CS 590 Algorithms

Assignment 1

February 16 2024

Array A is a two-dimensional array of size n x m where n is the total number of vectors that need to be sorted and m is the dimension of each vector.
Algorithms:Three algorithms with different time complexities have been implemented and analyzed in this assignment.
2.1 Naive Insertion Sort:- In this , for each row of the multi-dimensional array A, the length of the corresponding vector in that row is calculated and compared to the length of vector in the previous row of A. This is a naive and inefficient version of insertion sort because the length of m-dimensional vector has to be calculated each time the n-dimensional array A is traversed which gives rise to nested for loops.

2.2 Improved Insertion Sort :- In this , for each row of the multi-dimensional array A, the length of the corresponding vector is calculated prior to sorting and saved in a new one dimensional array of length n. This new array would have the lengths of each of the vectors and based on these lengths, the original vectors will be re-arranged in sorted order.
Time complexity analysis:-Worst Case: $O(n^2)$, Best Case: O(n), Average Case: $O(n^2)$.Time complexity is same as Insertion sort, the lengths of all vectors are precomputed before sorting, which requires iterating over the array once to compute the lengths. This step has a time complexity of O(r - l).Then, the sorting process is similar to regular Insertion Sort, but instead of recalculating vector lengths within the loop, it directly uses the precomputed lengths.The insertion part of the algorithm still has a time complexity of $O((r-l)^2)$ because it performs a similar number of comparisons and swaps as the regular Insertion Sort. Overall, the time complexity for Improved Insertion Sort is $O((r-l)^2)$ as well. The algorithm performance is quadratic

2.3 Merge Sort :- In this , the lengths of each of the n vectors are calculated and stored in a 1D array and it is sorted according to the standard divide-conquer-combine procedure that is characteristic of merge sort algorithm. The correspondingrows in the 2D array are then sorted to get the final result. Worst Case: O(n log n), Best Case: O(n log n), Average Case: O(n log n)

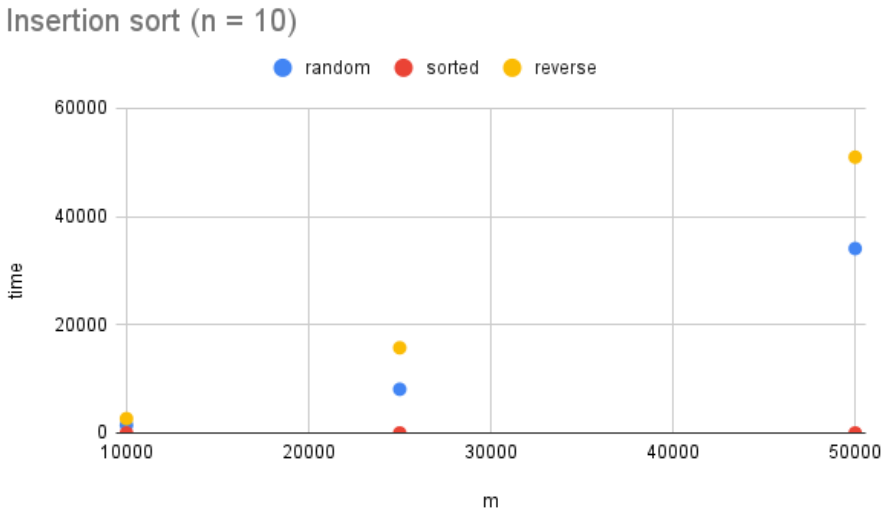

Figure 1: Insertion sort (n = 10)

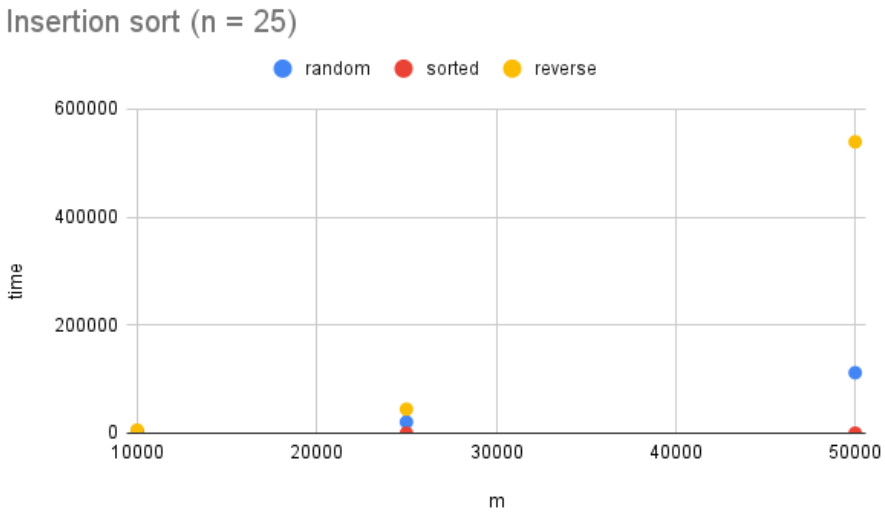

Figure 2: Insertion sort (n = 25)

