MARMARA UNIVERSITY

THE FACULTY OF ENGINEERING

CSE2046

Analysis of Algorithms

PROJECT 1

Comparison of Algorithms for Finding k’th Smallest Element

Submitted to: Assoc. Prof. Dr. Ömer Korçak

Due Date : 11.05.2022

Ömer Kibar - 150119037

Sinan Göçmen - 150120519

1. Contribution

Mainly, all steps are studied cumulatively in several online and face-to-face meetings. In specific, implementation of input generation and algorithms, generating data and testing are mostly done by Ömer Kibar and report writing, collocating graphs & charts and analysis of algorithms are mostly done by Sinan Göçmen. After one of us finished one of the tasks, the other one checked the work, did some corrections, and completed it.

2. Purpose

The purpose of this project is to compare a series of algorithms for the given problem which is ‘‘finding the k’th smallest element in a list of n numbers’’.

3. Procedure

The procedure is covered in the same way with the guideline. According to the guideline, the project has three main steps and each step proceed as follows :

- **Designing the Experiment**

  a) We want to explore the time complexities of algorithms and compare them. So, we test each algorithm with different input sizes and different k values;

  - 4 variations for the size of n : 1000, 2000, 3000, 4000

  - 3 variations for the k : k=1; k=n; and k = n/2 (median)

  Also, we have different variations of input arrays for each size to explore if the algorithm depends on the input characteristic or not. We choose characteristics by considering the ‘level of being sorted’. So, there are:

  - 5 variations for the characteristics of the input array(sorted, almost sorted, random, almost reversely sorted, reversely sorted).

  b) For complexity measurement, we decided to choose counting the number of basic operations. The reason behind that is to escape from the instant performance changes of the computer. In this way, we get more reliable results and we can just focus on the effect of changing k, size of n and characteristic of the input array.

- **Coding & Running**

  In this project, we used Java as the programming language. We created our own input generation class. With this class, we could easily create our sample inputs in any size and any characteristic. By using this class, we created 4 input files for each input type which contains 500 arrays having the same characteristic. So, we have 20 input files (10.000 arrays) in total to test our algorithms.

We implemented algorithms into two classes when one of them finds the k’th element in an array and the other one is modified to return the number of basic operations. Also, we have our Test class which is testing the chosen algorithm for the input files having different sizes & characteristics and prints the results for each k value to the output file. After testing, we get the average of 500 results and store all the data in the excel sheets.

- **Illustrating & Analyzing Results**
  a) The effectiveness of each algorithm will be analyzed for the variations mentioned. Also, by using Excel program, the results will be supported with plots & charts. Then, it will be determined if the results meet the theoretical expectations or not.
  b) After proceeding (a) for each algorithm, the big picture will be looked at and the algorithms will be compared.

4. Empirical Analysis of Algorithms

Presorting Algorithms

In presorting algorithms, we sort the array and return the k’th element of array. So, the time taken by these algorithms depends on the speed of the sorting algorithm, not on the value of k. Now, we will analyze these types of algorithms and learn about their time complexities.

1. Insertion Sort based Algorithm

In the insertion sort algorithm, we chose the basic operation as comparison. The results are given below.

graph-1 : analyzing for different input characteristics

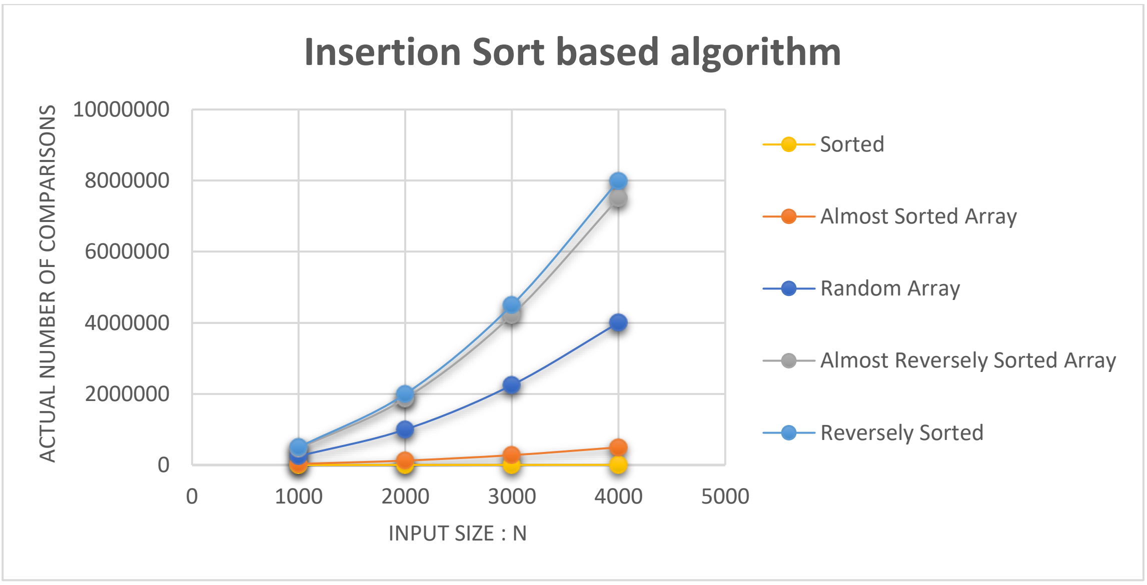


table-1 : analyzing for different input characteristics

Expectations		Experimental Results				
Input size (n)	n^2	Sorted	Almost Sorted	Random	Almost Reversely Sorted	Reversely sorted
1000	1.000.000	999	30.879	249.062	468.264	499.450
4000	16.000.000	3999	496.468	4.001.493	7.504.893	7.997.798
x4	x16	x4,003	x16,078	x16,066	x16,027	x16,013

Based on these results, we see that the best case is that the array is already sorted. In this case, the growth rate of the number of comparisons is almost equal to the growth rate of the input size (n). Other cases have a quadratic growth rate. Even if the array is almost sorted, it also grows quadratically. Besides that, in random, almost reversely sorted and reversely sorted types, the growth rate is also quadratic.

In conclusion, the time complexity of insertion sort algorithm is $O(n)$ in the best case when the array is already sorted. In average and worst cases, it is $O(n^2)$. So, the results meet with the theoretical expectations.

Analyzing for Different k values

Theoretically, the k value does not affect the number of basic operations. In our experiment, because we chose the basic operation as comparison, our experiments with different k values comes out with the identical results.

2. Merge Sort based Algorithm

In the merge sort, the basic operation is also comparison as in the insertion sort. it increases in the merge operation.

graph-2 : analyzing for different input characteristics

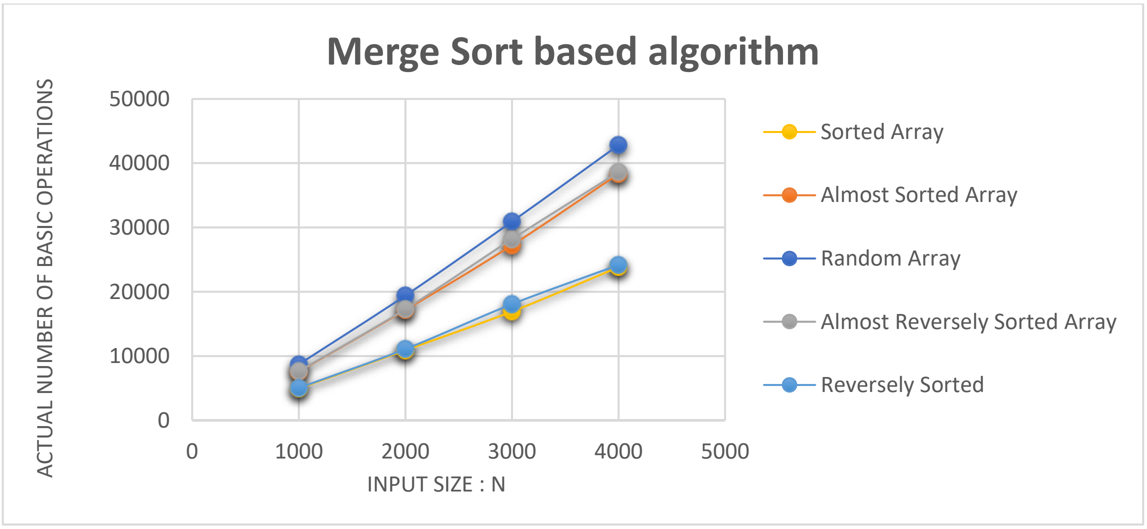


table-2 : analyzing for different input characteristics

Expectations		Experimental Results				
Input size (n)	$nlog_2n$	Sorted	Almost Sorted	Random	Almost Reversely Sorted	Reversely sorted
1000	9.965,8	4957	7580	8707	7640	5044
4000	47.863,1	23.828	38.350	42.820	38.624	24.176
x4	x4,803	x4,807	x5,059	x4,918	x5,056	x4,793

As it is seen in the graph, if the array is already sorted or close to it, the number of comparisons decreases. But, the important part is that they all have the same rate when the input size grows. Also, in the table, we see that the difference between the results are not that much. The time complexity of merge sort is $O(nlogn)$ in all cases which means these results also meet with the theoretical expectations.

