MARMARA UNIVERSITY

FACULTY OF ENGINEERING

COMPUTER ENGINEERING

ANALYSIS OF ALGORITHM
CSE 2046

Name: Student Number:

Mustafa Yanar ***150118048***

Hasan Fatih Başar ***150118015***

Bahadır Alacan ***150118042***

Submitted to: Assistant Prof. Ömer Korçak

Due Date: 16.05.2021

PURPOSE

The aim of this experiment is to decide which sorting algorithm is better by comparing different kind of sorting algorithms by different sequenced inputs and different input sizes.

EXPERIMENT DESIGN

We implement sorting algorithms by using c++ programming language. Firstly we write codes of each sorting algorithm and then we decide our sample inputs. We create our first input with random elements. We created 5 different array which has random elements and do our calculations for every array and take average of this calculations. Also we created different sample inputs for clarify the differences between algorithms. Our inputs are sorted array, reverse sorted array, duplicate, and distinct array. To obtain sorted array we sort one of the random array. For reverse sorted array we do same thing. For duplicate array, we created array which elements are all 1s. Our distinct array is random array which all elements are unique. Lastly we created some special arrays for some sorting algorithms.

We do our calculations by using both physical unit of time and number of basic operations. We use time for compare sorting algorithms with each other and use basic operations for compare same sorting algorithm's different input types. How do we calculate time complexities?

For n: $\frac{n}{2n} = \frac{1}{2}$ ⟹ $\frac{1}{2} \cong \frac{49(50\ input\ size)}{99(100\ input\ size)}$

When input size multiply by 2 number of basic operation should double for n time complexity.

For n^2: $\frac{n^2}{(2n)^2} = \frac{n^2}{4n^2}$ ⟹ $\frac{1}{4} \cong \frac{5048(100\ input\ size)}{20098(200\ input\ size)}$

When input size multiply by 2 number of basic operation should quadruple for n^2 time complexity.

For nlogn: $\frac{nlogn}{2nlog2n}$ basic operation's proportion should be close to this equation

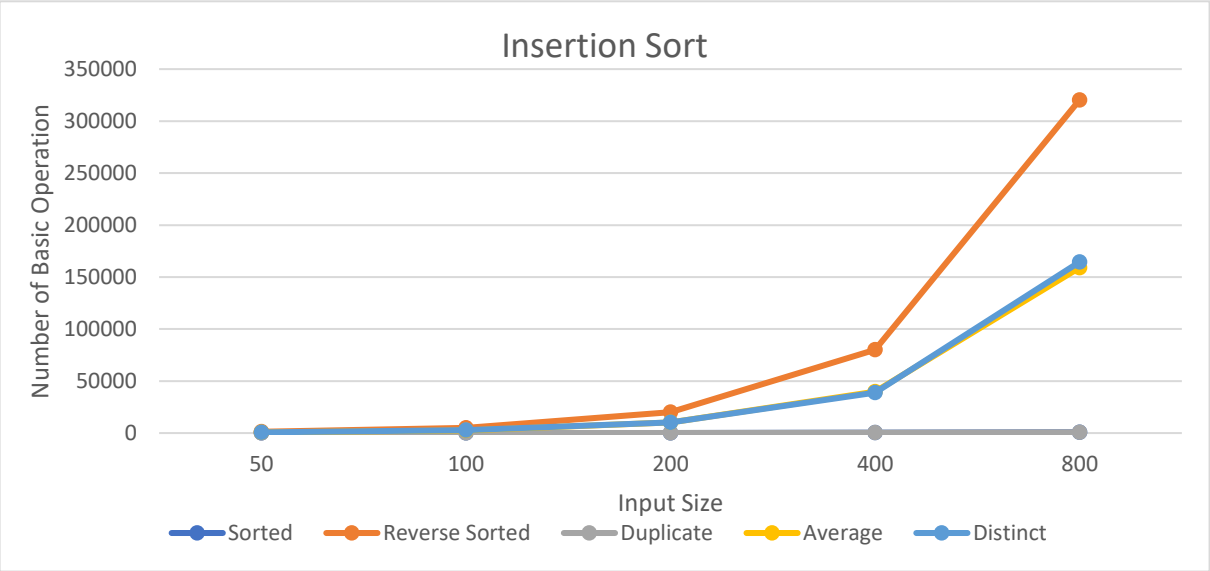
$$\frac{100 * \log 100}{200 * \log 200} \cong \frac{573}{1045}$$

Time calculations:

```
1.  auto start = chrono::high_resolution_clock::now();
2.
3.
4.      ios_base::sync_with_stdio(false);
5.
6.      quickSort(array, 0, size-1);
7.
8.      auto end = chrono::high_resolution_clock::now();
9.
10.     double time_taken =
11.       chrono::duration_cast<chrono::nanoseconds>(end - start).count();
12.
13.     time_taken *= 1e-9;
14.
15.     return time_taken;
```

SORTING ALGORITHMS

a-) Insertion Sort



Insertion Sort / Input Size	50	100	200	400	800
Sorted	49	99	199	399	799
Reverse Sorted	1274	5048	20098	80192	320366
Duplicate	49	99	199	399	799
Average	671	2662	10223	39637	159073
Distinct	697	2793	10388	38790	164538

In this algorithm, assume that we're picking an element in the array and inserting its appropriate place in the array. Our basic operation in this sorting algorithm is comparison. It's shown in black background in the code snippet below.

```
1.  for (i = 1; i < n; i++){
2.          insertionSortCounter++;
3.          key = arr[i];
4.          j = i - 1;
5.          while (j >= 0 && arr[j] > key)
6.          {
7.              arr[j + 1] = arr[j];
8.              j = j - 1;
9.              insertionSortCounter++;
10.         }
11.         arr[j + 1] = key;
12. }
```

