MARMARA UNIVERSITY

FACULTY OF ENGINEERING

CSE2246

Analysis of Algorithms

Project - 1

Instructure: Doç. Dr. ÖMER KORÇAK

Date: 14.05.2025

CONTENTS

PURPOSE

The aim of this project is to design an experiment to analyze the time complexities of different algorithms that focus on solving the same problem: *Finding The Majority Element* in an array.

INTRODUCTION

Finding Majority Element is a problem that in a number sequence with n elements, finding if there is a number which repeated more than half of n, and what that number is. The element which repeated more than half of the array size called the majority element of that array.

We solved *the Finding Majority Element* problem using these algorithms and we measured execution time for each algorithm at different arrays. We used *Brute Force*, sorting based algorithms (*Insertion sort, Merge sort, Quick sort*), *Divide and Conquer, Hashing, Boyer Moore Majority Vote* algorithms and we implement one more algorithm which name is *Abbasov's Algorithm* (splaying-based). We implemented these algorithms in C.

Here is the table showing theoretical time complexities. We will try to observe these asymptotic functions in our experiment.

Table 1: theoretical time complexities

Theoretical Time Complexities			
Algorithm	Best case	Worst Case	Avarage case
Brute Force	O(n)	O(n^2)	O(n^2)
Insertion-sort	O(n)	O(n^2)	O(n^2)
Merge-sort	O(nlogn)	O(nlogn)	O(nlogn)
Quick-sort	O(nlogn)	O(n^2)	O(nlogn)
Divide&Conquer	O(n)	O(nlogn)	O(nlogn)
Hashing	O(n)	O(n^2)	O(n)
Boyer-Moore	O(n)	O(n)	O(n)
Abbasov	O(n)	O(n^2)	O(nlogn)

We generated arrays for different cases and plotted graphs in Python. In input generation, there are different sizes (from 200 to 10000) arrays are generated. There are some special arrays which represent the best, worst and average cases for different algorithms for each size. These are going to be mentioned in algorithm's explanations.

We utilized predefined Python libraries such as Matplotlib and Seaborn for visualizing graphs, and NumPy and Pandas for data manipulation. Algorithms first executed in C, and their output which consist of array size and execution time were saved to a file. This file was then used as input in Python to generate the corresponding graphs.

Our performance metric is physical amount of time. To measure the time, we used Q*ueryPerformanceCounter*. Q*ueryPerformanceCounter* is a Windows API function that provides high-resolution timing by accessing a hardware-based performance counter. It returns the current value of this counter, which increases at a constant rate defined by *QueryPerformanceFrequency*. This allows us to precisely measure time intervals, making it ideal for performance profiling. Unlike regular timing functions, it is much more precise and gives accurate results even if the computer's clock changes or the processor speed goes up or down.

EXPERIMENT

DIFFERENT ALGORITHMS FOR ALL POSSIBLE INPUTS

In this part, we have an input file that contains all possible inputs with different sizes (about 1700). We run all algorithms one by one (100 times for reasonable time values) with that input, and we try to see whether behaviors of algorithms match theoretical expectations.

Brute-force

A brute force algorithm is a simple method that tries all possible solutions to solve a problem. It doesn't use any shortcuts or smart tricks—it just goes through every option until it finds the right one.

So, our brute force algorithm checks each element in the input array whether it is the majority element or not. If elements indexed from 0 to n/2 are equal, this is the best case for this algorithm.

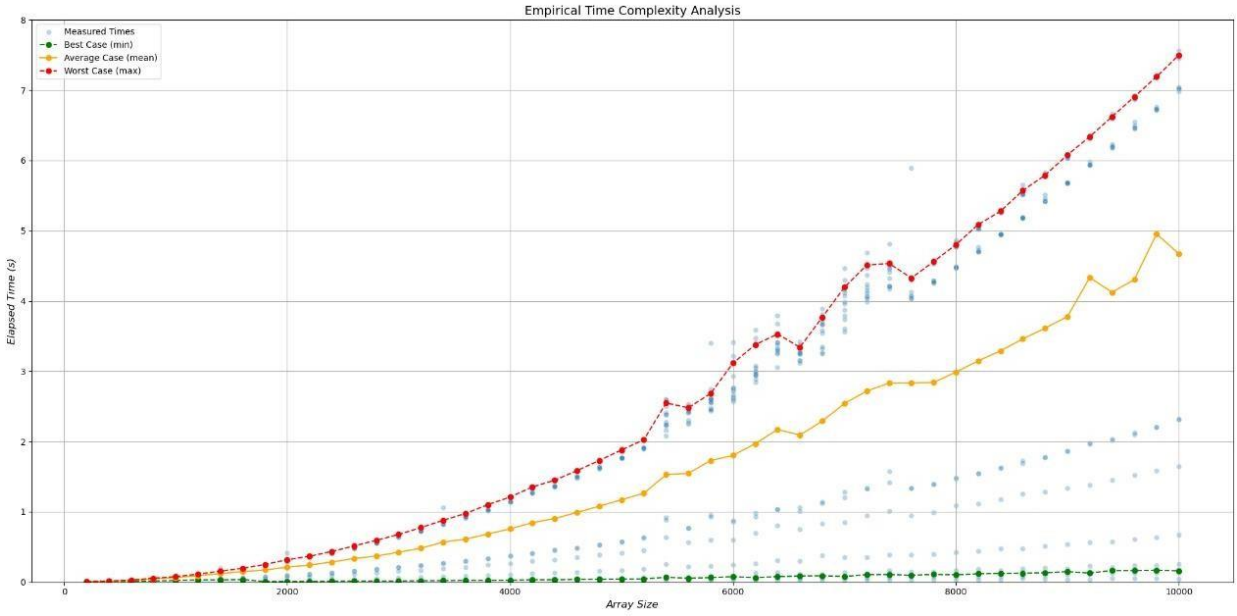
⇨ [7,7,7,7,1,2,3]

This is an example best case array because our algorithm checks firstly the first element and finds it is the majority. While finding, it just makes n/2 + 1 comparisons (O(n)).

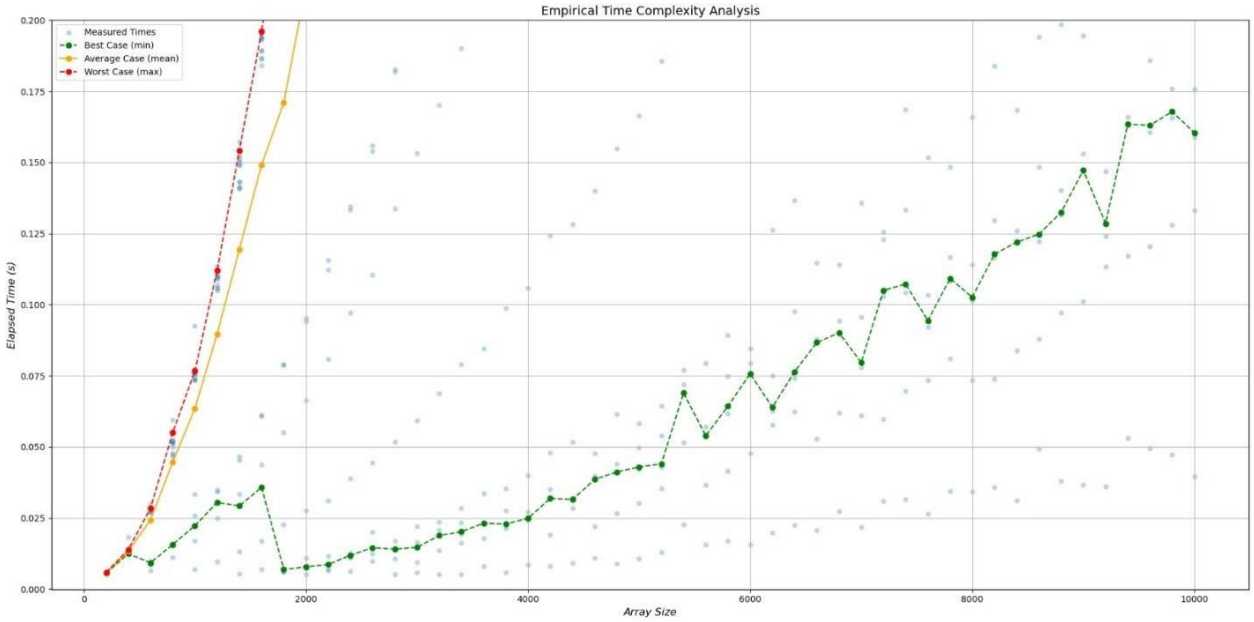
If there is no majority element in the input array, this case is the worst case for our brute-force algorithm.

⇨ [23,2,5,89,123,0,312]

Here it compares each element with each element. That makes n^2 comparisons. So, it is the worst-case scenario (O(n^2)).



Closer look:



As seen in the closer look, the best case (green) points are aligned linearly. On the other hand, yellow and red points have an quadratic growth rate. So, these results meet our expectations.

Insertion-sort Based Algorithm

Insertion sort-based algorithm is an algorithm that uses pre-sorting to solve Finding

Majority Problem. In this algorithm, the input array is sorted using insertion sort and algorithm counts middle element whether it exceeds more than half of array size or not. If it exceeds more than half of the array, algorithm returns this value, if that is not the case, it returns -1. Insertion sort is a sorting algorithm that processes array from the second element of array until the end one by one and swaps current processing elements with the previous element of it if it is greater. This swap operation continues until this element is located at its correct position through backward for every element.

For Insertion sort-based algorithm, an already sorted array is best case because at this case, algorithm does not do any shift operation.

⇨ [2,6,11,14,18,22]

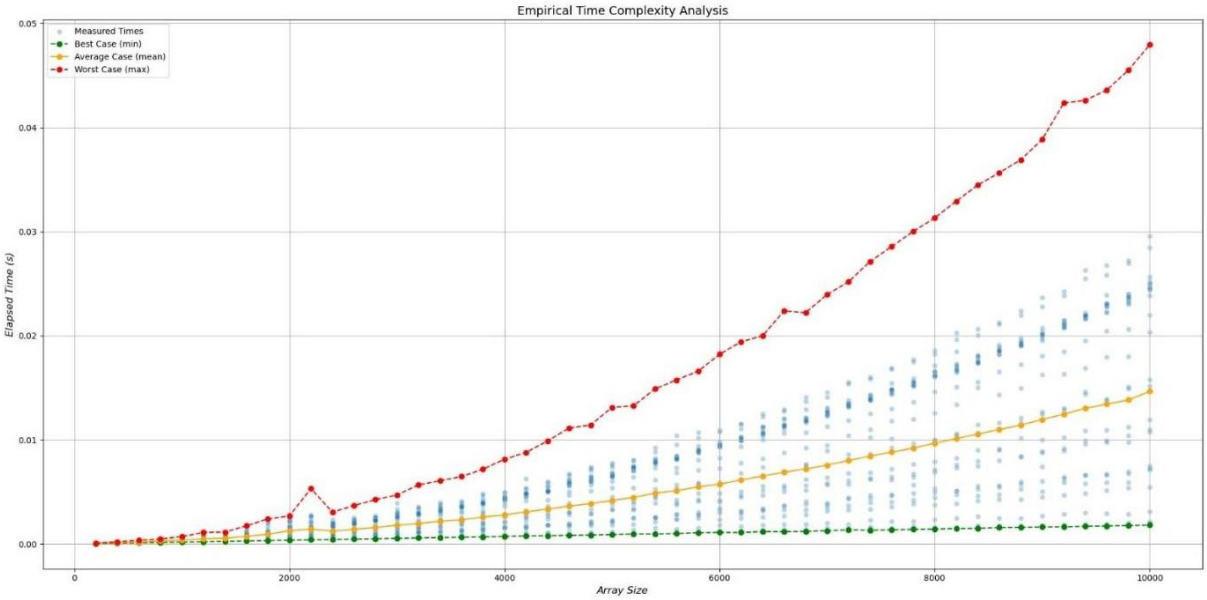
It makes only n-1 comparisons (in this example it makes 5 comparisons). This algorithm's best case time complexity is O(n).

For insertion sort-based algorithm, a reverse order array is the worst case because at this case, algorithm does swap operation for every element from their first position to beginning of the array.

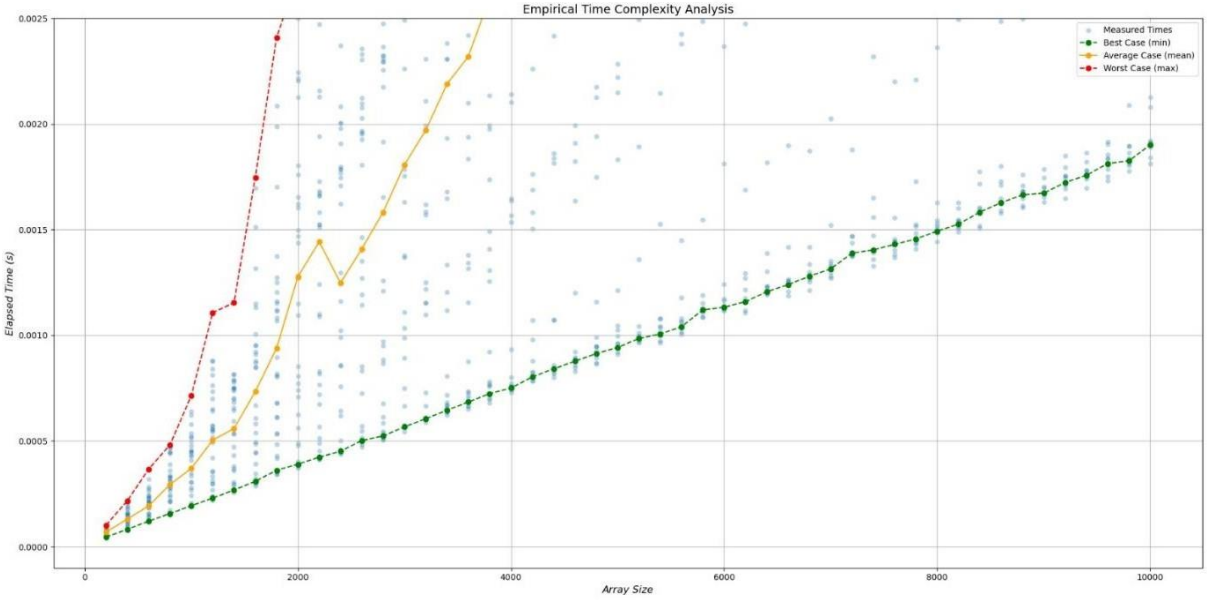
⇨ [22,18,15,11,6,2]

This array is the worst case of insertion sort-based algorithm because it is reverse ordered. For example, 2 will be swapped with every element one by one. Totally, it makes n*(n-1)/2 comparisons (10 comparisons for this example). This algorithm's worst case time complexity is $O(n^2)$.

In average case, this algorithms time complexity is $O(n^2)$ like worst case because in average algorithm swaps most of the elements toward back (not as much as worst case) and it takes O(n) time and algorithm does this for n elements. So the average time complexity is $O(n^2)$.



Closer look:



As seen in the graphs, the best case (green) points are aligned linearly. On the other hand, yellow and red points have a quadratic growth rate. So, these results meet our expectations.

Merge-Sort-Based-Algorithm

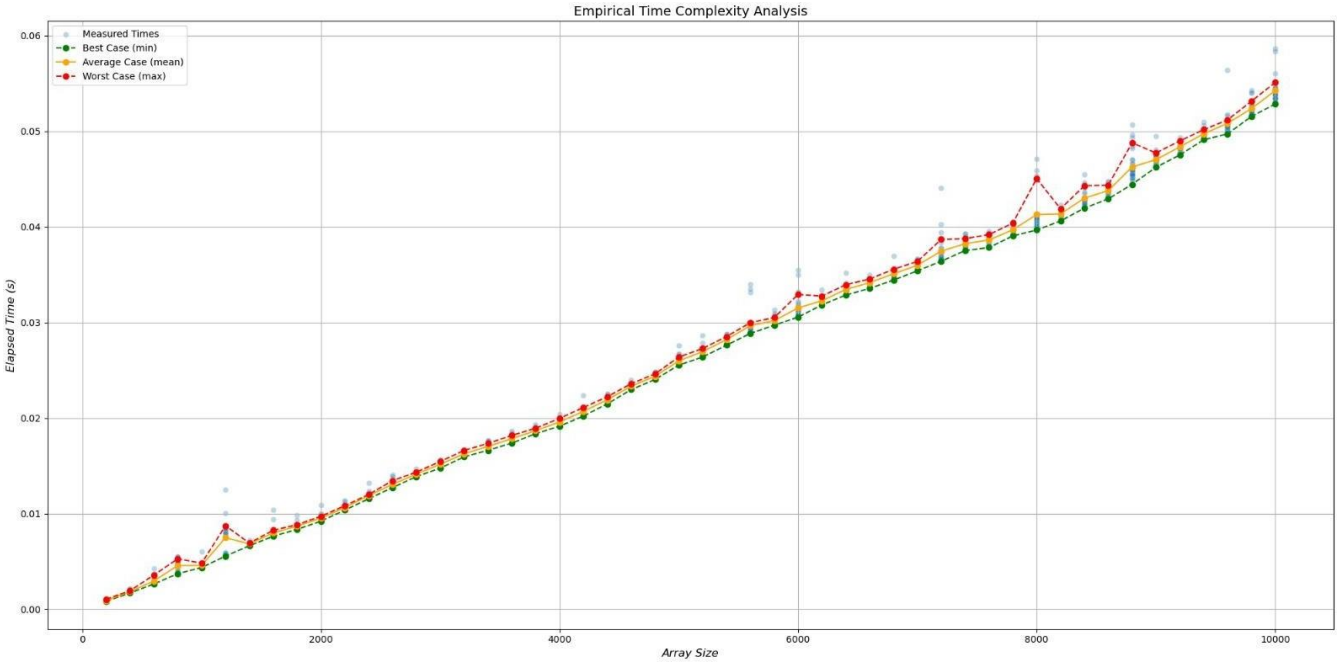
Merge sort-based algorithm is an algorithm that uses pre-sorting to solve the Finding Majority Problem. In this algorithm, the input array is sorted using merge sort and algorithm counts middle element whether it exceeds more than half of array size or not. if it exceeds more than half of the array, algorithm returns this value, if that is not the case, it returns -1. Merge sort is a sorting algorithm that if array has one element, merge sort returns it as sorted array. If array has more than one element, algorithm divides array to two arrays from middle and sorts both recursively, then merges both of sorted arrays.