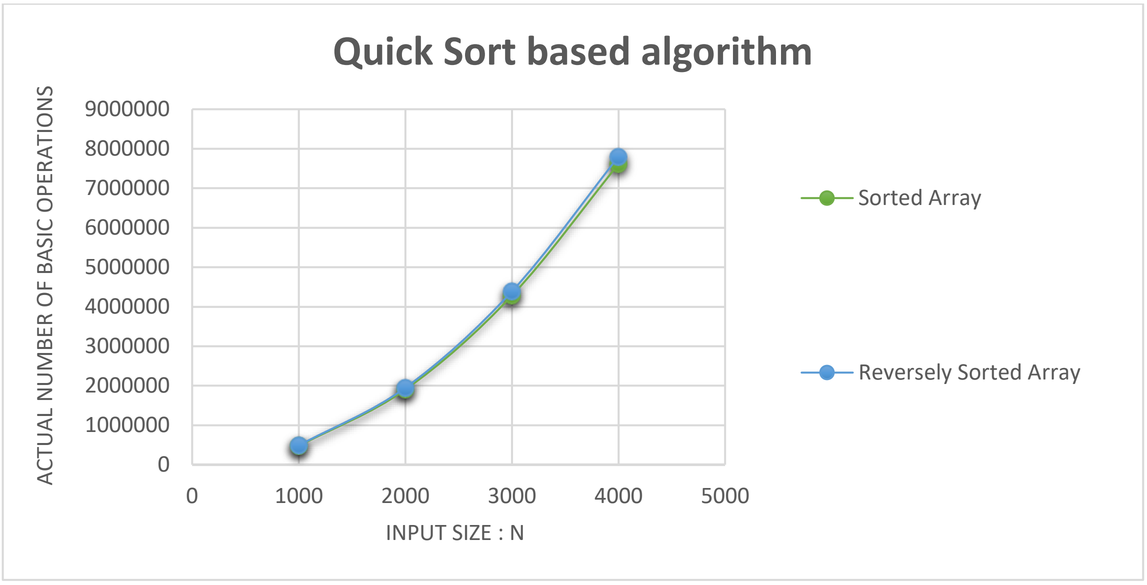
Analyzing for Different k values

Similar to insertion sort, when we change the value of k in our experiments, we saw that the results are identical. Because the different is just to change the nature of sorting, this result was not a surprise to us. Again, it also meets with the theoretical expectations.

3. Quick Sort based Algorithm

In this algorithm, the partition is used. We chose first element as pivot in the partition part.

graph-3 : analyzing for different input characteristics



graph-4 : analyzing for different input characteristics

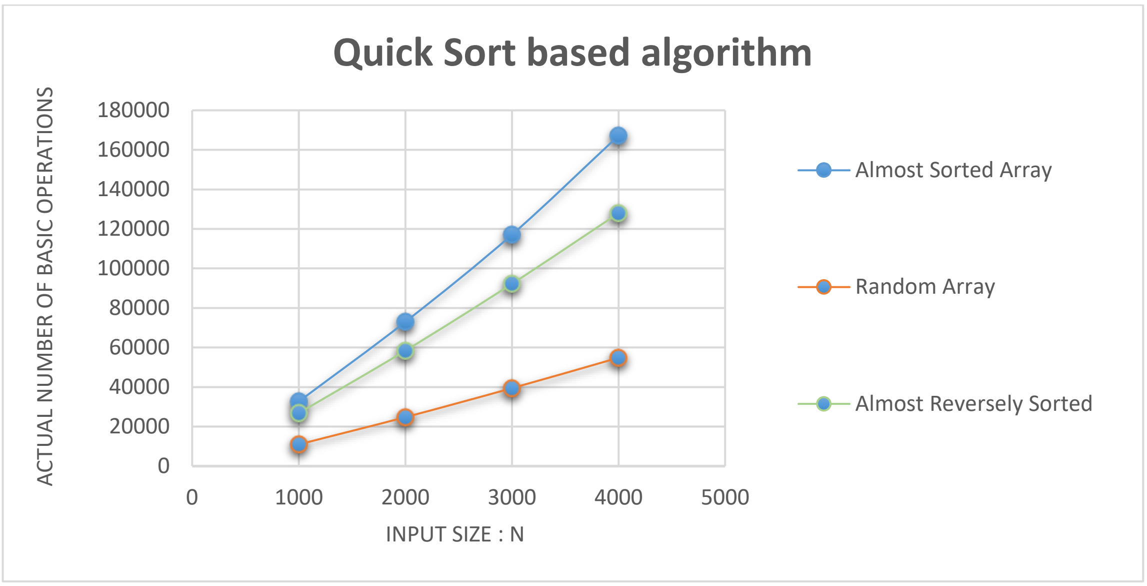


table-3 : analyzing for different input characteristics

Expectations		Experimental Results				
Input size (n)	$nlog_2n$	Sorted	Almost Sorted	Random	Almost Reversely Sorted	Reversely sorted
1000	9.965,8	475.491	32.671	10.979	26.813	487.196
4000	47.863,1	7.613.184	167.158	54.817	127.956	7.799.882
x4	x4,803	x16,011	x5,116	x4,993	x4,772	x16,010

Theoretically, the best case in quicksort occurs when we choose the mean as the pivot while partitioning. When we do the experiment, because we chose the pivot as the first element, if the array is random, the probability of selecting the pivot as a value close to mean is higher. That's why, the best case occurred when the array is random. Also, the time complexities are the lowest when the array is random or almost sorted in a way. On the other hand, when the array is sorted or reversely sorted, the pivot is chosen as the biggest or the lowest while partitioning and the nature of sorting act like a $O(n^2)$ algorithm.

To conclude, the time complexity of quicksort is $O(nlogn)$ in the best case and average case when the array is random or close to it. On the other hand, the worst case is $O(n^2)$ when the array is sorted in a way. When we compare with the theoretical expectations which are $O(nlogn)$ in the best & average cases and $O(n^2)$ in the worst case, the results meets with them.

Analyzing for Different k values

Again, when we do the experiment, we saw that there is no difference if we change the k value because of the reasons mentioned before.

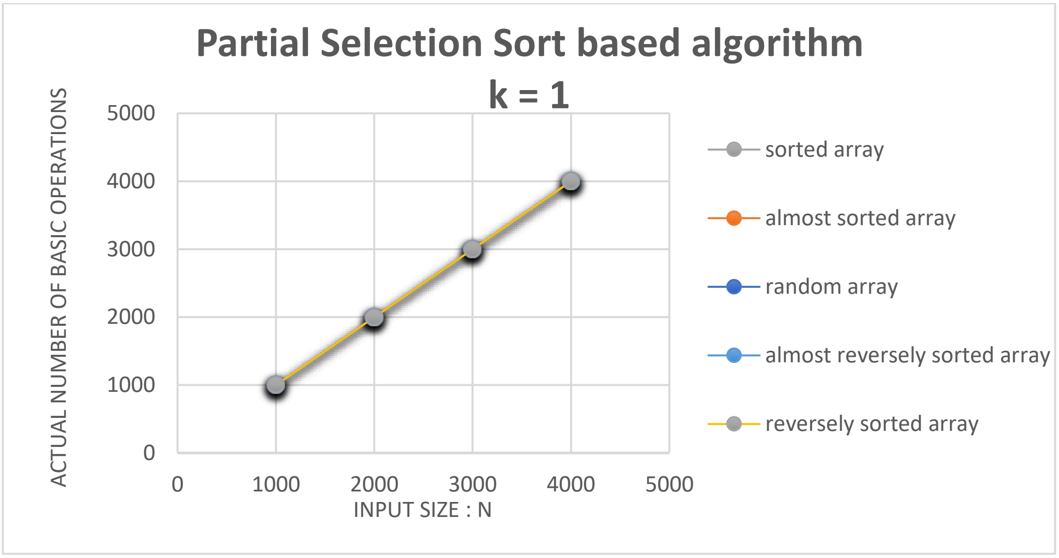
Partial Sorting based Algorithms

In partial sorting based algorithms we don't sort the array completely but we sort the array until we find k'th element. So we expect k to affect the time complexity.

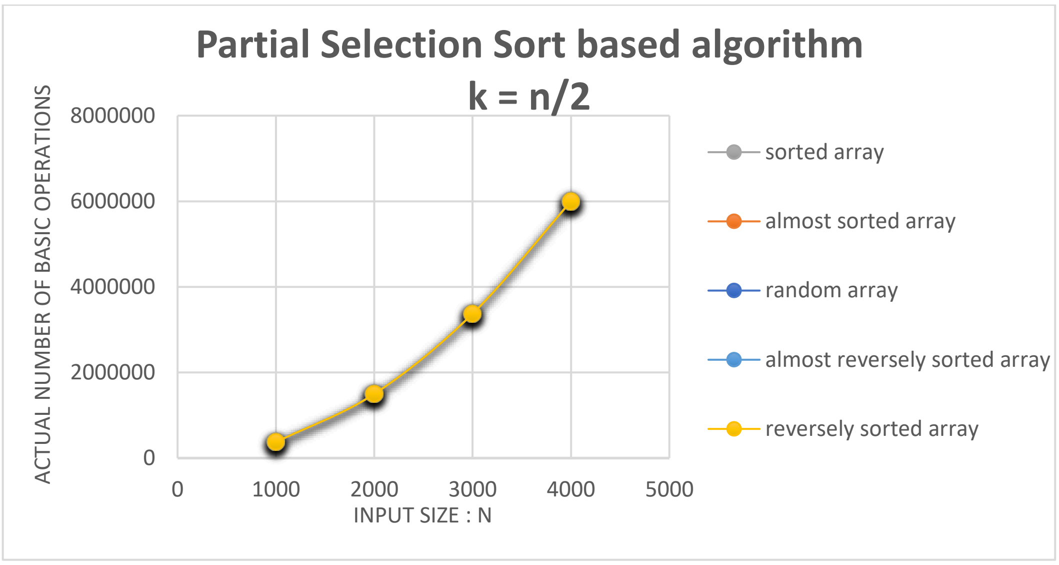
4. **Partial Selection Sort based Algorithm**

In this algorithm, comparison is chosen as the basic operation. When we do the experiment for the partial selection sort, as expected, we verified that the k value affects to the time complexity unlike the previous algorithms which means we have to use more data in our report to show how the algorithm behave clearly and also we have to determine k values so that all worst, best and avarage cases can occur. To do these, we chose the k values as 1, n/2 and n. we first determine if the time complexity changes in different characteristics for each k, then we exemine the effect of k by referencing random input type.

