Algorithms and Data Structures
ITCS 6114 Fall 2016
Final Programming Project

Joyta Choudhury (800966655)
Vipul Shukla (800935439)

1.Assume the n input elements are integers in the range [0,n-1]. For each algorithm, determine what are best, average, and worst-case inputs. Your write-up should list these for each algorithm. Include a sentence or two of justification for each one. Please note that this is a theoretical question. You should answer what you expect to be true based on a theoretical analysis (and you should not refer to experimental results). In the subsequent questions we will compare the experimental results to these theoretical predictions.

Theoretical Analysis:

Algorithm	Time Complexity:Best	Time Complexity:Average	Time Complexity:Worst
Bubble sort	O(n)	$O(n^2)$	$O(n^2)$
Merge sort	O(n log(n))	O(n log(n))	O(n log(n))
Counting sort	O(n+k)	O(n+k)	O(n+k)

Bubble Sort:

Best Case Input: Sorted List
Average Case Input: Random List
Worst Case Input: Reverse Sorted List

Bubble sort compare every adjacent pair, swap their position if they are not in the right order (the latter one is smaller than the former one). After each iteration, one less element (the last one) is needed to be compared until there are no more elements left to be compared.

Best Case: Bubble sort produces best case when the input array is already sorted. In this case since the input array is already sorted no swaps are made.
Worst Case: Bubble sort produces the worst case when the input array is reverse sorted. In this the algorithm has to make a swap for every input.

Merge Sort:

Best Case Input: Not affected by input array
Average Case Input: Not affected by input array
Worst Case Input: Not affected by input array

Merge sort works on the concept of Divide-and-conquer.

Conceptually, a merge sort works as follows:

1. Divide the unsorted list into *n* sublists, each containing 1 element (a list of 1 element is considered sorted).
2. Repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining. This will be the sorted list.

Merge sort is not affected by the input array. It produces the same running time for sorted and reverse sorted array. Since merge sort first divides the input array into halves and keeps repeating the process until the the list contains only single element. At this point, when algorithm starts merging the numbers, it assumes that the given arrays are already sorted. Hence, its performance is not affected by the input array being sorted or reverse sorted.

Counting Sort:

Best Case Input: Input array having small numbers
Average Case Input: Input array having comparatively larger numbers.
Worst Case Input: Input array having significantly large numbers.

Counting sorts works by creating a temporary array which stores the number of times a number appears in the input array. For this it creates a new array of size equal to the largest number in the input array. Hence, If the input contains repeated numbers and the highest number in the input is small, then algorithm has to create a small array.

However, if the input has a big number, in that case a huge temporary array needs to be created and initialized. This increases the running time of counting sort hence reducing the performance of the algorithm.

2. Describe your experimental setup.

Experimental Analysis:

a. What kind of machine did you use?

Language: Java
IDE: Intellij IDEA
Operating System: Windows 10
RAM: 6 GB
System Type: 64- bit Operation System
Processor - Intel Core i5 x

b. What timing mechanism?

Public static long currentTimeMillis() is used as the timing mechanism to calculate the time taken by the algorithms to sort the input array.This method returns time in milliseconds.

While the unit of time of the return value is a millisecond, the granularity of the value depends on the underlying operating system and may be larger.

c. How many times did you repeat each experiment?

We repeated each experiment 25 times with 25 input files.

d. What times are reported?

Experimental Results:

Input File Name	Input Size	Bubble Sort (Milliseconds)	Merge Sort (Milliseconds)	Counting Sort (Milliseconds)
Input_sorted.txt	1000	0	0	0
Input_sorted.txt	10000	16	0	0
Input_sorted.txt	50000	317	19	0
Input_sorted.txt	100000	1266	16	0
Input_sorted.txt	500000	33090	62	31
Input_sorted.txt	1000000	155173	133	32
Input_reverse_sorted.txt	1000	0	0	0
Input_reverse_sorted.txt	10000	110	16	0
Input_reverse_sorted.txt	50000	3308	16	15
Input_reverse_sorted.txt	100000	13786	32	15
Input_reverse_sorted.txt	500000	362751	62/78	31
Input_reverse_sorted.txt	1000000	1527915	188	48
Input_random.txt	1000	0	0	0
Input_random.txt	10000	108	0	15
Input_random.txt	50000	2434	16	0
Input_random.txt	100000	9181	32	15
Input_random.txt	500000	232971	94/109	31
Input_random.txt	1000000	941422	219	47

e. How did you select the inputs you would use?