In merge sort-based algorithms, the divide operation is performed regardless of the values in the array, but merging operation depends on values in the array.

[3,2,7,5] This array represents the best case for merge sort-based algorithm because there are 3 merging operations here and the last is merging [2,3] and [5,7]. While merging these two arrays, algorithm makes 2 comparisons because every element in first array is less than every element of the second array so comparison will be minimum (for two arrays which have n elements it makes n comparisons). This algorithm's best case time complexity is O(nlogn) because it divides into 2 (O(logn)) and merges(O(n)).

[5,2,3,7] This array represents the worst case for merge sort-based algorithm because there are 3 merging operations here and the last is merging [2,5] and [3,7]. While merging these two arrays, algorithm makes 3 comparisons because $2<3<5<7$ ($[a_1, a_2]$ and $[b_1, b_2]$ is the worst case for merging if $a_1<b_2<a_2<b_2$ or $b_1<a_1<b_2<a_2$) so comparison will be maximum (for two arrays which have n elements it makes 2*n-1 comparisons). This algorithm's worst case time complexity is O(nlogn) too because it divides into 2 (O(logn)) and merges(O(n)) again.

Merge sort-based algorithm has θ(nlogn) time complexity because its time complexity is n*logn at best, worst and average case.



As seen in the graphs, the best, worst and average cases are very near each other. Their asymptotic grow rate is same, and It is θ(nlogn) if we ignore measurement errors.

Quick-sort

Quick sort-based algorithm is an algorithm that uses pre-sorting to solve Finding Majority Problem. In this algorithm, the input array is sorted using quick sort and algorithm counts the middle element whether it exceeds more than half of array size or not. If it exceeds more than half of the array, algorithm returns this value, if that is not the case, it returns -1. Quick sort is a sorting algorithm that sorts the array using partitioning. Partitioning means that, in an array selecting an element as pivot and placing it to its correct position (every element before the pivot will be less than pivot and every element will be greater than pivot). Quick sort selects pivot and uses partition. It selects the first element as a pivot. With this method, it gets two halves divided by pivot. Then, it continues to select pivots and use them for partitioning in divided halves until every element is located at their correct position.

In quick sort-based algorithm, the efficiency depends on success of selecting pivot, if selecting pivot divides array into two halves approximately middle of the array at every iteration, algorithm can be more efficient according to time. If selecting pivot's correct position is near to first or last index of current array, algorithm's time efficiency reduces.

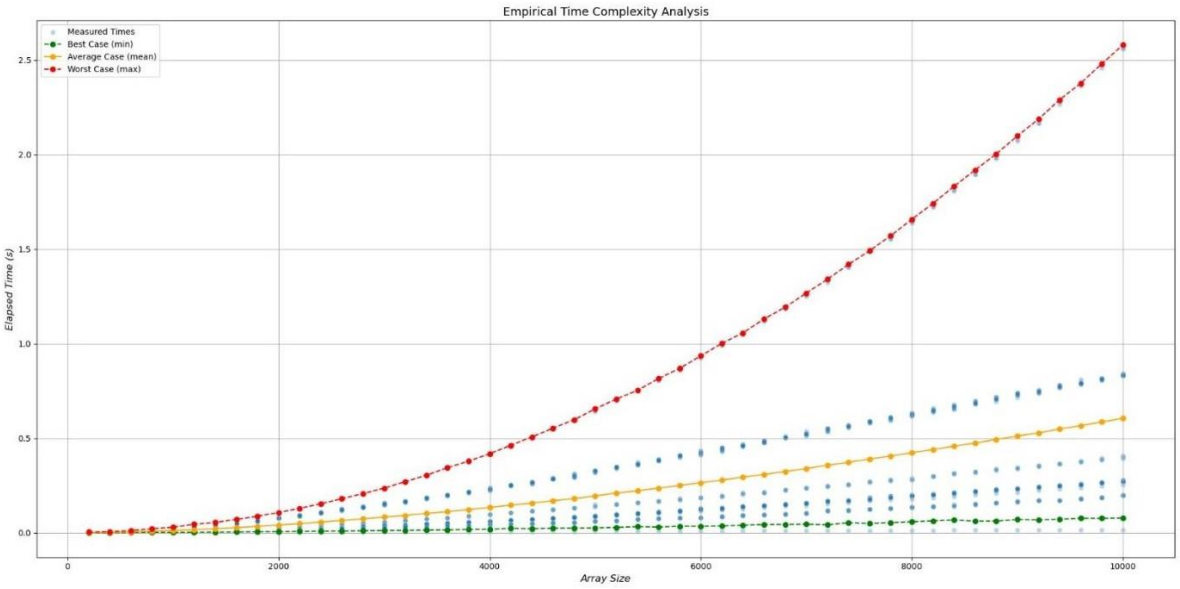
[2,1,3] is the best case for quick sort base algorithm because 2 is selected as pivot and new array is [1,2,3] which is sorted only one partitioning.

[1,2,3] is the worst case for quick sort algorithm because 1 is selected as pivot and new array is [1,2,3]. It seems to be sorted but algorithm does not know that because it did not process right of 1 so it selects 2 as pivot and makes one more partition too. After that it can be sure that the array is sorted.

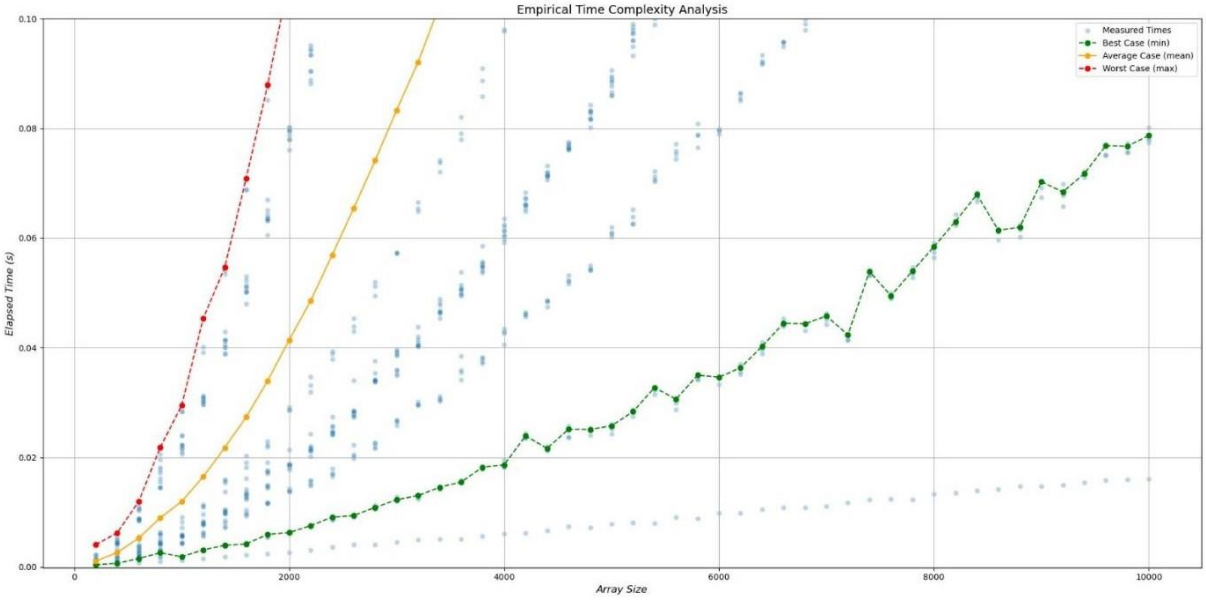
This algorithm's best case time complexity is O(nlogn) because at the best case, every time partition divides array into two arrays which are approximately equal size. So, there are O(logn) partitions and partitioning takes O(n) time. Counting middle element of sorted array takes O(n) time and O(nlogn) + O(n) equal to O(nlogn).

This algorithm's worst case time complexity is $O(n^2)$ because at the worst case, every time partition cannot divide array into two arrays, it can reduce not processed array only one element. So, there are O(n) partitions and partitioning takes O(n) time. Counting the middle element of sorted array takes O(n) time and $O(n^2) + O(n)$ equal to $O(n^2)$.

This algorithm's average case is O(nlogn) because selecting pivot badly at every step like worst case is very rare case. In average, selected pivot can divide array into two halves and counting middle element of sorted array takes O(n) time, O(nlogn) + O(n) equal to O(nlogn).



Closer look:



As seen in the graphs, the best case and average case grows n*logn asymptotically (average case has much bigger grow coefficient). On the other hand, the worst case grows with n^2 grow rate.

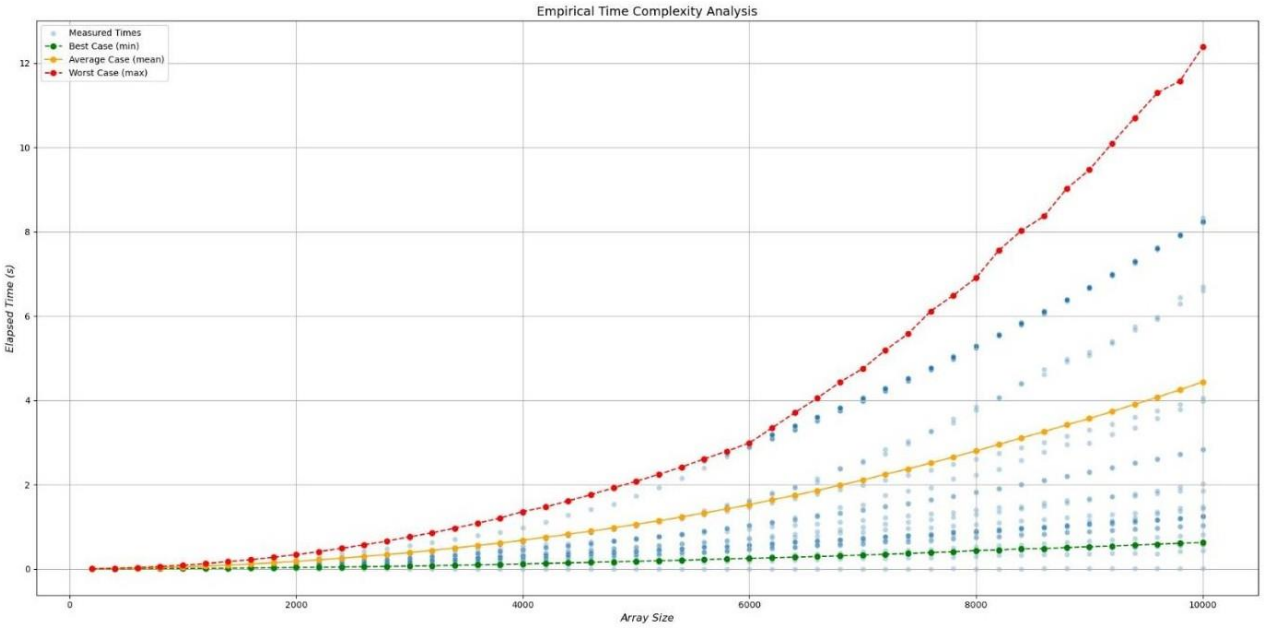
Divide & Conquer

Divide and conquer algorithm divides array into two subarrays and finds their majority recursively, if array has one element, then returns it. When two subarrays return a value (at merging step), if these two values are same, array's majority element is that value. If they are not the same, algorithm counts two numbers in the array and returns maximum one. If their counts are the same, then it returns the second's majority arbitrarily.

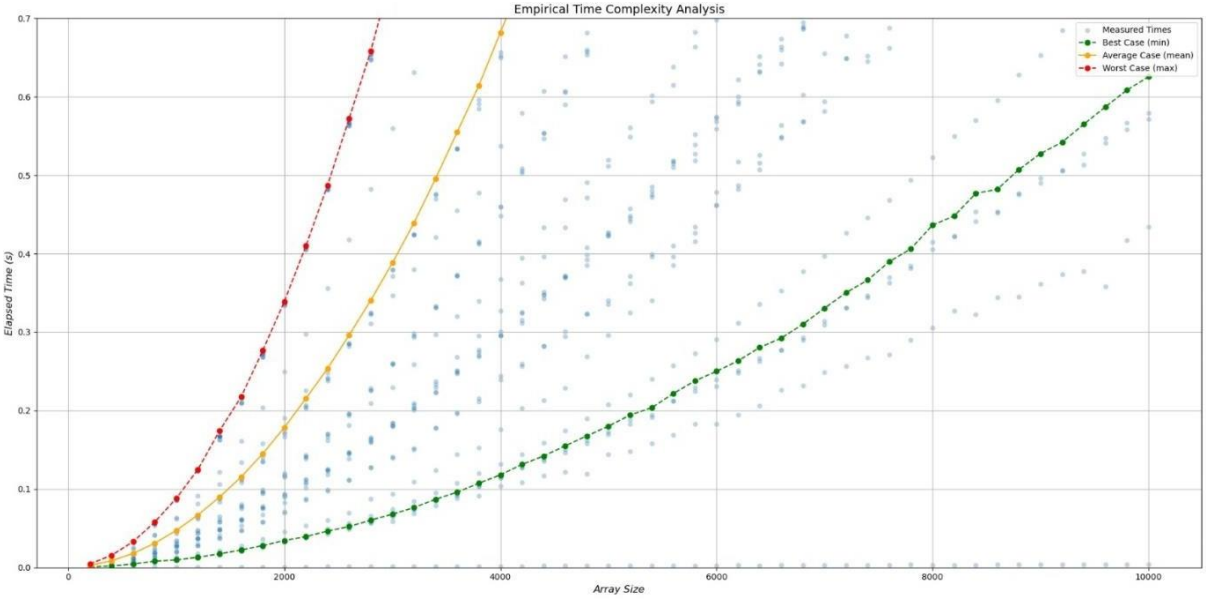
[2,2,2,2] is one of the best cases for this algorithm because the cases like this, algorithm never does count two numbers because returning value is always same for every subarray. At the best case, dividing and merging operations takes O(n) time.

[1,2,3,4] is one of the worst cases for this algorithm because it returns different values at every merging step and algorithm counts numbers in subarrays. Dividing and merging takes O(logn) time and counting takes O(n) time so total worst case time complexity of divide and conquer algorithm is O(nlogn) time.

In average, the possibility of returning different elements of two subarrays is very high so divide and conquer algorithm's average time complexity is O(nlogn) like worst case.



Closer look:



In the graphs, the worst case grows with n^2 grow rate which doesn't satisfy our expectations. Average case grows n*logn asymptoticly as expected. Best case grows linearly as expected.

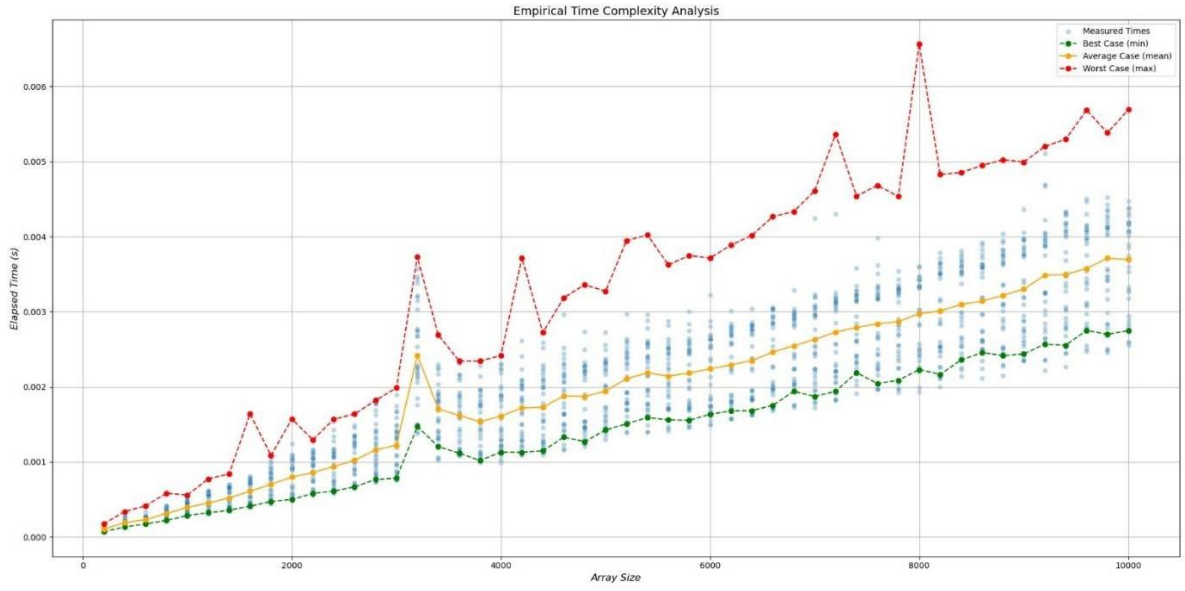
Hashing Algorithm

Hashing means placing every element of array to a hash table. If current value is already available at hash table, algorithm increments frequency of that value. This algorithm constructs a table which size is the smallest prime number of greater than two times array size. It places the numbers to the index of number mod hash size. If a collision occurs, algorithm places it to the next empty cell. After all elements are placed, algorithm counts every cell's frequency and if it finds a cell which frequency is greater than half of array, returns it. If there is no such cell, it returns -1.

