CSE 5311 – Design and Analysis of Algorithms
Order Statistics and Sorting- Project Report
Fall 2015

By,

Anurag Domakonda (1001151021)

Nagabhushanam Motamarri (1001151229)

Shravan Kokkula (1001153299)

Vineet Nair (1001163230)

Order statistics

Finding Kth smallest

1) **Worst Case Linear Time Approach:**
   To find the Kth smallest element using the worst case approach, we used list data structures to store the n elements that are fetched from the large input file. We tested the data for group size of 3, 5, 7. We sorted the data in each group using merge sort. After grouping and sorting, we found the medians in each group by calling a method which then returns all the medians in a list. Using this medians list as input we found the median of median which serves as a pivot. Based on the pivot value, we partitioned the entire input list in such a way that the elements that are smaller than the pivot resides to the left of the pivot and the elements that are greater than the pivot on the right side. By comparing the values of K and left partition size we recursively call the same method until we find the kth smallest element.

Functions Used:

findMedian(): returns median for given list.

kthSmallest(): find the kth smallest element with elements in groups of 3,5,7.

partition(): find the position of the particular element in array.

swap(): swap the list elements positions.

2) **Expected Linear Time:**

   In this approach, we randomly picked an element from the input array and used this element as pivot and used the same approach as in worst case linear time approach.

Functions Used:

getPivotIndex(): Select a random element as pivot.

quickSort(): Find the position of pivot and then apply quicksort to two partitions on either side of pivot.

partition():find the position of the particular element in array.

swap(): swap the list elements positions.

Finding Top K Numbers

Using Min-Heap:

In this approach, we insert the k elements into min-heap, these elements are inserted in such order that min-value is at the root. Now, we swap this minimum value with every element in list and perform heapify operation. At the end of this operation we are left with top k elements in the given array of elements. By using the combination of min heap of k elements and using the original list to compare with it, we ensure that this process is completed in O (n log k).

Functions Used:

min_Heapify(): For given element , find correct position in min heap

buildHeap(): For given elements, build a min heap with the smallest element at the root.

Sorting

1) Heap Sort:

We read the data from the input file and stored it in **list.** Based on the elements in the input list, we built the max heap. Then until the heap is empty, removed the maximum element from the heap (root element), heapifying the remaining elements in the heap and store the removed maximum element in the input array starting from the last index.

Functions Used:

max_Heapify(): For given element in list find its correct position in the heap.

buildHeap(): For given list convert them in heap structure.

Heap_Sort(): Perform sorting for the list after creating its heap structure.

Return_Max(): Return the top element in heap which is largest.

2) Classical Quick Sort:

The position of pivot in the array and divides the array into two parts (left, right) based on the pivot, sorts the elements on both sides recursively using an array data structure.

Functions Used:
quickSort(): To sort elements into 2 arrays with elements less than pivot in left and greater in right.

3) Quick Sort using random element as pivot:

In this approach, we randomly picked an element from the input list and used this element as pivot and performed quick sort.

Functions Used:

getPivotIndex(): Select a random element as pivot.

quickSort(): Find the position of pivot and then apply quicksort to two partitions on either side of pivot.

partition(): Find the position of the pivot in the list.

swap(): Swap the list elements positions.

4) **Quick Sort using median element of three random elements as pivot:**
In this approach, we find three random elements in list and then selected the median of these random numbers and used this element as pivot to perform quick sort.

Functions Used:

getPivotIndex(): To get pivot from three random elements in list and choose the appropriate one.

quickSort(): Find the position of pivot and then apply quicksort to two partitions on either side of pivot.

partition(): Find the position of pivot in the list.

swap(): Swap the list elements positions.

5) **Quick Sort with Insertion Sort:**
In this approach, we randomly picked an element from the input list and used this element as pivot and performed quick sort. Additionally, when the left and right part of the list are less than 10 in size we perform insertion sort as it is more efficient then quick sort for smaller lists.

Functions Used:

getPivotIndex(): Selects a random element as pivot.

quickSort(): Divides the list into two parts around pivot element using quicksort.

insertionSort(): If number of elements in list are less than 10 then use Insertion sort.

partition(): Find the position of pivot in the list.

swap(): Swap the list elements positions.