Basically we chose three types of input files:

1) Random input file: This file contains inputs which has random values.
2) Sorted input file: This file contains inputs which are already sorted. The performance of some algorithms and implementations may vary dramatically, particularly from their asymptotic worst-cases, if the input is already sorted.
3) Reverse-sorted input file: This file contains inputs which are reverse sorted. The performance may vary dramatically if the input to a sorting program is in reverse-sorted order.

f. Do you use the same inputs for all sorting algorithms?

We used same input files for all sorting algorithms so that we could conclude the asymptotic analysis of the three algorithms. Our motive behind taking same input files was to check the accuracy of the sorting algorithms and compare theoretical and experimental results with at most precision.

3. Which of the three sorts seems to perform the best? Graph the average case running time as a function of input size for the three sorts. Graph the best case running time as a function of input size for the three sorts. Graph the worst case running time as a function of input size for the three sorts. (For each case, your graphs should include the running times versus the input size). You will need to consider the best/worst case inputs for all algorithms.

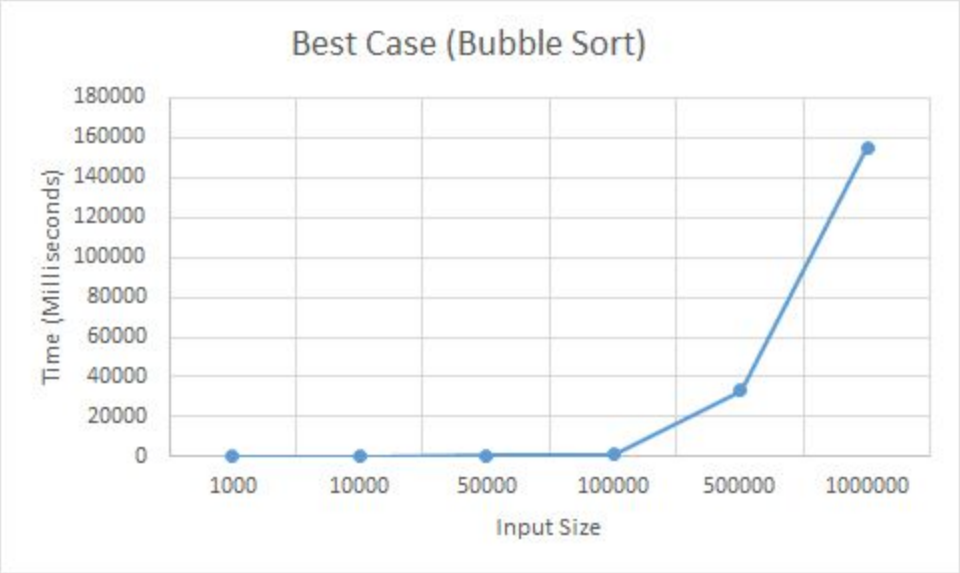
Counting sort seems to be performing the best. The running time of algorithms is:

COUNTING SORT > MERGE SORT > BUBBLE SORT

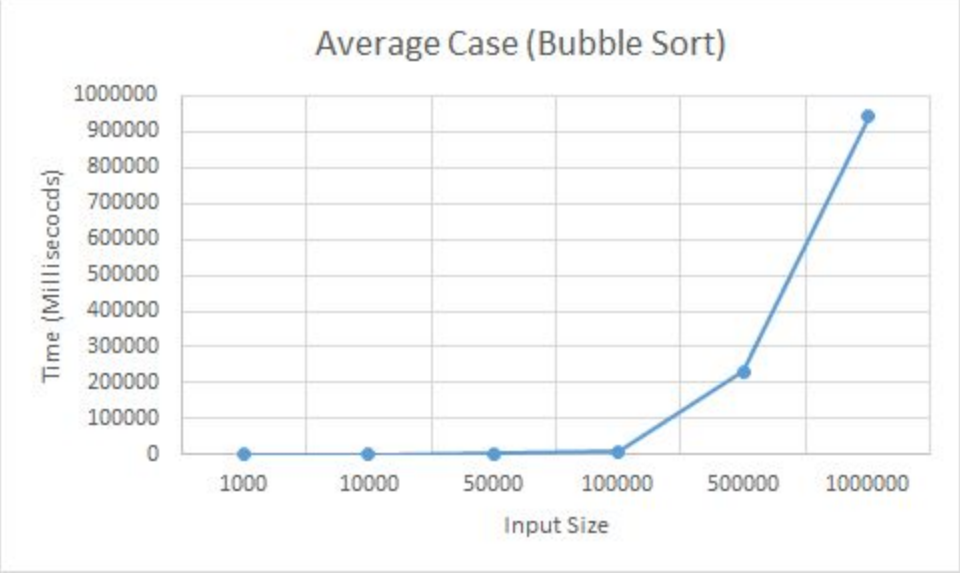
Below are the graphs which were plotted based on the results of experimental analysis:

BUBBLE SORT:

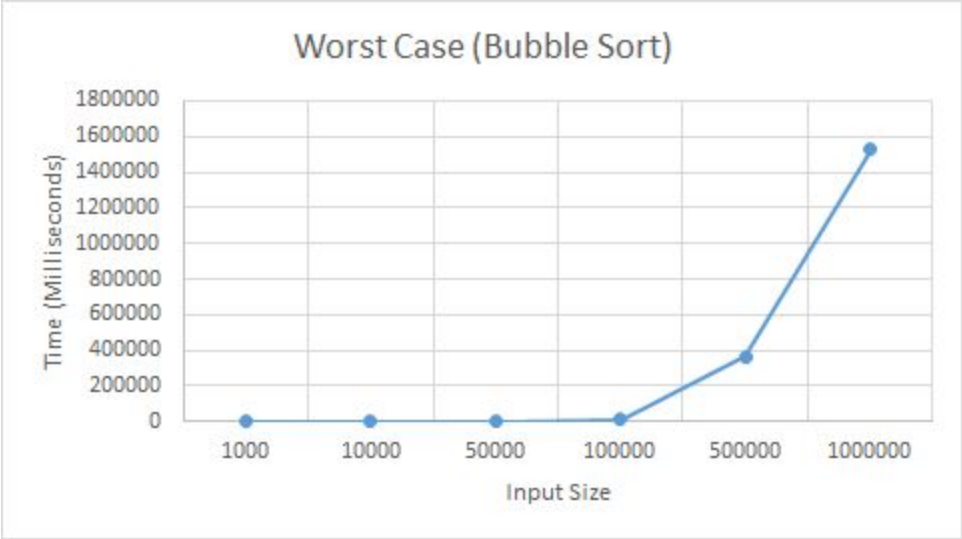
Best Case:



Average Case:

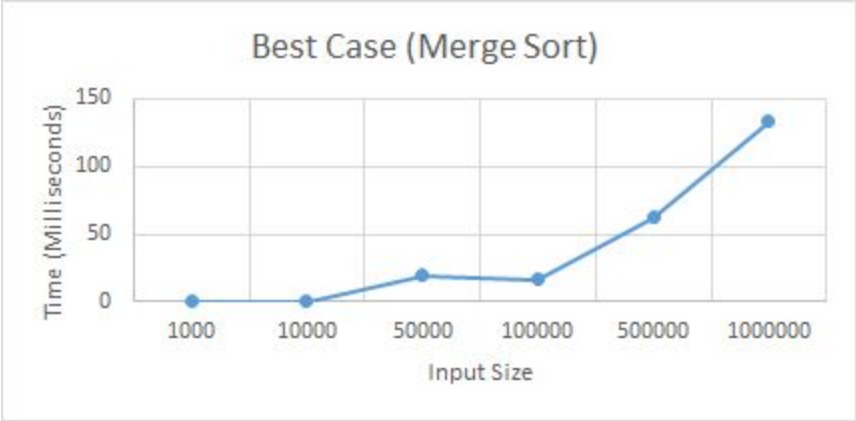


Worst Case:

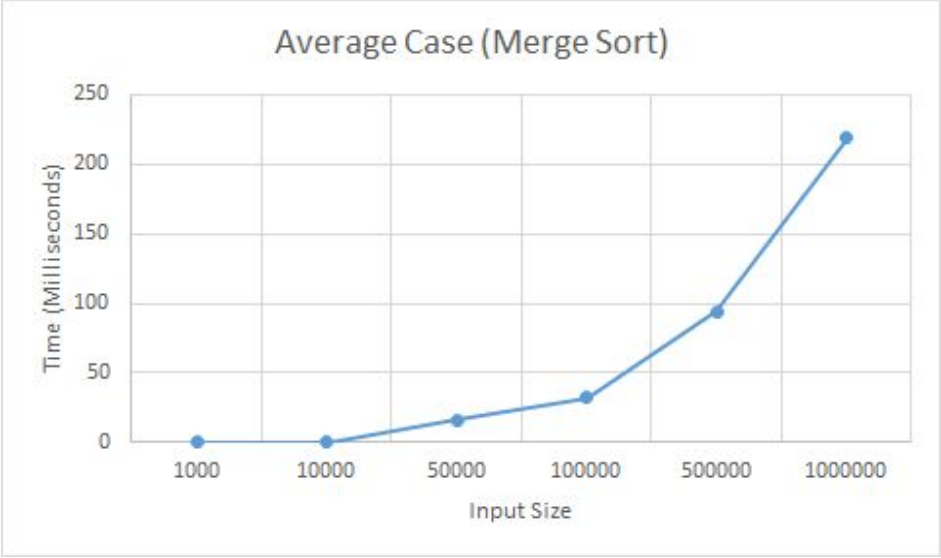


MERGE SORT:

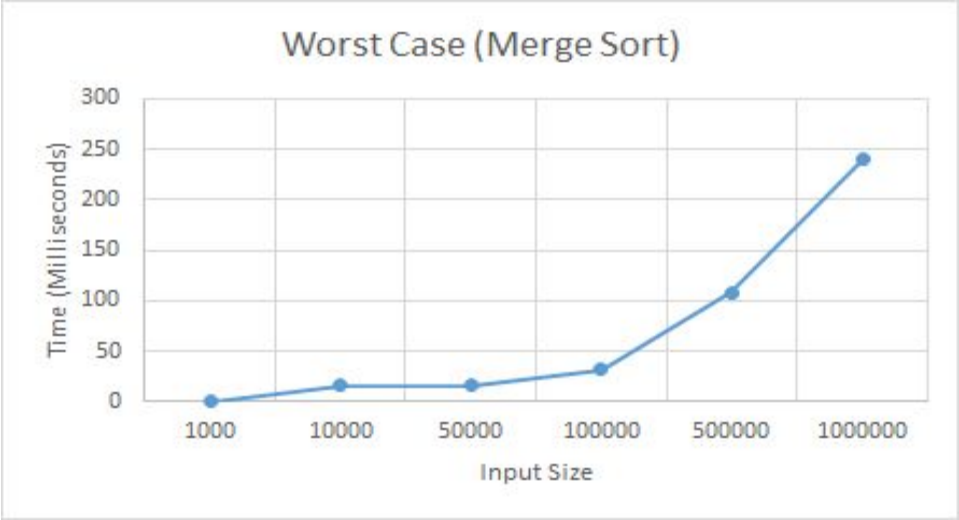
Best Case:



Average Case:

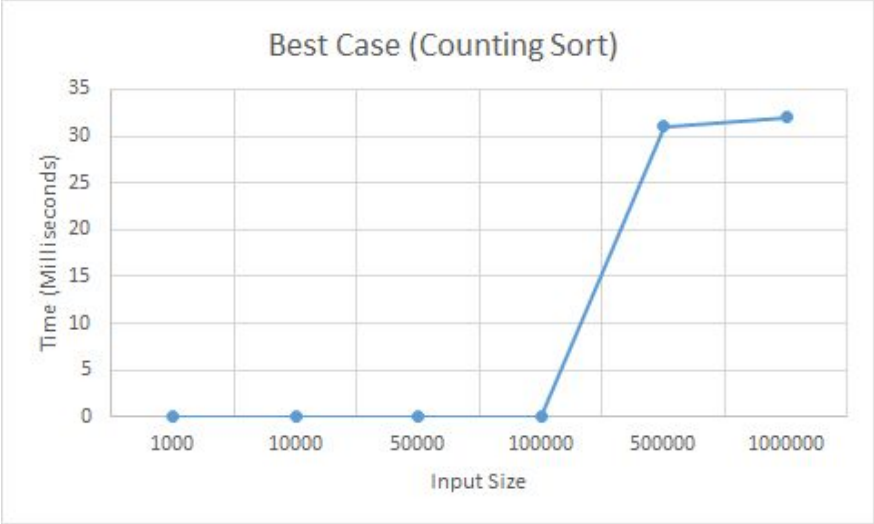


Worst Case:

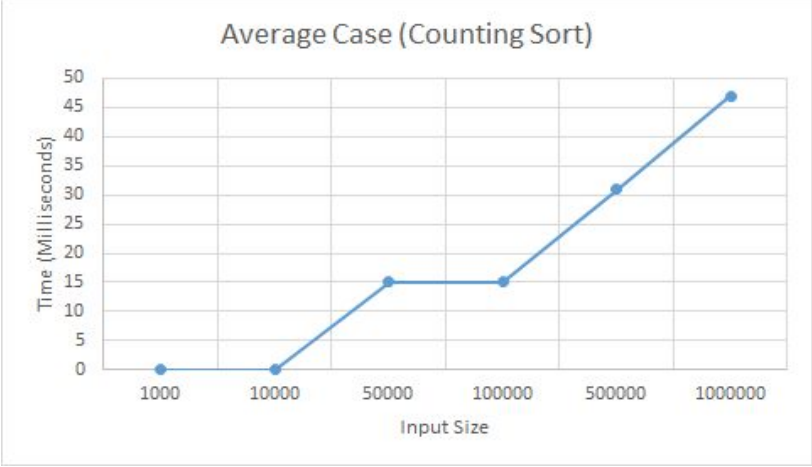


COUNTING SORT:

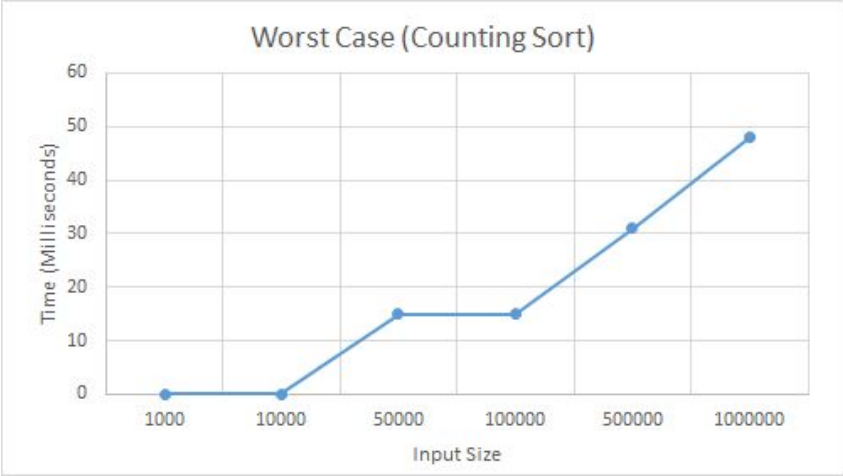
Best Case:



Average Case:



Worst Case:



Observation:

Counting sort is fast because of the way that elements are sorted using their values as array keys. This means that more memory is required for the extra array at the cost of running time. It runs in O(n + k) time where n is the number of elements to be sorted and k is the number of possible values in the range.

So as we increase the range of the input, the efficiency of Counting Sort Algorithm decreases as running time depends on k where k is the number of possible values in the range.

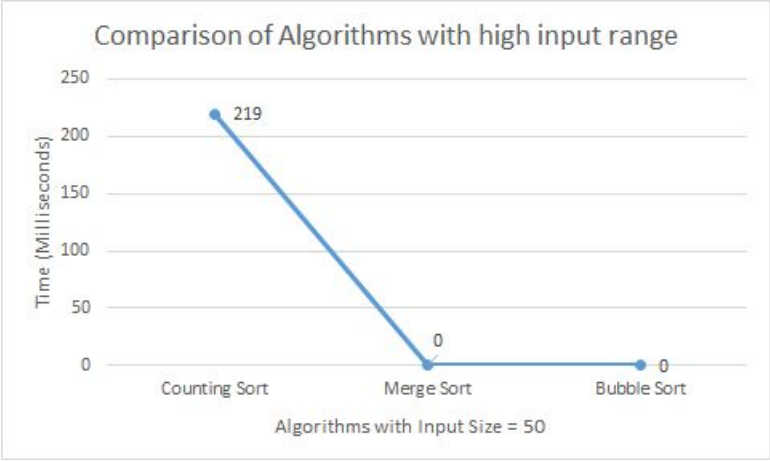
To justify this we took input file with Input size: 50