The best scenario for insertion sort is a sorted array or every elements are the same in the array. We have linear time complexity which is **O(n)**.

$$\frac{n}{2n} = \frac{1}{2} \quad \Rightarrow \quad \frac{1}{2} \cong \frac{49(50\ input\ size)}{99(100\ input\ size)}$$

The worst scenario for insertion sort is reversed sorted array. We must go into the inner loop for every each element in reversed sorted array. We will compare the element with the each other elements and shift it. We have a quadratic time complexity which is **O(n^2)** for worst case.

$$\frac{n^2}{(2n)^2} = \frac{n^2}{4n^2} \quad \Rightarrow \quad \frac{1}{4} \cong \frac{5048(100\ input\ size)}{20098(200\ input\ size)}$$

Worst Case ➔ $O(n^2)$

According to our results for each input size as we calculated, number of basic operation increases quadraticly. For example, we have 50 sized array and we have 100 sized array. In the worst case, we can observe that number of comparisons increased nearly 4 times between these two arrays. So we can say that O($\boldsymbol{n^2}$) time complexity is correct for the worst case of insertion sort.

$$\frac{n^2}{(2n)^2} = \frac{n^2}{4n^2} \quad \Longrightarrow \quad \frac{1}{4} \cong \frac{1274}{5048}$$

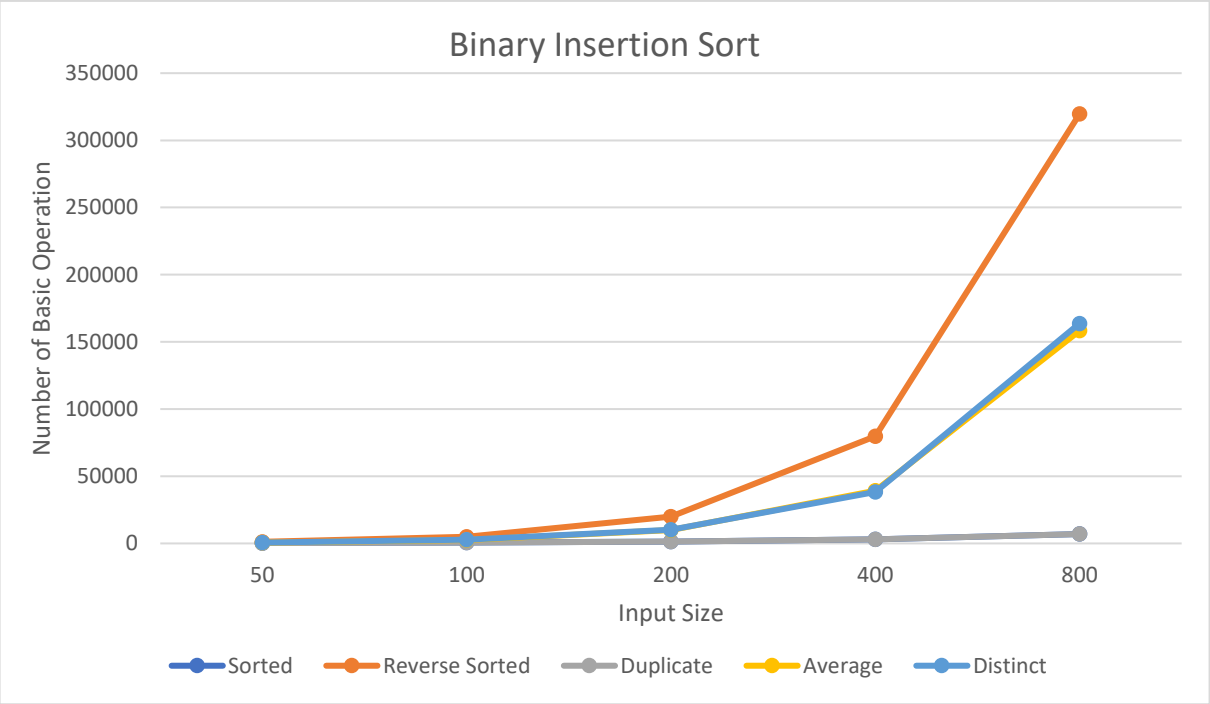
The average case have quadratic time complexity like the worst scenario. That's why this sorting algorithm is not used for large arrays. However, as we mentioned in the class, Insertion sort is one of the fastest algorithm to sort small arrays, even faster than mergesort, quicksort etc. Average case and distinct case are same.

$$\frac{n^2}{(2n)^2} = \frac{n^2}{4n^2} \quad \Longrightarrow \quad \frac{1}{4} \cong \frac{2662(100\ input\ size)}{10223(200\ input\ size)}$$

Average Case ➔ $O(n^2)$

We can observe that our empirical results according to calculations meets with the theoretical results. They are really close to each other.

b-) **Binary** **Insertion** **Sort**



Binary Inserton Sort / Input Size	50	100	200	400	800
Sorted	237	573	1345	3089	6977
Reverse Sorted	1225	4919	19899	79793	319567
Duplicate	237	573	1345	3089	6977
Average	622	2563	10024	39238	158274
Distinct	648	2694	10189	38391	163739