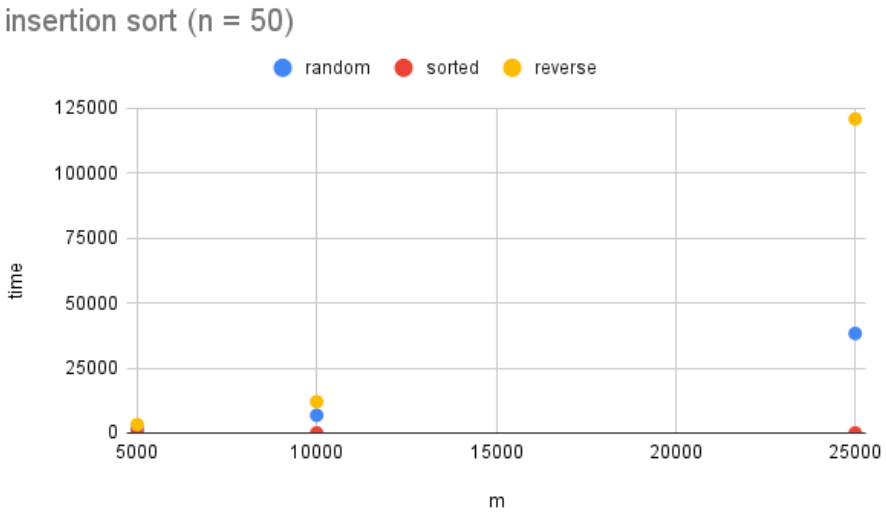

Figure 3: insertion sort (n = 50)

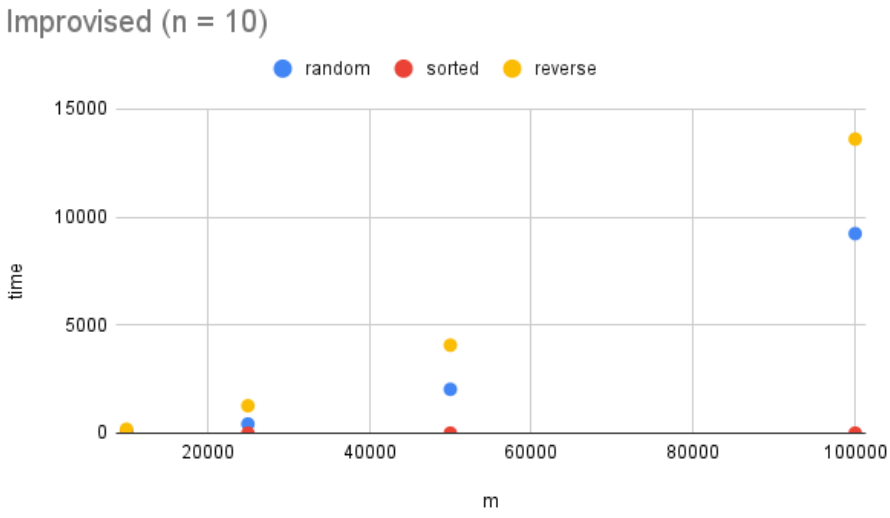

Figure 4: Improvised Insertion sort(n = 10)