Elements were :

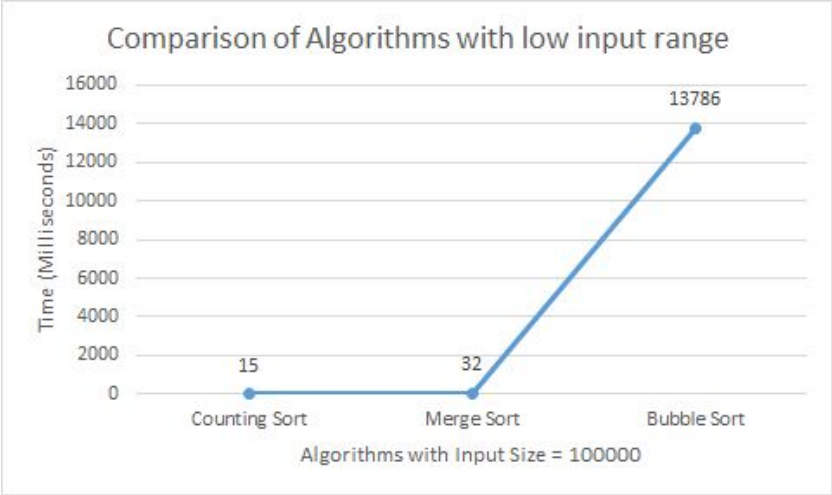
123456789,98876555,1,2,3,4,4,56,4,2,2,4,5,6,3,2,3,4,5,42,3,4,6,56,5,34,23,24,45,45,4,34,232,123455673,0,12,23,34,45,6,57,67,12,34,12,87,1,2,9,12

As the largest number is 123456789 so Counting Sort took 219 milliseconds whereas Bubble Sort and Merge Sort took 0 milliseconds. Thus we conclude that, in cases where range of the input is large, Counting Sort is not that efficient as compared to Merge Sort.

Below is the graph to depict this scenario.



When we take input file with small range it works as expected. This is shown in the below graph:



4. To what extent do the best, average and worst case analyses (from class/textbook) of each sort agree with the experimental results? To answer this question you need to find a way to compare the experimental results for a sort with its predicted theoretical times. For this you will need to do something somewhat rigorous. For example, one way to compare a time to a predicted time of O(n2) would be to divide the times for a number of runs with different input sizes by n2 and see if you observe a horizontal line (after some input size n0).

That n0 would represent the n0 value for the asymptotic analysis. The value on the y-axis (assuming you put input size on the x-axis) would tell you the constant value of the big-Oh. Finally - remember you are supposed to be analyzing the time for the experiments. I.e., it is not showing anything to show that the number of comparisons is O(n2) - you need to see that the time is O(n2) to determine if the asymptotic analysis is any good. In particular, you should:
a. For each sort, and for each case (best, average, and worst), determine whether the observed experimental running time is of the same order as predicted by the asymptotic analysis. Your determination should be backed up by your experiments and analysis and you must explain your reasoning. If you found the sort didn't conform to the asymptotic analysis, you should try to understand why and provide an explanation.

Firstly taking each case (best, average and worst) into account:

1) **Bubble Sort:**

We observed that experimental running time is of the same order as predicted by the asymptotic analysis. As depicted in the graphs in the previous question, we can see that running time of the algorithm for different cases is as follows:

BEST CASE > AVERAGE CASE > WORST CASE

which is expected.

Algorithm	Time Complexity:Best	Time Complexity:Average	Time Complexity:Worst
Bubble sort	O(n)	$O(n^2)$	$O(n^2)$

2) **Merge Sort:**

We observed that experimental running time is of the same order as predicted by the asymptotic analysis. As depicted in the graphs in the previous question, we can see that running time of the algorithm for different cases is as follows:

BEST CASE > AVERAGE CASE > WORST CASE

which is expected.

