# STEVENS INSTITUTE OF TECHNOLOGY

## Computer Science (Master’s)

CS590A - ALGORITHMS

ASSIGNMENT 1

### Submission to: **Professor In Suk Jang**

### Submission By: **Chetan Goyal CWID: 20005334**

Date: October 5, 2021

# OBJECTIVES

### 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, n) that computes and returns the length of vector v with dimension n as-



### An improved implementation of insertion sort for integer vector (insertion\_sort\_im) that pre-computes the length of each vector before the sorting. To precompute the length of vectors before sorting, and to sort as per the vector lengths.

### Implementation of merge sort for the array of integer vectors. To precompute the length of vectors before sorting, and to sort as per the vector lengths.

### Measure the runtime performance of insertion sort (naive and improved) and merge sort for random, sorted, and inverse sorted inputs of size m = 1000; 2500; 5000; 10000; 25000; 50000; 100000; 250000 and vector dimension n = 10; 25; 50.

## Improved Insertion Sort-

The improved insertion sort implemented in the code first of all computes the length of every vector in the 2 dimensional matrix, and it then stores these values in a new array before it begins to sort them through the implemented and improved insertion sort algorithm.

The insertion sort implemented here is better in two ways:

1. The first reason, which has already been mentioned above, is that it does not call the function to calculate the length of arrays that are being compared repeatedly with each iteration. But instead it stores all the lengths in a new array. This increases the size complexity of the program by O(m+n) but is highly efficient because now the calculation of length is only being done by O(m+n) time rather than in O(m\*n).
2. The second reason is that in the implementation of insertion sort, a binary search algorithm has been used to put all the search and put the elements at the correct places rather than linear approach. Therefore, initially it does O(n) comparisons, but with binary search the comparisons will reduce to O(logn). Thereby improving the efficiency slightly. However the worst case run time complexity here will stay O(n^2).

Merge Sort-

The merge sort algorithm is based on divide and conquer. We divide the vector from the middle element and keep on dividing until it reduces to single elements. Then we start to merge these subarrays and sort them as we do so. This method is a highly efficient method when compared to Insertion Sort as with this method our worst case time complexity reduces to O(nlogn). However it is considered a better approach only when the number of elements that we need to sort is extremely large.

Results in tabular format:-

## Improved Insertion Sort

### Random Vector:

M = 10

| N | Running Time |
| --- | --- |
| 1000 | 0.05193495750427246 |
| 2500 | 0.3763091564178467 |
| 5000 | 1.5938420295715332 |
| 10000 | 5.704725980758667 |
| 25000 | 45.033530950546265 |

M = 25

| N | Running Time (s) |
| --- | --- |
| 1000 | 0.059792280197143555 |
| 2500 | 0.3713700771331787 |
| 5000 | 1.4673199653625488 |
| 10000 | 5.929949998855591 |
| 25000 | 41.49009323120117 |

M = 50

| N | Running Time (s) |
| --- | --- |
| 1000 | 0.06480836868286133 |
| 2500 | 0.38515281677246094 |
| 5000 | 1.4560368061065674 |
| 10000 | 5.512821912765503 |
| 25000 | 39.791186809539795 |

### Sorted Vector:

M = 10

| N | Run Time (s) |
| --- | --- |
| 1000 | 0.003203153610229492 |
| 2500 | 0.010191917419433594 |
| 5000 | 0.022830724716186523 |
| 10000 | 0.04741096496582031 |
| 25000 | 0.16425681114196777 |
| 50000 | 0.24203896522521973 |
| 1000000 | 0.5100531578063965 |

| 250000 | 1.5658049583435059 |
| --- | --- |

M = 25

| N | Running Time (s) |
| --- | --- |
| 1000 | 0.005416154861450195 |
| 2500 | 0.013036012649536133 |
| 5000 | 0.02727198600769043 |
| 10000 | 0.05148482322692871 |
| 25000 | 0.13799786567687988 |
| 50000 | 0.28099799156188965 |
| 1000000 | 0.5905258655548096 |
| 2500000 | 1.5538029670715332 |