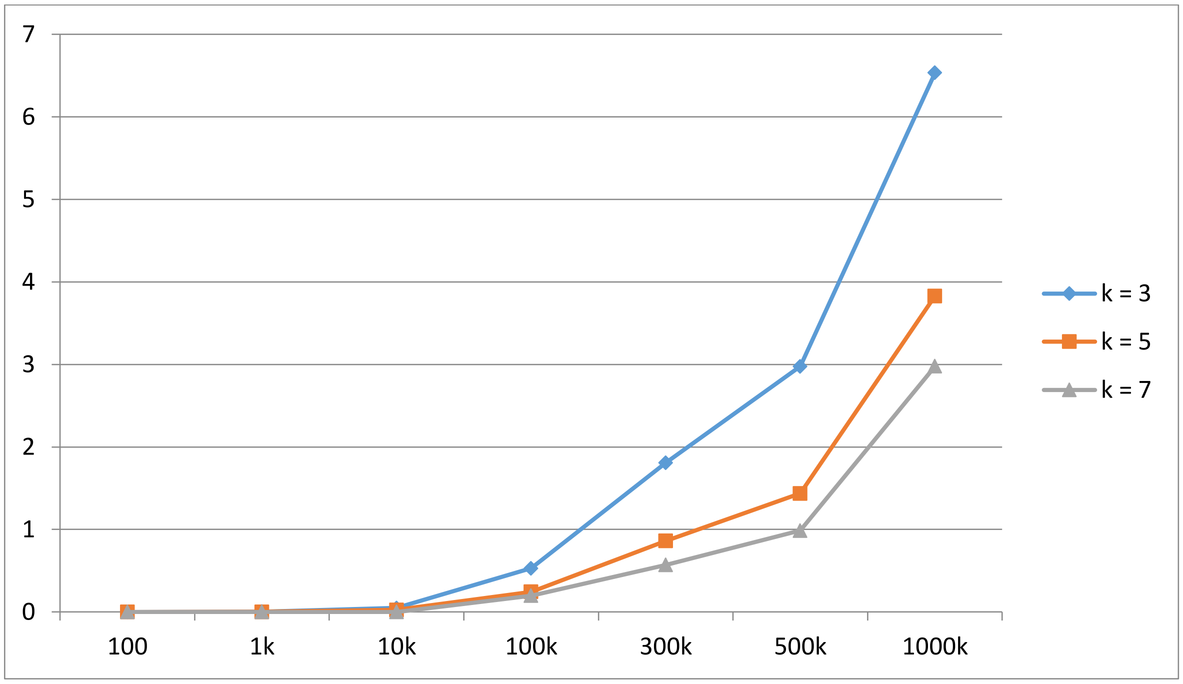
Experimental Analysis:

Notations Used in Graphs:

- CQS – Classical Quick Sort
- RQS – Randomized Quick Sort
- MQS – Median Heuristics Quick Sort
- IQS – Insertion Quick Sort
- JQS – Java Quick Sort
- PQS – Python Quick Sort
- HS – Heap Sort

MEDIAN of MEDIANS ALGORITHM

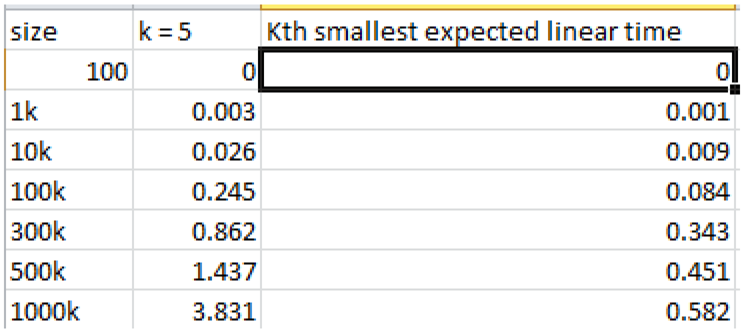
size	k = 3	k = 5	k = 7
100	0	0	0
1k	0.005	0.003	0
10k	0.048	0.026	0
100k	0.529	0.245	0.199
300k	1.807	0.862	0.571
500k	2.977	1.437	0.987
1000k	6.535	3.831	2.979