Binary Insertion Sort is a modified version of Insertion Sort. This time, We can find the appropriate place for every each element in array by using binary search. However this algorithm still have a worst scenario like Insertion sort which is O(n^2), due to shiftings required for each insertion.

```
1.  int binarySearch(int arr[], int temp, int start,
     int end)
2.  {
3.      while (start <= end) {
4.              int mid = start + (end - start) / 2;
5.              if (temp < arr[mid]) {
6.                  binarySearchCounter++;
7.                  end = mid - 1;
8.              } else {
9.                  binarySearchCounter++;
10.                 start = mid + 1;
11.             }
12.         }
13.     return start;
14. }
```

```
1.  for (i = 1; i < n; ++i)
2.      {
3.          j = i - 1;
4.          selected = a[i];
5.          loc = binarySearch(a, selected, 0, j);
6.          while (j >= loc)
7.          {
8.              binaryInsertionCounter++;
9.              a[j + 1] = a[j];
10.             j--;
11.         }
12.         a[j + 1] = selected;
13.     }
```

As seen in the above, we have two different counters in binary insertion sort. Why do we have two different counters? Since our basic operations differs between different arrays, we should calculate counters separately for the different basic operations. One of them is for swap operation and the other one is for comparison operation. We take as counter which is bigger.

In the average case, we still have quadratic time complexity **O(n^2)**. Because our basic operation is not comparison for average case. Its swapping.

$$\frac{n^2}{(2n)^2} = \frac{n^2}{4n^2} \quad \Rightarrow \quad \frac{1}{4} \cong \frac{2563(100\ input\ size)}{10024(200\ input\ size)}$$

Average Case ➔ $O(n^2)$

Average case and distinct case are same.

In the best case, Since the binary insertion sort does the binary search which has time complexity such as $O(logn)$, its best case time complexity is $\mathbf{O}(\boldsymbol{nlogn})$.

According to our results for each input size as we calculated, number of basic operation increases n times logarithmly. For example, we have 100 sized array and 200 sized array for best case, such as sorted array. We do 573 times basic operation for 100 sized array and 1345 times basic operation for 200 sized array. If we want to proportion between these comparison numbers, we would find result such as 0.42 which is almost correct for these equations

$$\frac{nlogn}{2nlog2n} \quad \Rightarrow \quad \frac{100 * \log 100}{200 * \log 200} \cong \frac{573}{1045}$$

or

$$\frac{nlogn}{4nlog4n} \quad \Rightarrow \quad \frac{200 * \log 200}{800 * \log 800} \cong \frac{10189}{163739}$$

So we can say that $\mathbf{O}(\boldsymbol{nlogn})$ time complexity meets with the best case of binary insertion sort.

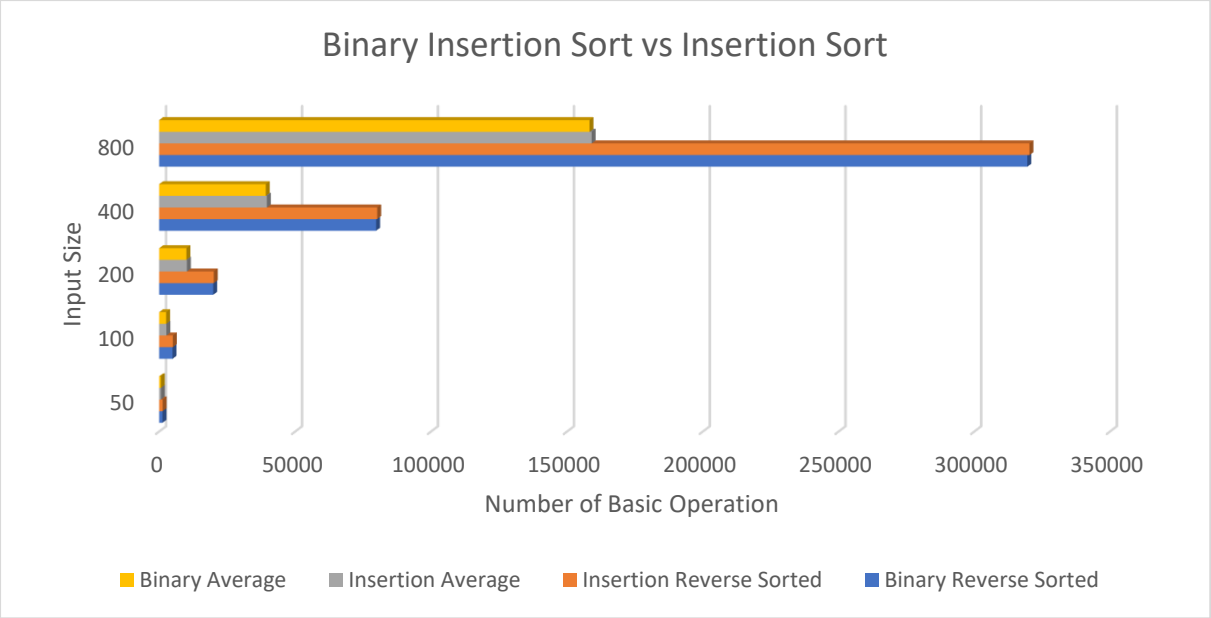
The worst scenario for binary insertion sort is reversed sorted array. It is most like insertion sort, we don't reduce number of basic operation in the worst case. Our basic operation is swapping for the worst case. Therefore, We have a quadratic time complexity which is **O(n^2)**.