Algorithm	Time Complexity:Best	Time Complexity:Average	Time Complexity:Worst
Merge sort	O(n log(n))	O(n log(n))	O(n log(n))

3) Counting Sort:

We observed that experimental running time is of the same order as predicted by the asymptotic analysis. As depicted in the graphs in the previous question, we can see that running time of the algorithm for different cases is as follows:

BEST CASE > AVERAGE CASE > WORST CASE

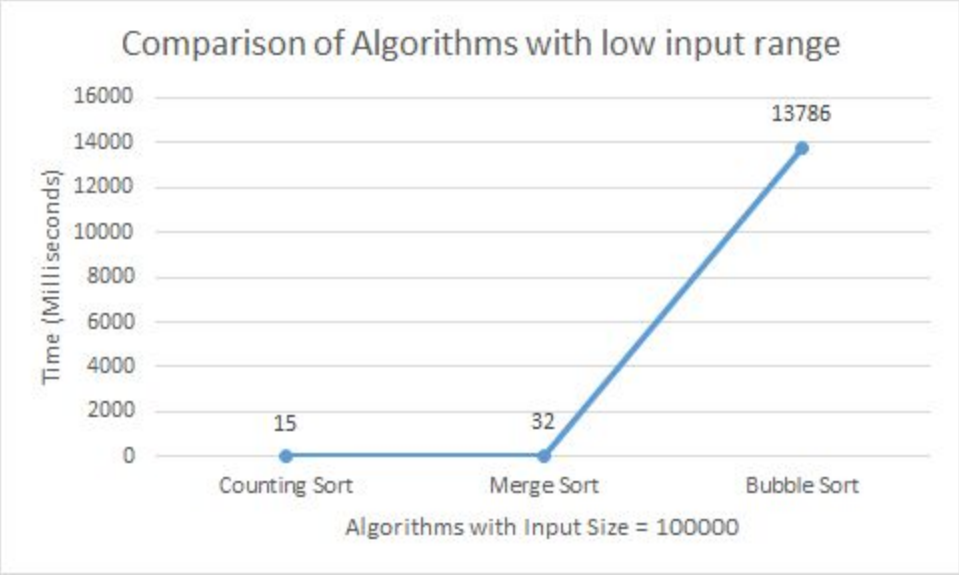
which is expected.

Algorithm	Time Complexity:Best	Time Complexity:Average	Time Complexity:Worst
Counting sort	O(n+k)	O(n+k)	O(n+k)

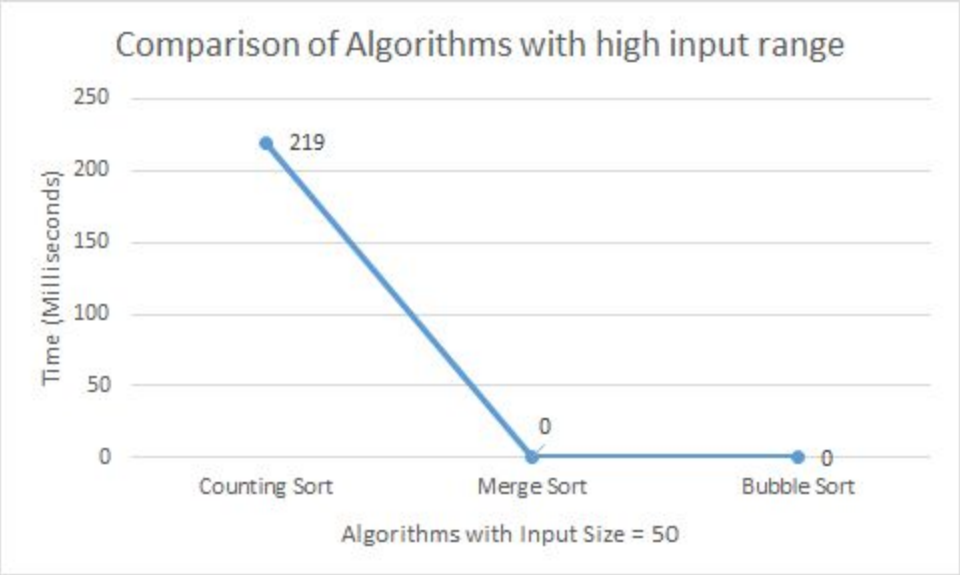
Now comparing algorithms with each other, we draw conclusion that Counting Sort consumes less time than Bubble and Merge Sort. This can be explained from the readings and graphs in the previous question. When we tried to run the algorithms with different input sizes and different types of files i.e. sorted input file, reverse sorted input file and random input file we concluded that the running time of algorithms is as follows:

COUNTING SORT > MERGE SORT > BUBBLE SORT

Generalised graph for the comparison:



But there is a special scenario where Counting Sort is not efficient as compared to Merge Sort. Counting sort is fast because of the way that elements are sorted using their values as array keys. This means that more memory is required for the extra array at the cost of running time. It runs in O(n + k) time where n is the number of elements to be sorted and k is the number of possible values in the range. So, during running the algorithm for this scenario with the input file with higher range, we concluded that Merge Sort is more efficient than Counting Sort.



b. For each sort, determine the constants and n0 that are hidden by the asymptotic analysis. These can be computed experimentally as discussed above. Again, your determination should be backed up by your experiments and analysis and you must explain your reasoning. If you found the sort didn't conform to the asymptotic analysis in the previous question, then you should try to determine what asymptotic behavior it does exhibit and answer this question for it.

Bubble Sort: For bubble sort the running time of algorithm is given by $O(n^2)$, i.e. while input increases at a linear rate, the time increases at a polynomial rate of n^2

The running time of bubble sort can be shown as:-
$T(n) = C * O(n^2)$.

Hence in order to calculate C, we need to divide T(n) by n^2.
$T(n)/n^2 = C$
Hence if we divide the timings by square of input size, we would get the constant.

Input File Name	Input Size	Bubble Sort (Milliseconds)	$C = T(n)/n^2$
Input_sorted.txt	1000	0	0
Input_sorted.txt	10000	16	1.6 * 10^ -7

Input_sorted.txt	50000	317	1.2 * 10^ -7
Input_sorted.txt	100000	1266	1.26* 10^ -7
Input_sorted.txt	500000	33090	1.32* 10^ -7
Input_sorted.txt	1000000	155173	1.5* 10^ -7

Hence in the case of Bubble sort constant comes out to be approximately 1.2* 10^ -7.

Merge Sort: For merge sort the running time of algorithm is given by O(n log(n)), i.e. while input increases at a linear rate, the time increases at a polynomial rate of (n log(n)).

The running time of bubble sort can be shown as:-
T(n) = C * O(n log(n)).

Hence in order to calculate C, we need to divide T(n) by (n log(n)).
T(n)/(n log(n)) = C
Hence if we divide the timings by (n log(n)), we would get the constant.

Input File Name	Input Size	Merge Sort (Milliseconds)	C = T(n)/(n log(n))
Input_sorted.txt	1000	0	0
Input_sorted.txt	10000	0	0
Input_sorted.txt	50000	19	0.000080
Input_sorted.txt	100000	16	0.000032
Input_sorted.txt	500000	62	0.000012
Input_sorted.txt	1000000	133	0.000018

Hence in the case of Merge sort constant comes out to be approximately 3.2* 10^ -5.

And the value for n0 is calculated based on the observation from experimental results which is :
n0 = 100,000

Counting Sort: For counting sort the running time of algorithm is given by O(n+k), i.e. both the input and running time increases at a linear rate.

The running time of counting sort can be shown as:-
T(n) = C * O(n + k).

Hence in order to calculate C, we need to divide T(n) by O(n + k).

T(n)/(n+k) = C

Hence if we divide the timings by square of input size, we would get the constant.

Input File Name	Input Size	Counting Sort (Milliseconds)	C = T(n)/(n+k)
Input_sorted.txt	1000	0	0
Input_sorted.txt	10000	0	0
Input_sorted.txt	50000	0	0
Input_sorted.txt	100000	0	0
Input_sorted.txt	500000	31	6.2 * 10^ -5
Input_sorted.txt	1000000	32	3.2 * 10^ -5

In the case of Counting sort constant comes out to be approximately 3.2* 10^ -5.

And the value for n0 is calculated based on the observation from experimental results which is :

n0 = 500,000