M = 50

| N | Running Time (s) |
| --- | --- |
| 1000 | 0.007026195526123047 |
| 2500 | 0.019675254821777344 |
| 5000 | 0.033274173736572266 |
| 10000 | 0.06849789619445801 |
| 25000 | 0.18483281135559082 |
| 50000 | 0.4501979351043701 |
| 1000000 | 0.7595870494842529 |
| 2500000 | 2.1211819648742676 |

### Reverse sorted Vector:

M = 10

| N | Running Time |
| --- | --- |
| 1000 | 0.10303997993469238 |
| 2500 | 0.6385641098022461 |
| 5000 | 2.546705961227417 |
| 10000 | 10.891072034835815 |

| 25000 | 70.49958682060242 |
| --- | --- |

M = 25

| N | Running Time |
| --- | --- |
| 1000 | 0.10982084274291992 |
| 2500 | 0.6707158088684082 |
| 5000 | 2.5740890502929688 |
| 10000 | 10.812655687332153 |

M = 50

| N | Running Time |
| --- | --- |
| 1000 | 0.11014604568481445 |
| 2500 | 0.6435689926147461 |
| 5000 | 2.8986928462982178 |
| 10000 | 11.438220024108887 |

## Merge Sort

### Random Vector:

M = 10

| N | Running Time (s) |
| --- | --- |
| 1000 | 0.0051729679107666016 |
| 2500 | 0.01635289192199707 |
| 5000 | 0.03643989562988281 |
| 10000 | 0.06676387786865234 |
| 25000 | 0.1770031452178955 |
| 50000 | 0.4441380500793457 |
| 1000000 | 1.0812780857086182 |
| 2500000 | 2.3853180408477783 |

M= 25

| N | Running Time (s) |
| --- | --- |
| 1000 | 0.00814509391784668 |
| 2500 | 0.020645856857299805 |
| 5000 | 0.04218316078186035 |
| 10000 | 0.08563876152038574 |
| 25000 | 0.21500015258789062 |
| 50000 | 0.44162678718566895 |
| 1000000 | 0.9403500556945801 |
| 2500000 | 2.554386854171753 |

M = 50

| N | Running Time (s) |
| --- | --- |
| 1000 | 0.011038064956665039 |
| 2500 | 0.023808717727661133 |
| 5000 | 0.04840278625488281 |
| 10000 | 0.11404180526733398 |
| 25000 | 0.33629894256591797 |
| 50000 | 0.6478557586669922 |
| 1000000 | 1.2149193286895752 |
| 2500000 | 3.474487066268921 |

### Sorted Vector:

M = 10

| N | Running Time (s) |
| --- | --- |
| 1000 | 0.005031108856201172 |
| 2500 | 0.014290809631347656 |
| 5000 | 0.027532100677490234 |
| 10000 | 0.05327177047729492 |
| 25000 | 0.21006011962890625 |
| 50000 | 0.4012751579284668 |
| 1000000 | 0.6848688125610352 |
| 2500000 | 1.846938133239746 |

M = 25

| N | Running Time (s) |
| --- | --- |
| 1000 | 0.00563502311706543 |
| 2500 | 0.015895605087280273 |
| 5000 | 0.032900094985961914 |
| 10000 | 0.06811785697937012 |
| 25000 | 0.17132806777954102 |
| 50000 | 0.42425107955932617 |
| 1000000 | 0.7718830108642578 |
| 2500000 | 2.1903841495513916 |

M = 50

| N | Running Time (s) |
| --- | --- |
| 1000 | 0.007799863815307617 |
| 2500 | 0.020740032196044922 |
| 5000 | 0.044361114501953125 |
| 10000 | 0.08614897727966309 |
| 25000 | 0.26850104331970215 |
| 50000 | 0.4388132095336914 |
| 1000000 | 0.9692678451538086 |
| 2500000 | 2.44084095954895 |

### Reverse Sorted Vector:

M= 10

| N | Running Time (s) |
| --- | --- |
| 1000 | 0.00438690185546875 |
| 2500 | 0.014435052871704102 |
| 5000 | 0.028276920318603516 |
| 10000 | 0.059342145919799805 |
| 25000 | 0.15201592445373535 |
| 50000 | 0.3405649662017822 |
| 1000000 | 0.6895740032196045 |
| 2500000 | 1.974276065826416 |