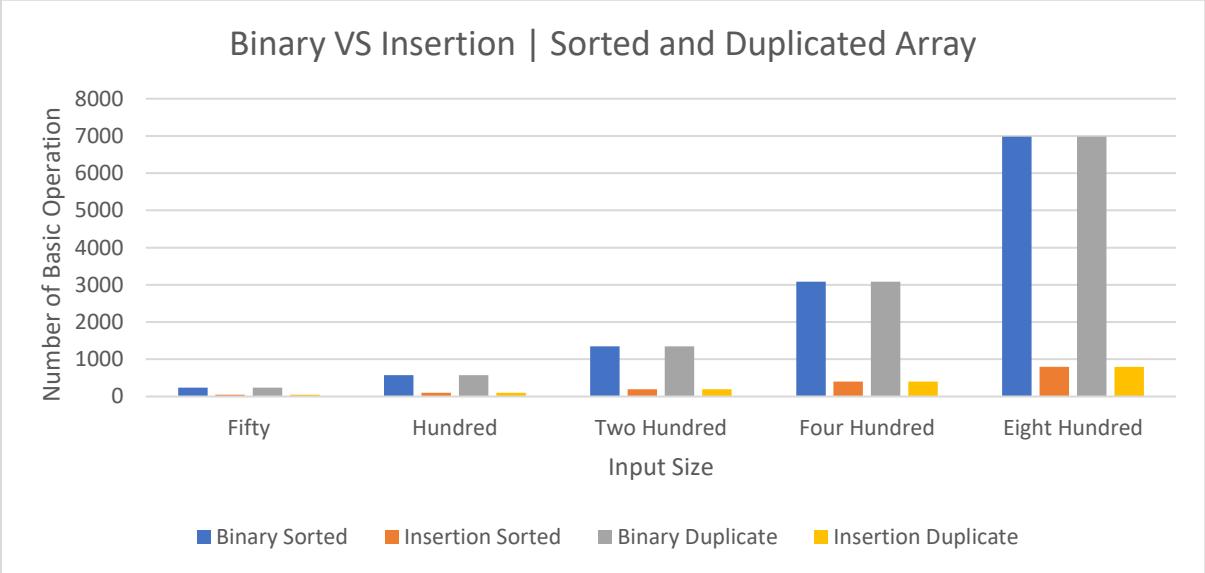
$$\frac{n^2}{(2n)^2} = \frac{n^2}{4n^2} \quad \Rightarrow \quad \frac{1}{4} \cong \frac{19899(200\ input\ size)}{79793(400\ input\ size)}$$

Worst Case ➔ $O(n^2)$

As a result we can observe that our empirical results according to calculations meets with the theoretical results. They are almost same between each other.

Let’s compare Binary Insertion Sort vs Insertion Sort…





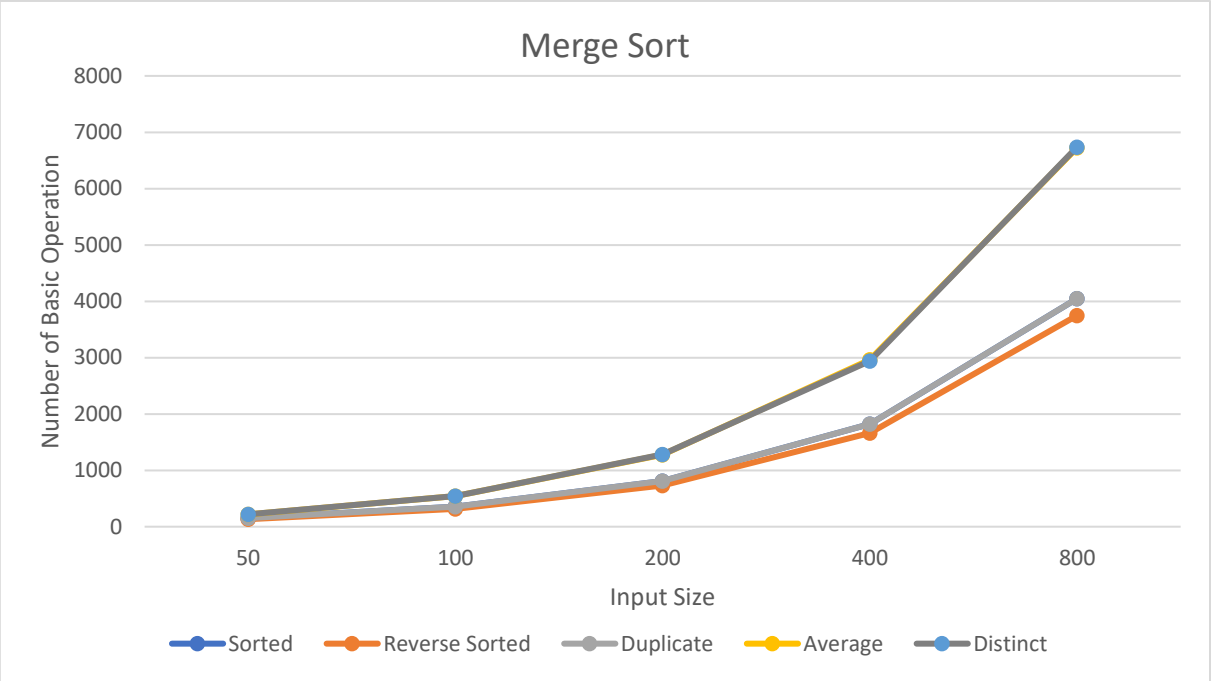
Insertion Sort vs Binary Insertion Sort:

As seen in the chart above, Although they are almost same in the worst scenarios, they have different types of basic operation. Insertion sort’s basic operation is comparison but binary insertion sort’s basic operation is swapping operation. However they have the same time complexity for average cases, number of basic operation differs very small between insertion sort and binary insertion sort. But in binary insertion sort we do less comparison than insertion sort. Therefore binary insertion sort is a little more faster in average and worst cases.

We observe that insertion sort is not practiced as much as binary insertion sort for the worst scenarios. How do we say that? We can look at difference between the number of basic operations for the worst case. However, even we have sorted arrays, binary insertion sort does much more basic

operations than insertion sort. So we can say that, insertion sort is better than binary insertion sort for the best case scenarios.

c-) Merge Sort



Merge Sort / Input Size	50	100	200	400	800
Sorted	153	356	812	1824	4048
Reverse Sorted	133	316	733	1665	3748
Duplicate	153	356	812	1824	4048
Average	219	542	1276	2965	6725
Distinct	220	542	1282	2940	6738

Merge sort is one of type of divide and conquer sorting methods. It can be implemented either recursively or bottom-up method.

As seen in the code snippet below, when we do comparison, since our basic operation is comparison, we increase the counter by one for each comparison.

```
1.  while (i < n1 && j < n2) {
2.          if (L[i] <= R[j]) {
3.              mergeCounter++;
4.              arr[k] = L[i];
5.              i++;
6.          }
7.          else {
8.              mergeCounter++;
9.              arr[k] = R[j];
10.             j++;
11.         }
12.         k++;
```

We can see that best cases for merge sort are sorted array, reverse sorted array or duplicated array. Their number of basic operasions are really close to each other. For example we have 200 sized and 400 sized arrays, we do 812 times number of basic operation for 200 sized array and we do 1824 times number of basic operation for 400 sized array. If we proportion these two number of comparisons, we find nearly 0,44 and this result nearly meets with this calculation:

$$\frac{nlogn}{2nlog2n} \longrightarrow \frac{200 * \log 200}{400 * \log 400} \cong \frac{733}{1665}$$

or

$$\frac{nlogn}{4nlog4n} \longrightarrow \frac{200 * \log 200}{800 * \log 800} \cong \frac{733}{3748}$$

So we can clearly observe:

Best Case ➔ $O(nlogn)$

The average case for merge sort is random array which is called as average in the table. In average case we have still O(nlogn) time complexity. Distinct and average cases are same.

$$\frac{nlogn}{2nlog2n} \longrightarrow \frac{200 * \log 200}{400 * \log 400} \cong \frac{1276}{2965}$$

or

$$\frac{nlogn}{4nlog4n} \longrightarrow \frac{200 * \log 200}{800 * \log 800} \cong \frac{1276}{6738}$$

So we can clearly observe:

Average Case ➔ $O(nlogn)$

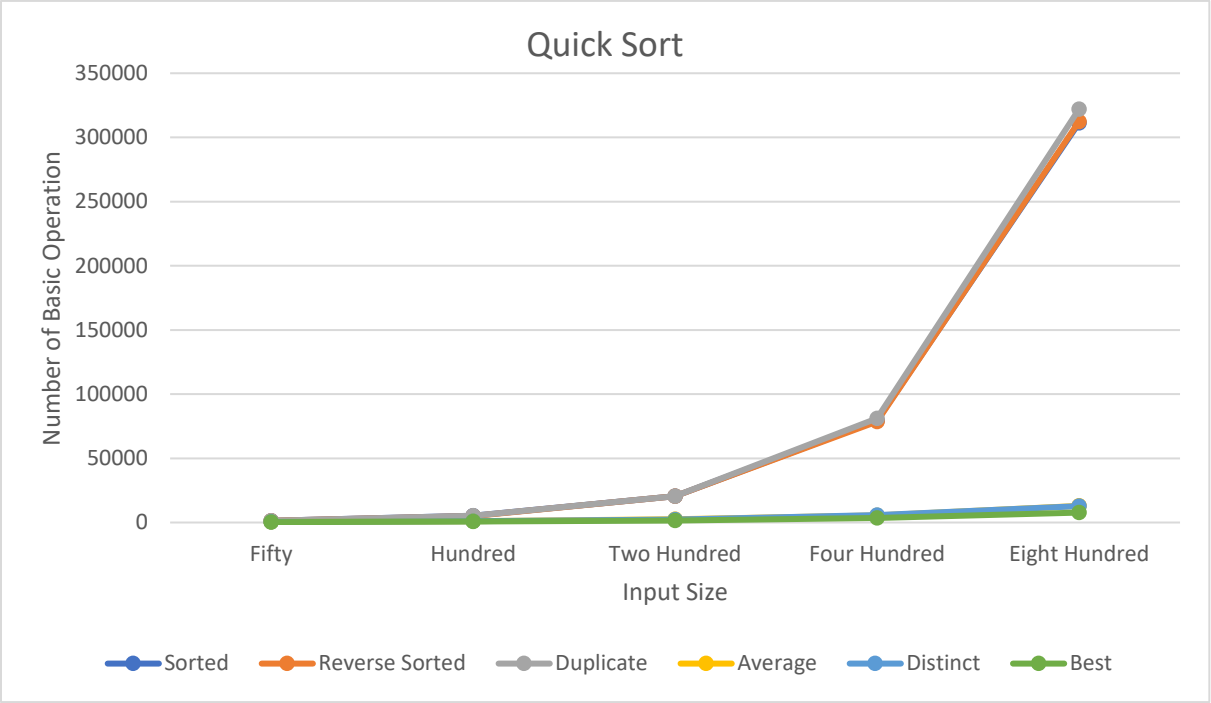
In worst case we have same time complexity as average and best case O(nlogn).

Worst Case ➔ $O(nlogn)$

We can observe that our empirical results according to calculations meets with the theoretical results. They are almost same between each other.

As a result we have same time complexity for every cases in merge sort.

d-) Quick Sort



Quick Sort / Input Size	Fifty	Hundred	Two Hundred	Four Hundred	Eight Hundred
Sorted	1372	5158	20404	79224	311266
Reverse Sorted	1372	5099	20497	78513	312405
Duplicate	1372	5247	20497	80997	321997
Average	420	1060	2485	5472	12914
Distinct	432	1010	2314	5745	12753
Best	290	673	1538	3467	7724

Like Merge sort, Quick sort is a divide and conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. In this experiment we choose first element as a pivot.

As seen in the code snippet below, when we do comparison, since our basic operation is comparison, we increase the counter by one for each comparison.

```
1.  In Partition Method:
2.  while(start<end){
3.          quickSortCounter++;
4.          while(array[start]<= pivot ){
5.              if(start <= high){
6.              quickSortCounter++;
7.              start++;
8.              }else{
9.                  break;
10.             }
11.         }
12.         quickSortCounter++;
13.         while(array[end] > pivot){
14.             if(end >= low){
15.             quickSortCounter++;
16.             end--;
17.             }else{
18.                 break;
19.             }
```

```
20.         }
21.         if(start < end){
22.             swap(array[start], array[end]);
23.         }
24.     }
25. #NOTE: WE USE THE SAME PARTITION METHOD FOR MEDIAN TOO#
```

In the QuickSort Algorithm, we see a huge difference between duplicate/sorted array/reverse sorted array and the others. These cases are worst case of this algorithm. The reason of this, the algorithm always check itself by decreasing one although the array is already sorted. Therefore, the algorithm make maximum comparison.

Reverse sorted: $\frac{n^2}{(2n)^2} = \frac{n^2}{4n^2}$ $\frac{1}{4} \cong$ ➡ $\frac{5099(100\ input\ size)}{20497(200\ input\ size)}$

Sorted: $\frac{n^2}{(2n)^2} = \frac{n^2}{4n^2}$ ➡ $\frac{1}{4} \cong \frac{5158(100\ input\ size)}{20404(200\ input\ size)}$

Duplicate: : $\frac{n^2}{(2n)^2} = \frac{n^2}{4n^2}$ ➡ $\frac{1}{4} \cong \frac{5247(100\ input\ size)}{20497(200\ input\ size)}$

Worst Case ➔ $O(n^2)$

As you can see from the table, when the array contains random variables, the algorithm might divide itself by any number (the best situation is to be half of total length). Theoretically in average, quick sort has time complexity O(nlogn). Also it has same time complexity for distinct array. Because the distribution of elements in distinct array is random. We cannot estimate where the median element is.

$\frac{nlogn}{2nlog2n}$ ➡ $\frac{200 * \log 200}{400 * \log 400} \cong \frac{1010}{2314}$

or

$\frac{nlogn}{4nlog4n}$ ➡ $\frac{200 * \log 200}{800 * \log 800} \cong \frac{1010}{5745}$

Average Case ➔ $O(nlogn)$

There is a best time complexity for the quicksort algorithm. If each calling has a median element as the pivot (first element of array), then the algorithm is exactly divided by half of it. We generate new input which is best case for quick sort. Best case time complexity O(nlogn) as well. If we have an array whose elements are 4,1,3,2,6,5,7; it is the best case for the quicksort algorithm. Because the first element of each recursion part is already the median of the part, that causes the distribution of parts to divide into two equal-sized pieces.

$\frac{nlogn}{2nlog2n}$ ➡ $\frac{50 * \log 50}{100 * \log 100} \cong \frac{290}{673}$

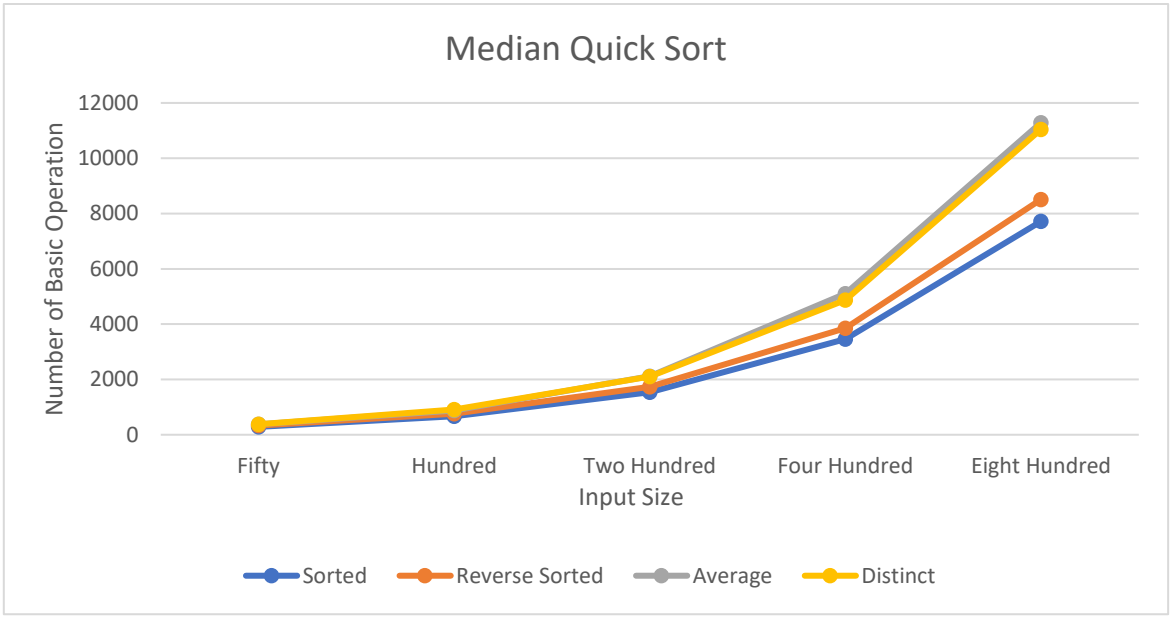
or

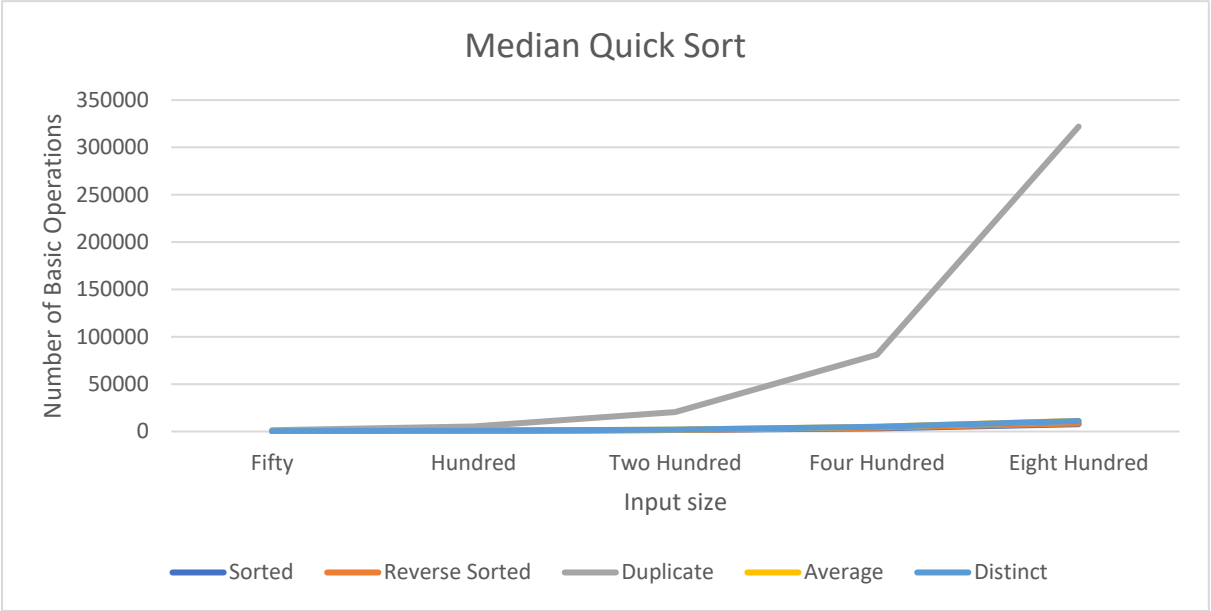
$\frac{nlogn}{4nlog4n}$ ➡ $\frac{200 * \log 200}{800 * \log 800} \cong \frac{1538}{7724}$

Best Case ➔ $O(nlogn)$

As we can see, our experimental and theoretical results are almost equal for quicksort algorithm.

e-) Quick-sort with median-of-three pivot selection





Median Quick Sort / Input Size	Fifty	Hundred	Two Hundred	Four Hundred	Eight Hundred
Sorted	286	669	1538	3463	7717
Reverse Sorted	332	765	1734	3856	8512
Duplicate	1372	5247	20497	80997	321997
Average	377	876	2115	5108	11288
Distinct	381	918	2109	4869	11047

In median of three quick sort we compare the first, median and the last element and find the middle value between these three elements. We choose it as a pivot. In this way we decrease number of comparisons.

We find basic operation by calculating with same method in the common quicksort.

It is pretty like a quicksort algorithm But this algorithm avoids running at maximum time complexity for sorted array. In the Median of Three QuickSort Algorithm, we see a huge difference between duplicate array and the others . Its worst case of this algorithm. This part is same as the normal quick sort. Pivot does not change because of that the median of three (first, middle, last) is already the first element, three elements are the same. Duplicate array elements are assumed as a sorted array. The algorithm always checks itself by decreasing one although the array is already sorted. Therefore, the algorithm makes a maximum comparison (one by one). It does basic operation n over 2 times.

$$\frac{n^2}{(2n)^2} = \frac{n^2}{4n^2} \quad \Rightarrow \quad \frac{1}{4} \cong \frac{1372(50\ input\ siz)}{5247(100\ input\ size)}$$

Worst Case ➔ $O(n^2)$

As you can see from the table, sorted has less comparison among all. It adjust itself to be best division of algorithm through take median of middle,first and last as pivot. That's why number of comparison of median of three quick sort for sorted array is exactly same with divide by half.

$$\frac{nlogn}{2nlog2n} \quad \Rightarrow \quad \frac{50 * \log 50}{100 * \log 100} \cong \frac{286}{669}$$

or

$$\frac{nlogn}{4nlog4n} \quad \Rightarrow \quad \frac{200 * \log 200}{800 * \log 800} \cong \frac{918}{4869}$$

Best Case ➔ $O(nlogn)$

When the array contains random variables, the algorithm might divide itself by any number (the best situation is to be half of total length). Theoretically in average, median of three quick sort has time complexity O(nlogn). It is better than the common quicksort because of that the algorithm tends to divide itself evenly distributed. If there is a better pivot selection, it adjusts for the best state. However, it does not always divide by half of it. Therefore, in average number of comparison is bigger than sorted array.

$$\frac{nlogn}{2nlog2n} \quad \Rightarrow \quad \frac{200 * \log 200}{400 * \log 400} \cong \frac{765}{1734}$$

