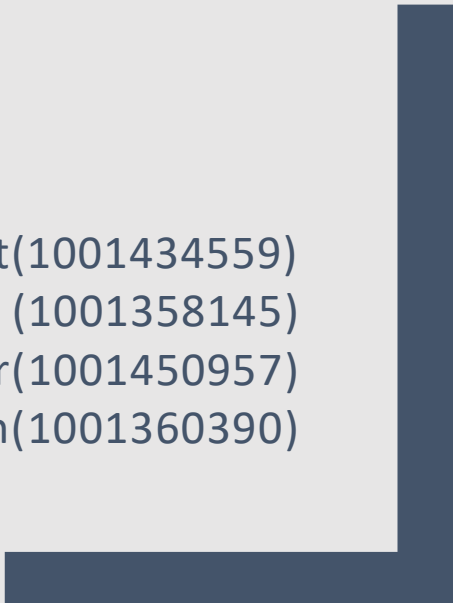




CSE 5311 – Design and Analysis of Algorithm

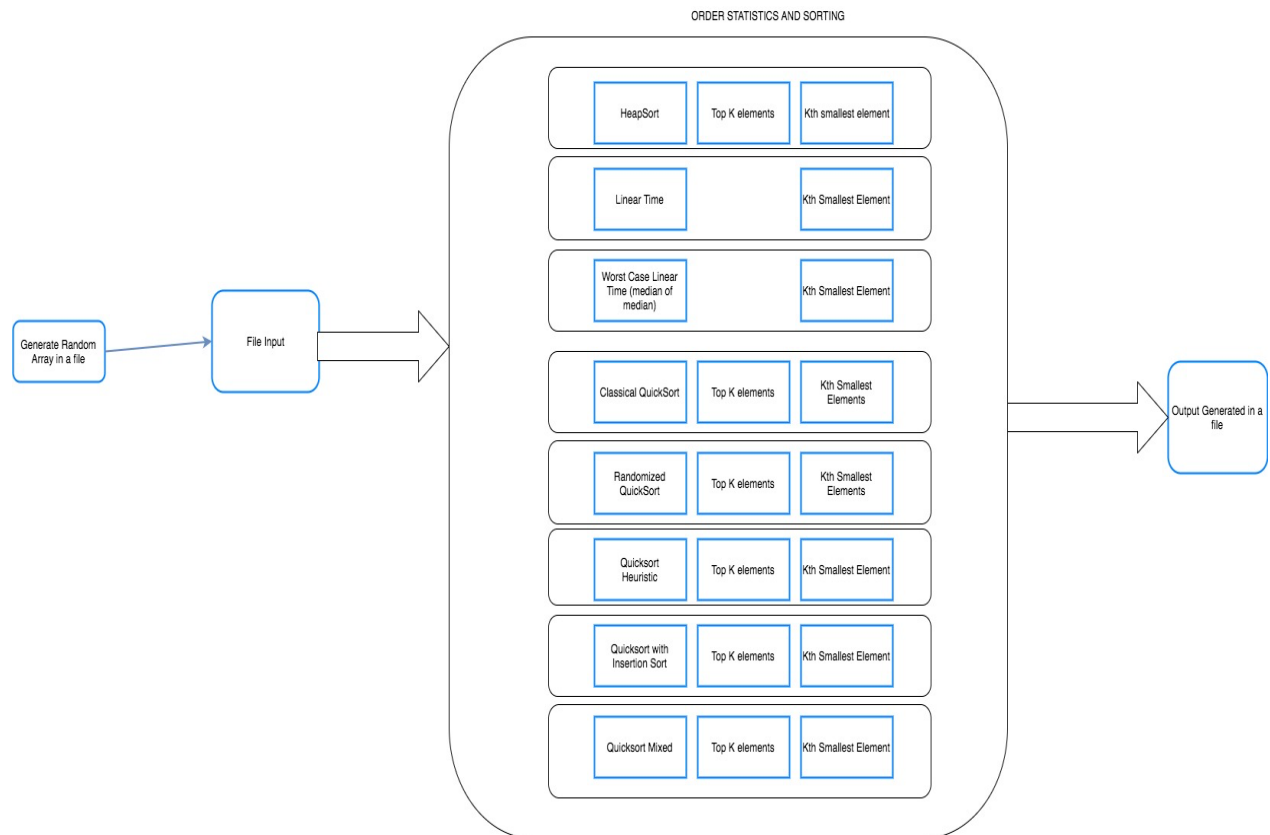
ORDER STATISTICS AND SORTING

Rushikesh Dixit(1001434559)
Sunny Shah (1001358145)
Nivedita Diwadkar(1001450957)
Darshan Hiremath(1001360390)



