Assignment 1: Analysis of Algorithm

Problem : Analyze the performance or efficiency of Merge Sort, Insertion Sort, Selection Sort and Bubble Sort.

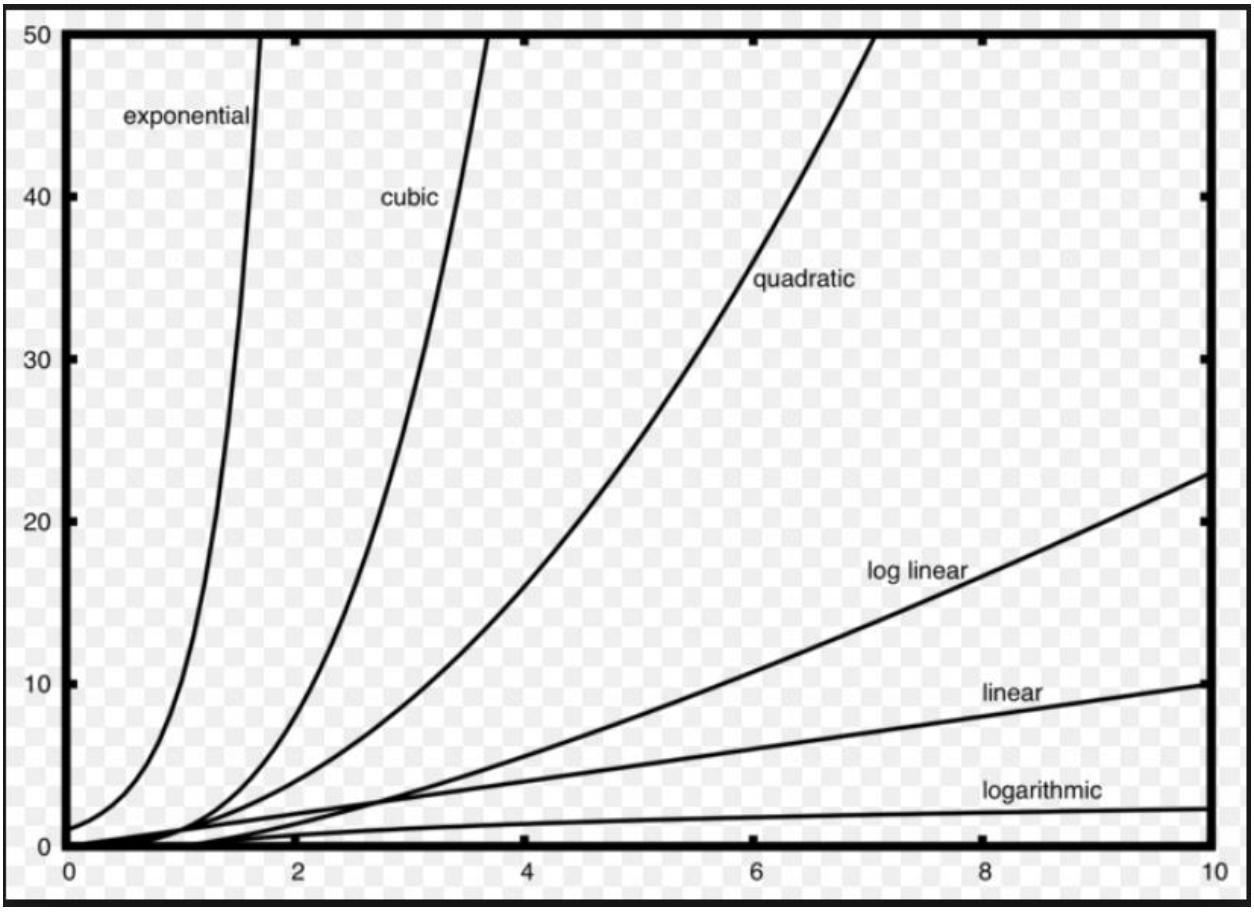
Method Used:

1. Theoretical Analysis
2. Empirical Analysis

Theoretical Analysis

	Merge Sort	Insertion Sort	Selection Sort	Bubble Sort
Average Case	Θ (nlog(n))	Θ (n^2)	Θ (n^2)	Θ (n^2)
Best Case	Ω (nlog(n))	Ω (n)	Ω (n^2)	Ω (n)
Worse Case	O(nlog(n))	O(n^2)	O(n^2)	O(n^2)

Understanding the Complexity graphs:



The order of growth in increasing order is as given as follows:

$1 < \log(\log(n)) < \log(n) < n < n\log(n) < n^2 < n^3 < 2^n < n!$

Merge Sort:

Merge Sort is a recursive algorithm which uses divide-and-conquer technique to sort the numbers. The Mergesort algorithm divides the input into many subseries resulting single numbers to sort at the end of these divisions and then merging and sorting these series together one by one.