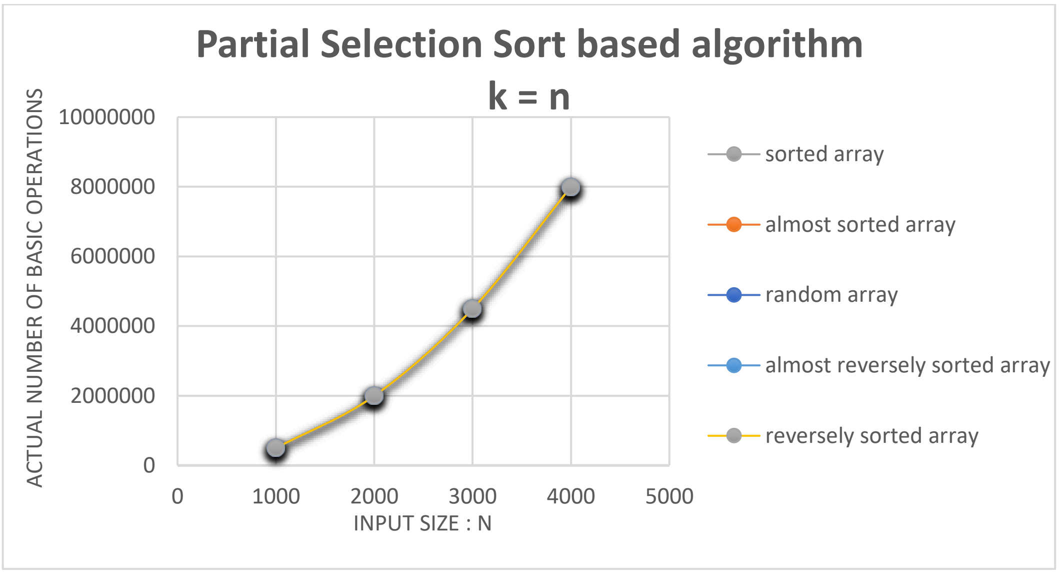
graph-5 : analyzing for different input characteristics



graph-6 : analyzing for different input characteristics



graph-7 : analyzing for different input characteristics



As it is seen in the graphs, when we change the characteristic of the input, the results comes out identicaly. We can understand why it behaves like this when we take a theoretical perspective to our code. the loop including the basic operation increment executes n.k times regardless from the nature of the array. It does not stop at anywhere until the end. So, the characteristic of the array have no effect on the number of basic operation.

Analyzing for Different k values

Because the number of basic operations does not change in different input characteristics, we chose one of them to show the effect of changing k value.

graph-8 : analyzing for different k values

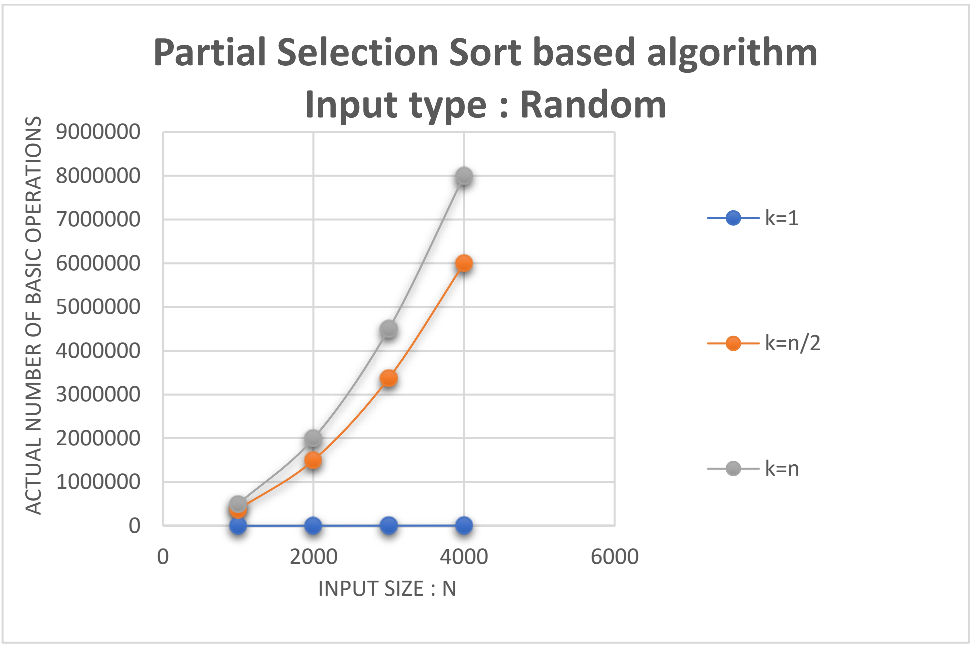


table-4 : analyzing for different k values

Expectations		Experimental Results		
Input size (n)	n^2	k = 1	k = n/2	k = n
1000	1.000.000	999	374.750	499.500
4000	16.000.000	3999	5.999.500	7.998.000
x4	x16	x4,003	x16,009	x16,012

Theoretically, the time complexity of partial selection sort is $O(kn)$. When we look at graph-8 & table-4, we see that the worst case is $O(n^2)$ which occurs when the k value is equal to n. if we change k as n/2, the total number of comparisons decreases but the time complexity is still the same. We can understand this by taking a little mathematical approach. By decreasing k from n to n/2, we just change the coefficient of it which have no effect on the time complexity theoretically. So, the total number of comparisons may decrease but the time complexity remains as the same. We can basically show this as:

$$n * (an + b) = an^2 + bn \in O(n^2)\,,\ \frac{n}{c} * (an + b) = \frac{an^2+bn}{c} \in O(n^2)\ where\ a, b, c \in R$$

Finally, the best case is $O(n)$ and it occurs when the k is 1. It also can be described as:

$1 * (an + b) = an + b \in O(n)\ where\ a, b \in R$

Based on mathematical approach and table-4, the results meet with the theoretical expectations.

Compared the presorting based algorithms, this algorithm may be preferrable for small values of k. But for large values of k, sorting with merge sort or quick sort would be a better choice because they have $O(nlogn)$ time complexity and $n * k > nlogn$ for greater values of k.

5. Partial Heap Sort based Algorithm

In this algorithm, comparison is chosen as the basic operation.

