

**CSE 5311 – DESIGN AND ANALYSIS OF ALGORITHM
PROJECT REPORT
ORDER STATISTICS AND SORTING**

**Prepared for
Dr. Gautam Das,
Professor, Department of Computer Science
University of Texas at Arlington**

**Prepared by,
Arun Kumar Pokharna (1001011005)
Samuel Benison (1000995539)
Sneha Kulkarni (1001038428)
Vivek Bhalala (1001001789)**

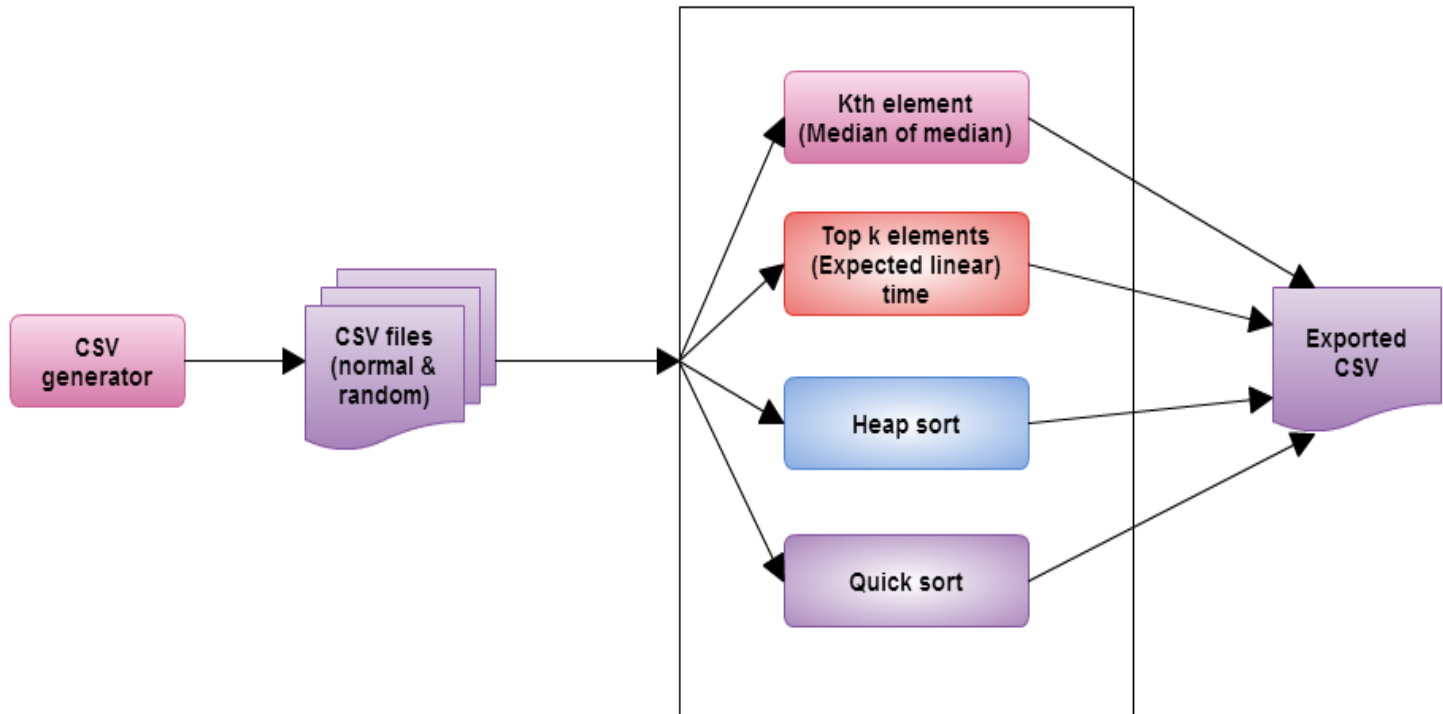
Contents

System Design	4
Median of median method	4
Description:	4
Algorithm:	4
Time Complexity	5
Data Structure	5
Analysis (Graphs/ Comparison)	6
Expected linear time	8
Description	8
Algorithm	8
Time Complexity	8
Data Structure	8
Analysis (Graphs/ Comparison)	9
Heap Sort	11
Description	11
Algorithm	11
Time Complexity	12
Data Structure	12
Analysis (Graphs/ Comparison)	12
Quick Sort	13
Description:	13
Algorithm:	13
Time Complexity:	14
Data Structure:	14
Analysis (Graphs/ Comparison) for Heap Sort and Quick Sort	15
Analysis for Heap sort and quick sort with all the pivot options for uniform random numbers ranging from 10,000 to 1,000,000	15
Analysis	15
Stack depth table for above graph.	16
Analysis for Heap sort and a combination of quick sort with all the pivot options and insertion sort with 5 chunk value for uniform random numbers ranging from 10,000 to 1,000,000	17
Analysis	17
Stack depth table for above graph.	17

Analysis for Heap sort and a combination of quick sort with all the pivot options and insertion sort with 10 chunk value for uniform random numbers ranging from 10,000 to 1,000,000	18
Analysis	18
Stack depth table for above graph.	18
Analysis for Heap sort and quick sort with all the pivot options for normal random numbers ranging from 10,000 to 1,000,000	19
Analysis	19
Stack depth table for above graph	19
Analysis for Heap sort and a combination of quick sort with all the pivot options and insertion sort with chunk value 5 for normal random numbers ranging from 10,000 to 1,000,000	20
Analysis	20
Stack depth table for above graph.	20
Analysis for Heap sort and a combination of quick sort with all the pivot options and insertion sort with chunk value 10 for normal random numbers ranging from 10,000 to 1,000,000	21
Analysis	21
Stack depth table for above graph	21
Pivot values when chosen randomly.....	22

System Design

System Design (Order Statistics and Sorting)



Median of median method

Description:

This module is named as Median of median in the figure of System design. It takes csv file (normal and uniform distribution) as input. This module extracts value of k from the csv file and start reading data from the next line. At the end of processing it will produce an output as csv file which contain k th smallest element and top k smallest elements.

Algorithm:

- Divide the array into N/C columns of elements, for small odd C .
- Find the median of each column by sorting it.
- Take only the medians and repeat steps 1-2 recursively until only one value remains. That value is picked as the pivot.

- Iterate through the array and count number of elements strictly smaller than the pivot (S), larger than the pivot (L) and equal to the pivot ($E=N-S-L$).
- If $N>K$, move all values smaller than the pivot to the beginning of the array and recursively run the whole algorithm on that sub-array.
- If $N+E>K$, conclude that Kth element is equal to the current pivot so return the pivot value and terminate the algorithm.
- Otherwise, move all values larger than the pivot to the beginning of the array and recursively run the whole algorithm on that sub-array
- Repeat above steps by reducing value of k by 1 till it becomes one to find top k smallest elements.

Time Complexity:

Kth smallest element (Median of median) = $O(n)$.

Top k smallest element = $O(k*n)$

Data Structure:

- An array of "double" to store floats elements.
- An integer variable to store value of k
- An ArrayList of double type to store sorted arrays of size $(n/3, n/5$ or $n/7)$.
- An Array of double to store numeric float values of left and right partition.

Analysis (Graphs/ Comparison):

Graph for finding top k elements for partition by 3, 5, and 7 by Median of median method.