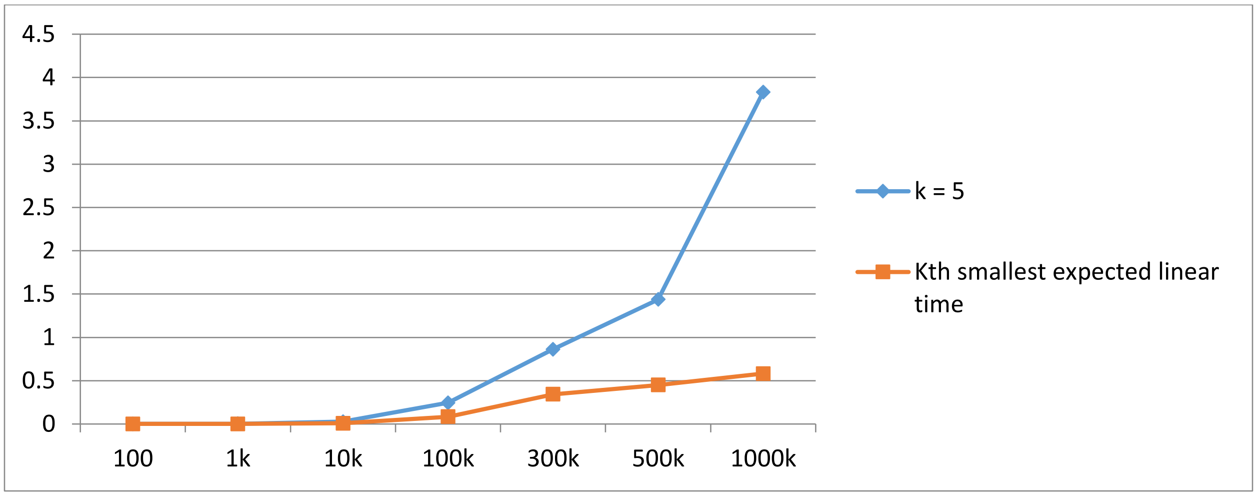
Runtime of median of median algorithm given, when we divide into groups of 3
T (n) = O (n) + T (n/3) + T (2n/3) so T (n) > O (n)

Grouping into 5 elements gives optimal solution. For millions of numbers grouping of 5, would led to large numbers of smaller set of 5 elements i.e. 5000000/5 = 1000000 sets. While grouping into 7 elements would led to a smaller set of 7 elements each i.e. 50000000/7=714285 which has less sets than group of 5, that would give faster runtime than groups of 5 and 3.

Comparison of deterministic and probabilistic algorithm for order statistics.



size	k = 5	Kth smallest expected linear time
100	0	0
1k	0.003	0.001
10k	0.026	0.009
100k	0.245	0.084
300k	0.862	0.343
500k	1.437	0.451
1000k	3.831	0.582



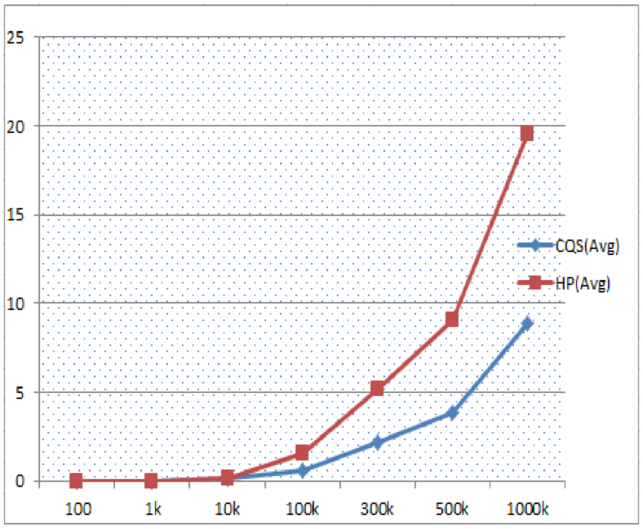
We have expected linear time algorithm running faster than worst case linear time algorithm when we increase input list as we are using randomized quick select which finds randomized pivot and then partitions the list.

Finding K Largest Numbers

In this, we have used min heap of size k to find top k elements of list. We first create a min heap of k elements and compare every element in array with root of min heap (smallest element) and replace the min element with element of list if it is smaller. This process continues till we reach end of the list. At the end, we are left with top k elements of the list. Due this process we are able to achieve O (nlogk) which is improvement on other methods.