M = 25

| N | Running Time (s) |
| --- | --- |
| 1000 | 0.00748896598815918 |
| 2500 | 0.01559591293334961 |
| 5000 | 0.03473615646362305 |
| 10000 | 0.0714879035949707 |
| 25000 | 0.17107725143432617 |
| 50000 | 0.33948469161987305 |
| 1000000 | 0.7545268535614014 |
| 2500000 | 2.007699966430664 |

M = 50

| N | Running Time (s) |
| --- | --- |
| 1000 | 0.007884979248046875 |
| 2500 | 0.02063894271850586 |
| 5000 | 0.042836904525756836 |
| 10000 | 0.08681702613830566 |
| 25000 | 0.2604689598083496 |
| 50000 | 0.4763147830963135 |
| 1000000 | 1.0249030590057373 |
| 2500000 | 2.5872292518615723 |

## Naive Insertion Sort

Random Vector:

M = 10

| N | Running Time (s) |
| --- | --- |
| 1000 | 0.5966799259185791 |
| 2500 | 3.5651299953460693 |
| 5000 | 14.964791059494019 |
| 10000 | 63.42429494857788 |

M = 25

| N | Running Time (s) |
| --- | --- |
| 1000 | 1.3159139156341553 |
| 2500 | 8.49966287612915 |
| 5000 | 31.231597185134888 |

M = 50

| N | Running Time (s) |
| --- | --- |
| 1000 | 2.4049301147460938 |
| 2500 | 15.206486701965332 |
| 5000 | 60.915148973464966 |

Sorted Vector:

M = 10

| N | Running Time (s) |
| --- | --- |
| 1000 | 0.001712799072265625 |
| 2500 | 0.005816221237182617 |
| 5000 | 0.010331153869628906 |
| 10000 | 0.022424936294555664 |
| 25000 | 0.049169063568115234 |
| 50000 | 0.12742400169372559 |
| 1000000 | 0.17562222480773926 |
| 2500000 | 0.4734470844268799 |

M = 25

| N | Running Time (s) |
| --- | --- |
| 1000 | 0.004749774932861328 |
| 2500 | 0.010516881942749023 |
| 5000 | 0.018542051315307617 |
| 10000 | 0.036093950271606445 |
| 25000 | 0.1329662799835205 |
| 50000 | 0.37932419776916504 |
| 1000000 | 0.36063480377197266 |
| 2500000 | 0.9327819347381592 |

M = 50

| N | Running Time (s) |
| --- | --- |
| 1000 | 0.008205890655517578 |
| 2500 | 0.02132391929626465 |
| 5000 | 0.03627276420593262 |
| 10000 | 0.07383489608764648 |
| 25000 | 0.18681812286376953 |
| 50000 | 0.3584909439086914 |
| 1000000 | 0.7238218784332275 |
| 2500000 | 1.845952033996582 |

Reverse sorted Vector:

M = 10

| N | Running Time (s) |
| --- | --- |
| 1000 | 0.8866279125213623 |
| 2500 | 5.797590970993042 |
| 5000 | 25.256773710250854 |

M = 25

| N | Running Time (s) |
| --- | --- |
| 1000 | 1.963404893875122 |
| 2500 | 12.345251083374023 |
| 5000 | 47.70496487617493 |

M = 50

| N | Running Time (s) |
| --- | --- |
| 1000 | 4.116827964782715 |
| 2500 | 25.740787029266357 |

# **Conclusion**

From the above tables we see that for each sorting algorithm the worst case occurs when the reverse sorted array is taken and the best time occurs when the sorted array for vectors is taken.

The naive insertion sort takes the largest of times, with improved insertion sort having far better time than naive insertion sort and the merge sort being the best of all the times.

Merge sort takes O(nlogn) time and we can see that we were actually able to run all the dimensions and get time whereas for others we had to stope earlier as the dimensions increased.

This agrees with what we have learnt in theory. O(n^2) is for all the insertion sort whereas O(nlogn) for Merge sort with naive insertion performing the most poorly.

NOTE- The code has been written in Python