Homework Assignment 1- Sparsh Oza.

Abstract

The process of determining the computational complexity, such as time, space, and other resources, that the algorithm demands is known as algorithm analysis. Although computing time is frequently what we want to measure to understand the method, other resources like memory, connection bandwidth, or computer hardware are also important. This report's major objective is to examine the insertion sort (naive and improved) and merge sort running times for various input sizes ranging from 10000 to 2500000 and various vector dimensions. Additionally, we will evaluate the effectiveness of the algorithms using input vectors with varying orders, and we will conduct a test at least ten times to determine the average algorithmic running time for various input types, input sizes, and vector dimensions.

Introduction:

An algorithm's execution time may be understood as a function of the input's size. Typically, the number of items in the input is used to determine the size of the input (array size n for sorting).

The amount of the input affects how time-consuming the insertion sort is. Insertion sort requires more time the larger the input is. Furthermore, depending on how closely sorted two identical-size input sequences are already, sorting them takes various amounts of time.

To sort a list or array, merge sort applies the divide and conquer method. Divided into two sub-arrays of n/2 items each, the n-element array that must be sorted is then sorted recursively using merge sort. The sorted result is then created by merging the two sorted sub-arrays.

Three versions of the sorting algorithms are available: merge sort, modified insertion sort, and naive insertion sort. We have employed several testing vectors with dimensions n = 10, 25, and 50 and sizes m = 10000, 25000, 50000, 100000, 250000, 500000, 1000000, and 2500000. To further evaluate the performance of these methods for various mixtures of size, dimension, and order, we employed random, sorted, and inverse sorted vectors.

Results and Evaluation:

The average runtime of naïve insertion sort for different parameters is described in the table below.

Insertion Sort Runtime in milliseconds (Average Summary)

	n = 10		
m	Random Vector	Sorted Vector	Inverse Sorted Vector
0	0	0	0
10000	894.2	0.1	1861.95
25000	5855.05	0.9	10599.85
50000	22454.4	1.1	40221.55
100000	93219.7	2.8	158850.5
250000	-	7.2	-
500000	-	14.7	-
1000000	-	30.1	-
2500000	-	77.6	-

	n = 25		
m	Random Vector	Sorted Vector	Inverse Sorted Vector
0	0	0	0
10000	2590.2	0.8	5092.5
25000	14394.5	2.3	27448.4
50000	54718	4.2	104620.9
100000	237804.7	8.4	-
250000	-	22.3	-
500000	-	45.5	-
1000000	-	93.4	-
2500000	-	260	-

	n = 50		
m	Random Vector	Sorted Vector	Inverse Sorted Vector
0	0	0	0
10000	5441.9	2.5	10755.7
25000	30211.4	7.1	57779.1
50000	120838.5	12.1	236185.3
100000	-	22.9	-
250000	-	57.3	-
500000	-	118.3	-
1000000	-	238.8	-
2500000	-	557.5	-

As we can see, the size of the input and the dimension of the vector array both affect how long the insertion sort takes to complete. When we supply a sorted vector as input, the insertion sort's time complexity is at its lowest, and when we pass a reverse sorted vector, it is at its highest.

According to the aforementioned findings, the insertion sort does not function effectively as the input size grows. When the input size for the sorting algorithm exceeds 250000 for n=10, we are unable to calculate its running time. The running time cannot be calculated for n=25, 50 with input sizes more than 100000.

The insertion sort works effectively for sorted input vectors, which is the ideal situation. The sorted vector input's running time is equal to best case time complexity O(n). As we raise either the input size n or the vector dimension n, the running time for the random and inverse vector sort virtually doubles and approaches the worst-case complexity of $O(n^2)$ for insertion sort.
The table below lists the average runtime of enhanced improved sort for various parameters.