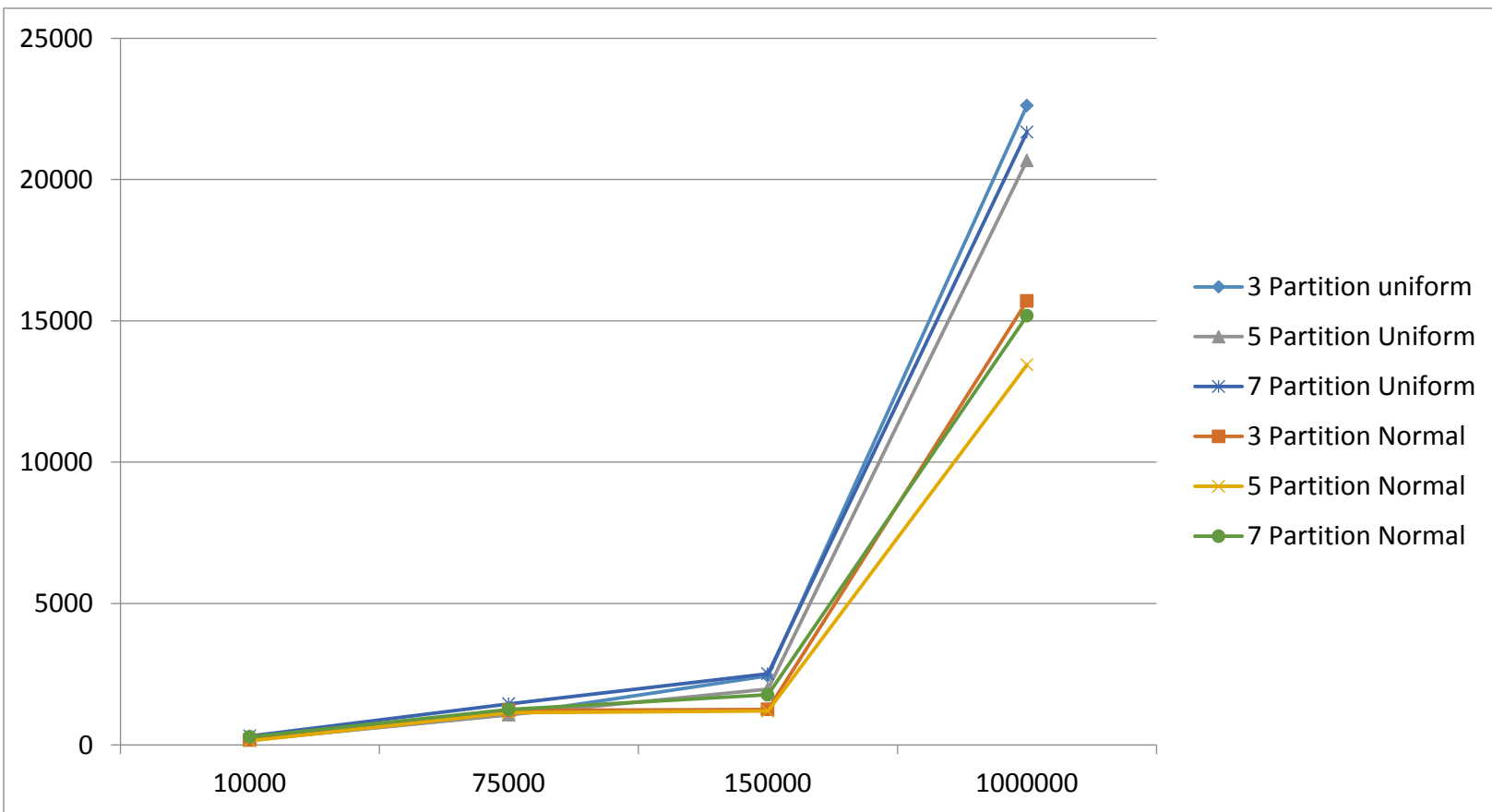


Figure: Finding top k elements by Normal and Random distribution

Analysis:

- Above graph shows the time taken to execute files of different size (10000, 75000, 150000, 1000000) for the same value of $k=200$ for **normal and uniform distribution**.
- Among each of Uniform and Random distribution it is also clear that execution time taken for partition by 3 is highest as it runs in quadratic time.
- Next comes is partition by 7 and partition by 5 takes least time for both uniform and random distribution of data.

Graph for finding Kth smallest element for partition by 3, 5, and 7 by Median of median method.

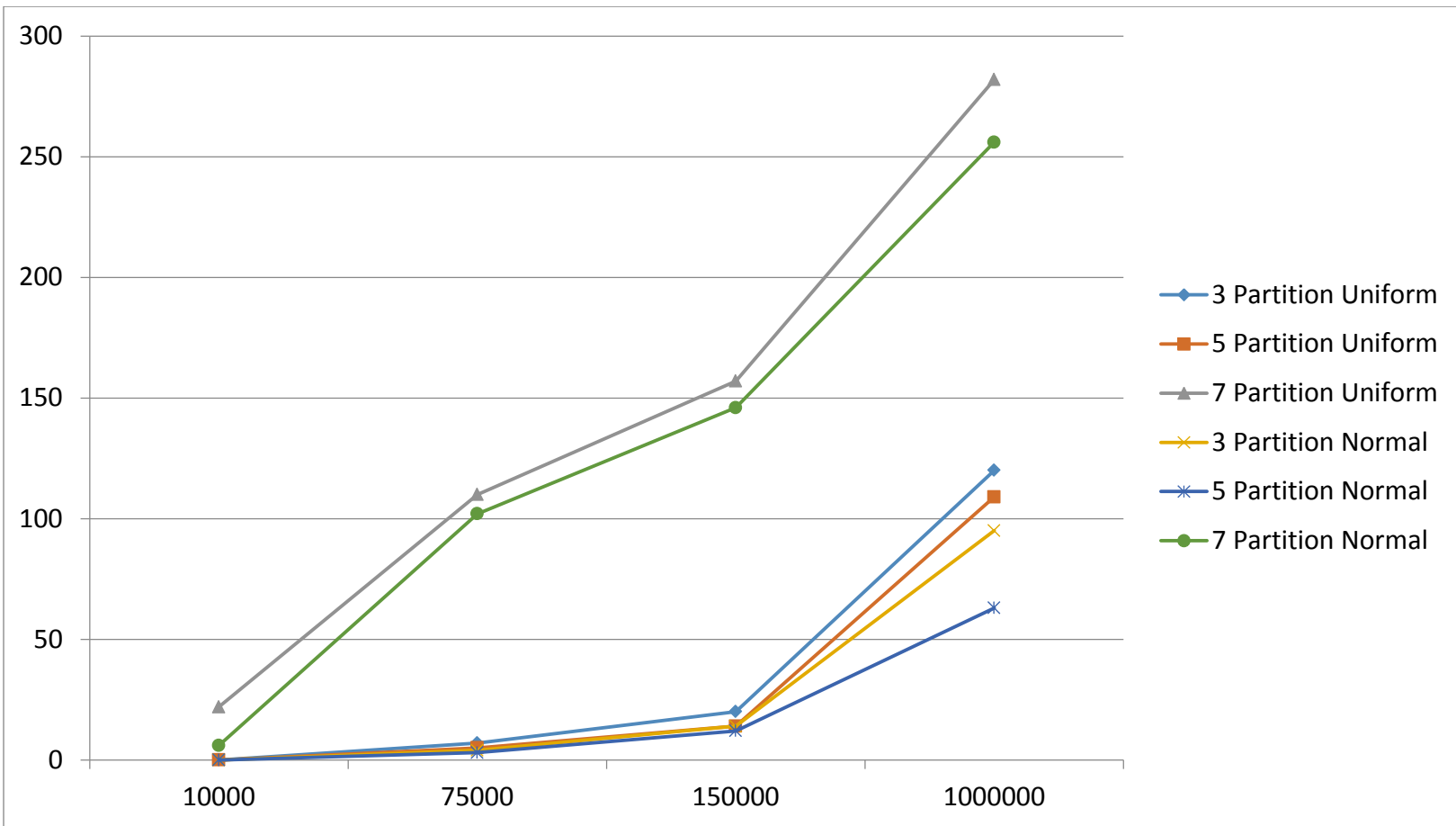


Figure: Finding Kth smallest element by Normal and Random distribution

Analysis:

- Above graph shows the time taken to execute files of different size (10000, 75000, 150000, 1000000) for the same value of $k=200$ for **normal and uniform distribution**.
- Same as previous graph among each of Uniform and Random distribution it is clear that execution time taken for partition by 3 is more as compared to partition by 5.
- The main difference here is that execution time of partition by 7 takes more time than that of partition by 3 and 5 in both normal and uniform distribution.

Expected linear time

Description:

This module is named as Expected LinearTime in the figure of System design. It takes csv file (normal and uniform distribution) as input. This module extracts value of k from the csv file and start reading data from the next line. At the end of processing it will produce an output as csv file which contain kth smallest element and top k smallest elements.