Contents	
System Design	03
Order Statistics	04
Worst Case Linear Method	05
Description	
Input/output Format	
Algorithm	
Time Complexity	
Data Structure	
Analysis (Graphs/Comparison)	
Expected Linear Time	7
Description	
Input/output Format	
Algorithm	
Time Complexity	
Data Structure	
Analysis (Graphs/Comparison)	
Sorting	09
Heap Sort	10
Description	
Input/output Format	
Algorithm	
Time Complexity	
Data Structure	
Analysis (Graphs/Comparison)	
Quick Sort	12
Description	
Algorithm	
Time Complexity	
Data Structure	
Analysis(Graphs/Comparison) for Heap Sort and Quick Sort.....	16

System Design



Array generator generates the input file, which is then used as input to the algorithm.

Steps to input generate file:

1. Array Generator class generates the input file for sorting and order statistics algorithm in '.txt' format.
2. Inputs:
 - a. Sorting algorithm:
 - Enter array input size
 - Enter data type (normalized data or random data)
 - Array of specified input size is written to 'sortingAlgorithmInput.txt' with array input size as first line of the file.
 - b. Order Statics:
 - Enter array input size
 - Enter 'k' element
 - Enter data type (normalized data or random data)
 - Array of specified input size is written to 'sortingAlgorithmInput.txt' with k and array input size as first line of the file.

Order Statistics

Worst Case Linear Time (via Median of Medians)

Description:

In this algorithm, we will implement the median of medians algorithm for the kth smallest element.

Input/output:

Input from a text file with first line containing k and input size of array, separated by a space followed by the input data (float integers stored as double) with one input per line.

Output to a file with kth smallest element on the first line and execution time (in 'ns') of the algorithm in the following line

Algorithm:

1. Group the elements in S array into small groups of equal chunk size.
2. Sort each chunk.
3. Let R be an array of all the medians of different chunks.
4. Let M be the median of medians. Retrieved by passing the array to kth smallest function.
5. Kth smallest function:
 - a. Pivot is the median of the array.
 - b. splitArray separates the array into three parts Left, median and right.
 - c. The kth smallest element is compared with median element.
 - d. If kth element is equal to median then kth element is returned.
 - e. If kth element is smaller than median then kth smallest is run on the Left array.
 - f. If kth element is greater than median then kth smallest is run on the right array.

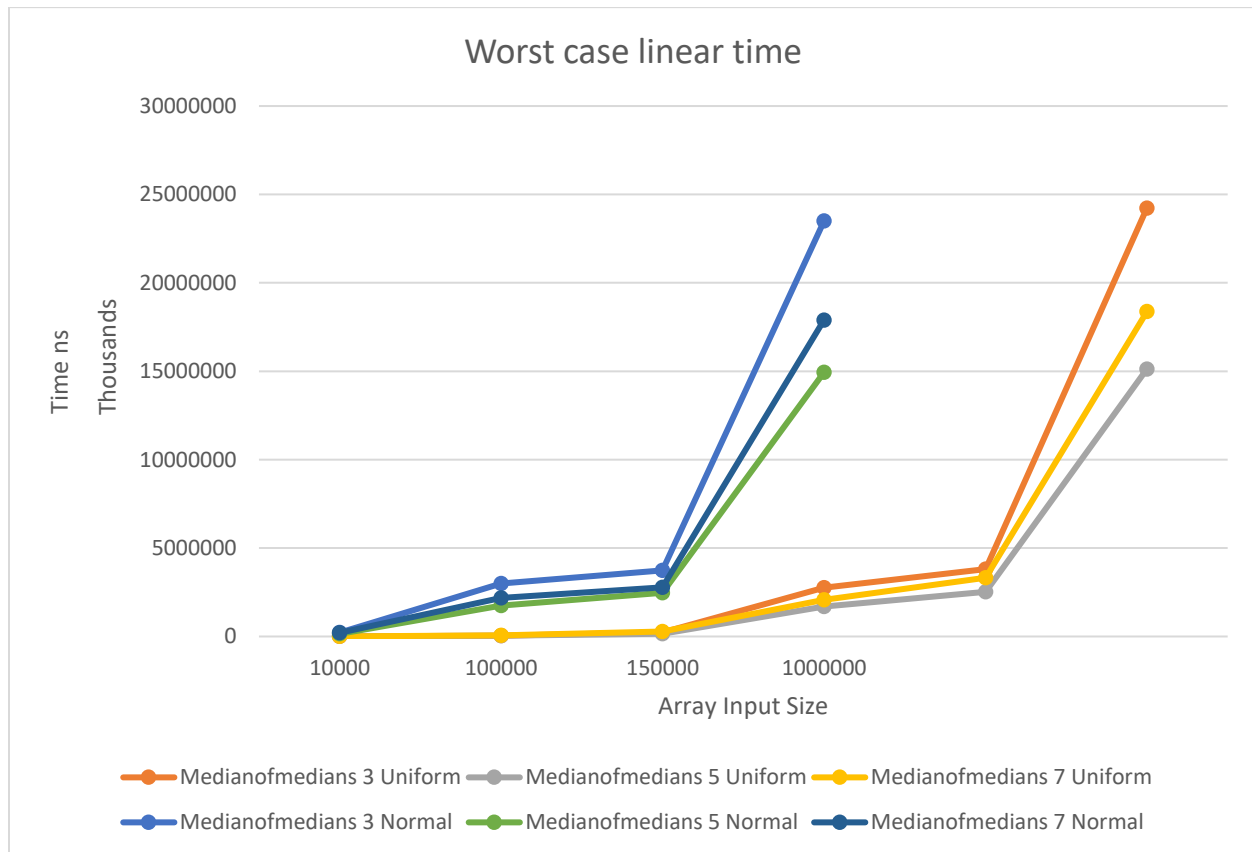
Time Complexity:

The time complexity of our algorithm is $O(n)$

Data Structure:

- Array (Double) to store floats.
- Integer k to store value of kth element
- Integer start, end, pivot to store indexes of the array.

Analysis(Graphs/Comparison):

**Analysis:**

- Above graph shows the running time of the algorithm for different input data sizes (as shown in graph) for the same value of k (5000) for both types of data distribution (normalized and uniform).
- Uniform data distribution has a slow rate of growth compared to the normalized data distribution.
- The execution time for 5 chunks of data takes the minimum time and 7 chunks of data takes the maximum amount of time in both normal or uniform distribution.

Expected Linear Time

Description:

Selection in expected linear time algorithm is used to find k^{th} selection (k^{th} smallest, more specifically) from unsorted data. It uses quicksort divide and conquer procedure to find the k^{th} smallest element, but not sorts the array completely.

Input/output:

Input from a text file with first line containing k and input size of array, separated by a space followed by the input data (float values) with one input per line.

Output to a file with k^{th} smallest element on the first line and execution time (in 'ns') of the algorithm in the following line.

Algorithm Pseudocode:

1. Read k and *input size* from file and create an array using the input size and *input data* from the input file.
2. Procedure: **quickSort** with *array*, *start index*, *last index* and k^{th} element as input to the procedure.
 - Check if $start \leq last$
False – Return pivot
True – Call procedure *splitArr* which returns the pivot index.
 - Compute k^{th} index ($k^{\text{th}} = \text{pivot} - \text{start} + 1$)
 - If $k \text{ element} < k^{\text{th}} \text{ index}$
Recursively call left partition and $k \text{ element}$
 - If $k \text{ element} > k^{\text{th}} \text{ index}$
Recursively call right partition and $k \text{ element} - k^{\text{th}} \text{ index}$
 - Else return pivot element. ($\text{pivot} == k \text{ element}$)
3. Procedure: **splitArr** with *array*, *start index* and *last index* as input.
 - Select random pivot.
 - while $first < last$
 - i. Increment first till first is less than pivot
 - ii. Increment last till last is greater than pivot
 - iii. Check if ($first < last$) -> swap first and last
 - Swap pivot and last element and return last index.