Improved Insertion Sort Runtime in milliseconds (Average Summary)

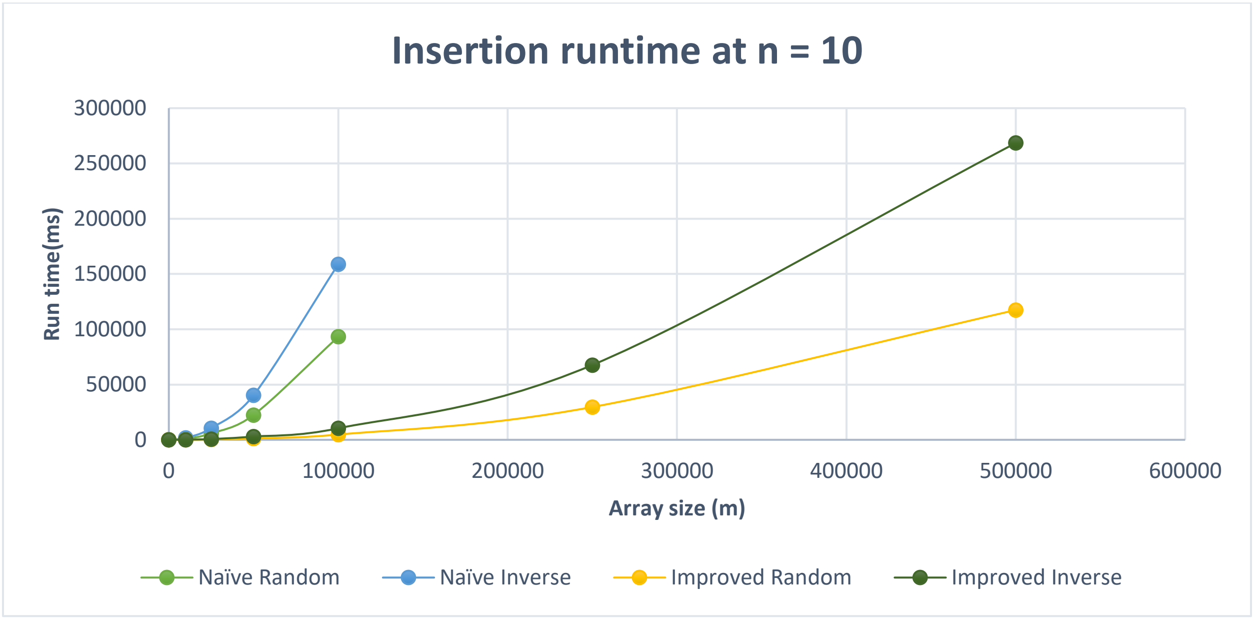
	n = 10		
m	Random Vector	Sorted Vector	Inverse Sorted Vector
0	0	0	0
10000	54.1	0.1	124.05
25000	342	0.6	710.6
50000	1314.9	1.1	2981.95
100000	4732.4	1.7	10406.3
250000	29629	4.7	67659.9
500000	117408.4	9.8	268478
1000000	-	19.1	-
2500000	-	44.6	-

