ANALYSIS BASED ON EXPERIMENTS

In this project, nine different experiments held to understand the total run-time of each sorting algorithm based on,

- array size and,
- array's element order (random elements, sorted elements and almost sorted elements).

On the table below, each sorting algorithm and its total run-time is shown, depending on array's element order. Although the total run time of each algorithm is correct above, the numbers below in the table are not copied from those snippets; they are copied from the console. Colors for rows indicate the size of the array,

- **light** orange is the array of size a thousand,
- **dark** orange is the array size of ten thousand,
- **darkest** orange is the array size of a hundred thousand.

- Each value is in nano-seconds. Colors represent array sizes 1000, 10000, and 100000 respectively.

	Selection Sort	Bubble Sort	Bubble w/ Swaps Count	Insertion Sort	Quick Sort	Merge Sort
Random	6792100	6877500	6839300	52600	3672100	1009700
Sorted	6777000	7050900	8063100	77800	4105300	1450800
Almost Sorted	6706500	7958500	9034100	50700	3329200	930300
Random	76465600	36273400	44091700	532200	38388500	3671200
Sorted	28975900	36451300	34867500	449900	35073100	3562400
Almost Sorted	74976100	34744700	35174700	442000	20393300	3149300
Random	5956554300	2298005900	2261315900	3230400	859371400	21632000
Sorted	1986926800	2408492500	2554496500	3131700	1916091000	18979900
Almost Sorted	6151878500	2407903300	2416010600	3073500	1562371200	18567300

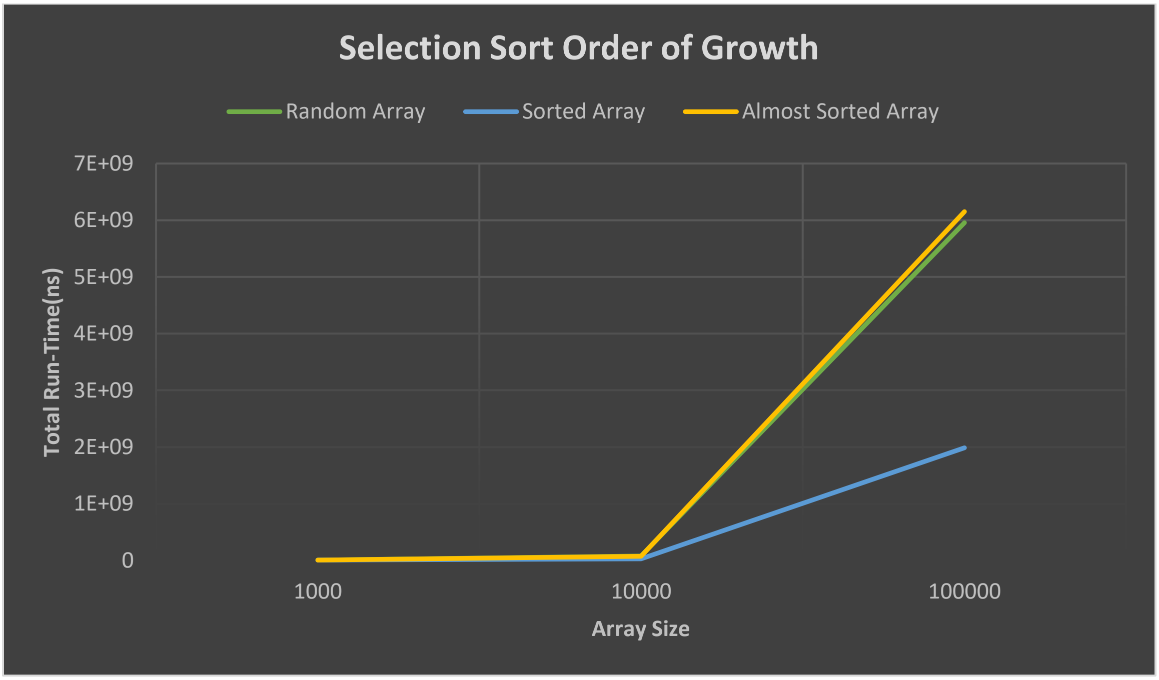
Sorting Algorithms and Their Graphs Based on Run-Time:

In this part of the analysis, each sorting algorithm is reflected on its own graph. In each graph, in a given array size, total run-time is shown with each array's element of order. Y-axis shows the total run-time of the sorting algorithm and the x-axis is the array size.

Using the data in the table above for each sorting algorithm, I will be explaining the results with their theoretical analysis. Each material and formulas will be cited from class notes and the textbook we are using currently.

