CSE2046 Homework 2 Report

Ahmet Elburuz Gürbüz 150116024

What is an algorithm?

An algorithm is a detailed series of instructions for carrying out an operation or solving a problem. In a non-technical approach, we use algorithms in everyday tasks, such as a recipe to bake a cake or a do-it-yourself handbook.

Technically, computers use algorithms to list the detailed instructions for carrying out an operation. For example, to sort an array, the computer uses an algorithm. To accomplish this task appropriate data must be entered into the system. In terms of efficiency, various algorithms are able to accomplish operations or problem solving easily and quickly.
Lets continue on the example; to sort an array we can use lots of different algorithms such as

1. Selection Sort
2. Bubble Sort
3. Recursive Bubble Sort
4. Insertion Sort
5. Recursive Insertion Sort
6. Merge Sort
7. Iterative Merge Sort
8. Quick Sort
9. Iterative Quick Sort
10. Heap Sort
11. Counting Sort
12. Radix Sort
13. Bucket Sort
14. ShellSort
15. TimSort
16. Comb Sort
17. Pigeonhole Sort
18. Cycle Sort
19. Cocktail Sort
20. Strand Sort

.
.
.

Homework 2

But in this homework we will just make experiment and implement
1. Insertion-sort,
2. Merge-sort,
3. Heap-sort,
4. Quick-select,
5. Median of Three

We designed an experiment to compare different algorithms for selection problem. Our implementation solves problem of finding median element in an unsorted list of n numbers. Firstly we implement ordinary sorting by Insertion-sort and returning the median element in the list, sorting by Merge-sort and returning the median element in the list, heap sort find the middle element by deleting n/2 elements ,for not sorting the list applying quick

select algorithm, which is based on array partitioning and applying quick select algorithm with using median-of-three selection.

In this experiment, we compared these five methods for different values of k and various input lists of various sizes and various characteristics. We considered to work on sample inputs from small to largest to make decisions best clarify the performance characteristics of algorithms. Also we used physical unit of time.

We prepared an array builder for the assignment, This array constructor creates an array with random values from 0 to array size, an array in ascending order, an array in descending order. The array is called in a 5000-frequency "for loop".

Reverse ordered array and sorted array had been chosed to show best case and worst case of the insertion sort algorithm, average case can be determined by random array.

Since merge sort has Θ(nlog(n)) time complexity, there is no certain worst, best or average case to show in input, the expecting result is to show the same result for all inputs.

Quick select could have worst case if it has implemented such a way that the pivot selected as first or last element. But if median has implemented as pivot then there is no worst case for quick select. And we have made the implementation with median so O(n(log(n)) time complexity has been waiting for all case of quick select.

Since the heap sort has heapConstrucion() function with O(n) time complexity and downheap() function with O(log(n)) time complexity, the final will be O(nlog(n)) time complexity. Thus, there is no specified input to see different time complexity as result.

1-) Insertion Sort

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

Sorting is typically done in-place, by iterating up the array, growing the sorted list behind it. At each array-position, it checks the value there against the largest value in the sorted list (which happens to be next to it, in the previous array-position checked). If larger, it leaves the element in place and moves to the next. If smaller, it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts into that correct position

Best Case Of Insertion Sort:

The best case input is an array that is already sorted. In this case insertion sort has a linear running time (i.e., O(n)). During each iteration, the first remaining element of the input is only compared with the right-most element of the sorted subsection of the array.

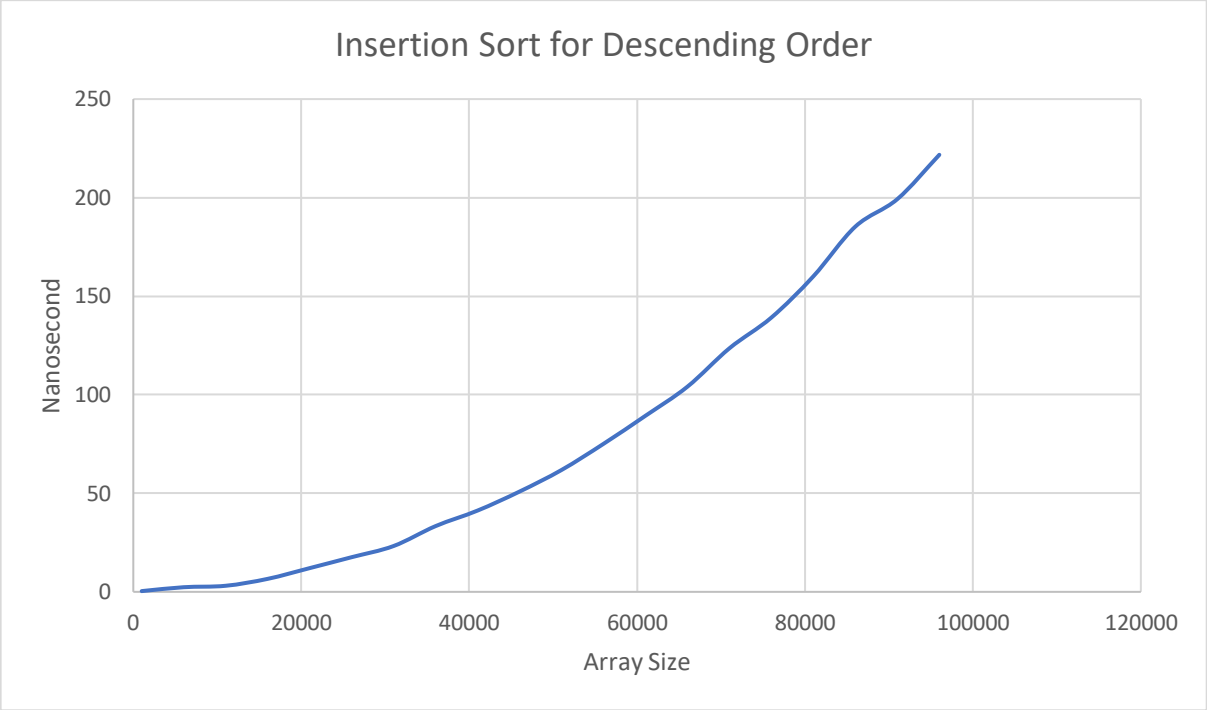
Worst Case Of Insertion Sort:

The simplest worst case input is an array sorted in reverse order. The set of all worst case inputs consists of all arrays where each element is the smallest or second-smallest of the elements before it. In these cases every iteration of the inner loop will scan and shift the entire sorted subsection of the array before inserting the next element. This gives insertion sort a quadratic running time (i.e., $O(n^2)$).

Average Case Of Insertion Sort

Suppose that the array starts out in a random order. Then, on average, we'd expect that each element is less than half the elements to its left. In this case, on average, a call to insert on a subarray of k elements would slide k/2 of them. The running time would be half of the worst-case running time. But in asymptotic notation, where constant coefficients don't matter, the running time in the average case would still be $\Theta(n^2)$, just like the worst case.

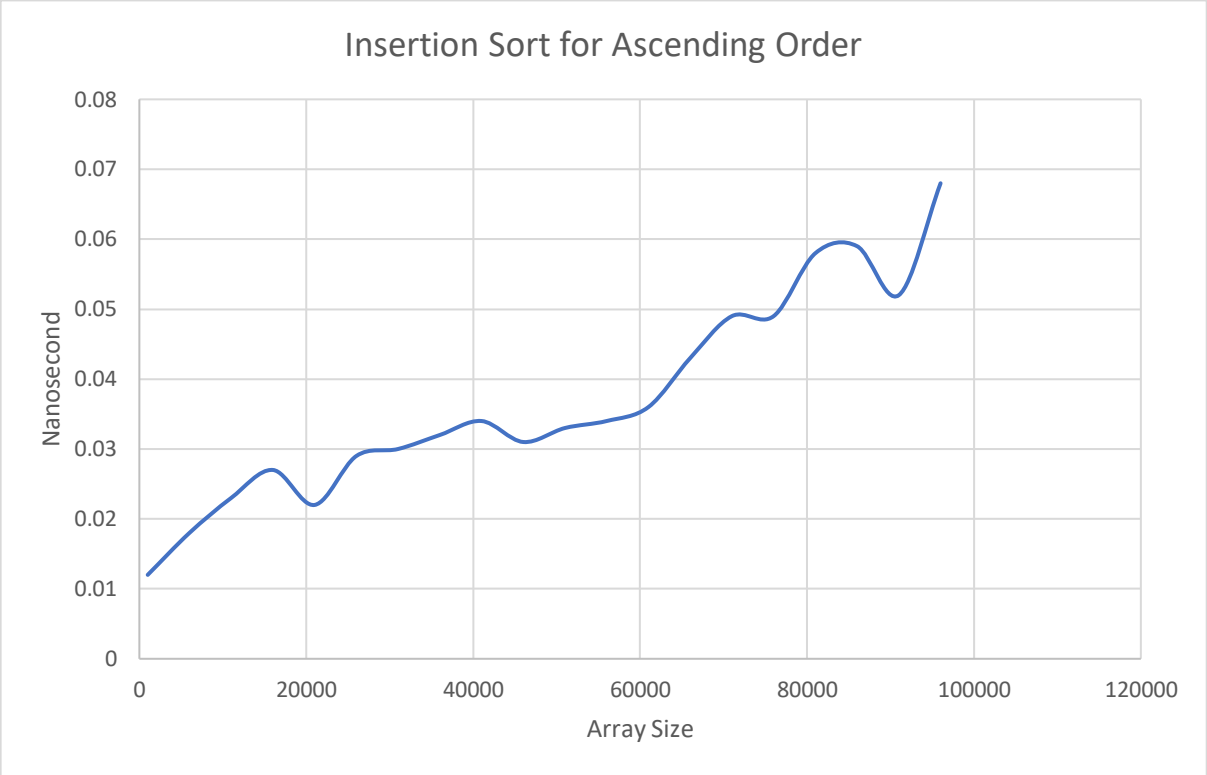
Array Size	Insertion Sort for Descending Order (Time)
1000	0,37
6000	2,352
11000	3,079
16000	6,587
21000	12,031
26000	17,579
31000	23,25
36000	33,35
41000	41,233
46000	50,872
51000	61,828
56000	75,086
61000	89,341
66000	104,103
71000	123,585
76000	139,174
81000	160,111
86000	185,324
91000	199,269
96000	221,735