Algorithm:

1. Pick random pivot from given input.
2. Iterate through the array and count number of elements strictly smaller than the pivot (S), larger than the pivot (L) and equal to the pivot ($E=N-S-L$).
3. If $N>K$, move all values smaller than the pivot to the beginning of the array and recursively run the whole algorithm on that sub-array.
4. If $N+E>K$, conclude that Kth element is equal to the current pivot so return the pivot value and terminate the algorithm.
5. Otherwise, move all values larger than the pivot to the beginning of the array and recursively run the whole algorithm on that sub-array.
6. We can run above algorithm with increasing value of K from 1 to K. So that we will get TopK elements.

Time Complexity:

Kth smallest element = $O(n)$.

Top k smallest element = $O(k*n)$

Data Structure:

1. An array of "double" to store floats elements.
2. An integer variable to store value of k
3. An Array of double to store numeric float values of left and right partition.

Analysis (Graphs/ Comparison):

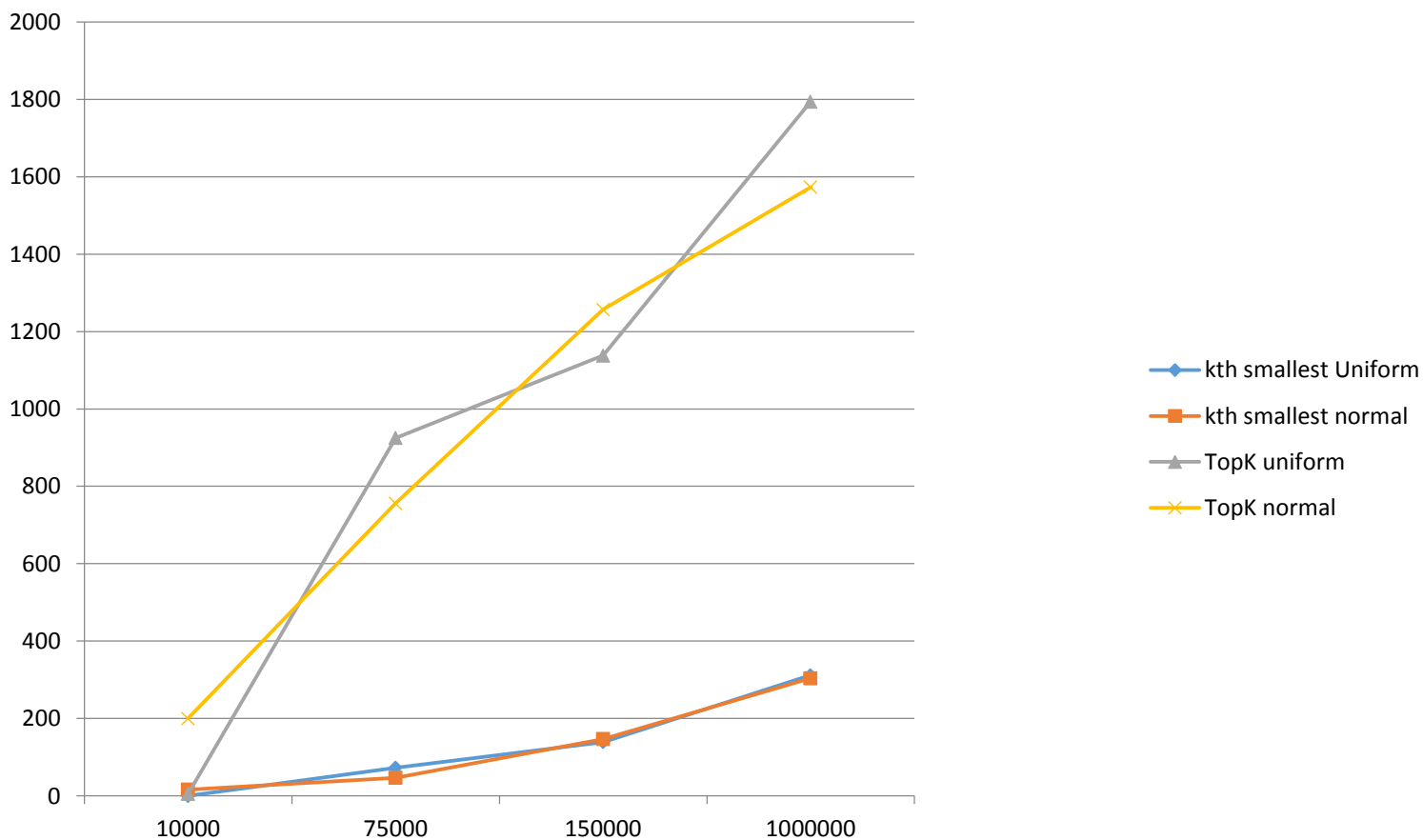


Figure: Finding Kth Smallest and Top k elements in uniform and normal distribution

Analysis:

- Above graph shows the time taken to execute files of different size (10000, 75000, 150000, 1000000) for the same value of $k=200$ for **normal and uniform distribution**.
- As we are selecting pivot randomly, it is hard to decide when will program behave well. So here we can see that sometimes uniform distribution is giving better result than normal distribution.
- Among each of Uniform and Random distribution it is also clear that execution time taken by Kth smallest is always smaller than time taken by TopK elements.

Comparison between Median of median and Expected linear time (Random pivot):

Median of median is better algorithm than selecting random pivot. Selecting pivot is very important while selecting Kth smallest or TopK. If pivot selected is good then program can behave same as median of median but If pivot is really bad, the performance can go worst. Median of median guarantees good pivot selection and ultimately performance of the algorithm.

Heap Sort

Description:

This module is named as Heap sort in the figure of System design. It takes csv file (normal and uniform distribution) as input. This module starts reading data line by line. At the end of processing it will produce an output as csv file which contain elements in sorted order.

Algorithm:

function heapSort(a, count) **is**

input: an unordered array *a* of length *count* (*first place a in max-heap order*)

heapify(a, count) end := count - 1

while end > 0

do (*swap the root(maximum value) of the heap with the last element of the heap*)

swap(a[end], a[0]) (*decrement the size of the heap so that the previous max value will stay in its proper place*)

end := end - 1 (*put the heap back in max-heap order*) siftDown(a, 0, end)

function heapify(a, count) **is** (*start is assigned the index in a of the last parent node*)

start := (count - 2) / 2

while start ≥ 0 **do** (*sift down the node at index start to the proper place such that all nodes below the start index are in heap order*)

siftDown(a, start, count-1)

start := start - 1 (*after sifting down the root all nodes/elements are in heap order*)

function siftDown(a, start, end) **is** (*end represents the limit of how far down the heap to sift*)

root := start

while root * 2 + 1 ≤ end

do (*While the root has at least one child*)

child := root * 2 + 1 (*root*2+1 points to the left child*) (*If the child has a sibling and the child's value is less than its sibling's...*)

if child + 1 ≤ end and a[child] < a[child + 1]

then child := child + 1 (*... then point to the right child instead*)

if a[root] < a[child]

then (*out of max-heap order*) swap(a[root], a[child]) root := child (*repeat to continue sifting down the child now*)

else return

Time Complexity:

Time complexity of heapify is **$O(\log n)$** .

Time complexity of createAndBuildHeap () is **$O(n)$**

Hence overall time complexity of Heap Sort is **$T(n) = O(n \log n)$** .

Data Structure:

1. Array is used for heap sort.
2. ArrayList do not work for large data. So it is not used

Analysis (Graphs/ Comparison):

(A combined comparison and analysis of Heap sort and Quick sort is performed under Quick sort description.)

Quick Sort

Description:

This module performs the quick sort operation on the input dataset. To perform the quick sort, we need to choose the pivot value upon which the dataset is divided to perform sorting. As per the requirements, the pivot value has been chosen by picking first element as the pivot, by choosing random number as the pivot, and median of three random numbers as the pivot. The experimental section discuss about the performances in detail.

