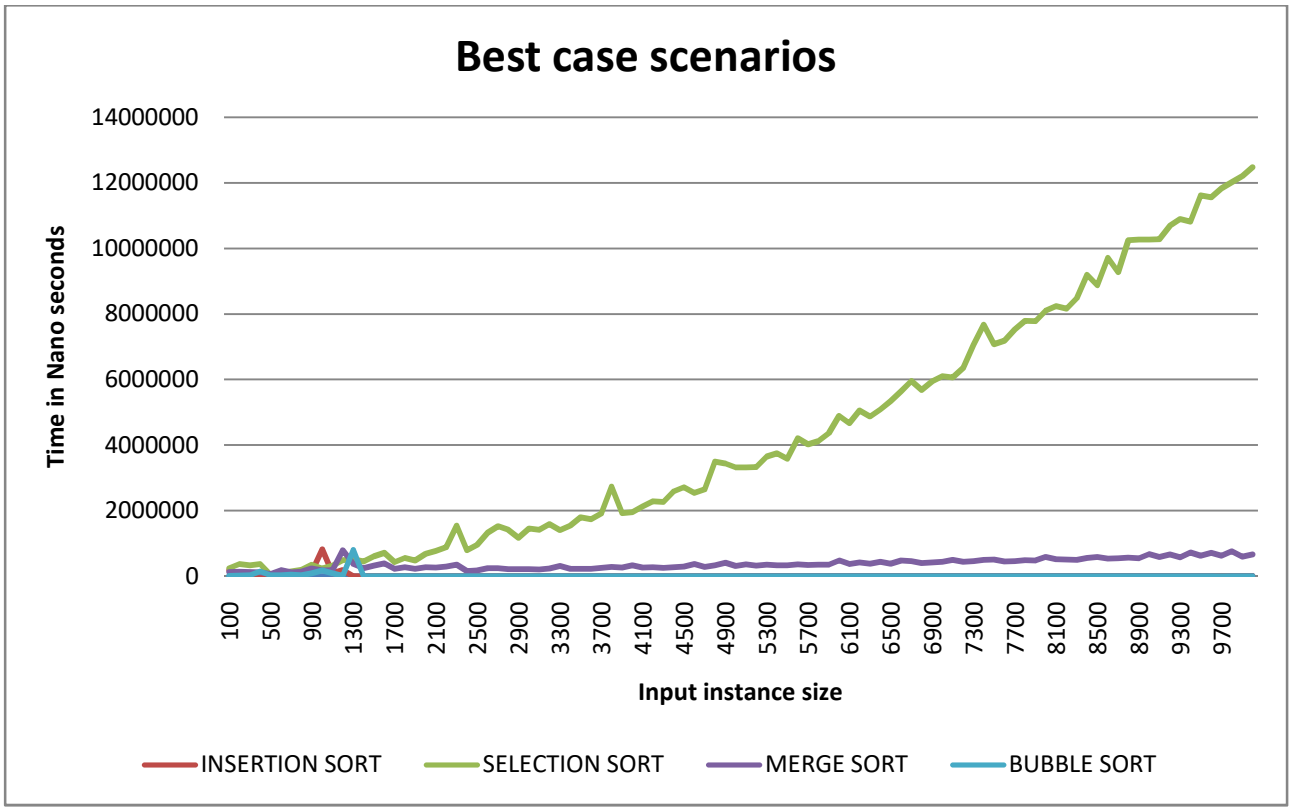
Algorithm	Best Case	Average Case	Worst Case
Bubble Sort	$O(n^2)$ O(n) with optimized code	$O(n^2)$	$O(n^2)$
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Merge Sort	O(n log(n))	O(n log(n))	O(n log(n))