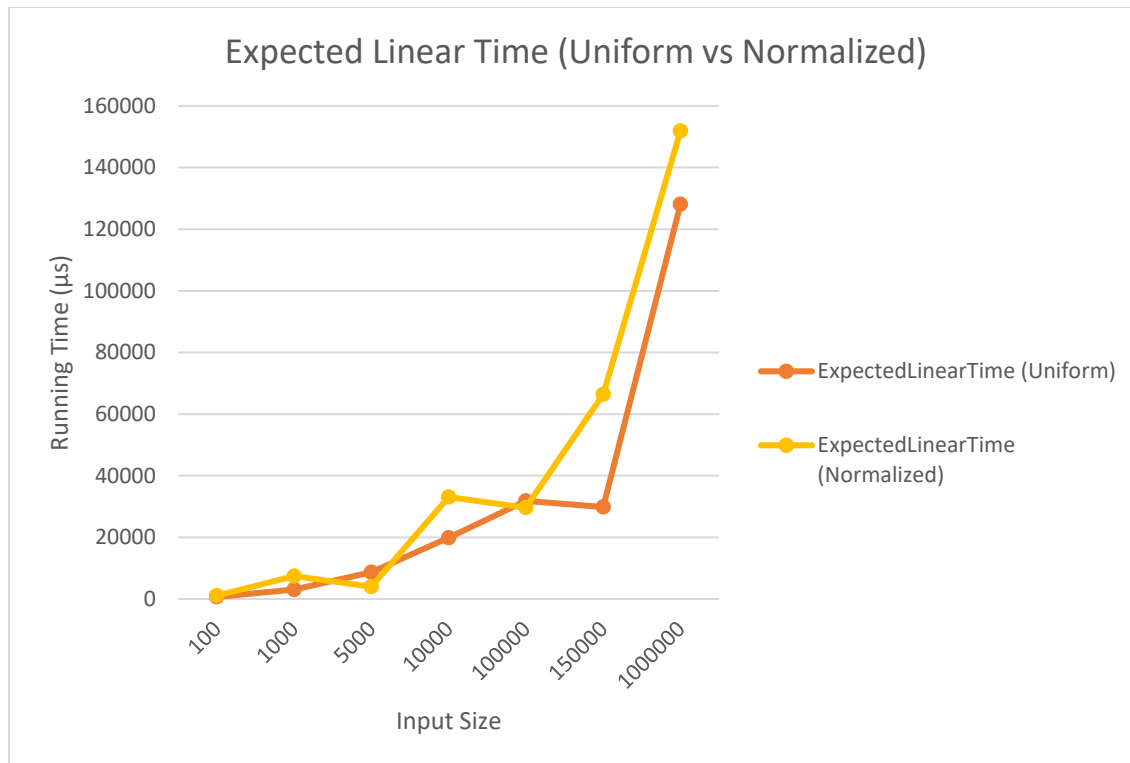
Time Complexity:

- $O(n)$

Data Structure:

- Array (Double) to store floats.
- Integer k to store value of k^{th} element
- Integer start, end, pivot to store indexes of the array.

Analysis (Graph/Comparisons):



- Above graph shows the running time of the algorithm for different input data sizes (as shown in graph) for the same value of k (100) for both types of data distribution (normalized and uniform)
- Pivot gets selected randomly, so it gets more difficult to fathom how input data distribution affects the algorithm. But, for the graph generated above, Uniform data distribution has a slow rate of growth compared to the normalized data distribution.

Sorting Algorithms

Heap Sort

Description:

Heapsort is a sorting algorithm that uses a heap type of special binary tree which is almost complete. It involves two steps, first it creates a heap of an unsorted list. Then the sorted array is created by repeatedly by removing largest element from the heap and inserting into the array. The heap is reconstructed after every removal.

Input/output:

Input from the text file containing array size in first line and unsorted input data (float values) in the following line with one input per line

Output to a file which contains the sorted array, k^{th} smallest element and top-k elements.

Functions & Algorithm:

- Sort function: This is main function where the sorting takes place.
- Random: Generates a random array in the file.
- Heapify function: This is the function to build a heap.
- Maxheap function: This function swaps the largest element in the heap.
- Swap function: This function swaps 2 numbers in the array.
- Main method: where we print the deliverables i.e. sorted array, k^{th} smallest element & top k elements.

Algorithm:

- a) Build a heap with the sorting array, using recursive insertion.
- b) Iterate to extract n times the maximum or minimum element in heap and Heapify the heap.
- c) The extracted elements form a sorted subsequence.

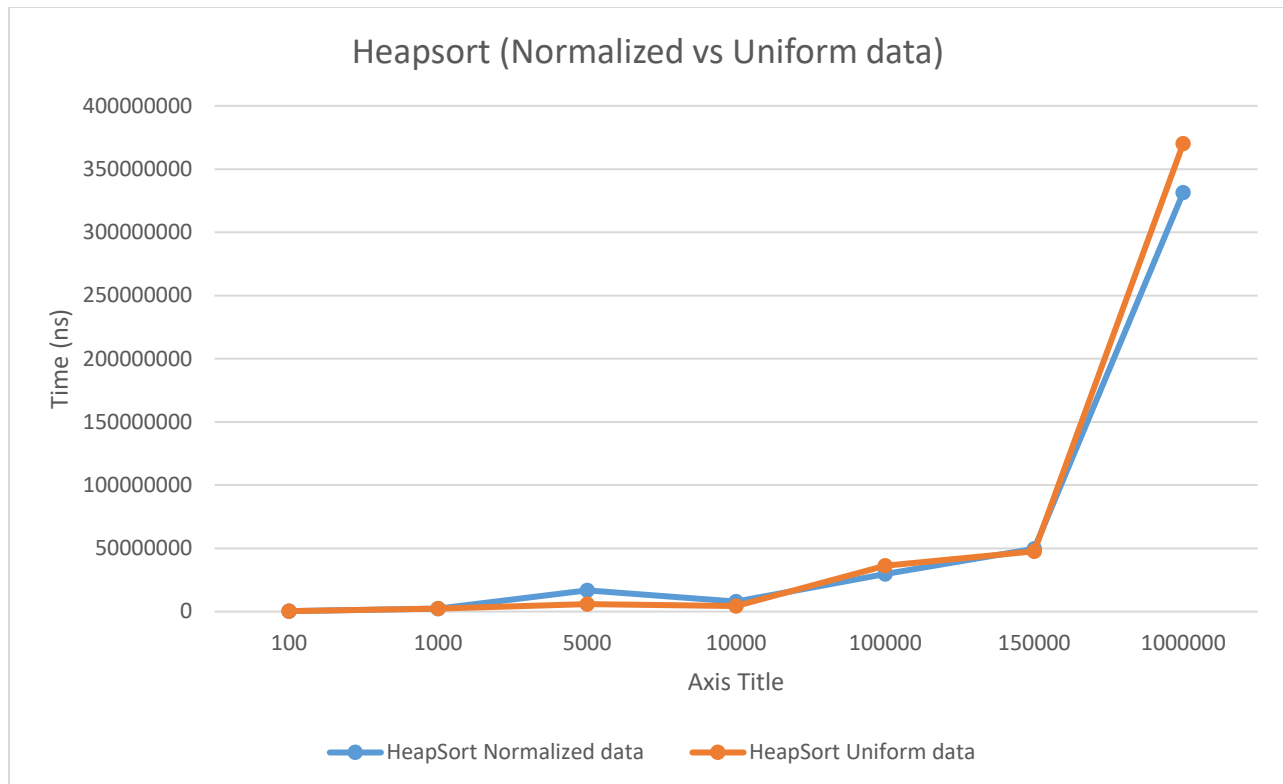
Time Complexity:

- Overall Time Complexity: $O(n \log n)$
- Heapify: $O(\log n)$
- Build Heap: $O(n)$

Data Structure:

- Arrays are used as data structure to perform the sorting.

Analysis (Graph/Comparisons):



Analysis:

- Above graph shows the running time of the algorithm for different input data sizes (as shown in graph) for both types of data distribution (normalized and uniform)
- Uniform data distribution has a slow rate of growth compared to the normalized data distribution.

Quick Sort

Description:

Quick Sort is a divide and conquer algorithm, which divides the array into smaller parts and sorts the array recursively. To perform the quick sort, a split point (pivot) needs to be selected, upon which all the elements smaller to pivot are on to the left of pivot and all elements larger than pivot are to right of pivot, which sort of creates a natural partition amongst the values. Various heuristics are applied to select the pivot for quick sort which are as bellows:

- First element as the pivot
- Random element as pivot
- Median of 3 heuristic (where 3 numbers are picked randomly and median of those is selected)

Algorithm:

Quick sort with insertion sort

If the input size is less than the value threshold value '*l*' call procedure: *insertionSort* else call procedure *quicksort*.

1. Procedure: *insertionSort* with array, start and end, index as input
 1. Take two variables first and last.
last = start + 1
 2. Repeat the steps below until last <= end
 - a. Swap last value with temp.
 - b. Store last index in first index
 - c. Swap value of first and element before first until first > start and element before first is greater than temp.
 - d. Store values in array
 3. Break from loop
1. Procedure: **quickSort** with *array*, *start index*, *last index* as input to the procedure.
 - a. Check if *start* <= *last*
False – Return pivot
True – Call procedure *splitArr* which returns the pivot index.
 - b. Compute k^{th} index ($k^{\text{th}} = \text{pivot} - \text{start} + 1$)
 - c. If k element < k^{th} index
 Recursively call left partition and right partition
 - d. Else return pivot element. ($\text{pivot} == k$ element)

2. Procedure: **splitArr** with *array*, *start index* and *last index* as input.

- a. Select random pivot.
- b. while first < last
 - True*:
 - i. Increment first till first is less than pivot
 - ii. Increment last till last is greater than pivot
 - iii. Check if (first < last)
 - True*: swap first and last
 - False*: break;
 - False*: break;
- c. Swap pivot and last element and return last index.

3. Pivot Selection:

- a. First Element as Pivot:
Pivot = start index
- b. Random Element as Pivot:
Pivot = any index between [end – start]
- c. Median of 3 Heuristics:
Select any 3 random numbers from the array index [end - start]
Find median of selected numbers
Pivot = median

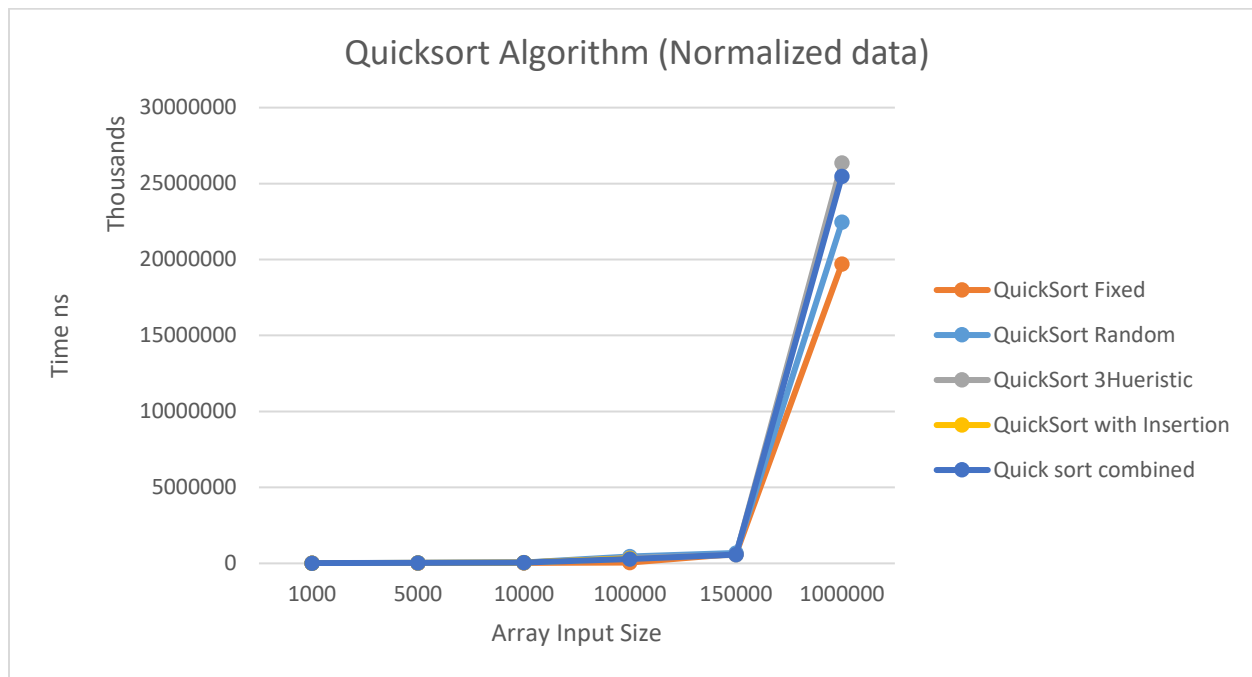
Time Complexity

- Quick Sort: $O(n^2)$
- Quick Sort with Insertion Sort: $O(nl + n \log(n/l))$

Data Structure

- Double *array* to store values
- Integer *depth* to measure stack depth
- Integer *l* to store threshold value

Analysis (Graph/Comparisons):



