T.C.

MARMARA UNIVERSITY

ENGINEERING FACULTY

COMPUTER ENGINEERING

CSE 2246 Analysis of Algorithms

Project I

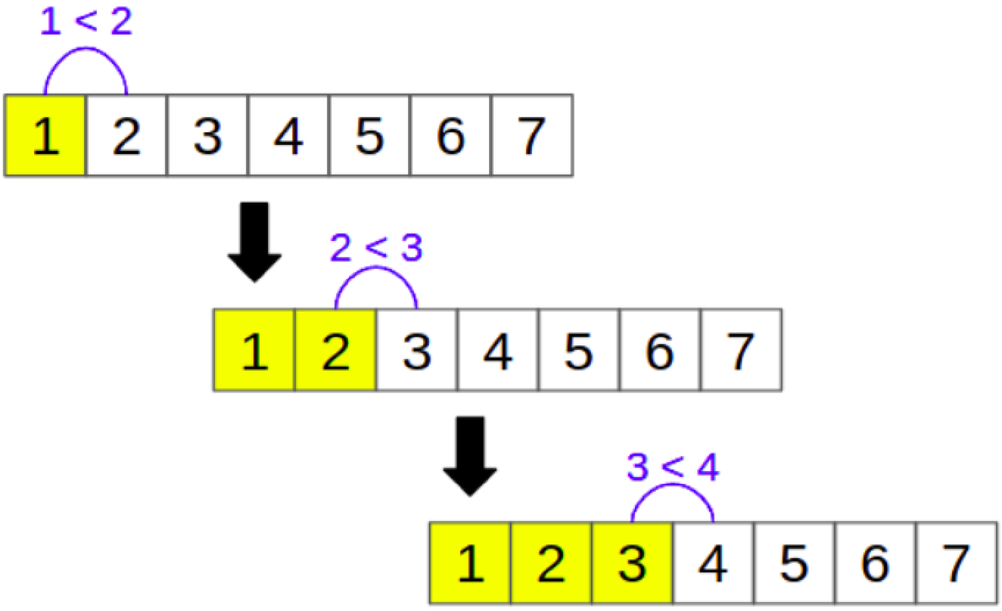
150119751 – Musa Meriç

INSERTION SORT

Best Case

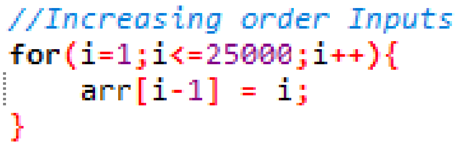
In the best case, our input values should be ordered from increasing order.

In the best-case scenario, time complexity is O(n) because it only checks each input value once(as shown in the graphic below).



<u>Example 1:</u>

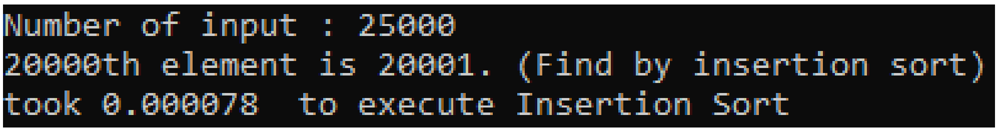
In this example , our input data are integers in increasing order from 1 to 25000.



```
//Increasing order Inputs
for(i=1;i<=25000;i++){
    arr[i-1] = i;
}
```

(We fill the array with the above code.)

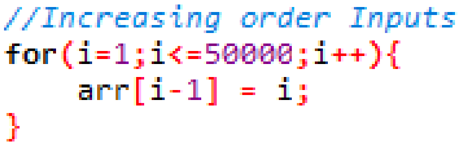
Because of it checks every input value only once it takes 0.000078 seconds to run the function.



```
Number of input : 25000
20000th element is 20001. (Find by insertion sort)
took 0.000078  to execute Insertion Sort
```

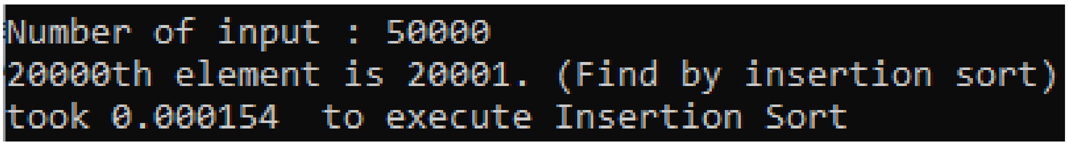
Example 2:

This time we are increasing our input size from 25000 to 50000 with increasing order as example 1 to check if it really provides O(n) time complexity for best-case insertion sort.



```
//Increasing order Inputs
for(i=1;i<=50000;i++){
    arr[i-1] = i;
}
```

(We fill the array with the above code.)



```
Number of input : 50000
20000th element is 20001. (Find by insertion sort)
took 0.000154  to execute Insertion Sort
```

Sorting 50000 input data took 0.000154 seconds for insertion sort.

As a result: When we increased the number of inputs from 25000 to 50000 , the time spent doubled. So it provides O(n) time complexity.

Worst Case

In the worst case, our input values should be ordered from decrasing order.

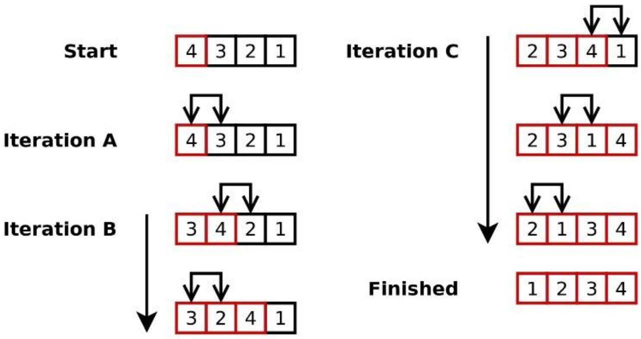
As seen in the graphic below in decrasing order , function must carry the new value from it’s position to first position every time one by one.

In the graphic when it sees input value ‘3’ , it run 1 time.

When it sees input value ‘2’ , it runs 2 times to carry it to first index in the array.

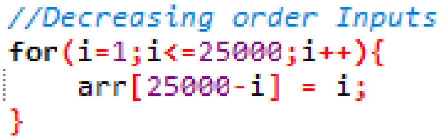
When it sees input value ‘1’ , it runs 3 times to carry it to first index in the array.

This is increasing like that and this situation gives us O(n^2) time complexity.



Example 3:

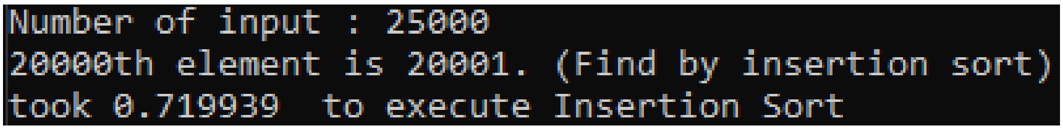
In this example , our input data are integers in decreasing order from 25000 to 1.



```
//Decreasing order Inputs
for(i=1;i<=25000;i++){
    arr[25000-i] = i;
}
```

(We fill the array with the above code.)

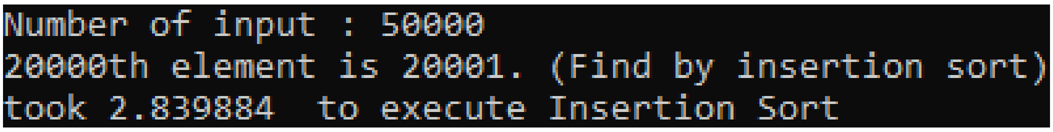
It takes 0.719939 seconds to sort the 25000 decrasing order inputs.



```
Number of input : 25000
20000th element is 20001. (Find by insertion sort)
took 0.719939  to execute Insertion Sort
```

Example 4:

In this example , our input data are integers in decreasing order from 50000 to 1.



```
Number of input : 50000
20000th element is 20001. (Find by insertion sort)
took 2.839884  to execute Insertion Sort
```

It takes 2.839887 seconds to sort the 50000 decrasing order inputs.

As a result:

If we compare ‘example 4’ and ‘example 3’

Example 3 took 0.719939 second to run the function. Example 4 has twice as much data as example 3. If we think 0.719939 as O(n^2) , O($(2n)^2$) should be 2.879756. So it provides approximately O(n^2).

If we compare ‘example 1’ and ‘example 3’

For the best case theoretical expectation is O(n) and for worst case O(n^2).

Our example 1 is best-case scenario with input size 25000 and example 3 is worst-case scenario with input size 25000.

Because of that example 3 must be square of example 1.

Example 1 time consuming is 0.000078 , Example 3 time consuming is 0.719939.

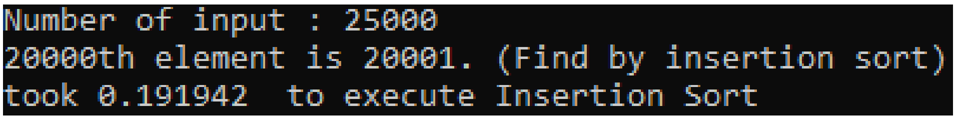
It satisfies the theoretical expectations.

Average Case

For the average case, the first half of our array should be in increasing order and the second half in decreasing order. It is close to the approximate average of the worst case and best case.

Example 5:

Our input values is array={1,2,3….12500,25000,24999,24998…12501}(first half increasing , second half decreasing order)



```
Number of input : 25000
20000th element is 20001. (Find by insertion sort)
took 0.191942  to execute Insertion Sort
```

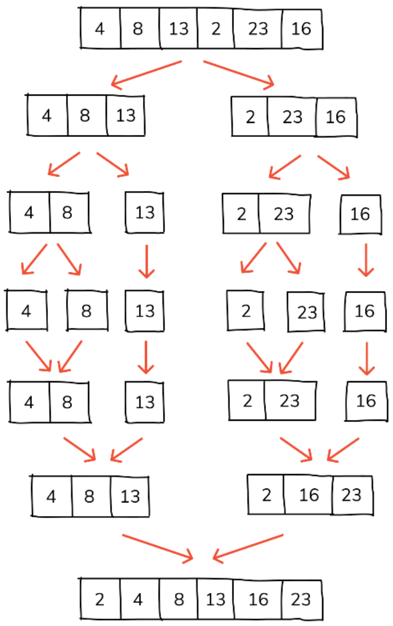
(Time consuming is 0.191942 for average case)

NOTE : For all Insertion Sort algorithms , “k” value doesn’t change the time consuming because we already sorted the whole array.

MERGE SORT

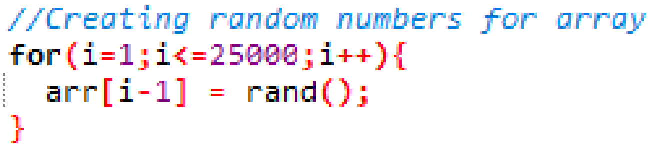
Best , Worst , Average Case

The merge-sort algorithm is a very fast algorithm that uses divide and conquer logic. The time complexity of merge Sort is O($n * \log(n)$) in all the 3 cases (worst, average and best) as the merge sort always divides the array into two halves and takes linear time to merge two halves.



Example 5:

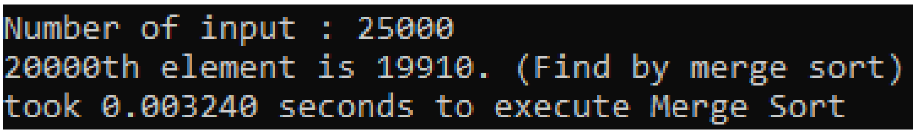
In this example we are generating 25000 input values with random values.



```
//Creating random numbers for array
for(i=1;i<=25000;i++){
    arr[i-1] = rand();
}
```

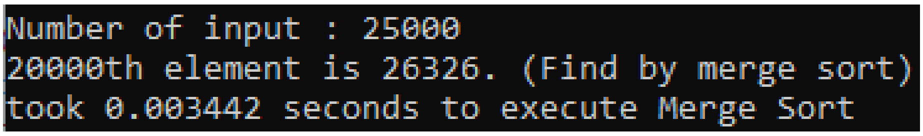
(Generating random values)

We run the function with 3 different arrays to prove that different input values does not make any important change in the time consuming of merge sort.



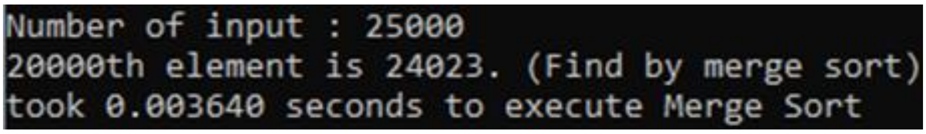
```
Number of input : 25000
20000th element is 19910. (Find by merge sort)
took 0.003240 seconds to execute Merge Sort
```

(Random Input 1)



```
Number of input : 25000
20000th element is 26326. (Find by merge sort)
took 0.003442 seconds to execute Merge Sort
```

(Random Input 2)



```
Number of input : 25000
20000th element is 24023. (Find by merge sort)
took 0.003640 seconds to execute Merge Sort
```

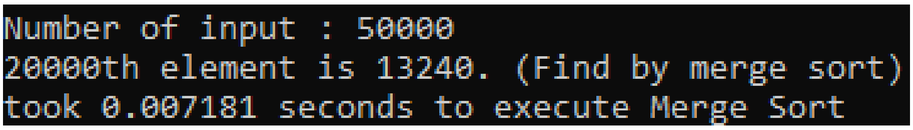
(Random Input 3)

As a result:

As seen in the 3 experiments, the time consuming is very close. The slight difference between the times is small delays due to the instant performance of the computer. This proved that merge sort takes the same time consuming in all 3 cases.

Example 6:

In this example , we are proving that merge sort takes O($n * \log(n)$) time complexity. To prove that we're getting twice inputs as in experiment 5.



```
Number of input : 50000
20000th element is 13240. (Find by merge sort)
took 0.007181 seconds to execute Merge Sort
```

If we say $n * \log(n)$ = 0.003442 , $2n * \log(2n))$ = 0.007181 (Approximately)

This proves that merge sort provides theoretical expextations.

QUICK SORT

The quicksort algorithm is a fast algorithm that uses divide and conquer logic. While doing quick sort we have 4 steps:

1- Choose first element as pivot
2- Place those larger than our pivot value to the right of our pivot.
3- Place those smaller than our pivot value to the left of our pivot.
4- Choose new pivot as first element that didn’t used as pivot and repeat.

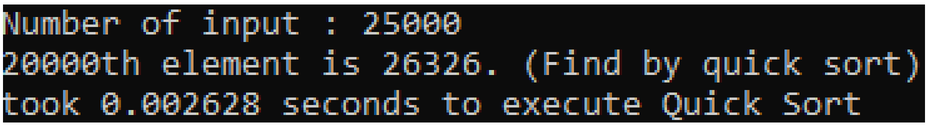
Best & Average Case

Best & average case have same time complexity O($n * \log(n)$).

In order to provide an average case, our input values need to be chosen randomly. When we compare the best case and the average case, since their values are very close, theoretically O($n * \log(n)$) is provided in both cases.

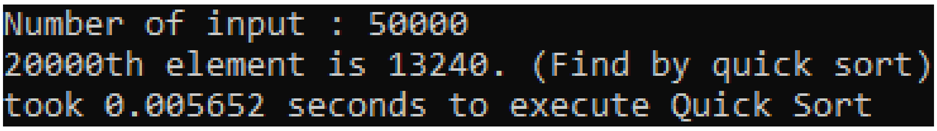
Example 7:

In this example we take 2 different inputs with random values. First input has 25000 random value , second input has 50000 random value. We will compare their time consuming.



```
Number of input : 25000
20000th element is 26326. (Find by quick sort)
took 0.002628 seconds to execute Quick Sort
```

(First input with size 25000 and random values)



```
Number of input : 50000
20000th element is 13240. (Find by quick sort)
took 0.005652 seconds to execute Quick Sort
```

(Second input with size 50000 and random values)

As a result:

When we use 25000 input size our time consuming is 0.002628

When we use 50000 input size our time consuming is 0.005652

If we compare their time consuming values we can see that it provides O($n * \log(n)$) theoretical time complexity.

Worst Case

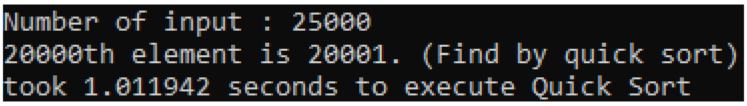
Complexity of Quick sort algorithm is based on which element you choose as pivot.

Worst case occurs in 3 different ways:

1- If elements are in increasing order, and we choose last or first element as pivot. Then this will be the worst case. And complexity will be O(n^2).
2- Array is already sorted in decreasing order.
3- If all elements are the same.

<u>Example 8:</u>

In this example , we take 25000 input value in increasing order and check it's time consuming.



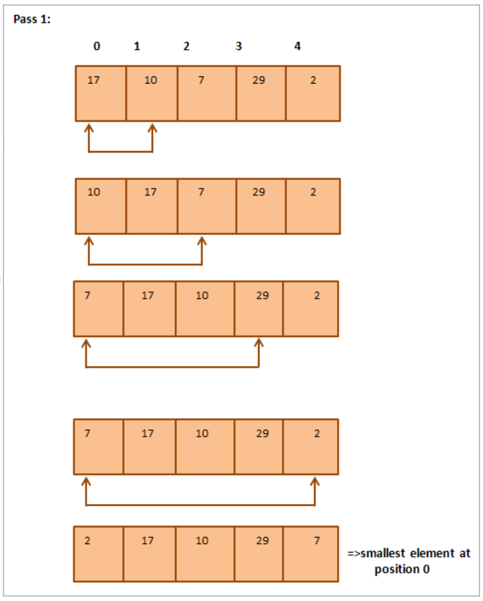
```
Number of input : 25000
20000th element is 20001. (Find by quick sort)
took 1.011942 seconds to execute Quick Sort
```

Because of array is in increasing order , the function took 1.011942 second to complete. If we compare this example with "example 7" we can clearly see that using increasing order in quick-sort algorithm results in worst-case.

PARTIAL SELECTION SORT

The concept used in selection sort helps us to partially sort the array up to k'th smallest element for finding the k'th smallest element in an array. Thus a partial selection sort is a simple selection algorithm that takes O(k*n) time to sort the array. (k = wanted smallest element)

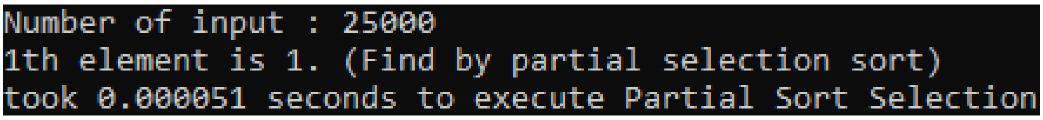
As we can see in the graph on the right side, when performing partial selection sort, the entire array is checked to find the smallest element and the smallest element is placed at the beginning.
We will repeat this process k time to find k'th smallest element.



Best Case

To get the best case in partial selection sort, we need to choose our k value as 1.

<u>Example 9:</u>



```
Number of input : 25000
1th element is 1. (Find by partial selection sort)
took 0.000051 seconds to execute Partial Sort Selection
```

As we can see here it takes very little time because algorithm checks every input only once and find first smallest element in the array. Because of it check every element once , it’s time complexity O(n).

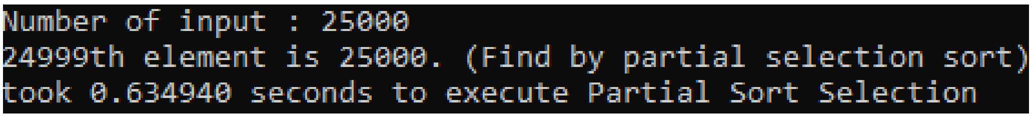
Average & Worst Case

Worst case occurs when we want to find largest element in the array. Worst case time complexity is O(n^2) because we will check every elements k time , so k will be equal to n , n*n = n^2.

For the average case if we choose k = n/2 that would give us average case.

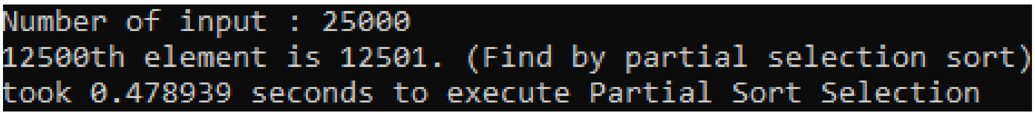
<u>Example 10:</u>

In this example , we find largest and median element in the array and compare their time consuming.



```
Number of input : 25000
24999th element is 25000. (Find by partial selection sort)
took 0.634940 seconds to execute Partial Sort Selection
```

(time consuming of largest element)



```
Number of input : 25000
12500th element is 12501. (Find by partial selection sort)
took 0.478939 seconds to execute Partial Sort Selection
```

(time consuming of median element)

In first example k=n , O(k*n) = O(n*n) = 0.634940

In second example k = n/2 , O(k*n) = O ((n/2)*n) = 0.478939

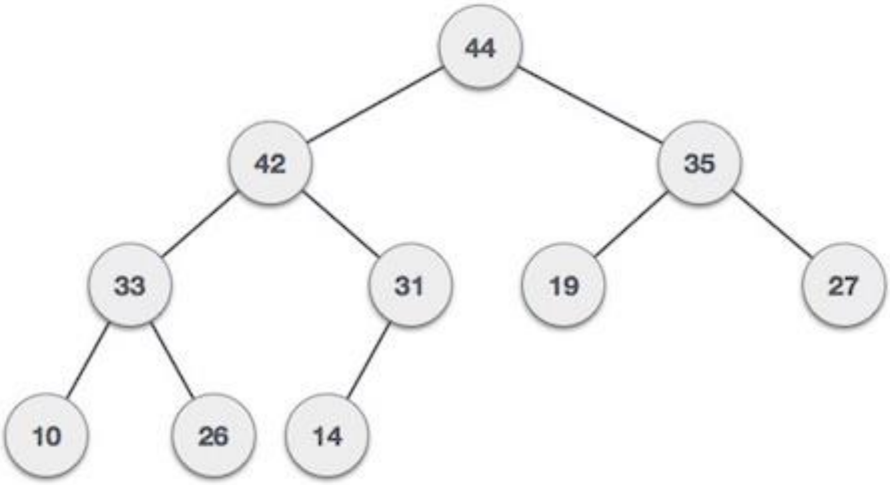
When we check their time consuming it is approximately corresponds to the theoretical values.

PARTIAL HEAP SORT

A max-heap is a complete binary tree in which the value in each internal node is greater than or equal to the values in the children of that node.

Inserting a new key takes O(Log n) time. Based on this, it takes O($n * \log(n)$) time to max-heap an array with n elements.

rootRemove(): Removes the maximum element from max-heap. The time complexity of this operation is O(Log n).

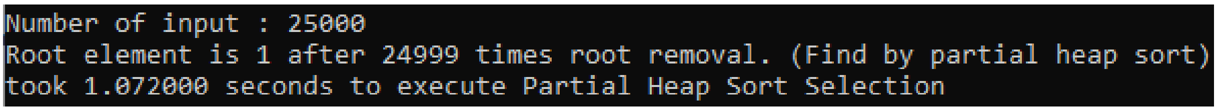


Worst Case

If we want to find smallest element in max-heap , the function must run the “root remove” function n times and that results in worst case.

Example 11:

In this example , we run “root remove” function n times.



```
Number of input : 25000
Root element is 1 after 24999 times root removal. (Find by partial heap sort)
took 1.072000 seconds to execute Partial Heap Sort Selection
```

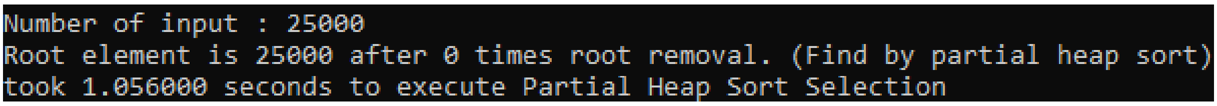
Because of we did a lot of root remove , it took 1.072 second.

Best Case

If we want to find largest element in max-heap , the function will run the “root remove” function only 1 times and that will give us best case.

Example 12:

In this example we are not doing any root remove operation.



```
Number of input : 25000
Root element is 25000 after 0 times root removal. (Find by partial heap sort)
took 1.056000 seconds to execute Partial Heap Sort Selection
```

Best case time consuming is 1.056 with 25000 input value.

NOTE : The root remove function works very fast as it takes O(Log n) time and this does not have a big impact on the time spent.

QUICK SELECT

Quick select is a very similar algorithm to quick sort. It is an optimized way to find the k'th smallest/largest element in an unsorted array. The partition part of the algorithm is same as that of quick sort. After the partition function arranges the elements in list according to the pivot and returns the pivot_index, instead of recursing both sides of the pivot index, we recurse only for the part that contains our desired element.

The time complexity for the average case for quick select is O(n) (reduced from O(n*logn) — quick sort). The worst case time complexity is still $O(n^2)$ but by using a random pivot, the worst case can be avoided in most cases. So, on an average quick select provides a O(n) solution to find the k'th largest/smallest element in an unsorted list.

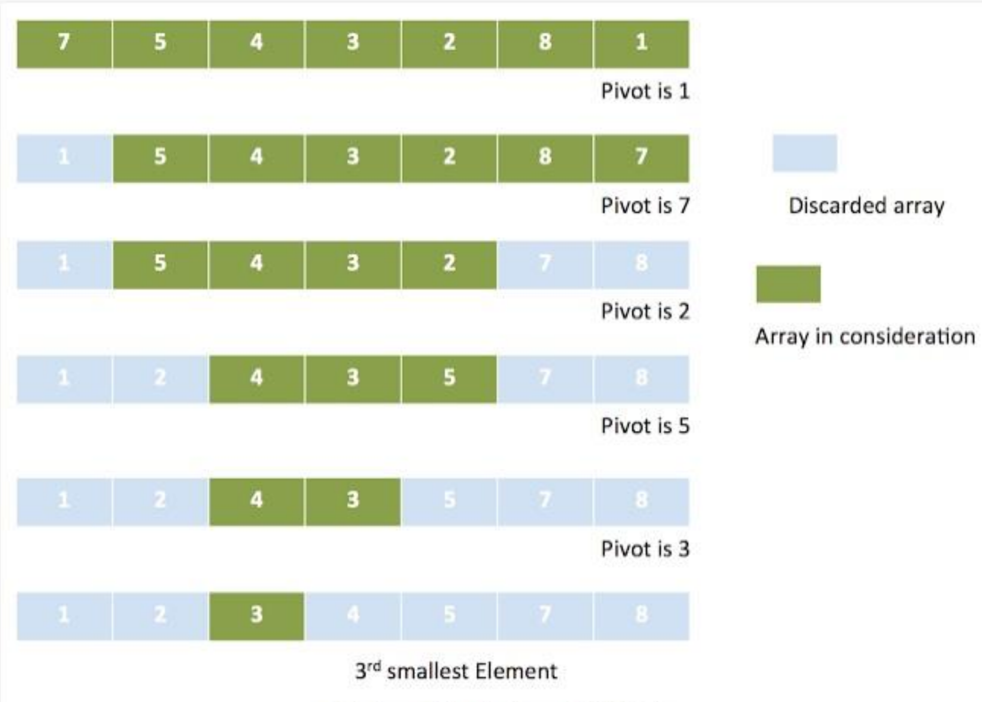


Fig1 : Demonstration the steps on Quick Select

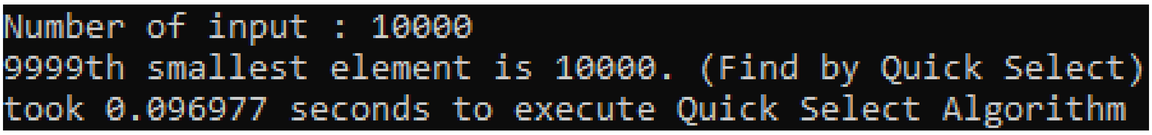
Worst Case

In worst case of quick selection algorithm , input should be in increasing order.

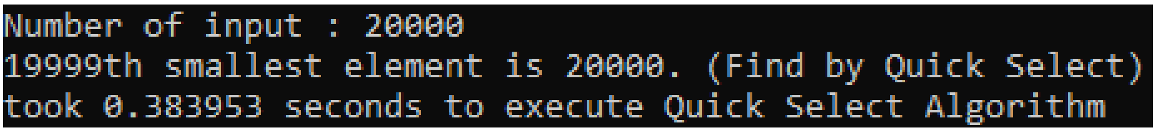
The worst-case corresponds to the longest possible execution of the algorithm. It’s the one in which the k’th largest element is the last one standing, and there are “n” recursive calls.

<u>Example 13:</u>

In this example , we have 2 different arrays with size of 10000 and 20000 in increasing order.



```
Number of input : 10000
9999th smallest element is 10000. (Find by Quick Select)
took 0.096977 seconds to execute Quick Select Algorithm
```



```
Number of input : 20000
19999th smallest element is 20000. (Find by Quick Select)
took 0.383953 seconds to execute Quick Select Algorithm
```

If we think as

O(n^2) = 0.096977

O($(2n)^2$) should be equal to 0.387908.

We found O($(2n)^2$) = 0.383953. Thus, we see that they are approximately the same and corresponds to theoretical expectations.

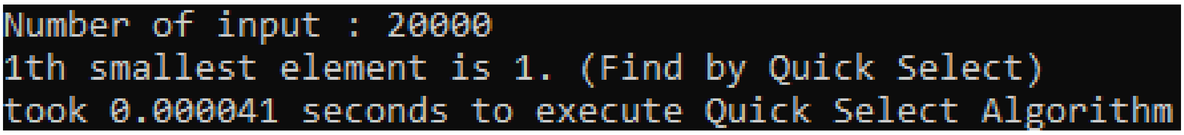
Best Case

The best-case corresponds to the smallest possible execution of the algorithm.

It’s the one in which the k’th smallest element is the first one standing, and there are “1” recursive calls.

<u>Example 14:</u>

To find the best case , we will use an increasing order input array and find it smallest element.



```
Number of input : 20000
1th smallest element is 1. (Find by Quick Select)
took 0.000041 seconds to execute Quick Select Algorithm
```

We took 20000 increasing order input and find it’s smallest element.

That operation took 0.000041 seconds to complete.

When we compare it with Example 13’s second input:

If we say O(n) = 0.000041

O(n^2) should be equal to 0.001681 but we found that O(n^2) = 0.383953.

Thus, we see that they are not close to each other and doesn’t corresponds to theoretical expectations.

MEDIAN OF THREE SELECTION

Median of three method is an optimized version of quick select algorithm. The median of three is looking at the first, middle and last elements of the array, and choose the median of those three elements as the pivot. Then continues with the part that contains our desired element.

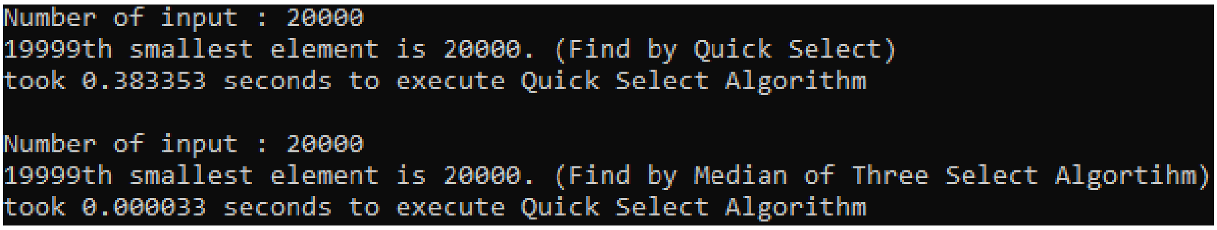
Using the median-of-3 method will allow us to never have to search the entire list for the right partition but rather start somewhere in the middle. Therefore, we can avoid the worst-case run-time of O(n^2) and settle at the average run-time of O(n*logn).

For the best and average case of median of three selection algorithm is very similar to quick selection algorithm and the time complexity is O(n*logn).

Worst Case

<u>Example 15:</u>

In this example , we are comparing quick select algorithm with median of three selection algorithm. In two case we have 20000 increasing order input values and we set k as latest element.



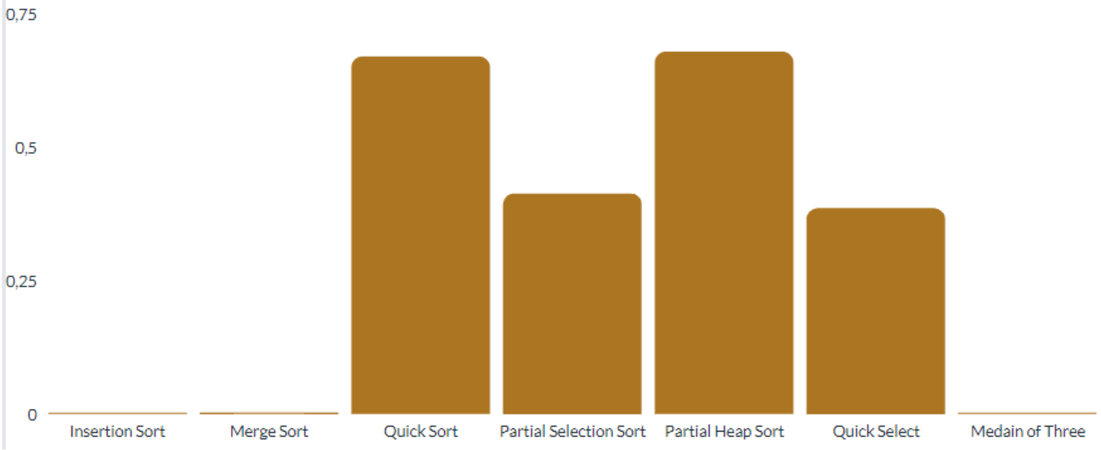
```
Number of input : 20000
19999th smallest element is 20000. (Find by Quick Select)
took 0.383353 seconds to execute Quick Select Algorithm

Number of input : 20000
19999th smallest element is 20000. (Find by Median of Three Select Algortihm)
took 0.000033 seconds to execute Quick Select Algorithm
```

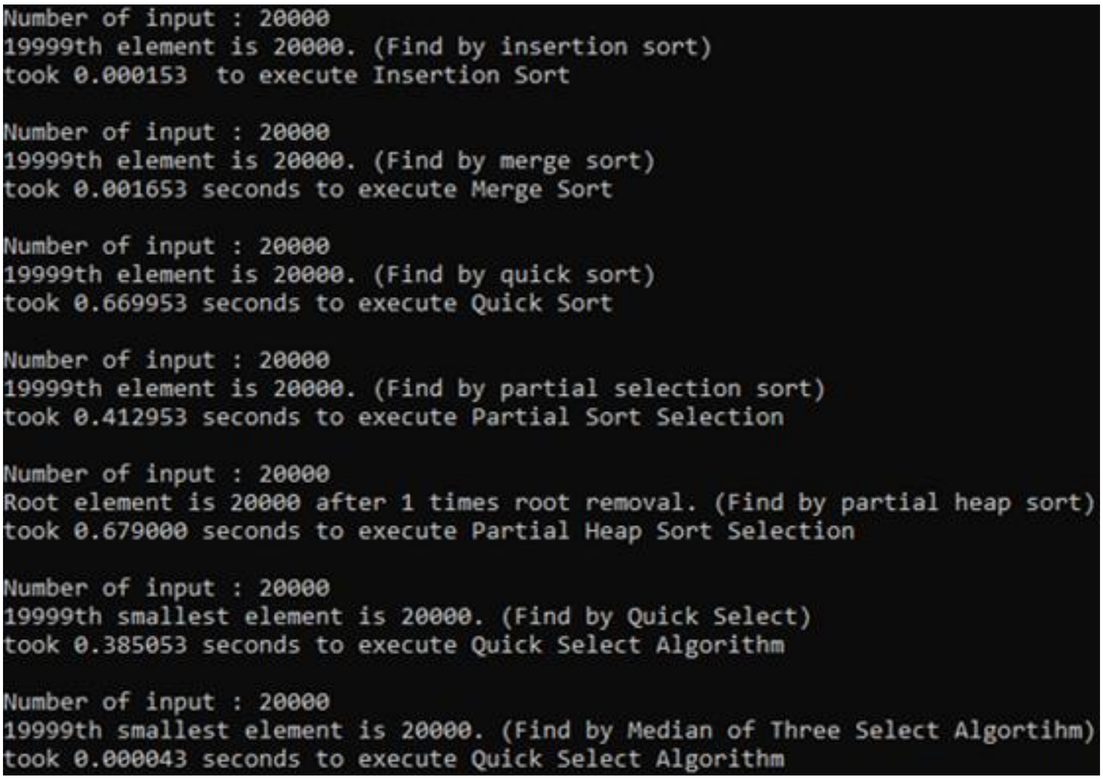
When we compare time consuming of two function , we can see that median of three helps us to avoid worst case scenario of quick select algorithm.

COMPARING FUNCTIONS WITH DIFFERENT INPUTS

Scenario 1:

In this scenario , we have 20000 input value in increasing order and we want largest element in the array.





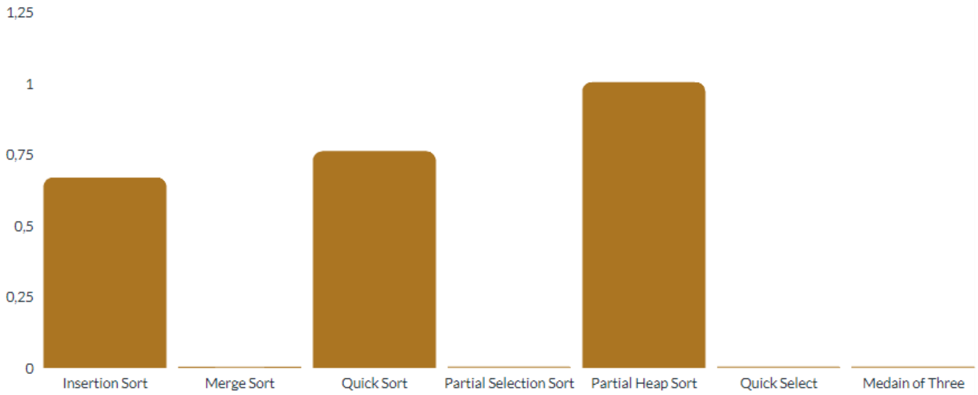
```
Number of input : 20000
19999th element is 20000. (Find by insertion sort)
took 0.000153  to execute Insertion Sort

Number of input : 20000
19999th element is 20000. (Find by merge sort)
took 0.001653 seconds to execute Merge Sort

Number of input : 20000
19999th element is 20000. (Find by quick sort)
took 0.669953 seconds to execute Quick Sort

Number of input : 20000
19999th element is 20000. (Find by partial selection sort)
took 0.412953 seconds to execute Partial Sort Selection

Number of input : 20000
Root element is 20000 after 1 times root removal. (Find by partial heap sort)
took 0.679000 seconds to execute Partial Heap Sort Selection

Number of input : 20000
19999th smallest element is 20000. (Find by Quick Select)
took 0.385053 seconds to execute Quick Select Algorithm

Number of input : 20000
19999th smallest element is 20000. (Find by Median of Three Select Algortihm)
took 0.000043 seconds to execute Quick Select Algorithm
```

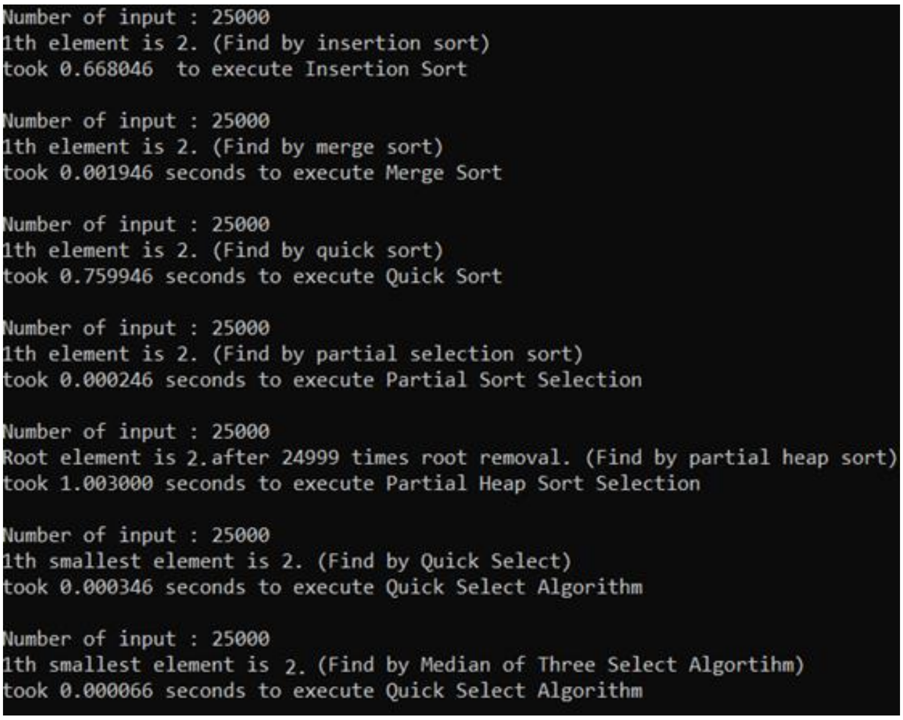
Insertion sort algorithm works very fast since the array is already sorted.

Because of we want largest element , partial selection sort results in worst case and that give large time consuming.

Scenario 2:

In this scenario , we have 25000 input value in decreasing order and we want smallest element in the array.





```
Number of input : 25000
1th element is 2. (Find by insertion sort)
took 0.668046  to execute Insertion Sort

Number of input : 25000
1th element is 2. (Find by merge sort)
took 0.001946 seconds to execute Merge Sort

Number of input : 25000
1th element is 2. (Find by quick sort)
took 0.759946 seconds to execute Quick Sort

Number of input : 25000
1th element is 2. (Find by partial selection sort)
took 0.000246 seconds to execute Partial Sort Selection

Number of input : 25000
Root element is 2.after 24999 times root removal. (Find by partial heap sort)
took 1.003000 seconds to execute Partial Heap Sort Selection

Number of input : 25000
1th smallest element is 2. (Find by Quick Select)
took 0.000346 seconds to execute Quick Select Algorithm

Number of input : 25000
1th smallest element is 2. (Find by Median of Three Select Algortihm)
took 0.000066 seconds to execute Quick Select Algorithm
```

Because of we want smallest element Partial Heap Sort Algorithm takes long time because it has to remove the root n times.

Insertion sort takes longer time compared to Scenario 1 because this time insertion sort is in worst case.

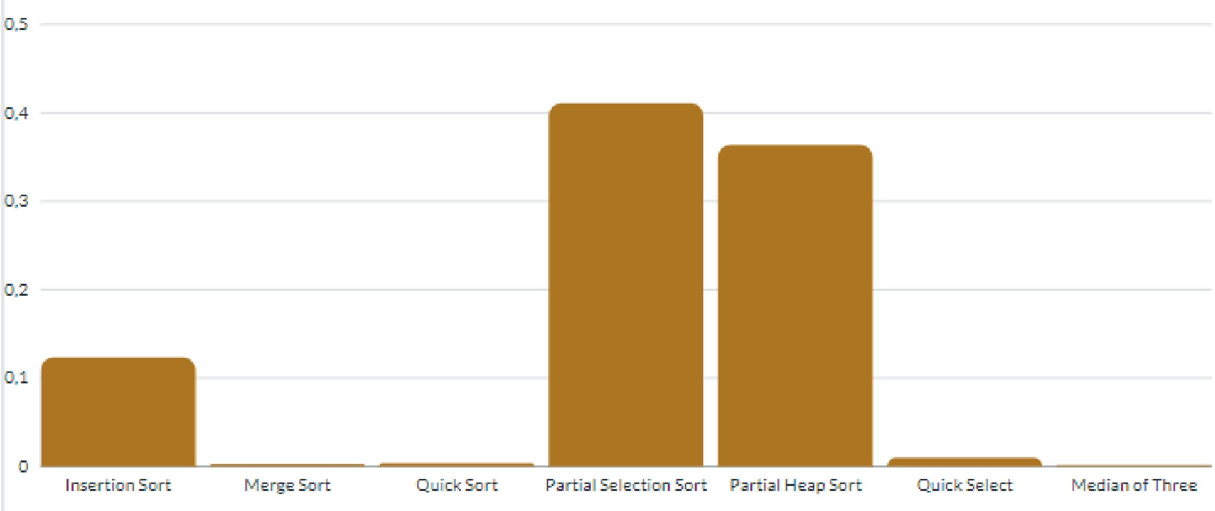
Because of we want smallest element , partial selection sort results in best case and that give small time consuming.

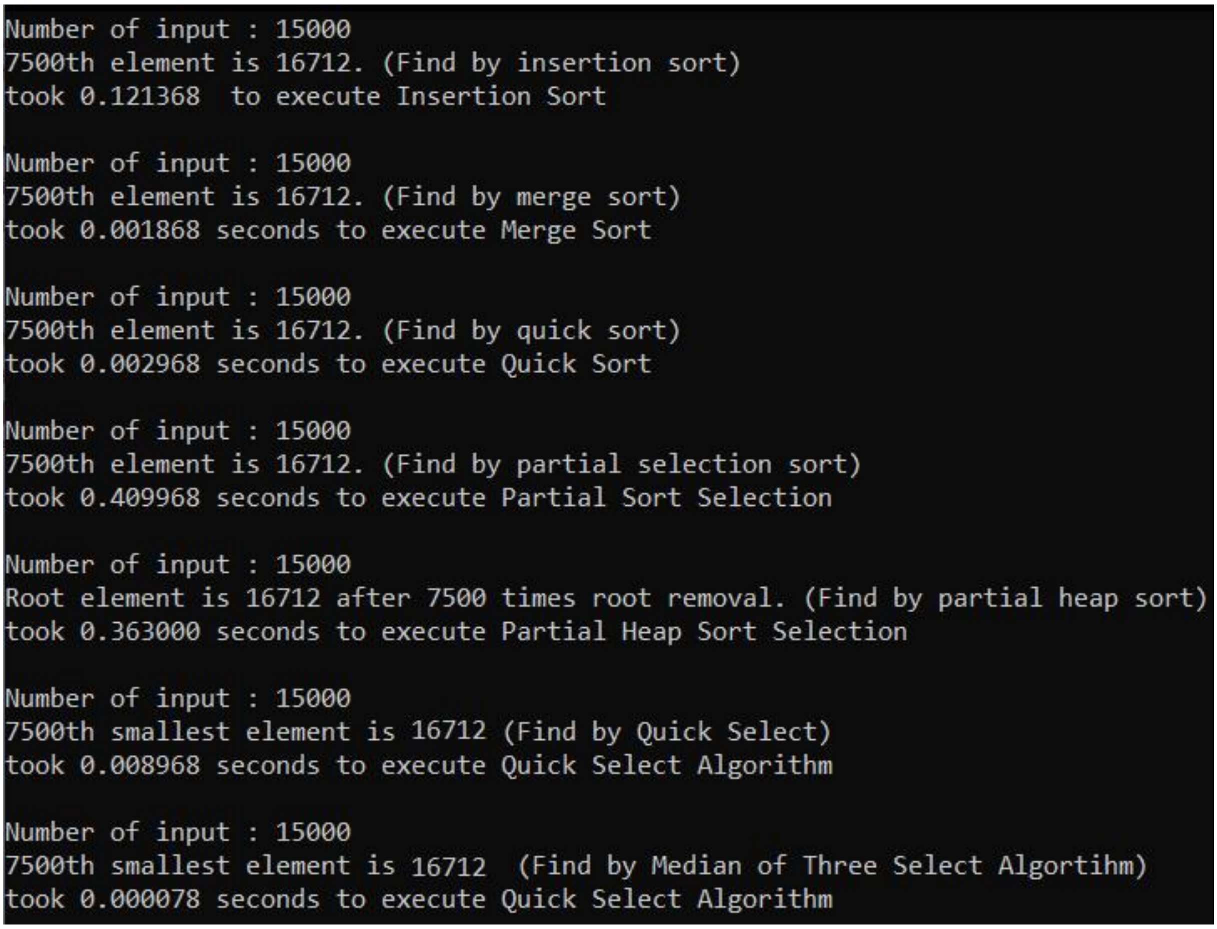
Quick select runs faster than Scenario 1 because this time we are searching the first element and this results in best case of quick selection algorithm.

Scenario 3:

In this scenario , we are taking 2 different random input arrays with size of 15000 & 30000 and checking their middle element.

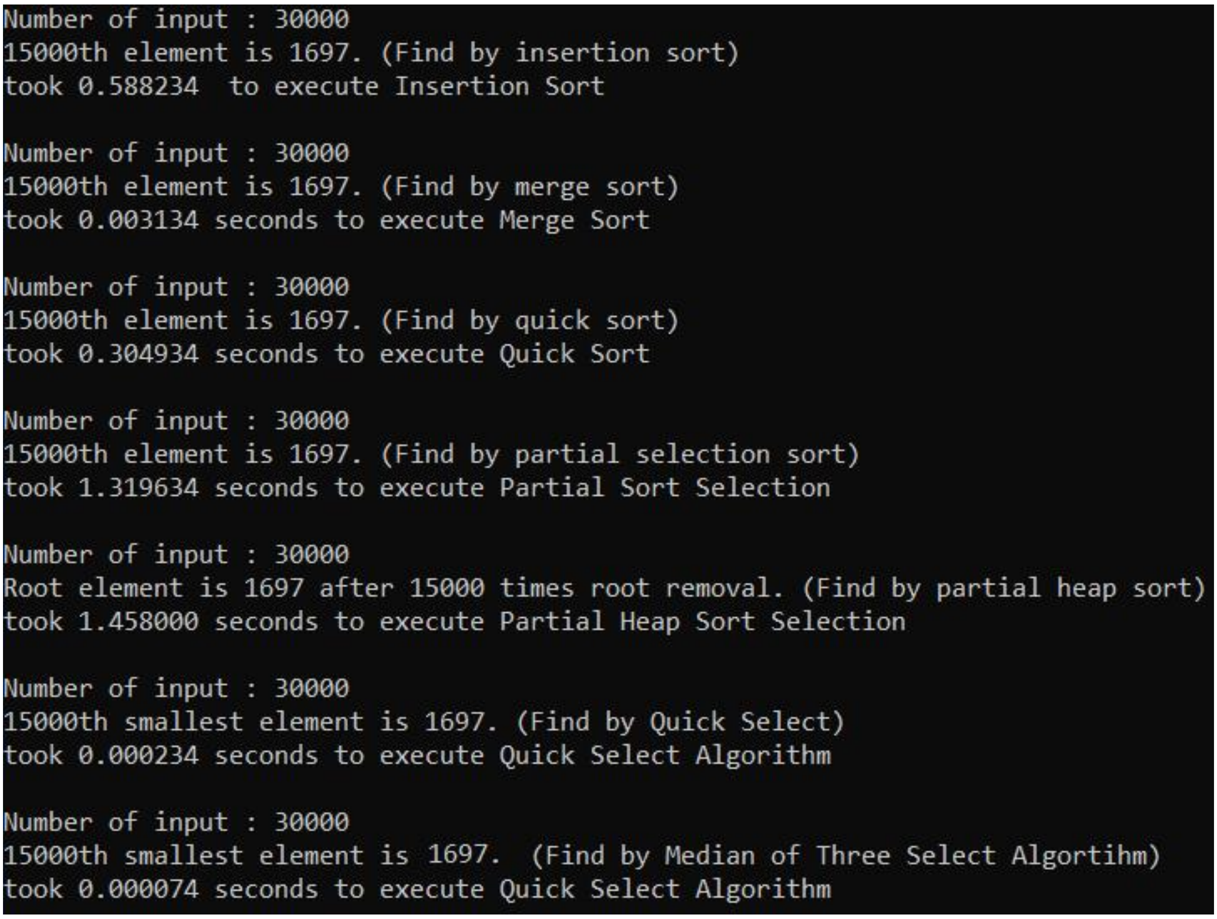
First array with 15000 random input values time complexity :



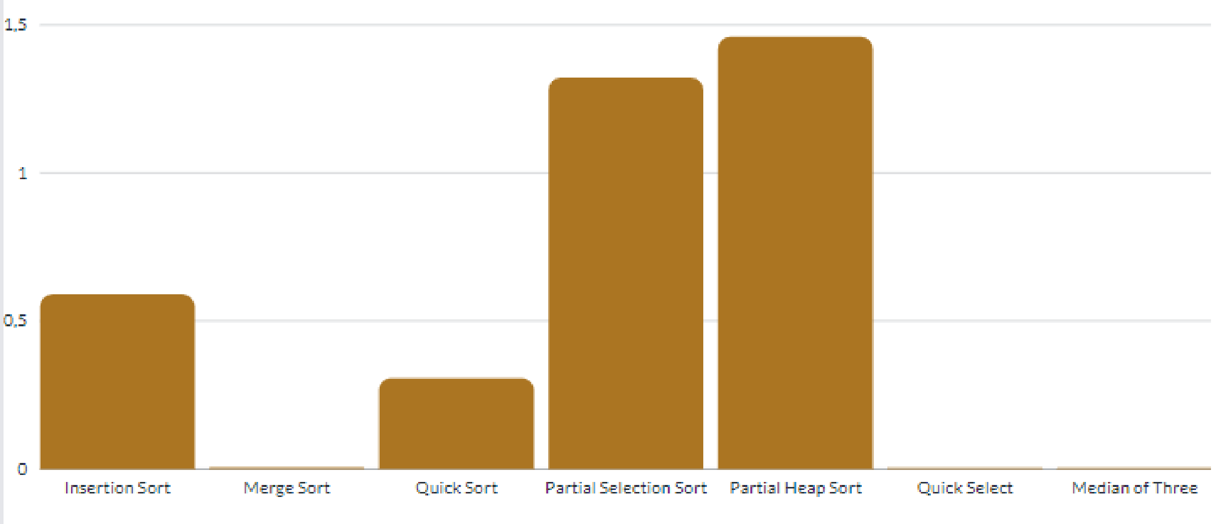


```
Number of input : 15000
7500th element is 16712. (Find by insertion sort)
took 0.121368  to execute Insertion Sort

Number of input : 15000
7500th element is 16712. (Find by merge sort)
took 0.001868 seconds to execute Merge Sort

Number of input : 15000
7500th element is 16712. (Find by quick sort)
took 0.002968 seconds to execute Quick Sort

Number of input : 15000
7500th element is 16712. (Find by partial selection sort)
took 0.409968 seconds to execute Partial Sort Selection

Number of input : 15000
Root element is 16712 after 7500 times root removal. (Find by partial heap sort)
took 0.363000 seconds to execute Partial Heap Sort Selection

Number of input : 15000
7500th smallest element is 16712 (Find by Quick Select)
took 0.008968 seconds to execute Quick Select Algorithm

Number of input : 15000
7500th smallest element is 16712  (Find by Median of Three Select Algortihm)
took 0.000078 seconds to execute Quick Select Algorithm
```

Second array with 30000 random input values time complexity :



```
Number of input : 30000
15000th element is 1697. (Find by insertion sort)
took 0.588234  to execute Insertion Sort

Number of input : 30000
15000th element is 1697. (Find by merge sort)
took 0.003134 seconds to execute Merge Sort

Number of input : 30000
15000th element is 1697. (Find by quick sort)
took 0.304934 seconds to execute Quick Sort

Number of input : 30000
15000th element is 1697. (Find by partial selection sort)
took 1.319634 seconds to execute Partial Sort Selection

Number of input : 30000
Root element is 1697 after 15000 times root removal. (Find by partial heap sort)
took 1.458000 seconds to execute Partial Heap Sort Selection

Number of input : 30000
15000th smallest element is 1697. (Find by Quick Select)
took 0.000234 seconds to execute Quick Select Algorithm

Number of input : 30000
15000th smallest element is 1697.  (Find by Median of Three Select Algortihm)
took 0.000074 seconds to execute Quick Select Algorithm
```



When we compare two array with each other , we can see that quicksort algorithm time consuming is increasing when it’s “k” value increased.

Merge sort and Median of Three sorting algorithm doesn’t change too much when array size increased because their time complexity are same in all cases (O(nlogn)).

As a result : We found that merge sort & median of three selection algorithm runs really fast in generally all cases.

HOW WE CALCULATE THE TIME CONSUMING

Since algorithms such as merge and median of three run very quickly, the time consuming can be obtained as 0 in some cases. To prevent this situation, we run the algorithm 100 times and divide the time we get by 100. As we run the sort algorithms over and over, we are restoring the array back to its original state, and this takes extra time. In order to prevent this situation, we calculate our array copy time at the beginning of the algorithm and subtract this time from the time we get.

NOTE : While doing this project, we completed all the steps together on discord.