HEAP SORT VS QUICK SORT (Average case)

size	CQS(Avg)	HP(Avg)
100	0.001	0.001
1k	0.004	0.01
10k	0.166	0.122
100k	0.624	1.532
300k	2.164	5.143
500k	3.84	9.098
1000k	8.877	19.572



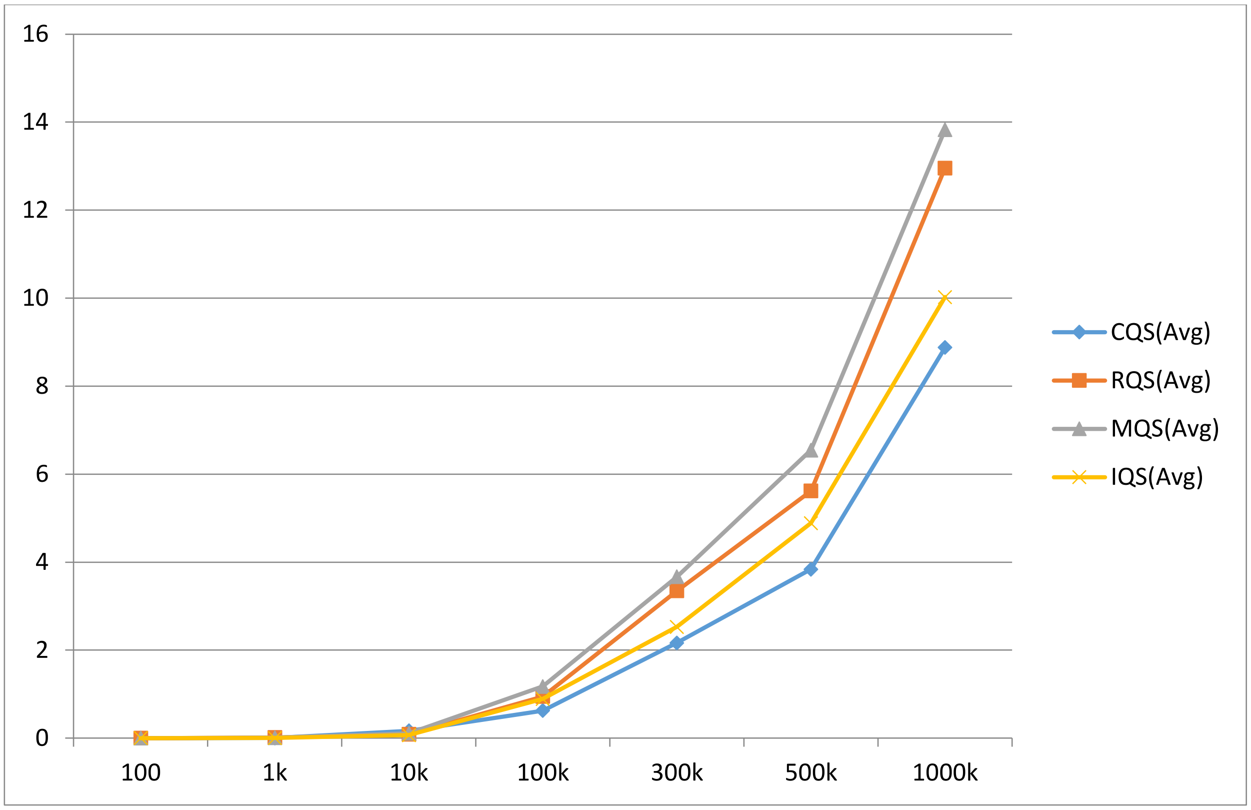
Quicksort is better than Heapsort though the worst case running time of heap sort is always O(nlogn), whereas Quick Sort worst case is O(n^2). But we can see above that Quick Sort runs faster than Heap Sort as because Quicksort almost doesn't do unnecessary element swaps whereas in Heap Sort even if all of your data is already ordered, we are going to swap 100% of elements to order the list. Secondly, Quick Sort has better locality of reference i.e. the next thing to be accessed is usually close in memory to the thing you just looked at. Whereas, heap sort jumps around significantly more. Since things that are close together will likely be cached together, quicksort tends to be faster.

However, quicksort's *worst-case* performance is significantly worse than heapsort's is. Because some critical applications will require guarantees of speed performance, heapsort is the right way to go for such cases.