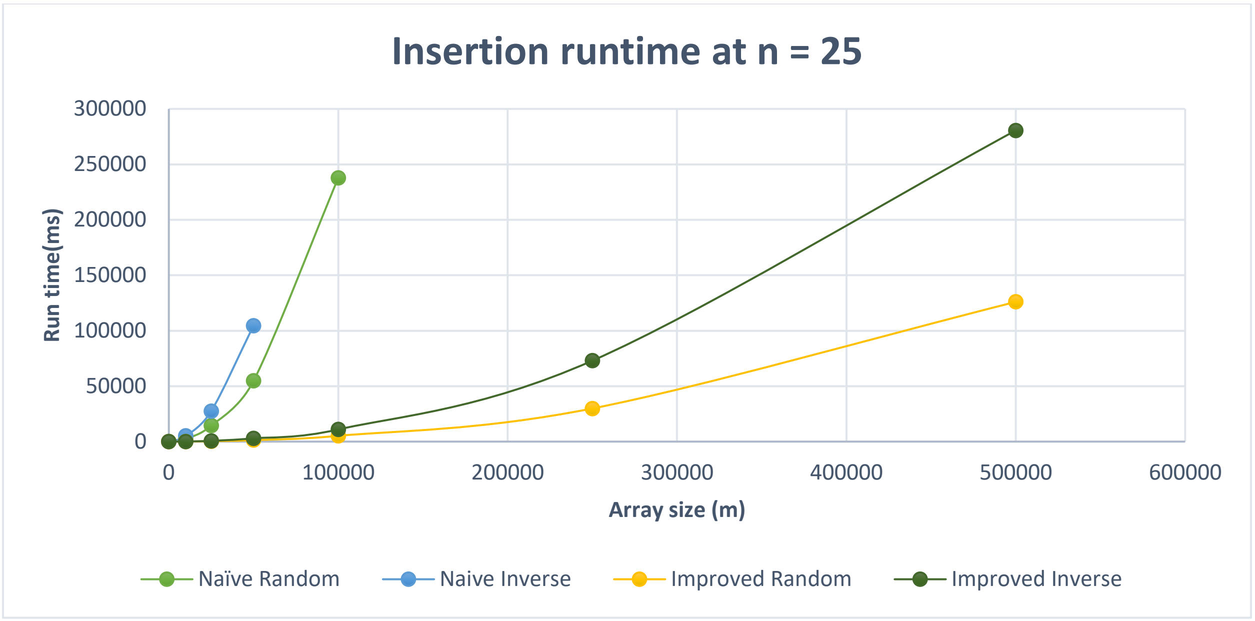
n = 25			
m	Random Vector	Sorted Vector	Inverse Sorted Vector
0	0	0	0
10000	60.1	0.6	126.3
25000	382.2	1.4	721.5
50000	1381.3	2.2	2992.6
100000	5196	4.7	10966.2
250000	29875.5	11.2	72948
500000	126006.55	23.3	280619.6667
1000000	-	47.8	-
2500000	-	129.4	-

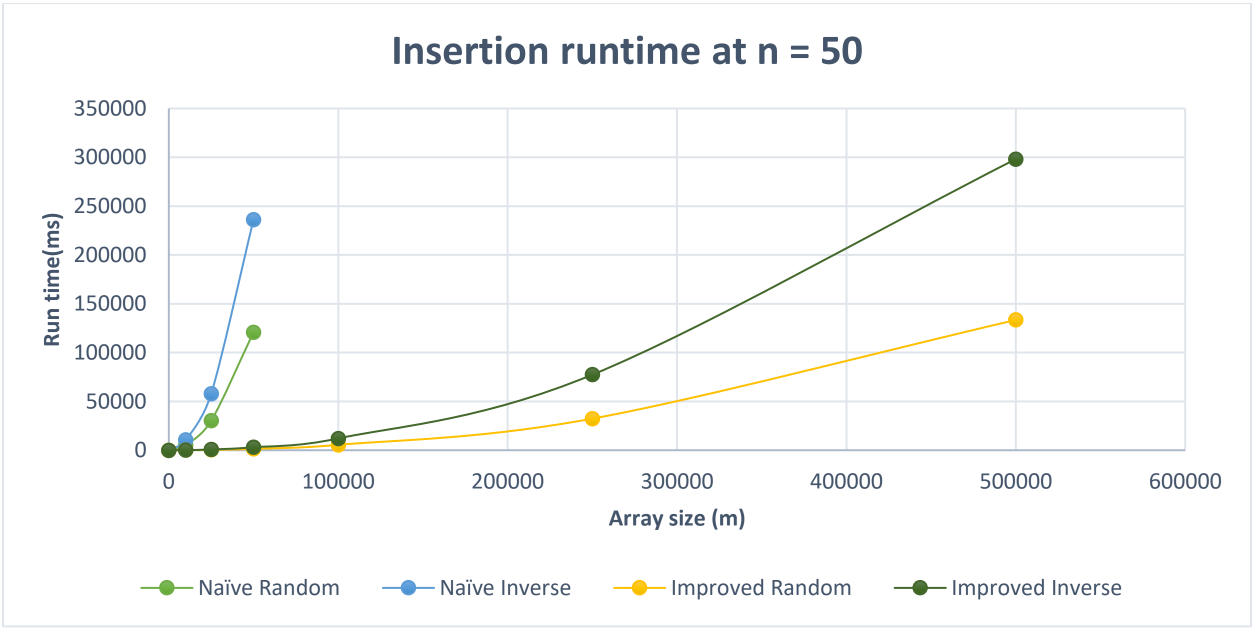
n = 50			
m	Random Vector	Sorted Vector	Inverse Sorted Vector
0	0	0	0
10000	65.8	1.2	128.6
25000	392	3.1	786.65
50000	1452.8	6.6	3086.4
100000	5527.45	12.6	12101.7
250000	32364.2	29.8	77353
500000	133410	57.4	298194.6667
1000000	-	112.3	-
2500000	-	296.3	-

Although the enhanced insertion sort outperforms the naive insertion sort in terms of performance, it behaves similarly regardless of the input size. The algorithm's execution time grows together with the size of the input. The size of the input vector has less of an effect on how quickly the insertion sort runs. The sorted input vector continues to represent the ideal situation, and the inverse sorted vector continues to represent the ideal situation, which is identical to the ideal and idealized time complexity.

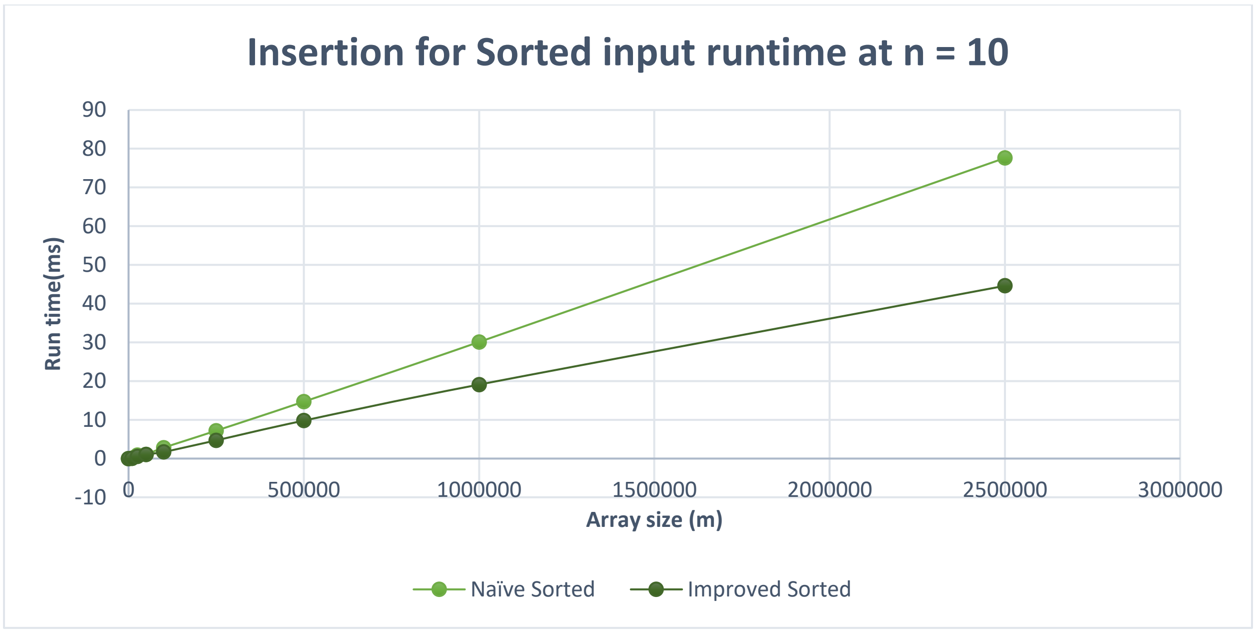
The comparison of input size and runtime for various vector dimensions, as well as the best-case and worst-case time complexity for insertion sort, are shown below. The graphs demonstrate the comparison between both the improved and naïve insertion sort for the given values of n.