graph-9 : analyzing for different input characteristics

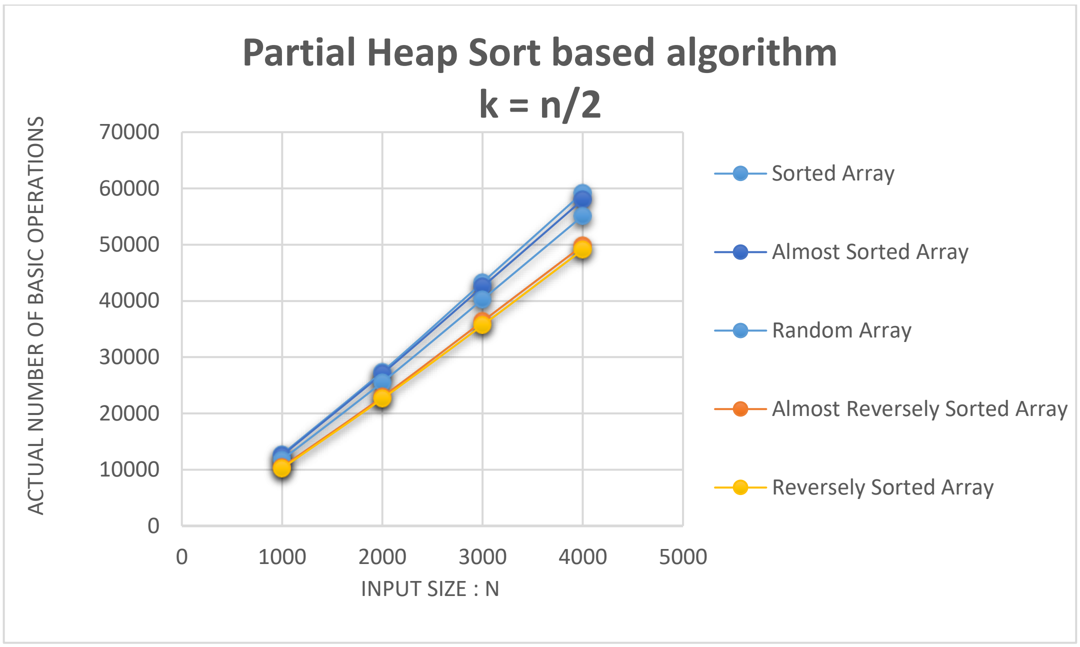


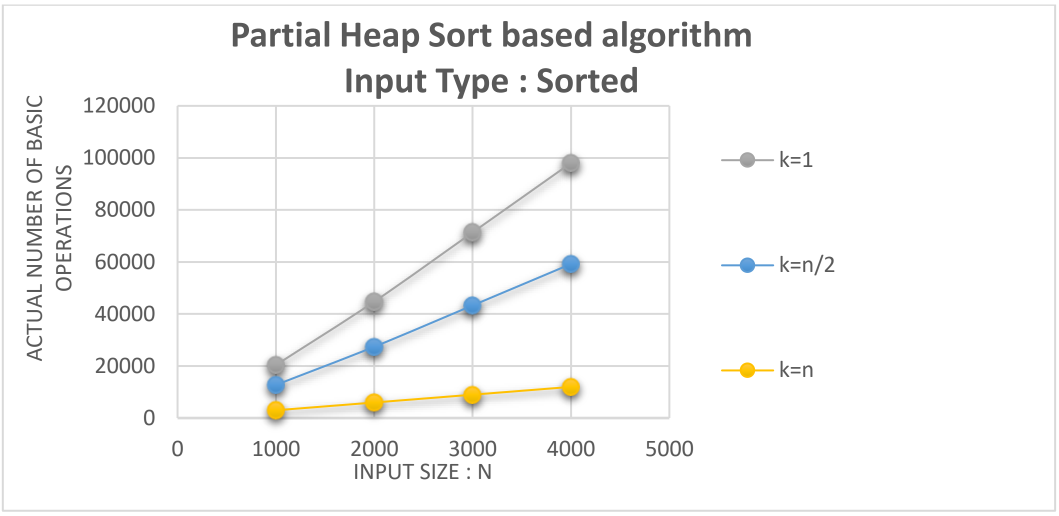
table-5 : analyzing for different input characteristics

Expectations		Experimental Results				
Input size (n)	$nlog_2n$	Sorted	Almost Sorted	Random	Almost Reversely Sorted	Reversely sorted
1000	9.965,8	12.726	12.535	11.773	10.473	10.241
4000	47.863,1	59.215	57.992	55.114	49.855	49.104
x4	x4,803	x4,653	x4,626	x4,681	x4,76	x4,795

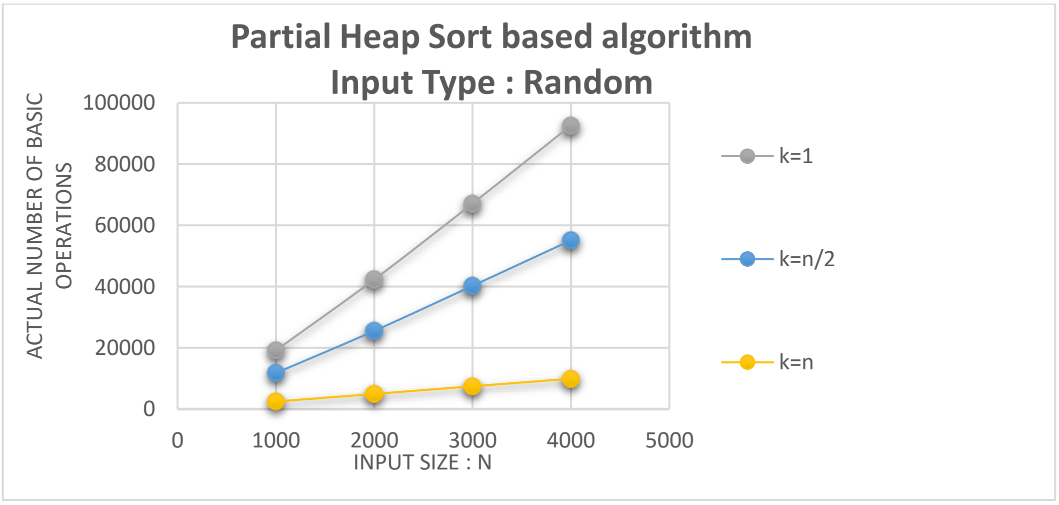
As it is seen it the graph-9 and table-5, the differences occurring based on the characteristics are not that much and it can be said that they all have the same time complexity of $O(nlogn)$.

We can also see that when the array is goes from sorted to reversely sorted, the number of comparisons decreases. The reason behind that is actually about heapifying the array. When the array is reversely sorted, the children are always smaller than the root while constructing a max-heap. So, there is no fixing operation affecting to the total number of comparisons.

graph-10 : analyzing for different k values



graph-11 : analyzing for different k values



graph-12 : analyzing for different k values

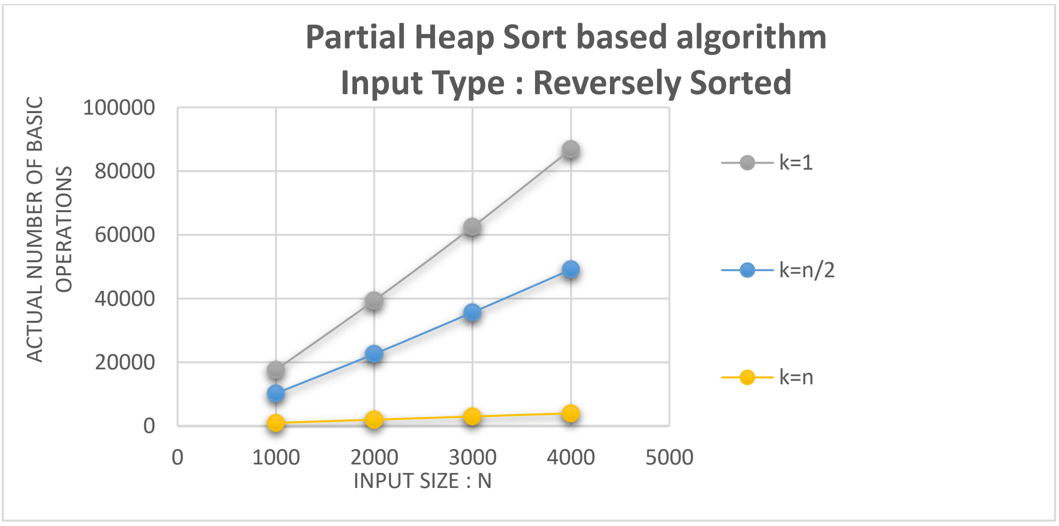


table-6 : analyzing for different k values

Expectations		Experimental Results (for random array)		
Input size (n)	$nlog_2n$	k = 1	k = n/2	k = n
1000	9.965,8	19.167	11.773	2476
4000	47.863,1	92.635	55.114	9931
x4	x4,803	x4,833	x4,681	x4,011

Theoretically, the time complexity for heap sort based algorithm in our project is: $O(n + (n - k)logn)$. When the k is equal to n, it behaves as $O(n)$. In other cases, because we just change the coefficient of n, number of comparisons may decrease but the degree of the equation does not change, and time complexity remains as $O(nlogn)$. We can see this in our experimental results. When the k is equal to 1 or n/2, the time complexity is $O(nlogn)$. But when the k is equal to n, It behaves like $O(n)$.

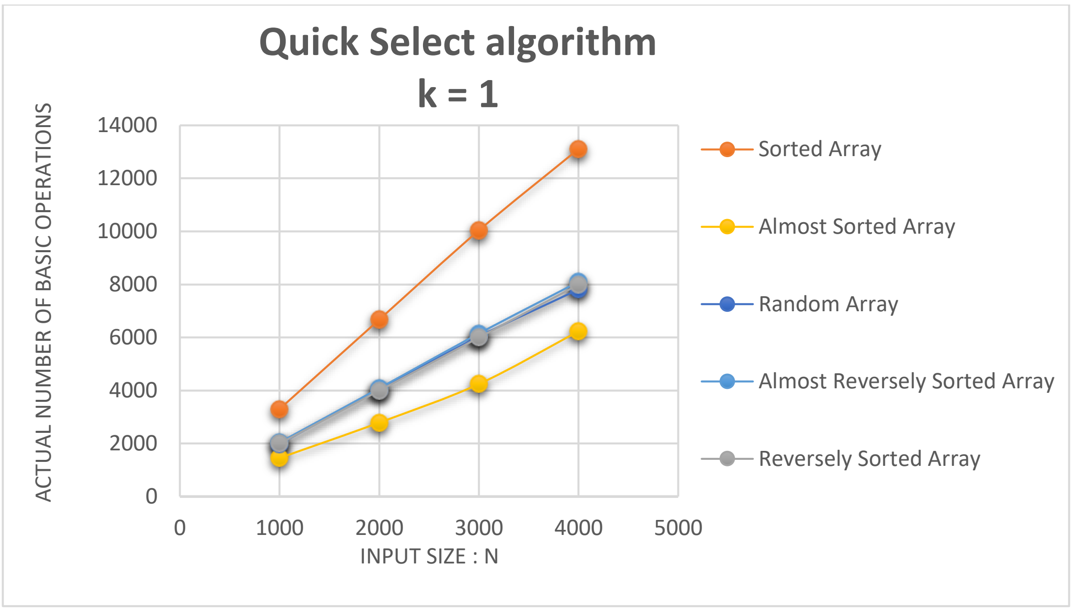
To conclude, the best case for heap sort based algorithm is $O(n)$ which occurs when k=n. Also, when we combine this with the reversely sorted array, we get the lowest result. Average and worst cases are $O(nlogn)$ which occurs when the k is any value except 1 regardless from the characteristic of the input file. The results are seem not to meet with theoretical expectations in the best case but it is about the nature of our problem. In presorting based algorithms, we see that the best case time complexity is $O(nlogn)$. But in this algorithm, the best case time complexity is $O(n)$ and in average, $O(nlogn)$. So, partial heap sort based algorithm can be preferred over any presorting based algorithm. Also, this algorithm is better than partial selection sort based algorithm since partial selection sort based algorithm has a time complexity of $O(n^2)$ in average and worst cases.

