CSE246 - ANALYSIS OF ALGORITHMS

2017/18 Spring

HOMEWORK 2

Sezin Gümüş 150113841

Ufuk Çetinkaya 150113824

Experiment Aim:

In this experiment, we will test and compare the following candidate sorting algorithms for different values of k and various input sizes/types n:

1- Sort by Insertion-sort and return the k'th element in the list

2- Sort by Merge-sort and return the k'th element in the list,

3- Do not sort the list. Apply the quick select algorithm (based on array partitioning). While partitioning, choose the pivot element as the first element in an array.

4- Do not sort the list. Apply the quick select algorithm again using median-of-three pivot selection.

1.1 Methodology:

We will do an emprical analysis by measuring total running time of algorithms.

Then we will compare these findings with theoretical values.

Consequently, we will add extensive comments for results.

1.2. Inputs:

a) We have prepared 4 input files, these being:

- mixdata.txt: This input includes 5000 mixed elements. We can also modify the code to use less inputs (500, 1000, 2500 etc) if needed.
- increase.txt: This input file includes 5000 increasing elements. We can also modify the code to use less inputs (500, 1000, 2500 etc) if needed.
- decrease.txt: This input file includes 5000 decreasing elements. We can also modify the code to use less inputs (500, 1000, 2500 etc) if needed.
- sameelement.txt: This input file includes the same element, repeating 5000 times. We can also modify the code to use less inputs (500, 1000, 2500 etc) if needed.

We have generated all for input files by a short code for generating inputs.

b) We have chosen the k-values as follows:

As we want to measure as many possibilities as possible, we want to search for one element in the starting position, one element close to the middle and one element at the end of the list.

For 1000 inputs, we will use k values: 3, 455, 990

For 5000 inputs, we will use k values: 3, 2550, 4900

c) Metrics: We will measure the total runtime for complexity. Counting basic operation executions was another option but because our code gives quite accurate runtime results, (and counting basic operations would lead almost to the same result) we decided that it was not necessary to double-calculate every finding.

2.1. Algorithms used:

We implemented our codes in C. We have 4 algorithm files as follows:

- insertionsort.c
- mergesort.c
- quickselect.c
- quickselectwithmedianofthree.c

All of these algorithms ask for an input file at the start and the k value is modifyable within the code.

It should be reminded that the operations for quickselect.c and quickselectwithmedianofthree.c are made without prior sorting as requested in the homework document.

2.2. Codes (with comments):

insertionsort.c

```
//s:starting point

//e:ending point for sorting

int insertionSort  (int  list [],int s, int e ,int k)

{

 int i =0;

 for (i=s+1;i<e;i++)

  {

     int val = list [i];

     int j = i-1;

     while (j>= 0 && val < list[j])

          {    list [j+1] = list[j];  j--; }

      list[j+1]= val;  }

     return list [k-1];    }
```

mergesort.c

```
// This method finds k th smallest element by using mergesort algorithm

int mergeSort(int vec[], int size_vec , int k)

{           int *vec_left, *vec_right;

            int i, middle;

            if(size_vec < 2) // base condition return;

          middle = size_vec / 2; // gets the middle of the vector

          // creates two vectors
```

```
        vec_left = (int*)malloc(middle * sizeof(int));

        vec_right = (int*)malloc((size_vec - middle) * sizeof(int));

         // fills the vectors

        for(i = 0; i < middle; i++)

              vec_left[i] = vec[i];

        for(i = middle; i < size_vec; i++)

             vec_right[i - middle] = vec[i];

         // recursive calls

         mergeSort(vec_left, middle , k);

         mergeSort(vec_right, size_vec - middle,k);

         merge(vec_left, vec_right, vec, middle, size_vec - middle, size_vec);

         free(vec_right);

         free(vec_left);

         return vec[k-1];

}
```

quickselect.c

```
//This function finds k th smallest element in a given array by using quickselect algorithm

//uses Lomuto Partioning Algorithm

//Solves the selection problem by recursive partition-based algorithm

//Input: Subarray A[l..r] of array A[0..n − 1] of orderable elements and

//integer k (1 ≤ k ≤ r − l + 1)

//Output: The value of the kth smallest element in A[l..r]

int quickSelect (int a [] ,int l , int r , int k)

{ if (l<=r) {

  int s = lomutoPartition(a,l,r);
```

```
if (s == k-1)
 return a[s];
else if (s >  k )
 quickSelect(a ,l, s-1 ,k);
else  quickSelect(a , s+1 , r , k ); }  }
```