Algorithm:

```
MERGE(A, p, q, r)
1 n1=q- p + 1
2 n2= r – q
3 let L[1..n1+1] an= R[1..n2 + 1] be new arrays
4 for i = 1 to n1
5     L[i] = A[p + i- 1]
6 for j = 1 to n2
7     R[j] = A[q + j]
8 Initializing L[n1 + 1]
9 Initializing R[n2 + 1]
10 i = 1
11 j = 1
12 for k = p to r
13     if L[i] <= R[j]
14         A[k] = L[i]
15         i = i + 1
16     else A[k] = R[j]
17         j = j+1
```

```
MERGE-SORT(A, p, r)

1 if p < r
2    q = [(p + r))2]
3    MERGE-SORT(A, p, q)
4    MERGE-SORT(A, q + 1, r)
5    MERGE(A, p, q, r)
```

Time taken by each step:

When an Algorithm uses recursive call its running time is best described by the recurrence. In the divide and conquer strategy of algorithm the running time is taken from the three steps from the basic strategy. Here, it is Divide , Conquer and Combine.

If the size of the input is smaller than some constant c, then the runtime T(n) is taken to be constant $\Theta(1)$. But, if the input size is large enough and if the algorithm divides the input

sequence into a subparts of size 1/b size of the original input sequence. Then the time taken by algorithm to solve that n/b size problem is given by T(n/b) and thus to solve the entire problem it takes total a*T(n/b) time. The divide and combine operations for the algorithm takes D(n) and C(n) time respectively.

Divide: The divide step just computes the middle of the subarray, which takes constant time. Thus, D(n) = Θ (1)
Conquer: We recursively solve two subproblems, each of size n=2, which contributes 2T(n/2) to the running time.
Combine: We have already noted that the MERGE procedure on an n-element subarray takes time O(n), and so C(n) = O(n)

Combining them we get T(n) = 2T(n/2) + O(n) which by using the master method gives T(n) = (nLogn).
The time taken by the algorithm is same for all the cases, as irrespective of the degree to which the input is sorted the algorithm keeps dividing the array into single elements and combines all the n elements by comparing each element. Thus, the running time equals to O(nlogn) for all the cases.

Insertion Sort:

In Insertion sort and number that is to be sorted is considered to be a key and it is assumed that all the cards before that are sorted, then the appropriate position of that number(n) is found from the already sorted A[1…n-1] series by comparing it with the numbers till a number less than that is found and inserted in that position. This method is exactly similar to how we sort the playing cards by removing one card at a time from the deck.

Algorithm:

```
INSERTION-SORT(A)
for j = 2 to A.length
     key = A[j]
     // insert A[j] into sorted sequence A[1 .. j-1]
     i = j-1
    while i > 0 and A[i] > key
            A[i+1] = A[i]
            I = i-1
    A[i+1] = key
```

Algorithm Analysis:

INSERTION-SORT(A)	Cost	Times
1. for j = 2 to A.length	c1	n
2. key = A[j]	c2	n-1

```
    // insert A[j] into sorted sequence A[1 .. j-1]
4.  i = j-1                                  c4          n-1
5   while i > 0 and A[i] > key               c5          Summatiion(j=2 to n)  t_j
6.          A[i+1] = A[i]                    c6          Summatiion(j=2 to n)  (t_j – 1)
7.          I = i-1                          c7          Summatiion(j=2 to n)  (t_j – 1)
8.  A[i+1] = key                             c8          n-1
```

Summing the product of cost and times we get the Total Time taken by the algorithm:

For the best case the total times turns out to be :
$T(n) = (c1 + c2 + c4 + c5 + c8)n – (c2 + c4 + c5 + c8)$

Since the summation for the time taken by the while loop at instruction 5 turns out to be just n-1 as in the best case the condition (A[i] > key) returns false and we exit the loop at the first iteration of while loop itself for all the n-1 numbers to be sorted assuming that the first element is always sorted.

