Anil Bhusal

Csc 22000

7 April 2021

Programming Assignment – 1

N	Insertion Sort	Merge Sort	Heap Sort	Quick Sort	Quick Sort (Random)	Radix Sort
10	6525	22133	27117	23721	41685	154780
100	593569	167236	263462	69320	164708	161082
1000	3932195	3148758	2280697	862353	11756797	1237439
10000	23250304	11126043	6536487	5879600	14235342	8167121
100000	1356359589	40403009	31620933	18466316	37561241	23257431
1000000	Na	224374856	215816253	106120219	154775016	128613752

Table 1: Running time (nano second) of different algorithm.

In this assignment, we are required to sort the array of different sizes and analyze their running time. To do so, I use the Java programming language. The sorting algorithms that I coded in the program are Insertion sort, Merge Sort, Heap Sort, Quick Sort, Quick Sort with random pivot, and Radix Sort. While creating the array of different sizes, I asked the user to input the array size. For my program, I passed different sizes 10, 100, 1000, 10000, 100000, and 1000000, which will contain the random integers between 1 to 1000000. The running time of each algorithm is in nanosecond with different sizes which is shown in table 1 above. The time represented above only includes the amount of time taken to sort the array. Total execution time will differ from the above time because the array will take some seconds to fill with random

numbers and run the whole program. Every time we run our program, we will get different compile-time because it depends on how best or worst set of numbers are generated by the random number generator. Also, it differs by algorithms. Let's discuss them individually.

a) Insertion sort: It is a brute force sorting algorithm whose average and worse time complexity is $\Theta(n^2)$. The best time complexity is $\Theta(n)$. It is easy to code, but an increase in the size of the array will increase the running time of the insertion sort. While dealing with a very large size array, the program is not efficient. For this program, while I pass 1000000 size array with random numbers, I didn't get compiler time. So, it is good to sort for the small size array.

b) Merge Sort: It is a divide and conquers algorithm whose average, worst, and best time complexity is $\Theta(n \lg n)$. But it does not sort in place. Merge sort time complexity is better than Insertion sort, but if we compare for smaller size array, we can see in the above table that insertion sort time is shorter. But an increase in the size of the array, the merge sort is better and will sort in a short time than the insertion sort, which is also shown in the above table.

c) Heap Sort: It is a transfer and conquers sorting algorithm, whose average, worst, and best time complexity is $O(n \lg n)$. It sorts in place, but not stable. It is slightly slower than Merge sort. But if we compare with insertion sort, Heap is faster for larger size values. And insertion sort is faster for small size array even its time complexity is greater than that of heapsort.

d) Quick Sort: It is also a divide and conquers algorithm whose average and best time complexity is θ(n lg n). In the worst case, quick sort time complexity is θ(n^2). In this sort, we partition the array into two subarrays and sort them. We chose the last element as a pivot and keep smaller values to the left of it and higher to the right. Merge step is not required in this algorithm. It is the fastest algorithm among the algorithms that we tested in our program. Also, we can see in the above table 1 that quicksort has taken the shortest time to sort the array of each size except for size 10 in which insertion sort has sorted fast. Thus, quicksort is fasted sorting algorithm among all sorting algorithms that we have mentioned here.

e) Quick Sort with random pivot: It is the same algorithm, but we choose pivot randomly for the partition of the array. The time complexity is also the same for it. The only reason we do with random pivot is to solve the weakness of quick sort, that if the array is already sorted.

f) Radix Sort: It is a non-comparative sorting algorithm. The time complexity of the radix sort is θ(dN + dK). where N is the size of the array, k is the range of the digits, and d is the number of digits in a given element. If we compare radix sort with insertion sort, radix sort is suitable for a higher size array. Insertion sort is faster for smaller size arrays. Comparing with merge and heap sort, it is slower for a smaller size, but it gets comparatively faster for a larger size array. But it is slower than quicksort.