By the empirical results time complexity are approximately $\frac{n^2}{4}$ for n input. So we can say $O(n^2)$ for empirical results and $O(n^2)$ for theoretical results. And our findings meet theoretical expectations.

Array Size	Insertion Sort for Ascending Order
1000	0,012
6000	0,018
11000	0,023
16000	0,027
21000	0,022
26000	0,029
31000	0,03
36000	0,032
41000	0,034
46000	0,031
51000	0,033
56000	0,034
61000	0,036
66000	0,043
71000	0,049
76000	0,049
81000	0,058
86000	0,059

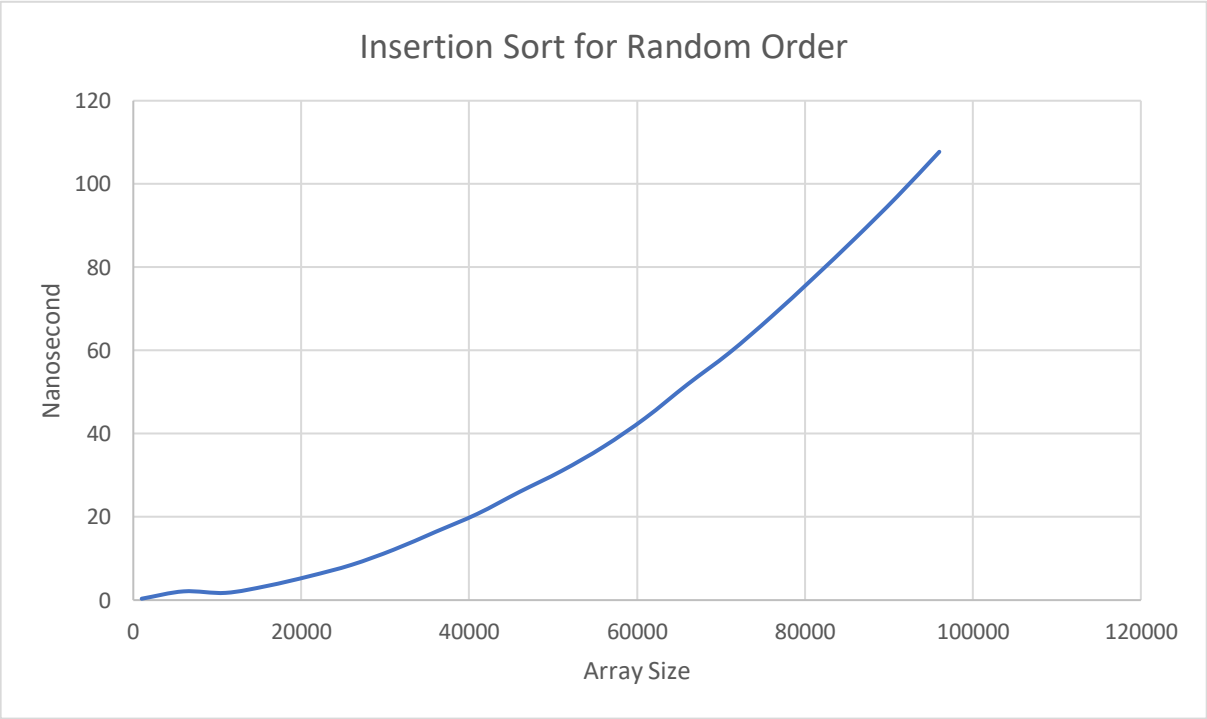
91000	0,052
96000	0,068



By the empirical results time complexity are approximately n for n input. So we can say O(n) for empirical results and O(n) for theoretical results. And our findings meet theoretical expectations.

Array Size	Insertion Sort for Random Order
1000	0,34
6000	2,108
11000	1,746
16000	3,438
21000	5,768
26000	8,481
31000	12,112
36000	16,405
41000	20,747
46000	26,007
51000	31,037
56000	36,866
61000	43,778
66000	51,83
71000	59,444

76000	68,106
81000	77,383
86000	86,988
91000	97,019
96000	107,721



By the empirical results time complexity are approximately n^2 for n input. So we can say $O(n^2)$ for empirical results and $O(n^2)$ for theoretical results. And our findings meet theoretical expectations.

2-) Merge Sort

Merge sort is a divide-and-conquer algorithm based on the idea of breaking down a list into several sub-lists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.

Idea:

- Divide the unsorted list into N sublists, each containing 1 element.
- Take adjacent pairs of two singleton lists and merge them to form a list of 2 elements. N will now convert into N/2 lists of size 2.
- Repeat the process till a single sorted list of obtained.

While comparing two sublists for merging, the first element of both lists is taken into consideration. While sorting in ascending order, the element that is of a lesser value becomes a new element of the sorted list. This procedure is repeated until both the smaller sublists are empty and the new combined sublist comprises all the elements of both the sublists.

Time Complexity Of Merge Sort

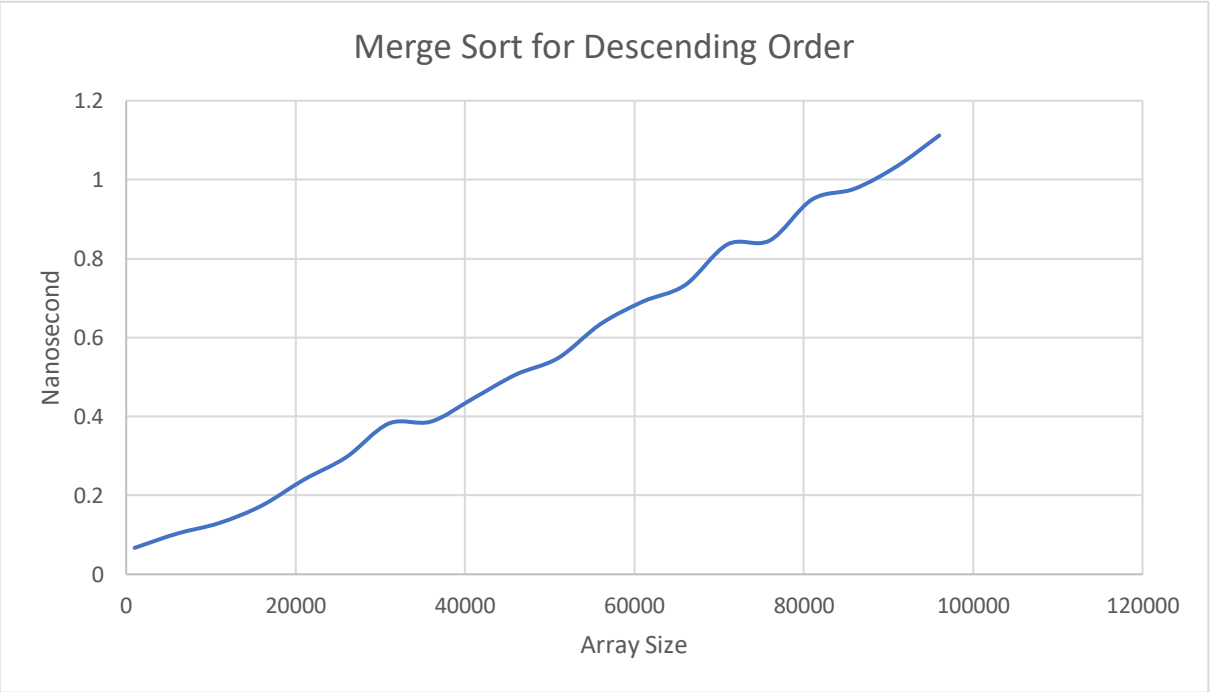
Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

$T(n) = 2T(n/2) + \Theta(n)$

The above recurrence can be solved either using Recurrence Tree method or Master method.