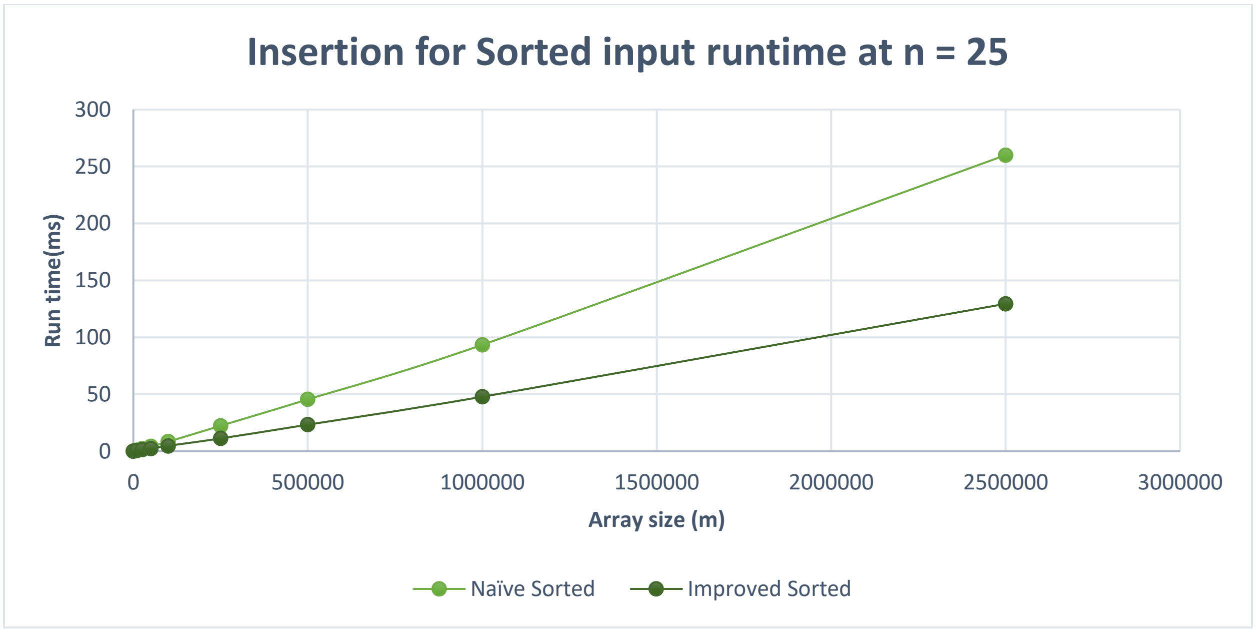


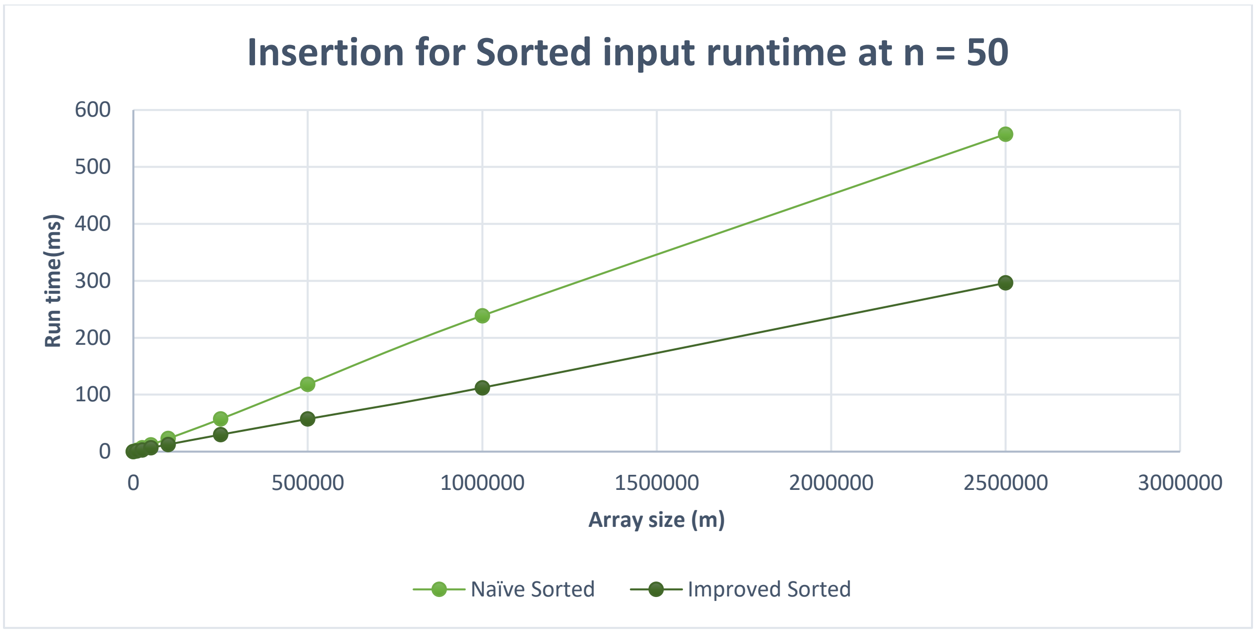




Below are the comparisons for sorted vectors, as the values were too small to be displayed on the above graphs a separate set of comparisons were made for viewers convenience.


Insertion for Sorted input runtime at n = 10
Run time(ms)
90
80
70
60
50
40
30
20
10
0
-10
0
500000
1000000
1500000
2000000
2500000
3000000
Array size (m)
Naïve Sorted
Improved Sorted


Insertion for Sorted input runtime at n = 25
Run time(ms)
300
250
200
150
100
50
0
0
500000
1000000
1500000
2000000
2500000
3000000
Array size (m)
Naïve Sorted
Improved Sorted



The aforementioned findings demonstrate that the enhanced insertion sort runs more quickly than the naive insertion sort and displays the expected behavior of insertion sorting. For all of the input sizes and dimensions that are specified, we are able to successfully calculate the insertion sort's running time. Since we pre-calculate the length of the vector before sorting the items, the vector dimension size does not significantly affect the