Thus, we can represent the equation as linear equation an+b giving the time taken as O(n).

For worst case, the input series is in the descending order and hence, the summation for while loop statement and its two inner statements turns out to be the arithematic series for sum of numbers operation and thus we represent the equation as an^2+bn+c giving the time taken for worst case to be $O(n^2)$.

Average case is almost equal to worst case. Suppose we choose the random n numbers and try to sort them. Assuming the input to be actually averagely sorted input series then on average, therefore, we check half of the subarray A[1..j-1] and so t_j is about j/2 thus applying the series expansion the average running time turns out to be like worst case.

Selection Sort:

Algorithm:

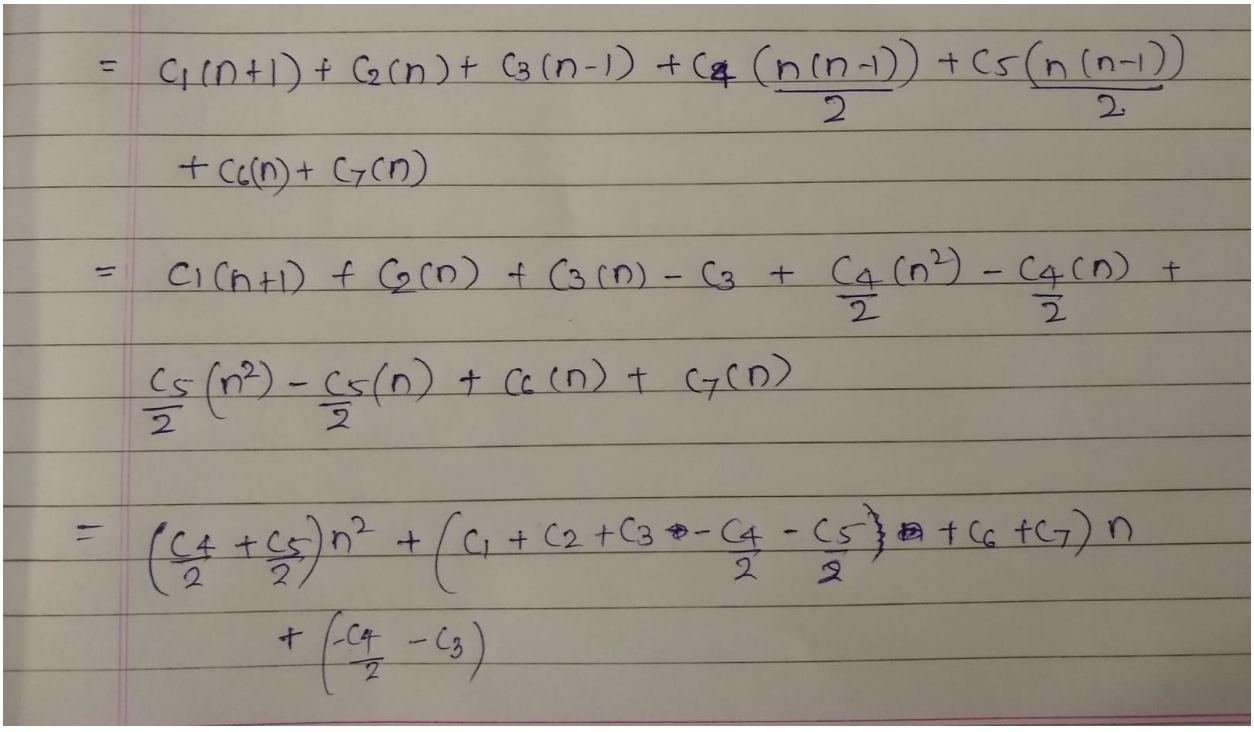
```
Selection-Sort
1 for i = 0 to n – 1
  /* set current element as minimum*/
2     min = i
  /* check the element to be minimum */
3     for j = i+1 to n

4        if list[j] < list[min]
5           min = j;
     /* swap the minimum element with the current element*/
6     if indexMin != i  then
7        swap list[min] and list[i]
```

Algorithm Analysis:

Selection-Sort	Cost	times
1 for i = 0 to n – 1	c1	n+1
2 min = i	c2	n
3 for j = i+1 to n	c3	n*(n-1) / 2
4 if list[j] < list[min]	c4	n*(n-1) / 2
5 min = j;	c5	n-1
6 if indexMin != i	c6	n
7 swap list[min] and list[i]	c7	n

Solving the equation further we get following result:



We can thus express the above equation as a quadratic equation $an^2 + bn + c$. For the latrger values of n, b(n) and c doesn't contribute much to the runtime calculation. Thus, we ignore those terms.