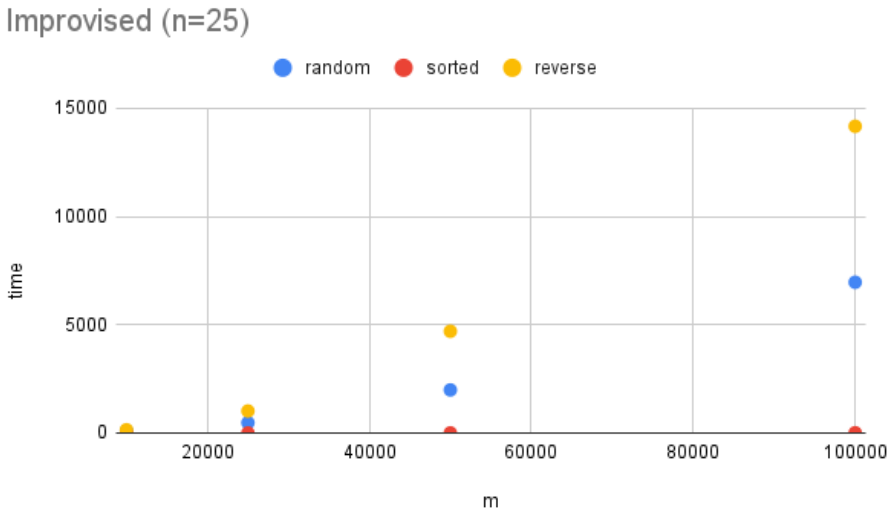

Figure 5: Improvised Insertion sort(n = 25)

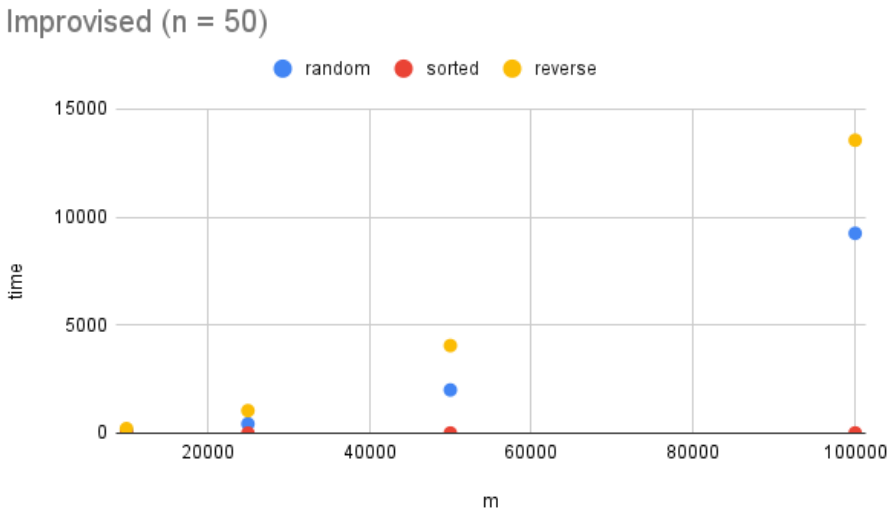

Figure 6: fig:Improvised Insertion sort(n = 50)

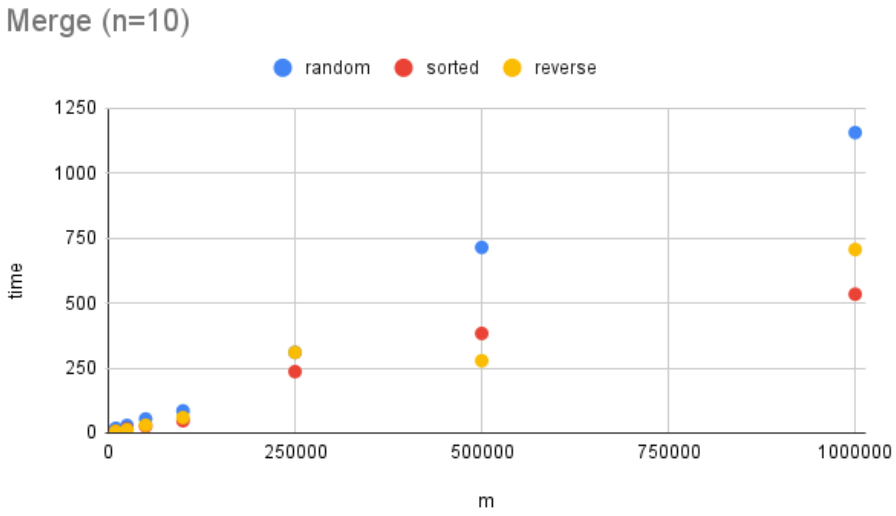

Figure 7: Merge (n=10)

