Bangladesh University of Engineering & Technology(BUET)

CSE 204: Data Structures and Algorithms Sessional

Offline 7 Report

Topic:

Sorting Algorithms

Submitted By:

Name: Syed Jarullah Hisham

Roll: 1805004

Section: A1

CSE '18

Date Of Submission: 11-06-2021

Complexity Analysis

Merge Sort Analysis:

Merge sort uses divide and conquer approach to sort any input array. Whether the input array is already sorted, or reversely sorted or in random order, this algorithm always has a complexity of O(n*logn). We can illustrate this through the scheme described below:

As we use divide and conquer, each time we split the array into two subarray of equal length which is exactly what we do at binary search. So, such split requires at most 1+ logn steps.

Each time finding mid of the array requires O(1) time.

Lastly, merge operation requires O(n) time.

Finally, the whole mergesort requires n*(logn+1) operations. So, the time complexity of the merge sort is O(n*logn). One important thing to note here that these 3 steps are not dependent on the order of the elements of the array. Ascending, descending or random order array all will be treated equally by this algorithm. So, for all orders, time complexity is O(n* logn)

Ascending Order Numbers: O(n* logn)

Descending Order Numbers: O(n* logn)

Random Order Numbers: O(n* logn)

Quick Sort Analysis:

In quick sort, we use partitioning such that partitioned element will split the input array into two parts where all the smaller elements of the partitioned element will be in one subarray and the other will be in another subarray. So, when the array is already sorted, taking the last item as partitioning pivot will make an unbalanced situation as one subarray will be empty and another will be same as the original array. First time partition method is called, there will be n swap operations. Next time when we will call

partition, it will do n-1 swap operations. So, taking a constant k, the whole quick sort will do operations in total –

Kn + k(n-1) + k(n-2) +…….. ≈ kn^2

So, in the worst case that means when the array is already in ascending order, it will require almost quadratic time to do the sort. Again, in descending order too, the above mentioned scenario occurs. So, the time complexity for both ascending and descending order is O(n^2). But for random order, on average, it is considered that the partition pivot will approximately evenly divide the array into 2 subarrays. In this case, the time complexity is O(n*logn)

Ascending Order Numbers: O(n^2).

Descending Order Numbers: O(n^2).

Random Order Numbers: O(n*logn)

Table showing the time complexities

Input Order	Merge	Quick
Ascending	O(n* logn)	O(n^2)
Descending	O(n* logn)	O(n^2)
Random	O(n* logn)	O(n*logn)

Machine Configuration

Processor - Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz 2.21 GHz

Installed RAM - 8.00 GB (7.85 GB usable)

System Type - 64-bit operating system, x64-based processor

Operating System - Windows

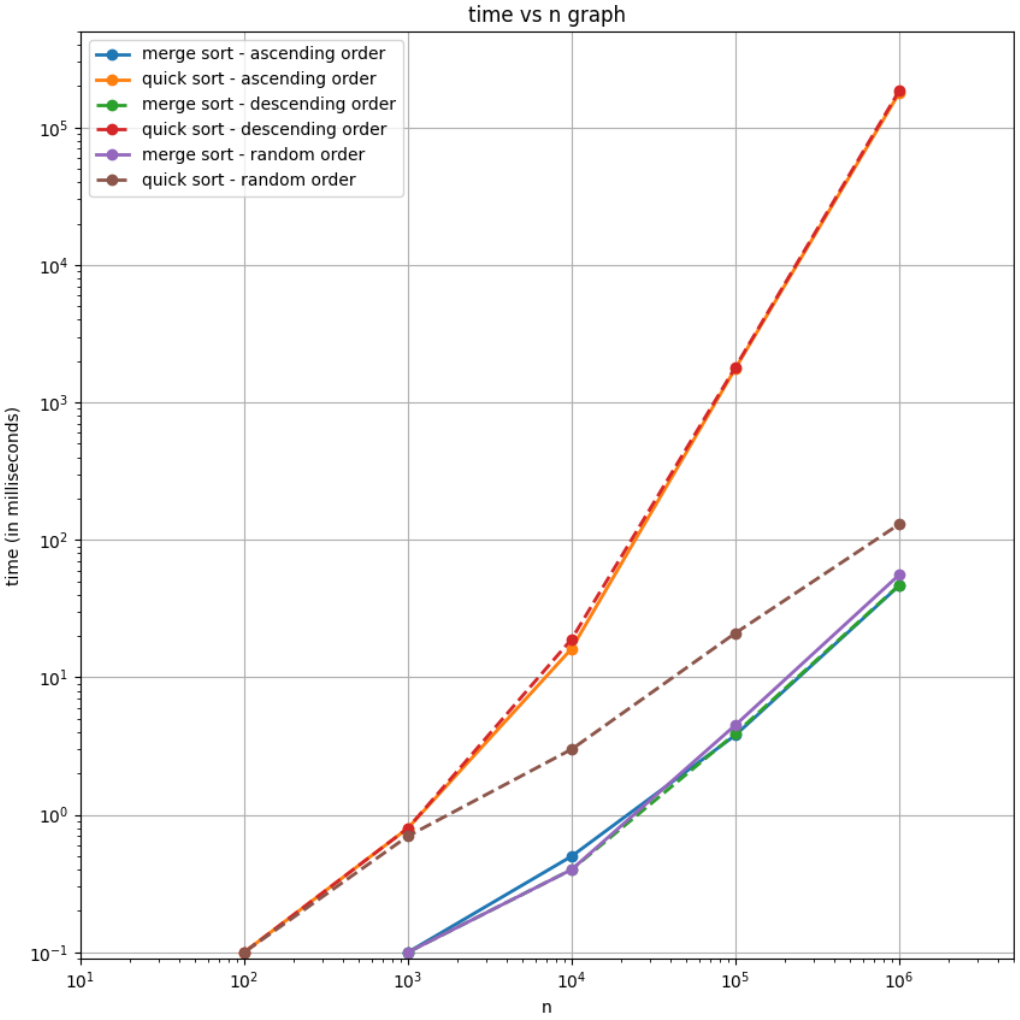
Windows Edition - Windows 10 Home Single Language