Selection Sort iterates till the end to find the minimum of all the elements in the array. Even if the array is sorted, the iteration continues till the end. In the first iteration it takes n comparisons, in the second it takes (n-1) till it reaches the end where it takes 1 comparison. When repeatedly smallest element is being searched, it takes n+(n-1)+...+1 which is $(n(n+1))/2 = (n^2+n)/2$ which in this case has the time complexity as $\Omega(n^2)$.

In the worst case, the array is arranged in the reverse order and every element is compared with every other elements. For e.g., {8,4,3,2,1}, here, 8 is compared to all the other elements

so there will be 5 comparisons. Similarly for 4, 3 and it goes on until it reaches 1 which makes it n*(n-1)/2 comparisons.

It has been observed that selection sort outperforms bubble sort. Selection and insertion have almost similar time complexities for average case, but the runtime for the insertion sort always varies. That's not the case in selection sort as it will always iterate till the end to find the smallest element. That's the reason it is often outperformed by the insertion sort. Selection sort has the time complexity of θ(n^2) for average case.

BubbleSort:

Bubble sort follows the method of bubbling or pushing the bigger number at the end of the series one after another byt comparing its consequent number.

Algorithm:

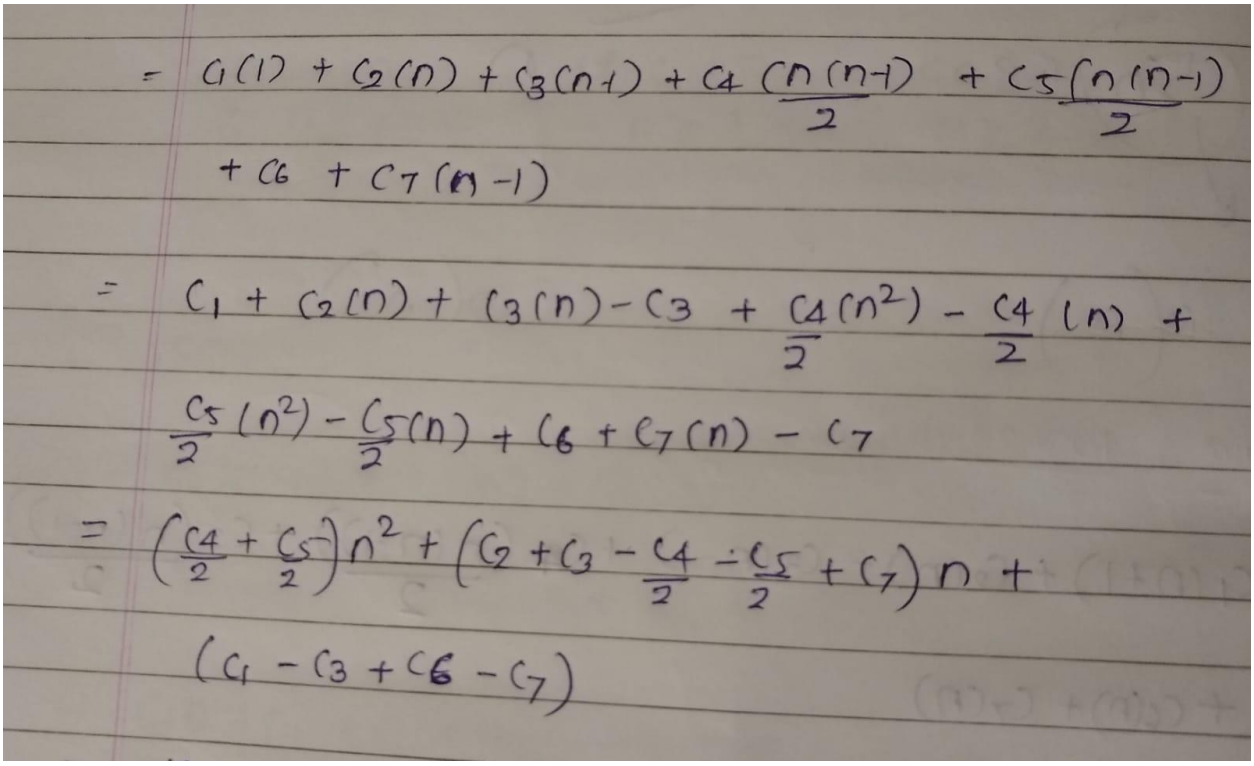
```
BubbleSort
1   n= arr.Length
2   for i = 0 to n-1
3     swapped_flag = false
4     for j = 0 to n-1
       /* compare the adjacent elements */
5       if list[j] > list[j+1]
         /* swap them */
6          swap( list[j], list[j+1] )
7          swapped_flag = true
     /*if no number was swapped that means
     array is sorted now, break the loop.*/

8       if(not swapped_flag)
9       break
```

