CSE2246 – ANALYSIS OF ALGORITHMS

HOMEWORK #1 – REPORT

Ayşe Sena AYDEMİR – 150119735

Emre Cihan VARLI – 150119711

Fatih Erkam DİNLER – 150119567

Abstract

This report presents an analysis of seven different algorithms designed to find the median value in an unsorted array of numbers. The algorithms evaluated include Insertion Sort, Merge Sort, Quick Sort, Max-Heap removal and three variations of the Quick Select algorithm

Designing the Experiment

Sample Inputs

We Created inputs in three different types: Random Order, Reversed Order and Sorted Order, with sizes ranging from 10 to 100.000.

Reasoning for Choices

Random Order: Arrays with elements in a random order represent the average case for most sorting algorithms. This characteristic is vital as it simulates real-world data, which is often not in any particular order. Testing algorithms with random order inputs helps in evaluating their performance under typical conditions.

Reversed Order: Reversed order arrays present a worst-case scenario for certain sorting algorithms, especially those that are comparison-based, like Insertion Sort. By including reversed order arrays, the analysis can stress-test the algorithms to understand their performance boundaries.

Sorted Order: Already sorted arrays can act as the best-case input for some algorithms and may present different challenges for others. For example, Quick Sort with the first element as the pivot can perform poorly on sorted data. Including sorted arrays helps in assessing the algorithms' performance when they encounter an ideal or over-optimized situation.

Complexity Measurement

Execution time was chosen as the primary metric for complexity measurement due to execution time provides a direct measure of performance that reflects real-world conditions. It is an objective metric that can be easily understood and compared across different algorithms and systems.

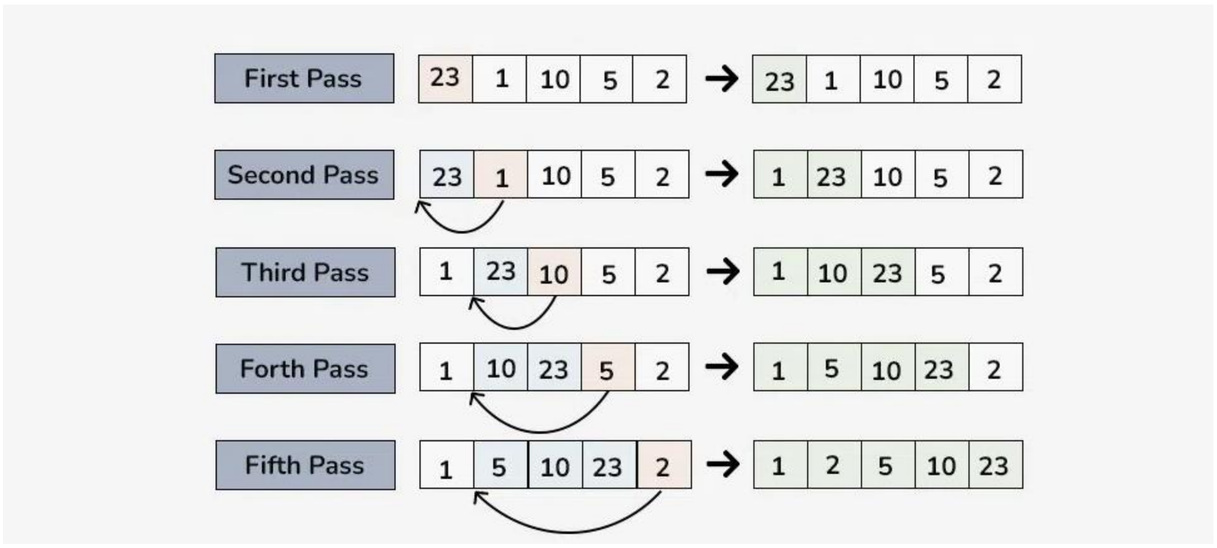
Insertion Sort and return the median

Insertion Sort: Insertion sort is a simple sorting algorithm that works by building a sorted array one element at a time. It is considered an “in-place” sorting algorithm, meaning it doesn’t require any additional memory space beyond the original array.

Algorithm:

To achieve insertion sort, follow these steps:

- We have to start with second element of the array as first element in the array is assumed to be sorted.
- Compare second element with the first element and check if the second element is smaller then swap them.
- Move to the third element and compare it with the second element, then the first element and swap as necessary to put it in the correct position among the first three elements.
- Continue this process, comparing each element with the ones before it and swapping as needed to place it in the correct position among the sorted elements.
- Repeat until the entire array is sorted.



Insertion Sort Example

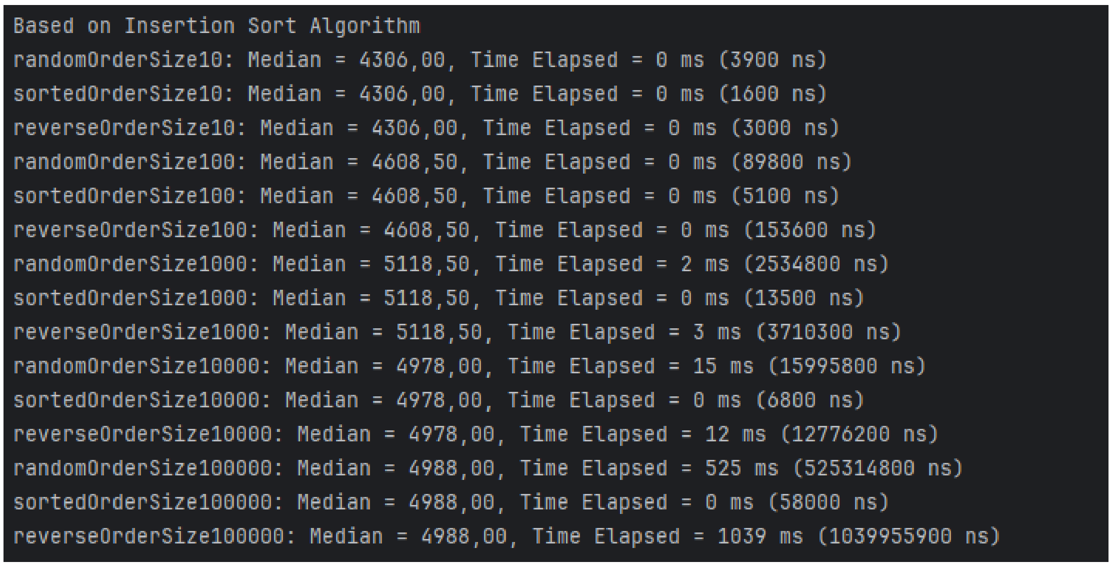
Time complexity of Insertion sort and return the median:

Best Case: If the array is already sorted, the algorithm only needs to check each element once. In this case, the time complexity is O(n).

Average Case: If the array is randomly organized, the algorithm's average performance will be O(n^2).

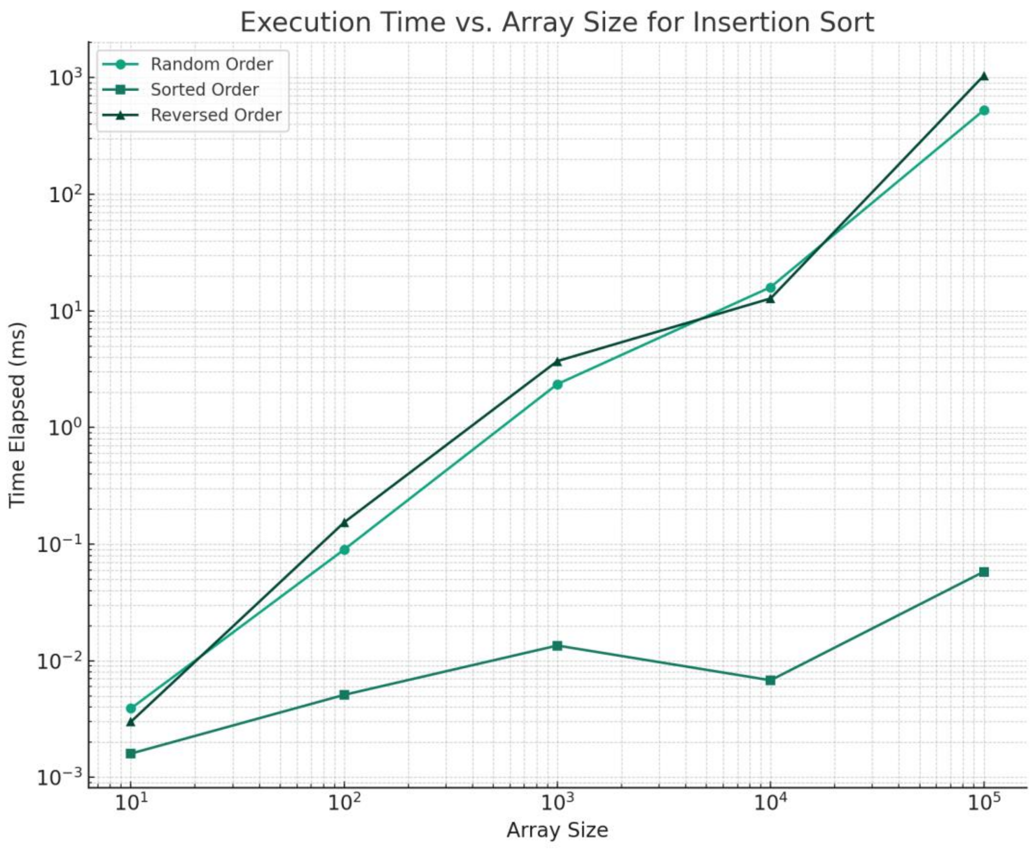
Worst Case: If the array is sorted in the reverse order (descending), every element must be compared with every other element. In this case, the time complexity is again O(n^2).

If we are finding the median after the sorting process is completed, then selecting the middle element from an already sorted array has a time complexity of O(1). Therefore, the overall time complexity for sorting the elements by Insertion Sort and then finding the median is O(n) in the best case and O(n^2) in the average and worst case.



```
Based on Insertion Sort Algorithm
randomOrderSize10: Median = 4306,00, Time Elapsed = 0 ms (3900 ns)
sortedOrderSize10: Median = 4306,00, Time Elapsed = 0 ms (1600 ns)
reverseOrderSize10: Median = 4306,00, Time Elapsed = 0 ms (3000 ns)
randomOrderSize100: Median = 4608,50, Time Elapsed = 0 ms (89800 ns)
sortedOrderSize100: Median = 4608,50, Time Elapsed = 0 ms (5100 ns)
reverseOrderSize100: Median = 4608,50, Time Elapsed = 0 ms (153600 ns)
randomOrderSize1000: Median = 5118,50, Time Elapsed = 2 ms (2534800 ns)
sortedOrderSize1000: Median = 5118,50, Time Elapsed = 0 ms (13500 ns)
reverseOrderSize1000: Median = 5118,50, Time Elapsed = 3 ms (3710300 ns)
randomOrderSize10000: Median = 4978,00, Time Elapsed = 15 ms (15995800 ns)
sortedOrderSize10000: Median = 4978,00, Time Elapsed = 0 ms (6800 ns)
reverseOrderSize10000: Median = 4978,00, Time Elapsed = 12 ms (12776200 ns)
randomOrderSize100000: Median = 4988,00, Time Elapsed = 525 ms (525314800 ns)
sortedOrderSize100000: Median = 4988,00, Time Elapsed = 0 ms (58000 ns)
reverseOrderSize100000: Median = 4988,00, Time Elapsed = 1039 ms (1039955900 ns)
```

Output of our implementation



execution time – array size graph of our findings

Comparison of our findings with theoretical expectations:

Based on the data shown in the output and graph:

Random Order: In randomly sorted arrays, the line generally shows an exponential increase. This is consistent with the expected O(n^2) time complexity for the average case.

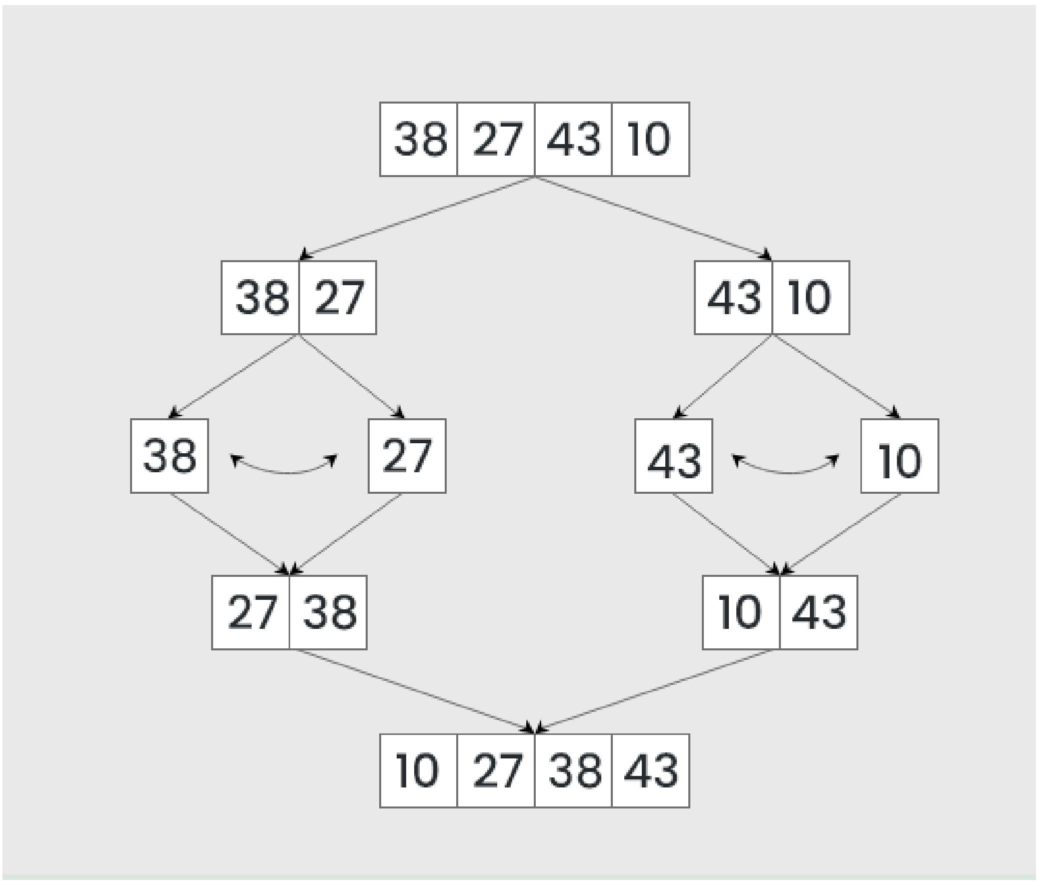
Sorted Order: In already sorted arrays, the curve should show a linear increase, but there is a slight curve in the graph. This may be due to other factors that prevent the best-case scenario from being exactly O(n). However, the shortest times are still seen in this scenario, which is consistent with theoretical expectations.