algorithm's execution time for higher input size values. However, the algorithm's execution time grows exponentially as we increase the amount of the input. The only situation where the vector dimension and input size only slightly affect the running time is when the input vector is sorted, which nevertheless has the greatest performance of all the input order combinations and is identical to the best-case time complexity.

The tables below list the average merge sort runtimes for various parameter values.

Merge Sort Runtime in milliseconds (Average Summary)

n = 10			
m	Random Vector	Sorted Vector	Inverse Sorted Vector
0	0	0	0
10000	1.9	1.1	1.4
25000	4.6	3.4	3.8
50000	9.7	6.8	7.4
100000	21.6	14.3	16.4
250000	58	39.6	44.8
500000	111.1	79	101.7
1000000	236.5	170	187.8
2500000	501.4	321.2	373.2

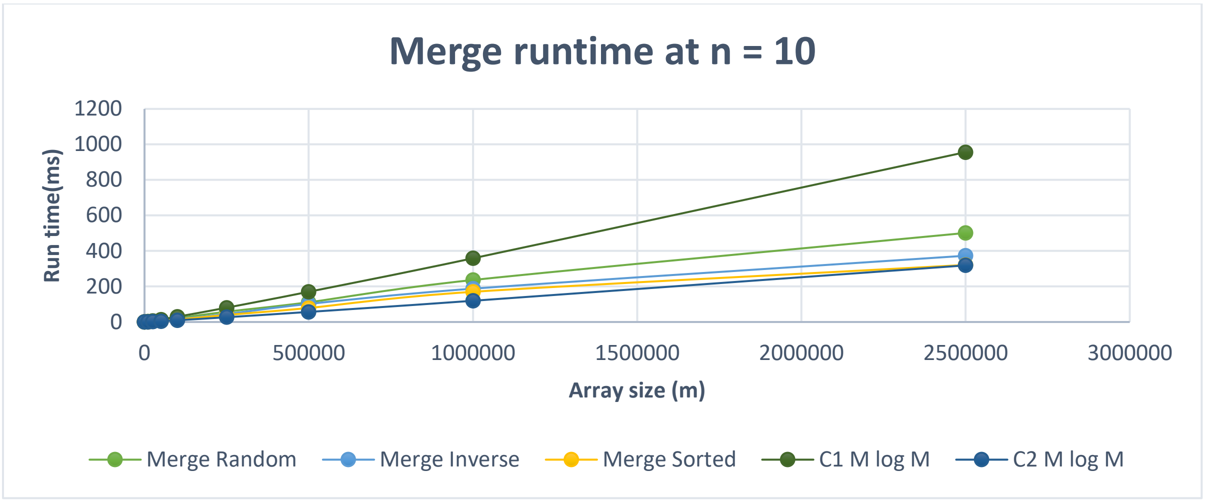
n = 25			
m	Random Vector	Sorted Vector	Inverse Sorted Vector
0	0	0	0
10000	2.1	1.6	1.8
25000	5.6	4.5	4.9
50000	11.8	9.2	9.5
100000	22.8	18.4	20.3
250000	62.5	48	50.7
500000	127.7	99	108.7
1000000	277.1	205	221.8
2500000	523.6	351.9	413.6