Algorithm Analysis:

BubbleSort	Cost	times
1 n= arr.Length	c1	1
2 for i = 0 to n-1	c2	n
3 swapped_flag = false	c3	n-1
4 for j = 0 to n-i-1	c4	n*(n-1)/2
5 if list[j] > list[j+1]	c5	n*(n-1)/2
6 swap(list[j], list[j+1])	c6	
7 swapped_flag = true		
8 if(not swapped_flag)	c7	n-1
9 break		

Summing the product of count and times we get the following answer:



$$= C_1(1) + C_2(n) + C_3(n-1) + C_4\frac{(n(n-1)}{2} + C_5\frac{(n(n-1)}{2} + C_6 + C_7(n-1)$$

$$= C_1 + C_2(n) + C_3(n) - C_3 + \frac{C_4}{2}(n^2) - \frac{C_4}{2}(n) + \frac{C_5}{2}(n^2) - \frac{C_5}{2}(n) + C_6 + C_7(n) - C_7$$

$$= \left(\frac{C_4}{2} + \frac{C_5}{2}\right)n^2 + \left(C_2 + C_3 - \frac{C_4}{2} - \frac{C_5}{2} + C_7\right)n + (C_1 - C_3 + C_6 - C_7)$$

Expressing this as a quadratic equation we get it in the terms of n^2 .

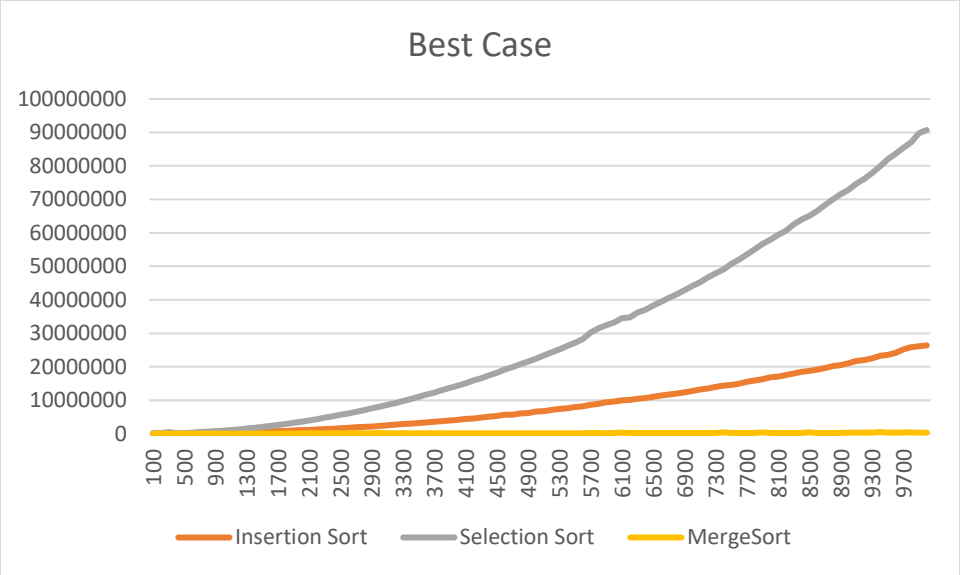
For Bubble sort , for the best case as the list will be sorted it will just compare the n-1 elements in one pass and as no swapping was done it will make the for loop to be exited thus making the best case runtime to be Ω (n-1)