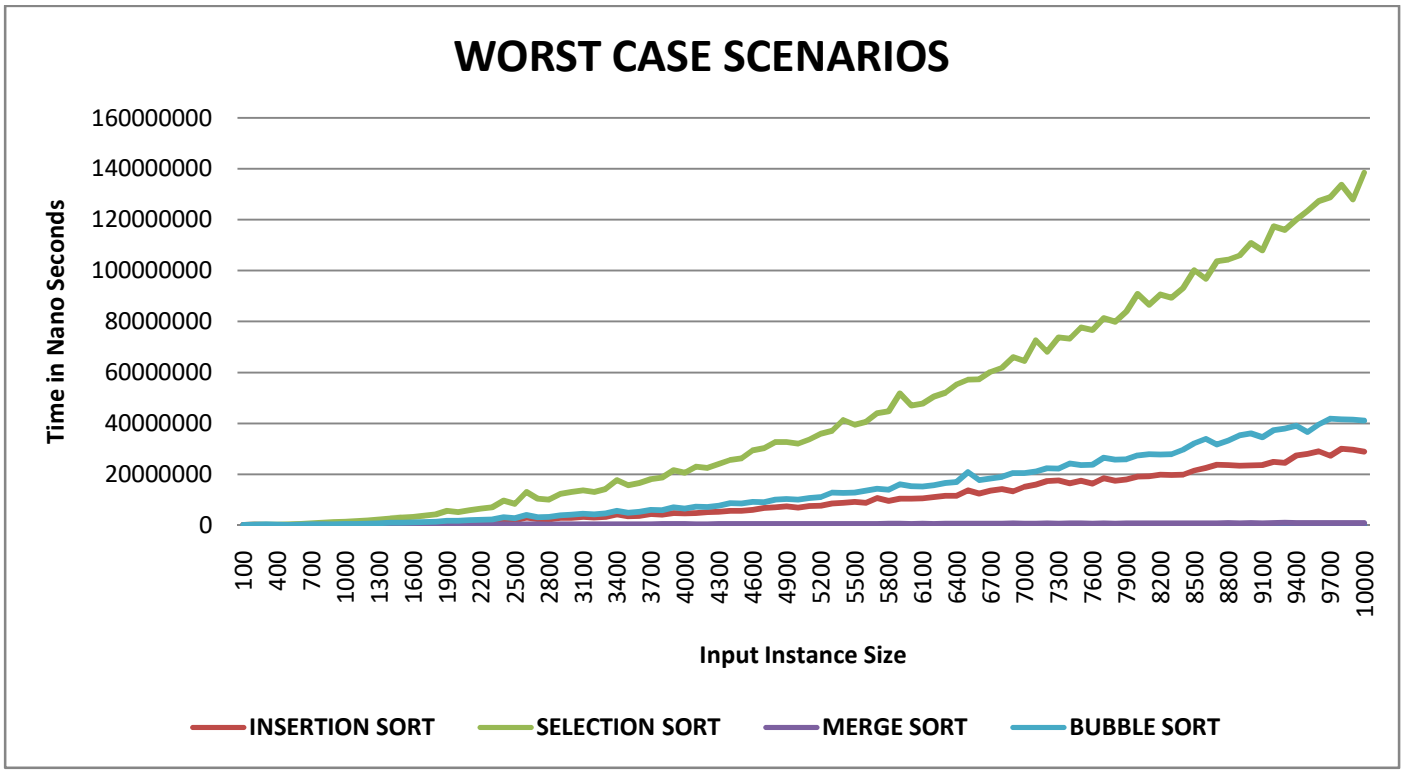
Analysis from the Graphs plotted using execution time :-

For the Empirical analysis, we run each sort for each case (Best, Worst and Average) with a varying range of input size starting from 100, 200, 300 so on up to 10000. For each of these inputs we run the program multiple times so as to get unbiased results. We record the time taken to sort each input and take the average of the multiple runs. Below are the graphs plotted with these execution times.



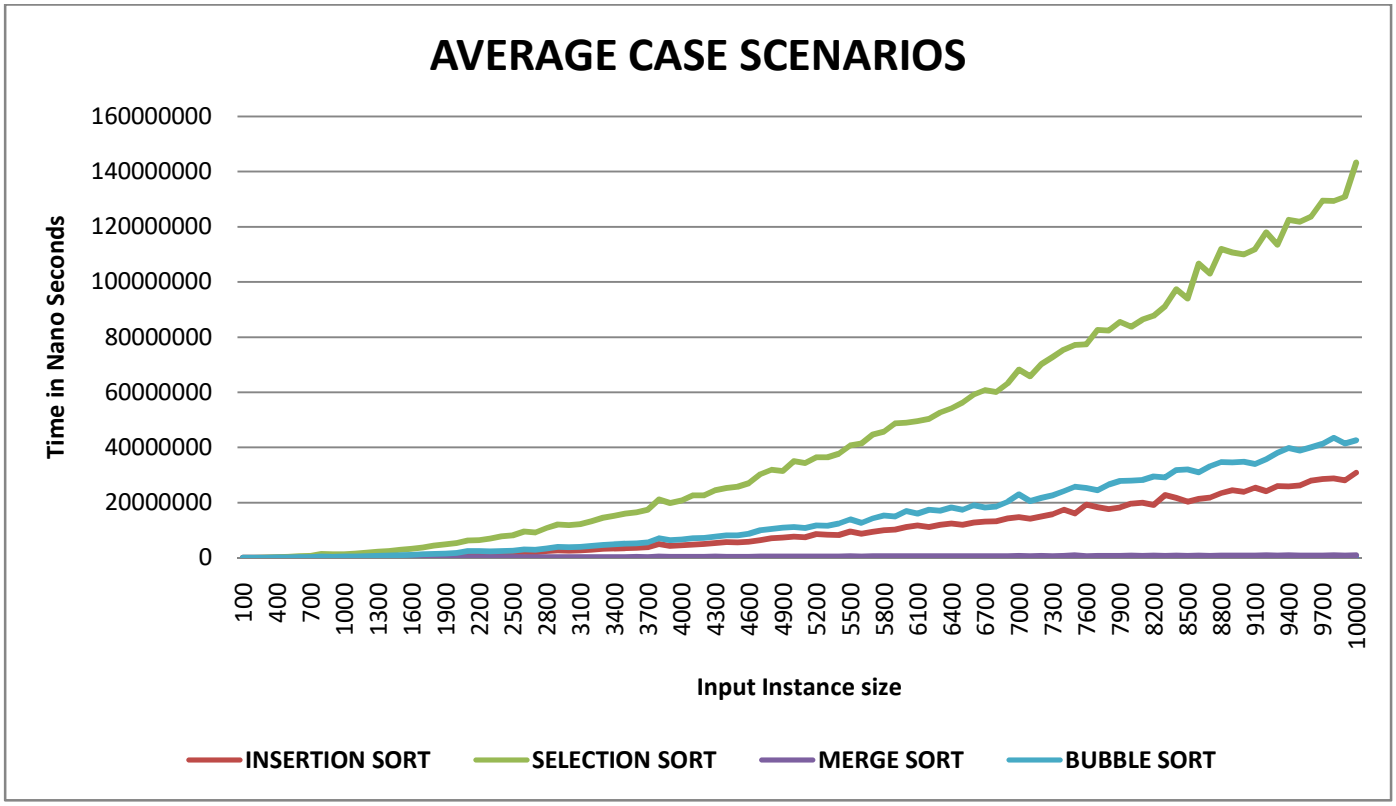
As seen in the graph, when the size of input is equal to or below 800 the 4 sort algorithms are overlapping and in a straight line. This shows that within this input size range of 100 to 800 the 4 sort algorithms are equally efficient that is even if the input size increases the time required to sort the input doesn't increase. After 800 input size we see that the execution time of Selection sort and Bubble sort starts to increase more rapidly. However the input size does not seem to affect the execution times of Insertion sort and Merge Sort even when the input size is increased by 100 times. If we look closely the Insertion sort is better than Merge sort in the best case scenarios. **Insertion sort is independent of the input size and hence the Most efficient**. Whereas **Bubble sort is the Least efficient**. **But if we use the optimized version of bubble sort it is equally efficient as insertion. The graph for the same is mentioned below:-**





