CS590 homework 1 – C++ & running times

The due date for this assignment is Friday, September 28th, at 11.59pm. This assignment is worth 10% of your final grade.

Any sign of collaboration will result in a 0 and being reported to the Graduate Academic Integrity Board. Late submission policy described in the syllabus will be applied.

You are given an integer vector which is represented by *int** an array of integers and its dimension m as a separate parameter. We are interested in sorting arrays of integer vectors according to a pre-defined notion of vector length. You therefore are given the function *ivector_length(v, m)* that computes and returns the length of vector v with dimension m as $\sum_{i=1}^{m} |v_i|$.

You are given a naive (and very inefficient) implementation of insertion sort for arrays of integer vectors.

Questions (100 points)

1. Develop an improved implementation of insertion sort for integer vector (*insertion_sort_im*) that precomputes the length of each vector before the sorting. Keep in mind that the vectors are sorted according to their length (see *ivector_length* function). You can test the correctness of your sorting algorithm using the provided *check_sorted* function.

2. Implement a merge sort for an array of integer vectors. As for the improved insertion sort algorithm, you should precompute the length of the vectors before the sorting and the sorting is done according to the vector length. Test the correctness of your merge sort implementation using the provided *check_sorted* function.

3. Measure the runtime performance of insertion sort (naive and improved) and merge sort for random, sorted, and inverse sorted inputs of size n = 10000; 25000; 50000; 100000; 250000; 500000; 1000000; 2500000 and vector dimension m = 10; 25; 50. You can use the provided functions *create_random_ivector*, *create_sorted_ivector, create reverse_sorted_ivector.*
Repeat each test a number of times (usually at least 10 times) and compute the average running time for each combination of algorithm, input, and size n. Report and comment on your results.

Remarks:

- You might have to adjust the value for n depending on your computers speed, but allow each test to take up to a couple of minutes.
- Start with smaller values of n and m and stop if one instance of the algorithm takes more than 10 min to complete (the insertion sort implementations will hit that limit early on).
- Report and comment means that you have to analyze and interpret your findings properly. What do the experiments tell you?
- The programming, testing and the experimentation will take some time. Start early.
- Feel free to use the provided source code for your implementation. You have to document your code.

STEVENS INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

CS590: ALGORITHMS

Homework Assignment 1

Submitted by:
Hadia HAMEED
CWID: 10440803

Yijia TAN
CWID: 10427079

Submitted to:
Prof. Iraklis
TSEKOURAKIS

October 1, 2018

1 Problem Statement:

We are interested in sorting arrays of integer vectors according to a pre-defined notion of vector length i.e.

$$\sum_{i=1}^{m} |v_i|$$

.

Array $\boldsymbol{A}$ 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.

2 Algorithms:

Three algorithms with different time complexities have been implemented and analyzed in this work.

2.1 Naive Insertion Sort (nIS)

In this technique, 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 Efficient Insertion Sort (eIS)

In this technique, 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. The time complexity of this technique is $O(n^2)$

2.3 Merge Sort (mS)

In this technique, 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 corresponding rows in the 2D array are then sorted to get the final result. The time complexity of this technique is $O(nlgn)$

3 Results:

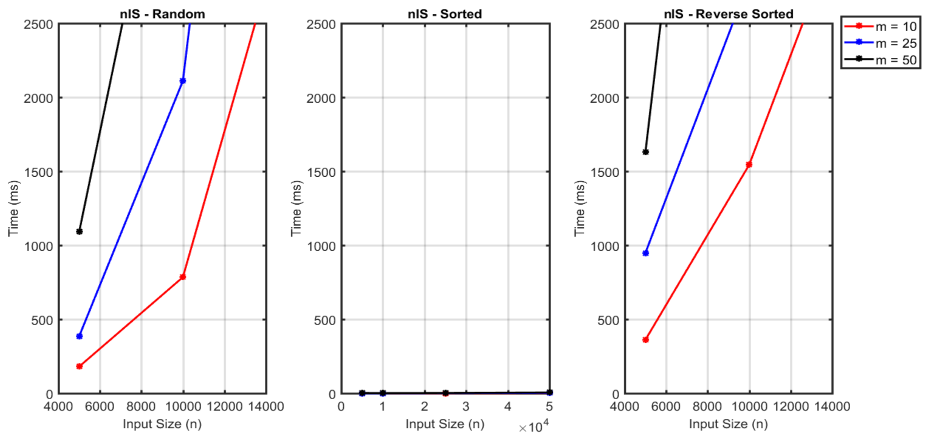

Figure 1: Performance for Naive Insertion Sort.