Algorithm:

- Input to this algorithm includes the float input array, start index, end index, choice, and chunkValue.
- Choice variable contains the user option for choosing pivot and ChunkValue variable is used to determine Insertion sort operation.
- If the input size is less or equal to 1, return the array. (One element is always sorted)
- If the input size is less than chunkValue, run Insertion Sort with the parameters as float input array, start index, and end index.
 1. Move the current index element of the float input array to temp variable.
 2. Move the current index to another variable.
 3. Repeat the below steps until the current index is greater than start index and the previous index element is greater than temp variable.
 1. Move previous index value to current index.
 2. Reduce the current index by 1.
 4. Move the temp value to the current index of the float input array.
 5. Return the sorted float input array.
- If the input size is greater than chunkValue,
 1. Choose the pivot value.
 1. If the choice is 1, return the first element of the float input array.
 2. If the choice is 2,
 1. Generate a random number between start and end index.
 2. Return the element of that index.
 3. If the choice is 3,
 1. Generate three random numbers between start and end index.

2. Return the middle value of the three values at these random indices.
2. Partition the input array.
 1. For each value of the float input array:
 1. While the element at start index is less than pivot value.
 1. Increase start index.
 2. While the element at end index is greater than pivot.
 1. Decrease end index.
 3. If start index is less than or equal to end index.
 1. Swap the start index value with end index value.
 2. Increase start index.
 3. Decrease end index.
 2. Return start index.
3. If start index is less than the partition-1 index
 1. Repeat the steps from 1 to 5 with parameters as float input array, start index, partition-1 index, choice, chunkValue.
4. If partition index is less than end index
 1. Repeat the steps from 1 to 5 with parameters as float input array, partition index, end index, choice, chunkValue.
- Return float input array.

Time Complexity:

In general quick sort = $O(n^2)$.

Using a combination of Insertion sort with quick sort = $O(nl + n\log(n/l))$. Where l = chunkValue

Data Structure:

- An array of “double” to store floats elements.
- An integer variable to store the chunk value.
- Integer variable maxSize to store the array length.
- An integer variable stackCtr to count the stack depth for the recursive quick sort call.

Analysis (Graphs/ Comparison) for Heap Sort and Quick Sort

The algorithm has been tested for all possible combinations pivots and sorting. Below are different graphs and their execution times for different test cases.

A common observation for quick sort execution was the randomness of the pivot element. Since we are asked to choose the pivot in 3 ways in which 2 are to choose a pivot randomly, and to choose the random values, random indexes are being chosen which, at times, are resulting in choosing the same pivot value more than once. This, in turn, is resulting in the discrepancies in the execution timings for the sort.

Another common observation is that stack depth for quick sort is also varying in different cases because of the same pivot being chosen. In general the recursive calls should be $n-1$, where n is the number of input values. However, the number of calls is more than $n-1$ when only quick sort is performing. When a combination of quick sort with Insertion sort is performed, the stack depth is significantly lower than $n-1$.

Analysis for Heap sort and quick sort with all the pivot options for uniform random numbers ranging from 10,000 to 1,000,000

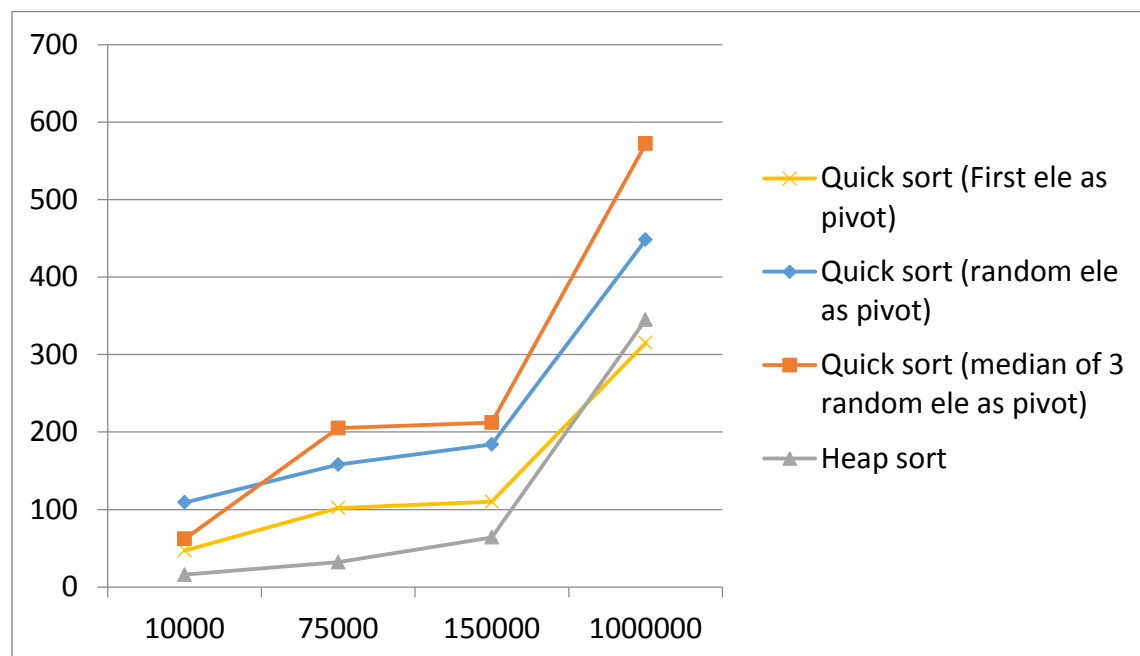


Figure: Various execution results for Heap Sort and Quick Sort

Analysis

- Above graph represents the execution times (on vertical axis) of different input sizes (on horizontal axis) for:

- Quick sort with first element as pivot.
- Quick sort with random element as pivot.
- Quick sort with median of three random elements as pivot.
- Heap Sort.
- From my observation, Quick sort with median of three random values takes highest time because of the randomness of the pivot. The same can be seen in the recursive calling of the quick sort module.
- Heap sort takes the minimum time for almost all the different input sizes.

Stack depth table for above graph.

Records	First Element as Pivot	Random number as Pivot	Median of three random number as Pivot
10000	9999	12647	12667
75000	74999	95093	95194
150000	149999	190459	189318
1000000	999999	1268087	1265750

Analysis for Heap sort and a combination of quick sort with all the pivot options and insertion sort with 5 chunk value for uniform random numbers ranging from 10,000 to 1,000,000

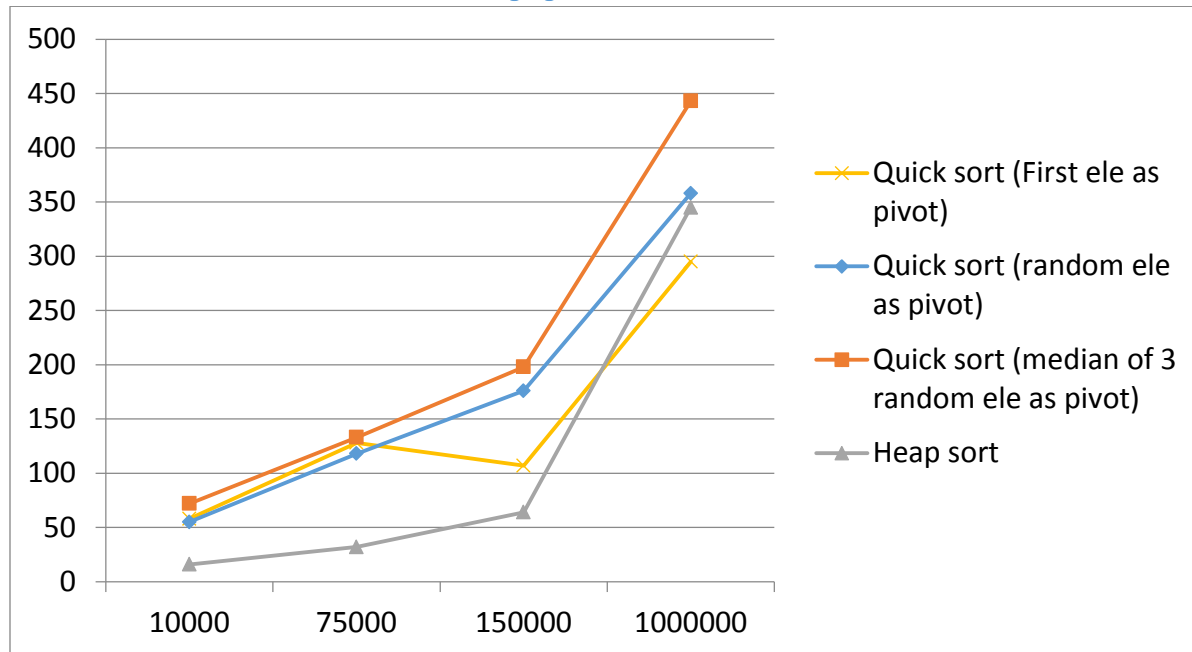


Figure: Various execution results for Heap Sort and Quick Sort-Insertion sort for chunk value 5