Table: Average time for sorting n integers in different input orders

Input Order	n = / Sorting Algorithm	10	100	1000	10000	100000	1000000
Ascending	Merge	0	0.0	0.1	0.5	3.8	46.5
	Quick	0	0.1	0.8	16.2	1770.1	178249.4
Descending	Merge	0	0	0.1	0.4	3.9	46.9
	Quick	0	0.1	0.8	19	1809.9	184963.4
Random	Merge	0	0	0.1	0.4	4.5	55.7
	Quick	0	0.1	0.7	3	21	130

Times in milliseconds and 10 iterations per test

Plot Graph



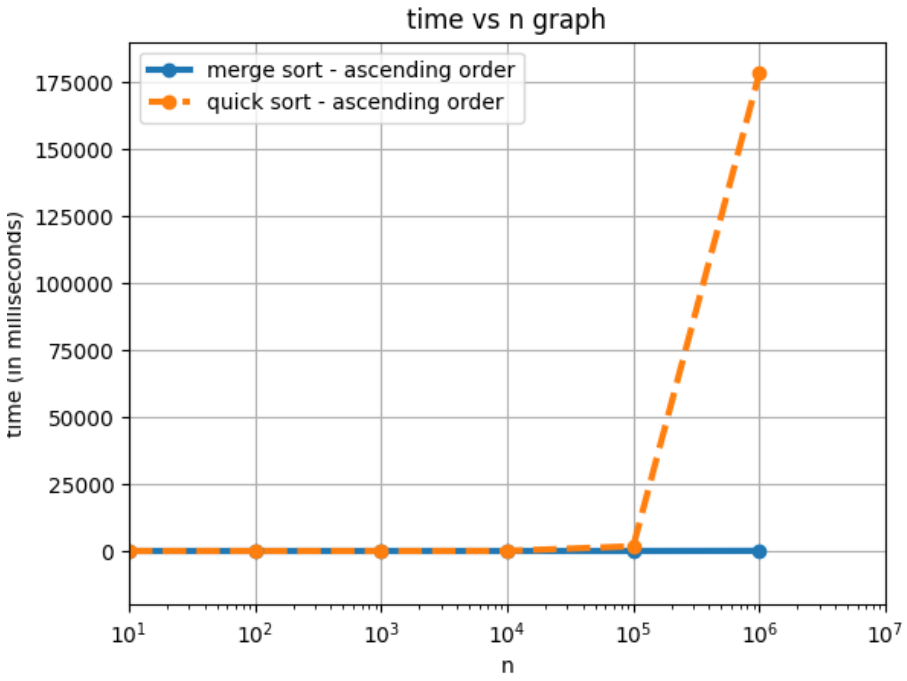
(N.B: Graph plotted in logarithmic scale for both axis)

For properly visualizing the running time of the algorithms, I have added the individual graph plot for each input order.

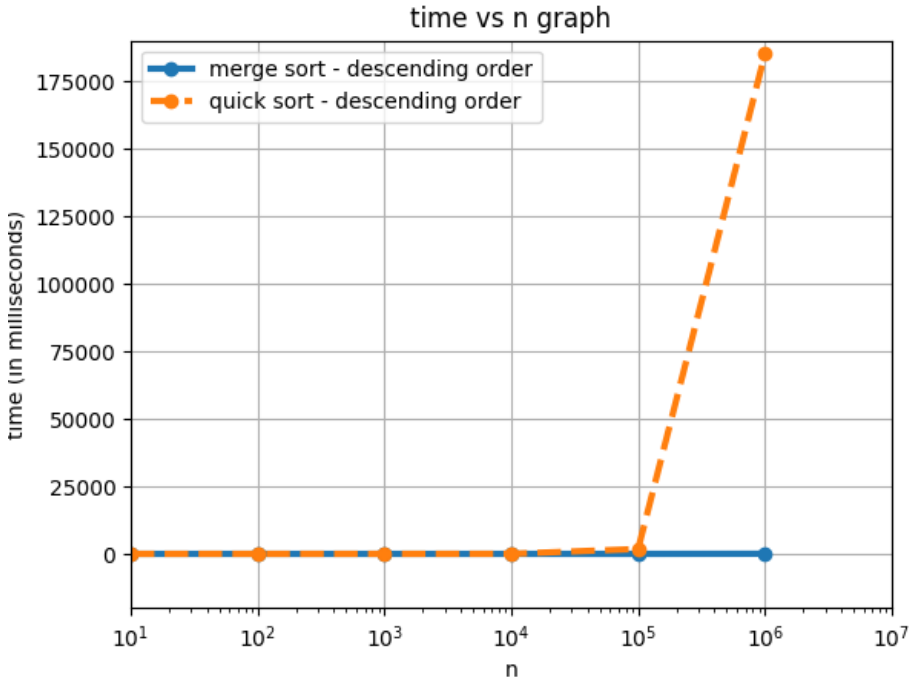
Plot Graph By Input Order

(N.B: Graph plotted in logarithmic scale for X axis)

Ascending Order:



Descending Order:



Random Order: