CSE 5311 – Design and Analysis of Algorithms

Sorting - Project Report

Fall 2014

Done By,

Apurva M Lembi (1001045144)

Bhargavi Urs (1001050437)

Rakibul Hasan (1000672837)

Shenbagaram Kandan (1001047364)

Order statistics

1) Worst Case Linear Time Approach:

Data Structures Used:

To find the Kth smallest element using the worst case approach, we used **array** data structures to store the n elements that are fetched from the large input file. After fetching the data we grouped and sorted the data based on the group size (3, 5, 7). We sorted the data in each group using insertion sort. After grouping and sorting, we found the medians in each group by calling a method "**getMedians**()" which then returns all the medians in an **array**. Using this **medians array** as input we found the median of median which serves as a pivot. Based on the pivot value, we partitioned the entire **input array** 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.

Components :

Classses Used:

- KthSmallest

Functions Used:

- select() – Used to return the kth smallest element from the input array
- groupAndSort() – groups and sorts the entire input elements based on the group size
- insertionSort() – sorts the elements in each group
- getMedians() – gets the input array and return an array of medians
- partition() – partitions the input array based on the pivot value
- swap() – this method just swap the positions of two elements in an array
- getIndex() – it returns the index of any particular element

2) Expected Linear Time:

Data Structures Used:

In this approach, instead of grouping and sorting based on the group size, finding medians and median of medians, 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.

Components :

Classses Used:

- **R**andomizedKthSmallest

Sorting

1) Heap Sort:

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

Components :

Classses Used:

- HeapSort

Functions Used:

- buildHeap() – it builds a max heap from an unsorted array
- sort() – it sorts the given array elements
- getSortedArray() – it returns the sorted array
- swap() – this method just swap the positions of two elements in an array
- heapify() – it max heapfies the subtree rooted at the specified node
- left() – it returns the left child of the given node
- right() – it returns the right child of the given node

2) Quick Sort:

It finds 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.

Components :

Classses Used:

- QuickSort – This is the Base class for the quick sort. All other types of quick sort techniques inherits this class to implement the basic quick sort.

Functions Used:

- Sort() – it sorts the elements in the array
- partition() – partitions the input array based on the pivot value
- swap() – this method just swap the positions of two elements in an array
- getPivotIndex() – it returns the index of the pivot element based on the types of sorting methods. This is an abstract method and all the subclass implement this method to return the pivot of each sorting method

3) Classical:

Components:

Classes Used:

- **ClassicalQuickSort –** It extends QuickSort class.

Functions Used:

- **getPivotIndex()** - It returns the first element as the pivot element.

4) Randomized:

Components:

Classes Used:

- **RandomizedQuickSort –** It extends QuickSort class.

Functions Used:

- **getPivotIndex()** - It returns a random element as the pivot element.

5) Median of 3 heuristics:

Components:

Classes Used:

- **MedianHeuristicQuickSort –** It extends QuickSort class.

Functions Used:

- **getPivotIndex()** - It picks three random elements and return the median of these elements as the picot element.

6) Quicksort with insertion sort:

Components:

Classes Used:

- **InsertionQuickSort –** It extends QuickSort class.

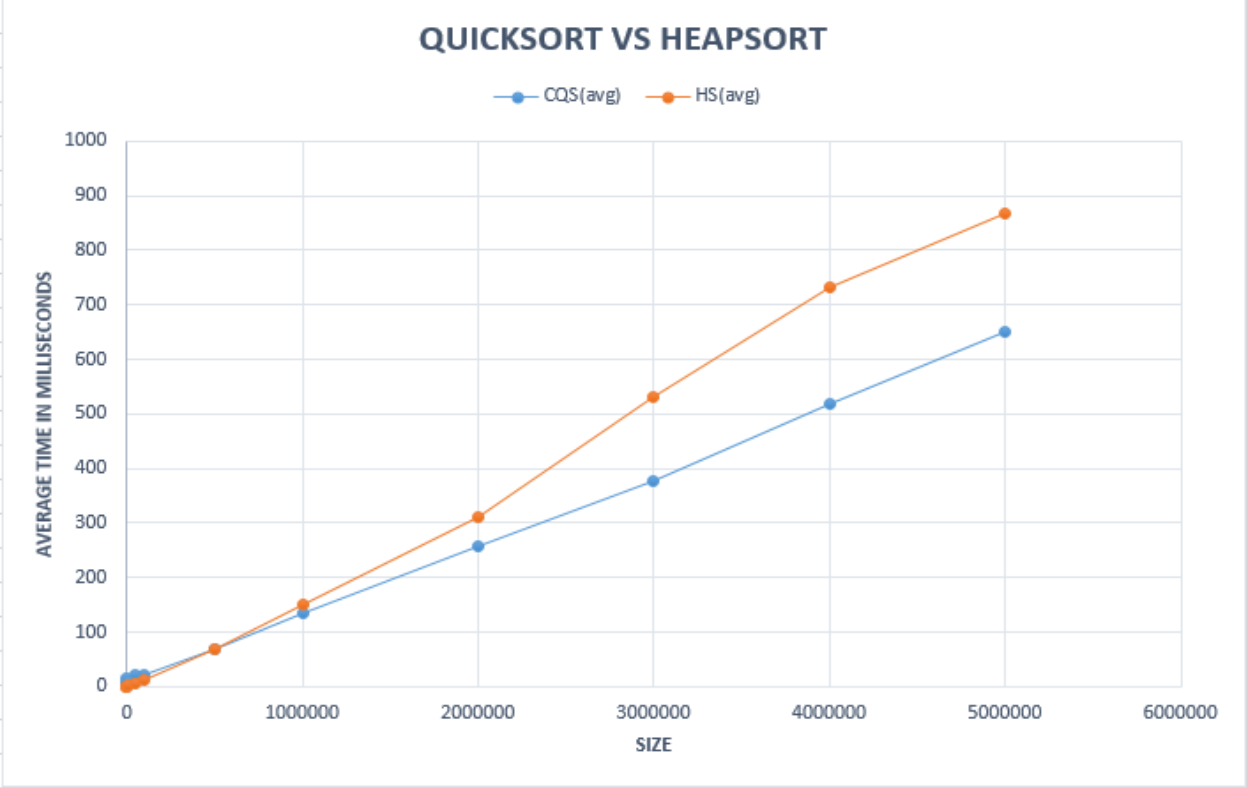
Functions Used:

- **sort() –** It overrides the parent sort class and call the insertion sort if the no of elements is less than 7 and quicksort otherwise.
- **getPivotIndex()** - It returns a random element as the pivot element.

Experimental Results:

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
- HS – Heap Sort

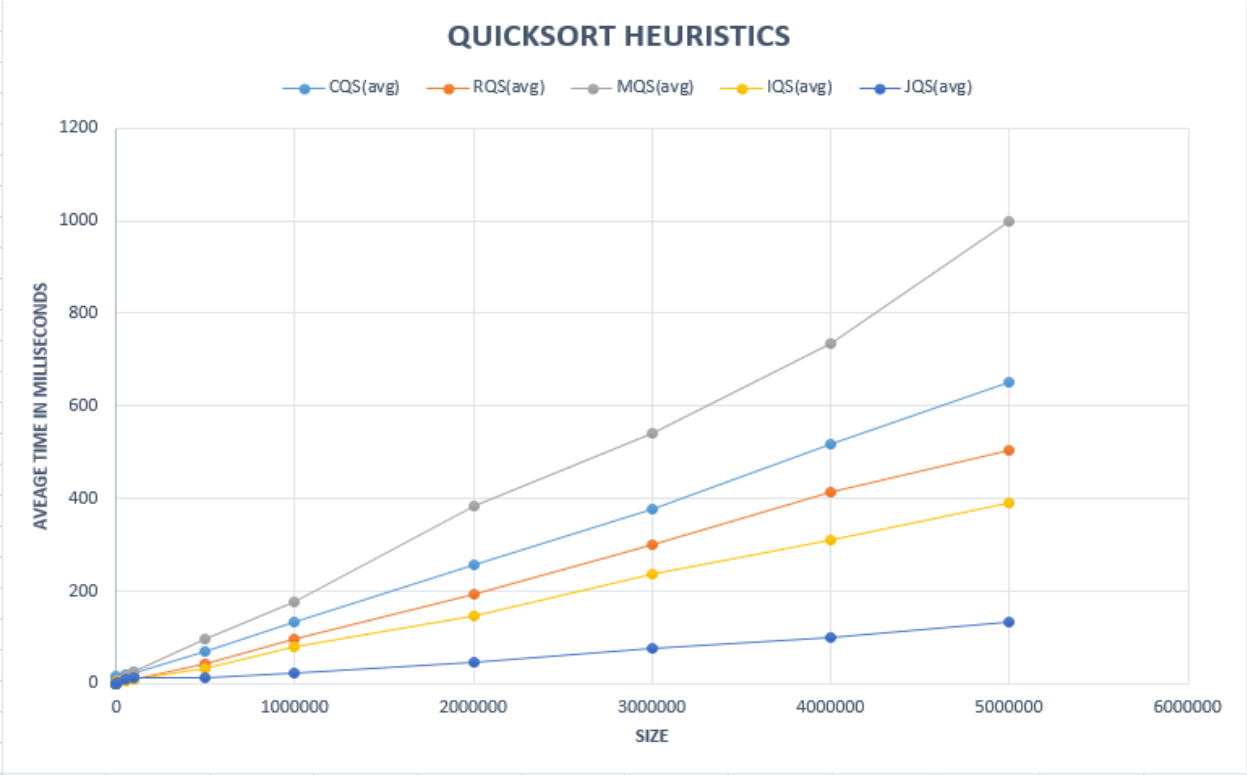


- Quicksort is slightly better than Heapsort though the worst case running time of heap sort is always O (nlogn) because Quicksort has better locality of reference i.e the next thing to be accessed is usually close in memory to the thing you just looked at. By contrast, heap sort jumps around significantly more. Since things that are close together will likely be cached together, quicksort tends to be faster.
- The heapify phase destroys the original order. The "sortdown" phase, repeatedly extracts the maximum and restores heap order, (operations in the innermost loop are simpler in quicksort)
- Heap sort in general makes more number of comparison as compared to Quick sort as elements are put at the top and are allowed to slide down
- When both algorithms have same complexity (α N log N for the quick-sort, and βN log N for the heap-sort), quick-sort is faster because it has a proportionality coefficient which equals the half

of the heap-sort's proportionality coefficient.

I . e $\alpha = \frac{\beta}{2}$

- Heap sort requires extra space (heap is nearly complete binary tree and there are pointers overhead) than quick sort as Quick sort sorts in place which don't require any extra spaces



- Randomized quicksort takes O (nlogn) no matter what the input size is. This is called Worst-case Expected-Time bound. This is always better than Classical quick sort which takes $O(n^2)$ in worst case.
- While Quicksort runs, it breaks down the list into smaller lists. Once the lists become small enough that an Insertion sort becomes more efficient than the Quicksort, it will switch to the Insertion sort to finish the job, which takes lesser time than the classical quicksort.

Size	CQS(avg)	RQS(avg)	MQS(avg)	IQS(avg)	JQS(avg)	HS(avg)
50	5.2	0	0	0	0	0
100	9.4	0	0	0	0	0
1000	16	3.2	3.2	3.2	0	0
10000	9.2	6.2	6.2	6.2	3.2	2
50000	20.4	6.2	15.8	6.2	9.4	5.6
100000	21.8	9.4	25	9	12.4	12.4
500000	69	43.2	97	34.4	12.6	68.8
1000000	133.8	97	178	78.8	22	150.4
2000000	257	193.8	384.4	146.8	47	311.2
3000000	376	300	542.4	236.2	78	532.4
4000000	518	413	736.2	309.4	100	732.6
5000000	651.6	503.2	999.8	391	134.4	867.4

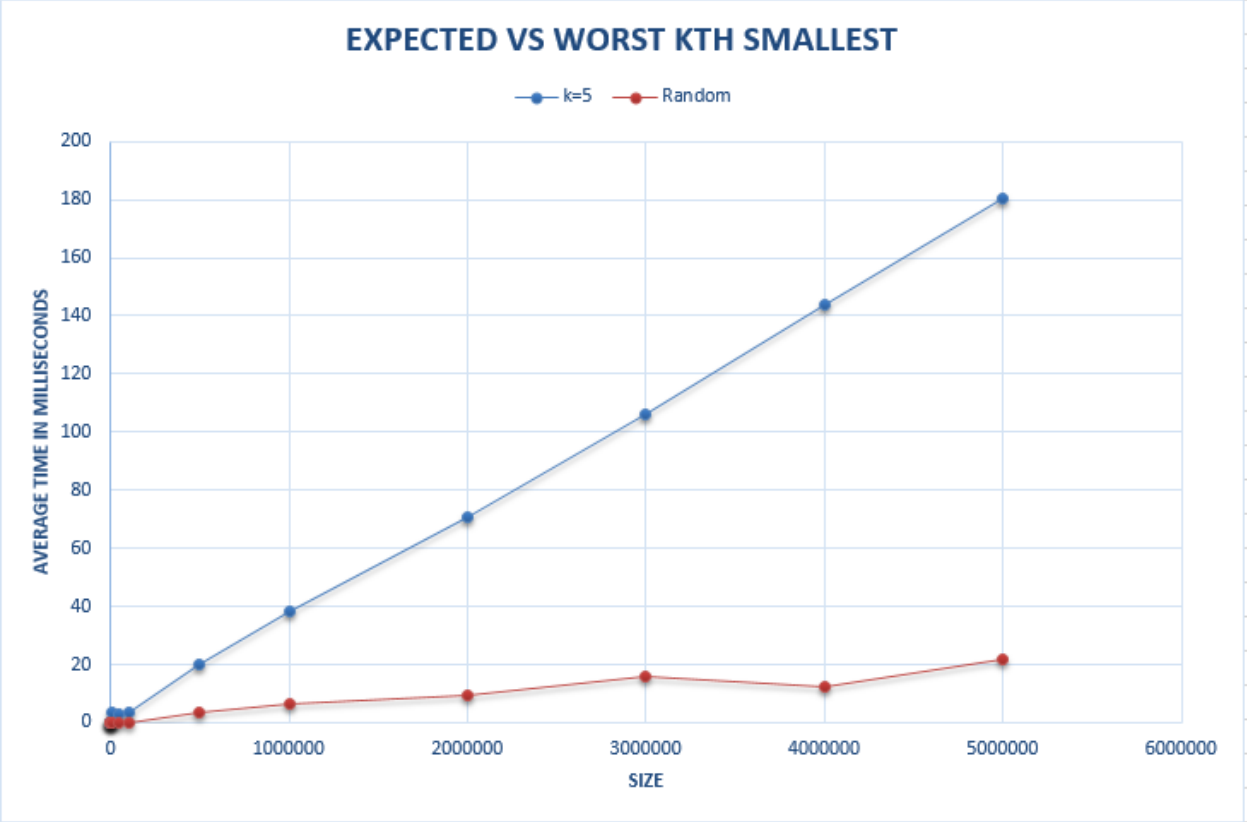
- If we divide elements into groups of 3 then we will have

  T (n) = O (n) + T (n/3) + T (2n/3) so T (n) > O (n)

- Usually grouping of 5 elements leads to optimal solution. But since we are using millions of numbers , grouping of 5 may have led to large numbers of small sets of 5 elements each .

i.e 5000000/5 = 1000000
While grouping of 7 elements has led to a smaller set of 7 elements each
i.e 50000000/7=714285 which is less than group of 5, that gave a faster running time compared to 5 and 3.

Size	k=3	k=5	k=7
50	0	0	0
100	0	0	0
1000	3	0	0
10000	3.2	3.2	0
50000	6.2	3	0
100000	6.2	3.2	3.2
500000	31.2	19.8	18.8
1000000	63.4	38.2	30.8
2000000	132.2	70.6	57
3000000	189.8	106.2	90.2
4000000	266.6	144	114.6
5000000	331	180.6	148.4



Recursion count:

Size	RQS(avg)	MQS(avg)	IQS(avg)
50	67	65.8	19.8
100	133.4	129.4	47
1000	1331	1305	443
10000	13344.2	12953	4451
50000	66658.2	64981	22256
100000	133285.4	129892	44592
500000	666811	649002	222468
1000000	1333149	1297761	444101
2000000	2667798	2596061	889219
3000000	4003188	3895015	1333531
4000000	5341585	5196002	1777770
5000000	6681789	6499506	2222730

For Gaussian Inputs