Analysis

- Above graph represents the execution times (on vertical axis) of different input sizes (on horizontal axis) for uniform random numbers:
 - Quick sort with first element as pivot.
 - Quick sort with random element as pivot.
 - Quick sort with median of three random elements as pivot.
 - Heap Sort.
- From my observation, Quick sort with median of three random values takes highest time because of the randomness of the pivot. The same can be seen in the recursive calling of the quick sort module.
- Heap sort takes the minimum time for almost all the different input sizes.

Stack depth table for above graph.

Records	First Element as Pivot	Random number as Pivot	Median of three random number as Pivot
10000	7090	7109	6892
75000	53267	53340	51540
150000	106484	106380	103398
1000000	710615	710261	687268

Analysis for Heap sort and a combination of quick sort with all the pivot options and insertion sort with 10 chunk value for uniform random numbers ranging from 10,000 to 1,000,000

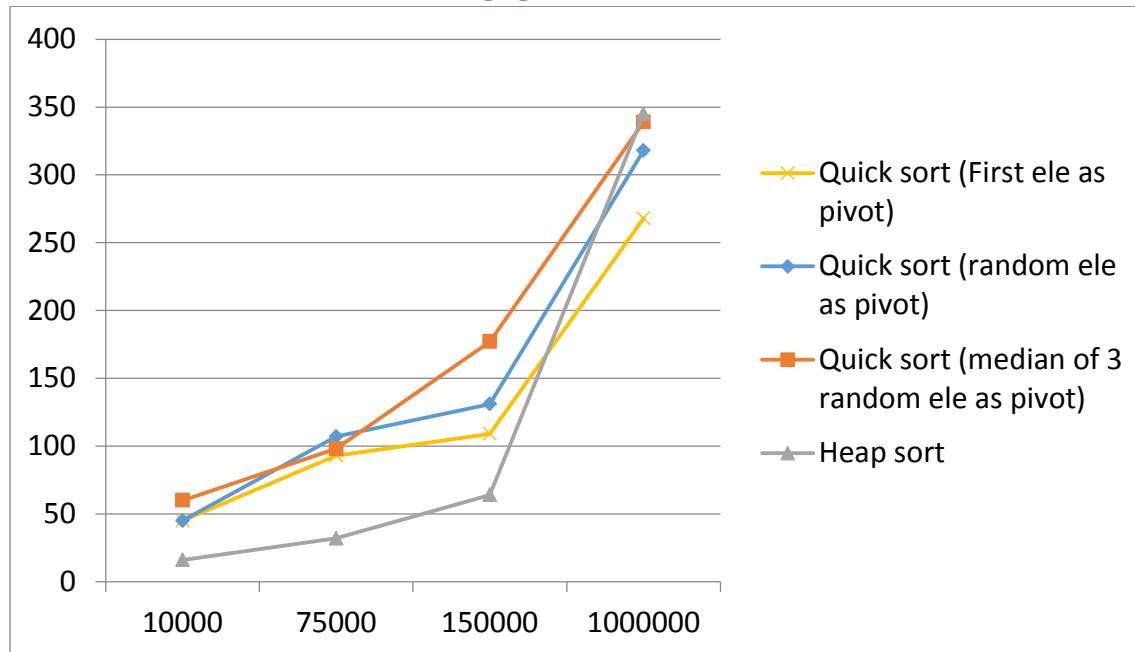


Figure: Various execution results for Heap Sort and Quick Sort-Insertion sort for chunk value 10

Analysis

- Above graph represents the execution times (on vertical axis) of different input sizes (on horizontal axis) for uniform random numbers:
 - Quick sort with first element as pivot.
 - Quick sort with random element as pivot.
 - Quick sort with median of three random elements as pivot.
 - Heap Sort.
- Due to randomness of the pivot, Quick sort with random pivot (random pivot, and median of 3 random numbers) takes higher time than quick sort with first element as pivot. The same can be seen in the recursive calling of the quick sort module.
- Heap sort takes the minimum time for almost all the different input sizes.

Stack depth table for above graph.

Records	First Element as Pivot	Random number as Pivot	Median of three random number as Pivot
10000	3873	3766	3469
75000	29654	28295	26064
150000	59074	56581	52171
1000000	393465	378507	348907

Analysis for Heap sort and quick sort with all the pivot options for normal random numbers ranging from 10,000 to 1,000,000

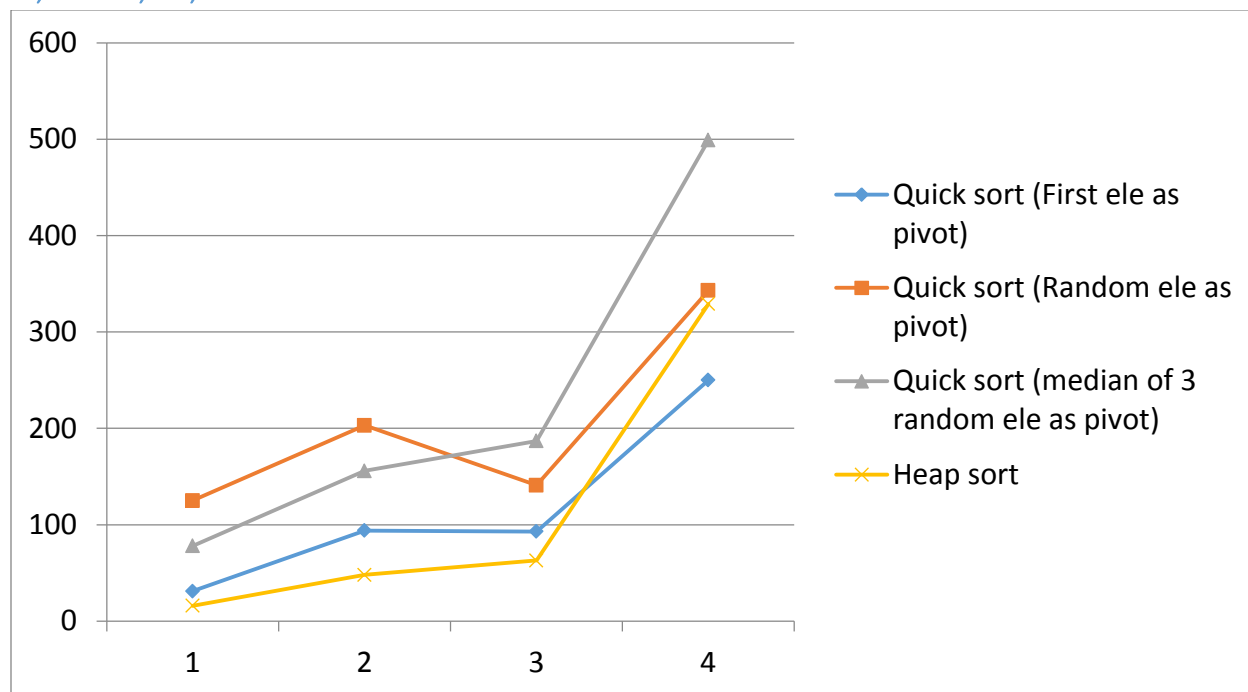


Figure: Various execution results for Heap Sort and Quick Sort

Analysis

- Above graph represents the execution times (on vertical axis) of different input sizes (on horizontal axis) for:
 - Quick sort with first element as pivot.
 - Quick sort with random element as pivot.
 - Quick sort with median of three random elements as pivot.
 - Heap Sort.
- Due to randomness of the pivot, Quick sort with random pivot (random pivot, and median of 3 random numbers) takes higher time than quick sort with first element as pivot. The same can be seen in the recursive calling of the quick sort module.
- Heap sort takes the minimum time for almost all the different input sizes.