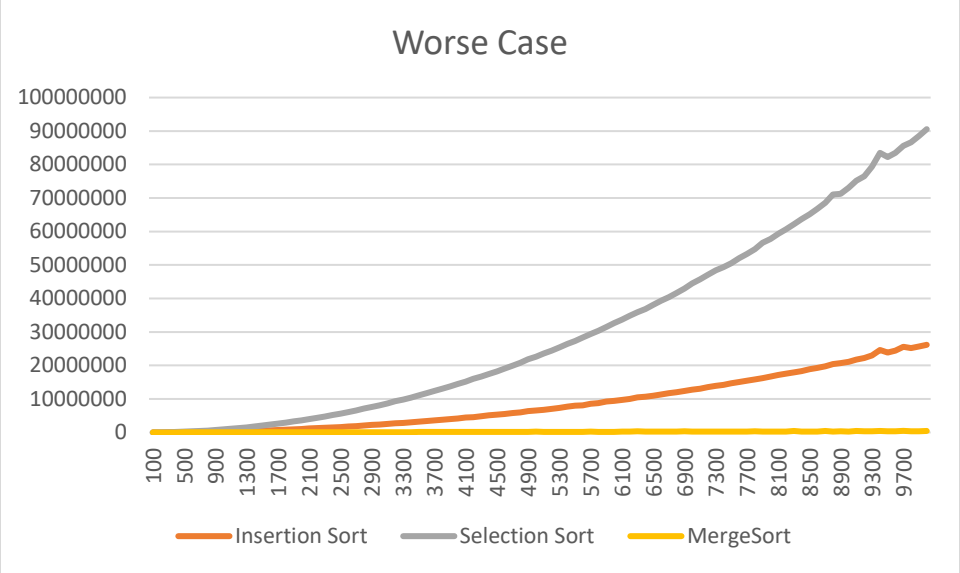
For the Average and the Worst case the input being in the descending order it will swap the element at each comparison and thus this makes total n*(n-1)/2 comparisons. This gives the runtime for the Bubble Sort algorithm to be $O(n^2)$.

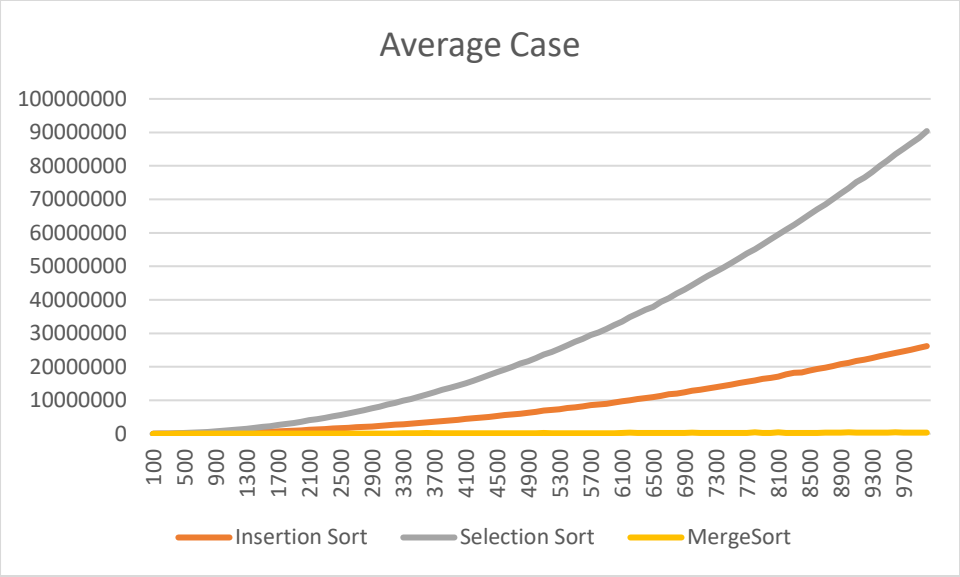
From the graphs it can be seen that Insertion sort performs well for the smaller input values but as the input size increases Merge Sort starts performing well which goes exactly like it has been stated that Insertion sort works well for small input size but Merge sort's effect of log(n) starts reflecting as the input size increases like it can be seen in our graph when the input size increases to 10000. Also, Selection sort takes lesser amount than Bubble sort and thus outperforms it.