Partition based Algorithms

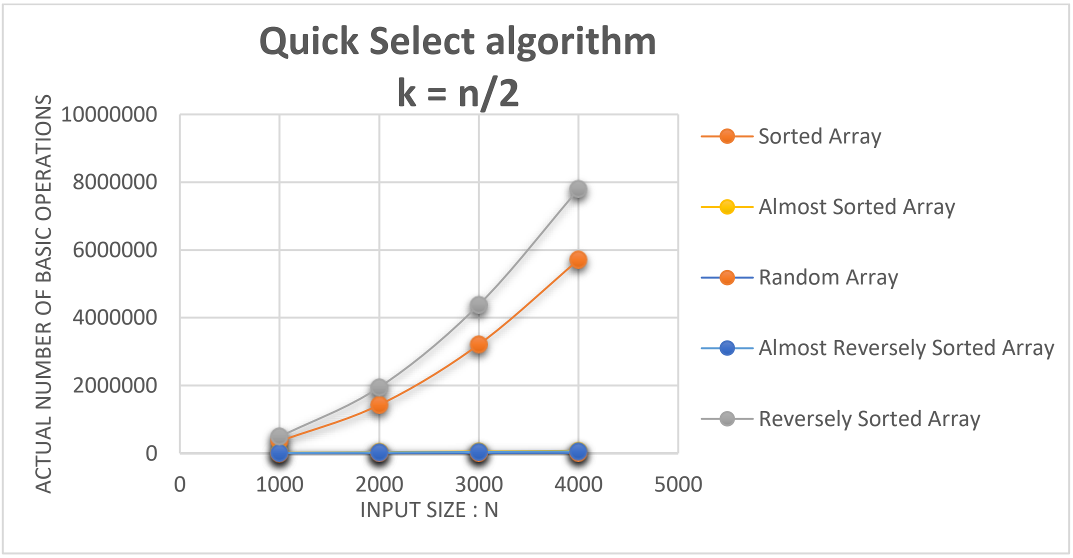
6. Quick Select Algorithm

In this algorithm, comparison is used as the basic operation, and it is incremented in the partitioning part. First element is chosen as pivot.

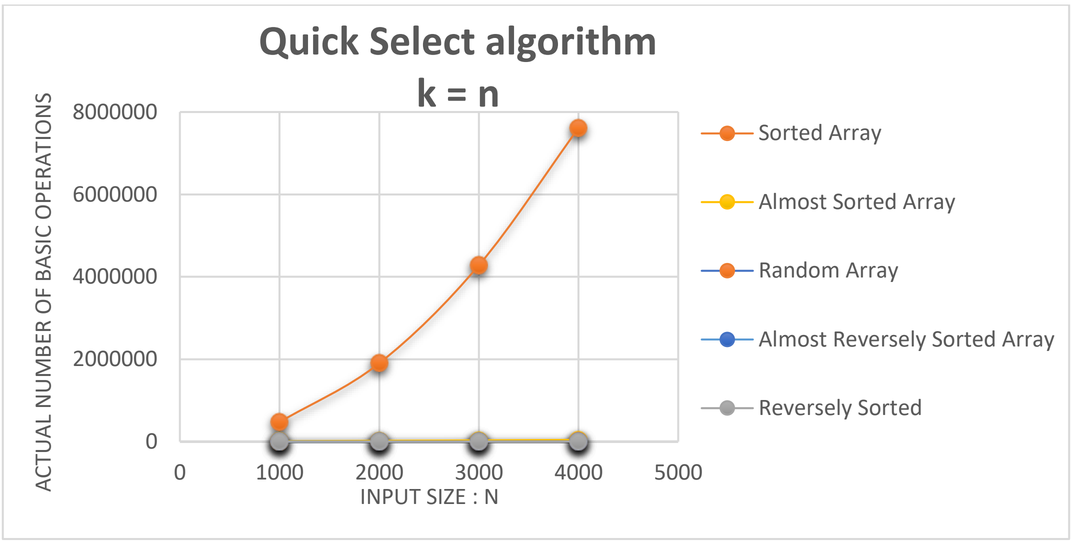
graph-13 : analyzing for different input characteristics for k = 1



graph-14 : analyzing for different input characteristics for k = n/2

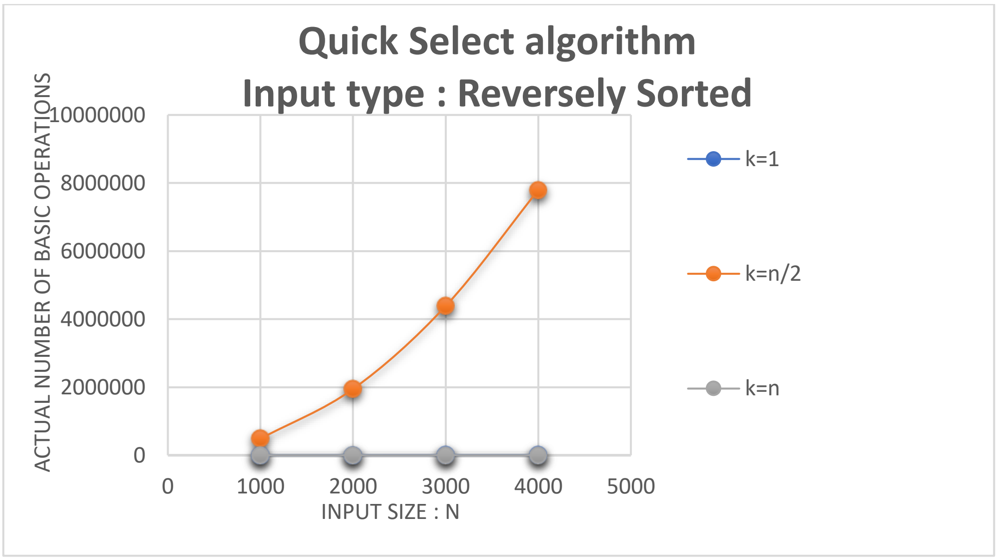


graph-15 : analyzing for different input characteristics for k = n



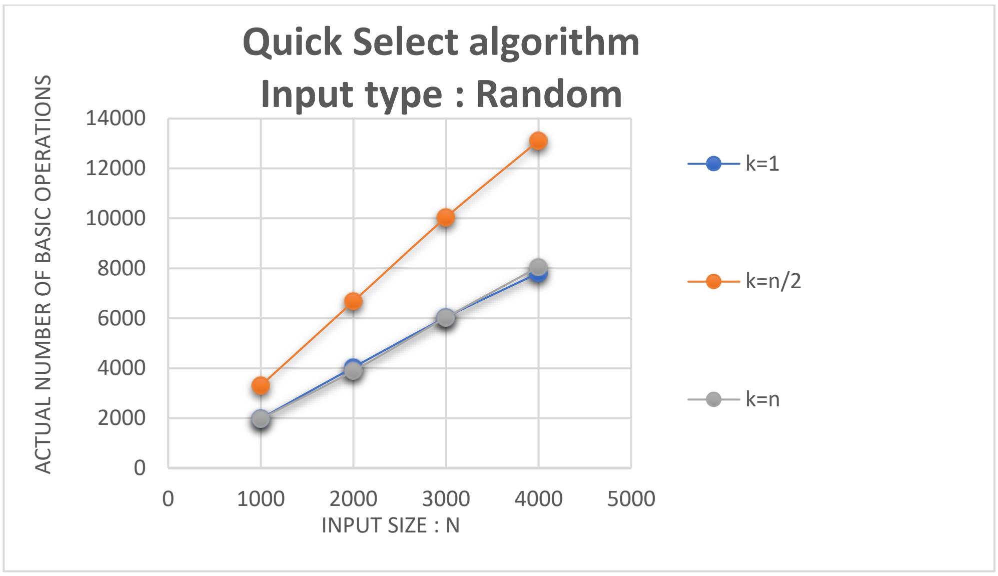
With a few exceptions, the results are taking a linear time. And exceptions, are not depending on a familiar reason. Combining specific input characteristics and k values occurs $O(n^2)$ time complexity because of the nature of quick selection. In the inputs sorted in a way, because we choose the first element as pivot, we can not decrease the size of array sensibly while partitioning. More information will be given after the graphs about the effect of k value.

graph-16 : analyzing for different k values

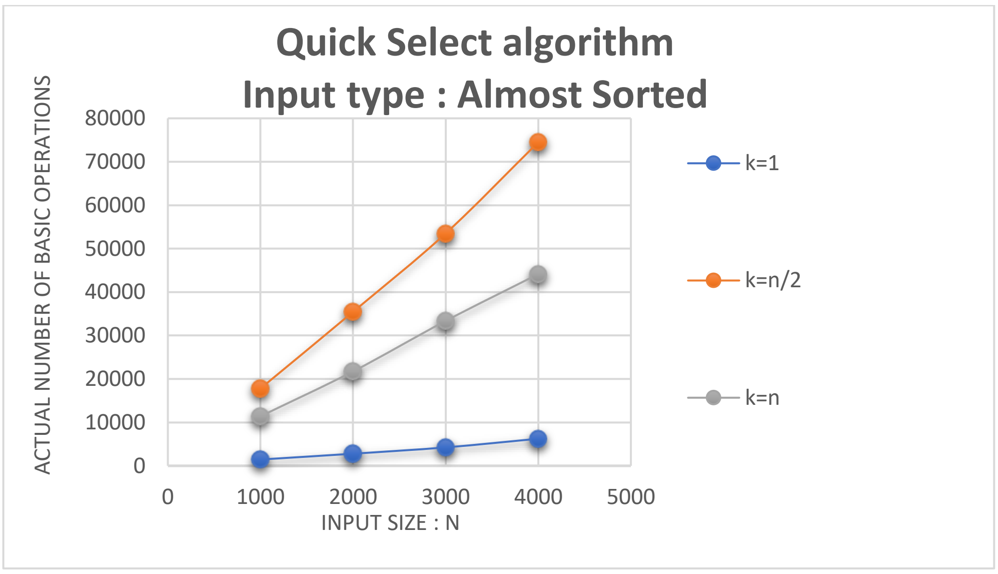


In this algorithm, the worst case occurs when the array is sorted in a way as mentioned. For example, when the array is sorted and k=n, we search the whole array and decrease the remaining array by one until k'th smallest element is found (n times). Another example, similar to first one, when the array is reversely sorted and k = n/2, we choose the largest element as pivot all the time which is the only element can be eliminated. So, we decrease the remaining array by one until k'th smallest element is found (n/2 times in this case). In both examples, the time complexity is $O(n^2)$. (Worst case)

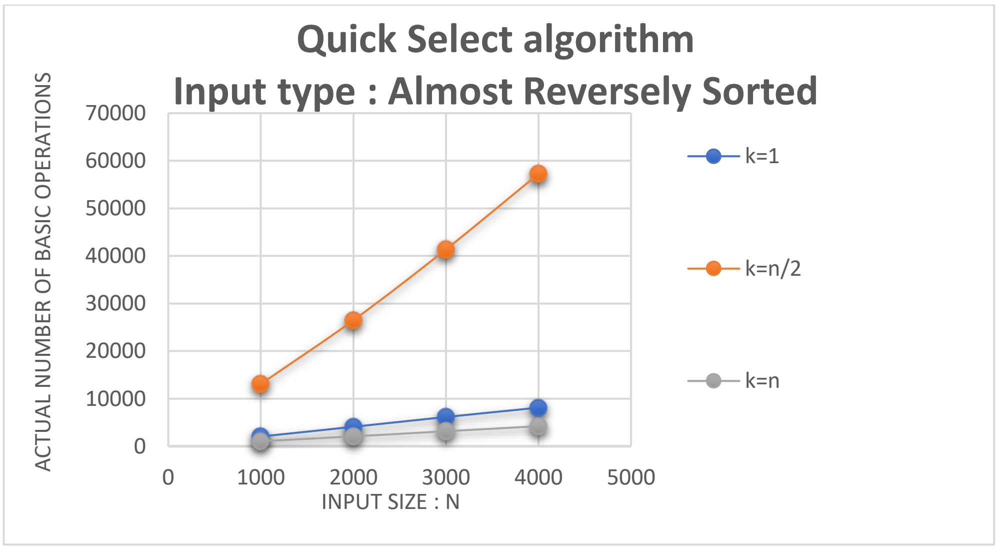
graph-17 : analyzing for different k values



graph-18 : analyzing for different k values



graph-19 : analyzing for different k values

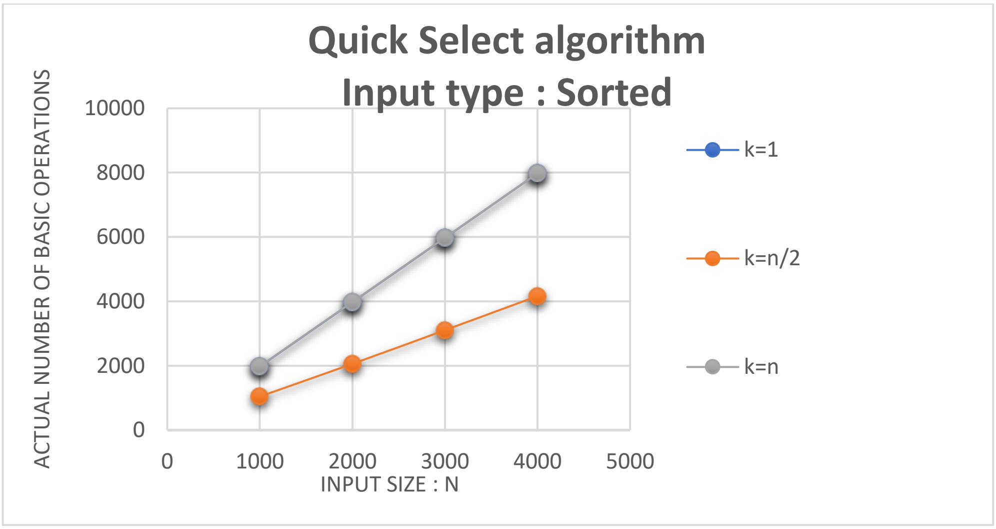


On the other hand, when the array is not sorted in a way. We do not see a result having $O(n^2)$ time complexity. They all have a linear growth rate. When we also adjust the k based on the characteristic of the array, we can see lower results but it would be still linear. To conclude, in this algorithm, both characteristic and k value affects to the result but these effects are coercive and worst case can be avoided easily. Mainly, the best case and average cases for this algorithm is $O(n)$ when the worst case is $O(n^2)$. The results meets with the theoretical expectations. Since this algorithm has a linear average time complexity, it should be preferred over the presorting or partial sorting based algorithms. But it should be considered that the worst case is $O(n^2)$!

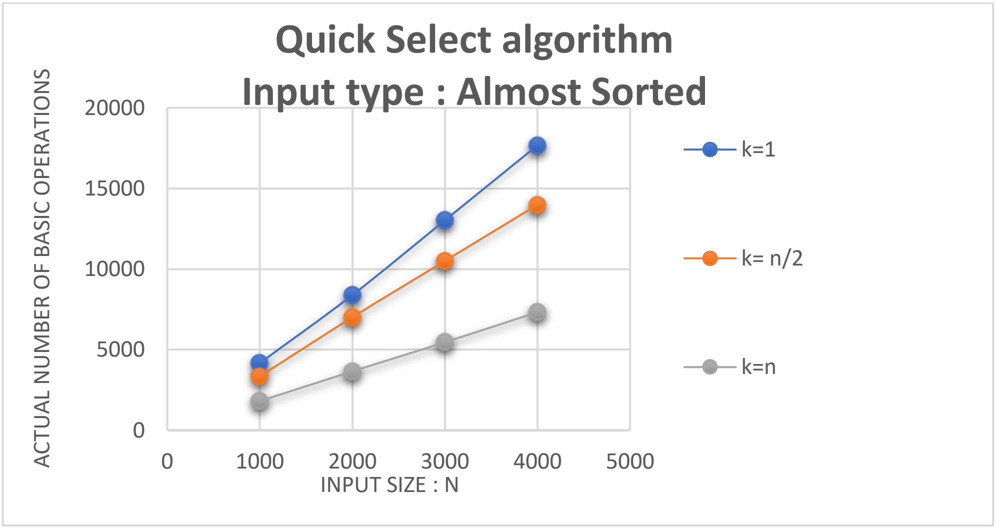
7. Quick Select Algorithm with Median of three pivot selection.

The only difference of this algorithm from the previous one is just selecting the pivot element as the median of first, last and middle element. The comments will be given after the graphs.

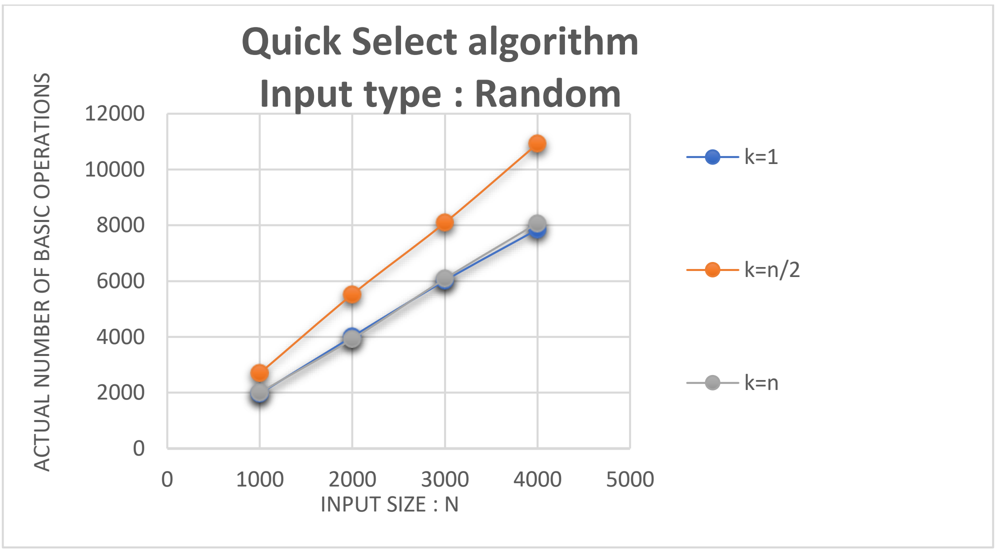
graph-20 : analyzing for different k values