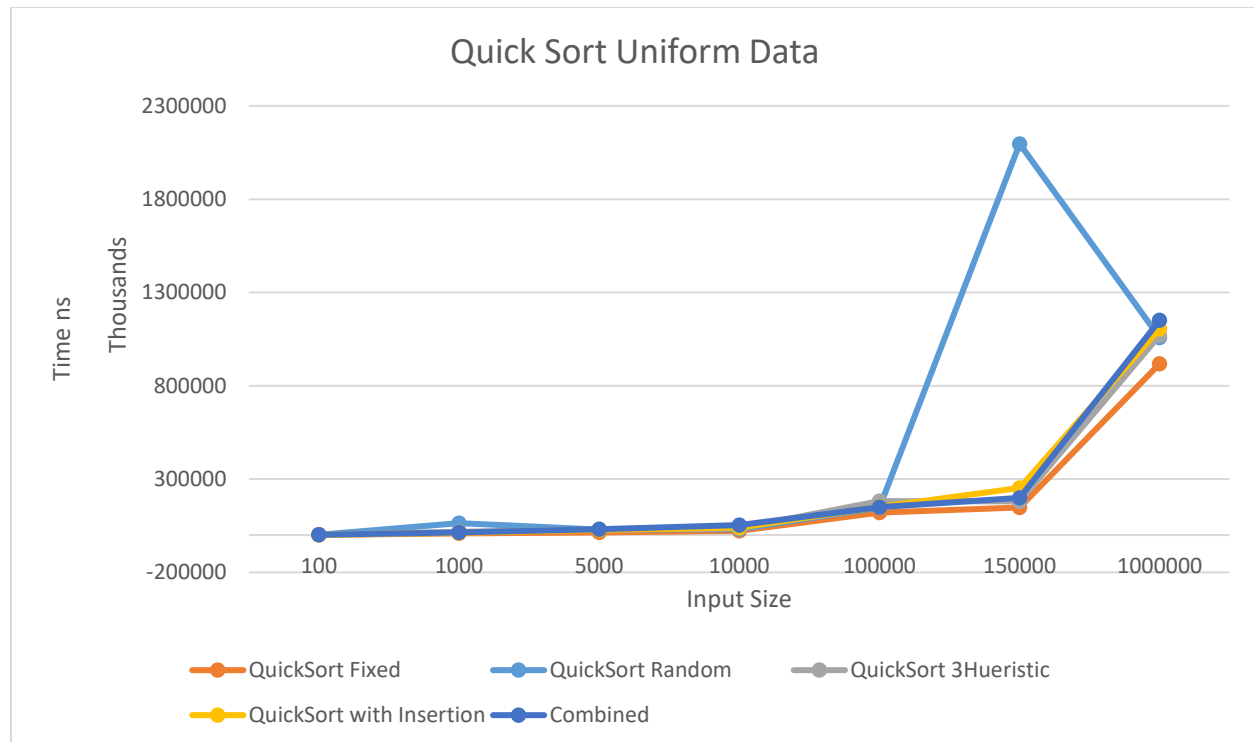
The above graph shows running time of different quick sort algorithms with normalized data as input for algorithms as shown in the graph above.

- From the observations above, Quick Sort 3 Heuristics takes the maximum running time. It is because, the pivot chosen is very random. Same is the case for Quick Sort Randomized algorithm with just slightly good performance
- Quick Sort with fixed pivot gives the most efficient output

Stack depth table for above graph:

Quick sort Stack Depth (normalized)							
Quick Sort Algorithms	Input Size						
	100	1000	5000	10000	100000	150000	1000000
Quick Sort Fixed	67	821	4743	9713	99646	149630	999582
Quick Sort Random	66	818	4738	9705	99642	149629	999583
Quick Sort 3 Heuristic	64	827	4740	9707	99648	149626	999582
Quick Sort with insertion sort	30	375	3860	8699	98348	148258	998000
Quick Sort Combined	67	822	4740	9707	99646	149628	999582

- As you can see in the table above, Quick Sort with Insertion sort has very drastic reduction in stack depth. Whereas, all the other quick sort algorithm are almost near to the worst case (n-1) stack depth



The above graph shows running time of different quick sort algorithms with uniform data as input for algorithms as shown in the graph above.

- From the observations above, Quick Sort Combined algorithm heuristics takes the maximum running time. It is because, the pivot chosen depends on the option selected while program execution. So, for all input sizes, pivot could have been practically anything, with huge random behavior. Same is the case for Quick Sort Randomized algorithm with just slightly good performance
- Quick Sort with fixed pivot gives the most efficient output for uniform data set as well.