[0,0,0,0] is the best case for hashing algorithm because it locates value of 0 at 0^{th} index with frequency 4. And when it starts to count every cell one by one, it finds frequency 4 which is greater than half of the array, at first index of hash table and returns it. There is n hashing for an array which size is n so the best case time complexity of this algorithm is O(n).

[7,18,29,40] is one of the worst cases of hashing algorithm because every value's hashing index is same (hash size is 11 and 7 mod 11 = 18 mod 11 = 29 mod 11 = 40 mod 11 = 7). The hashing takes $O(n^2)$ time for an array size of n and counting every frequency takes O(n) time, but it runs after hashing so total worst case time complexity of hashing algorithm is $O(n^2) + O(n) = O(n^2)$.

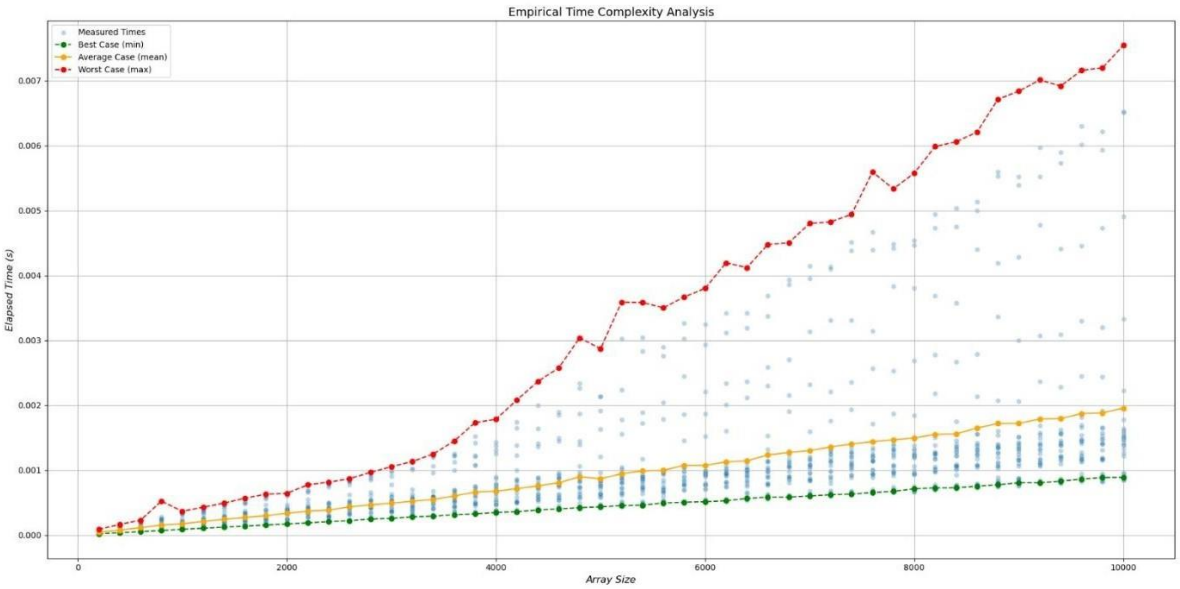
In average, hashing takes O(n) time, getting same hash values at most of the step is very rare case. Hashing takes O(n) time and counting every cell's frequency takes O(n) time, but it runs after hashing, so O(n) + O(n) = O(n) at the average case.



In hashing algorithm, getting true time values for algorithm is very hard because in some special cases, finding prime number for hash size can be harder and it increases time of algorithm unexpectedly. The linearity of the best and the average case seems correct. However, due to the unexpected measurements, worst case graph seems linear because of oscillating at very critical points, this graph should be quadratic.

Boyer-Moore Majority Vote

Boyer Moore majority vote is best algorithm for finding majority problem. It takes O(n) time for every input because this algorithm looks at every index one by one only one time and gets a candidate value for majority and checks if it is majority. It has one counter and one candidate. Firstly, the algorithm chooses the first element of the array as the candidate, then the algorithm starts to look at every index. If the current index has the same value as the candidate, the counter is incremented by one. If not, the counter is decremented by one. If the counter is equal to 0, the next index's value will be the new candidate. After this process is finished, the algorithm counts the number of the last candidate in array and if it occurs more than half of array size, algorithm returns it. If not, it returns -1 which means there is no majority element in that array. Because of searching array only two times at every case, Boyer Moore majority vote algorithm's time complexity is O(n) in best case, worst case and average case.



In this graph, the linearity of the best and average cases is very clear. The worst case is linear too but there is a breaking point at array size 4000. Most probably that, it stems from computer's measurement error. Also, there are some distances between lines of the best, the worst and the average cases because at the counting last candidate in the array step, if candidate is majority and algorithm finds it at earlier indexes, returns majority quickly.

Abbasov's Algorithm

Abbasov's algorithm is splaying based algorithm which constructs Binary Search Tree from the input array and apply splaying depending on the frequency of the last added node. If the frequency of that node is bigger than the frequency of the root, then it applies splaying operation.

The splay function is an operation that moves a given node to the root of the tree using a series of tree rotations. There are three main rotation cases:

- Zig (single rotation):

When the node is a left or right child of the root.

- Zig-Zig (double rotation in same direction):

Node and its parent are both left children or both right children.

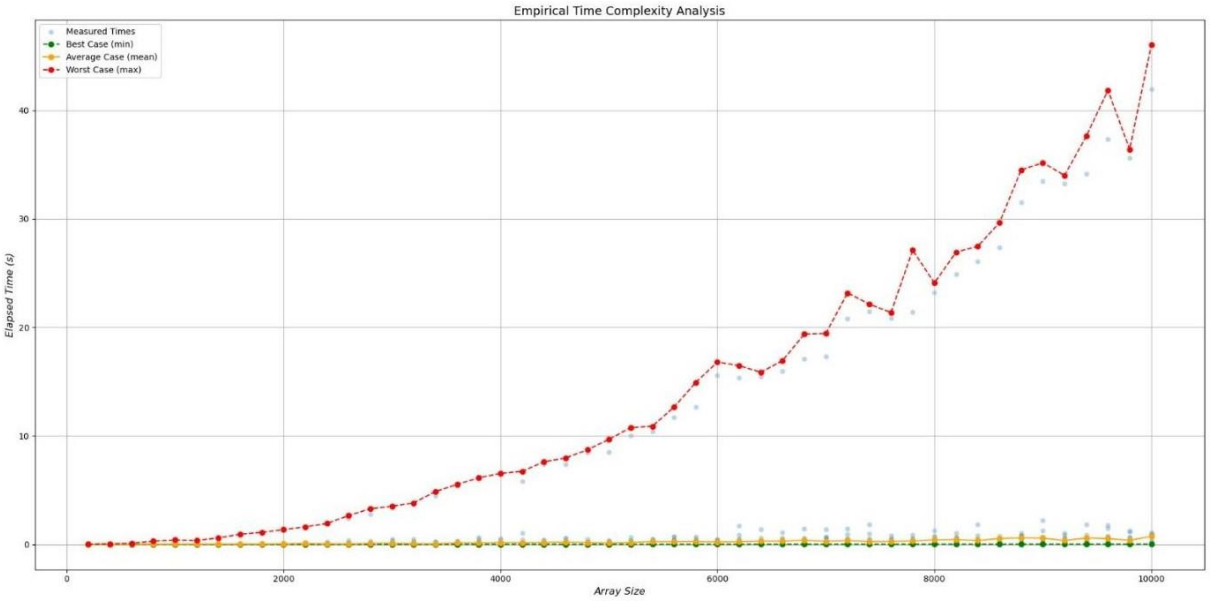
- Zig-Zag (double rotation in opposite directions):

Node is a left child, and parent is a right child, or vice versa.

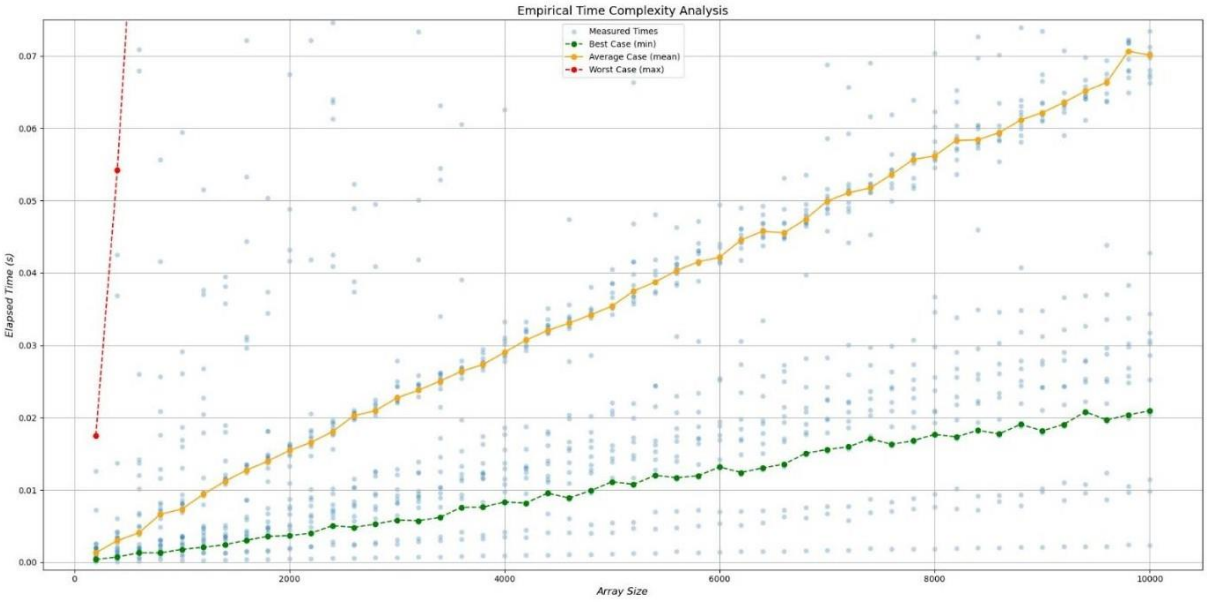
In this algorithm, at the end of the run the most repeated value is located at the root. Then, we check the root's frequency to see whether it is greater than n/2 (half of the array size). If it is, we return the value of the root, if not we return -1.

An added advantage of this algorithm is that, in most cases, it produces a balanced binary search tree (BST), which reduces the search time to O(log⁡n)O(\log n).

In this algorithm, the most cost appears when all the elements of the array are unique. Because insertion is too expensive. When some elements are the same and splay operations happen the time decreases.



Closer look:

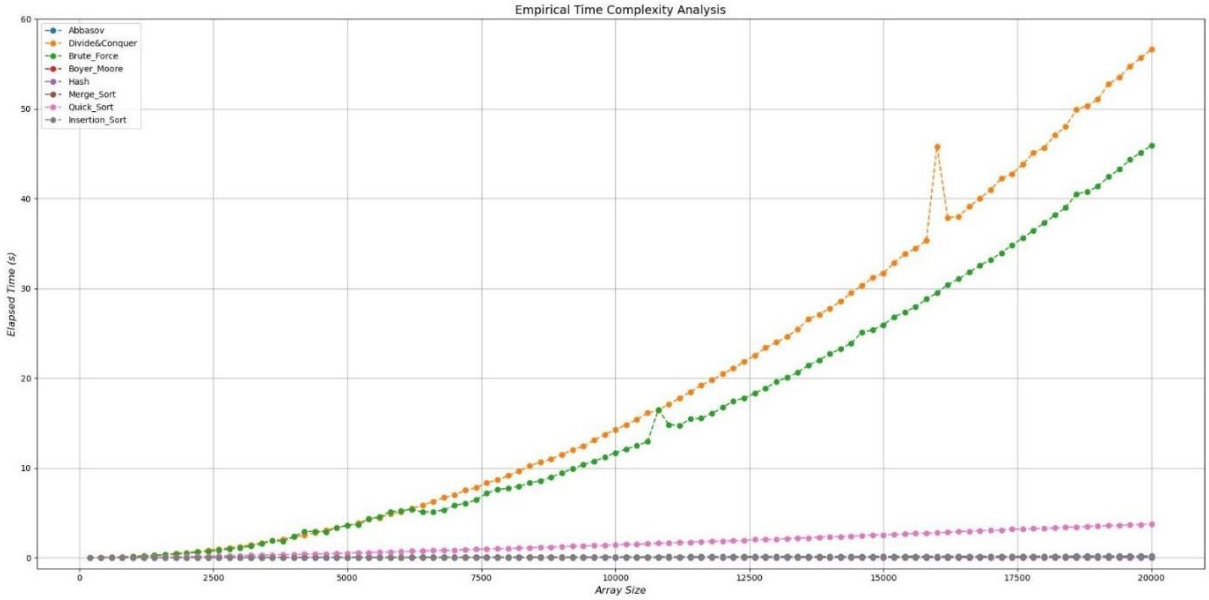


In the graphs, it seems that this algorithm very efficient for its best and worst case. But its worst case is very slow.

DIFFERENT ARRAY TYPES WITH DIFFERENT SIZES

In this part, we have some special input cases in different input files, for each type there are different sizes from 200 to 20000. Using these specific inputs, we compare algorithms to see how different their behaviors are from each other.

Shuffled array (no majority):



Worst case for brute force.

Analysis Of the Graph:

Brute force: For that input, brute force's performance is very slow because it is worst case for brute force.

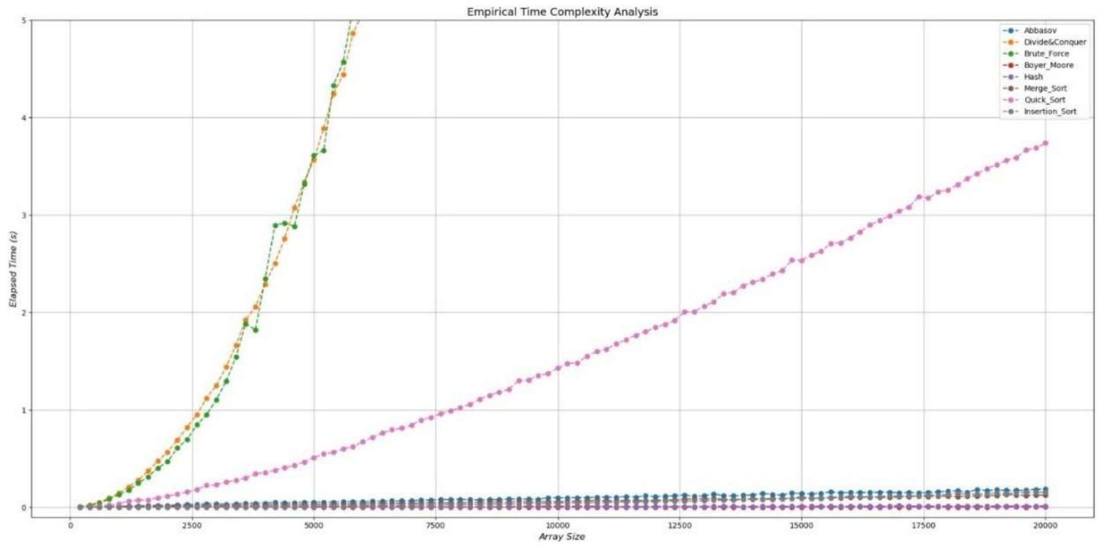
Boyer Moore: Since Boyer Moore is the best algorithm for finding majority, it is the fastest one for that input.

Hash: This algorithm is fastest algorithm after Boyer Moore for that input. If most of the time array's elements have different hash values, Hashing performs its operations fast.

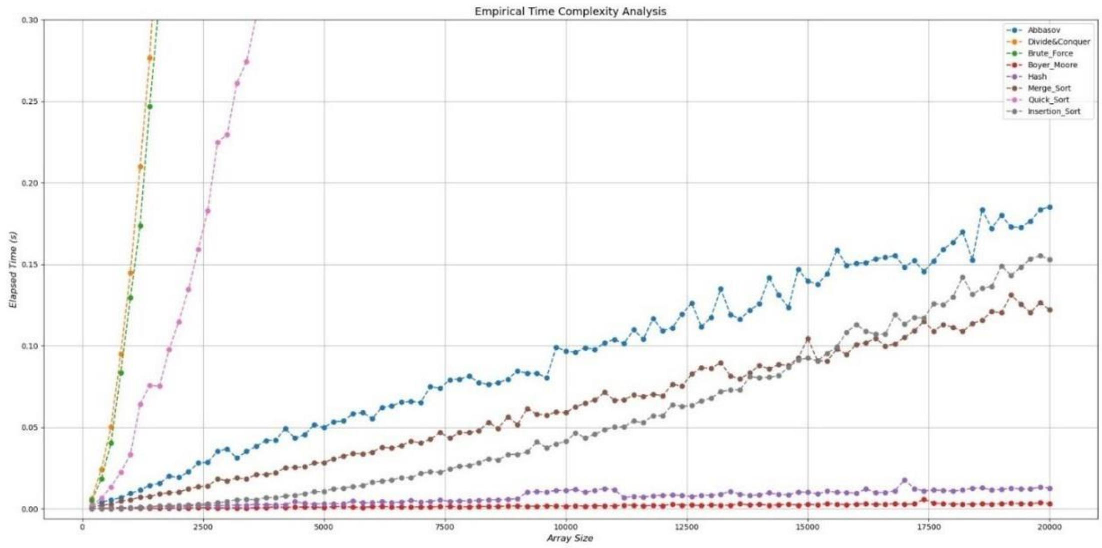
Divide and Conquer: That algorithm is the slowest one for this case because counting elements for most of the subarray when returned values are different can be very costly.

Quick Sort: For this input, quick sort performs partitioning very slowly. It is possible that selecting a pivot is not efficient for that input.

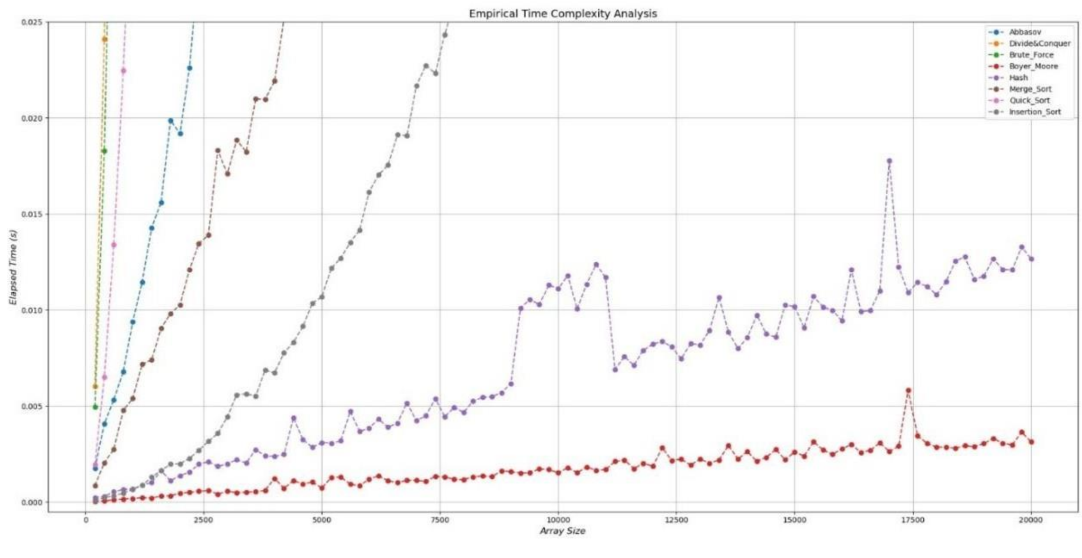
Closer looks:



Empirical Time Complexity Analysis
Abbasov
Divide&Conquer
Brute_Force
Boyer_Moore
Hash
Merge_Sort
Quick_Sort
Insertion_Sort
Elapsed Time (s)
Array Size

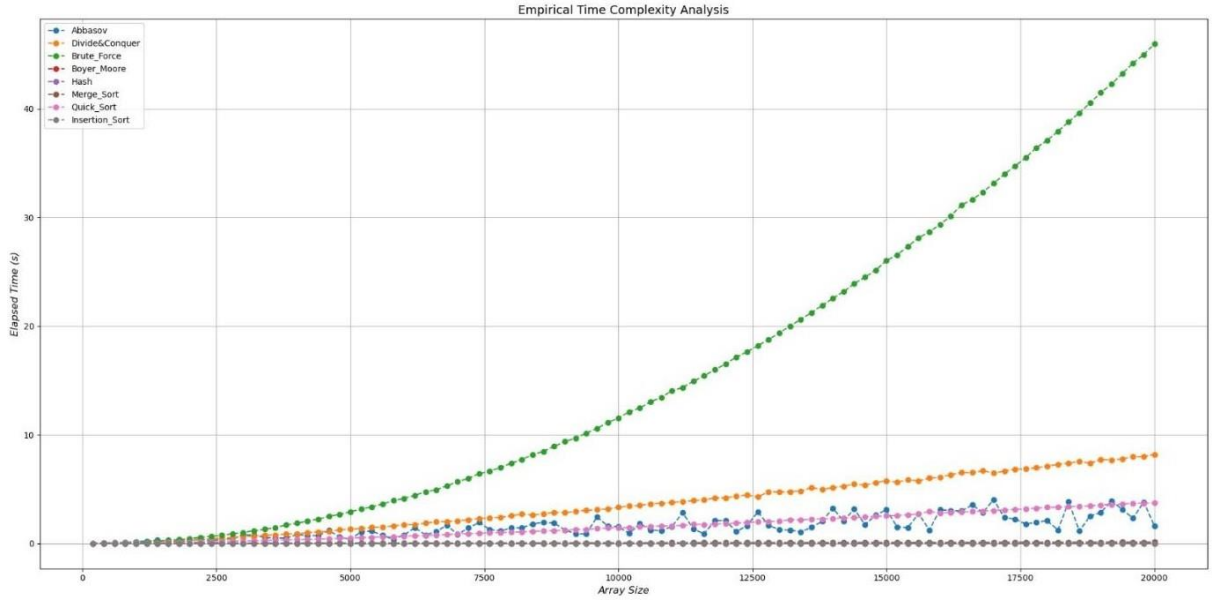


Empirical Time Complexity Analysis
Abbasov
Divide&Conquer
Brute_Force
Boyer_Moore
Hash
Merge_Sort
Quick_Sort
Insertion_Sort
Elapsed Time (s)
Array Size



Empirical Time Complexity Analysis
Abbasov
Divide&Conquer
Brute_Force
Boyer_Moore
Hash
Merge_Sort
Quick_Sort
Insertion_Sort
Elapsed Time (s)
Array Size

Sorted array (no majority):



Best case for Insertion sort, worst case for brute force and quick sort.

Analysis Of the Graph:

Brute force: For that input, brute force's performance is very slow because it is worst case for brute force.

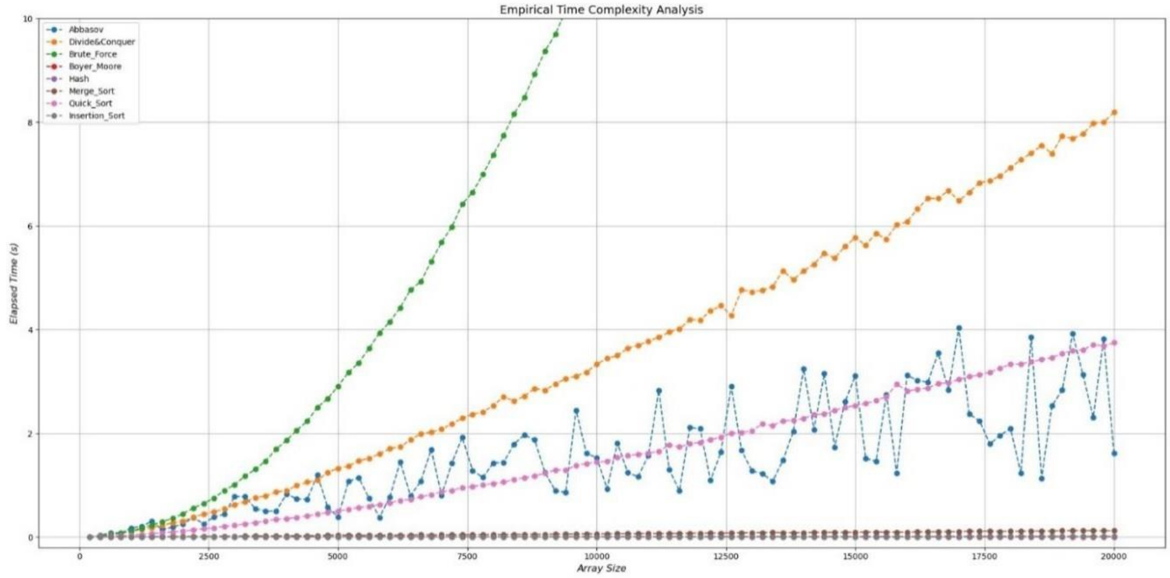
Boyer Moore: Since Boyer Moore is the best algorithm for finding majority, it is the fastest one for that input.

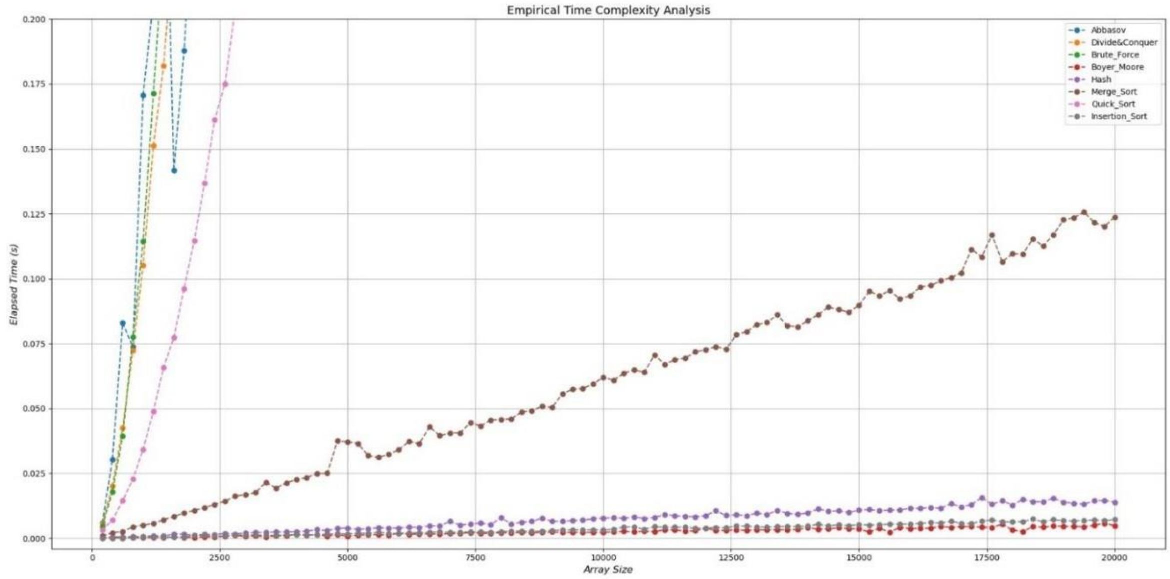
Insertion Sort: This algorithm is the fastest algorithm after Boyer Moore for that input. Because sorted arrays are best case for insertion sort. It takes O (n) time.

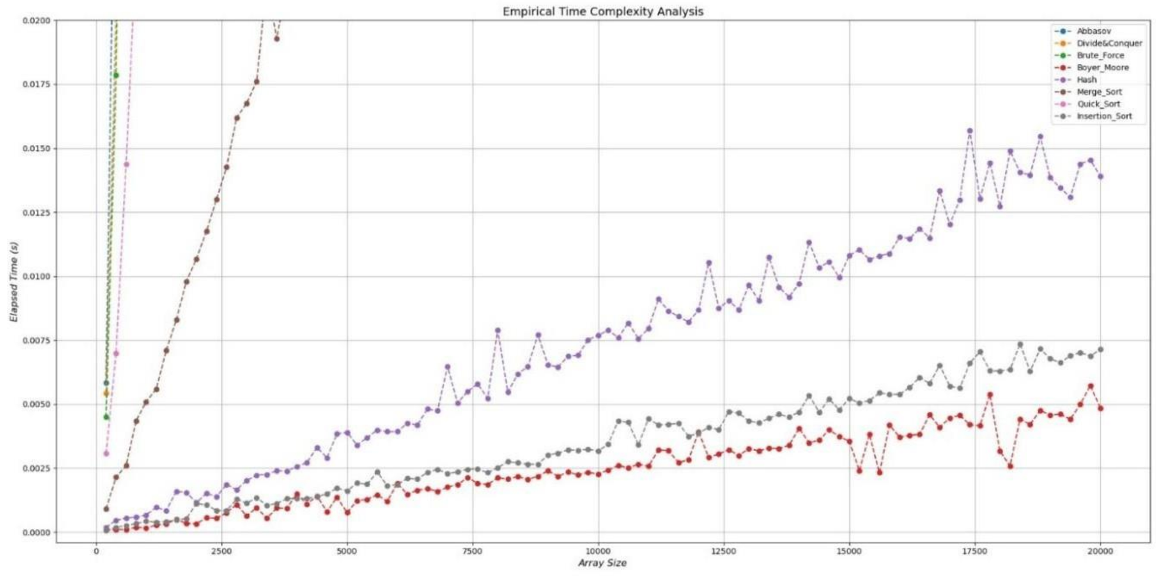
Hash: This algorithm is fastest algorithm after Boyer Moore and insertion sort for that input. If most of the time array's elements have different hash values, Hashing performs its operations fast.

Divide and Conquer: That algorithm is the slowest one after brute force for this case because of counting elements for most of the subarray when returned values are different can be very costly.

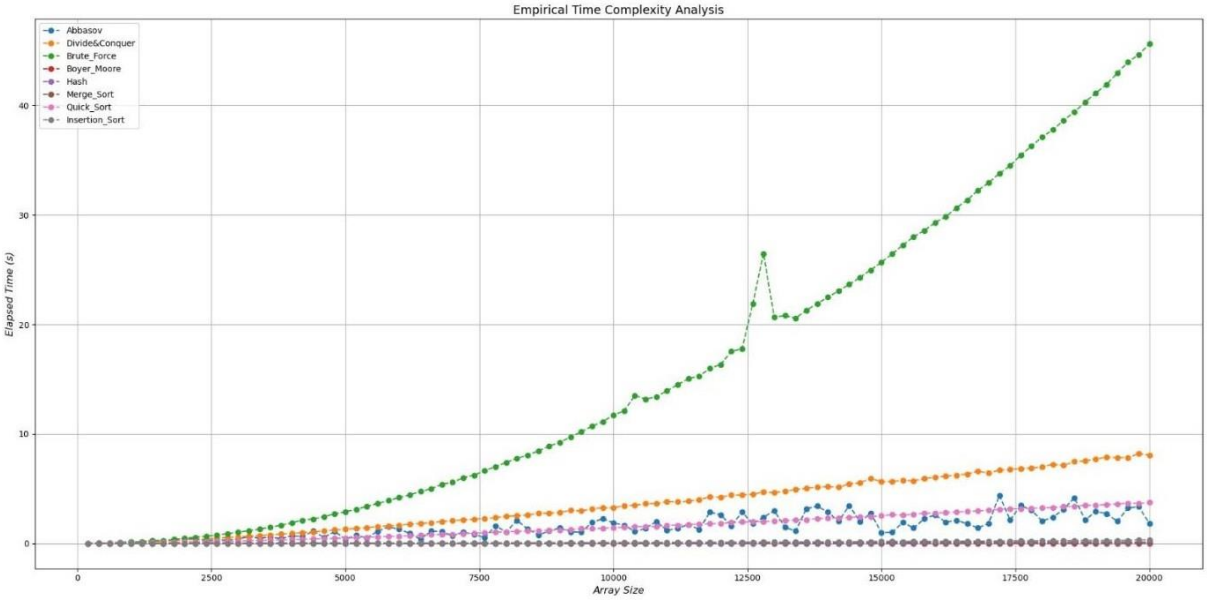
Closer looks:


Empirical Time Complexity Analysis
Abbasov
Divide&Conquer
Brute_Force
Boyer_Moore
Hash
Merge_Sort
Quick_Sort
Insertion_Sort
Elapsed Time (s)
Array Size


Empirical Time Complexity Analysis
Abbasov
Divide&Conquer
Brute_Force
Boyer_Moore
Hash
Merge_Sort
Quick_Sort
Insertion_Sort
Elapsed Time (s)
Array Size


Empirical Time Complexity Analysis
Abbasov
Divide&Conquer
Brute_Force
Boyer_Moore
Hash
Merge_Sort
Quick_Sort
Insertion_Sort
Elapsed Time (s)
Array Size

Reverse sorted (no majority):



Worst case for insertion sort and brute force.

Analysis Of the Graph:

Brute force: This algorithm is the slowest algorithm for this input.

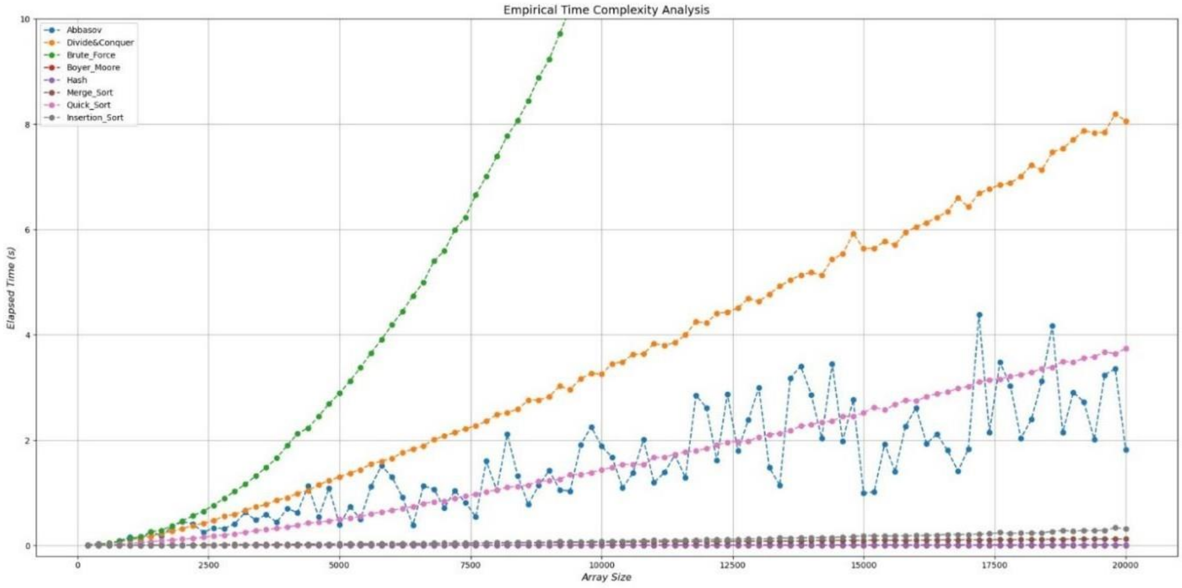
Boyer Moore: Since Boyer Moore is the best algorithm for finding majority, it is the fastest algorithm for that input.

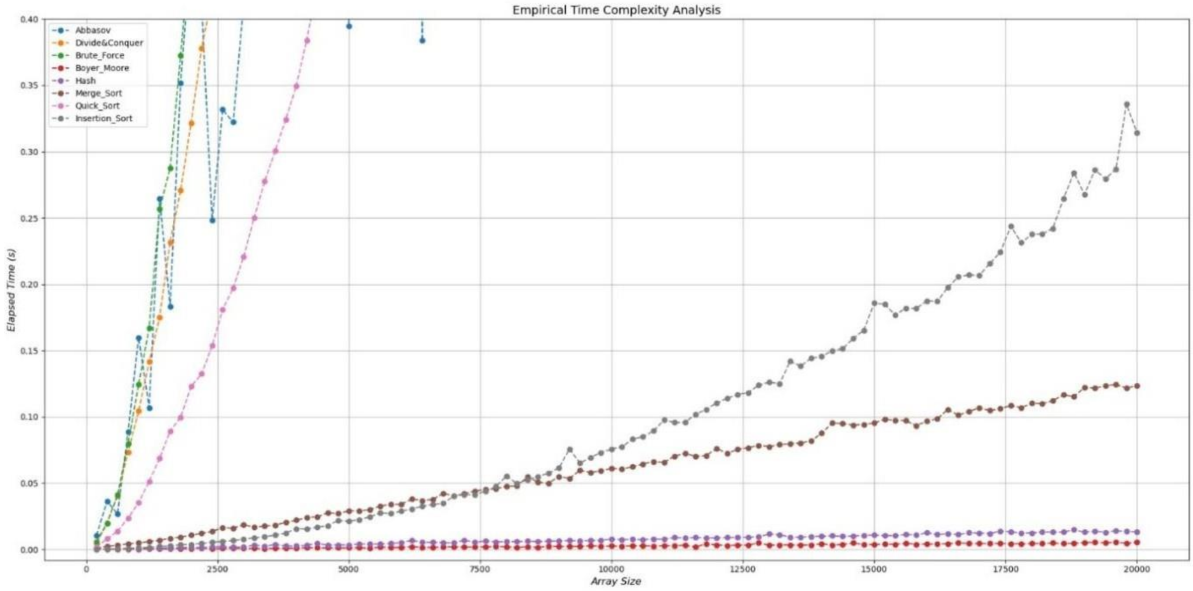
Insertion Sort: Due to this input is the worst case for insertion sort, this algorithm runs very fast.

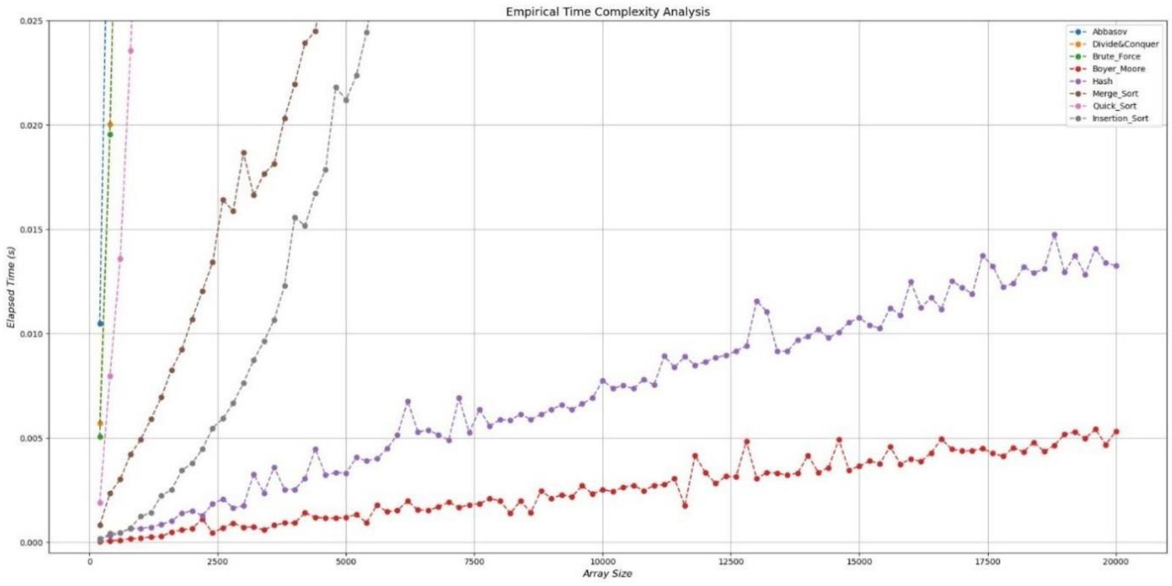
Hash: This algorithm is the fastest algorithm after Boyer Moore. If most of the time array's elements have different hash values, Hashing performs its operations fast.

Divide and Conquer: That algorithm is the slowest one after brute force for this case because of counting elements for most of the subarray when returned values are different can be very costly.

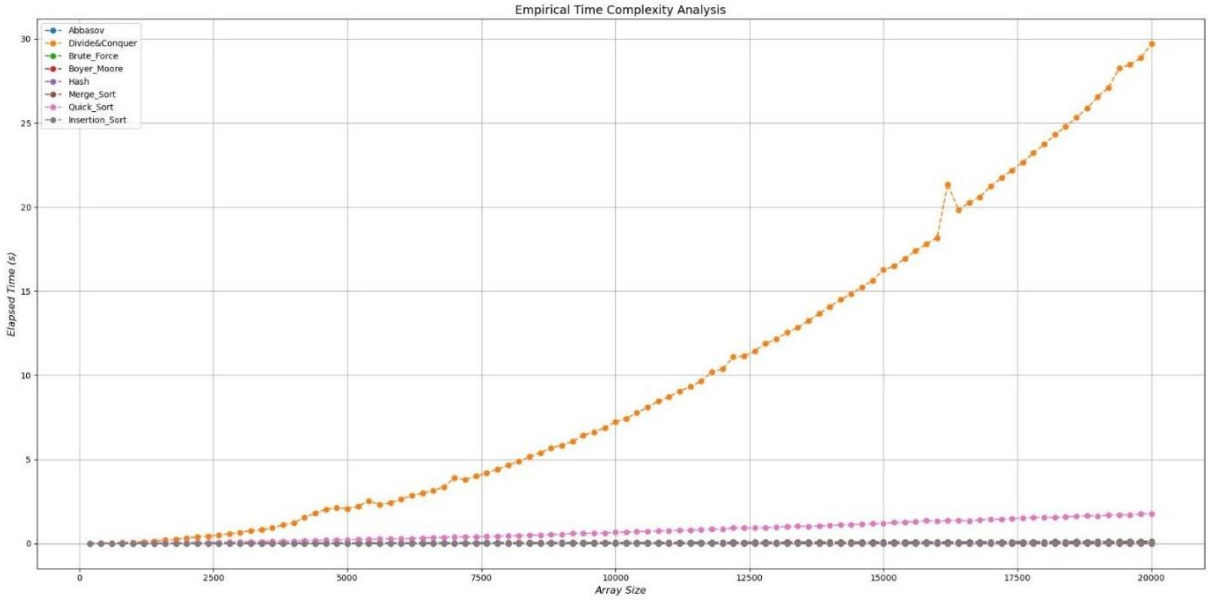
Closer looks:


Empirical Time Complexity Analysis
Abbasov
Divide&Conquer
Brute_Force
Boyer_Moore
Hash
Merge_Sort
Quick_Sort
Insertion_Sort
Elapsed Time (s)
Array Size


Empirical Time Complexity Analysis
Abbasov
Divide&Conquer
Brute_Force
Boyer_Moore
Hash
Merge_Sort
Quick_Sort
Insertion_Sort
Elapsed Time (s)
Array Size


Empirical Time Complexity Analysis
Abbasov
Divide&Conquer
Brute_Force
Boyer_Moore
Hash
Merge_Sort
Quick_Sort
Insertion_Sort
Elapsed Time (s)
Array Size

The first half of the array is the majority:



Best case for brute force, average case for other ones.

Analysis Of the Graph:

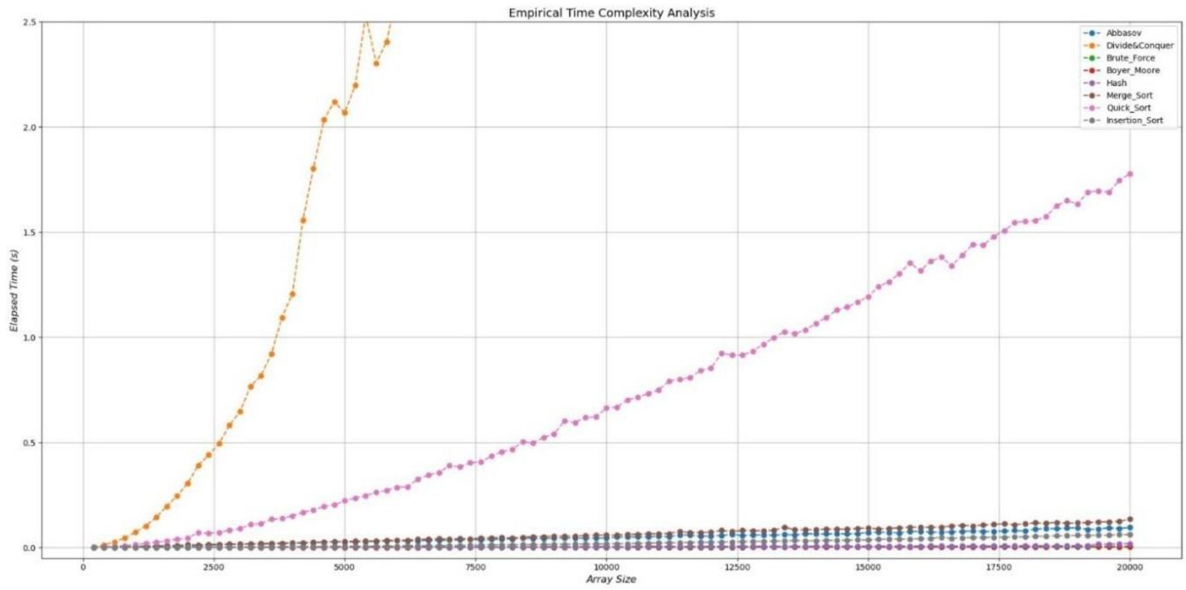
Brute force: Despite brute force algorithm is one of the slowest algorithm, in that input it is the fastest one because that input is best case for brute force. Its time complexity is O(n) at that case and this algorithm makes approximately n/2 comparisons to find majority if first n/2 + 1 element is same value.

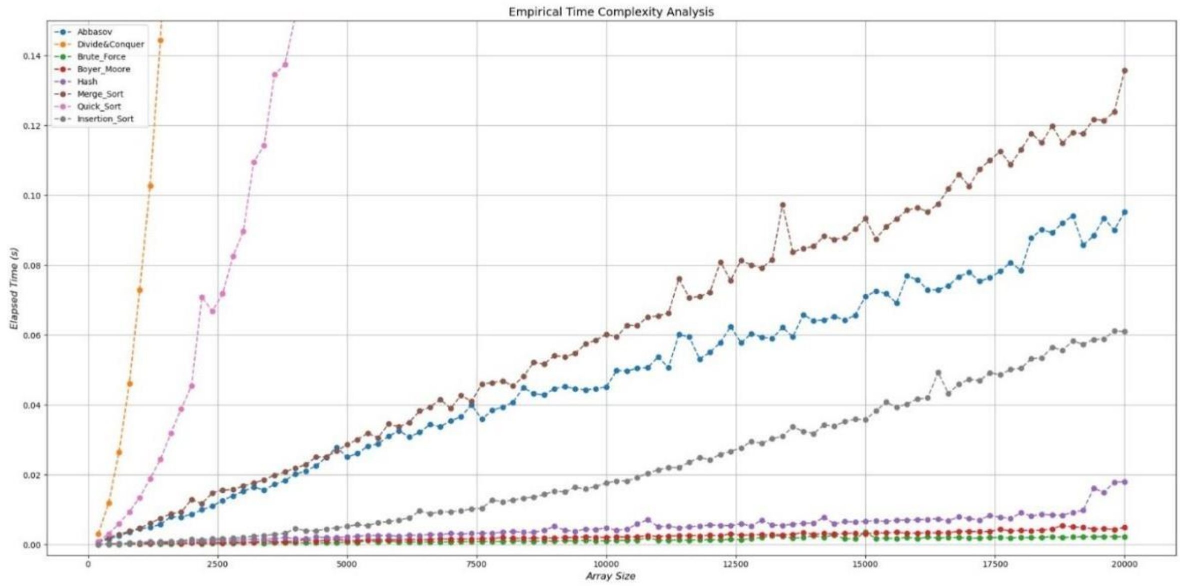
Boyer Moore: Boyer Moore algorithm is very efficient for that input because it finds majority as candidate n comparison and finds that this candidate is majority at n/2 + 1 comparison.

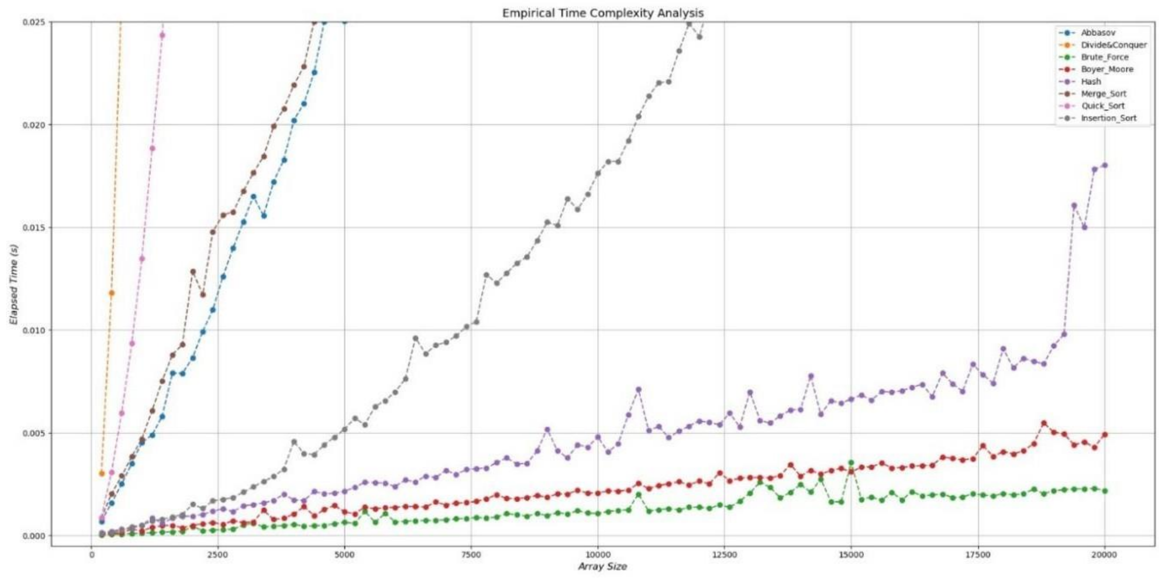
Hash: This algorithm is efficient in this case. If most of the time array's elements have different hash values, Hashing performs its operations fast. The reason of oscillating is finding prime number for hash size.

Divide and Conquer: This algorithm is the slowest algorithm for this input.

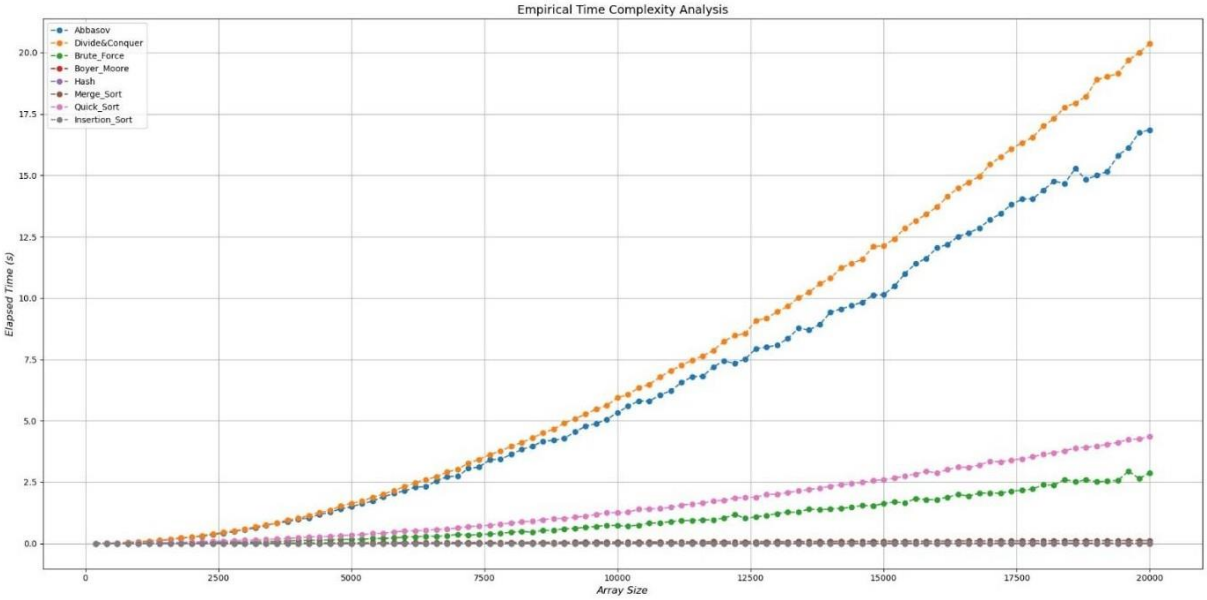
Closer looks:


Empirical Time Complexity Analysis
Abbasov
Divide&Conquer
Brute_Force
Boyer_Moore
Hash
Merge_Sort
Quick_Sort
Insertion_Sort
Elapsed Time (s)
Array Size


Empirical Time Complexity Analysis
Abbasov
Divide&Conquer
Brute_Force
Boyer_Moore
Hash
Merge_Sort
Quick_Sort
Insertion_Sort
Elapsed Time (s)
Array Size


Empirical Time Complexity Analysis
Abbasov
Divide&Conquer
Brute_Force
Boyer_Moore
Hash
Merge_Sort
Quick_Sort
Insertion_Sort
Elapsed Time (s)
Array Size

Sorted array with majority:



Best case for insertion sort. Worst case for quick sort.

Analysis Of the Graph:

Brute force: This algorithm is still very slow but not the slowest one because this input consists of majority, so this algorithm does not look at every pair combination.

Boyer Moore: Since Boyer Moore is the best algorithm for finding majority, it is the fastest algorithm for that input.

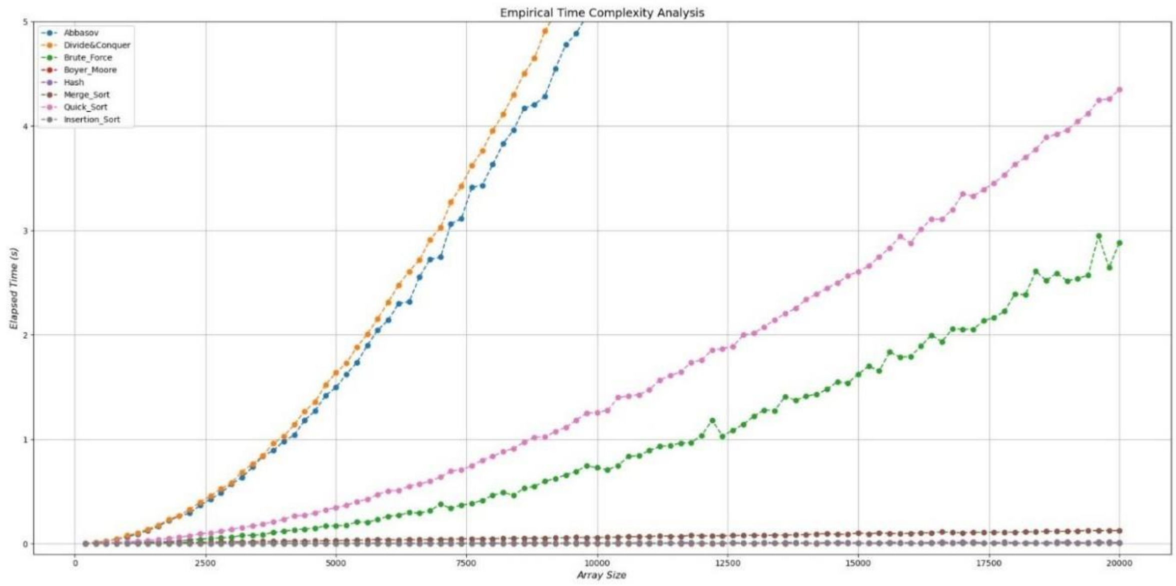
Insertion Sort: For this input, insertion sort is very effective because array is sorted. Insertion sort's time complexity is O(n) for this input.

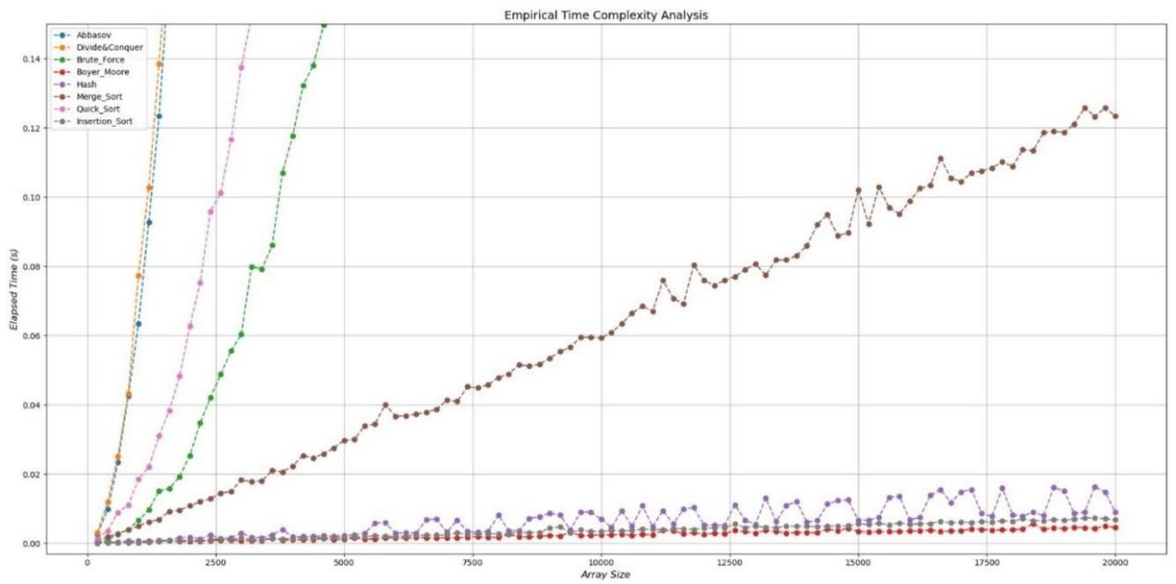
Hash: This algorithm is efficient in this case. If most of the time array's elements have different hash values, Hashing performs its operations fast. The reason for oscillating is finding prime number for hash size.

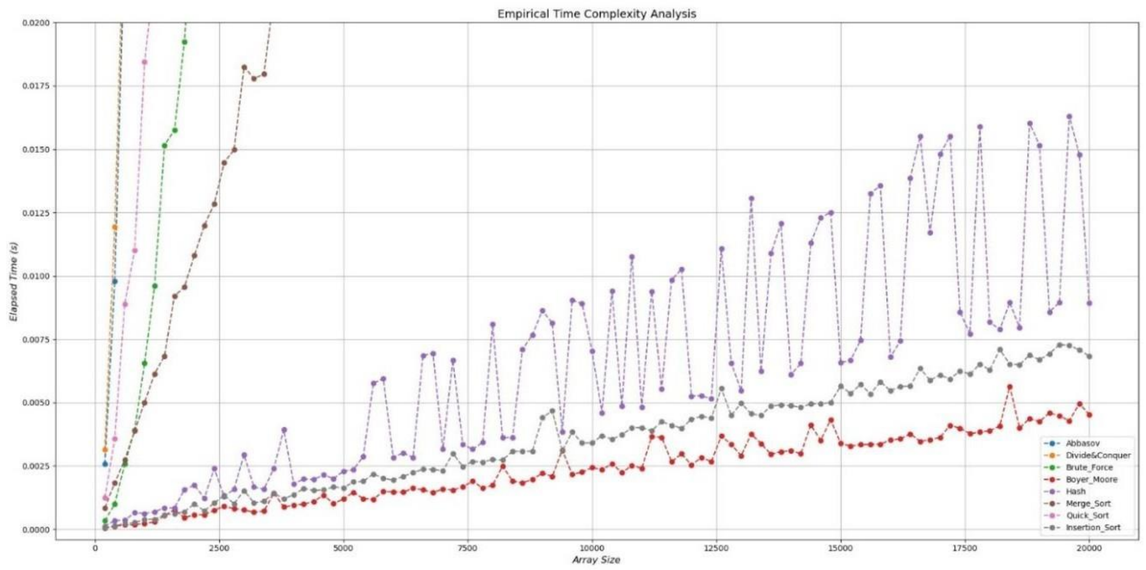
Abbasov's Algorithm: This algorithm is not efficient for sorted inputs.

Divide and Conquer: This algorithm is the slowest algorithm for this input.

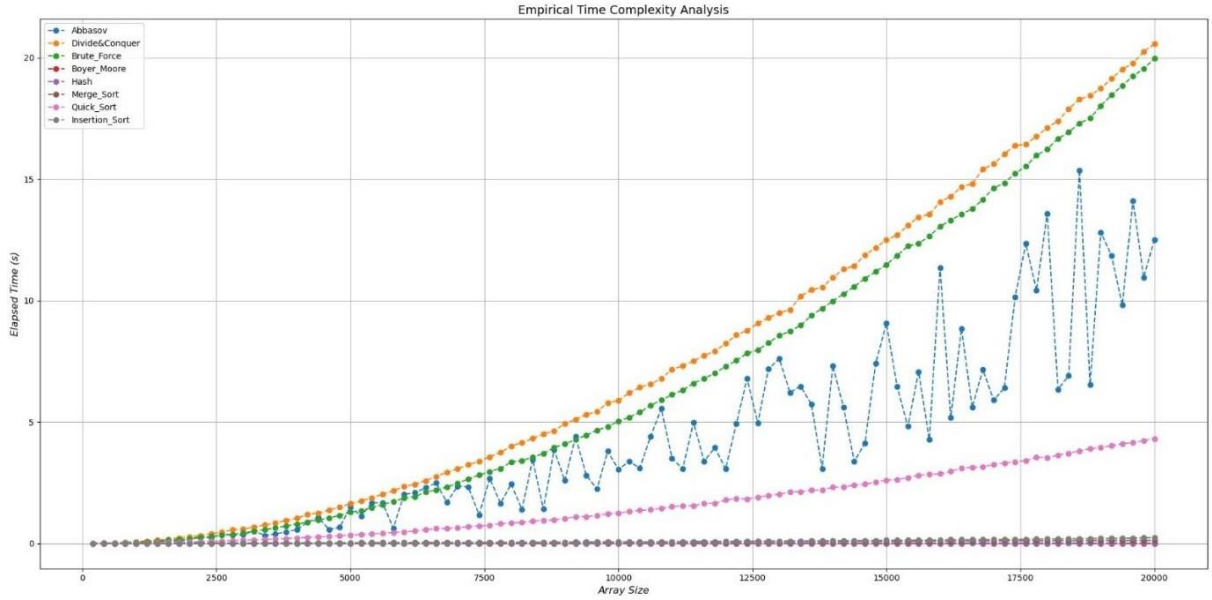
Closer looks:


Empirical Time Complexity Analysis
Abbasov
Divide&Conquer
Brute_Force
Boyer_Moore
Hash
Merge_Sort
Quick_Sort
Insertion_Sort
Elapsed Time (s)
Array Size


Empirical Time Complexity Analysis
Abbasov
Divide&Conquer
Brute_Force
Boyer_Moore
Hash
Merge_Sort
Quick_Sort
Insertion_Sort
Elapsed Time (s)
Array Size


Empirical Time Complexity Analysis
Abbasov
Divide&Conquer
Brute_Force
Boyer_Moore
Hash
Merge_Sort
Quick_Sort
Insertion_Sort
Elapsed Time (s)
Array Size

Reverse sorted array with majority:



Worst case for insertion sort.

Analysis Of the Graph:

Brute force: This algorithm is still very slow but not the slowest one because this input consists of majority, so this algorithm does not look at every pair combination.

Boyer Moore: Since Boyer Moore is the best algorithm for finding majority, it is the fastest algorithm for that input.

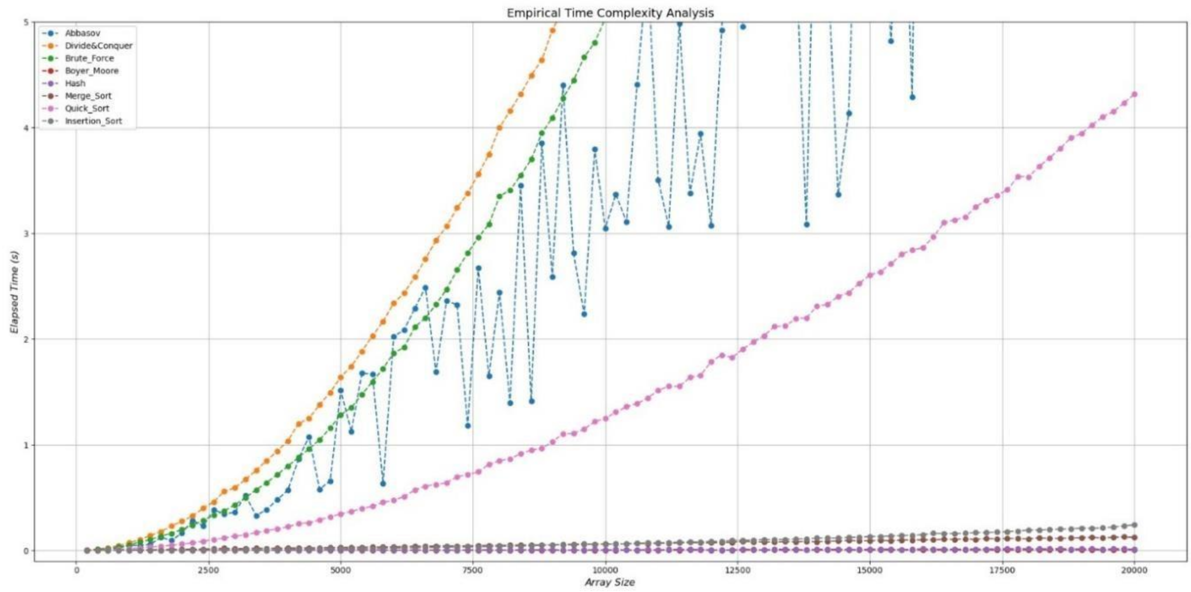
Insertion Sort: For this input, insertion sort is a slow algorithm because reverse ordered arrays are the worst case for Insertion sort. It takes $O(n^2)$ time.

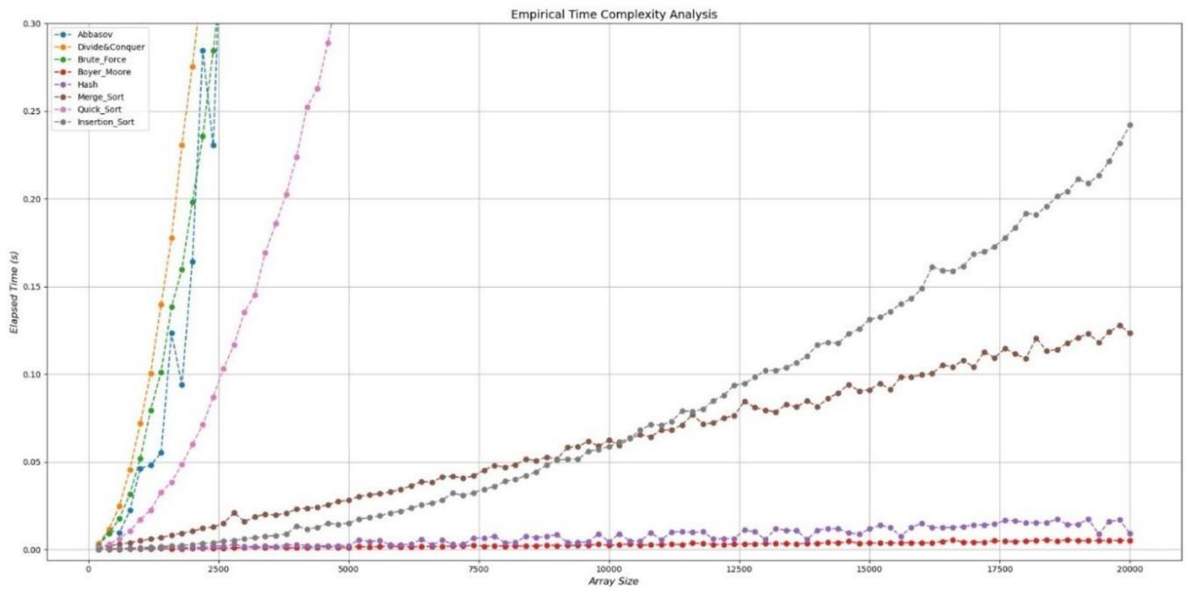
Hash: This algorithm is efficient in this case. If most of the time array's elements have different hash values, Hashing performs its operations fast. The reason for oscillation is finding the prime number for hash size.

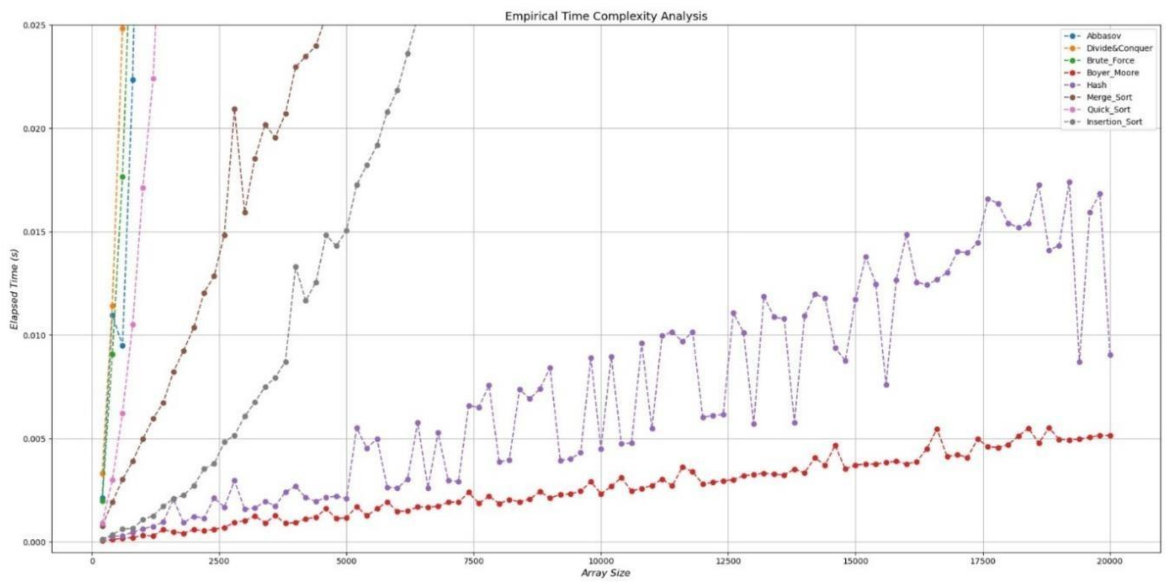
Abbasov's Algorithm: This algorithm is not efficient for sorted inputs.

Divide and Conquer: This algorithm is the slowest algorithm for this input.

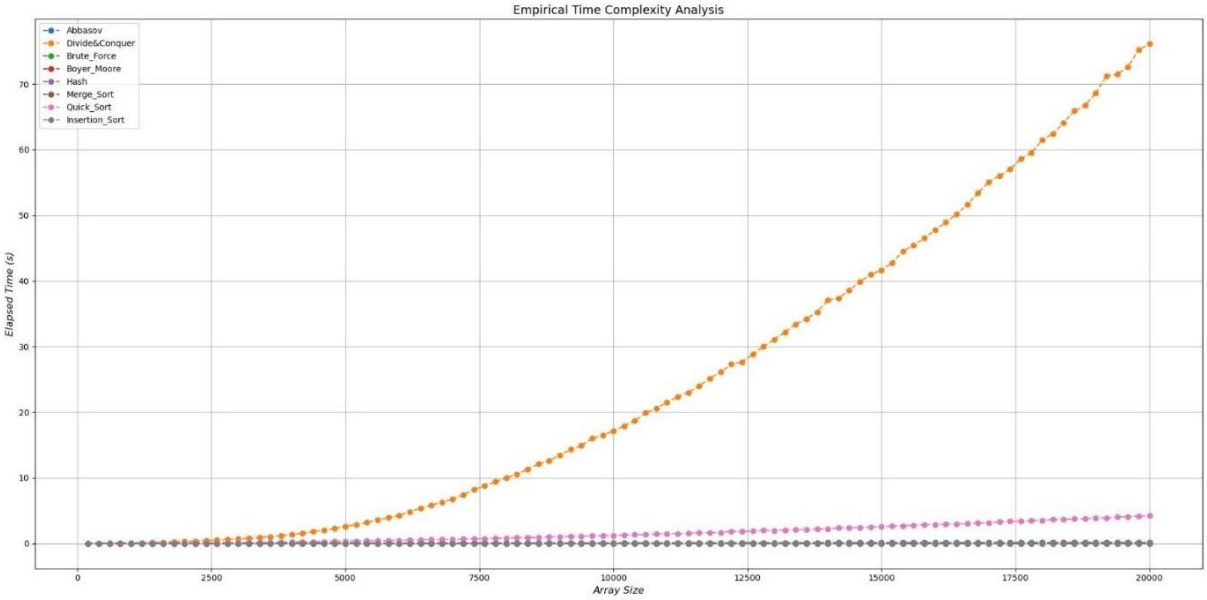
Closer looks:


Empirical Time Complexity Analysis
Abbasov
Divide&Conquer
Brute_Force
Boyer_Moore
Hash
Merge_Sort
Quick_Sort
Insertion_Sort
Elapsed Time (s)
Array Size


Empirical Time Complexity Analysis
Abbasov
Divide&Conquer
Brute_Force
Boyer_Moore
Hash
Merge_Sort
Quick_Sort
Insertion_Sort
Elapsed Time (s)
Array Size


Empirical Time Complexity Analysis
Abbasov
Divide&Conquer
Brute_Force
Boyer_Moore
Hash
Merge_Sort
Quick_Sort
Insertion_Sort
Elapsed Time (s)
Array Size

Shuffled array with majority:



Analysis Of the Graph:

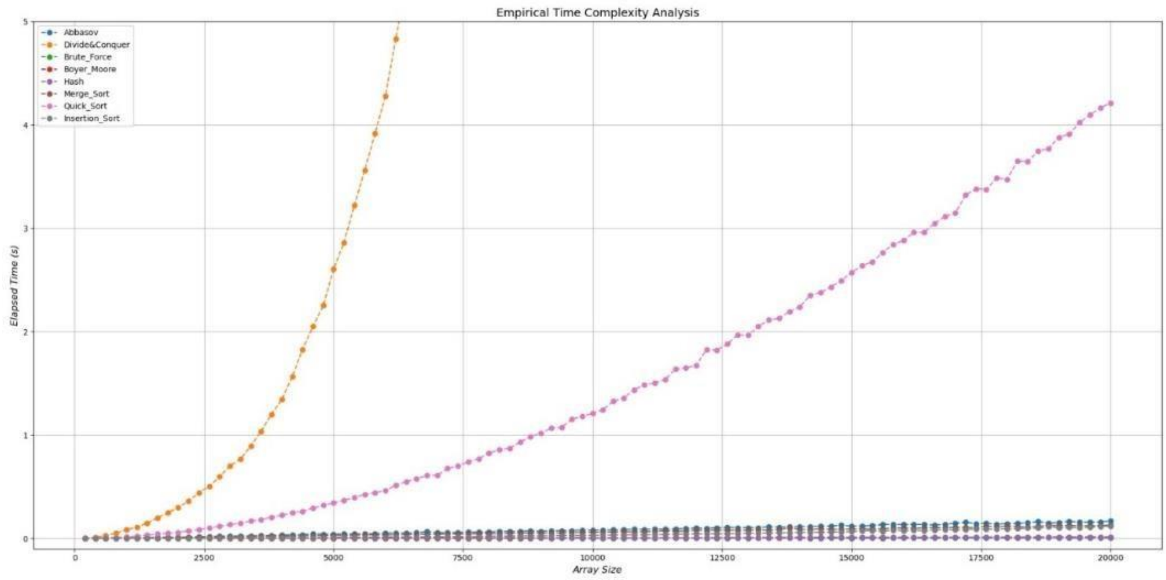
Brute force: This algorithm's performance is very fast but not stable because this input is randomly distributed and has majority element. If majority elements are available at first indexes, the algorithm performs fast. If not, algorithms become slower.

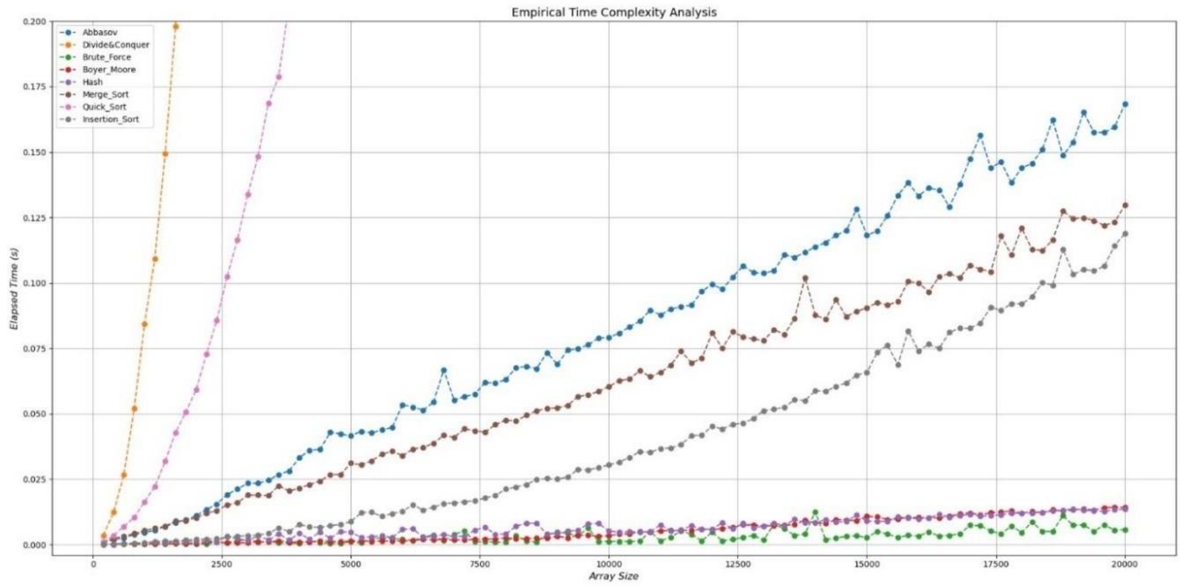
Boyer Moore: Since Boyer Moore is the best algorithm for finding majority, it is the fastest algorithm with brute force for that input.

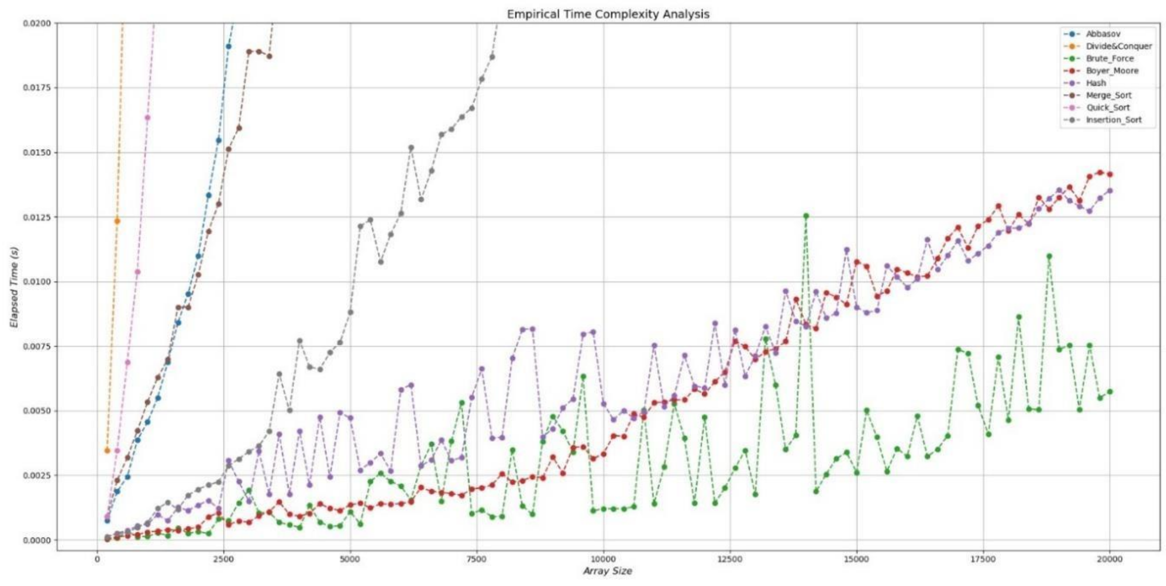
Insertion Sort: This input is an average case for insertion sort, so it takes $O(n^2)$ time. It is not so effective.

Divide and Conquer: This algorithm is the slowest algorithm for this input

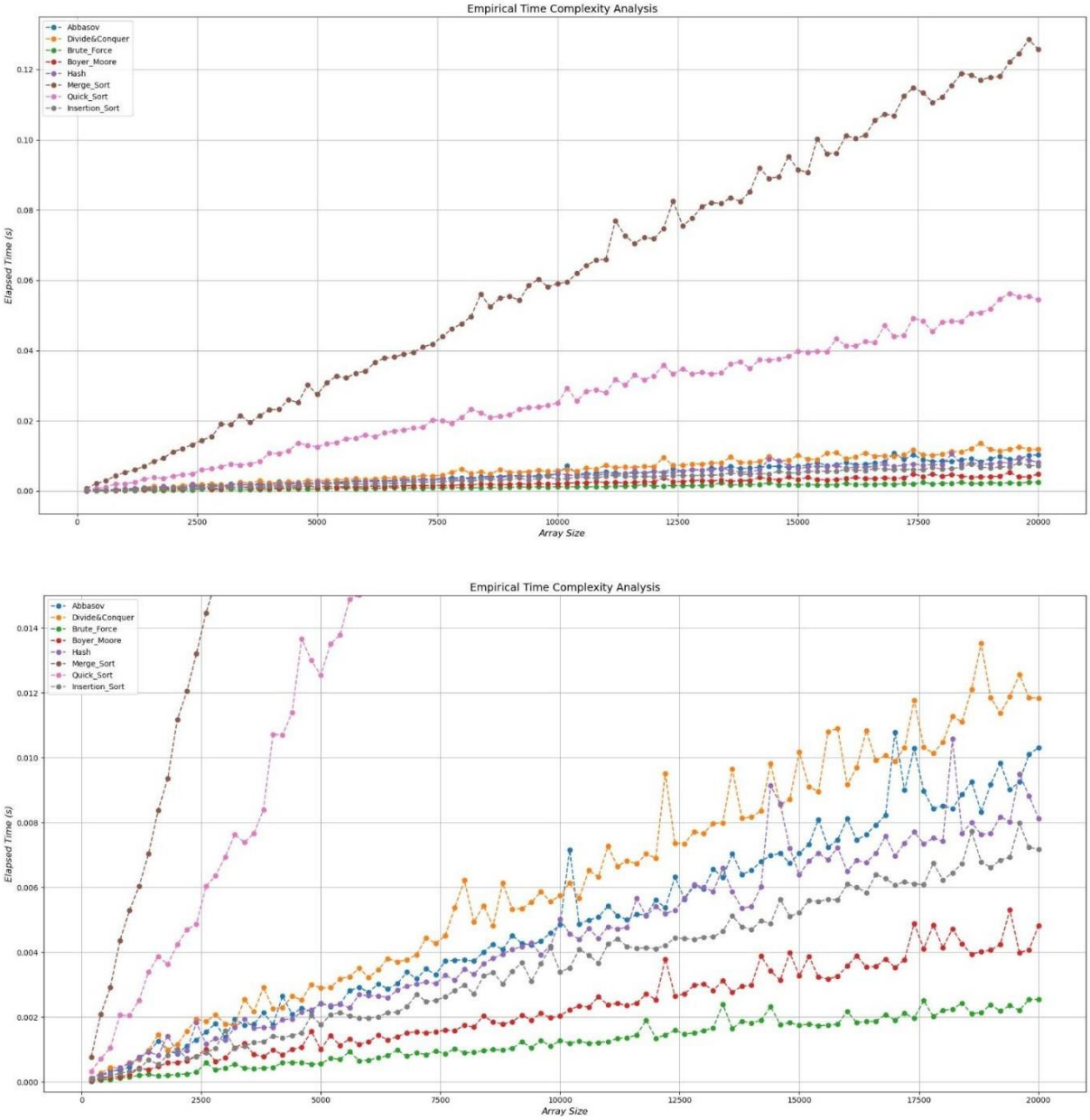
Closer looks:


Empirical Time Complexity Analysis
Abbasov
Divide&Conquer
Brute_Force
Boyer_Moore
Hash
Merge_Sort
Quick_Sort
Insertion_Sort
Elapsed Time (s)
Array Size


Empirical Time Complexity Analysis
Abbasov
Divide&Conquer
Brute_Force
Boyer_Moore
Hash
Merge_Sort
Quick_Sort
Insertion_Sort
Elapsed Time (s)
Array Size


Empirical Time Complexity Analysis
Abbasov
Divide&Conquer
Brute_Force
Boyer_Moore
Hash
Merge_Sort
Quick_Sort
Insertion_Sort
Elapsed Time (s)
Array Size

All elements are the majority:



Best case for Abbasov's algorithm, divide and conquer, brute force, hash, and insertion sort.

Analysis Of the Graph:

Brute force: Despite the brute force algorithm is one of the slowest algorithms, in that input it is the fastest one because that input is best case for brute force. Its time complexity is O(n) at that case and this algorithm makes approximately n/2 comparisons to find majority if all elements are the same.

Boyer Moore: Boyer Moore algorithm is very efficient for that input because it finds majority as candidate, then it do n comparisons and finds that this candidate is majority at n/2 + 1 comparison.

Insertion Sort: This input is the best case for Insertion sort because it is like already sorted array so Insertion Sort is very efficient. It solves the problem O(n) time.

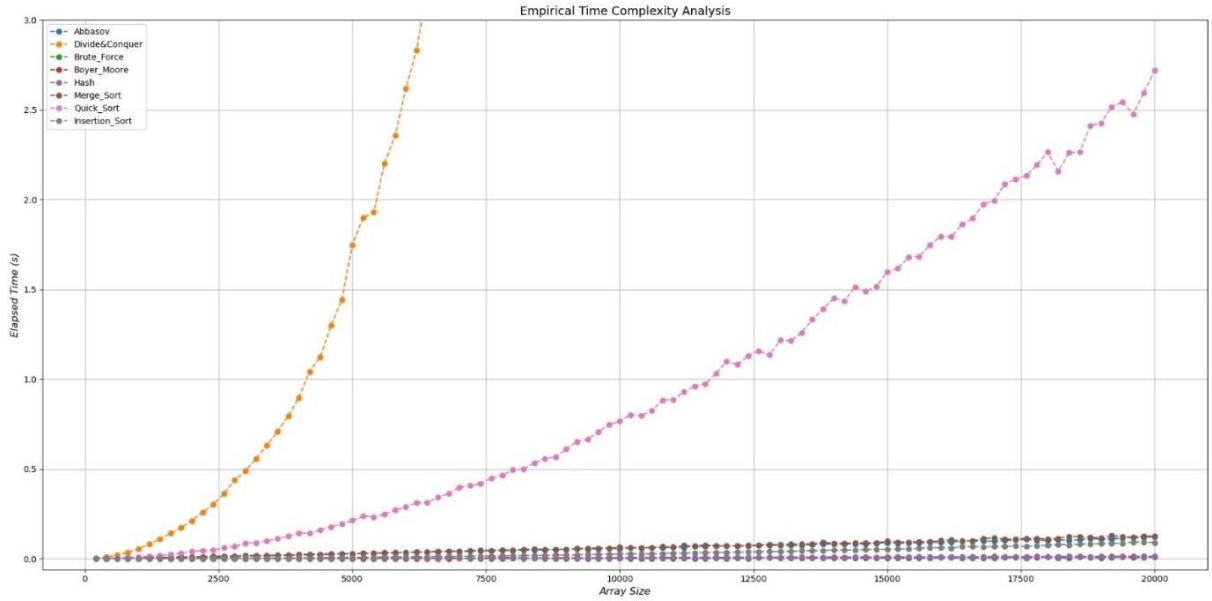
Hash: Because every element is the same, there is no collision probability for this array, so it is best case for hash algorithm. There may be some oscillations because of finding prime number but in total, hash algorithm solves this problem with O(n) time for the given input.