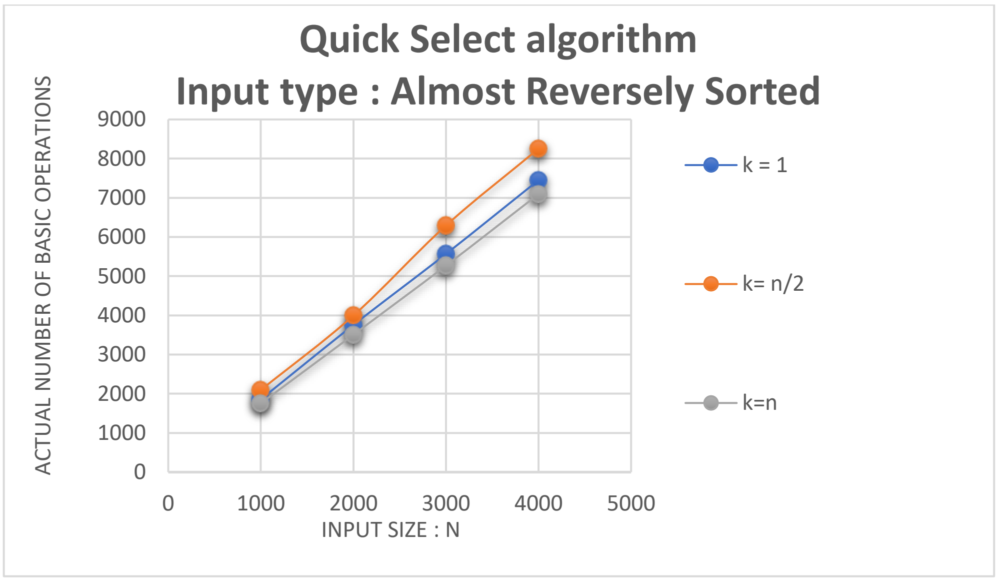
graph-21 : analyzing for different k values



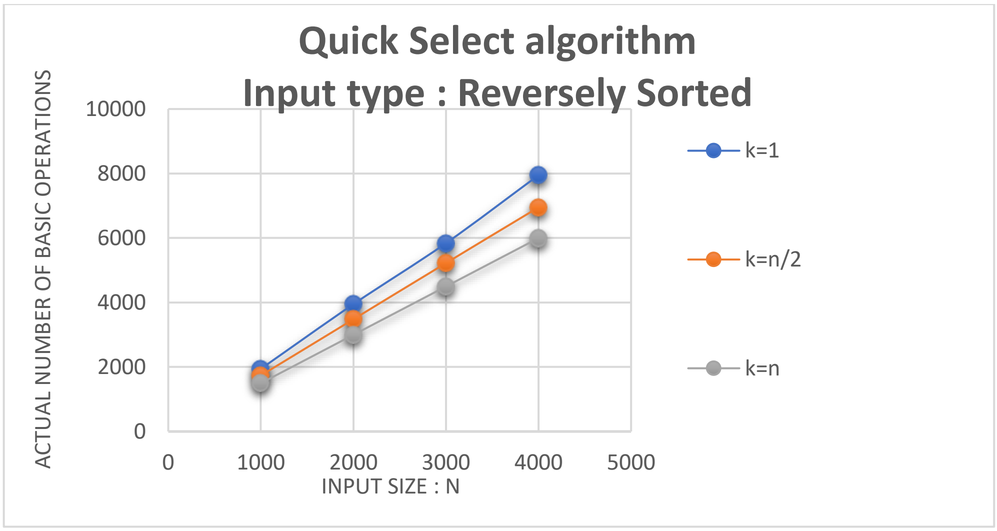
graph-22 : analyzing for different k values



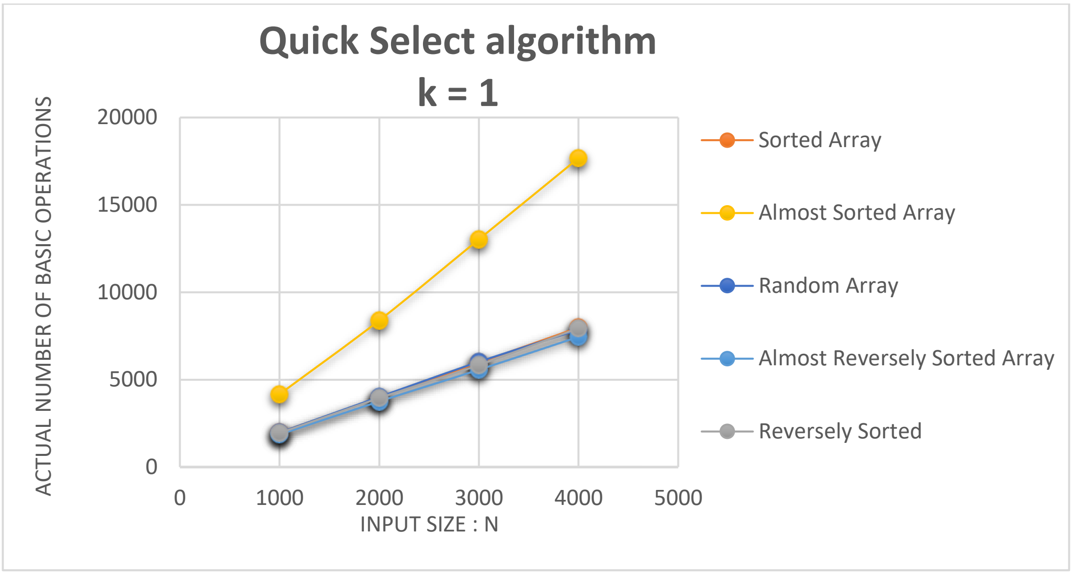
graph-23 : analyzing for different k values



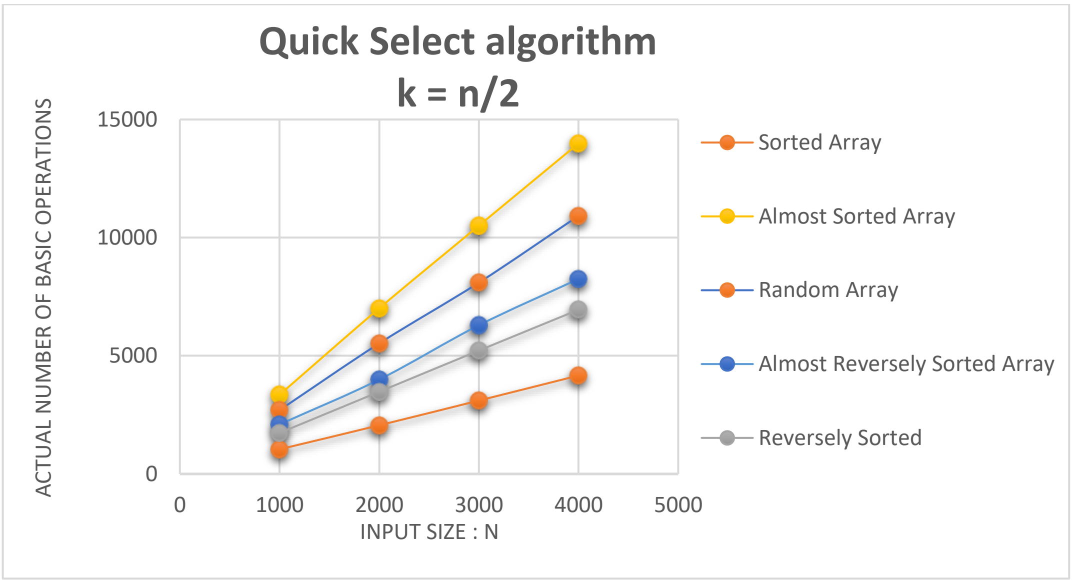
graph-24 : analyzing for different k values



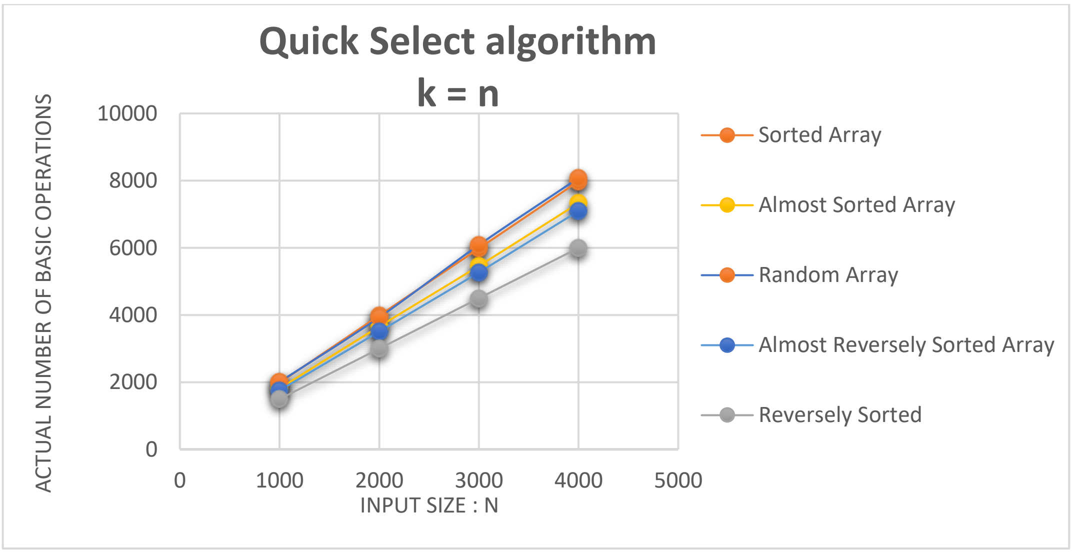
graph-25 : analyzing for different input characteristics



graph-26 : analyzing for different input characteristics



graph-26 : analyzing for different input characteristics



Theoretically, choosing median of three as pivot provides us a better dispersion. Based on these experimental results, we saw that selecting pivot as median of three provides us not just a better dispersion but also protect us from specific results having a $O(n^2)$ time complexity. As it is seen in the graphs, all input characteristic-k value combinations have a $O(n)$ time complexity. Again, the results meet with the theoretical expectations and as also expected, selecting pivot as median of three is a better way than selecting as the first element. Overall, this is the best algorithm to choose since it performs in linear time regardless from the value of k or input characteristic.