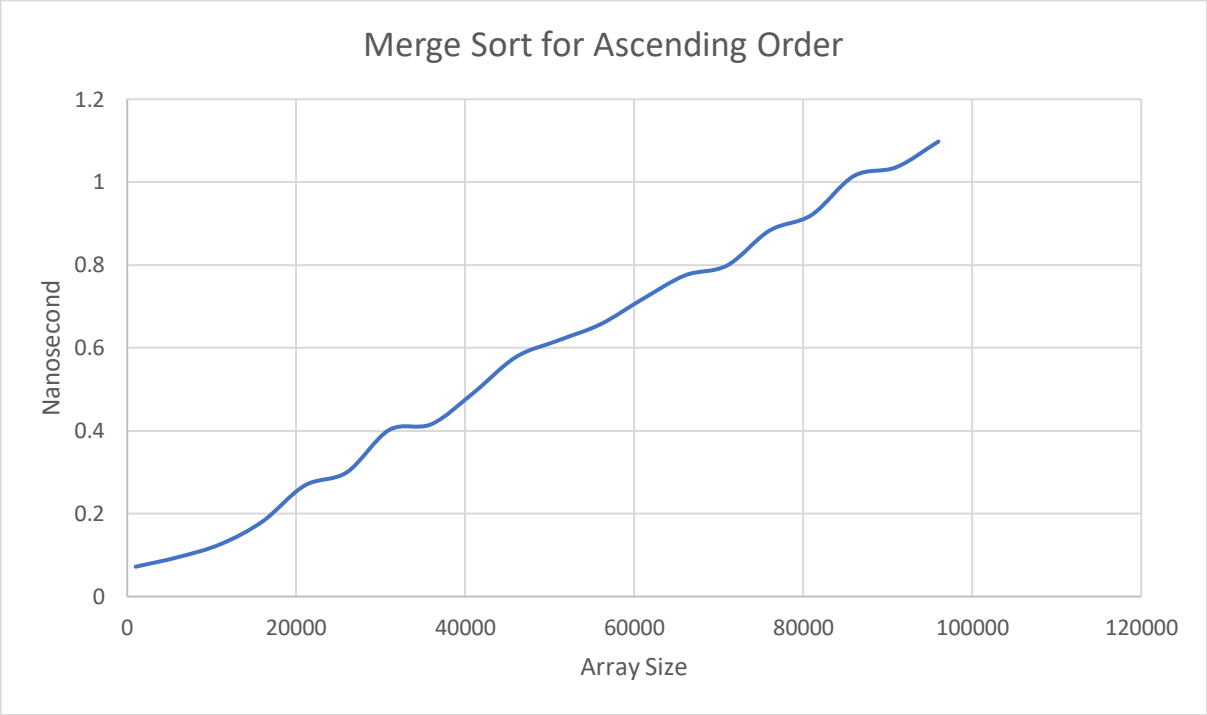
It falls in case II of Master Method and solution of the recurrence is ***Θ(nlog(n))***. Since the algorithm is same for the input size, it doesnt mather its worst, best or average case and it is *Θ(nlog(n))*.

Array Size	Merge Sort for Descending Order
1000	0,067
6000	0,103
11000	0,13
16000	0,174
21000	0,24
26000	0,297
31000	0,382
36000	0,387
41000	0,446
46000	0,506
51000	0,548
56000	0,634
61000	0,691
66000	0,733
71000	0,836
76000	0,846
81000	0,95
86000	0,977
91000	1,034
96000	1,112



Array Size	Merge Sort for Ascending Order
1000	0,072
6000	0,095
11000	0,126
16000	0,181
21000	0,268
26000	0,3
31000	0,402
36000	0,416
41000	0,492
46000	0,578
51000	0,618
56000	0,657
61000	0,718
66000	0,775
71000	0,799
76000	0,883
81000	0,921
86000	1,015
91000	1,036
96000	1,098

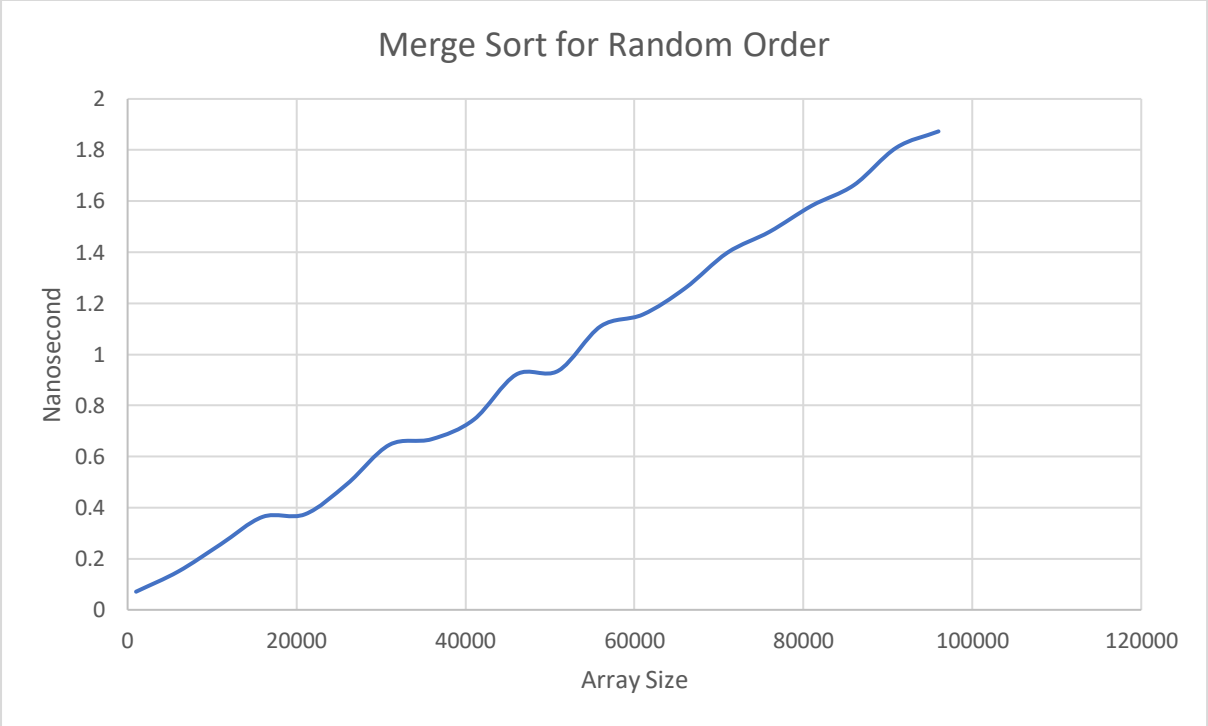
By the empirical results time complexity are approximately $3(nlogn)$ for n input. So we can say $O(nlogn)$ for empirical results and $O(nlogn)$ for theoretical results. And our findings meet theoretical expectations



By the empirical results time complexity are approximately $3(nlogn)$ for n input. So we can say $O(nlogn)$ for empirical results and $O(nlogn)$ for theoretical results. And our findings meet theoretical expectations.

Array Size	Merge Sort for Random Order
1000	0,071
6000	0,15
11000	0,256
16000	0,364
21000	0,374
26000	0,493
31000	0,646
36000	0,668
41000	0,745
46000	0,921
51000	0,936
56000	1,11
61000	1,156
66000	1,259
71000	1,398

76000	1,48
81000	1,582
86000	1,663
91000	1,809
96000	1,873



By the empirical results time complexity are approximately $4(nlogn)$ for n input. So we can say $O(nlogn)$ for empirical results and $O(nlogn)$ for theoretical results. And our findings meet theoretical expectations.

3-) Heap Sort

The Heapsort algorithm involves preparing the list by first turning it into a max heap. The algorithm then repeatedly swaps the first value of the list with the last value, decreasing the range of values considered in the heap operation by one, and sifting the new first value into its position in the heap. This repeats until the range of considered values is one value in length.

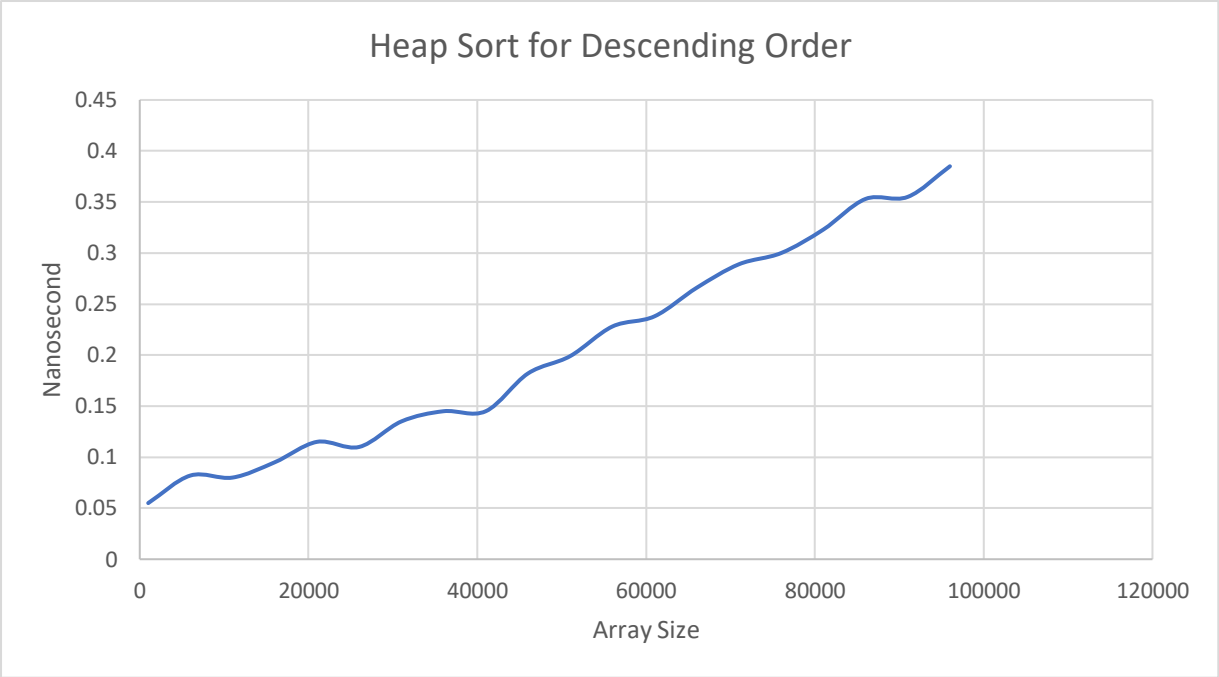
The steps are:

1. Call the buildMaxHeap() function on the list. Also referred to as heapify(), this builds a heap from a list in O(n) operations.
2. Swap the first element of the list with the final element. Decrease the considered range of the list by one.

3. Call the siftDown() function on the list to sift the new first element to its appropriate index in the heap.
4. Go to step (2) unless the considered range of the list is one element

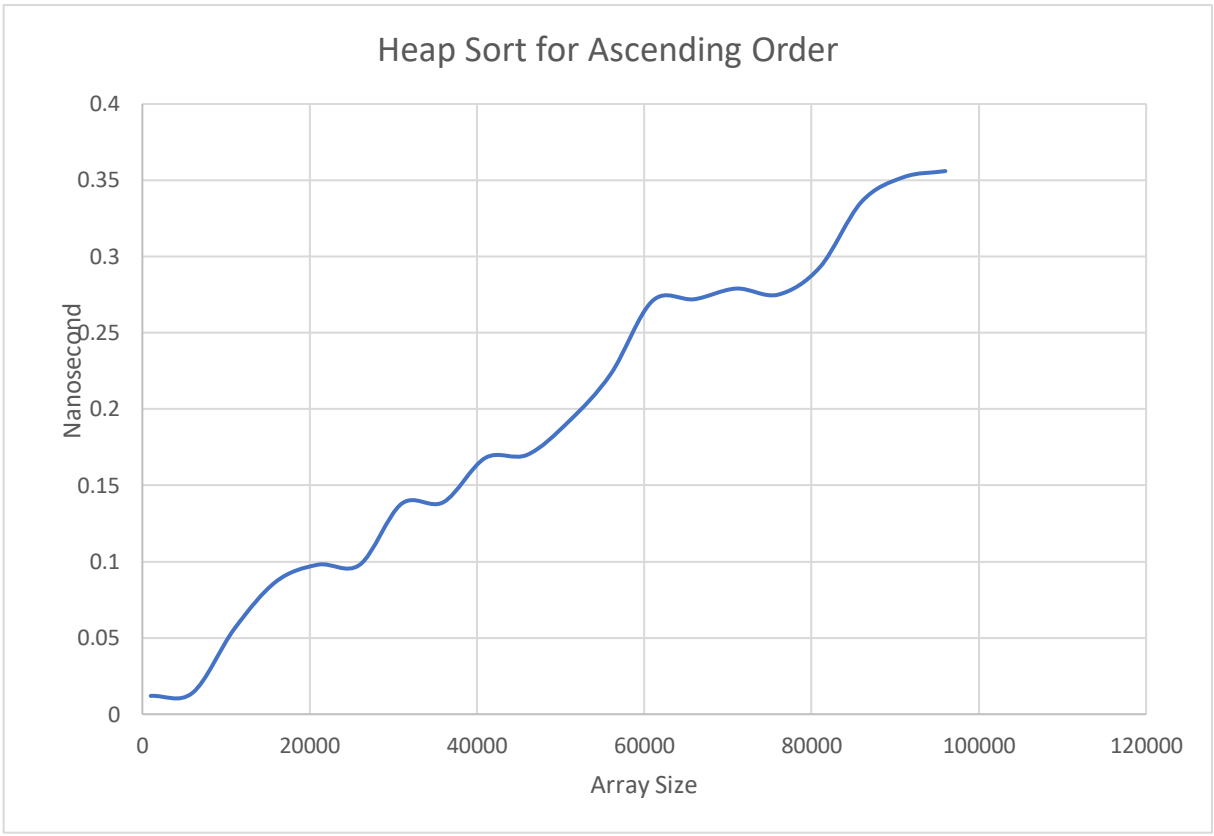
The buildMaxHeap() operation is run once, and is O(*n*) in performance. The siftDown() function is O(log *n*), and is called *n* times. Therefore, the performance of this algorithm is O(*n* + *n* log *n*) = ***O(n log n).***

Array Size	Heap Sort for Descending Order
1000	0,055
6000	0,082
11000	0,08
16000	0,095
21000	0,115
26000	0,11
31000	0,135
36000	0,145
41000	0,145
46000	0,182
51000	0,199
56000	0,228
61000	0,238
66000	0,266
71000	0,289
76000	0,3
81000	0,323
86000	0,353
91000	0,355
96000	0,385



By the empirical results time complexity are approximately (nlogn) for n input. So we can say O(nlogn) for empirical results and O (nlogn) for theoretical results. And our findings meet theoretical expectations.

Array Size	Heap Sort for Ascending Order
1000	0,012
6000	0,014
11000	0,056
16000	0,087
21000	0,098
26000	0,098
31000	0,138
36000	0,139
41000	0,168
46000	0,17
51000	0,192
56000	0,223
61000	0,271
66000	0,272
71000	0,279
76000	0,275
81000	0,293
86000	0,336
91000	0,352
96000	0,356

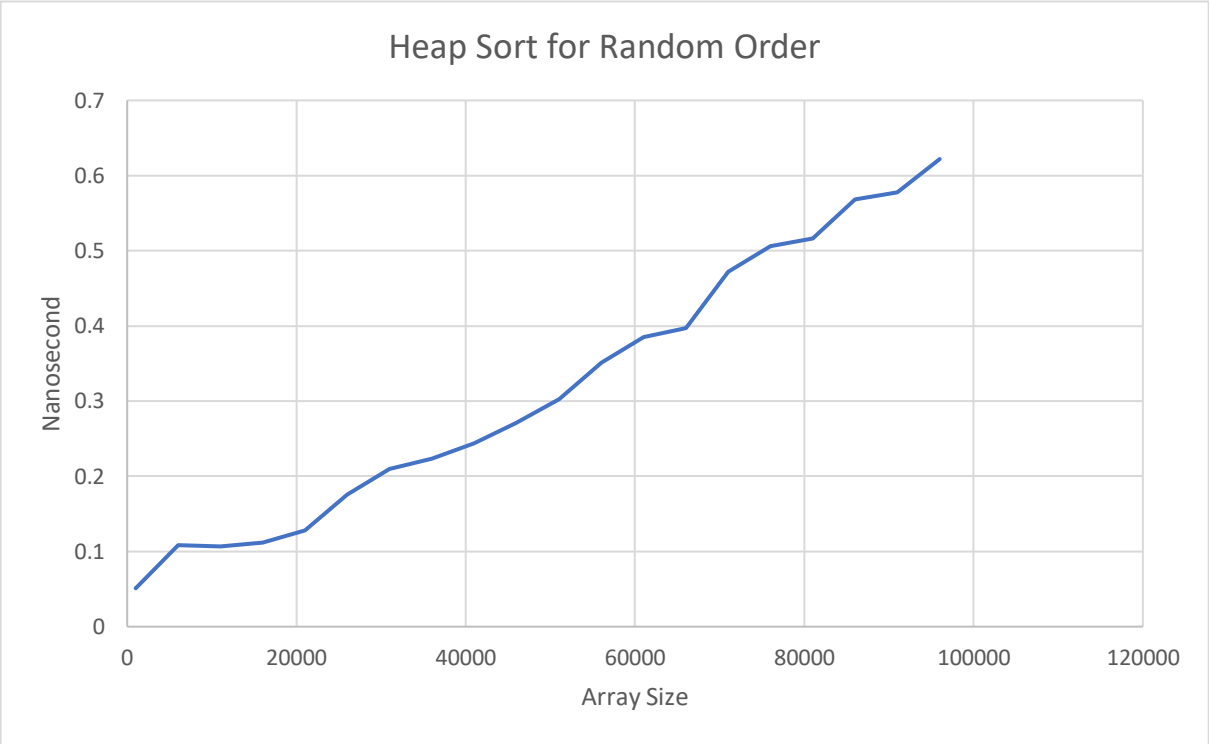


For best case :

We thought that for the best case, the inputs are the same. And the size of inputs should be as small as possible By the empirical results time complexity are approximately (nlogn) for n input. So we can say O(nlogn) for empirical results and O (nlogn) for theoretical results. And our findings meet theoretical expectations.

Array Size	Heap Sort for Random Order
1000	0,051
6000	0,108
11000	0,107
16000	0,112
21000	0,128
26000	0,176
31000	0,21
36000	0,223
41000	0,244
46000	0,271
51000	0,303
56000	0,351
61000	0,385
66000	0,397

71000	0,472
76000	0,506
81000	0,516
86000	0,568
91000	0,578
96000	0,622

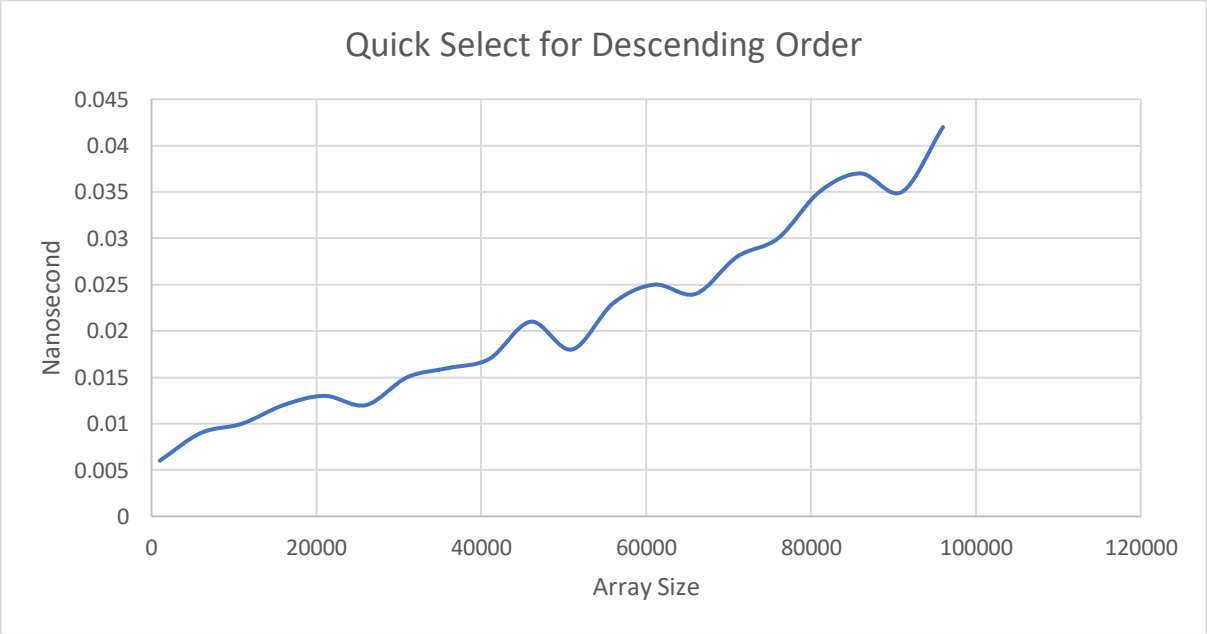


By the empirical results time complexity are approximately 2(nlogn) for n input. So we can say O(nlogn) for empirical results and O(nlogn) for theoretical results. And our findings meet theoretical expectations.

4-) Quick Select

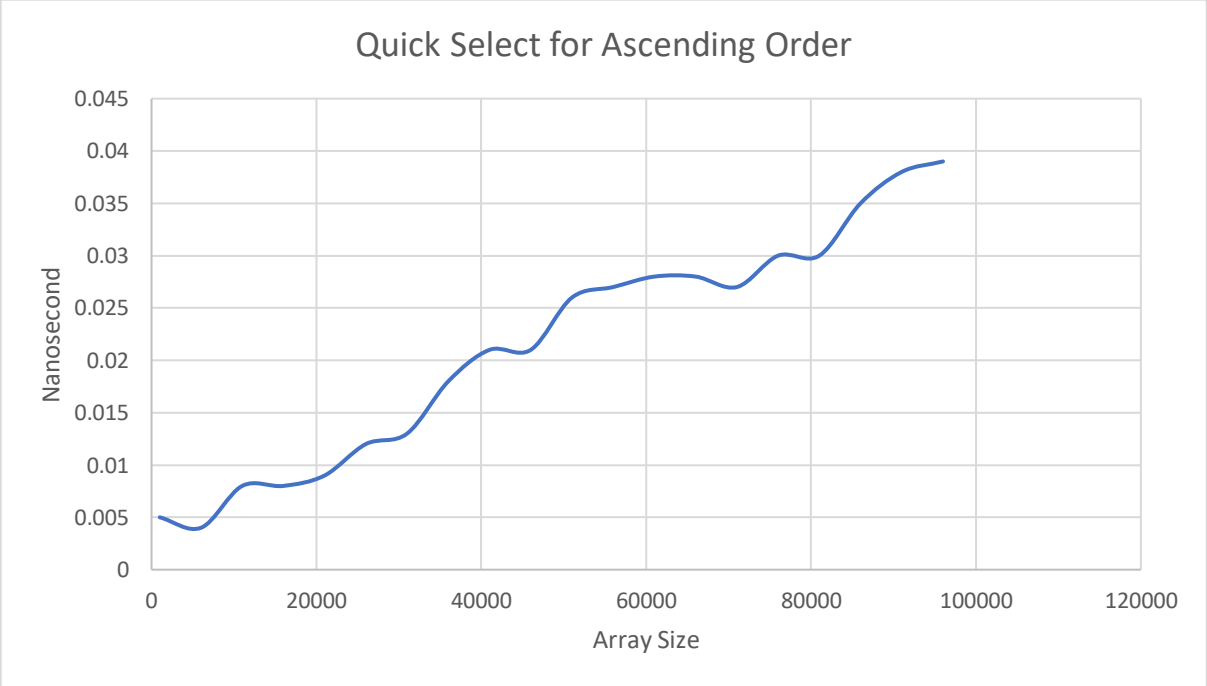
Like quicksort, the quickselect has good average performance, but is sensitive to the pivot that is chosen. If good pivots are chosen, meaning ones that consistently decrease the search set by a given fraction, then the search set decreases in size exponentially and by induction one sees that performance is linear, as each step is linear and the overall time is a constant times this (depending on how quickly the search set reduces). However, if bad pivots are consistently chosen, such as decreasing by only a single element each time, then worst-case performance is quadratic: $O(n^2)$. This occurs for example in searching for the maximum element of a set, using the first element as the pivot, and having sorted data and here we have dealt with one of the worst cases, choosing the first element pivot. Average complexity from $O(n\log n)$ to $O(n)$, with a worst case of $O(n^2)$.