quickselectwithmedianofthree.c

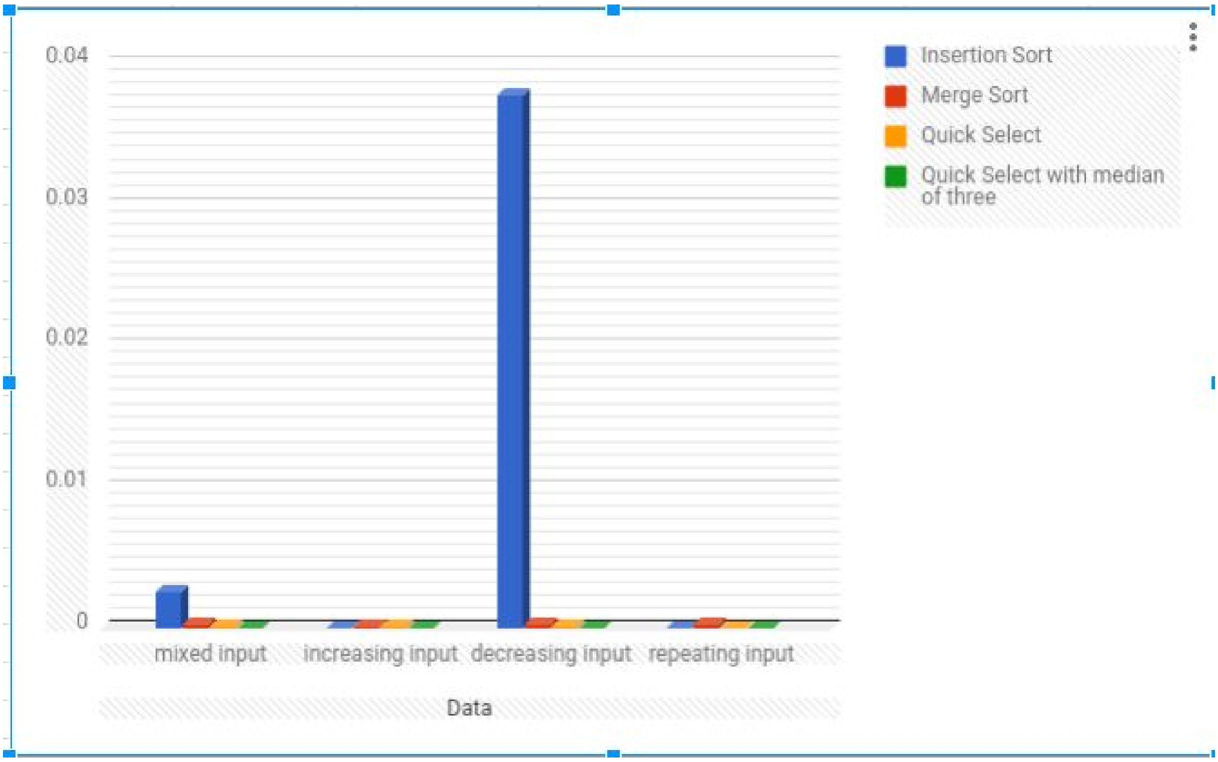
```
//This function finds k th smallest element in a given array by using quickselect algorithm
//uses median of three partition algorithm (median3)
//Solves the selection problem by recursive partition-based algorithm
//Input: Subarray A[l..r] of array A[0..n − 1] of orderable elements and
//integer k (1 ≤ k ≤ r − l + 1)
//Output: The value of the kth smallest element in A[l..r]
int quickSelect (int a [] ,int l , int r , int k)
{
  if (l<=r) {
    int s =median3(a,l,r);
      if (s == k-1)
       return a[s];
      else if (s >  k - 1 )
        quickSelect(a ,l, s - 1 ,k);
      else
        quickSelect(a , s + 1 , r , k);}}
```

3.1. Results in Graphs&Tables and Evaluating Results:

As mentioned before, we did the measurements in terms of time, by comparing 4 different algorithms in the x-axis with the time dimension (in seconds) in y-axis. We did it for 2 different n and 3 different k values. So we measured 6 cases in total. The results are as follows:

Case 1: n=1000 ; k=3

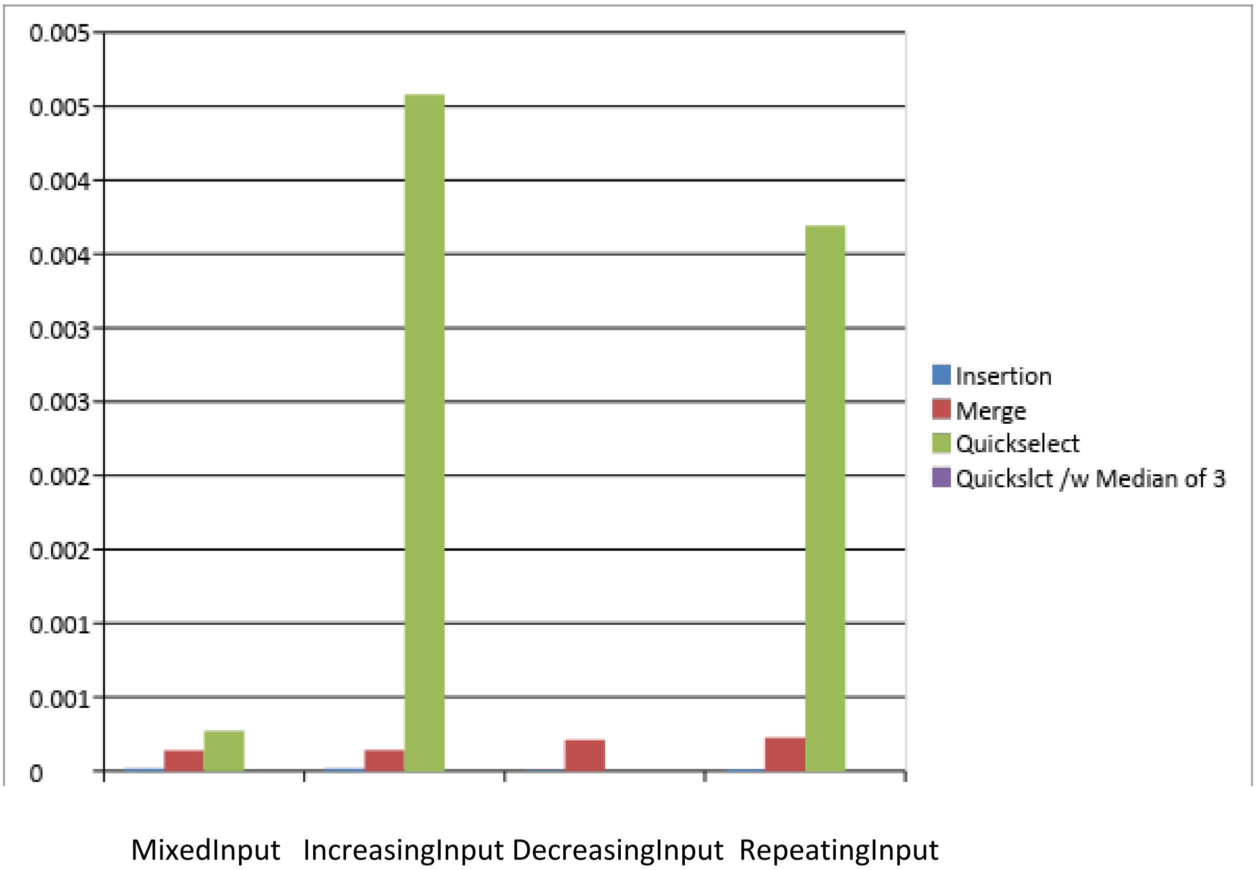
	Insertion	Merge	Quickselect	Quickslct /w Median of 3
mixdata.txt	0,002506	0,000223	0,000073	0.000003
increase.txt	0,000028	0,000147	0,000066	0.000003
decrease.txt	0,03781	0,000217	0,000146	0.000028
sameelement.txt	0,000013	0,000216	0,000052	0.000003



(Runtime in seconds; n=1000 ; k=3)

Case 2:   n=1000 ; k=455

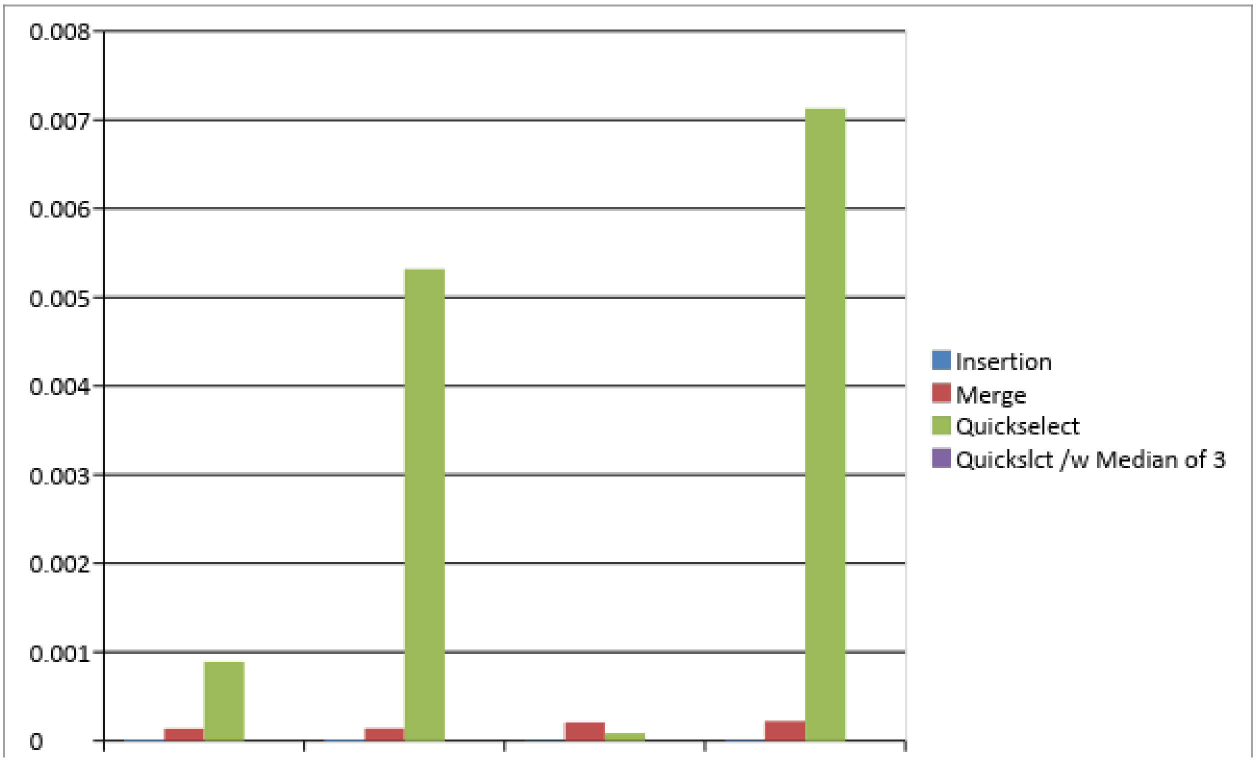
	Insertion	Merge	Quickselect	Quickslct /w Median of 3
mixdata.txt	0,000018	0,000139	0,000273	0.000003
increase.txt	0,000016	0,000141	0,004579	0.000003
decrease.txt	0,000006	0,00021	0.006737	0.000028
sameelement.txt	0,000012	0,000226	0,003693	0.000003



(Runtime in seconds; n=1000 ; k=455)

Case 3: n=1000 ; k=990

	Insertion	Merge	Quickselect	Quickslct /w Median of 3
mixdata.txt	0,000016	0,000137	0,000888	0.000003
increase.txt	0,000015	0,00014	0,005316	0.000002
decrease.txt	0,000005	0,000205	0,000084	0.000015
sameelement.txt	0,000012	0,00022	0,007133	0.000001

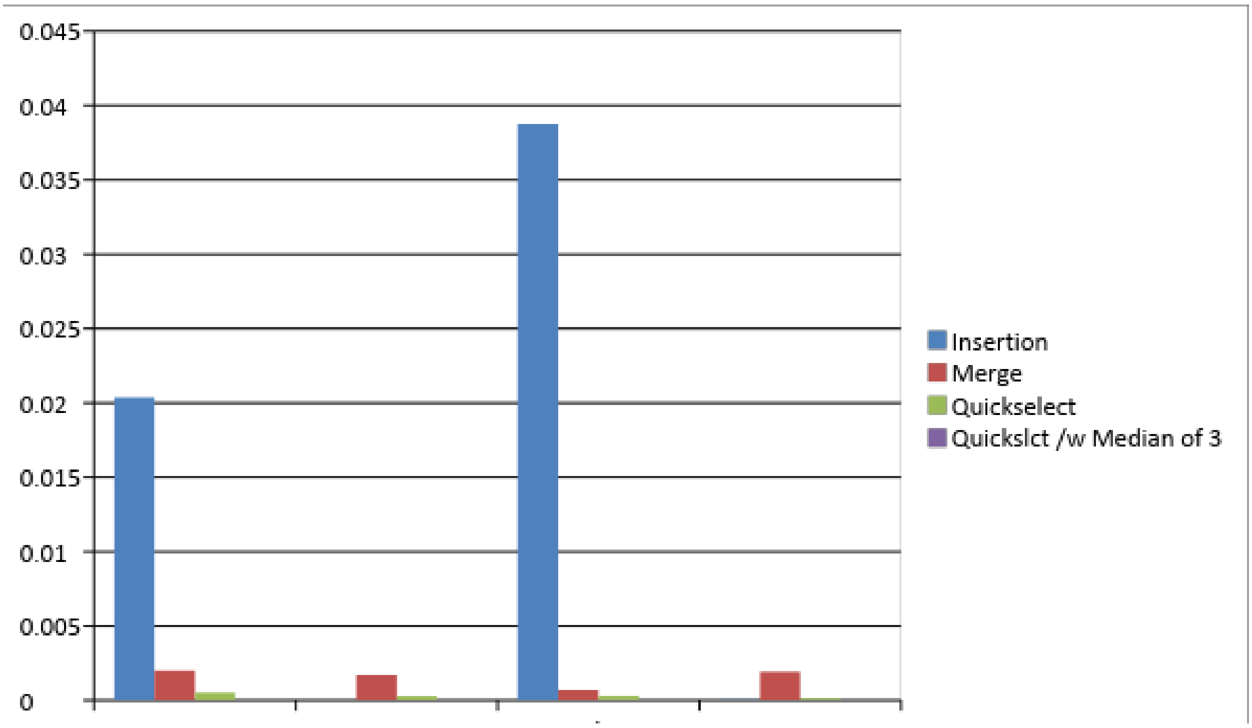


MixedInput IncreasingInput DecreasingInput RepeatingInput

(Runtime in seconds; n=1000 ; k=990)

Case 4: n=5000 ; k=3

	Insertion	Merge	Quickselect	Quickslct /w Median of 3
mixdata.txt	0,020361	0,002007	0,000527	0.000003
increase.txt	0,000021	0,001695	0,000276	0.000004
decrease.txt	0,038753	0,000707	0,000295	0.000112
sameelement.txt	0,000023	0,001927	0,000105	0.000003

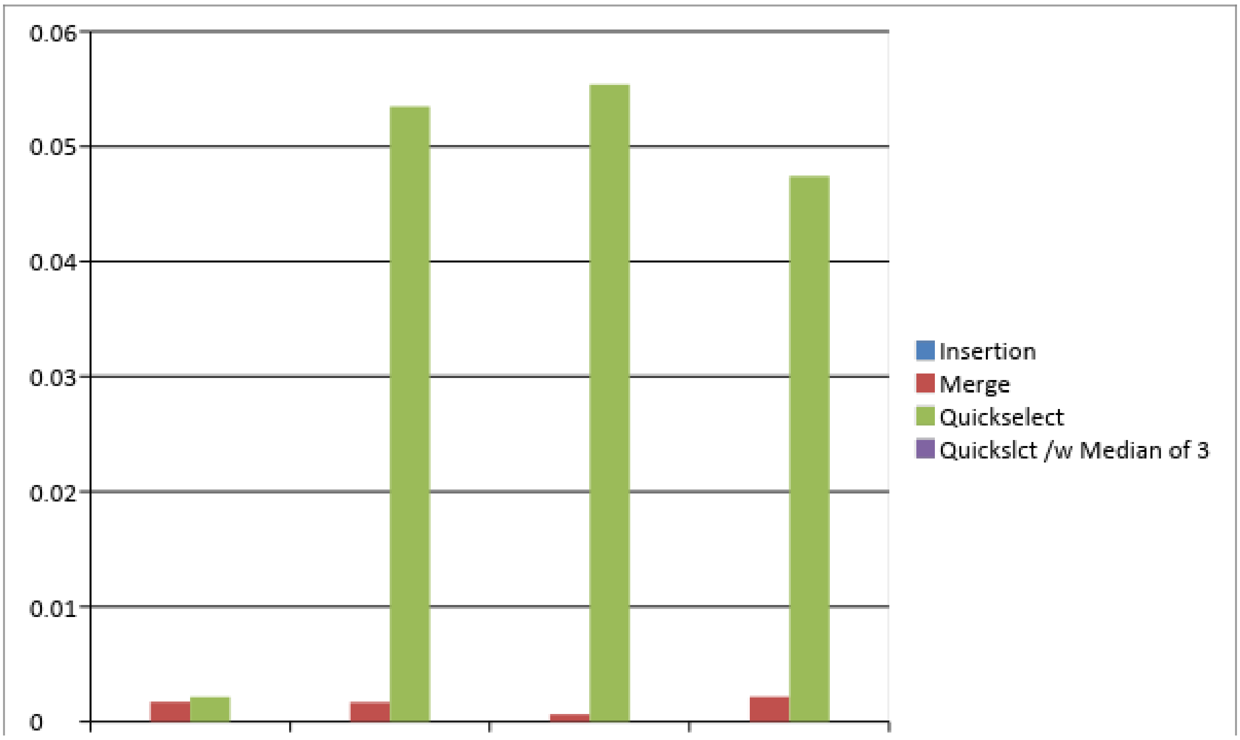


MixedInput IncreasingInput DecreasingInput RepeatingInput

(Runtime in seconds; n=5000 ; k=3)

Case 5: n=5000 ; k=2550

	Insertion	Merge	Quickselect	Quickslct /w Median of 3
mixdata.txt	0,00002	0,00169 7	0,002129	0.000003
increase.txt	0,00002	0,00166 7	0,053452	0.000004
decrease.txt	0,000031	0,00061 6	0,055384	0.000052
sameelement.txt	0,000023	0,00214 5	0,04738	0.000002

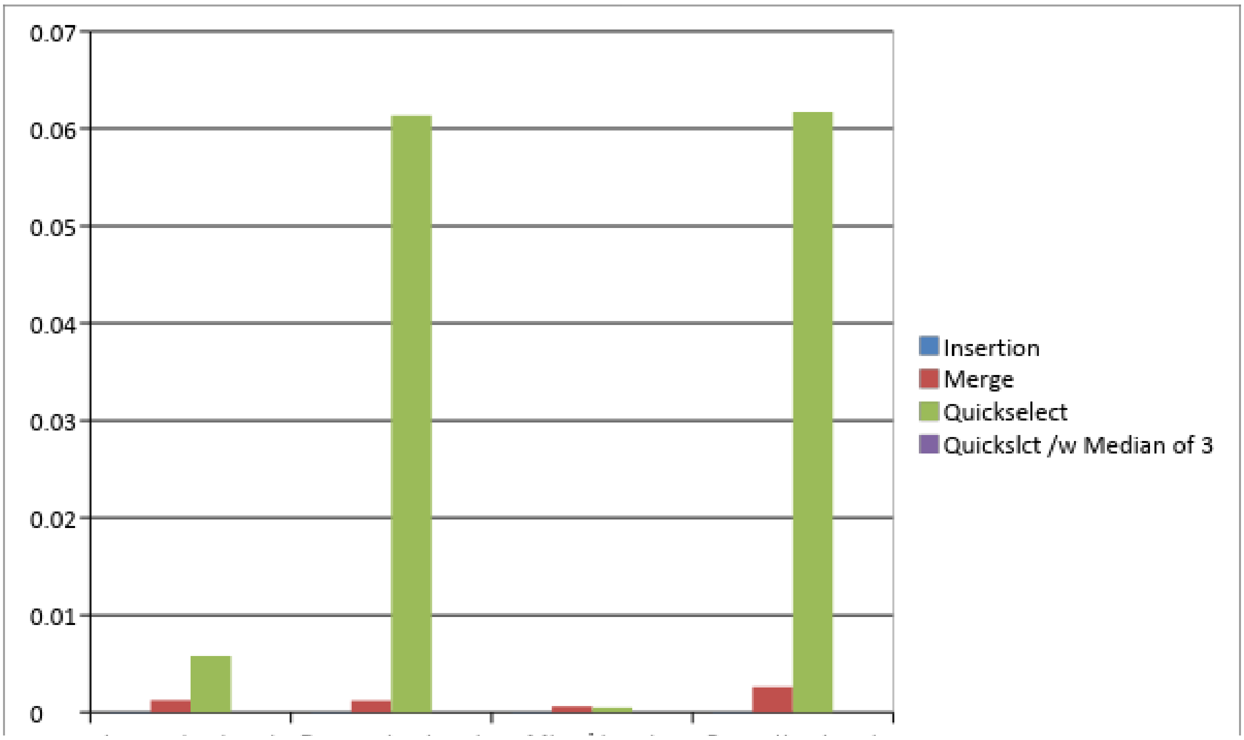


MixedInput IncreasingInput DecreasingInput RepeatingInput

(Runtime in seconds; n=5000 ; k=2550)

Case 6: n=5000 ; k=4990

	Insertion	Merge	Quickselect	Quickslct /w Median of 3
mixdata.txt	0,00002	0,001235	0,005786	0.000004
increase.txt	0,000022	0,001192	0,061303	0.000003
decrease.txt	0,000031	0,000637	0,0005	0.000098
sameelement.txt	0,000023	0,002637	0,061688	0.000002



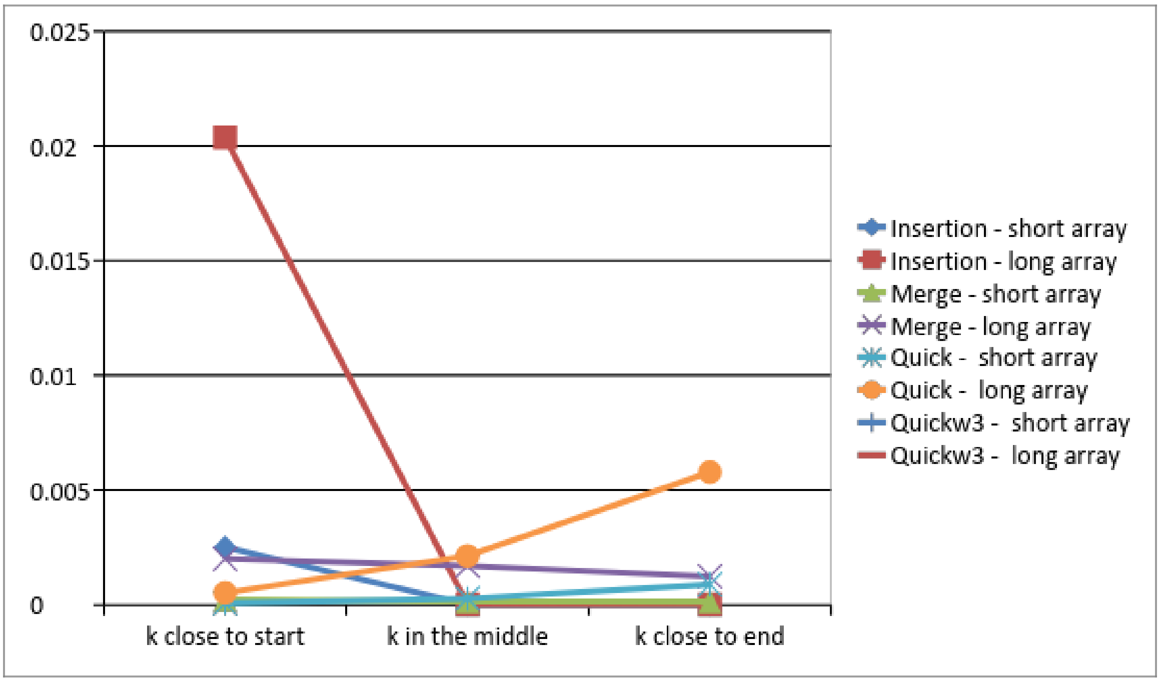
MixedInput IncreasingInput DecreasingInput RepeatingInput

(Runtime in seconds; n=5000 ; k=4990)

3.3. Comments and Conclusion:

a) Did the value of k has any impact on the performance?

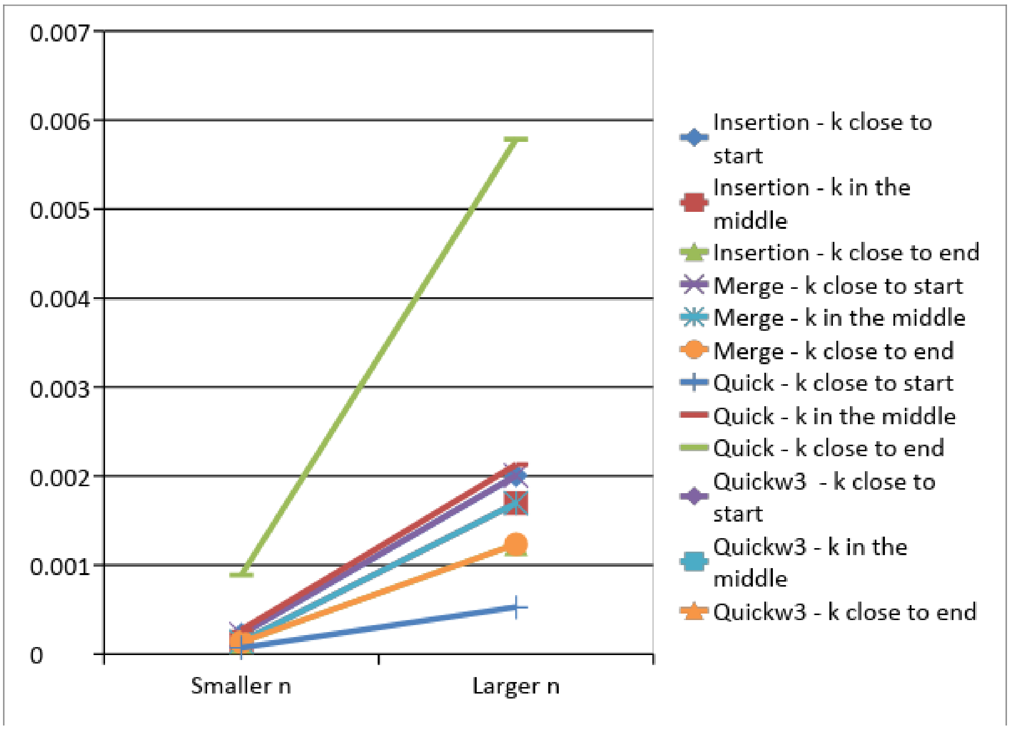
We will take our Mixed Input for instance:



As we see here, it affects especially the insertion sort with long array and quicksort with long array.

b) Did the value of n has any impact on the performance?

Once again, we will take our Mixed Input for instance:



We see that especially the insertion sort with k close to end works more slowly with mixed data.

According to this graph , n impacts on the performance.

c) Did the comparison with Theoretical Values match the expectation?

Expected results were as follows at the beginning of the experiment:

	InsertionSort	MergeSort	QuickSelect	Q /w Medof3
Best Case	O(n)	O(nlogn)	O(n)	Θ(nlogn)
Average Case	O(n^2)	O(nlogn)	O(n)	Θ(nlogn)
Worst Case	O(n^2)	O(nlogn)	O(n^2)	Θ(n^2logn)

According to our observations , our results are close to theoretical values.

Conclusion:

According to our results , QuickSelect Algorithm with median of three partitioning has the lowest time complexity and QuickSelect Algorithm which chooses first element as a pivot has the highest time complexity . As the time complexity is sorted in ascending order :

QuickSelect with Median of Three < Insertionsort < MergeSort < QuickSelect

As a result , QuickSelect with Median of Three is the most efficient way to find k th smallest number in a given input file among these four algorithms.