Reversed Order: In reverse sorted arrays, the line shows a significant exponential increase, which matches the expected O(n^2) complexity for the worst-case scenario.

Merge-Sort and return the median

Merge-Sort: Merge sort is a sorting algorithm that follows the divide-and-conquer approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array.

In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.



Merge-Sort Example

Time Complexity of Merge-sort and return the median:

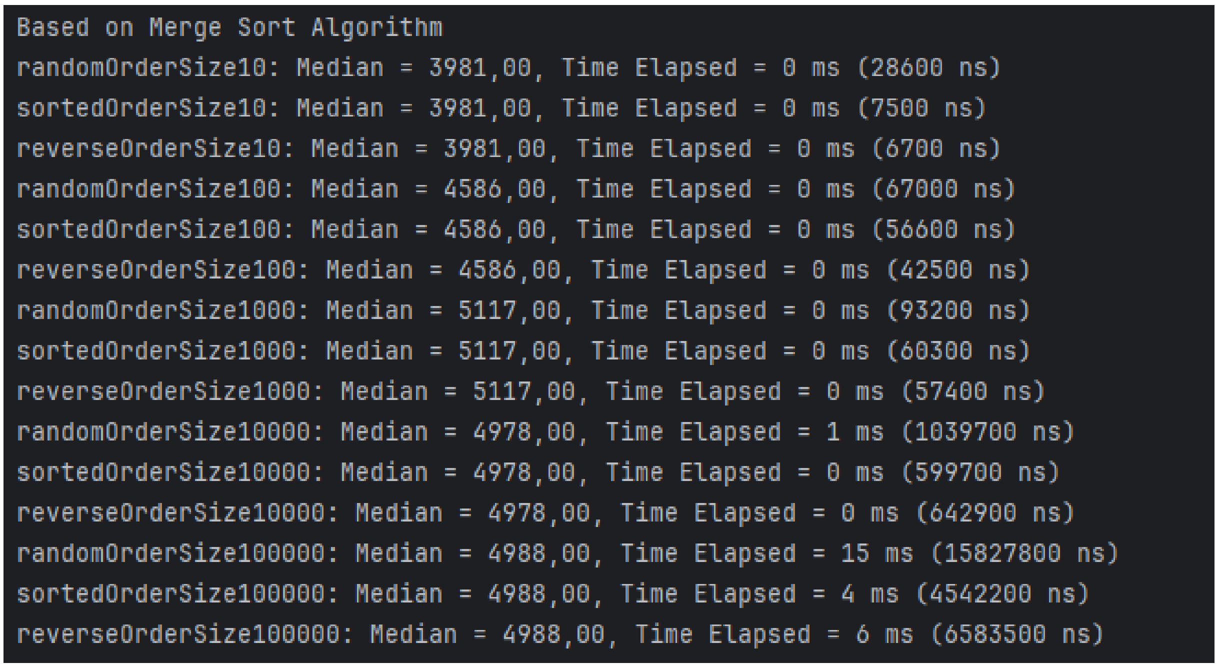
Time complexity of merge-sort in all cases is O(nlogn). This consistent performance across different scenarios is one of the key strengths of merge sort.

Divide: The algorithm divides the array into two halves, recursively sorting each half. The division process continues until it reaches subarrays of size one.

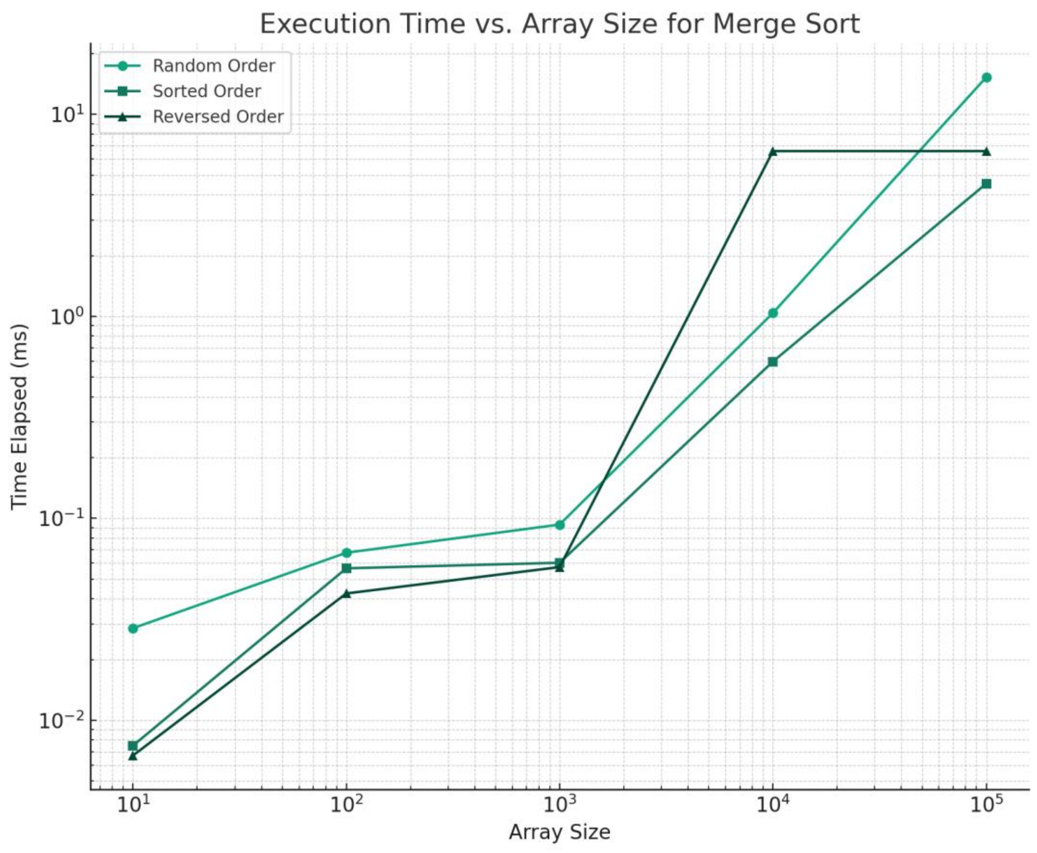
Conquer: It then merges these sorted subarrays back together. The merging process ensures that the newly formed array is also sorted.

Combining Costs: The merging of two halves takes linear time, i.e. O(n), for each merge step. Since the array is repeatedly divided into half until the size of sub arrays reaches one, there will be “log n” levels of merging. Thus, the total time taken for the merging process across all levels of the recursion is O(nlogn)

We are finding the median after the sorting process is completed and time complexity of finding median in a sorted array is O(1). Therefore, the overall time complexity is still O(nlogn)



```
Based on Merge Sort Algorithm
randomOrderSize10: Median = 3981,00, Time Elapsed = 0 ms (28600 ns)
sortedOrderSize10: Median = 3981,00, Time Elapsed = 0 ms (7500 ns)
reverseOrderSize10: Median = 3981,00, Time Elapsed = 0 ms (6700 ns)
randomOrderSize100: Median = 4586,00, Time Elapsed = 0 ms (67000 ns)
sortedOrderSize100: Median = 4586,00, Time Elapsed = 0 ms (56600 ns)
reverseOrderSize100: Median = 4586,00, Time Elapsed = 0 ms (42500 ns)
randomOrderSize1000: Median = 5117,00, Time Elapsed = 0 ms (93200 ns)
sortedOrderSize1000: Median = 5117,00, Time Elapsed = 0 ms (60300 ns)
reverseOrderSize1000: Median = 5117,00, Time Elapsed = 0 ms (57400 ns)
randomOrderSize10000: Median = 4978,00, Time Elapsed = 1 ms (1039700 ns)
sortedOrderSize10000: Median = 4978,00, Time Elapsed = 0 ms (599700 ns)
reverseOrderSize10000: Median = 4978,00, Time Elapsed = 0 ms (642900 ns)
randomOrderSize100000: Median = 4988,00, Time Elapsed = 15 ms (15827800 ns)
sortedOrderSize100000: Median = 4988,00, Time Elapsed = 4 ms (4542200 ns)
reverseOrderSize100000: Median = 4988,00, Time Elapsed = 6 ms (6583500 ns)
```



Comparison of our findings with theoretical expectations:

- The execution time for sorting arrays does not seem to change significantly with the initial ordering of the array (random, sorted, reversed order).
- As expected, the time taken to sort arrays increases with the size of the array
- The graph supports this, showing a roughly linear increase in the log-log scale, which is characteristic of O(nlogn)

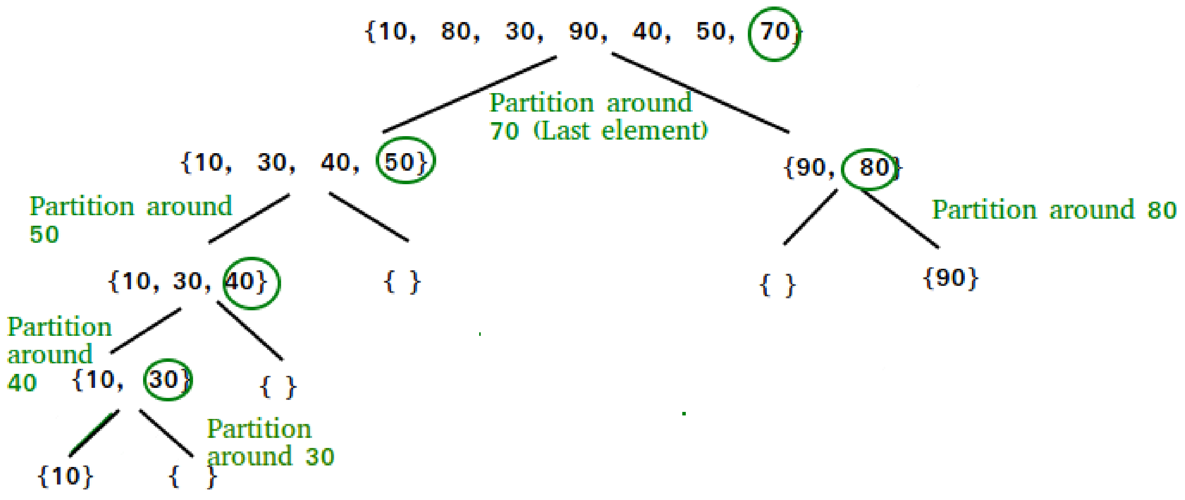
In conclusion, our findings meets the theoretical expectations. The process of sorting and then finding the median runs within the expected time bounds and the performance scales appropriately with the size of the input array

Quick-Sort and return the median(first element as pivot)

Quick-Sort: QuickSort is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

The key process in quickSort is a partition(). The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot.

Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.



Quick-Sort Example

In our scenario, the pivot is always the first element of the array or sub-array. This choice of pivot affects the performance of Quick-Sort.

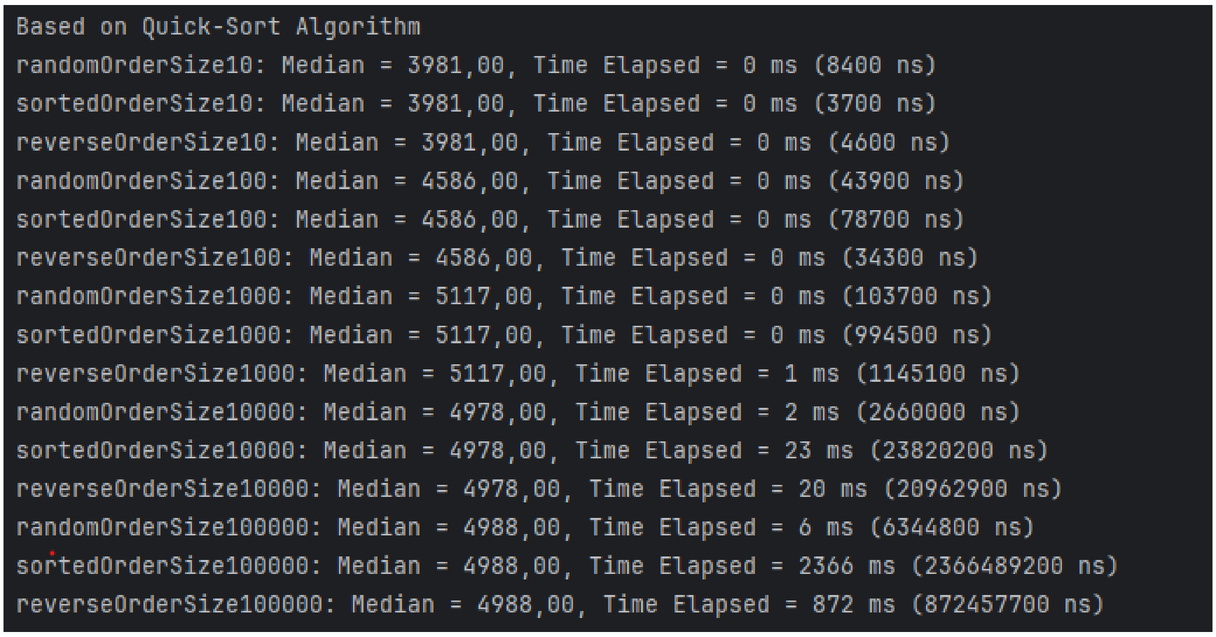
Time Complexity of Quick-Sort and return the median:

Worst-case scenario: The worst-case occurs when the pivot element is the smallest or largest element in the array. This happens if the array is already sorted (either in ascending or descending order). Every partition then divides the array into two sub-arrays of size 0 and n-1. This results in a time complexity of O(n^2)

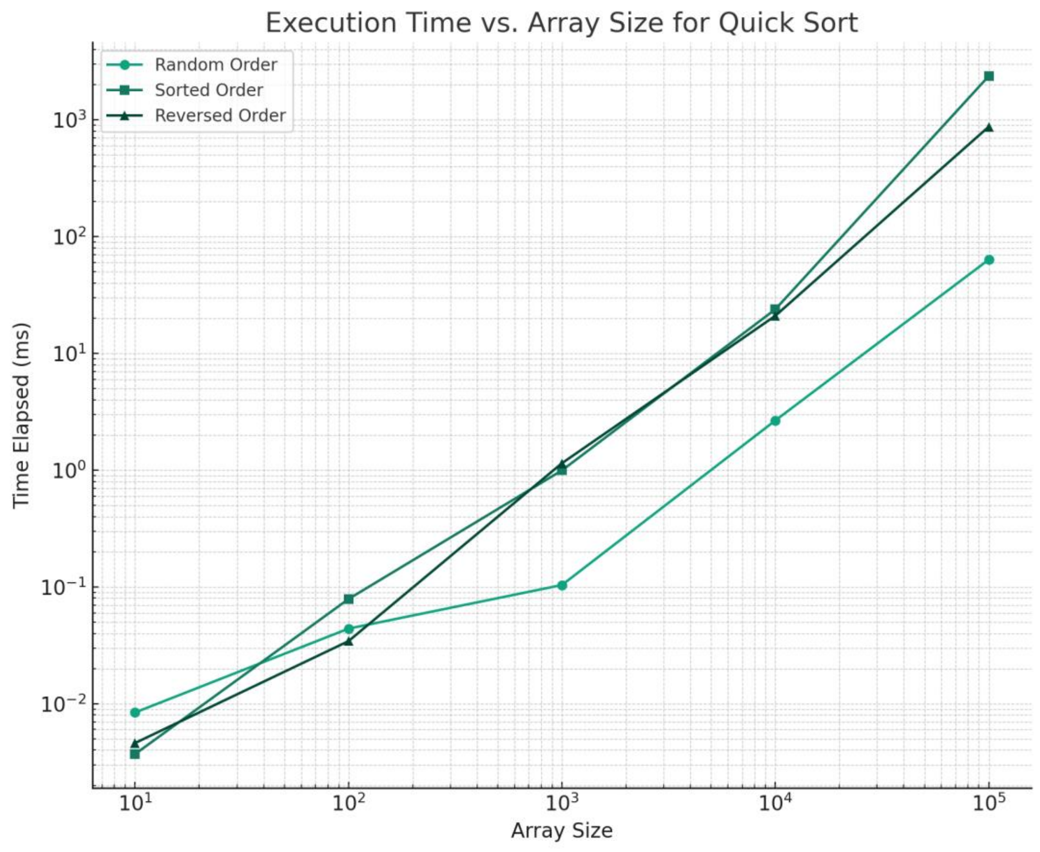
Best-case scenario: The best-case occurs when the pivot element is always the median of the array, resulting in two equal halves. This reduces the height of the recursion tree, leading to a time complexity of O(nlogn).

Average-case scenario: On average, assuming that the pivot element is randomly chosen (though in our case, it's fixed to the first element, but let's consider random selection for theoretical average case), the array is divided into two parts of relatively equal size. This also results in a time complexity of O(nlogn).

After all elements are sorted time complexity of finding median is O(1).



```
Based on Quick-Sort Algorithm
randomOrderSize10: Median = 3981,00, Time Elapsed = 0 ms (8400 ns)
sortedOrderSize10: Median = 3981,00, Time Elapsed = 0 ms (3700 ns)
reverseOrderSize10: Median = 3981,00, Time Elapsed = 0 ms (4600 ns)
randomOrderSize100: Median = 4586,00, Time Elapsed = 0 ms (43900 ns)
sortedOrderSize100: Median = 4586,00, Time Elapsed = 0 ms (78700 ns)
reverseOrderSize100: Median = 4586,00, Time Elapsed = 0 ms (34300 ns)
randomOrderSize1000: Median = 5117,00, Time Elapsed = 0 ms (103700 ns)
sortedOrderSize1000: Median = 5117,00, Time Elapsed = 0 ms (994500 ns)
reverseOrderSize1000: Median = 5117,00, Time Elapsed = 1 ms (1145100 ns)
randomOrderSize10000: Median = 4978,00, Time Elapsed = 2 ms (2660000 ns)
sortedOrderSize10000: Median = 4978,00, Time Elapsed = 23 ms (23820200 ns)
reverseOrderSize10000: Median = 4978,00, Time Elapsed = 20 ms (20962900 ns)
randomOrderSize100000: Median = 4988,00, Time Elapsed = 6 ms (6344800 ns)
sortedOrderSize100000: Median = 4988,00, Time Elapsed = 2366 ms (2366489200 ns)
reverseOrderSize100000: Median = 4988,00, Time Elapsed = 872 ms (872457700 ns)
```



Comparison of our findings with theoretical expectations:

- The time taken to sort random order arrays increases roughly linearly on the log-log plot, which suggest that the time complexity is closer to O(nlogn), which is expected for average case of Quick-sort.
- For sorted and reversed order arrays the time increases significantly with larger array sizes. The time taken is much higher than for random order. This indicates the worst case time complexity of $O(n^2)$, which is expected when the first element is the pivot and the array is already sorted (whether in order or reversed).

In conclusion, Our findings meets the theoretical expectations of Quick-sort. Random order demonstrates average-case as expected while sorted and reversed order demonstrates worst-case as expected.

Store in Max-heap and apply [n/2] times max removal.

Max Heap: Where the value of the root node is greater than or equal to either of its children.

Max Heap Construction Algorithm:

Step 1 − Create a new node at the end of heap.

Step 2 − Assign new value to the node.

Step 3 − Compare the value of this child node with its parent.

Step 4 − If value of parent is less than child, then swap them.

Step 5 − Repeat step 3 & 4 until Heap property holds.

Max Heap Deletion Algorithm:

Step 1 − Remove root node.

Step 2 − Move the last element of last level to root.

Step 3 − Compare the value of this child node with its parent.

Step 4 − If value of parent is less than child, then swap them.

Step 5 − Repeat step 3 & 4 until Heap property holds.

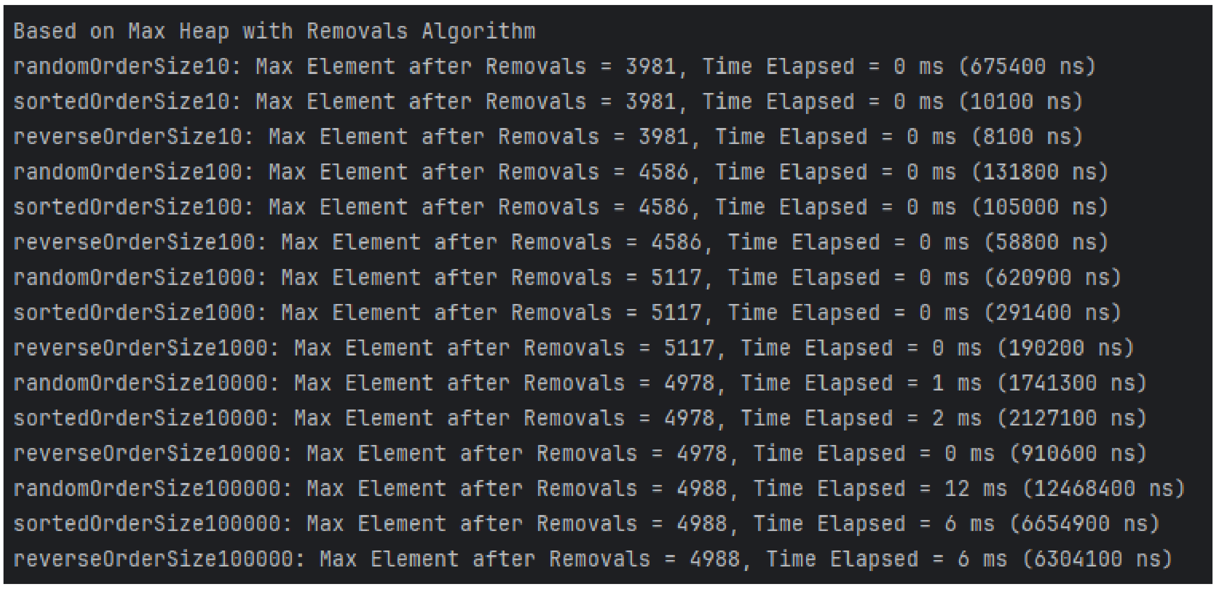
Time complexity of store in max-heap and apply [n/2] max removal:

Constructing the Max-Heap: Inserting n elements into a max-heap has a time complexity of O(n).

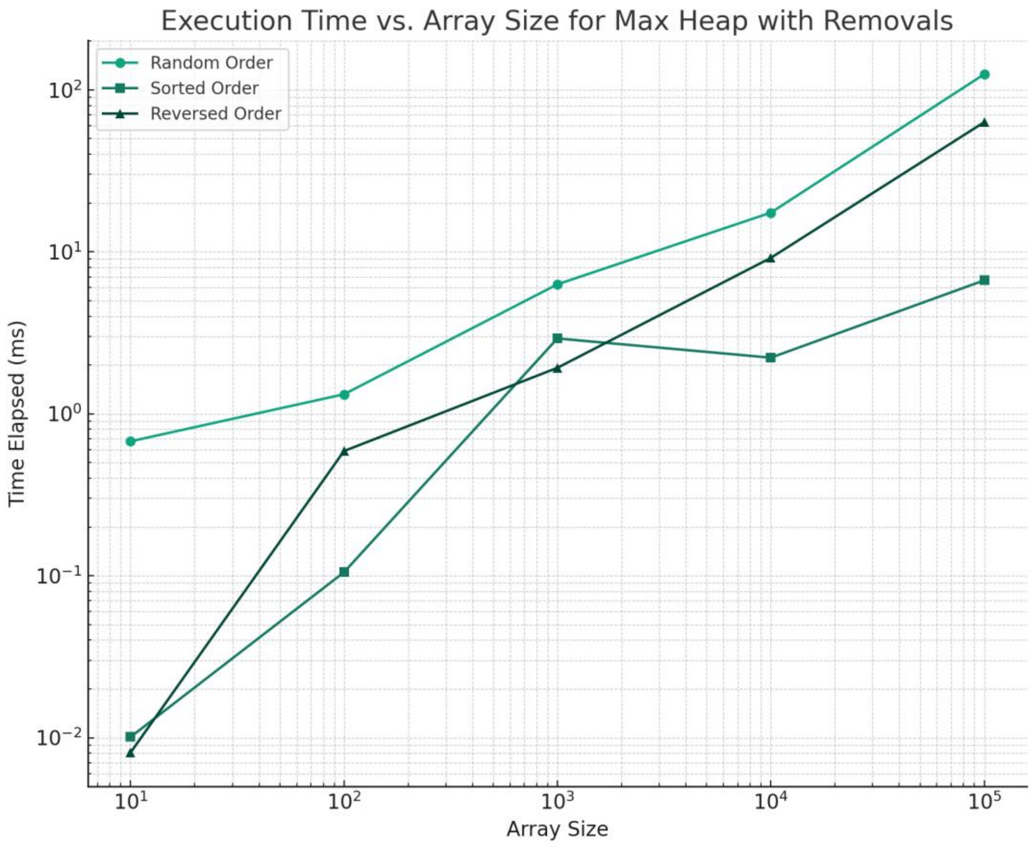
Removing the Max Element [n/2] Times: Each removal of the maximum element from a max heap is O(log n). Since we are doing this [n/2] times, the time complexity is O((n/2)logn).

Total time Complexity: O(n) + O((n/2)logn) = O(nlogn).

For this algorithm best, worst and average case scenarios has same time complexity.



```
Based on Max Heap with Removals Algorithm
randomOrderSize10: Max Element after Removals = 3981, Time Elapsed = 0 ms (675400 ns)
sortedOrderSize10: Max Element after Removals = 3981, Time Elapsed = 0 ms (10100 ns)
reverseOrderSize10: Max Element after Removals = 3981, Time Elapsed = 0 ms (8100 ns)
randomOrderSize100: Max Element after Removals = 4586, Time Elapsed = 0 ms (131800 ns)
sortedOrderSize100: Max Element after Removals = 4586, Time Elapsed = 0 ms (105000 ns)
reverseOrderSize100: Max Element after Removals = 4586, Time Elapsed = 0 ms (58800 ns)
randomOrderSize1000: Max Element after Removals = 5117, Time Elapsed = 0 ms (620900 ns)
sortedOrderSize1000: Max Element after Removals = 5117, Time Elapsed = 0 ms (291400 ns)
reverseOrderSize1000: Max Element after Removals = 5117, Time Elapsed = 0 ms (190200 ns)
randomOrderSize10000: Max Element after Removals = 4978, Time Elapsed = 1 ms (1741300 ns)
sortedOrderSize10000: Max Element after Removals = 4978, Time Elapsed = 2 ms (2127100 ns)
reverseOrderSize10000: Max Element after Removals = 4978, Time Elapsed = 0 ms (910600 ns)
randomOrderSize100000: Max Element after Removals = 4988, Time Elapsed = 12 ms (12468400 ns)
sortedOrderSize100000: Max Element after Removals = 4988, Time Elapsed = 6 ms (6654900 ns)
reverseOrderSize100000: Max Element after Removals = 4988, Time Elapsed = 6 ms (6304100 ns)
```



Comparison of our findings with theoretical expectations:

- The execution time increases as the size of the input array grows, which is expected because both heap construction and element removal scale with the size of the input.
- The lines appear to have a slope of approximately 1 in the log-log scale, which suggests an O(nlogn) time complexity which is total time complexity for storing all elements in a max-heap and apply (n/2) times max removal.
- There is a bit difference between random order and sorted, reversed order in terms of execution time. Normally there should not be but it is a small difference and it might be caused by different factors.

In conclusion, our experimental findings meets with the theoretical expectations.

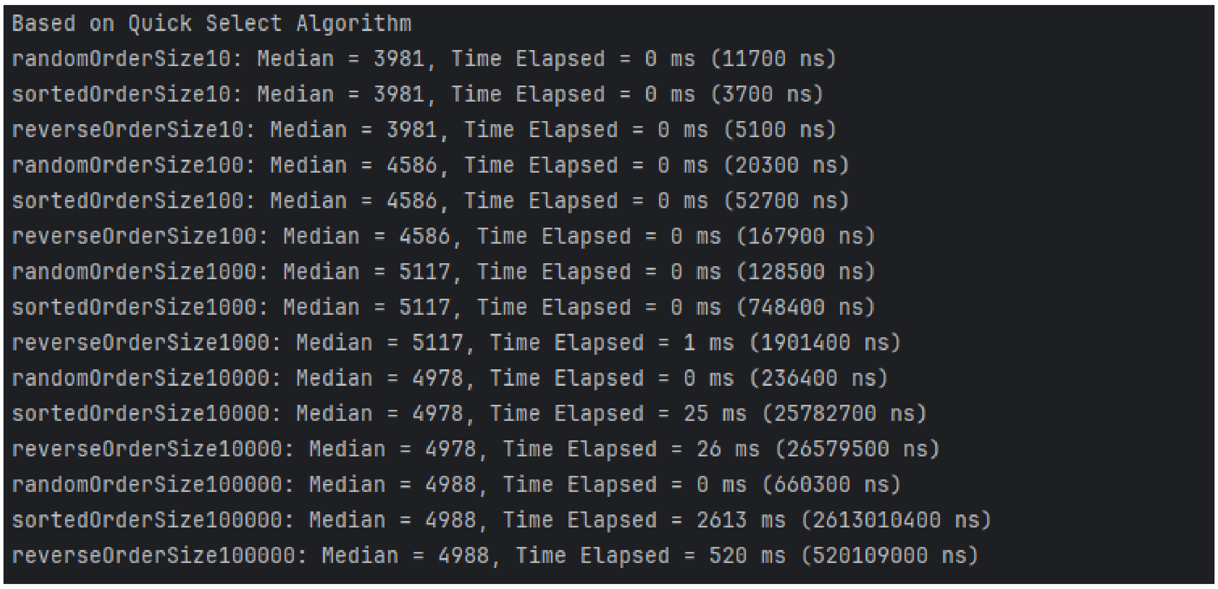
Apply quick select algorithm, first element as pivot

Time Complexity of Quick Select:

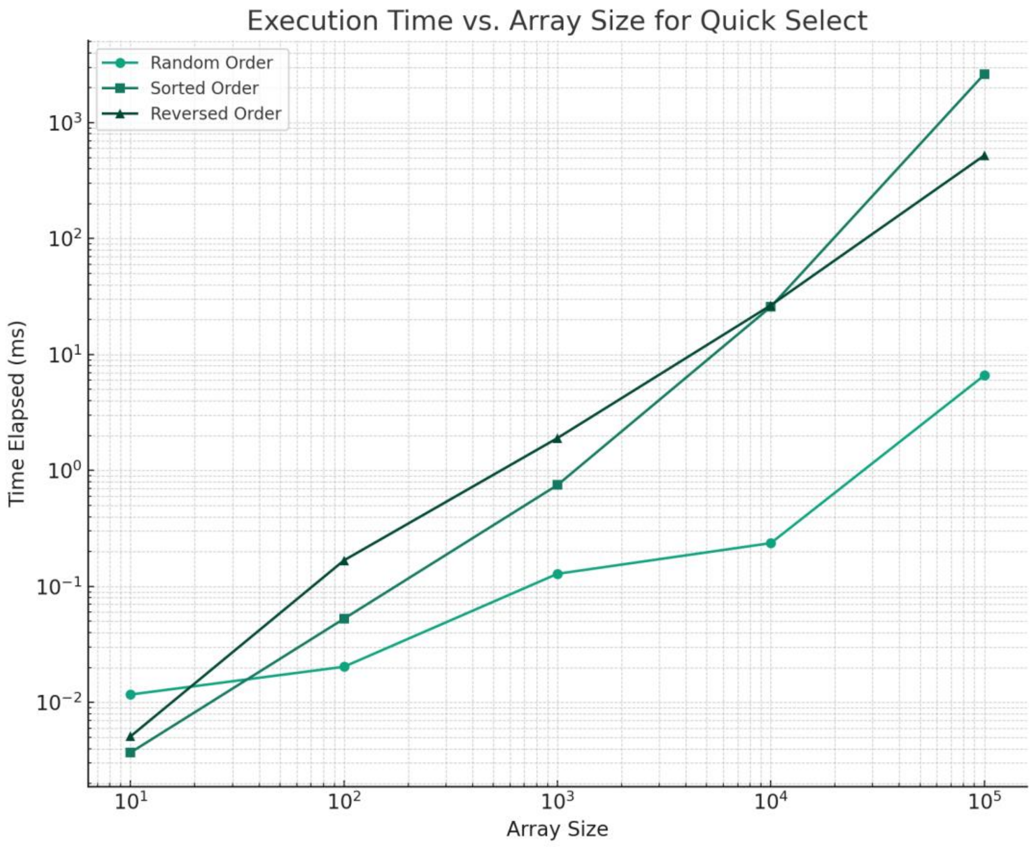
Best and Average Case: On average, the partitions split the array into parts that reduce the problem size significantly with each recursive step. This gives Quick select an average-case time complexity of O(n).

Worst Case: The worst case occurs when the chosen pivot is the smallest or largest element in the array, leading to very unbalanced partitions. This results in recursive calls that decrease the problem size by only one element at each step, leading to a time complexity of O(n^2).

Since we chose first element as pivot it can lead to the worst case scenario, especially if the array is already sorted.



```
Based on Quick Select Algorithm
randomOrderSize10: Median = 3981, Time Elapsed = 0 ms (11700 ns)
sortedOrderSize10: Median = 3981, Time Elapsed = 0 ms (3700 ns)
reverseOrderSize10: Median = 3981, Time Elapsed = 0 ms (5100 ns)
randomOrderSize100: Median = 4586, Time Elapsed = 0 ms (20300 ns)
sortedOrderSize100: Median = 4586, Time Elapsed = 0 ms (52700 ns)
reverseOrderSize100: Median = 4586, Time Elapsed = 0 ms (167900 ns)
randomOrderSize1000: Median = 5117, Time Elapsed = 0 ms (128500 ns)
sortedOrderSize1000: Median = 5117, Time Elapsed = 0 ms (748400 ns)
reverseOrderSize1000: Median = 5117, Time Elapsed = 1 ms (1901400 ns)
randomOrderSize10000: Median = 4978, Time Elapsed = 0 ms (236400 ns)
sortedOrderSize10000: Median = 4978, Time Elapsed = 25 ms (25782700 ns)
reverseOrderSize10000: Median = 4978, Time Elapsed = 26 ms (26579500 ns)
randomOrderSize100000: Median = 4988, Time Elapsed = 0 ms (660300 ns)
sortedOrderSize100000: Median = 4988, Time Elapsed = 2613 ms (2613010400 ns)
reverseOrderSize100000: Median = 4988, Time Elapsed = 520 ms (520109000 ns)
```



Comparison of our findings with theoretical expectations:

- Random Order Arrays: In our findings execution time increases linearly with the number of elements in the array like in average case time complexity of O(n).
- Sorted and Reversed Order Arrays: worst case time complexity for quick select, when always choosing the first element as the pivot, occurs if the array is already sorted (in order or reversed) which is O(n^2). Our results reflect this, with time dramatically increasing as the array size increases.

In conclusion, our experimental findings meets the theoretical expectations.

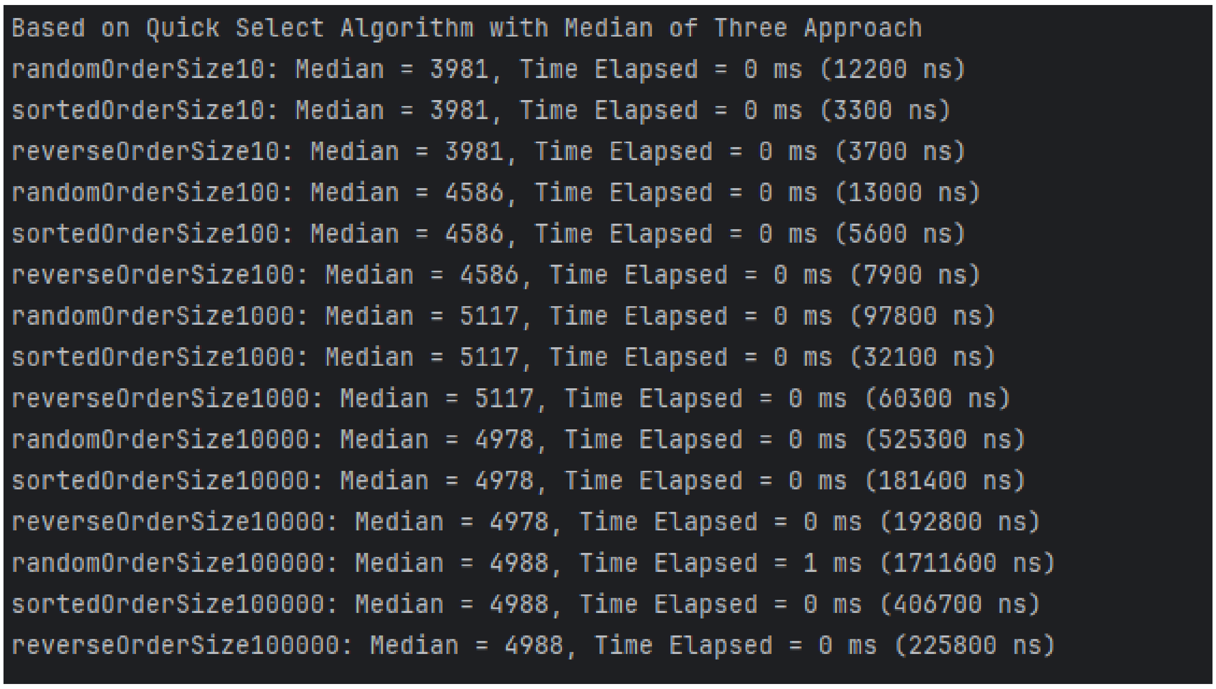
Apply quick select algorithm, use median-of-three pivot selection

Time Complexity of quick select with median-of-three pivot selection:

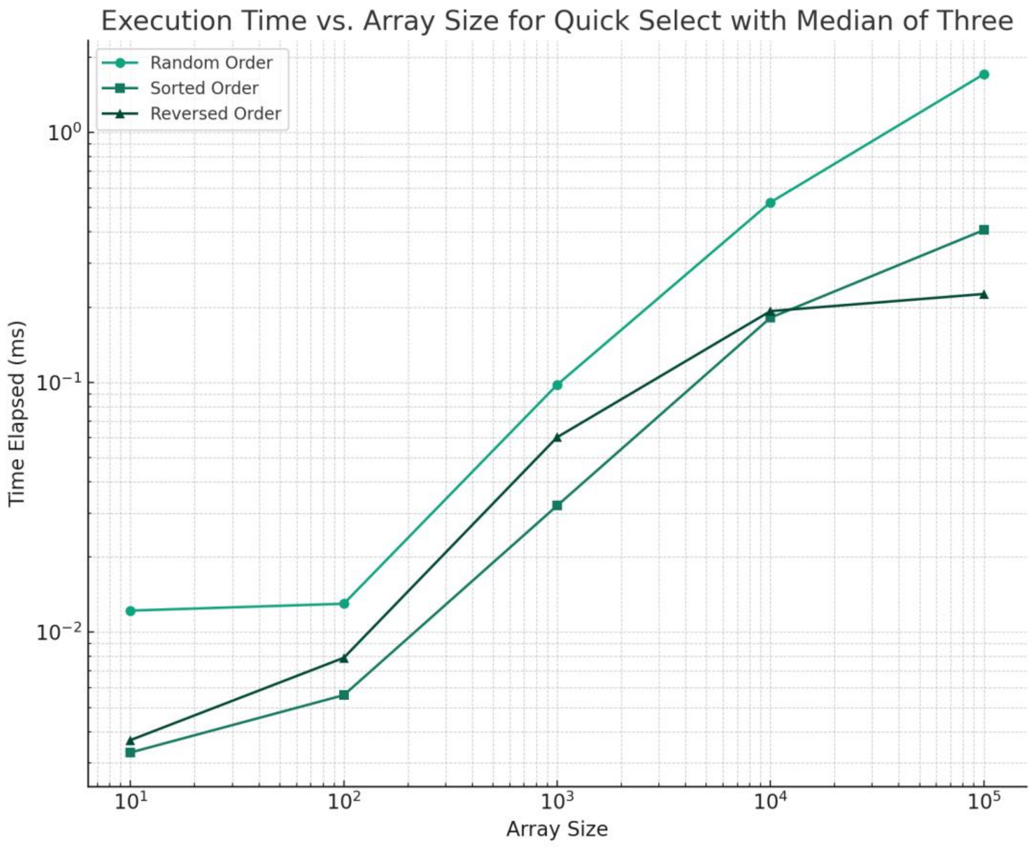
Average Case: Average case time complexity of quick select using median-of-three pivot selection remains O(n), similar to quick select.

Worst Case: The worst case time complexity remains $O(n^2)$. Even though using median-of-three pivot selection reduces the chance of consistently bad pivot choices, it does not eliminate the possibility.

In summary, while the median-of-three method does not change the theoretical worst-case time complexity of Quick select, it significantly improves the practical performance and reliability of the algorithm, especially in the average case.



```
Based on Quick Select Algorithm with Median of Three Approach
randomOrderSize10: Median = 3981, Time Elapsed = 0 ms (12200 ns)
sortedOrderSize10: Median = 3981, Time Elapsed = 0 ms (3300 ns)
reverseOrderSize10: Median = 3981, Time Elapsed = 0 ms (3700 ns)
randomOrderSize100: Median = 4586, Time Elapsed = 0 ms (13000 ns)
sortedOrderSize100: Median = 4586, Time Elapsed = 0 ms (5600 ns)
reverseOrderSize100: Median = 4586, Time Elapsed = 0 ms (7900 ns)
randomOrderSize1000: Median = 5117, Time Elapsed = 0 ms (97800 ns)
sortedOrderSize1000: Median = 5117, Time Elapsed = 0 ms (32100 ns)
reverseOrderSize1000: Median = 5117, Time Elapsed = 0 ms (60300 ns)
randomOrderSize10000: Median = 4978, Time Elapsed = 0 ms (525300 ns)
sortedOrderSize10000: Median = 4978, Time Elapsed = 0 ms (181400 ns)
reverseOrderSize10000: Median = 4978, Time Elapsed = 0 ms (192800 ns)
randomOrderSize100000: Median = 4988, Time Elapsed = 1 ms (1711600 ns)
sortedOrderSize100000: Median = 4988, Time Elapsed = 0 ms (406700 ns)
reverseOrderSize100000: Median = 4988, Time Elapsed = 0 ms (225800 ns)
```



Comparison of our findings with theoretical expectations:

- With median of three pivot selection execution time reduced significantly compared to selecting first element as pivot.
- The results show that the algorithm scales well as the array size increases. Even at the largest tested size, the time taken to find the median remains minimal.
- For random order arrays, the time complexity appears to behave very close to the average case expectation which is O(n).
- For sorted and reversed arrays, which typically represent the worst case scenarios for many quick select pivot selection strategies, our algorithm performing very well. This indicates that the median-of-three pivot selection is effective at avoiding worst case scenarios.

In conclusion, our findings meets theoretical expectations.

Apply quick select algorithm with median-of-medians pivot selection

We could not implement the algorithm.

Division of Labor

Step 1: Designing the Experiment: Ayşe Sena AYDEMİR, Emre Cihan VARLI, Fatih Erkam DİNLER

Step 2: Coding and Running: Ayşe Sena AYDEMİR, Emre Cihan VARLI, Fatih Erkam DİNLER

Step 3: Illustrating and Analyzing Results: Ayşe Sena AYDEMİR, Emre Cihan VARLI, Fatih Erkam DİNLER