As seen in this graph, when the size of input is equal to or below 1100 the 4 sort algorithms are overlapping and in a straight line. This shows that within this input size range of 100 to 1100 the 4 sort algorithms are equally efficient that is even if the input size increases the time required to sort the input doesn't

increase. However after 1100 input size we see that the execution time of Selection sort shoots up rapidly. Bubble sort and Insertion sort are comparable with similar growth with respect to input size. However the input size does not seem to affect the execution time of Merge Sort even when it is increased by 100 times. From this we can deduce that in the worst case scenarios **Merge Sort is independent of the input size and hence the most efficient**. Whereas, **Selection sort is the least efficient**.



As seen in this graph, when the size of input is equal to or below 1900, insertion, merge and bubble sort are more or less overlapping and in a straight line. This shows that they are equally efficient within this input size range. Execution time for **Selection sort** seems to increase from 1000 input size, and continues to increase rapidly making it the **least efficient of all**. Insertion and Bubble sort have a similar slope. **Merge sort** however has the its graph very close to the horizontal axis stating that even when the input size increases from 100 to 10000 it does not affect the execution time making it the **most efficient of all**.

If we **compare the behavior of Selection sort in best, average and worst case** we can see that for any input size between 100 to 10000 the time taken to sort the input never goes above $14*10^7$ nanoseconds in worst case and $15*10^7$ nanoseconds in best case. In average case selection sort is same as that in worst case. We can therefore deduce that the behavior of selection sort is slightly better in worst case than best case.

If we **compare the behavior of Insertion sort in best, average and worst case** we can see that for any input size between 100 to 10000 Insertion sort is along the horizontal axis and never goes above $1*10^7$ nanoseconds whereas in the worst case it does. In average case insertion sort is same as that in worst case. We can therefore deduce that the behavior of insertion sort is better in best case than worst case.

If we **compare the behavior of Merge sort in best, average and worst case** we can see that for any input size between 100 to 10000 the time taken to sort the input is slightly more in best case than in worst case. This difference is just marginal. In average case merge sort is same as that in worst case.

If we **compare the behavior of Bubble sort in best, average and worst case** we can see that for any input size between 100 to 10000 the time taken to sort the input never goes above $4*10^7$ nanoseconds in worst case but in best case it reaches $20*10^7$ nanoseconds. In average case bubble sort is same as that in worst case. We can therefore deduce that the behavior of bubble sort is much better in worst case than best case.

Theoretical Analysis:-

Advantages:-

- It is not necessary to implement difficult code, only algorithms are enough.
- Faster compared to Empirical Analysis

Disadvantages:-

- Results are approximate and may differ in actual implementations.

Empirical Analysis:-

Advantages:-

- Results are from actual implementation and can be trusted more.

Disadvantages:-

- It is subject to human error.
- Results are dependent on the resources you use like compilers, version used, hardware limitations of laptop etc.
- Results may differ from implementation to implementation, based on skills of coder, choice of language used for coding.

References:-

Textbook -> Introduction to algorithms by Thomas Cormen, Charles Leiserson, Ronald Rivest and Clifford Stein.

Also referred slides presented in class which are published on canvas.