Side note: Unfortunately, as it can be seen, results for run-time are very unstable. I will do my best to analyze each sorting algorithm with the help of course materials.

Selection Sort:

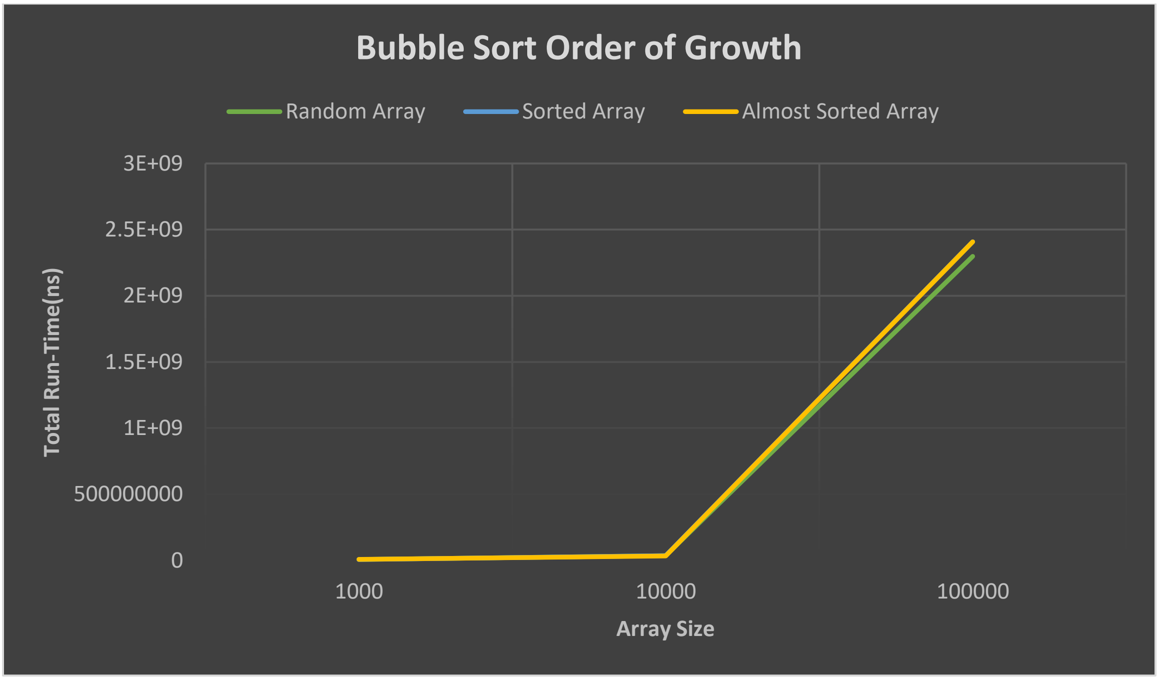


Random	Sorted	Almost Sorted	Random	Sorted	Almost Sorted	Random	Sorted	Almost Sorted
6792100	6777000	6706500	76465600	28975900	74976100	5956554300	1986926800	6151878500

As it was mentioned in the class, the selection sorts is in $\boldsymbol{\theta}(\boldsymbol{n}^2)$ algorithm for all inputs. Fairly, our graph and data from the table shows an approximate exponential growth as well, as the array size increases.

However, when the sorted array's size got larger, the total run-time has a significant decrease, compared to random and almost sorted arrays. Since it has a poor efficiency, it is not advisable to use this algorithm for large input size and for the arrays which are not sorted.

Bubble Sort:



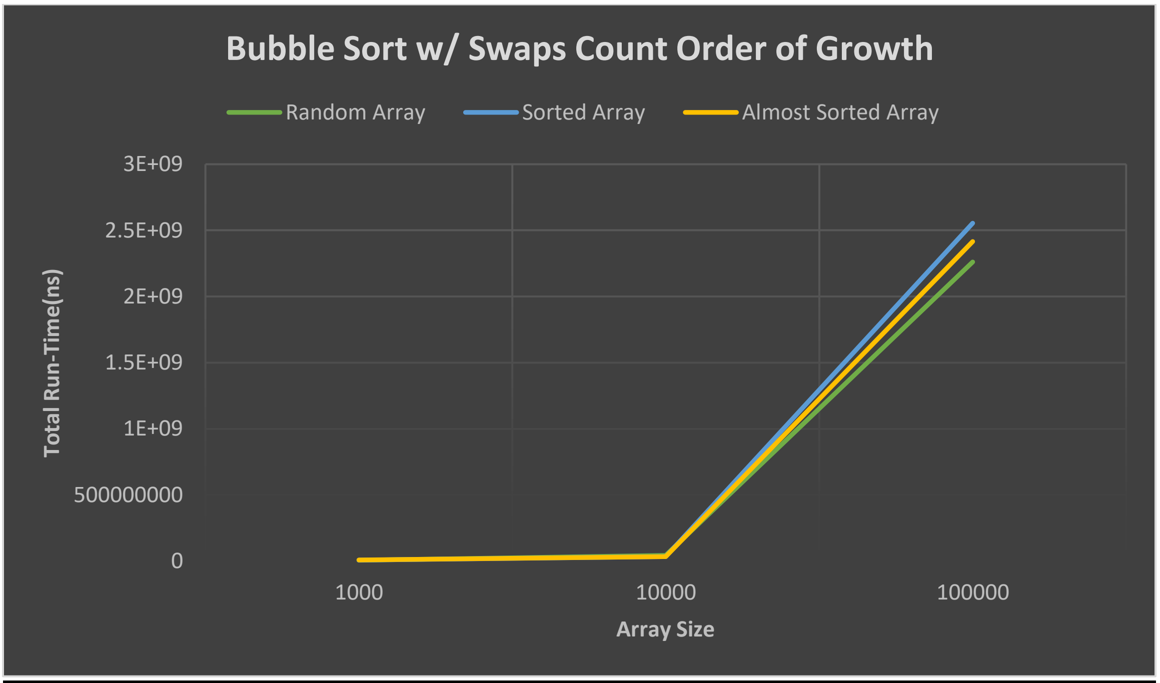
Random	Sorted	Almost Sorted	Random	Sorted	Almost Sorted	Random	Sorted	Almost Sorted
6877500	7050900	7958500	36273400	36451300	34744700	2298005900	2408492500	2407903300

Bubble sort is in $\boldsymbol{\theta}(\boldsymbol{n}^2)$ in its average and worst case for all its input size. We can see from the table and the graph, growth is an approximate exponential growth, similar to the selection sort.

Although it is like the selection sort, the growth of the run-time for the sorted array is same as the random array and almost sorted array. The reason is the comparison operation is the same as (**n-1**).

Same as selection sort, bubble sort is one of the worst efficient sorting algorithms. The input array's nature is not an effective factor, as it seems. It is not advisable to use this algorithm to sort large input sizes.

Bubble Sort w/ Swaps Count:

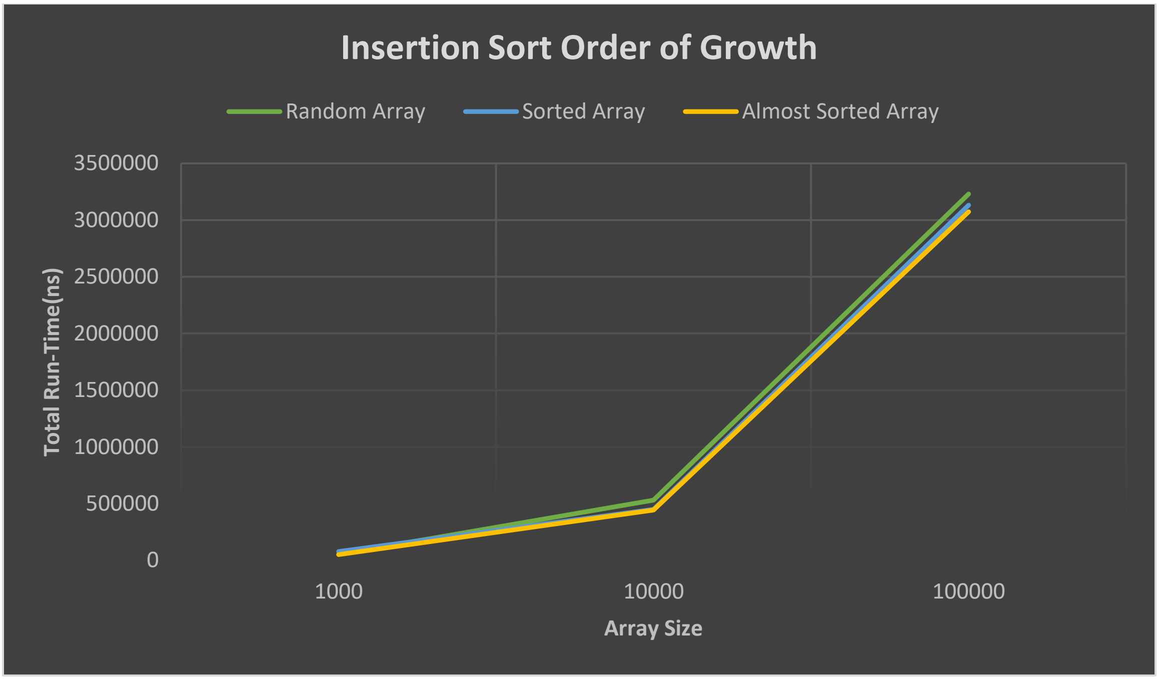


Random	Sorted	Almost Sorted	Random	Sorted	Almost Sorted	Random	Sorted	Almost Sorted
6839300	8063100	9034100	44091700	34867500	34744700	2261315900	2554496500	2416010600

Based on the table and the graph above, since this algorithm is approximately same as the bubble algorithm, it is in $\boldsymbol{\theta(n^2)}$.

Since this algorithm is the same as bubble sort, puts itself as an inferior choice in all sorting algorithms.

Insertion Sort:



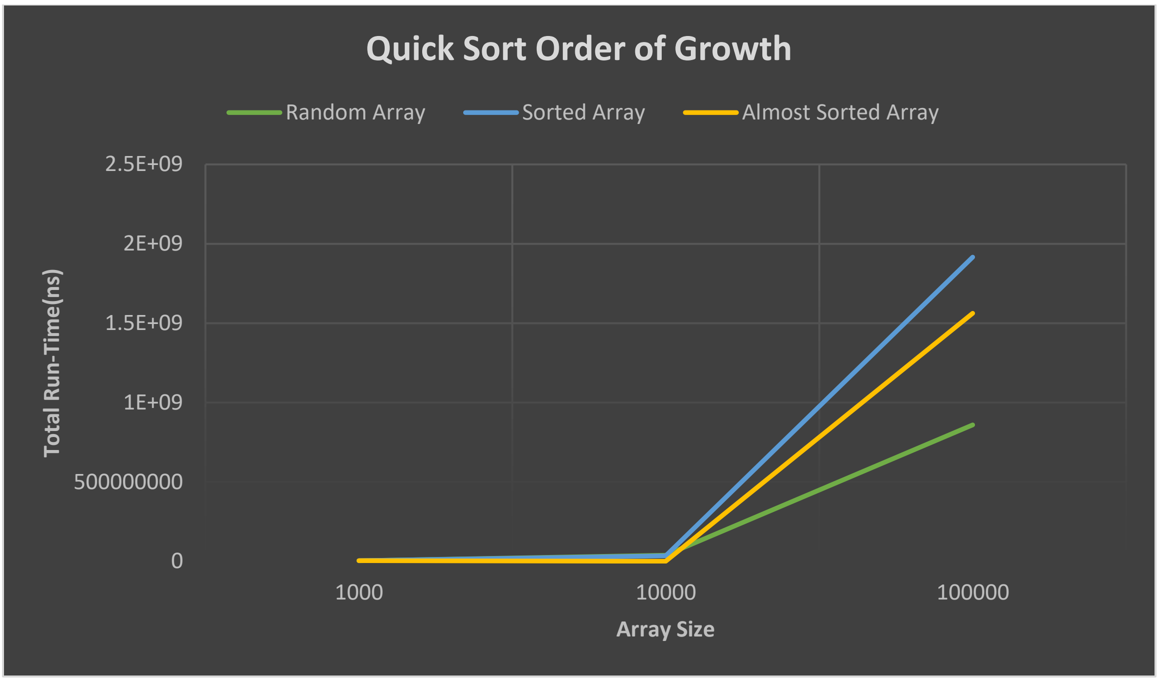
Random	Sorted	Almost Sorted	Random	Sorted	Almost Sorted	Random	Sorted	Almost Sorted
52600	77800	50700	532200	449900	442000	3230400	3131700	3073500

Insertion sort has a basic operation as comparison and the number of comparisons will depend on the nature of the array. In our case, as seen on the table and the graph, our worst case is the random array. However, it is perfectly efficient to conclude the performance of the algorithm by using the data above.

As it was mentioned in my notes, the worst and the average case for this sorting algorithm is in $\boldsymbol{\theta}(\boldsymbol{n}^2)$, where the input array is in decreasing order. The best case is, however, in $\boldsymbol{\theta}(\boldsymbol{n})$, where the input array is non-decreasing order.

Insertion sort is one of the best elementary sorting algorithms, compared to insertion and bubble sort. It is really efficient when the input array is sorted. However, for the large input sizes, it is not advisable to use this sorting algorithm.

Quick Sort:



Random	Sorted	Almost Sorted	Random	Sorted	Almost Sorted	Random	Sorted	Almost Sorted
3672100	4105300	3329200	38388500	35073100	20393300	859371400	1916091000	1562371200

As it is seen in the graph and the table above, unlike the other sorting algorithms as it is mentioned in the book, the quick sort has an interesting fact about its order of growth: When the input array is sorted, the run-time goes to an approximate exponential growth, which is in as the worst case

$$C_{worst}(n) = (n+1) + n + \cdots + 3 = \frac{(n+1)(n+2)}{2} - 3 \in \theta(n^2).$$

The best case for this algorithm is

$$C_{best}(n) = 2C_{best}(n/2) + n \;\; for \;\; n > 1, \qquad C_{best}(1) = 0.$$

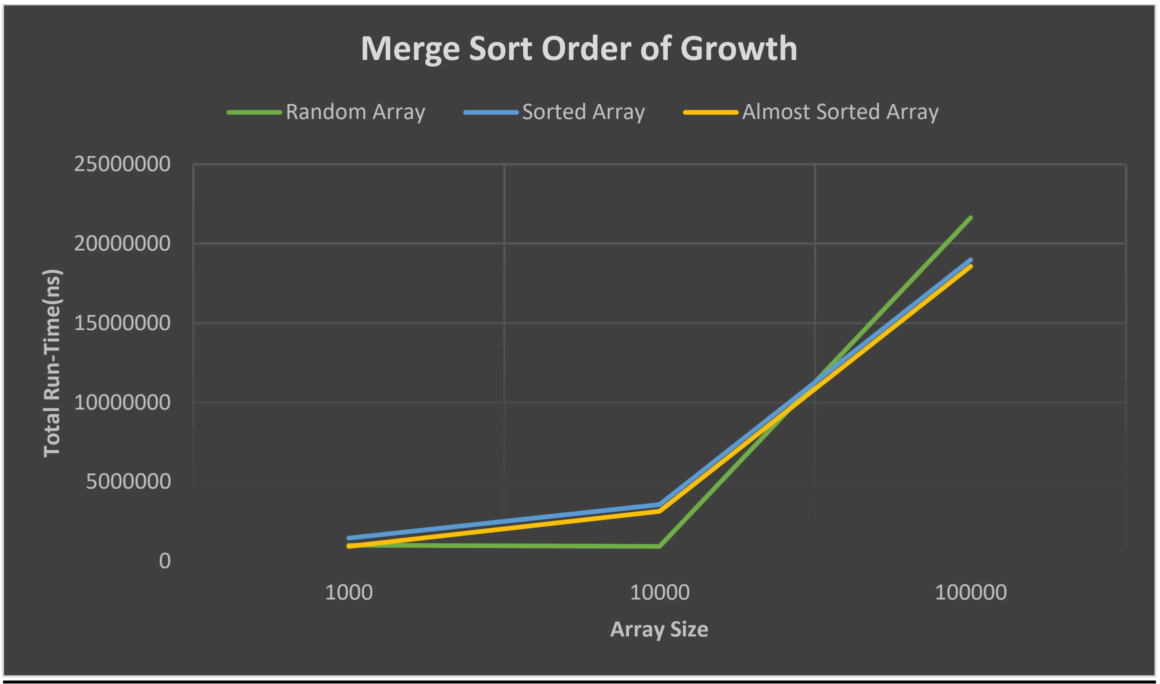
Therefore, $C_{best}(n) \in \theta(nlogn)$.

Surprisingly, its average case is $C_{avg}(n) \in \theta(nlogn)$ as well.

Note: The data above collected by using quick sort algorithm which implemented improved 2-way partition (Hoare’s Partition).

When the quick sort algorithm is improved (better pivot selection or switching to insertion sort when input size is relatively small), it is a really efficient algorithm for the large input sizes. However, it is sensitive to randomly ordered arrays.

Merge Sort:



Random	Sorted	Almost Sorted	Random	Sorted	Almost Sorted	Random	Sorted	Almost Sorted
1009700	1450800	930300	3671200	3562400	3149300	21632000	18979900	18567300

The main part for this sorting algorithm is where two arrays merged. This operation will be repeated. So, it is better to take account of the merge part of this algorithm. It is hard to see on the graph but when we compare the run-times on the table above, it is an efficient algorithm even with the large array with random nature, compared to other algorithms.

As it is mentioned in the book, its worst and average case is $\boldsymbol{C_{worst}(n) \in \theta(nlogn)}$, which makes it the one of the fastest algorithm, even in the worst case.

Merge algorithm is a powerful advanced algorithm that has an advantage over quick sort with its amount of storage requirement. The storage is not important for our case but there is a chance to cause stack overflow problem with quick sort algorithm, if is not implemented carefully. Merge sort has a linear amount of extra storage requirement and fast with large input sizes.