Array Size	Quick Select for Descending Order
1000	0,006
6000	0,009
11000	0,01
16000	0,012
21000	0,013
26000	0,012
31000	0,015
36000	0,016
41000	0,017
46000	0,021
51000	0,018
56000	0,023
61000	0,025
66000	0,024
71000	0,028
76000	0,03
81000	0,035
86000	0,037
91000	0,035
96000	0,042



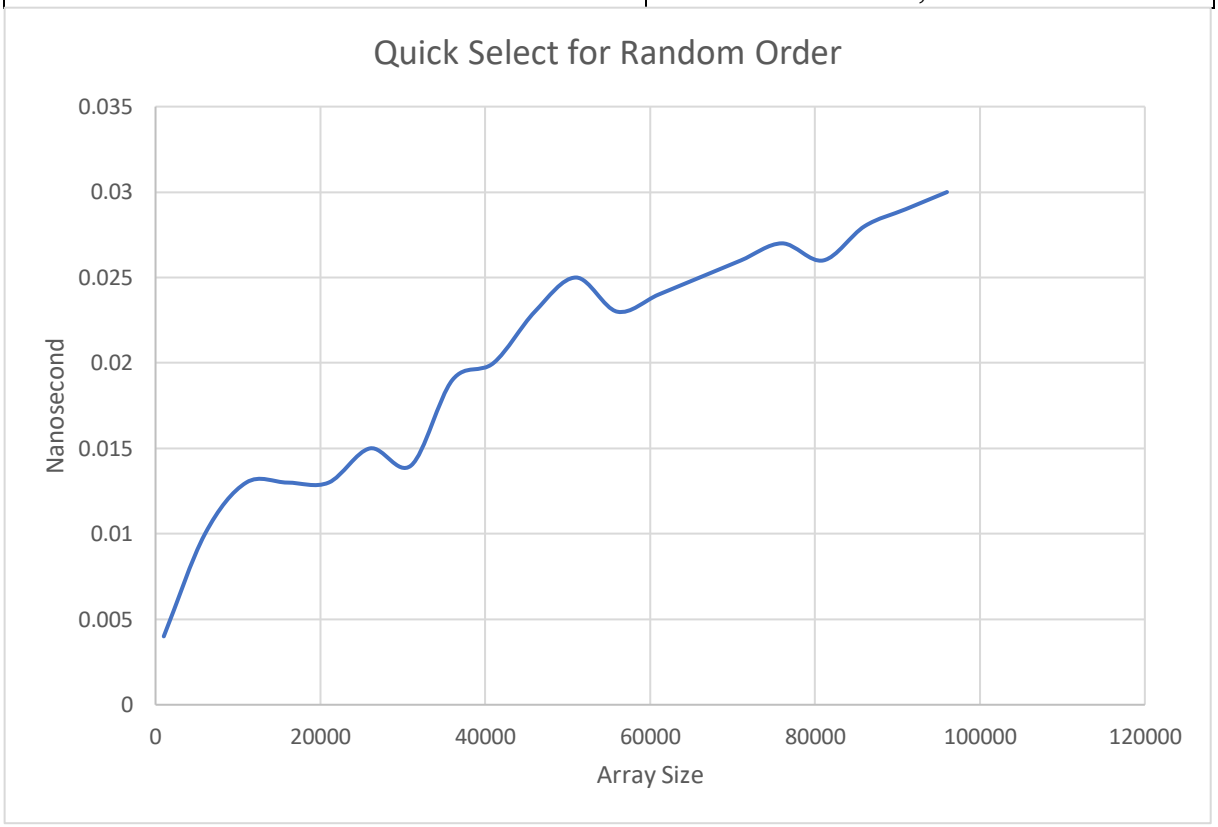
By the empirical results time complexity are approximately (nlogn) for n input. So we can say O(nlogn) for empirical results and O(nlogn) for theoretical results. And our findings meet theoretical expectations.

Array Size	Quick Select for Ascending Order
1000	0,005
6000	0,004
11000	0,008
16000	0,008
21000	0,009
26000	0,012
31000	0,013
36000	0,018
41000	0,021
46000	0,021
51000	0,026
56000	0,027
61000	0,028
66000	0,028
71000	0,027
76000	0,03
81000	0,03
86000	0,035
91000	0,038
96000	0,039



By the empirical results time complexity are approximately (nlogn) for n input. So we can say O(nlogn) for empirical results and O(nlogn) for theoretical results. And our findings meet theoretical expectations.

Array Size	Quick Select for Random Order
1000	0,004
6000	0,01
11000	0,013
16000	0,013
21000	0,013
26000	0,015
31000	0,014
36000	0,019
41000	0,02
46000	0,023
51000	0,025
56000	0,023
61000	0,024
66000	0,025
71000	0,026
76000	0,027
81000	0,026
86000	0,028
91000	0,029
96000	0,03

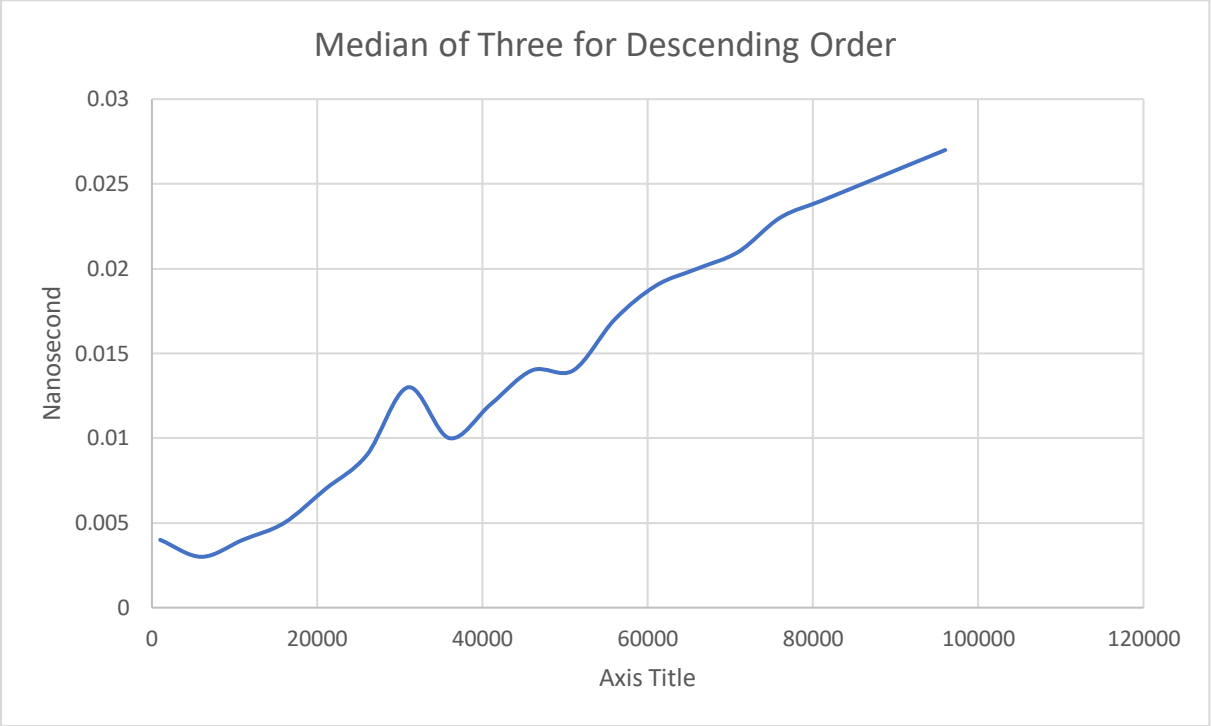


By the empirical results time complexity are approximately $\frac{3}{4} * (nlogn)$ for n input. So we can say O(nlogn) for empirical results and O(nlogn) for theoretical results. And our findings meet theoretical expectations.