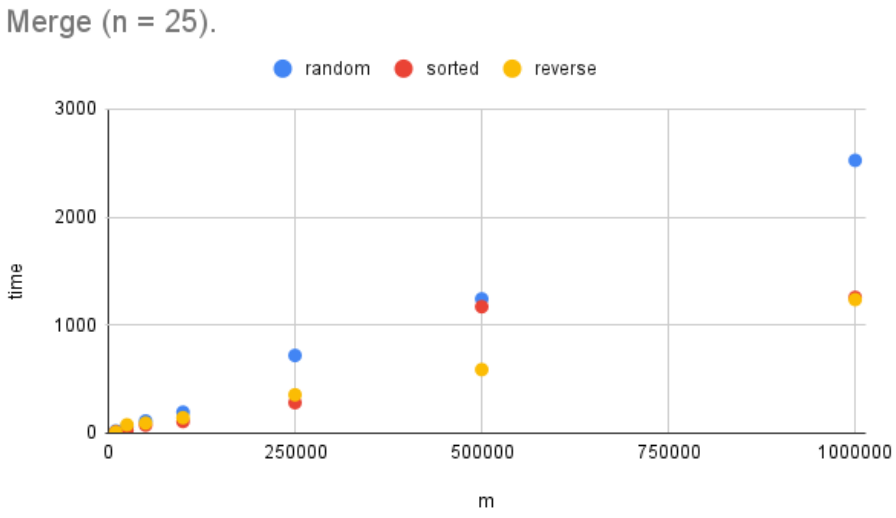

Figure 8: Merge (n = 25)

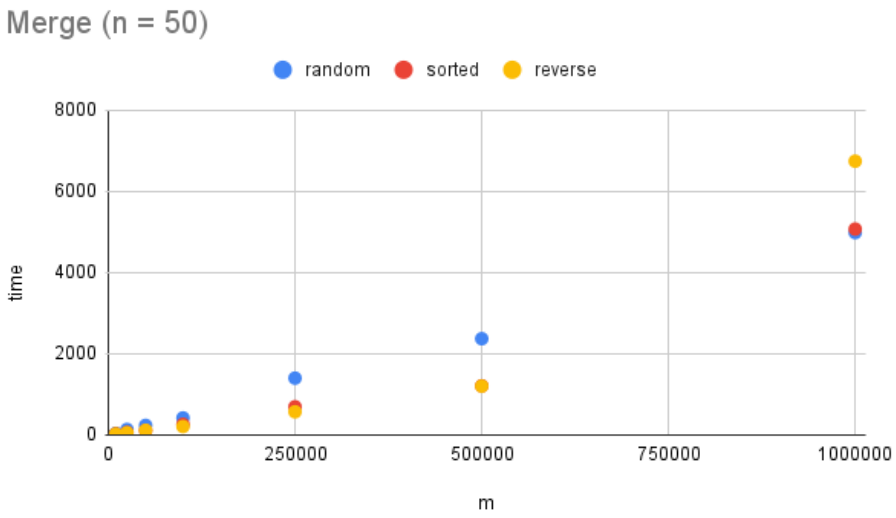

Figure 9: Merge (n = 50)

m	random	sorted	reverse
10000	1436	1	2654
25000	8076	1	15732
50000	34065	3	50951

Table 1: Insertion sort(n=10)

m	random	sorted	reverse
10000	3258	1	5693
25000	20581	10	44322
50000	111786	8	539093

Table 2: Insertion sort(n=25)

m	random	sorted	reverse
5000	1543	1	3222
10000	6805	2	12029
25000	38282	10	120822

Table 3: Insertion sort(n=50)

m	random	sorted	reverse
10000	90	1	182
25000	425	1	1268
50000	2029	3	4071
100000	9230	4	13603

Table 4: Improvised Insertion sort (n=10)

m	random	sorted	reverse
10000	95	1	147
25000	475	2	1014
50000	1988	4	4697
100000	6960	10	14163

Table 5: Improvised Insertion sort (n=25)

m	random	sorted	reverse
10000	111	2	211
25000	431	4	1042
50000	2000	8	4052
100000	9244	13	13551

Table 6: Improvised Insertion sort (n=50)

m	random	sorted	reverse
10000	18	6	6
25000	29	15	12
50000	54	26	30
100000	85	47	59
250000	310	236	310
500000	714	383	278
1000000	1156	534	706

Table 7: Merge sort (n=10)

m	random	sorted	reverse
10000	20	11	10
25000	57	27	77
50000	113	74	93
100000	194	108	143
250000	719	282	355
500000	1242	1170	588
1000000	2524	1256	1236

Table 8: Merge sort (n=25)

m	random	sorted	reverse
10000	39	26	26
25000	136	61	55
50000	236	108	116
100000	420	263	208
250000	1400	695	571
500000	2371	1210	1204
1000000	4984	5071	6743

Table 9: Merge sort (n=50)