Empirical Analysis:

Graphs comparison between Merge Sort, Insertion Sort and Selection Sort:

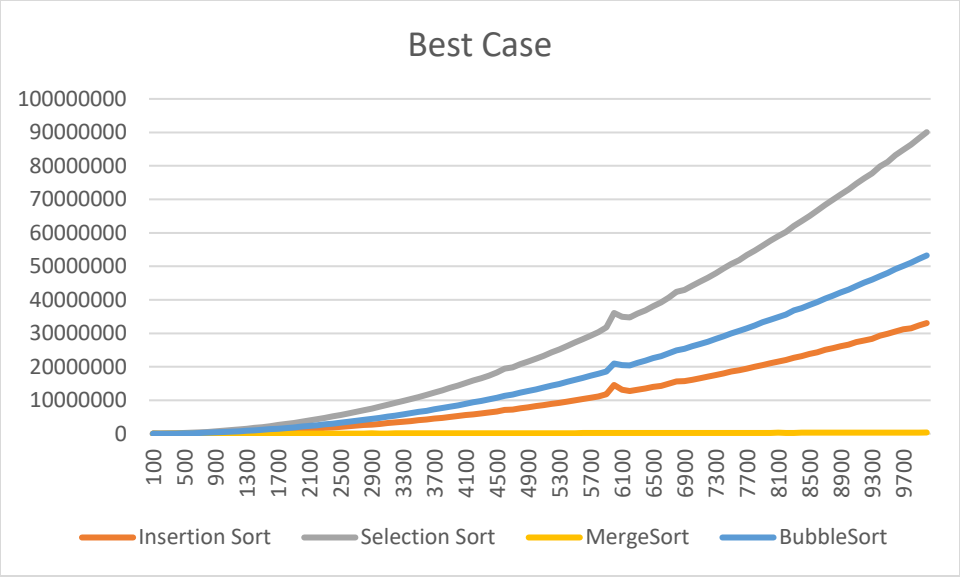

Best Case
100000000
90000000
80000000
70000000
60000000
50000000
40000000
30000000
20000000
10000000
0
100
500
900
1300
1700
2100
2500
2900
3300
3700
4100
4500
4900
5300
5700
6100
6500
6900
7300
7700
8100
8500
8900
9300
9700
Insertion Sort
Selection Sort
MergeSort

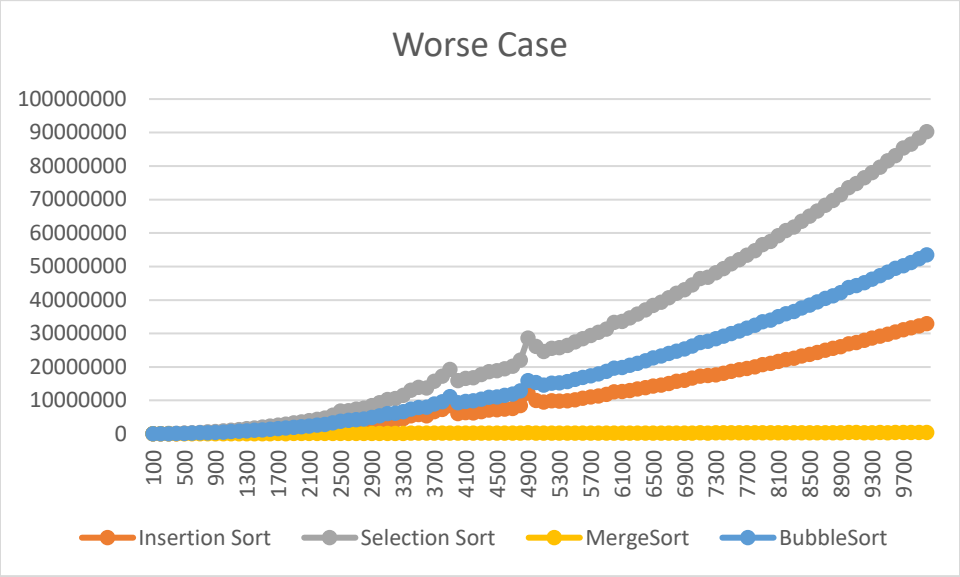

Worse Case
100000000
90000000
80000000
70000000
60000000
50000000
40000000
30000000
20000000
10000000
0
100
500
900
1300
1700
2100
2500
2900
3300
3700
4100
4500
4900
5300
5700
6100
6500
6900
7300
7700
8100
8500
8900
9300
9700
Insertion Sort
Selection Sort
MergeSort