Results:

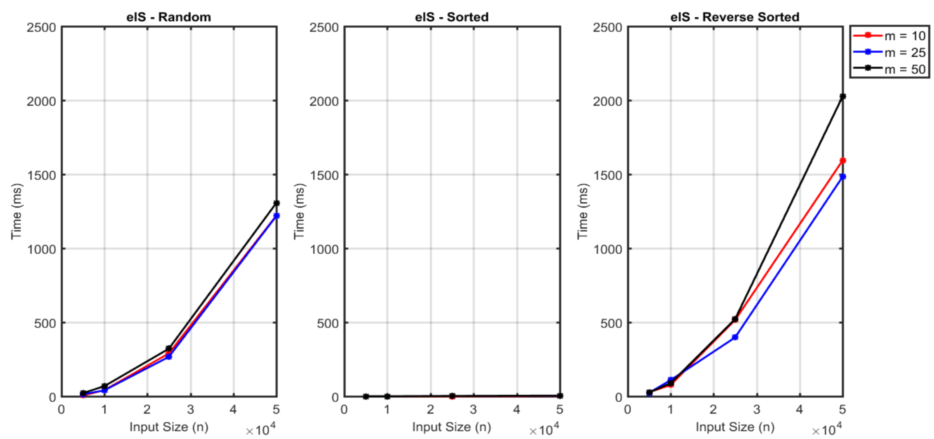

Figure 2: Performance for Efficient Insertion Sort.

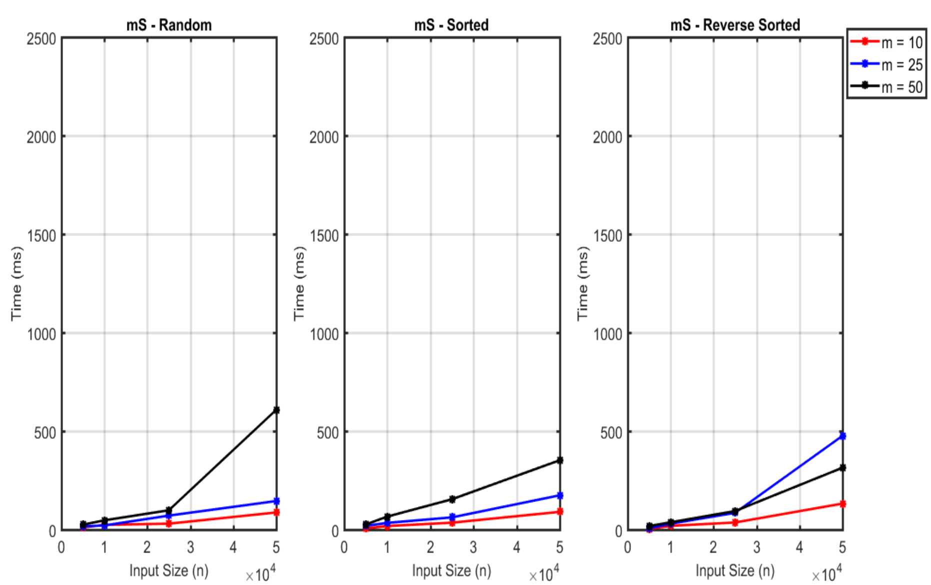

Figure 3: Performance for Merge Sort

3.1 Naive Insertion Sort (nIS):

Naive Insertion Sort (nIS) for randomly sorted array (d = 0)		
Dimension of each vector (m)	No. of vectors (n)	Time (ms)
10	5000 10000 25000 50000	180 784 8162 41696
25	5000 10000 25000 50000	385 2110 19515 70314
50	5000 10000 25000 50000	1090 4466 30043 169899

Naive Insertion Sort (nIS) for reverse sorted array (d = -1)		
Dimension of each vector (m)	No. of vectors (n)	Time (ms)
10	5000 10000 25000 50000	361 1543 7092 25450
25	5000 10000 25000 50000	948 2790 25229 89015
50	5000 10000 25000 50000	1628 7574 35768 137303