Stack depth table for above graph

Records	First Element as Pivot	Random number as Pivot	Median of three random number as Pivot
10000	9999	10727	10805
75000	74999	76004	76024
150000	14999	151004	151020
1000000	999999	1001283	1001377

Analysis for Heap sort and a combination of quick sort with all the pivot options and insertion sort with chunk value 5 for normal random numbers ranging from 10,000 to 1,000,000

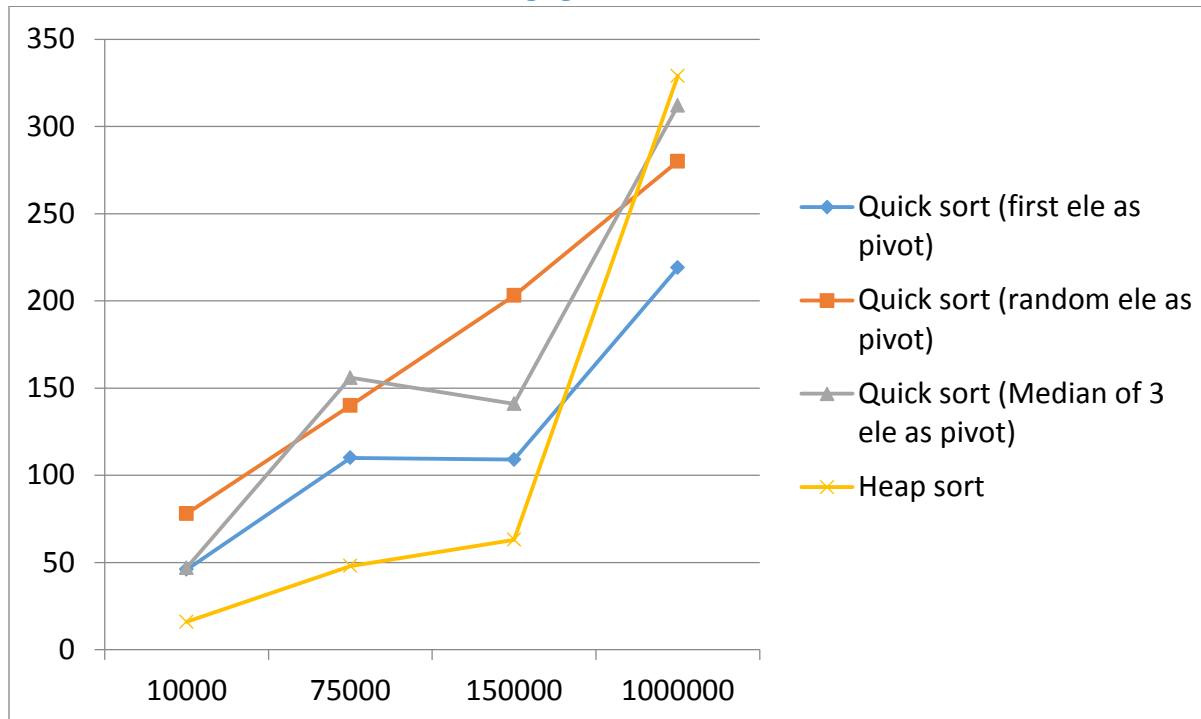


Figure: Various execution results for Heap Sort and Quick Sort-Insertion sort for chunk value 5

Analysis

- Above graph represents the execution times (on vertical axis) of different input sizes (on horizontal axis) for normal random numbers:
 - Quick sort with first element as pivot.
 - Quick sort with random element as pivot.
 - Quick sort with median of three random elements as pivot.
 - Heap Sort.
- Due to randomness of the pivot, Quick sort with random pivot (random pivot, and median of 3 random numbers) takes higher time than quick sort with first element as pivot. The same can be seen in the recursive calling of the quick sort module.
- Heap sort takes the minimum time for almost all the different input sizes.

Stack depth table for above graph.

Records	First Element as Pivot	Random number as Pivot	Median of three random number as Pivot
10000	6850	6799	6699
75000	48738	48895	48948
150000	93730	97127	97141
1000000	644053	642116	643241

Analysis for Heap sort and a combination of quick sort with all the pivot options and insertion sort with chunk value 10 for normal random numbers ranging from 10,000 to 1,000,000

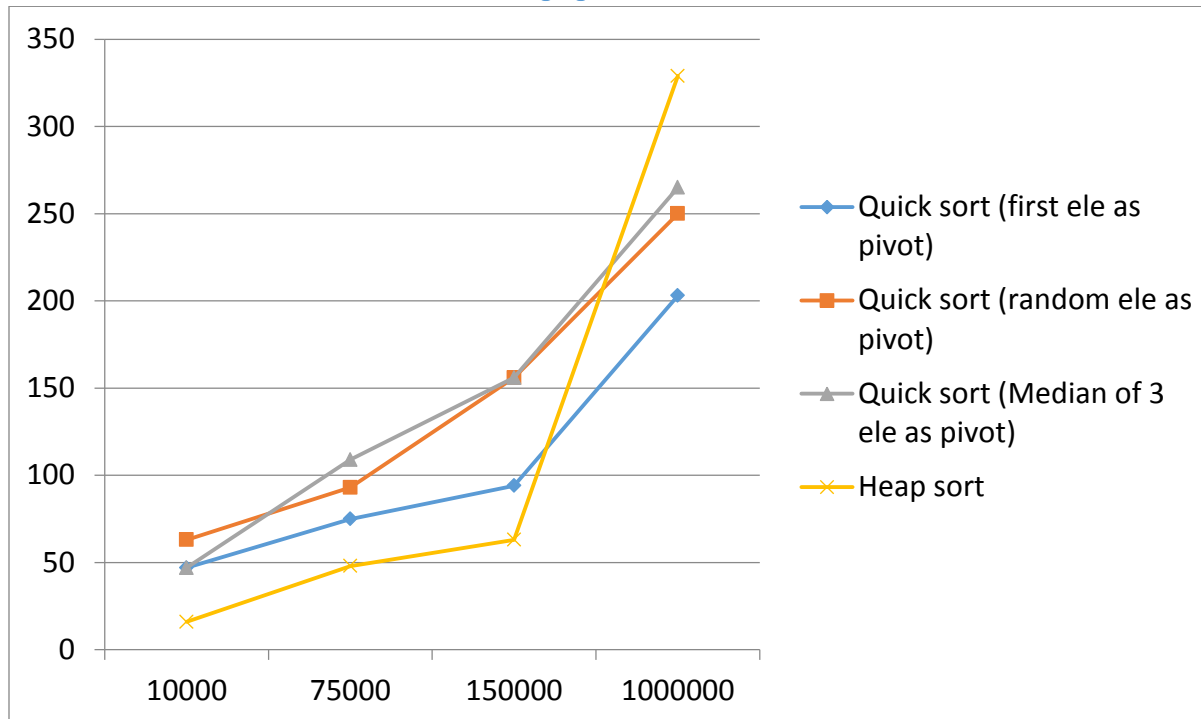


Figure: Various execution results for Heap Sort and Quick Sort-Insertion sort for chunk value 10

Analysis

- Above graph represents the execution times (on vertical axis) of different input sizes (on horizontal axis) for uniform random numbers:
 - Quick sort with first element as pivot.
 - Quick sort with random element as pivot.
 - Quick sort with median of three random elements as pivot.
 - Heap Sort.
- Due to randomness of the pivot, Quick sort with random pivot (random pivot, and median of 3 random numbers) takes higher time than quick sort with first element as pivot. The same can be seen in the recursive calling of the quick sort module.
- Heap sort takes the minimum time for almost all the different input sizes.

