Stevens Institute of Technology

Department of Computer Science

CS590: ALGORITHMS

Prof. Iraklis Tsekourakis

Homework Assignment 1

Submitted by: **Malav Shastri**

CWID**: 10456244**

Introduction:

Given problem is all about sorting an array of vectors. The sorting criteria is to sort according to vector length. Vector Length is defined as the sum of all vector elements. So an array element which has smaller vector length (Sum of vector components) should come first in that array. As the sum increases the length increases and so is the place (Index) of that element in array increases. The concept of array of vectors has been implemented by using multidimensional array, and so there is n – dimensional array. Length of array is m and dimension of array is n.

In the skeleton code there is already a function insertion_sort has been given which is a naïve method, where vector length is being computed alongside the sorting process. In the solution we were expected to make an improved insertion sort algorithm with less time complexity by pre-computing the vector length. We were also expected to implement a merge_sort algorithm which runs on the same ideology of pre-computing vector length as in improved insertion sort algorithm.

Functions used:

- **insertion_sort():** Usually insertion sort in average and worst case take $O(n^2)$. But here in the case of sorting according to vector length the author has called the vector length computation function inside the while loop where we compare the ith and (i-1)th values for sorting. This structure makes the function to run 3 for loops, all 3 nested. The reason behind that is the ivector_length() function to calculate the vector length traverse over n (Array dimension) in order to sum all the values of vector elements. Thus this structure takes its complexity to $O(n^3)$.
- **Insertion_sort_im():** This is the improved version of insertion_sort() function. Here pre-computed vector length has been stored in an array and a pointer to that array has been passed in the while condition while comparing the vector length values of two side by side elements. As we pre compute the vector array length now the while loop is not containing any for loop inside condition and traversal on n is being done separately which boils down the complexity to $O(n^2)$.

- **merge_sort():** To apply merge sort and sort according to vector length the program pre-computes the vector length in a function called merge_sort() it's a dummy function to call main merge sort logic as we also want to pass the pre computed vector lengths. The vector lengths then uses the divide and concur logic to sort according to it. Complexity is O(nlogn).

Below is the time taken by each cases for different values of parameters m, n and d.

Naïve Insertion Sort				
		d=0(Random)	**d=1**(Sorted)	**d= -1**(Reverse Sorted)
N	**M**	**Real Time(ms)**		
10	5000	1105	1	2210
	10000	4244	2	8740
	25000	27300	5	55671
	50000	111597	9	222500
	100000	478218	18	>10 mins
	250000	>10 mins	45	>10 mins
	500000	>10 mins	89	>10 mins
	1000000	>10 mins	178	>10 mins
25	5000	2473	2	5018
	10000	11563	4	19723
	25000	65665	9	126424
	50000	318300	19	508490
	100000	>10 mins	37	>10 mins
	250000	>10 mins	110	>10 mins
	500000	>10 mins	188	>10 mins
	1000000	>10 mins	380	>10 mins
50	5000	5508	4	9148
	10000	22470	7	40461
	25000	139280	18	254129
	50000	686535	38	>10 mins
	100000	>10 mins	69	>10 mins
	250000	>10 mins	174	>10 mins
	500000	>10 mins	354	>10 mins
	1000000	>10 mins	692	>10 mins

Improved Insertion Sort				
		d=0(Random)	**d=1**(Sorted)	**d= -1**(Reverse Sorted)
N	**M**	**Time(ms)**	**Time(ms)**	**Time(ms)**
10	5000	44	1	88
	10000	203	1	335
	25000	1095	2	2212
	50000	4047	5	8249
	100000	16437	10	34021
	250000	104323	22	173472
	500000	441819	50	>10 mins
	1000000	>10 mins	104	>10 mins
25	5000	45	1	91
	10000	172	2	360
	25000	1136	6	2538
	50000	4187	11	8800
	100000	16796	22	36267
	250000	108274	51	231520
	500000	>10 mins	103	>10 mins
	1000000	>10 mins	204	>10 mins
50	5000	47	2	92
	10000	184	4	360
	25000	1114	9	2232
	50000	4580	20	9023
	100000	21272	36	36850
	250000	118727	94	234371
	500000	>10 mins	180	>10 mins
	1000000	>10 mins	375	>10 mins

Merge Sort				
		d=0(Random)	**d=1**(Sorted)	**d= -1**(Reverse Sorted)
N	**M**	**Time(ms)**	**Time(ms)**	**Time(ms)**
10	5000	4	4	4
	10000	9	8	8
	25000	23	24	24
	50000	50	46	47
	100000	94	96	94
	250000	249	238	225
	500000	557	452	507
	1000000	925	945	952
25	5000	5	5	5
	10000	10	9	9
	25000	26	23	25
	50000	50	48	51
	100000	112	100	102
	250000	275	215	235
	500000	568	515	558
	1000000	1145	1050	1073
50	5000	5	6	6
	10000	11	11	11
	25000	30	29	30
	50000	63	66	69
	100000	122	123	125
	250000	314	300	305
	500000	618	590	596
	1000000	1285	1198	1221

Conclusion:

From the results we can see that in the worst case merge sort is fastest among all of 3. In worst case merge sort has the complexity of O(nlogn) while improved insertion sort and naïve insertion sort has the respective complexities of $O(n^2)$ and $O(n^3)$. Talking about the relative speed improved is 15 times slower for (50000x50) than merge and naïve is more than 30 times slower than improved. As the input value grows the time differences increases. And we can see the difference more clearly for the larger input values.

Talking about the average case d=0 the complexity of merge sort is same as O(nlogn). And this also reflects in the result table. You can see that times for d =1 , 0 and -1 are almost same. In case of improved insertion the time complexity is same as $O(n^2)$ and its continues to be slower than merge sort same is the case with naïve insertion sort, it has the complexity of $O(n^3)$. With the larger input values we can see the difference where naïve and for some inputs improved takes more than 10 mins but merge sort easily do it in negligible time.

For best case d=1 the naïve insertion sort works best. As it never enters the while loop and complexity sticks to O(n). So is the case with improved one as it also has the complexity of O(n). Though merge sort also performs well but not as well as insertion sort because regardless of input it always divides and merge.

According to input size and how sorted the data already is these algorithms can be used. But overall merge sort performs pretty well in all the cases.