Abbasov's Algorithm: It is the best case for Abbasov's algorithm because there is no splay and insertions are made to root, so they are not costly. The time complexity is O(n).

Quick Sort: For this input, quick sort performs partitioning very slowly, so quick sort is not effective for this input. It takes O(nlogn) time.

Merge Sort: Merge sort algorithm is slowest algorithm at this input because it did not check the values before sorting. In spite of every element is same, it divides and merging so it became slower than other algorithms. It solves this problem O(nlogn) time.

65% of the array is the majority:



Analysis Of the Graph:

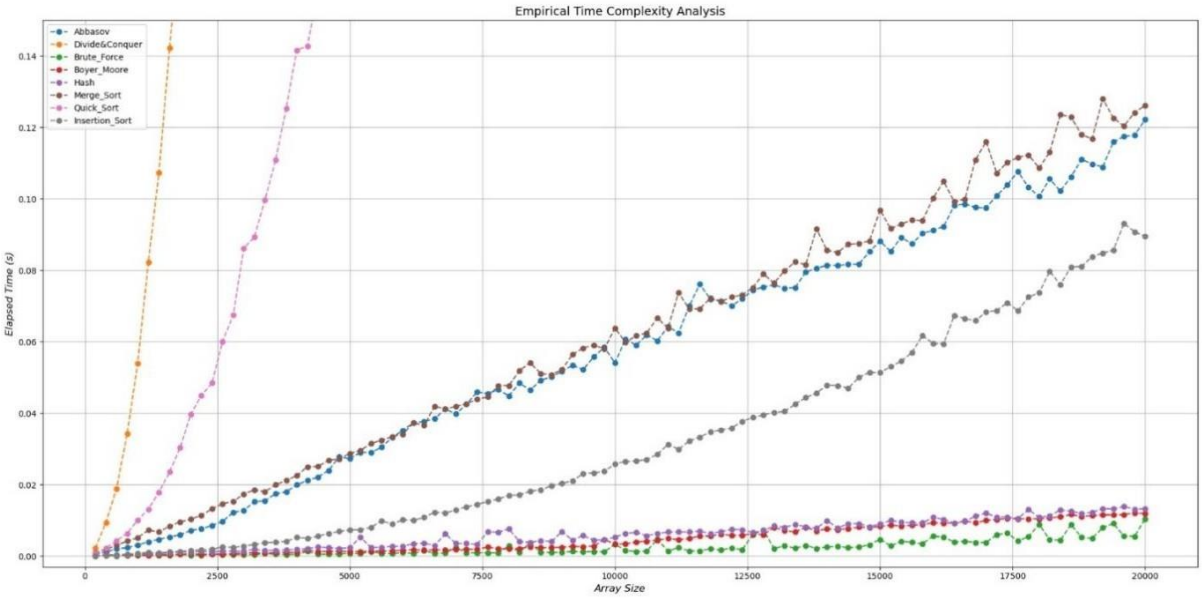
Brute force: This algorithm's performance is very fast but not stable because this input is randomly distributed and it has majority element. If majority elements are available at first indexes, the algorithm performs fast. If not, the algorithm becomes slower.

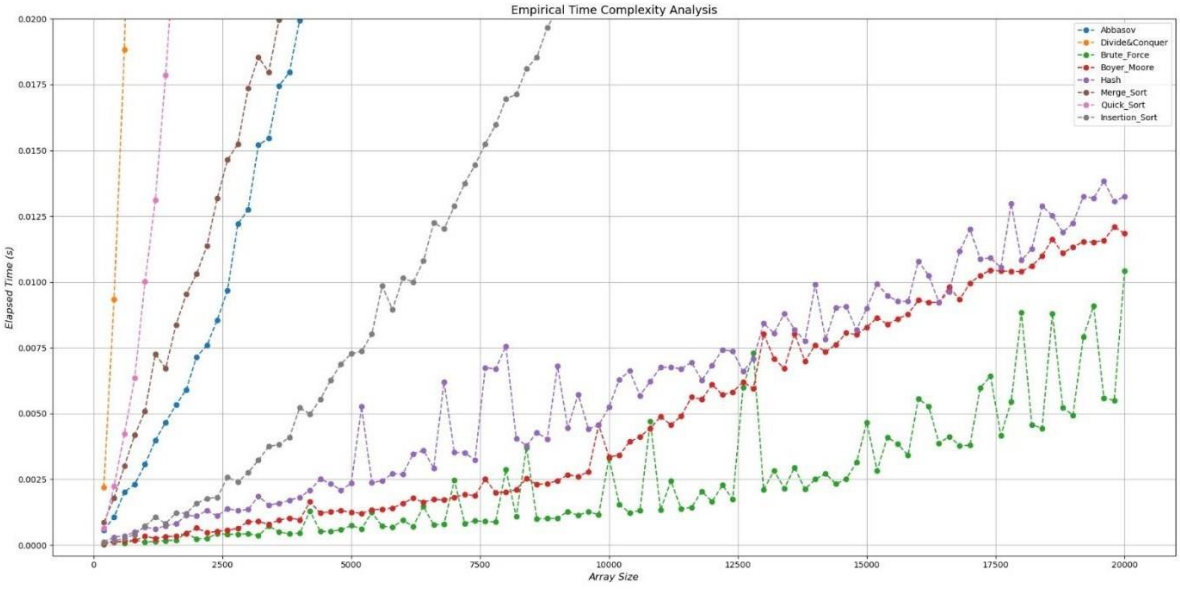
Boyer Moore: Since Boyer Moore is the best algorithm for finding majority, it is the fastest algorithm with brute force for that input

Insertion Sort: This input is an average case for insertion sort, so it takes $O(n^2)$ time. It is not so effective.

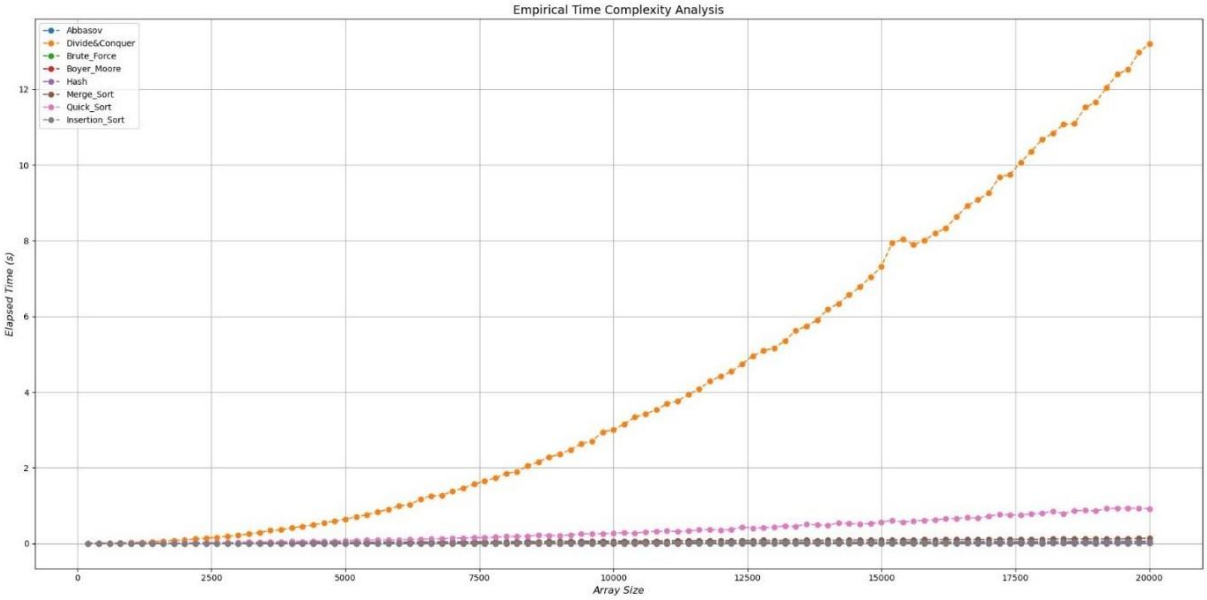
Divide and Conquer: This algorithm is the slowest algorithm for this input.

Closer looks:





85% of the array is the majority:



Analyze of graph:

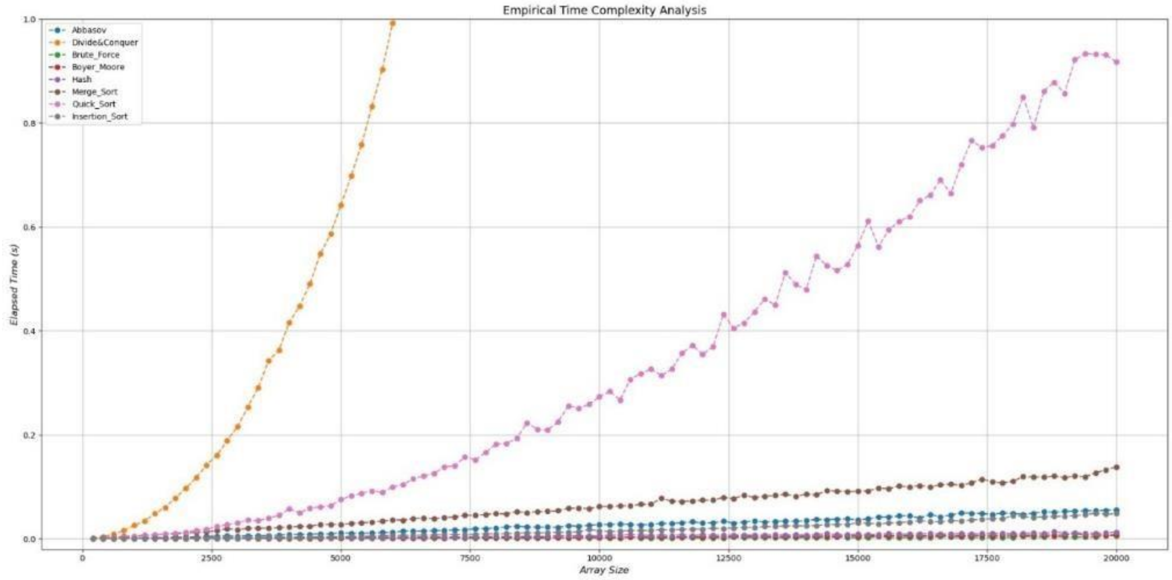
Brute force: This algorithm's performance is very fast but not stable because this input is randomly distributed and it has majority element. If majority elements are available at first indexes, the algorithm performs fast. If not, the algorithm becomes slower.

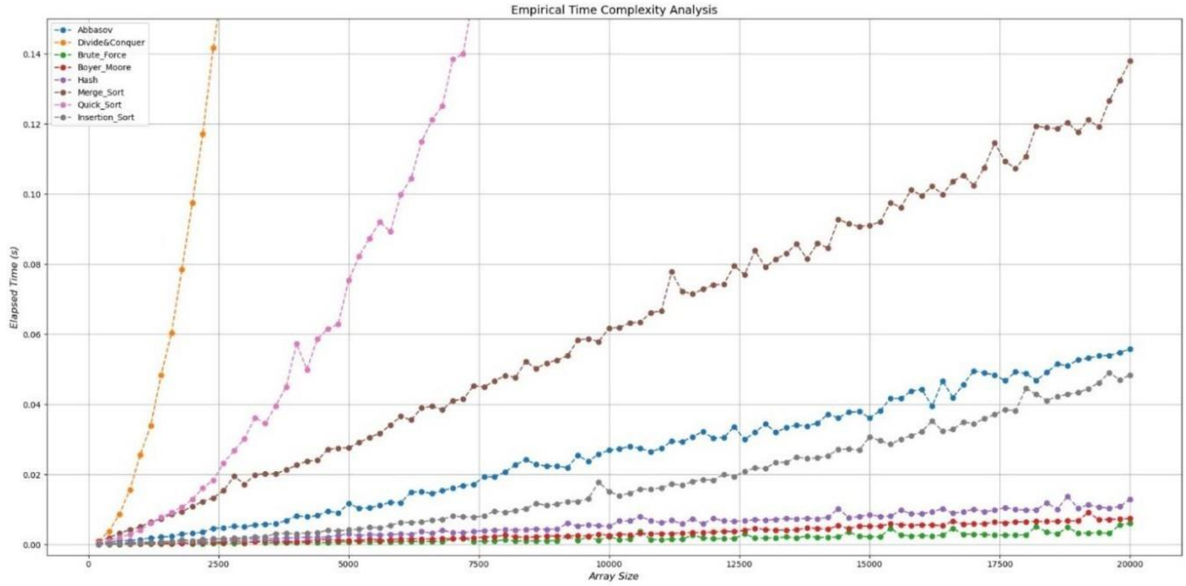
Boyer Moore: Since Boyer Moore is the best algorithm for finding majority, it is the fastest algorithm with brute force for that input

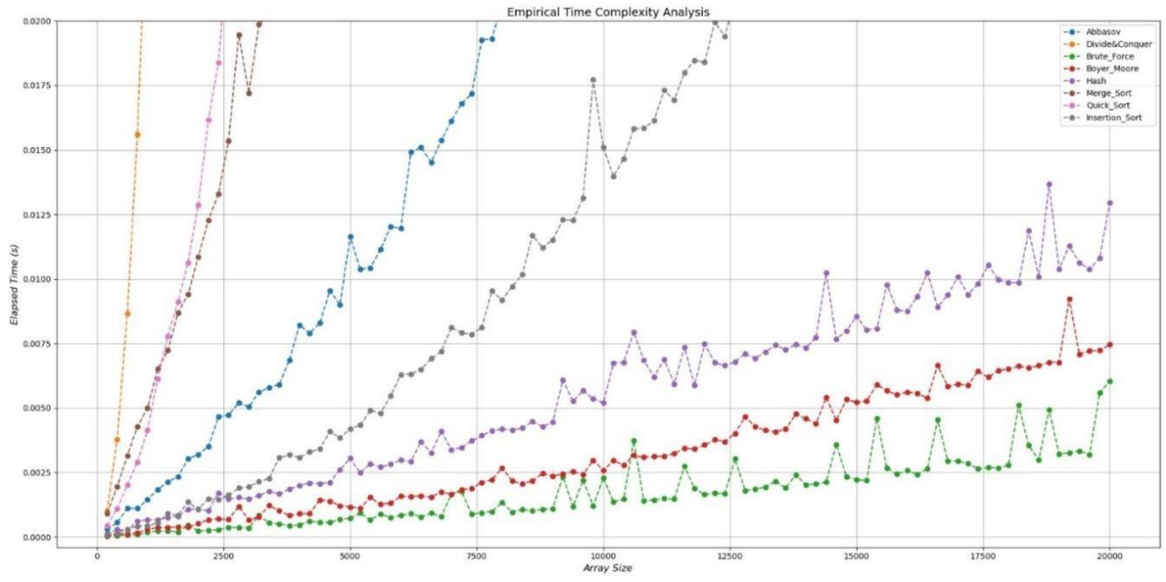
Insertion Sort: This input is an average case for insertion sort, so it takes $O(n^2)$ time. It is not so effective.

Divide and Conquer: This algorithm is the slowest algorithm for this input.

Closer looks:


Empirical Time Complexity Analysis
Abbasov
Divide&Conquer
Brute_Force
Boyer_Moore
Hash
Merge_Sort
Quick_Sort
Insertion_Sort
Elapsed Time (s)
Array Size


Empirical Time Complexity Analysis
Abbasov
Divide&Conquer
Brute_Force
Boyer_Moore
Hash
Merge_Sort
Quick_Sort
Insertion_Sort
Elapsed Time (s)
Array Size


Empirical Time Complexity Analysis
Abbasov
Divide&Conquer
Brute_Force
Boyer_Moore
Hash
Merge_Sort
Quick_Sort
Insertion_Sort
Elapsed Time (s)
Array Size

RESULTS

This was an experiment about comparing different algorithms, finding their best worst and average cases. Some algorithms have very huge differences at their best and worst cases (Brute force, Abbasov's Algorithm, Quick sort, Insertion sort). In contrast, there are no big differences at some algorithms in the best and worst cases (Merge sort, Boyer Moore majority vote algorithms). At the part of comparing different algorithms in the same input, it is clear that algorithms' performance can change very much. For example, brute force is known as the worst algorithm, but for the input which half of majority; it performed best performance. Graphs show that generally Divide&Conquer algorithm is the worst one generally. On the other hand, Boyer Moore majority vote algorithm is the best algorithm generally as we expected from theoretical part. Abbasov's Algorithm and hashing algorithm perform average performance among other algorithms but these algorithms can be unstable depending on input. Insertion sort algorithm and quick sort algorithm are very slow in their worst cases but they can be fast at their best case. Merge sort algorithm can be preferred as a stable and fast algorithm. It is possible that there are some unexpected time values in the graphs. Computer's instant performance drop or measurement errors can cause these values. However, interpretable graphs were constructed. Finally, these graphs gave very useful information about different cases for same algorithm and different algorithms for same input.