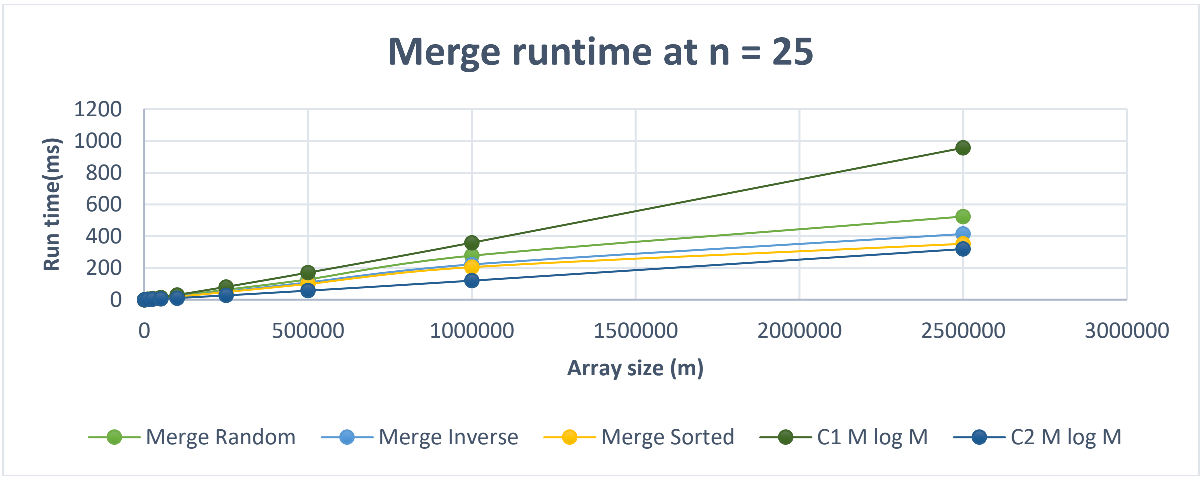
n = 50			
m	Random Vector	Sorted Vector	Inverse Sorted Vector
0	0	0	0
10000	2.4	2.2	2.1
25000	7.2	6	5.8
50000	14.1	12.1	11.6
100000	28.9	25.2	23.7
250000	69.4	66.7	62.1
500000	144.6	133.9	122.9
1000000	283.8	279.8	244.4
2500000	531.1	396.2	430.2