Results:

Naive Insertion Sort (nIS) for sorted array (d = 1)		
Dimension of each vector (m)	No. of vectors (n)	Time (ms)
10	5000	0
	10000	0
	25000	0
	50000	1
25	5000	0
	10000	0
	25000	1
	50000	2
50	5000	2
	10000	1
	25000	2
	50000	5

3.2 Efficient Insertion Sort (eIS):

Efficient Insertion Sort (eIS) for randomly sorted array (d = 0)		
Dimension of each vector (m)	No. of vectors (n)	Time (ms)
10	5000 10000 25000 50000	5 41 289 1219
25	5000 10000 25000 50000	13 40 266 1217
50	5000 10000 25000 50000	21 68 321 1305

Efficient Insertion Sort (eIS) for reverse sorted array (d = -1)		
Dimension of each vector (m)	No. of vectors (n)	Time (ms)
10	5000 10000 25000 50000	27 76 515 1592
25	5000 10000 25000 50000	23 109 397 1482
50	5000 10000 25000 50000	25 88 521 2024

Efficient Insertion Sort (eIS) for sorted array (d = 1)		
Dimension of each vector (m)	No. of vectors (n)	Time (ms)
10	5000	0
	10000	0
	25000	0
	50000	1
25	5000	0
	10000	0
	25000	2
	50000	3
50	5000	0
	10000	0
	25000	2
	50000	4

3.3 Merge Sort (mS):

Merge Sort (mS) for randomly sorted array (d = 0)		
Dimension of each vector (m)	No. of vectors (n)	Time (ms)
10	5000	11
	10000	24
	25000	30
	50000	87
25	5000	16
	10000	21
	25000	71
	50000	145
50	5000	25
	10000	47
	25000	98
	50000	608

Merge Sort (mS) for reverse sorted array (d = -1)		
Dimension of each vector (m)	No. of vectors (n)	Time (ms)
10	5000	4
	10000	19
	25000	36
	50000	132
25	5000	7
	10000	30
	25000	85
	50000	476
50	5000	18
	10000	37
	25000	94
	50000	314

Merge Sort (mS) for sorted array (d = 1)		
Dimension of each vector (m)	No. of vectors (n)	Time (ms)
10	5000	6
	10000	18
	25000	35
	50000	90
25	5000	19
	10000	34
	25000	62
	50000	174
50	5000	26
	10000	66
	25000	154
	50000	352

4 Conclusion:

4.1 Worst Case (Reverse sorted array, $d = -1$)

The results show that when the array is reverse sorted, the fastest algorithm is mS. nIS is more than fifty times slower than eIS and four hundred times slower than mS for an array of size 50000x50. Whereas, eIS is more than five times slower than mIS. This is because there are no nested for loops in mS as compared to eIS which has a conditional while loop within a for loop which gives a time complexity of the order of n^2 in worst case while the time complexity for mS is $O(nlgn)$.

4.2 Best Case (Sorted array, $d = 1$)

The time to sort the array is almost negligible for both eIS nd nIS in case of an already sorted array. This is because the condition for while loop is never satisfied and time complexity remains $O(n)$. There isn't much difference in the performance of mS since there are no best or worst cases for this because the array has to be divided-conquered-combined in any case which is $O(nlgn)$.

4.3 Average Case (Randomly sorted array, $d = 0$)

In this case, the performance of eIS and nIS continues to be slower than mS since on average the time complexity for insertion sort is still $O(n^2)$ and is as bad as the worst case. nIS is more than a hundred times slower than eIS and more than two hundred times slower than mS for an an array of size 50000x50. Whereas, eIS is half as fast as mS. The time complexity for mS continues to be $O(nlgn)$.

The time complexity between insertion sort and merge sort differs greatly depending on the input array. The best way to choose algorithm wisely is using insertion sort for well sorted arrays, and merge sort for reversed arrays, if related information about input data is known. In most circumstances, when input information is unknown or random, merge sort is a much better choice, because $O(nlogn)$ grows much slower than $O(n^2)$. As we can see in the graphs, when n approaches to 5000 and m equals to 10, the merge time is around tens times faster. When n is 50000 and m gets to 50, merge sort is about twice the speed. The difference will be more obvious as the input size grows.