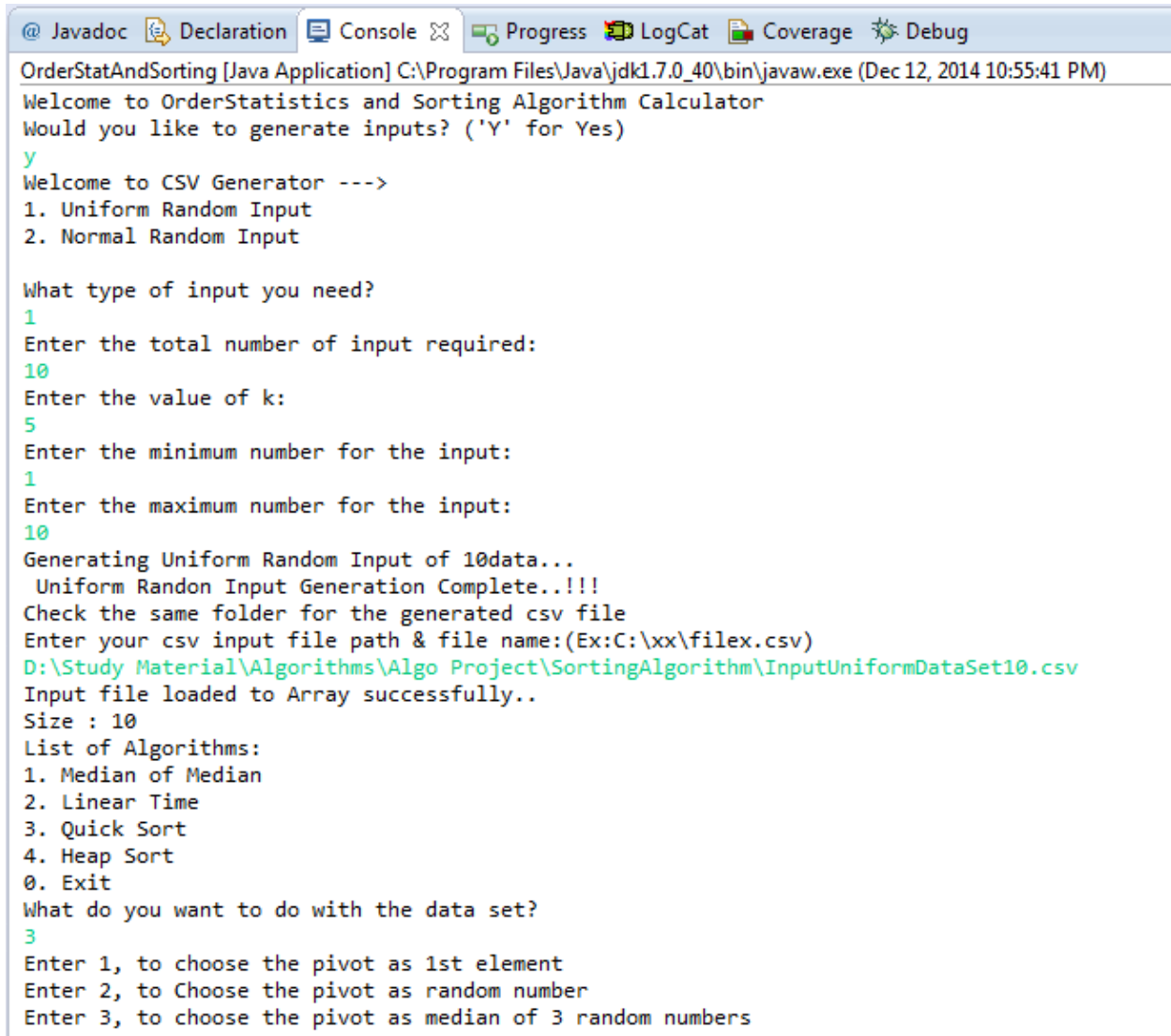
Stack depth table for above graph

Records	First Element as Pivot	Random number as Pivot	Median of three random number as Pivot
10000	3718	3511	3334
75000	23940	23760	23543
150000	46786	46643	46247
1000000	306277	305653	303942

Pivot values when chosen randomly.

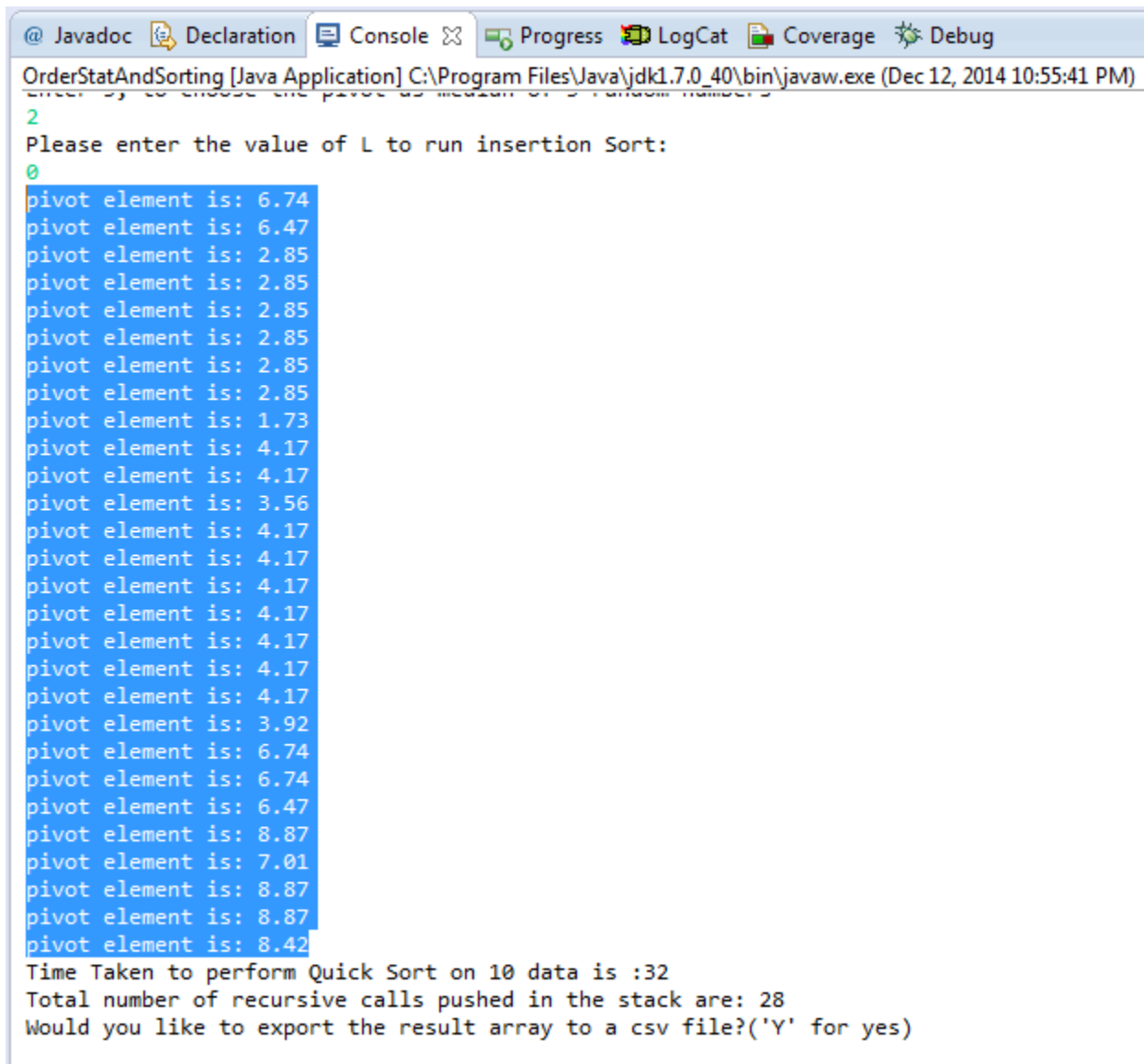
Below screenshot shows that random number is being chosen repeatedly.