5-) Median of Three

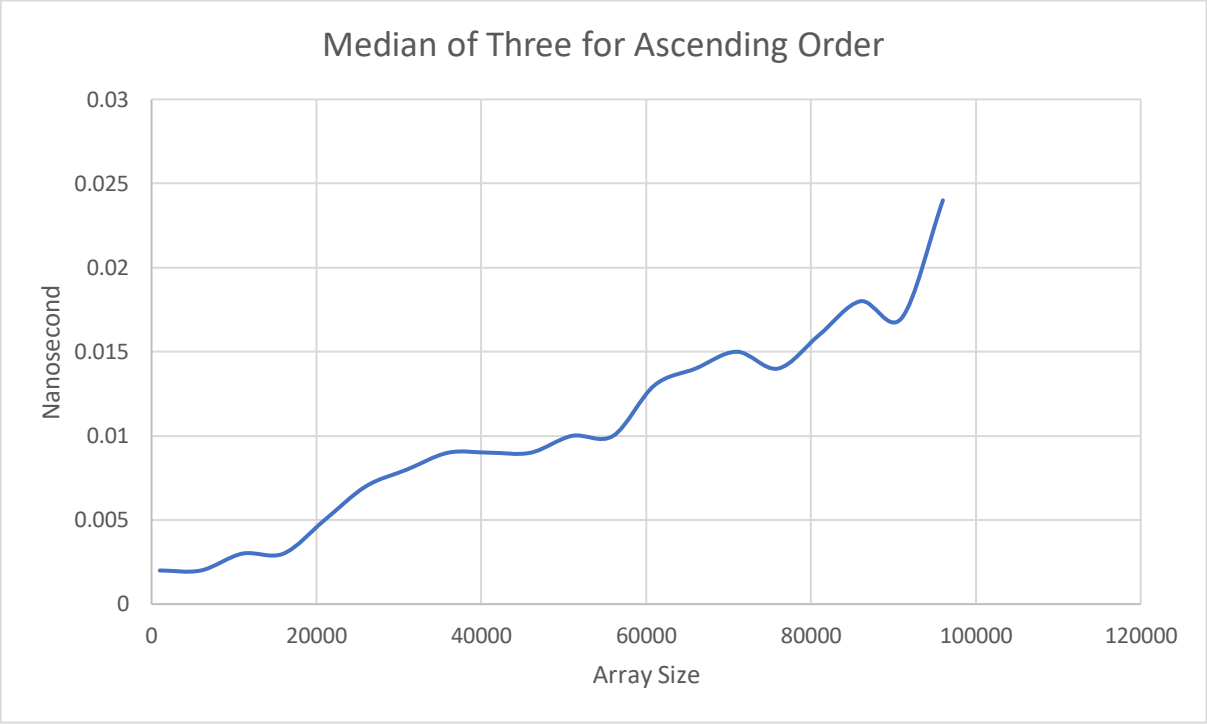
We know median of three's time complexity in average case $O(nlogn)$ and worst case $O(n^2)$. We got values close to the values which we were expecting.

Array Size	Median of Three for Descending Order
1000	0,004
6000	0,003
11000	0,004
16000	0,005
21000	0,007
26000	0,009
31000	0,013
36000	0,01
41000	0,012
46000	0,014
51000	0,014
56000	0,017
61000	0,019
66000	0,02
71000	0,021
76000	0,023
81000	0,024
86000	0,025
91000	0,026
96000	0,027



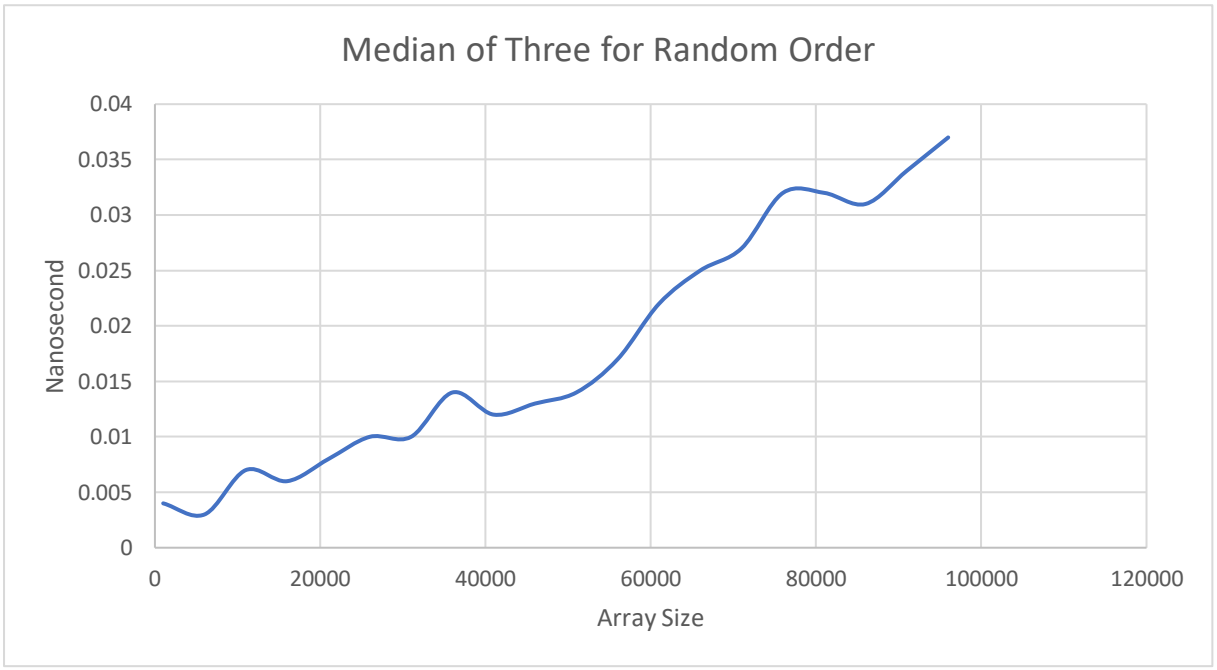
By the empirical results time complexity are approximately $\frac{1}{2} * (nlogn)$ for n input. So we can say O(nlogn) for empirical results and O(nlogn) for theoretical results. And our findings meet theoretical expectations.

Array Size	Median of Three for Ascending Order
1000	0,002
6000	0,002
11000	0,003
16000	0,003
21000	0,005
26000	0,007
31000	0,008
36000	0,009
41000	0,009
46000	0,009
51000	0,01
56000	0,01
61000	0,013
66000	0,014
71000	0,015
76000	0,014
81000	0,016
86000	0,018
91000	0,017
96000	0,024



By the empirical results time complexity are approximately $\frac{1}{2} * (nlogn)$ for n input. So we can say O(nlogn) for empirical results and O(nlogn) for theoretical results. And our findings meet theoretical expectations.

Array Size	Median of Three for Random Order
1000	0,004
6000	0,003
11000	0,007
16000	0,006
21000	0,008
26000	0,01
31000	0,01
36000	0,014
41000	0,012
46000	0,013
51000	0,014
56000	0,017
61000	0,022
66000	0,025
71000	0,027
76000	0,032
81000	0,032
86000	0,031
91000	0,034
96000	0,037



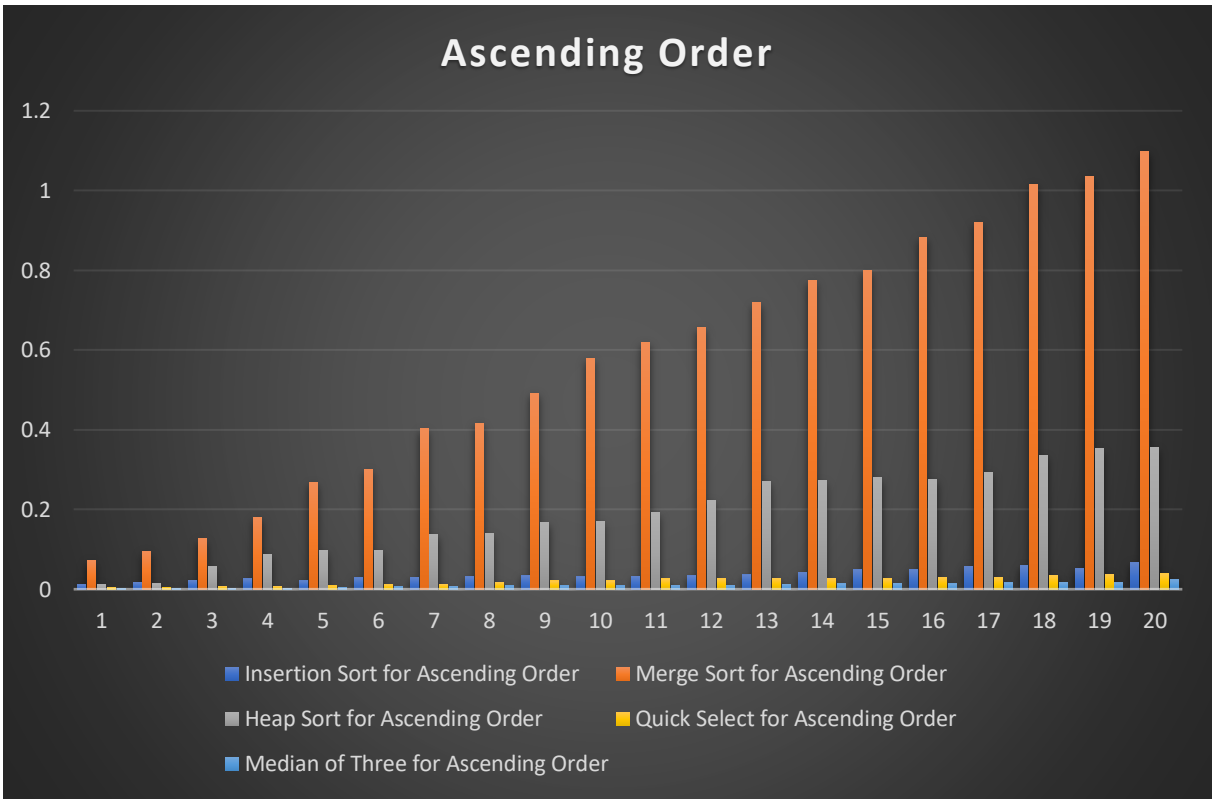
By the empirical results time complexity are approximately $\frac{3}{4} * (nlogn)$ for n input. So we can say O(nlogn) for empirical results and O(nlogn) for theoretical results. And our findings meet theoretical expectations.