QUICKSORT HUERISTICS

Readings are for Average case:

size	CQS(Avg)	RQS(Avg)	MQS(Avg)	IQS(Avg)
100	0.001	0.001	0.001	0.001
1k	0.004	0.013	0.009	0.004
10k	0.166	0.089	0.109	0.07
100k	0.624	0.945	1.176	0.893
300k	2.164	3.349	3.668	2.529
500k	3.84	5.617	6.545	4.888
1000k	8.877	12.957	13.829	10.023

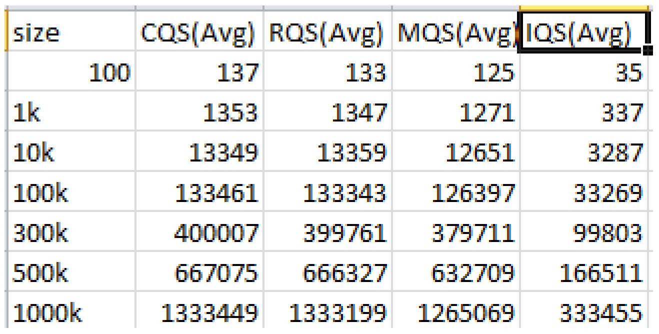


Randomized Quick Sort is always better than Classical Quick Sort as randomized quick sort takes O (n log n) independent of input size i.e. known as Worst-case Expected-Time bound which is always better than Classical quick sort which takes O (n^2) in worst case.

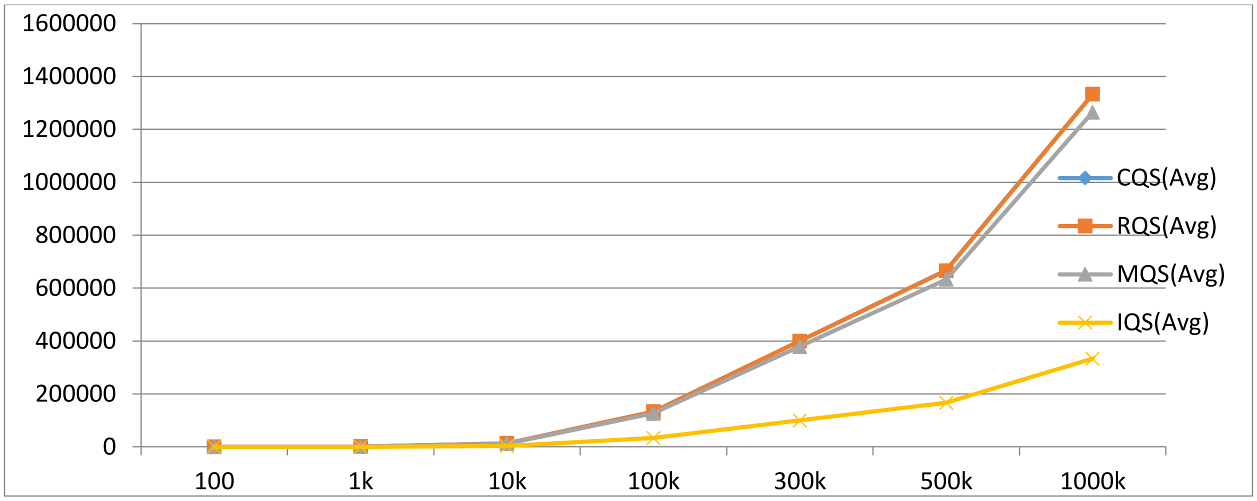
As Quick Sort runs, it breaks down the list into smaller lists. Once the lists becomes small enough than an Insertion sort becomes more efficient than the Quicksort, it will switch to the Insertion sort to complete sorting, which would take less time than the classical quicksort.

RECURSION DEPTH FOR QUICKSORT HEURISTICS

Recursion Count (Depth):

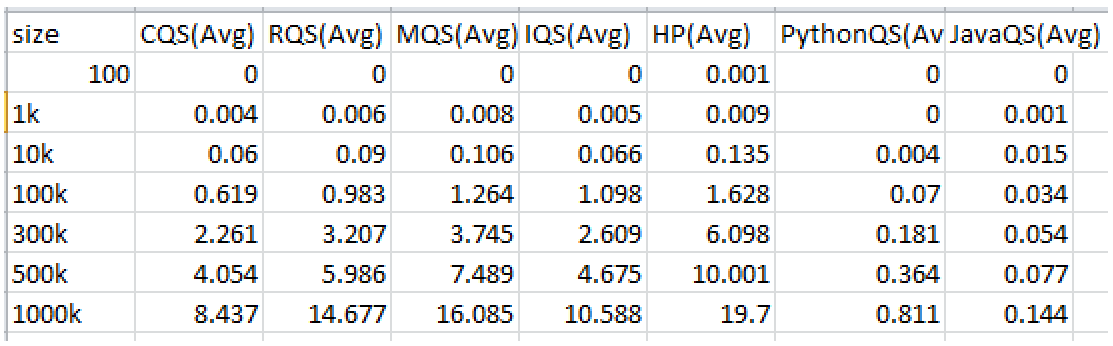


size	CQS(Avg)	RQS(Avg)	MQS(Avg)	IQS(Avg)
100	137	133	125	35
1k	1353	1347	1271	337
10k	13349	13359	12651	3287
100k	133461	133343	126397	33269
300k	400007	399761	379711	99803
500k	667075	666327	632709	166511
1000k	1333449	1333199	1265069	333455