5. Comparison of algorithms

table -7 : generalized time complexities of algorithms

	Best	**Average**	**Worst**
Insertion Sort based	$O(n)$	$O(n^2)$	$O(n^2)$
Merge Sort based	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$
Quick Sort based	$O(nlogn)$	$O(nlogn)$	$O(n^2)$
Partial Selection Sort based	$O(n)$ k = 1	$O(kn)$	$O(n^2)$ k = n
Partial Heap Sort based	$O(n)$ k = n	$O(nlogn)$	$O(nlogn)$
Quick Select based	$O(n)$	$O(n)$	$O(n^2)$ k = n
Quick Select Median based	$O(n)$	$O(n)$	$O(n)$

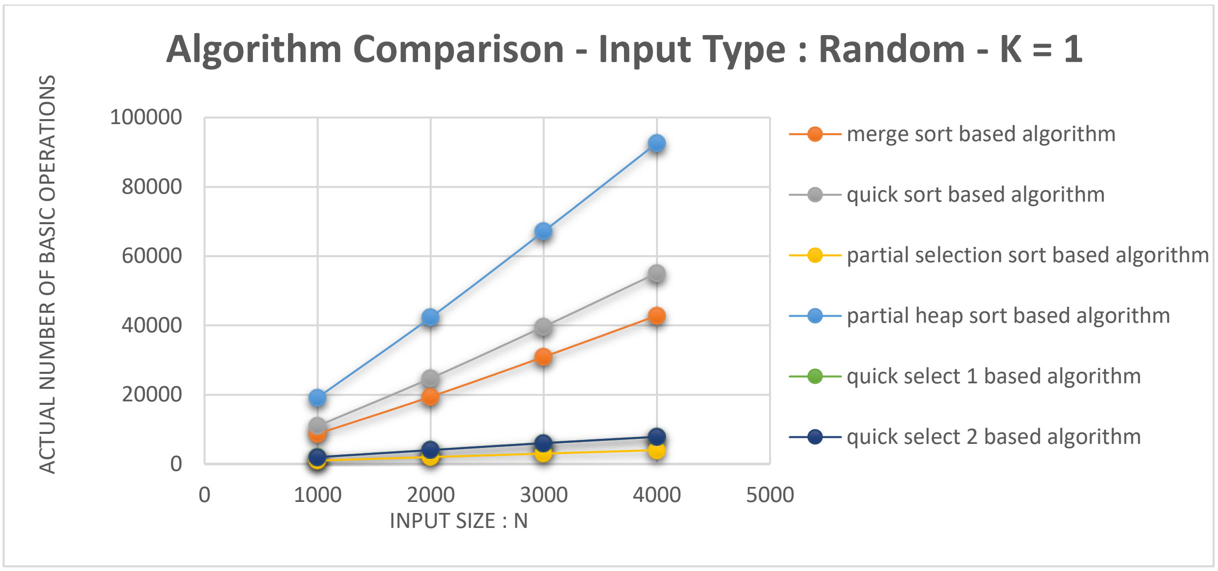
When we look at the table, we can see that quick select based algorithm with median of three is the fastest algorithm to find k'th smallest element in an array. Other rankings are seems debatable. If we consider that the probability of seeing the worst case of quick select algorithm with first element as pivot, it may be the second best option.

On the other hand, in real world, the data to be processed is usually random. So, the fallowing graphs are showing the number of basic operations for different kinds of k with random input file.

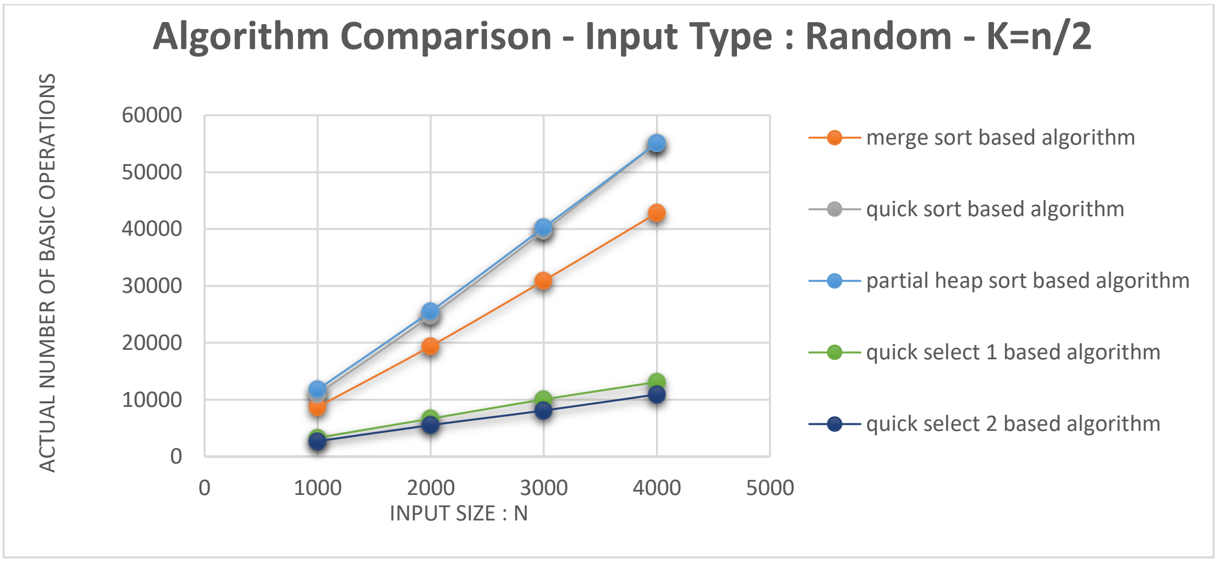
! Because insertion sorted based algorithm results are huge in all cases, we do not add this to our graph.

! Because partial selection sort based algorithm results are huge except for k=1, we do not add this to our graph except first one.

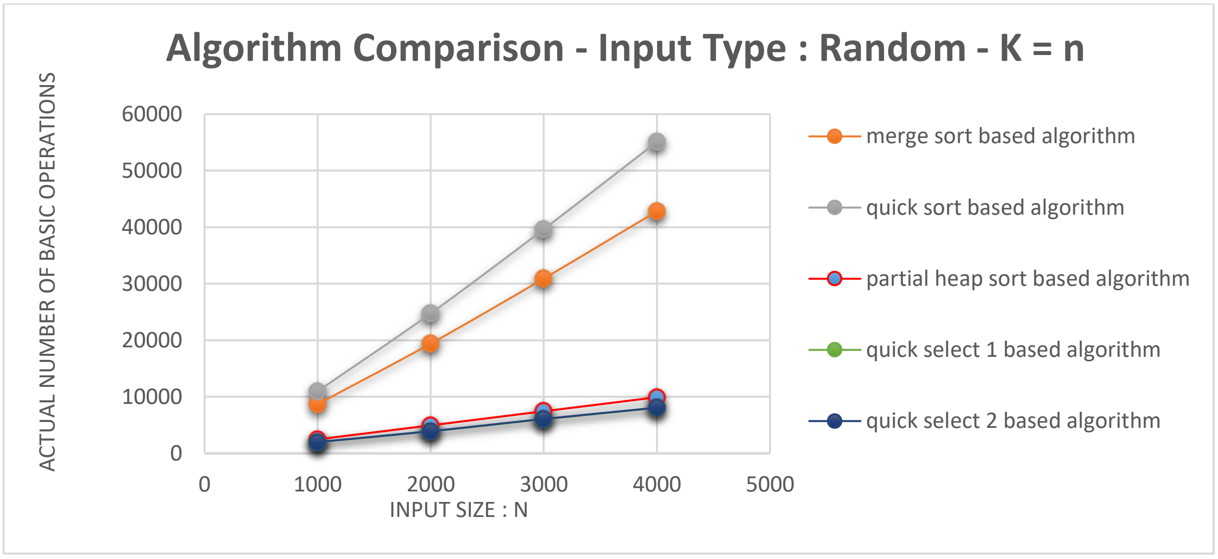
graph – 26 : algorithm comparison for random characteristic with k = 1



graph – 26 : algorithm comparison for random characteristic with k = 1



graph – 26 : algorithm comparison for random characteristic with k = 1



As we can see in graphs 26-28 and table – 7, in real world examples, the best option seems to be quick select algorithm with median of three pivot selection. Even in the worst cases, it has a $O(n)$ time complexity.

On the other hand, worst option seems to be insertion sort. The number of comparisons grows so fast that we could not show this in the graph of all the algorithms. Likewise, partial selection sort grows really fast as k increases.