T.C.

MARMARA UNIVERSITY

FACULTY of ENGINEERING

COMPUTER ENGINEERING DEPARTMENT

CSE2046 - Analysis of Algorithms

Homework #1 Report

Group Members

150119639 - Erdem PEHLİVANLAR

150120056 - Haldun Halil OLCAY

170517033 - Yasin Alper BİNGÜL

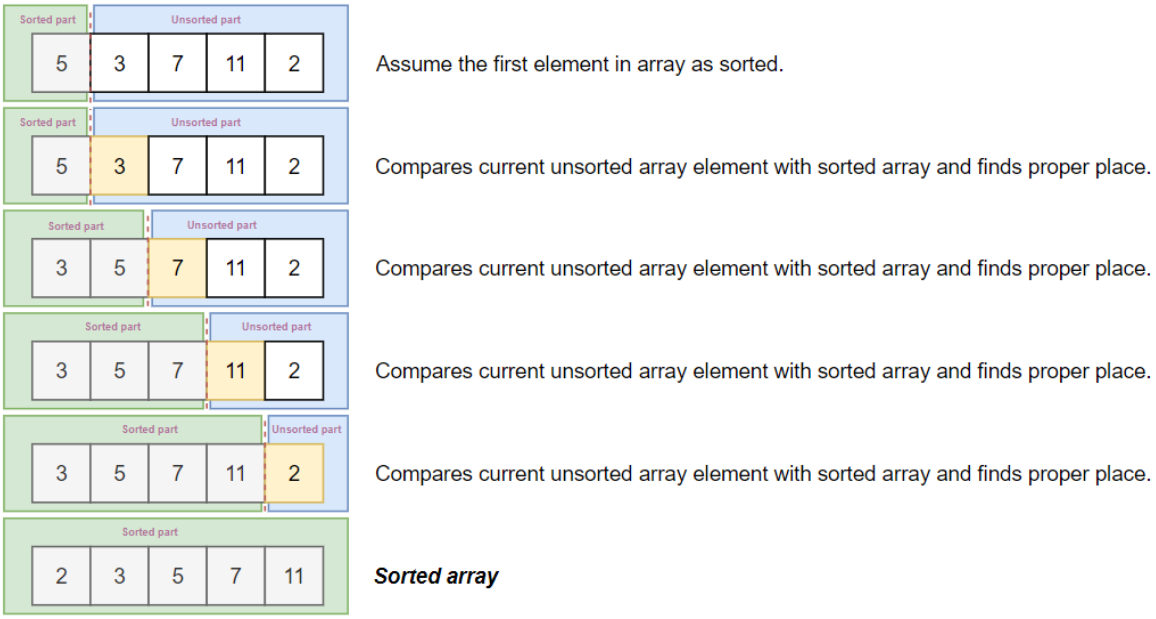
SORTING ALGORITHMS

1. Insertion-Sort Algorithm

Insertion sort is a simple sorting algorithm that sorts the elements in the arrays/lists. As mentioned in resources, insertion sort works similarly with sorting cards by hand in a card game.

We can imagine slicing the array into two pieces virtually. Sorted elements are placed in the left part of this piece. Unsorted elements are placed in the other part. In the beginning, the first element is assigned as the lowest element. Also this element is considered the only sorted element in the left part. Then the algorithm selects a number from unsorted elements. If the selected element is greater than the first element, the selected element will be placed into the right, otherwise into the left. Each iteration,like the previous one, algorithm selects a number from unsorted elements, compares it with the current index of sorted elements and puts it into the right ordered place.

We can see an example down below using the algorithm step by step.



When applying insertion sort, there might be some cases that happen the most number of comparisons or vice versa. Suppose, if the unsorted input array elements in descending order and ,if the user wants to sort the entire array in ascending order, the maximum number of comparisons happens. Each element has to be compared with the other elements. So that for every nth element, (n-1) number of comparisons happens. Thus, as a result total number of comparisons will be n*(n-1). That case occurs the worst case.

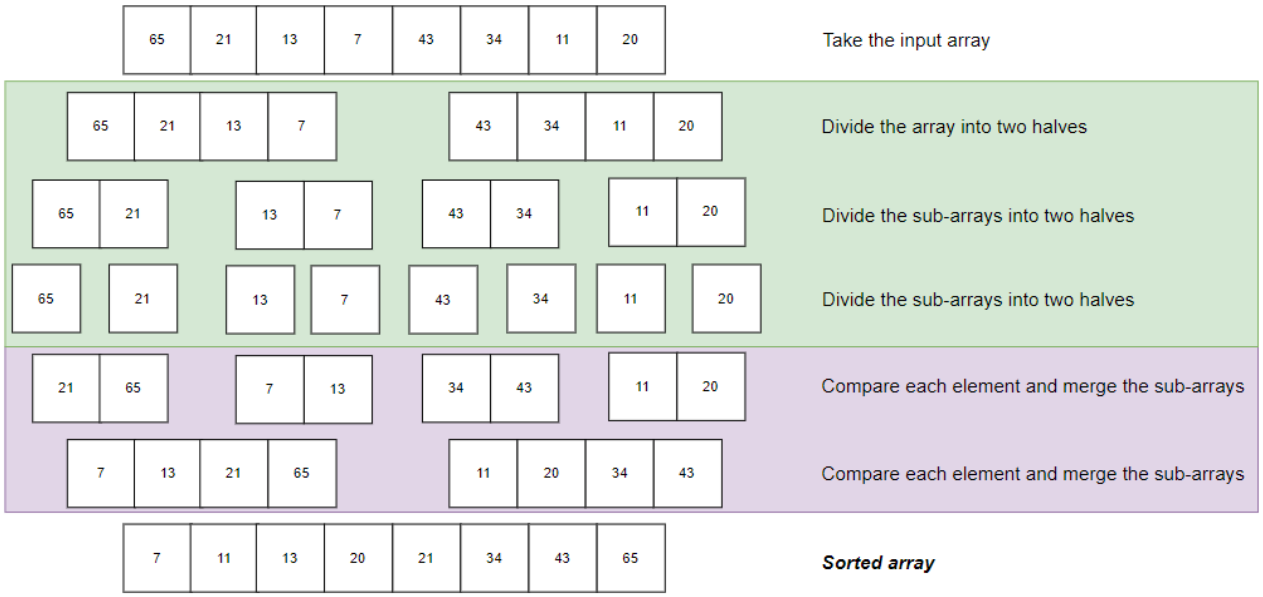
On the other hand, if the array is already sorted, then the minimum number of comparisons happens. In that case, the outer loop runs for n times and the inner loop doesn't. So that, there will be only n times comparisons. That case occurs the best case of insertion sort algorithm.

2. Merge-Sort Algorithm

Merge-sort algorithm is a divide and conquer algorithm that sorts the given lists/arrays. In an ordinary divide and conquer algorithm, the algorithm completes similar steps. Firstly, we simply take the problem and divide it into small parts. After we prepare solutions to these subproblems, the algorithm combines the whole subproblems to solve the main problem.

Like the other divide and conquer algorithms, in the merge-sort algorithm we divide the array into two halves. Each iteration this process continues until try to perform dividing on an sub-array that has size of 1. After that, the algorithm merges the sorted subarrays to gradually sort the entire array. Merge process is the most important part in this algorithm. In the merge process, the algorithm compares two sub-arrays. If iteration didn't reach the end of the array elements, compares current elements of both arrays, copies smaller elements into a bigger array. Then moves the index of the element containing the smaller element. This process continues until the sub-arrays are finished.

We can summarize the working principle of Merge-Sort with an example like down below.



When applying the merge-sort algorithm, there might be some cases that happen the most number of comparisons or vice versa. After dividing the unsorted array into two halves operations, if all numbers have to be compared and switched, then that case will occur the worst case. For example if one half of an unsorted array is {11,13,15,17} and the other half is {12,14,16,18}, then every element of the array will be compared at-least once. In that case occurs the worst case of merge-sort algorithm.

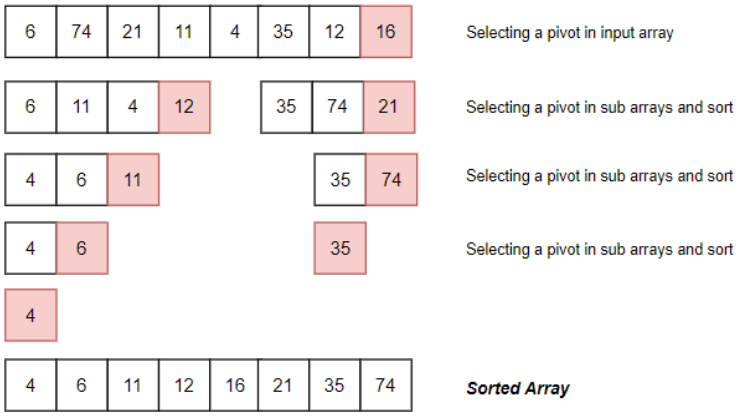
On the other hand, if the input array is already sorted, before merging there will not be switching operations and extra comparisons. So that case occurs the best case of merge-sort algorithm.

3. Quick-Sort Algorithm

Quicksort is a divide-and-conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. For this reason, it is sometimes called partition-exchange sort. While dividing the array, the algorithm compares the pivot element with each element in the array. Then puts the elements that less than pivot into the left side, elements that greater than pivot into the right side of the array. Also the divided left and right part of the array are divided using the same approach. This process continues until each subarray has a single element. In this step, the elements are already sorted. As a final step, the whole elements merge and form a sorted array.

We can illustrate the algorithm step by step as it seems below.

Detailed explanation of example.



When applying a quick-sort algorithm, there might be some cases that happen the most number of comparisons or vice versa. If the chosen pivot element is the greatest or smallest element in the array, then this case occurs the worst case of a quick-sort algorithm. In that case the pivot element is at the end of the array. This situation causes one subarray to always empty and another subarray has n-1 elements. Thusi quick-sort is called only on the current subarray. So, the quick-sort algorithm has better performance in scattered pivots.

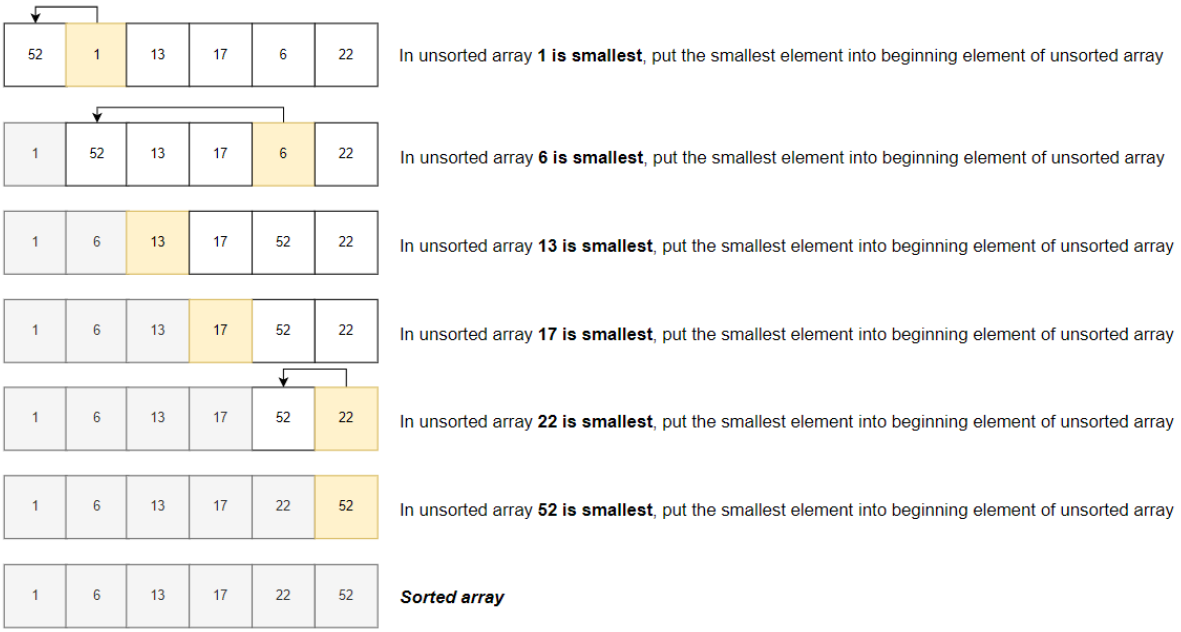
In the other way, if the pivot element would be selected as the middle element of the array, it occurs the best case of a quick-select algorithm. Because the subarrays are already divided into two halves, there is no need to replace the elements at all.

4. Partial Selection-Sort Algorithm

Selection-sort is a sorting algorithm that selects the smallest element from an unsorted array. It replaces the smallest element with the beginning element in an unsorted array. At each iteration, the algorithm does the same process until it finishes the elements in the unsorted array. This algorithm completes the run efficiently for selecting multiple elements. Therefore, if we want to select just one element (like kth smallest element), the algorithm should be improved.

Instead of sorting the entire array, we can do partial sorting. Partial sorting algorithm, partially sorts the array up to kth smallest element for finding the kth smallest element in an array. Thus, the algorithm iterates sorting k times, not n times (k<n).

We can summarize the working principle of Partial Selection-Sort with an example like down below.



When the algorithm is partial selection-sort, if we have an array in descending order and if we want to sort in ascending order, in that case the worst case of the algorithm. Because if the user wants to reach the kth element, and the array is in reversed order, there will be a maximum number of comparisons and moving operations.

On the other hand, if the array is already sorted, in that case there will be a minimum number of comparisons and causes the best case of the algorithm.

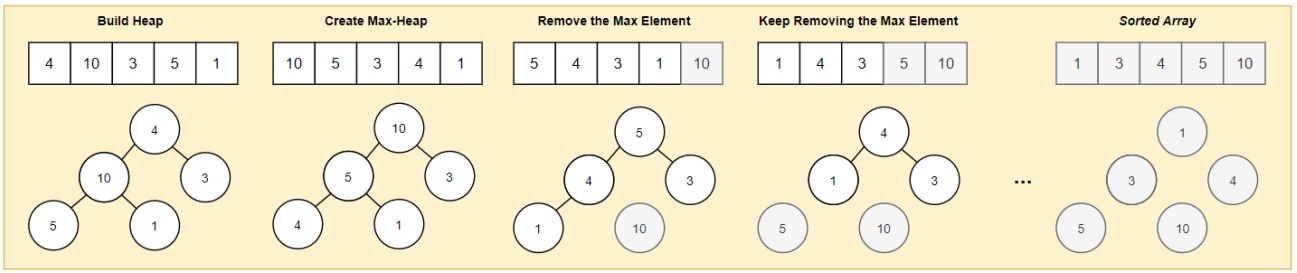
5. Partial Heap-Sort Algorithm

Heap sort algorithm is a comparison-based sorting algorithm based on Binary Heap. As mentioned in resources, the heap sort algorithm is similar to the selection sort algorithm. Because both algorithms firstly find the min/max element and puts at the beginning of the array. Then the same process repeats for the remaining elements.

As the first step, the algorithm creates a max-heap from input data with the algorithm of heapify for sorting descending order. After creating max-heap, the heap root has the maximum value. Each iteration root will be the maximum value in the max-heap. Algorithm replaces it with the last element of the heap and constructs a new max-heap which has reduced size by 1. This process continues while the size of the heap is greater than 1.

The heap sort algorithm performs the same performance in both worst case, best case.

We can illustrate the algorithm step by step as it seems below.

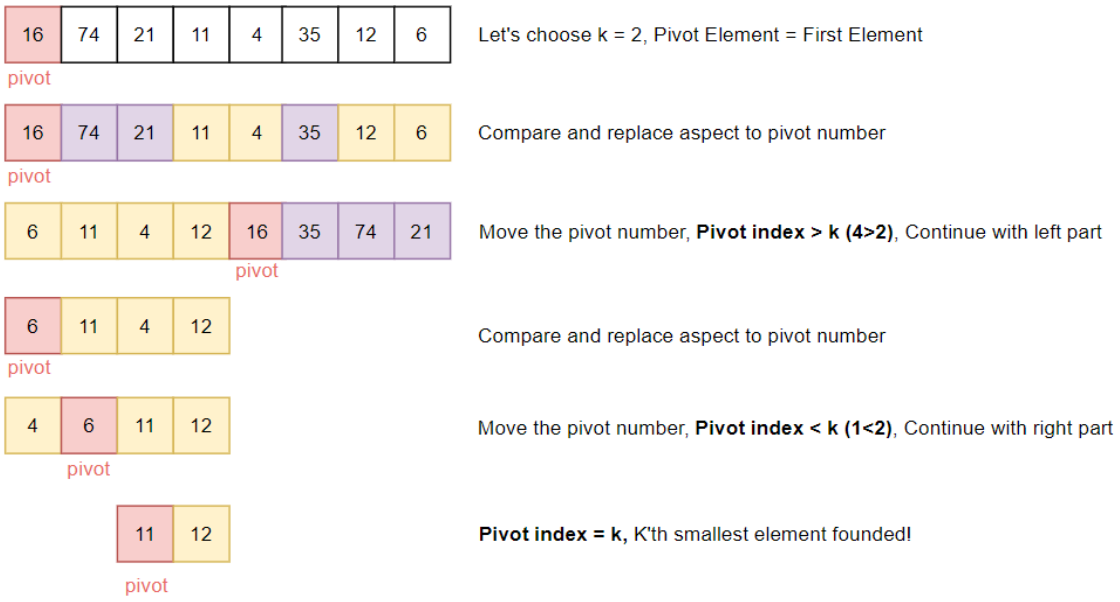


6. Quick-Select Algorithm

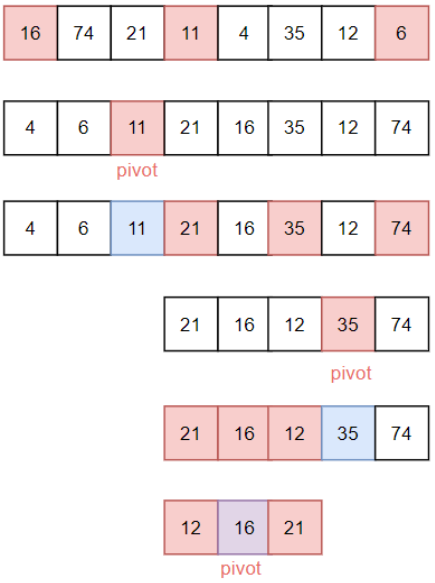
The quick select algorithm is similar to quick sort algorithm. The difference between these algorithms is, instead of recurring both sides after choosing the pivot element, it recurs only one part that contains the k-th smallest element. If the index is the same as k, then the kth value has been found.

In this experiment, there are two different quick select algorithms. The difference between both of them is selecting a specific pivot element. The first quick select algorithm uses array partitioning. Also while partitioning, the algorithm chooses the pivot element as the first element in the array. The other quick select algorithm uses median-of-three pivot selection process. In this case, the algorithm compares the first, last and middle element of the list, and selects the median value of three of them as pivot.

We can summarize the working principle of Quick Select Algorithm with two different examples like down below. First example selecting pivot as first element.



Second example is selecting pivot using the median-of-three method.



Let's choose k = 4, Pivot Element = Median of Three (In this case selected 11)

Compare and replace aspect to pivot number

Move the pivot number, **Pivot index < k (2<4)**, Continue with right part

New Pivot (New Median) = 35

Compare and replace aspect to pivot number

Move the pivot number, **Pivot index > k (6>4)**, Continue with left part

New Pivot (New Median) = 16

Compare and replace aspect to pivot number

Move the pivot number, **Pivot index = k (4=4)**, K'th smallest element founded!

DESIGNING THE EXPERIMENT

To measure each of the algorithms, we studied on each case very sensitive and decided the best suitable inputs for our experiment in the following:

We formed 10 different inputs for each case (Best, Average, Worst). Varying the size between 1000 and 10000. In these algorithms, we decide that the reasonable metric for all of these algorithms is comparison. Because in the experiment, each algorithm is a kind of comparison based algorithm. Also the majority of basic operation is comparison in all of the algorithms. Thus, to measure the performance of these algorithms, we selected the metric as comparison.

1- Insertion sort: The reasonable metric for this algorithm is comparison which is carried out two times in every iteration.

Best case inputs are the ones which are already in the ascending sorted order. Since we are dealing with a sorted list, the algorithm will not carry out any insertion operations.

Worst case inputs are the ones which are already in descending sorted order. Thus, the algorithm will carry out an insertion operation each time it compares the number with the previous one.

2- Merge sort: The reasonable metric for this algorithm is comparison which is carried out in every iteration.

Best case: The idea is to consider the case when the array is already sorted. Before merging, just check if arr[mid] > arr[mid+1], because we are dealing with sorted subarrays. This will lead us to the recursive relation T(n) = 2*T(n/2) + 1 which can be resolved by the master's theorem, so T(n) = n.

Worst case: To generate the worst case of merge sort, the merge operation that resulted in the above sorted array should result in maximum comparisons. To do so, the left and right sub-array involved in merge operation should store alternate elements of sorted array i.e., left sub-array should be {1,3,5,7} and right sub-array should be {2,4,6,8}. Now every element of the array will be compared at-least once and that will result in maximum comparisons. We apply the same logic for the left and right sub-array as well. For array {1,3,5,7}, the worst case will be when it's left and right sub-array are {1,5} and {3,7} respectively and for array {2,4,6,8} the worst case will occur for {2,4} and {6,8}.

3- Quick sort:

Worst case: This scenario of quick sort occurs when the partition process always picks the largest or smallest element as a pivot. In this scenario, the partition process would be highly unbalanced i.e. one subproblem with n - 1 element and the other with 0 elements. This situation occurs when the array will be sorted in increasing or decreasing order.

Best case: The best-case behavior for quicksort occurs when the partition process always picks the middle element as a pivot. In other words, this is a case of the balanced partition where both sub-problems are n/2 size each.

4- Partial heap-sort:

Worst case: The worst case for heap sort might happen when all elements in the list are distinct. Therefore, we will need to call max-heapify every time we want to remove an element. Thus,the maximum number of comparisons will be made.

Best case: The best case for heapsort would happen when all elements in the list are identical. Removing each node from the heap would take only a constant runtime and there is no need to change the element indexes in the list because of their equality.

5- Partial selection-sort

Worst case: This scenario occurs when we have an array in descending order and we want to sort the list in ascending order. Because when the user wants to reach the kth element, and the array is in reversed order, there will be the maximum number of comparisons and moving operations.

Best case: If the array is already sorted, in that case there will be the minimum number of comparisons and provides the best case for the algorithm.

6- Quick select:

- **Pivot as the first element:**

Worst case: For example, this occurs in searching for the maximum element of a set, using the first element as the pivot, and having sorted data. In this case every time the algorithm will try to find the k’th (last) element of the list and it will only eliminate the first element (pivot) and do the maximum comparisons.

Best case: k'th element is the pivot. So the algorithm directly finds the k’th element without any comparison.

- **Median of three:**

Best case: In this case sorted list is given as an input. The first selected pivot is the middle element of the list which is also selected as the k'th value by the user. Thus, the algorithm finds the kth element in the list without any comparison.

Worst case: In this case a sorted list is given as an input. K'th value is selected as the first or the last element of the list.The algorithm always selects the median of the list and the comparison will be made until reaching the last element of the first element. Therefore, the maximum number of comparisons are made.

COMPLEXITY ANALYZING RESULTS:

In our experiment, We compared seven methods that sorts the given input lists and find k'th smallest element in the lists.

We tested every case of each algorithm exactly ten times with five different input sizes and calculated the average execution time below the table. While doing our empirical analysis, we used a physical unit of time (in seconds). Also we provided every algorithm's best, average and worst case graphics in the following. At the end of these graphics we gathered every algorithm and compared their performances in one graphic with the different input sizes of 1000, 5000 and 9000.

While conducting our analysis we used the same computer in order to get reasonable results as stated in the experiment document. Also while running Quick Sort algorithm, we realized that a big amount of input sizes crashes the RAM. When the algorithm runs with 9000 inputs, we get a "Stack overflow" error on our console screen. Because of that we couldn't run the program with 9000 inputs with the Quick Sort algorithm.

You can examine the results in the unit of seconds down below the table. Also you can see these values as a graph at the end of this topic.

	Insertion Sort		
Input Size (n)	Best	Average	Worst
1000	0.00038682	0.0386892	0.07962838
3000	0.000613219	0.37572418	0.69961316
5000	0.0011221	1.08917417	1.08917417
7000	0.00157203	2.15513528	3.92483089
9000	0.0019328	3.395082	6.58733992

	Merge Sort		
Input Size (n)	Best	Average	Worst
1000	0.02302488	0.0206244	0.00424918
3000	0.03654891	0.04899396	0.06561218
5000	0.10373224	0.10889162	0.13564845
7000	0.17802194	0.18058282	0.19345211
9000	0.19492204	0.1753719	0.17654107

	Quick Sort		
Input Size (n)	Best	Average	Worst
1000	0.00939486	0.01059479	0.20296705
3000	0.02976722	0.02886511	0.236766355
5000	0.05325175	0.06752654	0.36265587
7000	0.05951727	0.13924329	0.416609196
9000	-	-	-

	Partial Selection Sort		
Input Size (n)	Best	Average	Worst
1000	6.388E-05	8.82E-05	0.00016528
3000	0.00017617	0.0002107	0.00055314
5000	0.0002705	0.00027531	0.00080916
7000	0.00043146	0.00037819	0.00102832
9000	0.00048937	0.00061027	0.00152186

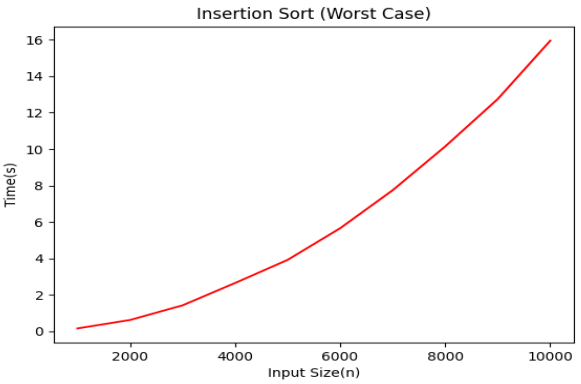
	Quick Select First element as pivot		
Input Size (n)	Best	Average	Worst
1000	0.00207513	0.00289219	0.00295641
3000	0.00809559	0.00584861	0.00596241
5000	0.01036709	0.01268813	0.01295562
7000	0.013562	0.01057146	0.01425693
9000	0.0192732	0.03224482	0.03058423

	Quick Select Median of three as pivot		
Input Size (n)	Best	Average	Worst
1000	0.00041694	0.00013164	0.00037855
3000	0.00122889	0.00056696	0.00131227
5000	0.00231481	0.00226685	0.00169068
7000	0.00279902	0.00214501	0.00274104
9000	0.00456978	0.00245111	0.00317179

	Partial Heap Sort		
Input Size (n)	Best	Average	Worst
1000	0.00271727	0.02651234	0.0217855
3000	0.01342455	0.0819323	0.08198756
5000	0.01985022	0.13388389	0.15022041
7000	0.02384842	0.20076727	0.20282873
9000	0.02825882	0.26788387	0.27486032

1. Insertion Sort Algorithm Analysis:

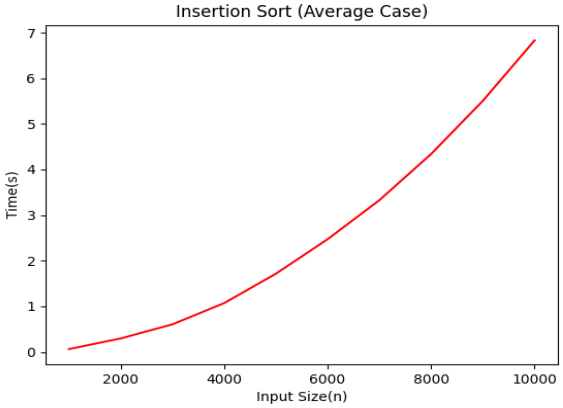
While using insertion sort algorithm, as mentioned before, ascending ordered sorted list is used for the best case input, descending ordered sorted list is used for the worst case input. You can see the results in the unit of seconds, distributed over various sizes.



Also, we created different sizes of random numbers with specific function in the python library similar to the one below. Also we used this function for the average inputs of each algorithm. Detailed analysis graphic is also down below.

ALGORITHM *Random*$(n, m, seed, a, b)$

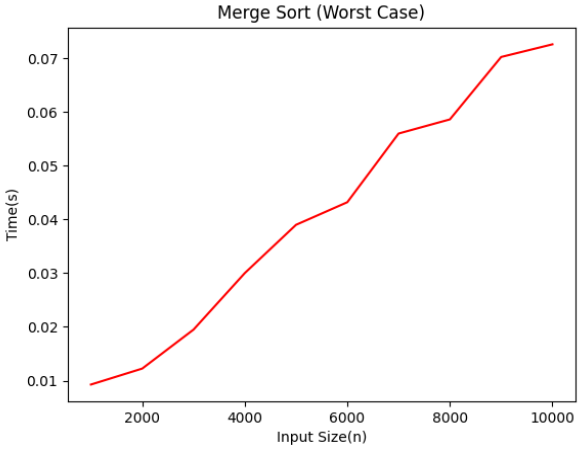
//Generates a sequence of n pseudorandom numbers according to the linear
// congruential method
//Input: A positive integer n and positive integer parameters m, $seed$, a, b
//Output: A sequence $r_1, \ldots, r_n$ of n pseudorandom integers uniformly
// distributed among integer values between 0 and $m - 1$
//Note: Pseudorandom numbers between 0 and 1 can be obtained
// by treating the integers generated as digits after the decimal point
$r_0 \leftarrow seed$
for $i \leftarrow 1$ **to** n **do**
  $r_i \leftarrow (a * r_{i-1} + b) \bmod m$



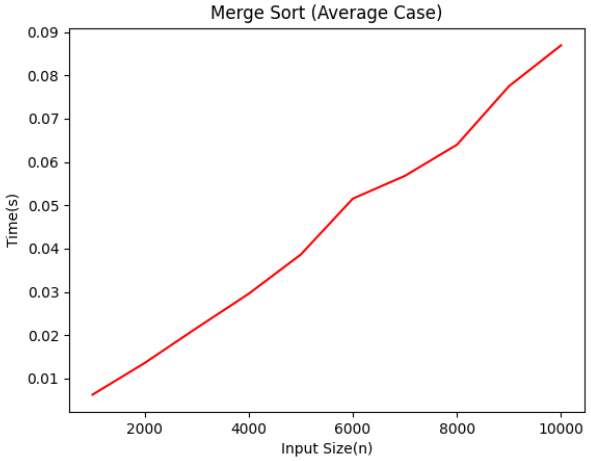
We can see that if the given numbers are sorted, this algorithm runs in ***O(n)*** time. If the given numbers are in reverse order, the algorithm runs in approximately ***O(n^2)*** time. If we consider the theoretical expectations, we can say that this algorithm behaves as expected. So, it behaves as ***linear*** in the best case and behaves as ***quadratic*** in the worst case. Also, insertion sort is a stable sorting algorithm which means the equal elements are ordered in the same order in the sorted list. Also this algorithm has constant space complexity of **O(1)**.

2. Merge-Sort Algorithm Analysis

While using the merge-sort algorithm, ascending ordered sorted list is used for best case input, specified list is used for worst case input. You can see the results in the unit of seconds, distributed over various sizes.



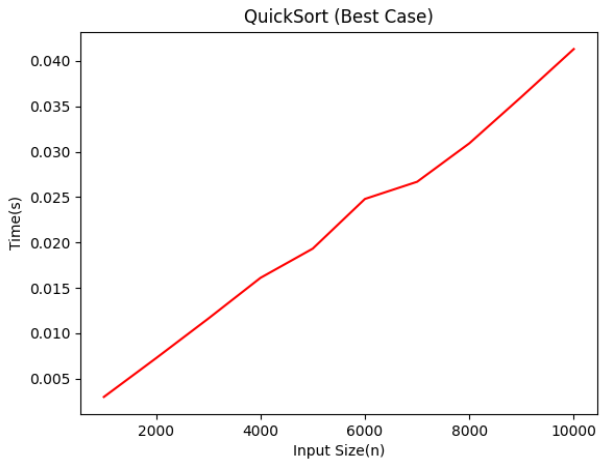
For average cases detailed analysis graphic is down below.

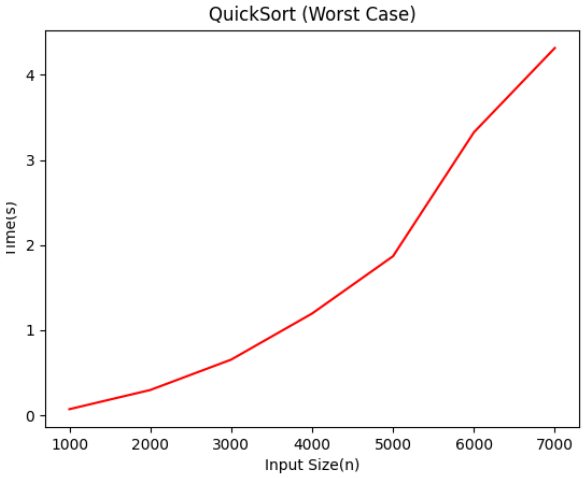


We can see that while running the merge-sort algorithm, the input size doesn't affect the performance of the program. Each step, the program behaves in the same behavior. Also the results are approximately the same in each algorithm running result. We can say that the algorithm behaves as a runtime of **quasilinear** at each run. If we consider the theoretical expectations, we can say that this algorithm behaves as expected **O(nlogn)** complexity. In addition to this, the algorithm uses extra memory, so that the algorithm has constant space complexity of **O(n)**.

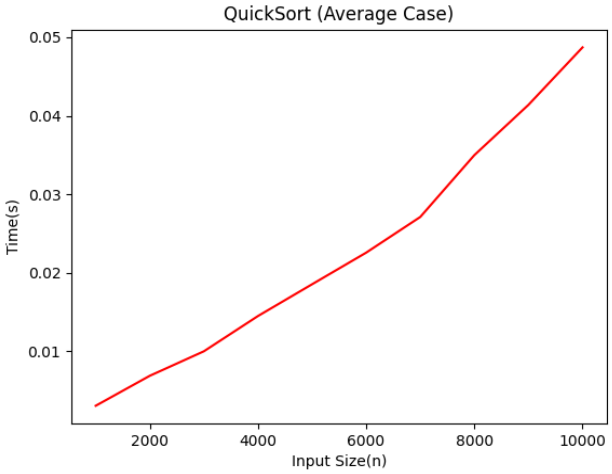
3. Quick-Sort Algorithm Analysis

While using a quick-sort algorithm, as mentioned before, the middle element selected as a pivot in best case input, the largest or smallest element selected as a pivot in worst case input. You can see the results in the unit of seconds, distributed over various sizes.





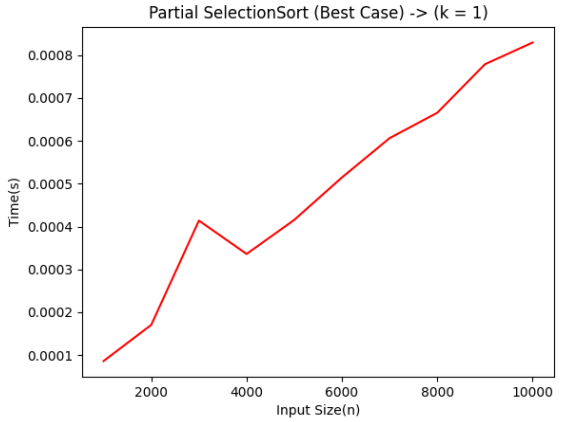
For average cases detailed analysis graphic is down below.

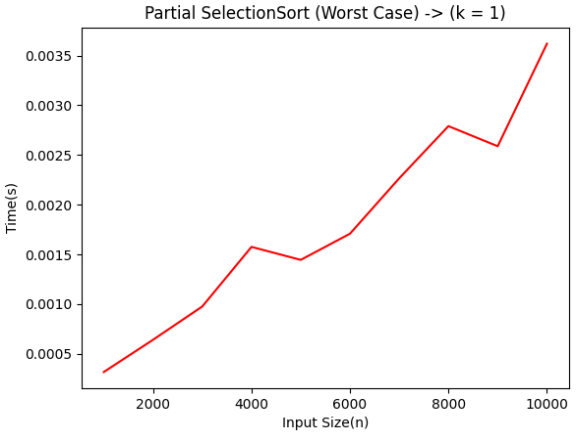


We can see that while running the quick-sort algorithm if the selected pivot is the middle element of the list, this algorithm runs in ***O(n)*** time. If the selected pivot is the largest or smallest element in the list, the algorithm runs in approximately ***O(n^2)*** time. So, it behaves as ***linear*** in the best case and behaves almost as ***quadratic*** in the worst case. Also this algorithm is not stable.

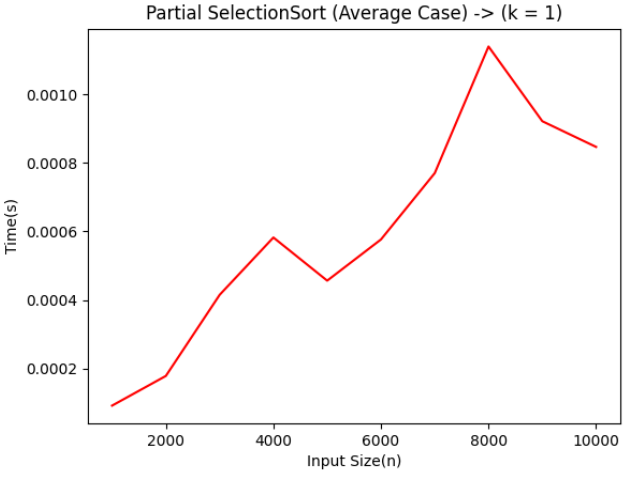
4. Partial Selection-Sort Algorithm Analysis

While using partial selection sort algorithm, as mentioned before, ascending ordered sorted list is used for best case input, descending order sorted list is used for worst case input. You can see the results in the unit of seconds, distributed over various sizes.





For average cases detailed analysis graphic is down below.

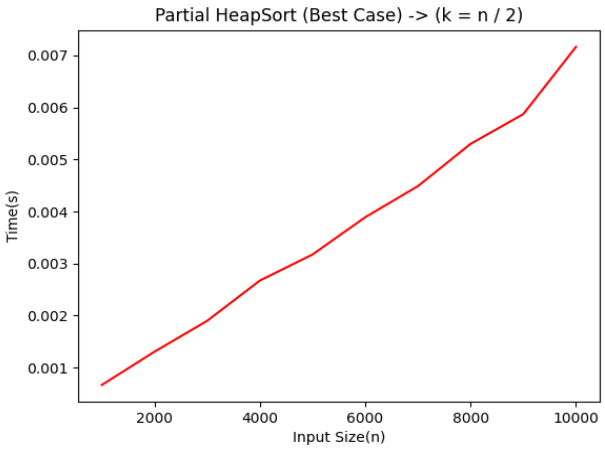


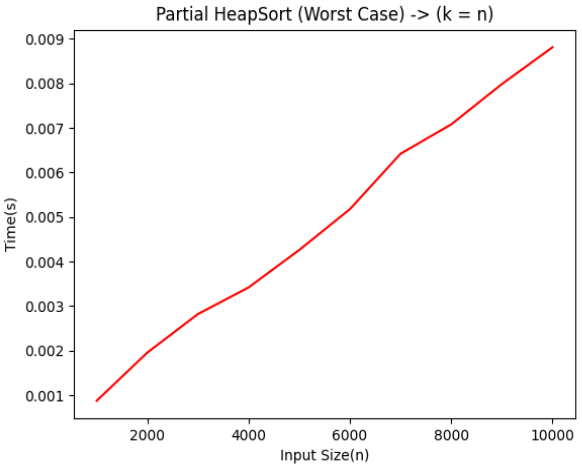
While running the partial selection sort algorithm, we can see that the graphics are not stable enough. Even though we got the correct k'th value, the timing

is not stable between the various input sizes. Therefore if we consider the theoretical expectations, the algorithm doesn't behave as expected about timing.

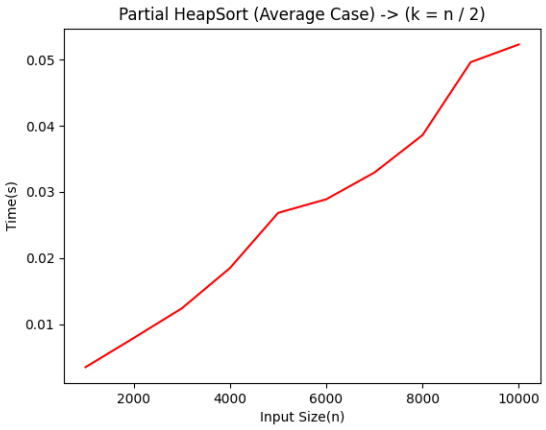
5. Partial Heap Sort Algorithm Analysis

While using Partial Heap Sort algorithm, the best case for heapsort would happen when the input list is sorted. On the other hand, when we use a list where all elements are distinct, the worst case scenario occurs. You can see the results in the unit of seconds, distributed over various sizes.





For average cases detailed analysis graphic is down below.

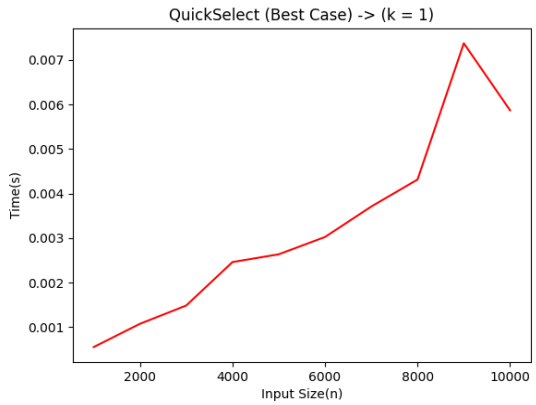


While running the Heap Sort algorithm, as we can see in the plots, when doubling the input quantity, sorting takes a little more than twice as long; this corresponds to the expected ***quasilinear runtime O(n log n)***. If we consider the theoretical expectations, we can say that the algorithm behaves as expected.

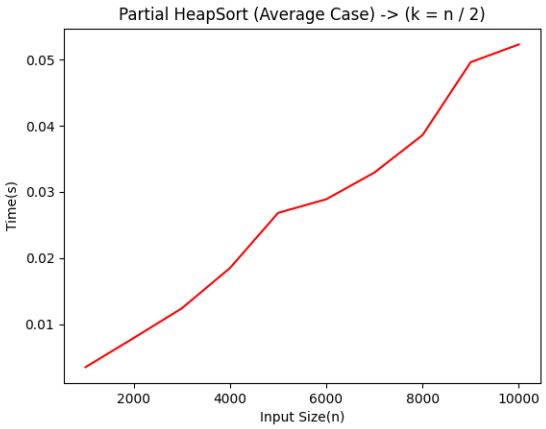
Heapsort can be performed **in place**. The array can be split into two parts, the sorted array and the heap. The heap's invariant is preserved after each extraction, so the only cost is that of extraction.

6. Quick-Select (First Element as Pivot) Algorithm Analysis

While using the quick-select algorithm, sorted list is used as an input for both best and worst case scenarios. You can see the results in the unit of seconds, distributed over various sizes.



For average cases detailed analysis graphic is down below.

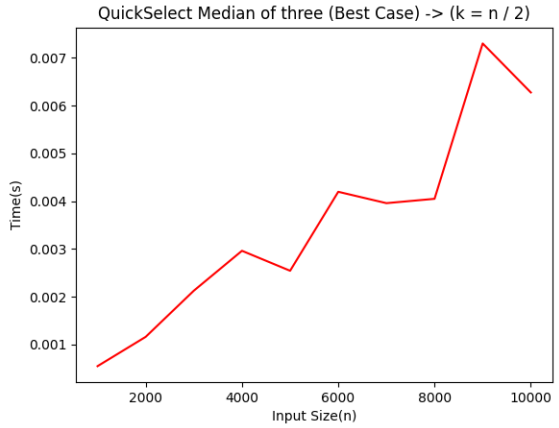


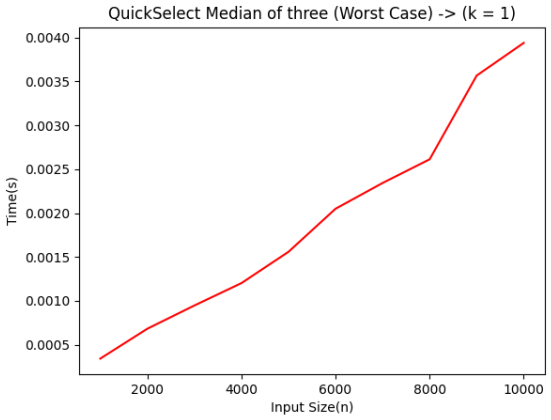
While running the quick select algorithm which uses the first element as pivot, we can see that the best and the worst case graphics are not stable enough.The reason for this issue is that we are dealing with milliseconds. Even though we got the correct k'th value, the timing is not stable between the various input sizes. Therefore

if we consider the theoretical expectations, the algorithm doesn't behave as expected about timing.

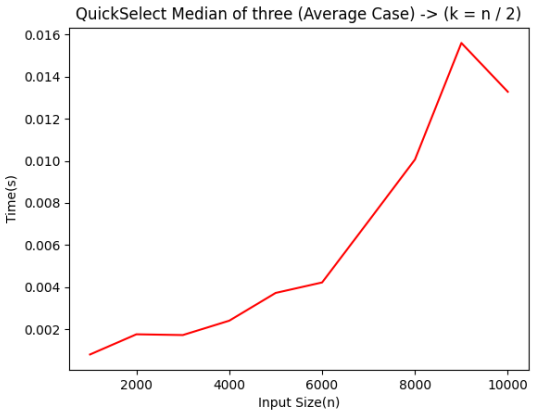
7. Quick-Select (Median-of-Three) Algorithm Analysis

While using the quick-select algorithm, sorted list is used as an input for both best and worst case scenarios. You can see the results in the unit of seconds, distributed over various sizes.





For the average cases detailed analysis graphic is down below.



The worst case for Quicksort using the median-of-3 method is when the selected pivot reduces the problem size by the smallest possible amount. This means that the selected pivot provides as far-off a partition as is possible. 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(nlogn).**

DETAILED ANALYSIS

In the final part of our experiment with input size of 1000, 5000, 9000 we executed every algorithm and plotted their various performances in one graph.

	Insertion Sort		
Input Size (n)	Best	Average	Worst
1000	0.00038682	0.0386892	0.07962838
3000	0.000613219	0.37572418	0.69961316
5000	0.0011221	1.08917417	1.08917417
7000	0.00157203	2.15513528	3.92483089
9000	0.0019328	3.395082	6.58733992

	Merge Sort		
Input Size (n)	Best	Average	Worst
1000	0.02302488	0.0206244	0.00424918
3000	0.03654891	0.04899396	0.06561218
5000	0.10373224	0.10889162	0.13564845
7000	0.17802194	0.18058282	0.19345211
9000	0.19492204	0.1753719	0.17654107

	Quick Sort		
Input Size (n)	Best	Average	Worst
1000	0.00939486	0.01059479	0.20296705
3000	0.02976722	0.02886511	0.236766355
5000	0.05325175	0.06752654	0.36265587
7000	0.05951727	0.13924329	0.416609196
9000	-	-	-

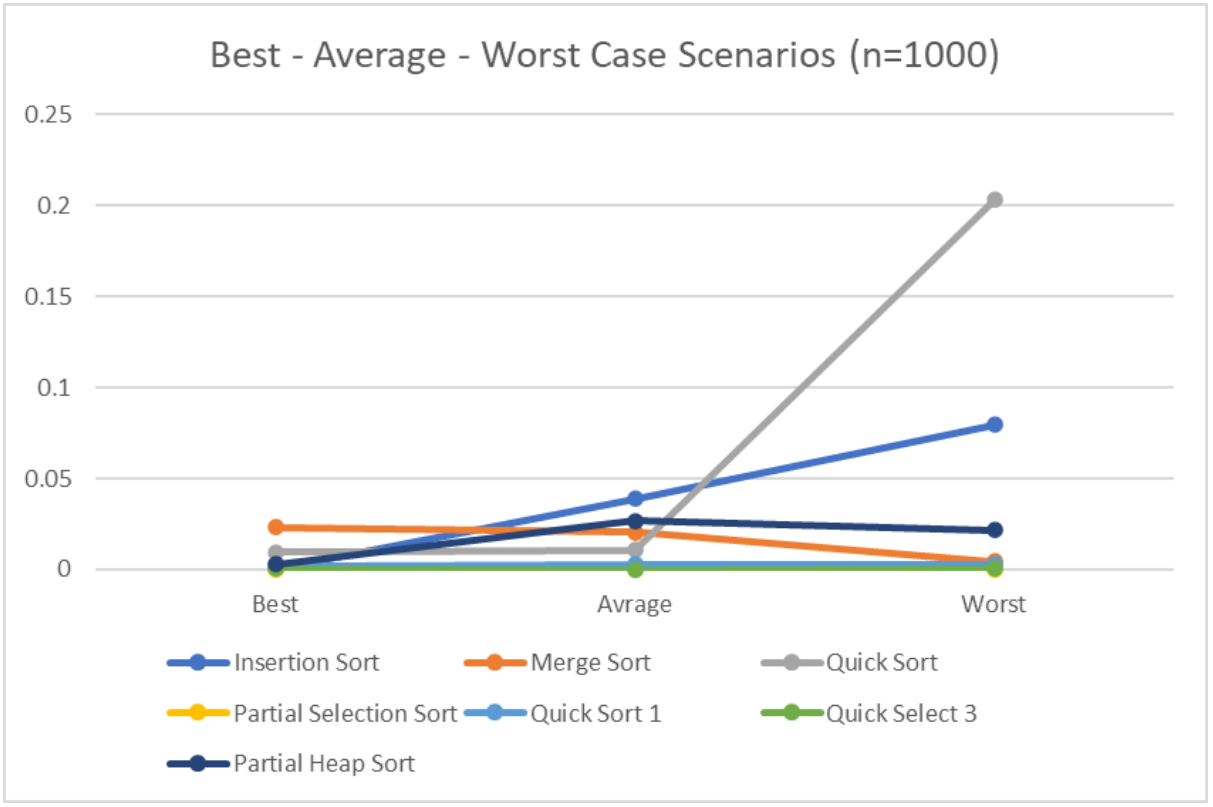
	Partial Selection Sort		
Input Size (n)	Best	Average	Worst
1000	6.388E-05	8.82E-05	0.00016528
3000	0.00017617	0.0002107	0.00055314
5000	0.0002705	0.00027531	0.00080916
7000	0.00043146	0.00037819	0.00102832
9000	0.00048937	0.00061027	0.00152186

	Quick Select First element as pivot		
Input Size (n)	Best	Average	Worst
1000	0.00207513	0.00289219	0.00295641
3000	0.00809559	0.00584861	0.00596241
5000	0.01036709	0.01268813	0.01295562
7000	0.013562	0.01057146	0.01425693
9000	0.0192732	0.03224482	0.03058423

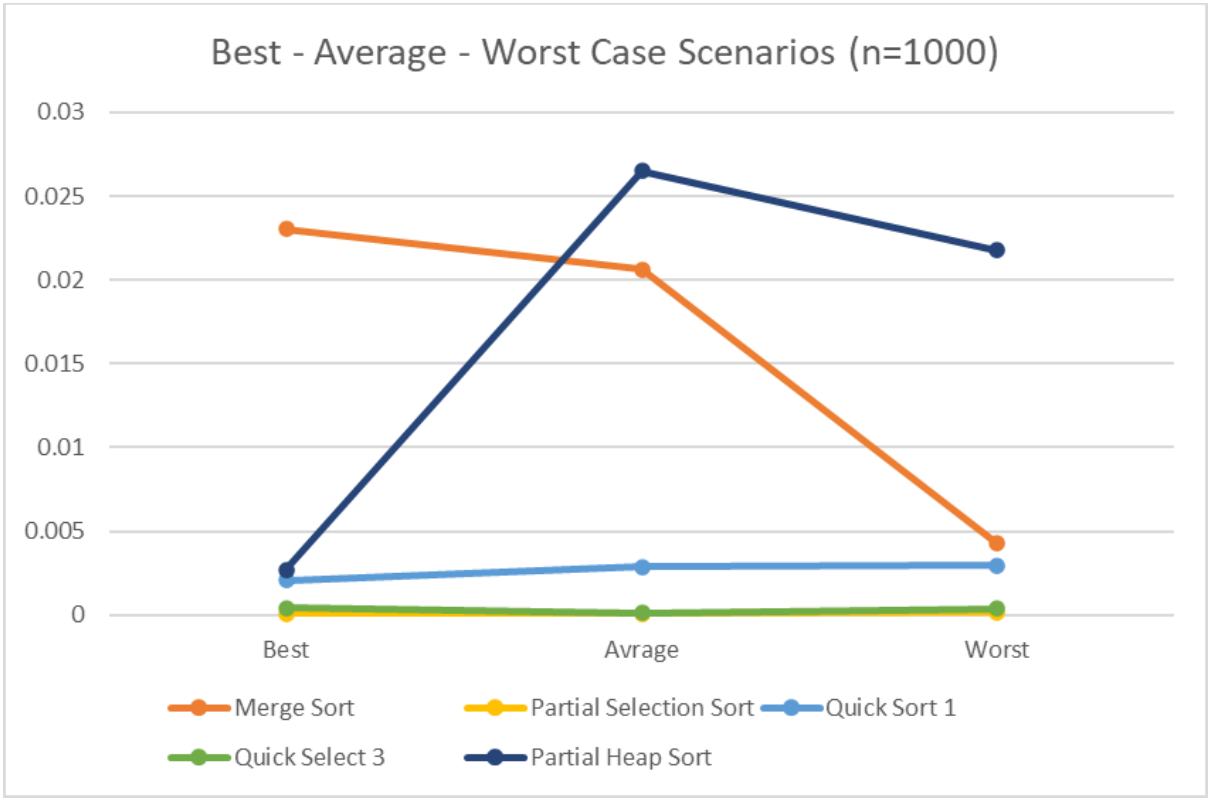
	Quick Select Median of three as pivot		
Input Size (n)	Best	Average	Worst
1000	0.00041694	0.00013164	0.00037855
3000	0.00122889	0.00056696	0.00131227
5000	0.00231481	0.00226685	0.00169068
7000	0.00279902	0.00214501	0.00274104
9000	0.00456978	0.00245111	0.00317179

	Partial Heap Sort		
Input Size (n)	Best	Average	Worst
1000	0.00271727	0.02651234	0.0217855
3000	0.01342455	0.0819323	0.08198756
5000	0.01985022	0.13388389	0.15022041
7000	0.02384842	0.20076727	0.20282873
9000	0.02825882	0.26788387	0.27486032

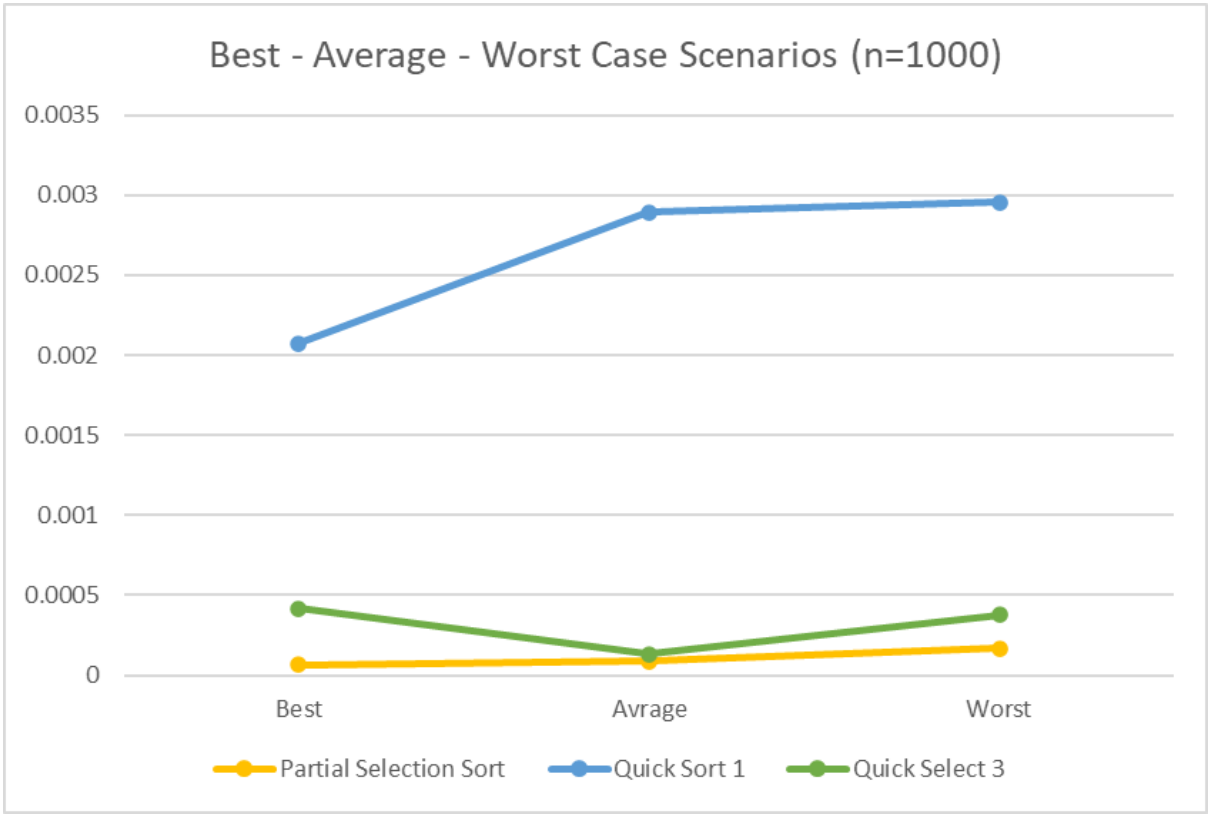
INPUT SIZE OF 1000:



Since the complexity of the insertion sort is O(n^2) it grows faster than the other algorithms which are in O(nlogn). We provided the graph below without insertion sort's performance line to show other sorting algorithms more clearly(in seconds).

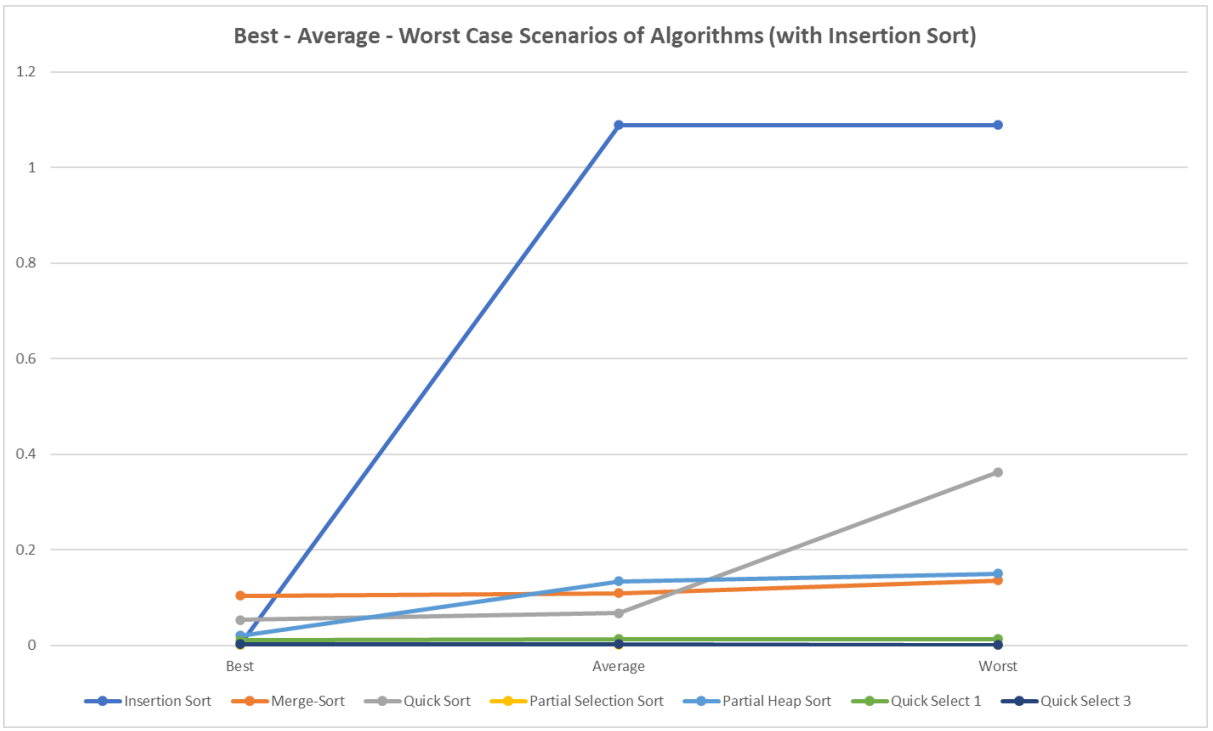


This graph is for algorithms except insertion sort to show their performances in seconds more clearly.

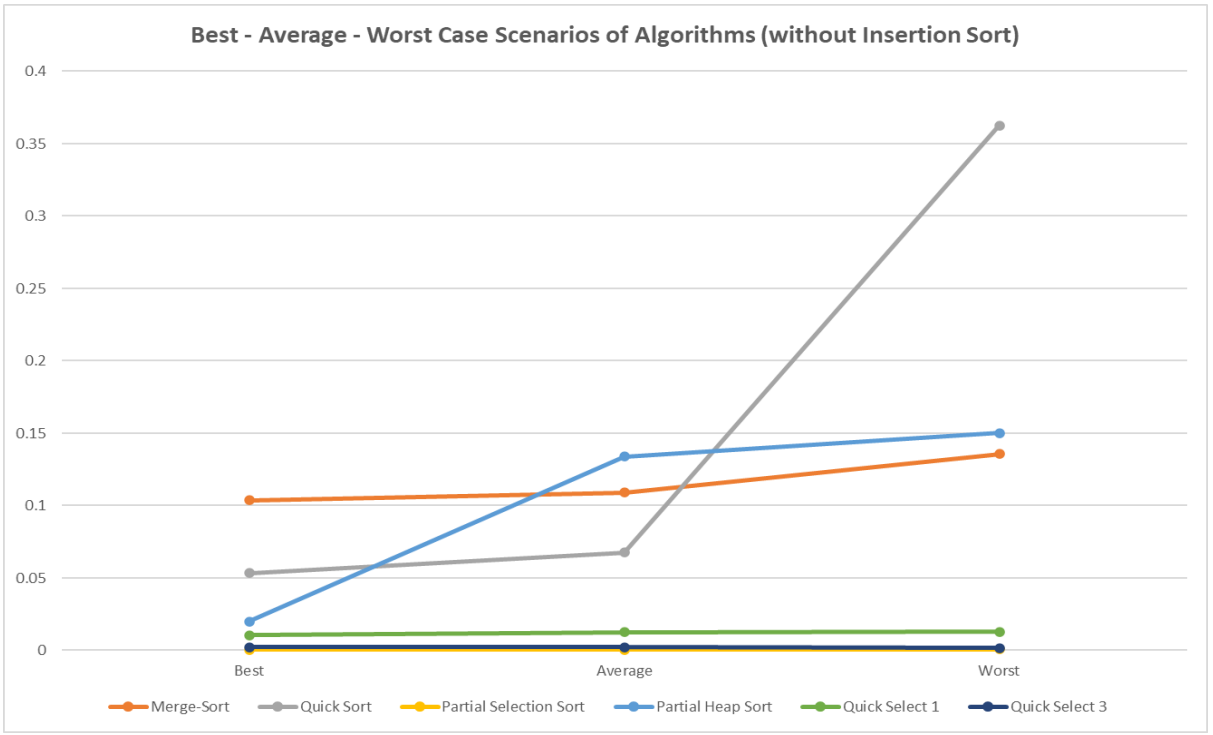


This graph is for algorithms other than insertion sort, merge sort and quick sort to show their performances in seconds more clearly since their execution times are very small compared to the other ones.

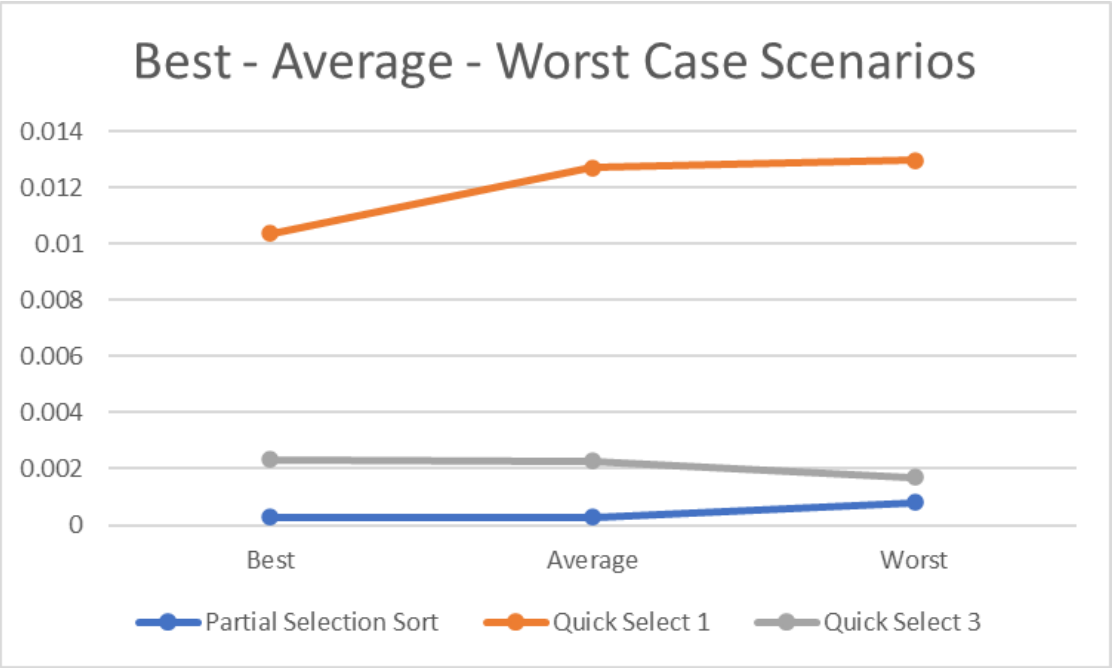
INPUT SIZE OF 5000:



Since the complexity of the insertion sort is O(n^2) it grows faster than the other algorithms which are in O(nlogn). We provided the graph below without insertion sort's performance line to show other sorting algorithms more clearly.(in seconds)

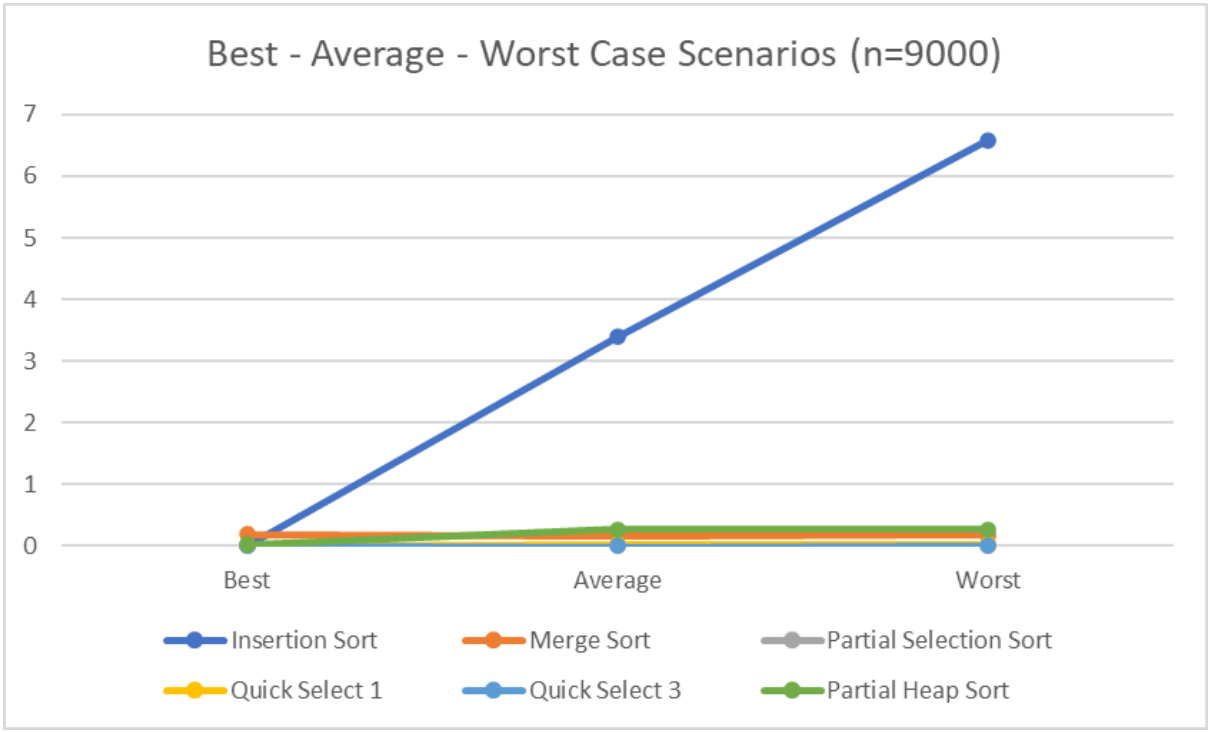


This graph is for algorithms except insertion sort to show their performances in seconds more clearly.

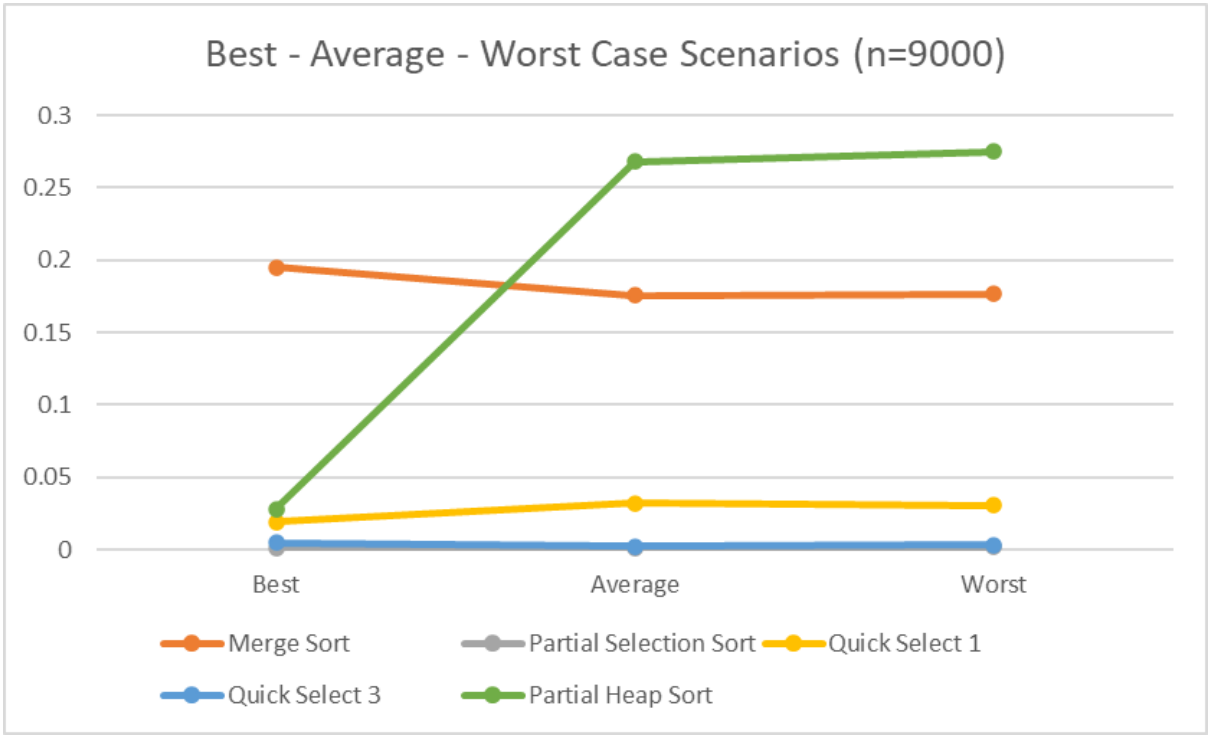


This graph is for algorithms other than insertion sort, merge sort and quick sort to show their performances in seconds more clearly since their execution times are very small compared to the other ones.

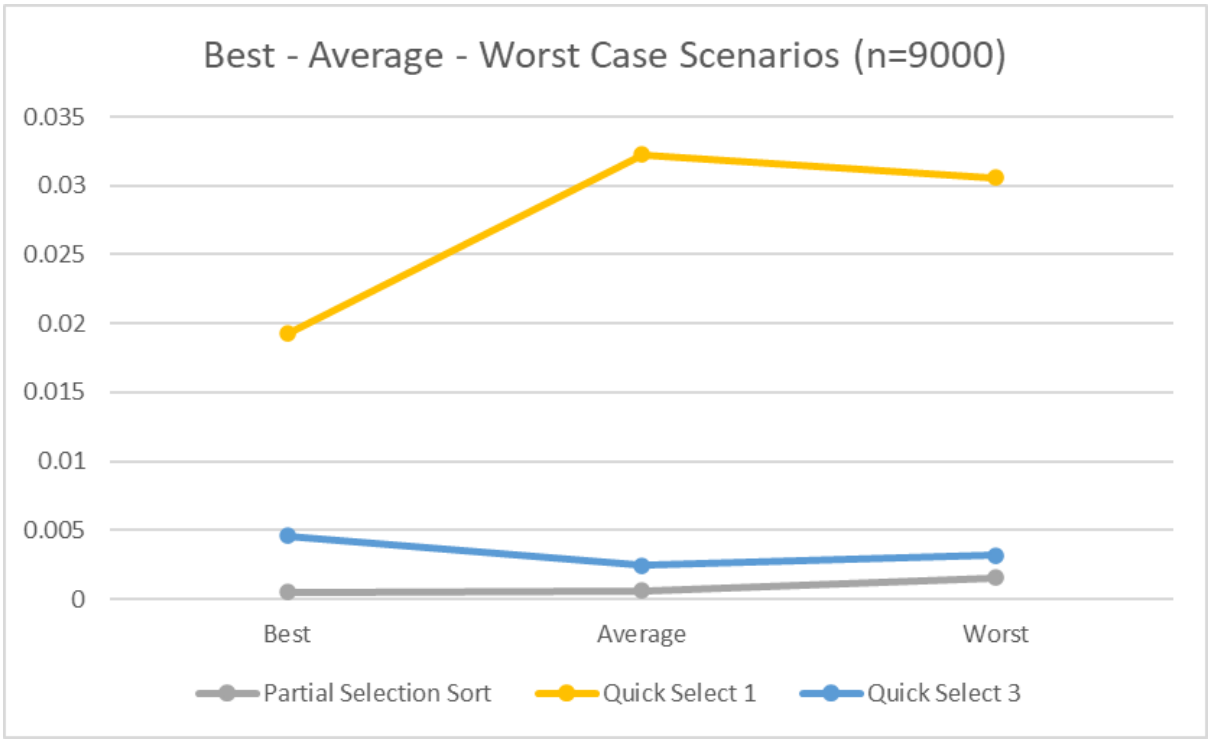
INPUT SIZE OF 9000:



Since the complexity of the insertion sort is O(n^2) it grows faster than the other algorithms which are in O(nlogn). As input size gets bigger we can examine that the performance of the insertion sort decreases dramatically. We provided the graph below without insertion sort's performance line to show other sorting algorithms more clearly(in seconds).

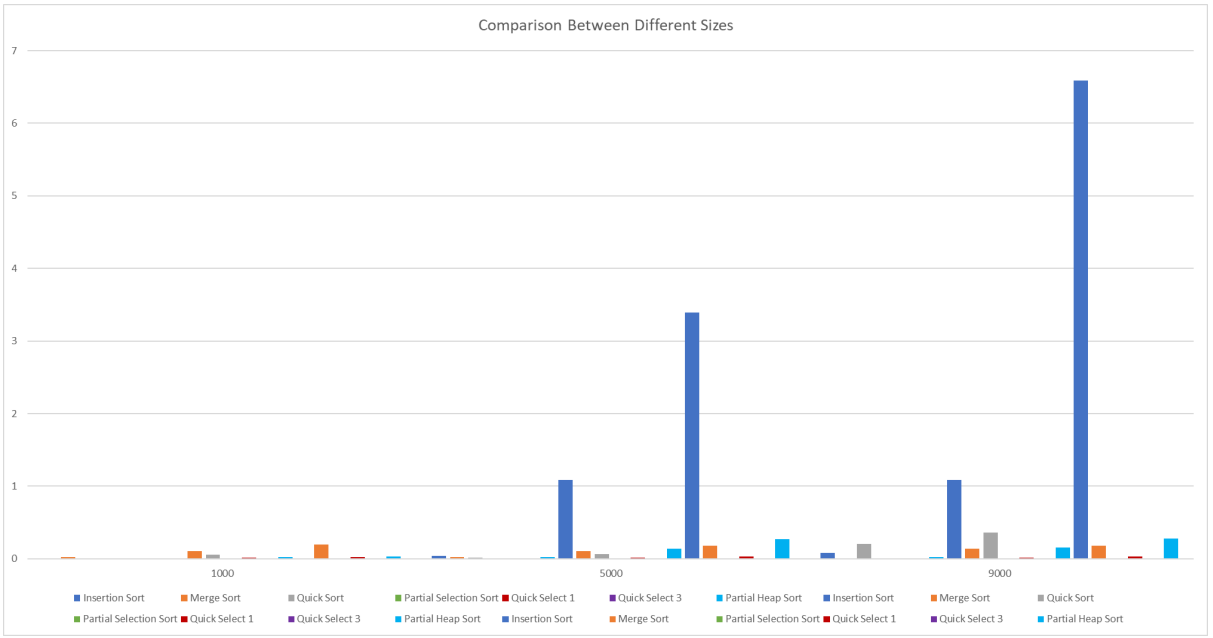


This graph is for algorithms except insertion sort to show their performances in seconds more clearly.

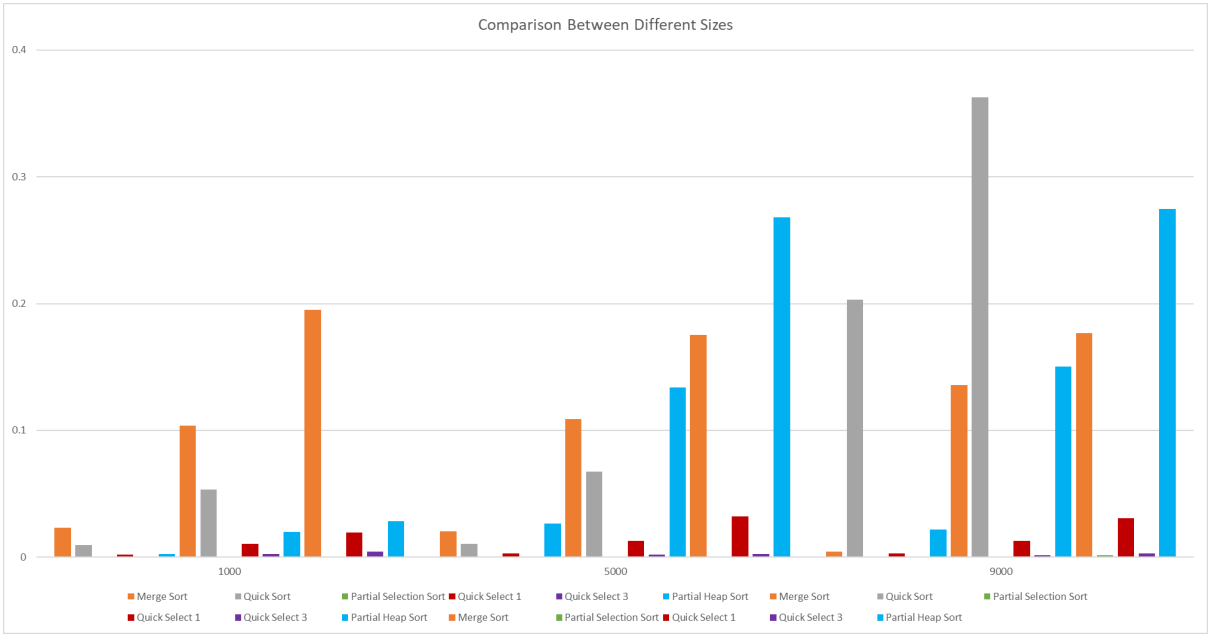


This graph is for algorithms other than insertion sort, merge sort and quick sort to show their performances in seconds more clearly since their execution times are very small compared to the other ones.

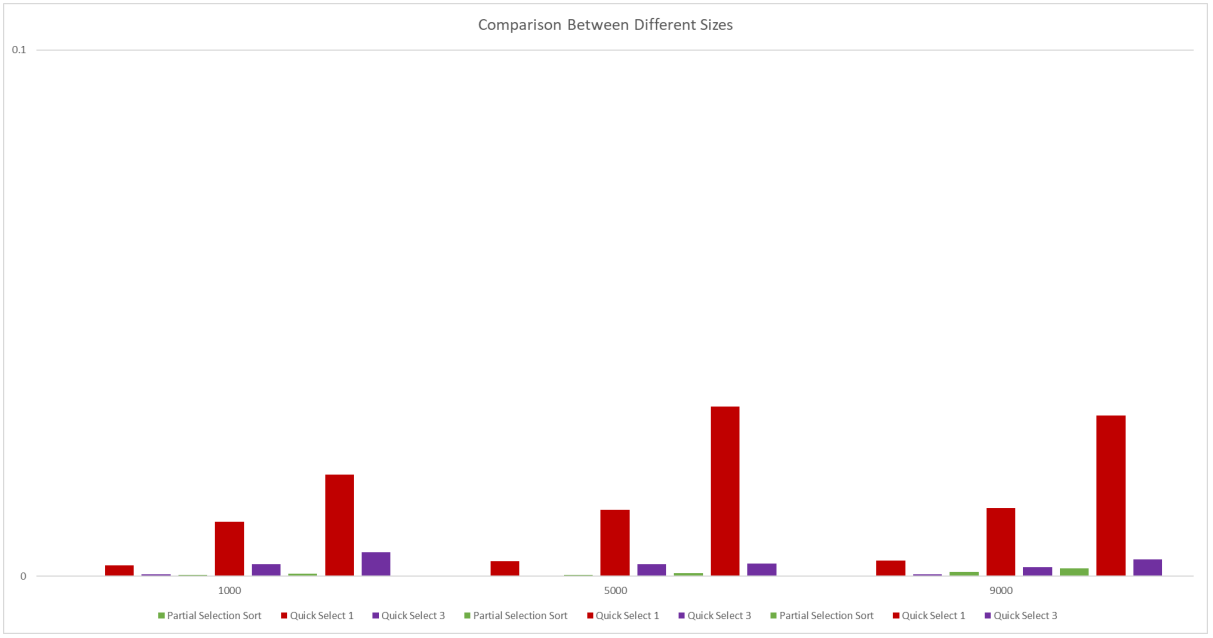
GRAPHS FOR ALL ALGORITHMS WITH ALL INPUTS AND INPUT SIZES:



Comparison of all seven algorithms.



Comparison without insertion sort algorithm.



Comparison without insertion ,merge and quick sort algorithms.

CONCLUSION:

As a result, In this experiment, we compared 7 different algorithms with various input sizes and found k'th element values. We conclude that :

- All the seven sorting algorithms (Insertion Sort, Merge sort, Quick sort, Partial Heap Sort and Partial Selection sort and Quick Selects) were implemented in Python programming language and tested for the specific inputs of length 1000, 5000, 9000. All these sorting algorithms were executed on a machine Operating System having Intel(R) Core(TM) i5-3230M CPU @2.60 Ghz and installed memory (RAM) 8096 MB.

- Result shows that no matter the input, the performance for the best case of all seven algorithms are close to each other, but for larger inputs, insertion sort is giving a bad performance for the worst case.

- On the other hand, Partial Selection Sort is the fastest for the worst case scenario.

- For the Quick Sort, we conclude that it may be a good algorithm in terms of time complexity. However, it provides poor performance for space complexity, because when we are running the quick-sort algorithm with 10000 input size, we get a memory error (stack overflow) caused by the number of recursions with Python.

Therefore, If we have an input of more than 10000, the program will not be able to sort the array and the program will give an error (stack overflow) to us.

- There are some plots that are not as stable as expected. The reason for this issue is that we are dealing with milliseconds and some system related delays may happen during the execution of the algorithms. While executing the same algorithm with the same input, we sometimes get different timing results in every iteration

Who did what ? :

- **Erdem Pehlivanlar :**
  - Insertion Sort Algorithm
  - Partial Selection Sort Algorithm
  - Report
  - Plotting graphics
  - Preparing Input files

- **Haldun Halil Olcay :**
  - Quick Sort Algorithm
  - Quick Select Algorithm Pivot as First Element
  - Quick Select Algorithm Pivot as Median of Three
  - Report

- **Yasin Alper Bingül :**
  - Merge Sort Algorithm
  - Partial Heap Sort Algorithm
  - Report
  - Plotting graphics

REFERENCES:

- **https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.736.3357&rep=rep1&type=pdf**

  Review on Sorting Algorithms A Comparative Study (Khalid Suleiman Al-Kharabsheh, Ibrahim Mahmoud AlTurani, Abdallah Mahmoud Ibrahim AlTurani & Nabeel Imhammed Zanoon)

- https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort

- https://en.wikipedia.org/wiki/Partial_sorting

- https://www.geeksforgeeks.org/insertion-sort/

- https://stackoverflow.com/questions/24407555/partial-sorting-to-find-the-kth-largest-smallest-elements

- https://www.happycoders.eu/algorithms/heapsort/

- https://www.programiz.com/dsa/merge-sort

- https://stackoverflow.com/questions/50912873/python-quicksort-with-median-of-three

- For drawing graph : https://pythonguides.com/matplotlib-multiple-plots/