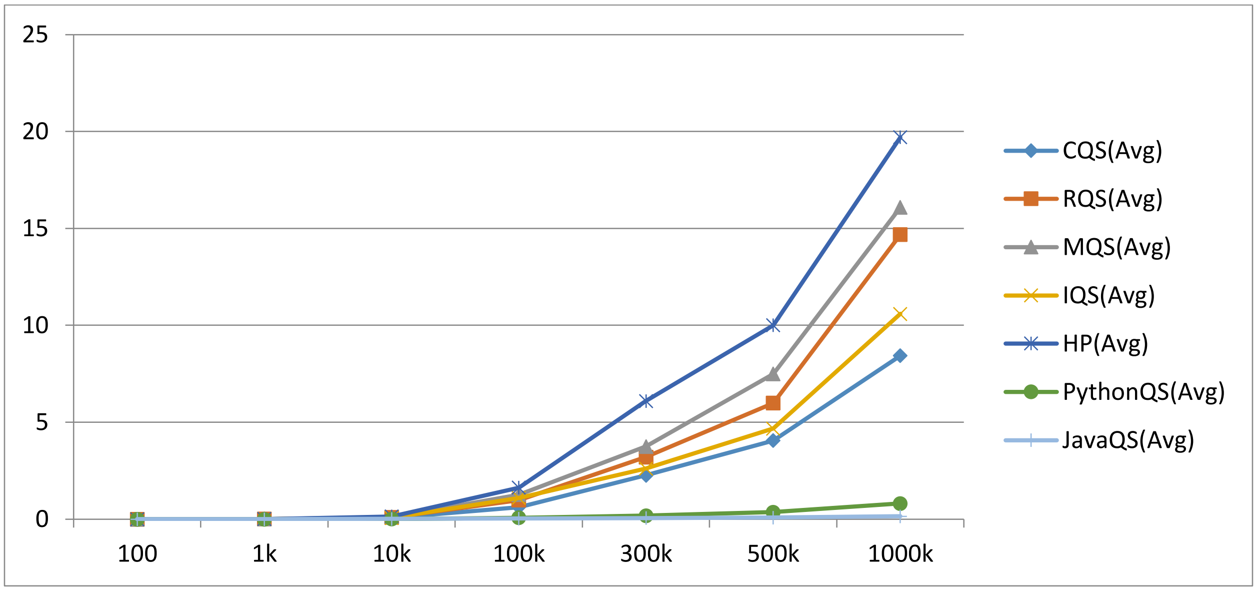


In the above graph the, the recursion count of Quick Sort algorithm is plotted. The randomized quick sort takes more number of steps than other types of quick sort algorithms as input size increases. As it selects random element as pivot and partitions the list around it.

SORTING ALGORITHMS HEURISTICS for (Normal Distribution) Inputs



size	CQS(Avg)	RQS(Avg)	MQS(Avg)	IQS(Avg)	HP(Avg)	PythonQS(Av	JavaQS(Avg)
100	0	0	0	0	0.001	0	0
1k	0.004	0.006	0.008	0.005	0.009	0	0.001
10k	0.06	0.09	0.106	0.066	0.135	0.004	0.015
100k	0.619	0.983	1.264	1.098	1.628	0.07	0.034
300k	2.261	3.207	3.745	2.609	6.098	0.181	0.054
500k	4.054	5.986	7.489	4.675	10.001	0.364	0.077
1000k	8.437	14.677	16.085	10.588	19.7	0.811	0.144



Normal distribution is an arrangement of a data set in which most values cluster in the middle of the range and the rest taper off symmetrically toward either extreme. A graphical representation of a normal distribution is sometimes called a bell curve because of its flared shape. Here, in our case as we increase the normal distribution input size, we find the results are similar to average case float values we had above with slight difference in time of execution of above mentioned algorithms.