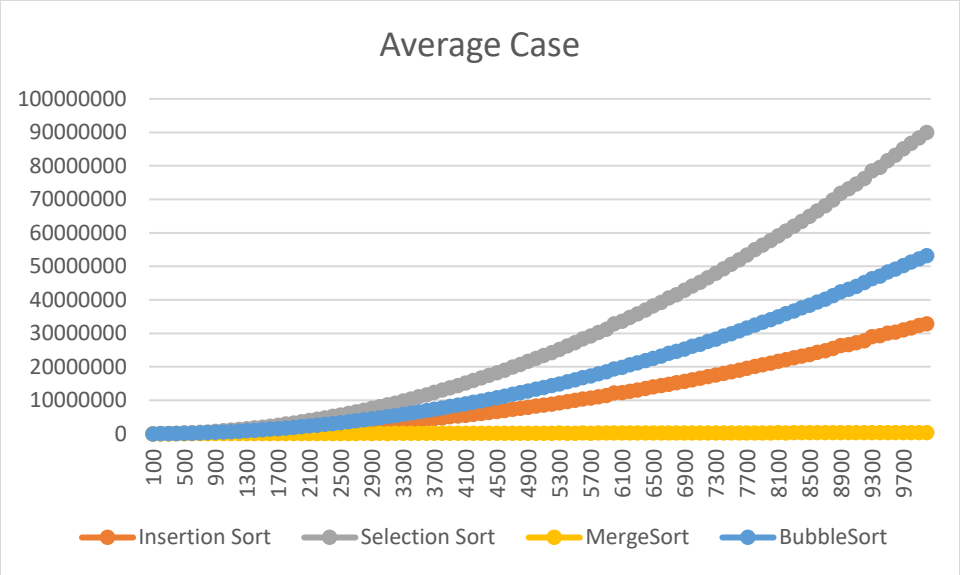


As can be seen from the above graph it goes with the above explanation:
For Best Case, Merge Sort follows logarithmic curve, Insertion Sort follows linear curve and Selection follows quadratic curve.
For Worst Case, Merge Sort follows logarithmic curve, Insertion Sort follows quadratic curve and Selection also follows quadratic curve.
For Average Case, Merge Sort follows logarithmic curve, Insertion Sort follows quadratic curve and Selection also follows quadratic curve.
Merge Sort takes least time as log(n) for all the cases. Followed by Insertion sort which tales linear time for best case and then quadratic for the rest two cases. Whereas, Selection Sort takes quadratic for all the three cases.

<u>Graphs comparison between Merge Sort, Insertion Sort, Selection Sort and Bubble Sort:</u>


Best Case
100000000
90000000
80000000
70000000
60000000
50000000
40000000
30000000
20000000
10000000
0
100
500
900
1300
1700
2100
2500
2900
3300
3700
4100
4500
4900
5300
5700
6100
6500
6900
7300
7700
8100
8500
8900
9300
9700
Insertion Sort
Selection Sort
MergeSort
BubbleSort


Worse Case
100000000
90000000
80000000
70000000
60000000
50000000
40000000
30000000
20000000
10000000
0
100
500
900
1300
1700
2100
2500
2900
3300
3700
4100
4500
4900
5300
5700
6100
6500
6900
7300
7700
8100
8500
8900
9300
9700
Insertion Sort
Selection Sort
MergeSort
BubbleSort



For Best Case, Merge Sort takes the least runtime of logarithmic curve, Insertion Sort follows linear curve and takes slightly more time than Merge sort. Whereas, Bubble Sort takes lesser time n than Selection Sort as the check for is_sorted = True makes the loop to exit but in selection sort the code still checks for all the n values.
For Worst Case and Average Case, Merge Sort takes the least time and follows logarithmic curve, followed by Insertion Sort taking more time than Merge Sort and then followed by Bubble Sort and Selection sort. As, in the worst case the input being in descending order it swaps almost all elements in Bubble sort and checks or compares for all the values of n with all other n values thus increasing its runtime.