When pivot is being chosen randomly



```
@ Javadoc Declaration Console Progress LogCat Coverage Debug
OrderStatAndSorting [Java Application] C:\Program Files\Java\jdk1.7.0_40\bin\javaw.exe (Dec 12, 2014 10:55:41 PM)
Welcome to OrderStatistics and Sorting Algorithm Calculator
Would you like to generate inputs? ('Y' for Yes)
y
Welcome to CSV Generator --->
1. Uniform Random Input
2. Normal Random Input

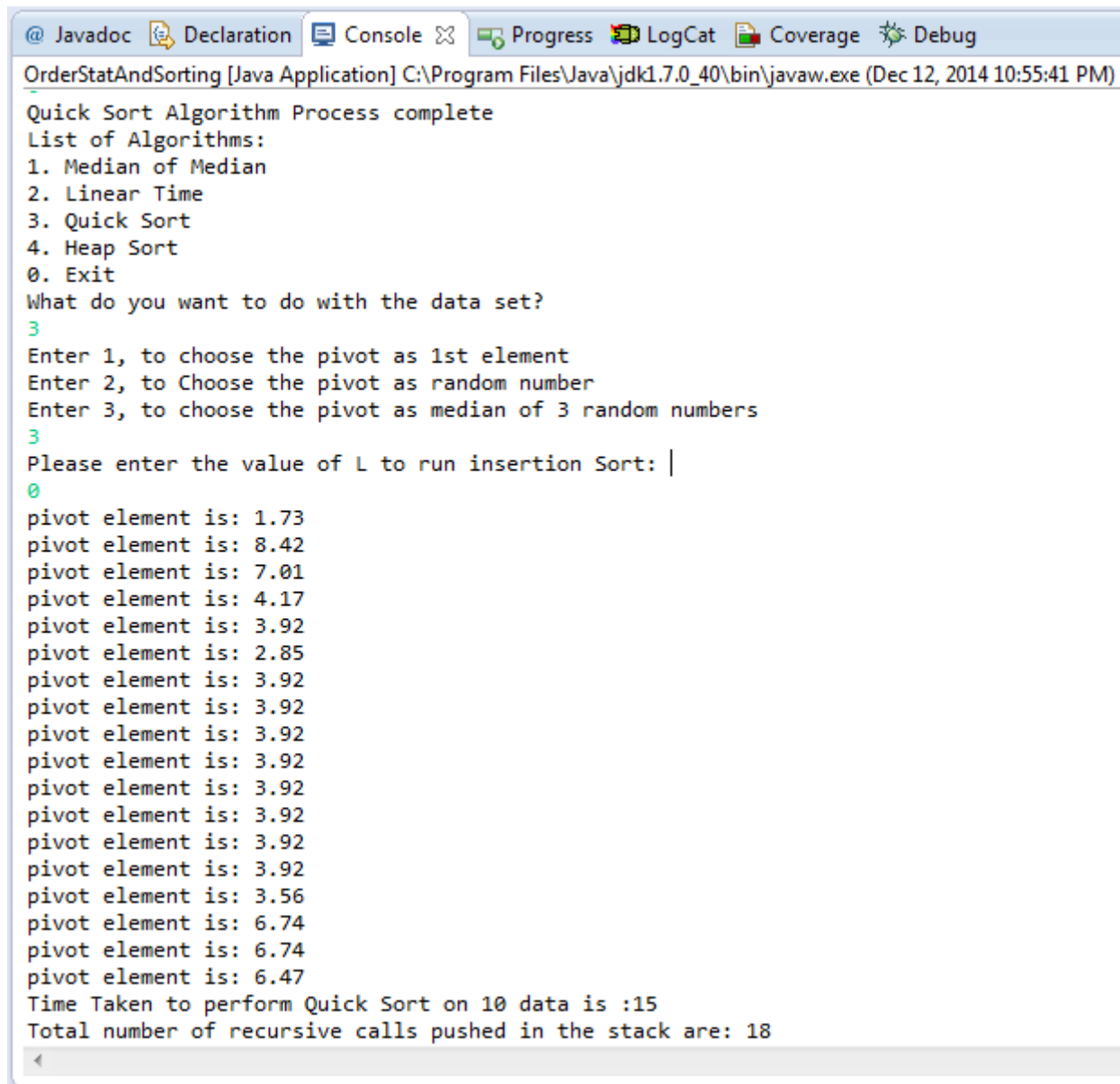
What type of input you need?
1
Enter the total number of input required:
10
Enter the value of k:
5
Enter the minimum number for the input:
1
Enter the maximum number for the input:
10
Generating Uniform Random Input of 10data...
Uniform Random Input Generation Complete..!!!
Check the same folder for the generated csv file
Enter your csv input file path & file name:(Ex:C:\xx\filex.csv)
D:\Study Material\Algorithms\Algo Project\SortingAlgorithm\InputUniformDataSet10.csv
Input file loaded to Array successfully..
Size : 10
List of Algorithms:
1. Median of Median
2. Linear Time
3. Quick Sort
4. Heap Sort
0. Exit
What do you want to do with the data set?
3
Enter 1, to choose the pivot as 1st element
Enter 2, to Choose the pivot as random number
Enter 3, to choose the pivot as median of 3 random numbers
```



The screenshot shows a Java IDE with the 'Console' tab active. The application is titled 'OrderStatAndSorting [Java Application]' and is running at 'C:\Program Files\Java\jdk1.7.0_40\bin\javaw.exe (Dec 12, 2014 10:55:41 PM)'. The console output shows the user entering '2' for the value of L to run insertion sort. A list of 20 pivot elements is displayed, followed by the time taken to perform Quick Sort (32) and the total number of recursive calls (28). The user is prompted to export the result array to a CSV file.

```
@ Javadoc Declaration Console Progress LogCat Coverage Debug
OrderStatAndSorting [Java Application] C:\Program Files\Java\jdk1.7.0_40\bin\javaw.exe (Dec 12, 2014 10:55:41 PM)
2
Please enter the value of L to run insertion Sort:
0
pivot element is: 6.74
pivot element is: 6.47
pivot element is: 2.85
pivot element is: 2.85
pivot element is: 2.85
pivot element is: 2.85
pivot element is: 2.85
pivot element is: 2.85
pivot element is: 1.73
pivot element is: 4.17
pivot element is: 4.17
pivot element is: 3.56
pivot element is: 4.17
pivot element is: 4.17
pivot element is: 4.17
pivot element is: 4.17
pivot element is: 4.17
pivot element is: 4.17
pivot element is: 4.17
pivot element is: 3.92
pivot element is: 6.74
pivot element is: 6.74
pivot element is: 6.47
pivot element is: 8.87
pivot element is: 7.01
pivot element is: 8.87
pivot element is: 8.87
pivot element is: 8.87
Time Taken to perform Quick Sort on 10 data is :32
Total number of recursive calls pushed in the stack are: 28
Would you like to export the result array to a csv file?('Y' for yes)
```

When pivot is chosen through median of three random numbers



```
@ Javadoc Declaration Console Progress LogCat Coverage Debug
OrderStatAndSorting [Java Application] C:\Program Files\Java\jdk1.7.0_40\bin\javaw.exe (Dec 12, 2014 10:55:41 PM)

Quick Sort Algorithm Process complete
List of Algorithms:
1. Median of Median
2. Linear Time
3. Quick Sort
4. Heap Sort
0. Exit
What do you want to do with the data set?
3
Enter 1, to choose the pivot as 1st element
Enter 2, to Choose the pivot as random number
Enter 3, to choose the pivot as median of 3 random numbers
3
Please enter the value of L to run insertion Sort: |
0
pivot element is: 1.73
pivot element is: 8.42
pivot element is: 7.01
pivot element is: 4.17
pivot element is: 3.92
pivot element is: 2.85
pivot element is: 3.92
pivot element is: 3.92
pivot element is: 3.92
pivot element is: 3.92
pivot element is: 3.92
pivot element is: 3.92
pivot element is: 3.92
pivot element is: 3.56
pivot element is: 6.74
pivot element is: 6.74
pivot element is: 6.47
Time Taken to perform Quick Sort on 10 data is :15
Total number of recursive calls pushed in the stack are: 18
```

In both the scenarios, the chosen pivot element is being repeated. The stack counter is also shown to prove that the number of recursive calls are higher than $n-1$.