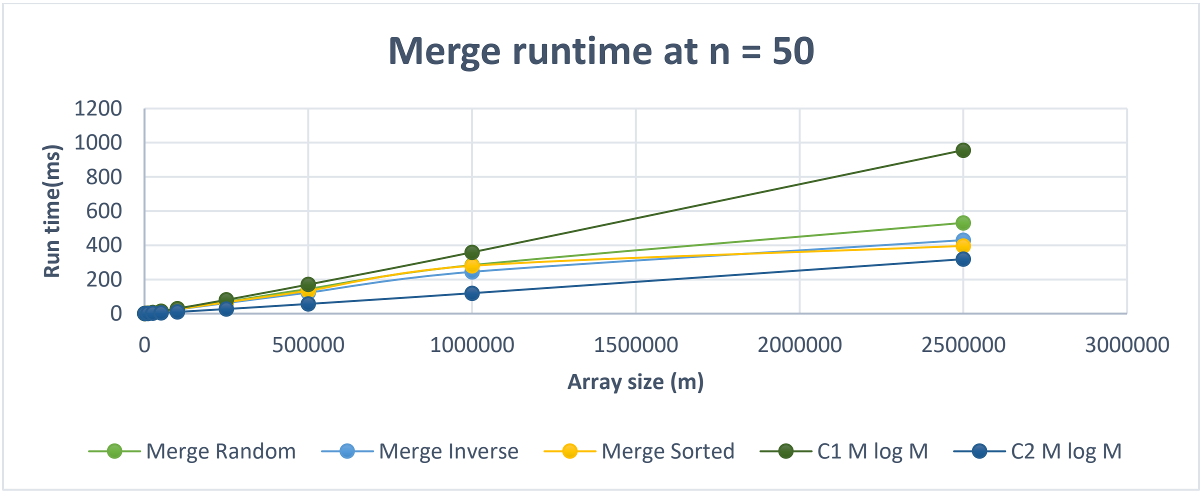
Both the naive and refined insertion sort don't perform as well as the merging sort. The algorithm's execution time only slightly rises with the amount of the input. The input's order has little bearing on how long the merging sort takes to complete.

The running time complexity for Merge sort is O(nlogn). To help calculate the variations in the table for theoretical and practical values we use a constant C1 and C2 multiplied by nlogn or in this case mlogm to help show the merge sort implementation falls within the boundaries of the theoretical definitions.

The worst-case time complexities for merge sort are shown below along with the input size vs. runtime comparison for various vector dimensions.


Merge runtime at n = 10
Run time(ms)
1200
1000
800
600
400
200
0
0
500000
1000000
1500000
2000000
2500000
3000000
Array size (m)
Merge Random
Merge Inverse
Merge Sorted
C1 M log M
C2 M log M


Merge runtime at n = 25
Run time(ms)
1200
1000
800
600
400
200
0
0
500000
1000000
1500000
2000000
2500000
3000000
Array size (m)
Merge Random
Merge Inverse
Merge Sorted
C1 M log M
C2 M log M


Merge runtime at n = 50
Run time(ms)
1200
1000
800
600
400
200
0
0
500000
1000000
1500000
2000000
2500000
3000000
Array size (m)
Merge Random
Merge Inverse
Merge Sorted
C1 M log M
C2 M log M

As demonstrated in the graphs above, the implementation of merge sort falls within the theoretical boundaries where C1 = 0.000018 and C2 =0.000006 as constants which could affect the running time of the code, such as hardware environment.

Of the three sorting algorithms, merge sort has the fastest running time. The input vector's order has very little to no bearing on how long the merge sort takes to complete. The length of the vector dimension also has no bearing on how quickly the method runs. The input size will be the main determining factor for merging sort. The algorithm's execution time grows together with the size of the input vector. However, compared to insertion sort, the increase in running time for merge sort is not as significant. This is mostly attributable to the merge sort's divide and conquer method, which makes it effective even for large input volumes.

Conclusion:

We can effectively test many sorting algorithms for various order, input, and dimension sizes and contrast each algorithm's effectiveness. We may infer from the aforementioned findings and study that the merge sort is the most effective method when the input size is big, and that the input vector's dimension and order have little to no effect on the merge sort's effectiveness. The insertion sort works well for modest input sizes, but as the input size rises, the insertion's running time increases quickly. As we compute the length of each vector within the insertion sort method, the input vector's dimension also has a significant influence on the naive insertion sort. The length of the input vector is computed beforehand before sorting the input; hence this is not the case for better insertion sort. Because of this, the input dimension has little to no effect on how long the enhanced insertion sort takes to execute. The insertion sort's processing time is also influenced by the input order. Since the sorted input vector represents the ideal situation for insertion sort, it works well for the sorted input. The worst-case situation and most time-intensive insertion sort input, however, is the inverse sorted vector input.

The behavior is consistent with the test findings, which support the insertion and merge sort's time complexity. Due to machine limitations, we are sometimes unable to calculate the insertion sort's execution time. If we use high speed computers to run the algorithms, we ought to be able to calculate these scenarios as well. The efficiency of these sorting algorithms would remain the same in general, while it may speed up certain other scenarios as well.