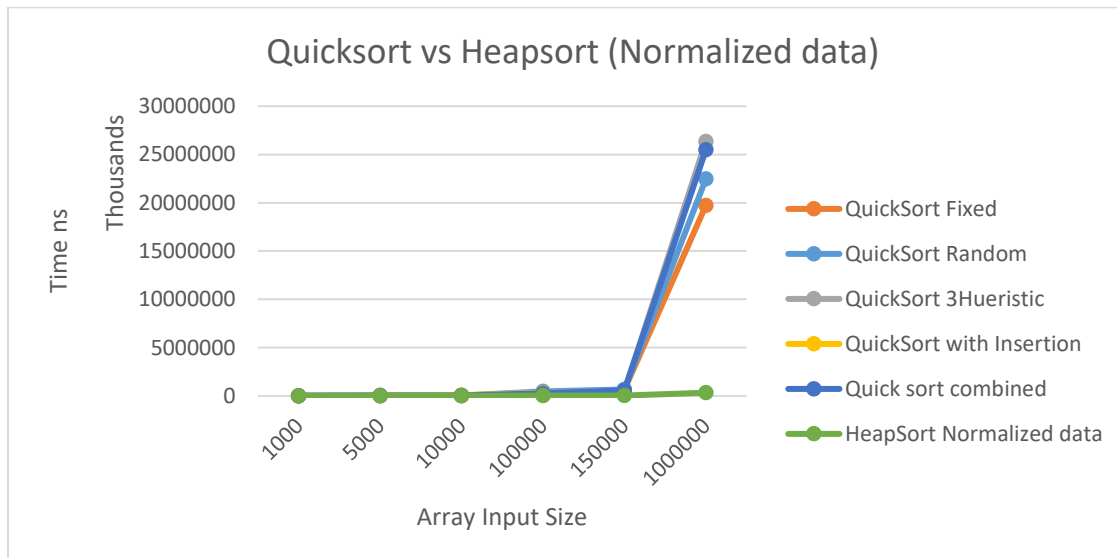
Analysis

Stack depth table for above graph

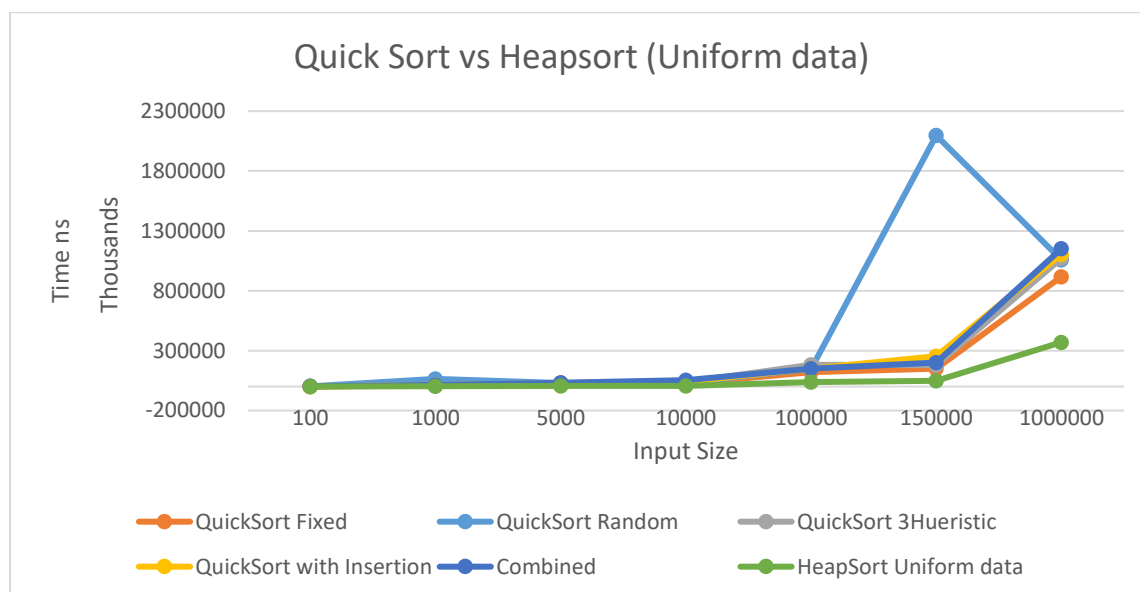
Quick sort Stack Depth (uniform)							
Quick Sort Algorithms	Input Size						
	100	1000	5000	10000	100000	150000	1000000
Quick Sort Fixed	65	666	3326	6763	90007	139999	989999
Quick Sort Random	72	665	3346	6803	90005	139999	989999
Quick Sort 3 Heuristic	67	654	3352	6788	90006	139999	989997
Quick Sort with insertion sort	29	283	1446	2863	51516	100066	949995
Quick Sort Combined	67	657	3336	6784	90004	139999	989999

- As you can see in the table above, Quick Sort with Insertion sort has very drastic reduction in stack depth. Whereas, all the other quick sort algorithm are almost near to the worst case (n-1) stack depth.

Analysis (Graph/Comparisons):

Quick Sort heuristics vs Heap Sort

- As can be seen above, heap sort works unbelievably fast than the different quick sort algorithms and for all the input sizes, it running time is most efficient for normalized data



- As can be seen above, heap sort works unbelievably fast than the different quick sort algorithms and for all the input sizes, it running time is most efficient for uniform data