Analyzing Results:

I did not place the insertion order in descending and random ordered graphs because their values were too large, reducing the readability of the graph.

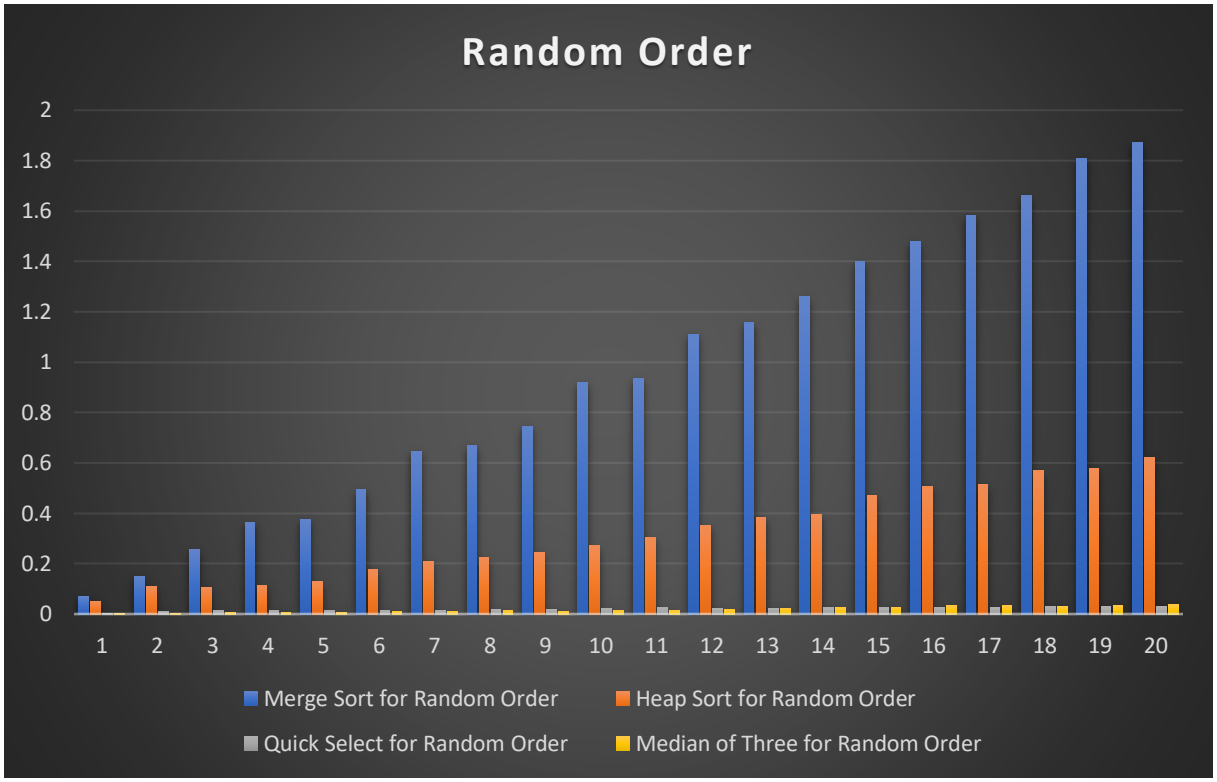
Insertion Sort for Descending Order	Merge Sort for Descending Order	Heap Sort for Descending Order	Quick Select for Descending Order	Median of Three for Descending Order
0,37	0,067	0,055	0,006	0,004
2,352	0,103	0,082	0,009	0,003
3,079	0,13	0,08	0,01	0,004
6,587	0,174	0,095	0,012	0,005
12,031	0,24	0,115	0,013	0,007
17,579	0,297	0,11	0,012	0,009
23,25	0,382	0,135	0,015	0,013
33,35	0,387	0,145	0,016	0,01
41,233	0,446	0,145	0,017	0,012
50,872	0,506	0,182	0,021	0,014
61,828	0,548	0,199	0,018	0,014
75,086	0,634	0,228	0,023	0,017
89,341	0,691	0,238	0,025	0,019
104,103	0,733	0,266	0,024	0,02
123,585	0,836	0,289	0,028	0,021
139,174	0,846	0,3	0,03	0,023
160,111	0,95	0,323	0,035	0,024
185,324	0,977	0,353	0,037	0,025

199,269	1,034	0,355	0,035	0,026
221,735	1,112	0,385	0,042	0,027



Insertion Sort for Ascending Order	Merge Sort for Ascending Order	Heap Sort for Ascending Order	Quick Select for Ascending Order	Median of Three for Ascending Order
0,012	0,072	0,012	0,005	0,002
0,018	0,095	0,014	0,004	0,002
0,023	0,126	0,056	0,008	0,003
0,027	0,181	0,087	0,008	0,003
0,022	0,268	0,098	0,009	0,005
0,029	0,3	0,098	0,012	0,007
0,03	0,402	0,138	0,013	0,008
0,032	0,416	0,139	0,018	0,009
0,034	0,492	0,168	0,021	0,009
0,031	0,578	0,17	0,021	0,009
0,033	0,618	0,192	0,026	0,01
0,034	0,657	0,223	0,027	0,01
0,036	0,718	0,271	0,028	0,013
0,043	0,775	0,272	0,028	0,014
0,049	0,799	0,279	0,027	0,015
0,049	0,883	0,275	0,03	0,014
0,058	0,921	0,293	0,03	0,016
0,059	1,015	0,336	0,035	0,018

0,052	1,036	0,352	0,038	0,017
0,068	1,098	0,356	0,039	0,024



Insertion Sort for Random Order	Merge Sort for Random Order	Heap Sort for Random Order	Quick Select for Random Order	Median of Three for Random Order
0,34	0,071	0,051	0,004	0,004
2,108	0,15	0,108	0,01	0,003
1,746	0,256	0,107	0,013	0,007
3,438	0,364	0,112	0,013	0,006
5,768	0,374	0,128	0,013	0,008
8,481	0,493	0,176	0,015	0,01
12,112	0,646	0,21	0,014	0,01
16,405	0,668	0,223	0,019	0,014
20,747	0,745	0,244	0,02	0,012
26,007	0,921	0,271	0,023	0,013
31,037	0,936	0,303	0,025	0,014
36,866	1,11	0,351	0,023	0,017
43,778	1,156	0,385	0,024	0,022
51,83	1,259	0,397	0,025	0,025
59,444	1,398	0,472	0,026	0,027
68,106	1,48	0,506	0,027	0,032

77,383	1,582	0,516	0,026	0,032
86,988	1,663	0,568	0,028	0,031
97,019	1,809	0,578	0,029	0,034
107,721	1,873	0,622	0,03	0,037

- They all showed an increase in nanoseconds when the size of the array increased, but at least the quick select algorithm and the median of three.
- Insertion sort is less efficient on large size input lists than quicksort, heapsort, median of three or merge sort algorithms. However insertion sort is more efficient on small size input lists than other algorithms. The main advantage of the insertion sort is its simplicity. The insertion sort is an in-place sorting algorithm so the space requirement is minimal. With n-squared steps required for every n element to be sorted, the insertion sort does not deal well with a huge list.
- Merge sort can be applied files of any size. But merge sort less efficient than other sort.It requires extra space »N
- Quick sort is very efficient algorithms. because it gives best performance on large list of inputs.But it gives worst performance on ascending or descending ordered input lists. The advantage of Quick sort is that it is used for small elements of array as well as large elements of array. Disadvantage of Quick sort is that the worst case of quick sort is same as a bubble sort or selection sort.
- Heap sort is very efficiency algorithms. This means it performs equally well in the best, average and worst cases.
- Merge sort and heap sort are independent of distribution of inputs

Summary, we have found some results.For example,if we want to sort an array with efficient time complexity and less comparison ,we can use merge sort;if we find element from beginning or end in very big data with efficient time complexity and less comparison, we can use quick select algoritm.However, if we have small array size ,we can use insertion sort because of basic implementation reason.If we want to find median of array , it would be wise to use the median of three method.

Finally we got that different techniques work efficiently in different situations and conditions. The programer decides which method to use with considering number of comparison,time complexity and memory usage to make efficient program.

Note : I tried to keep the fluctuations in these graphics to a minimum because the fluctuations in my own computer were quite high when no program was open while measuring on the computer, so I could take measurements on another computer.