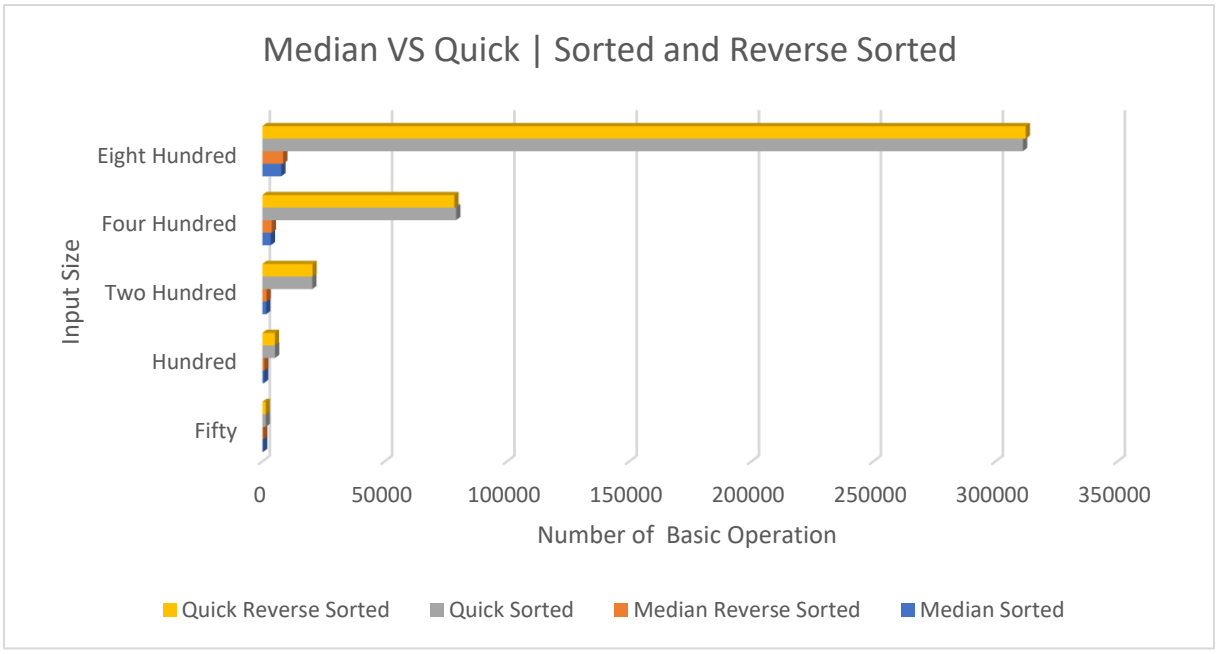
or

$$\frac{nlogn}{4nlog4n} \quad \Rightarrow \quad \frac{200 * \log 200}{800 * \log 800} \cong \frac{765}{3586}$$

Average Case ➔ $O(nlogn)$

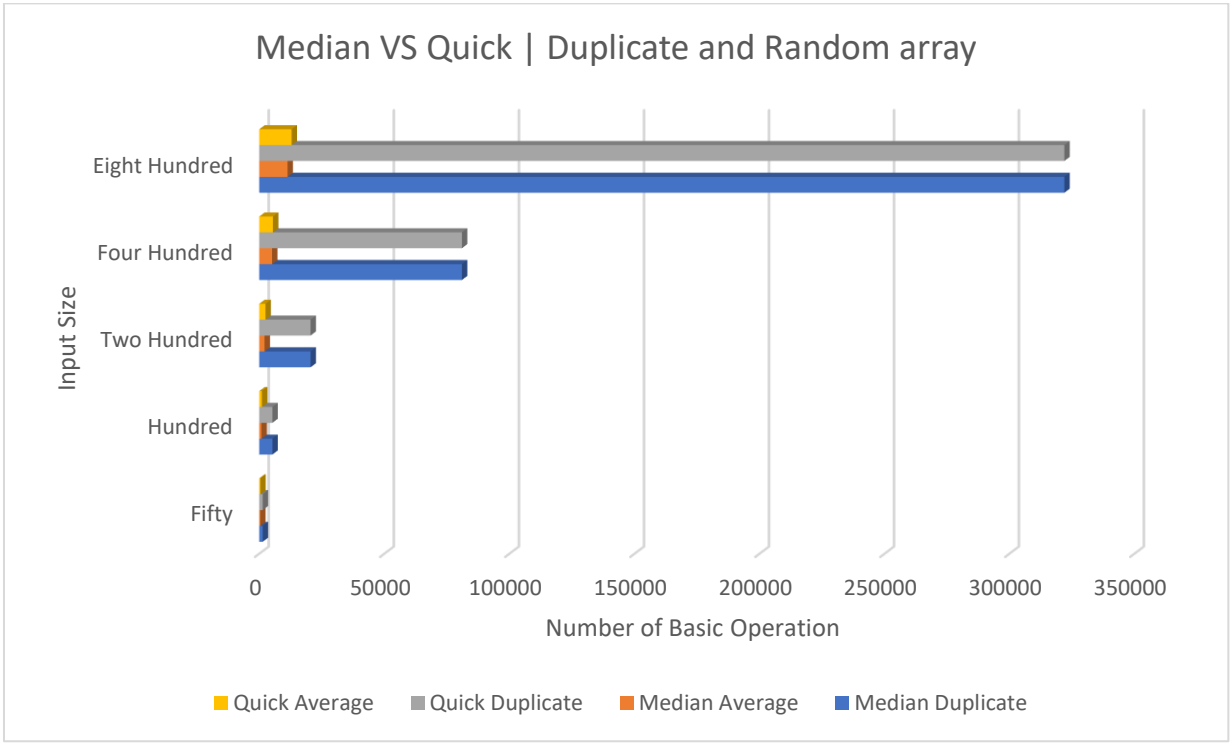
As we can see, our experimental and theoretical results are almost equal for median of three quicksort algorithm.

Let's compare Quicksort vs Median Of Three QuickSort…



As seen in the chart above, the difference between the two graphs is too much. And it is increasing more and more. Actually, this difference is only because of pivot selection. Pivot divides the array into two parts.

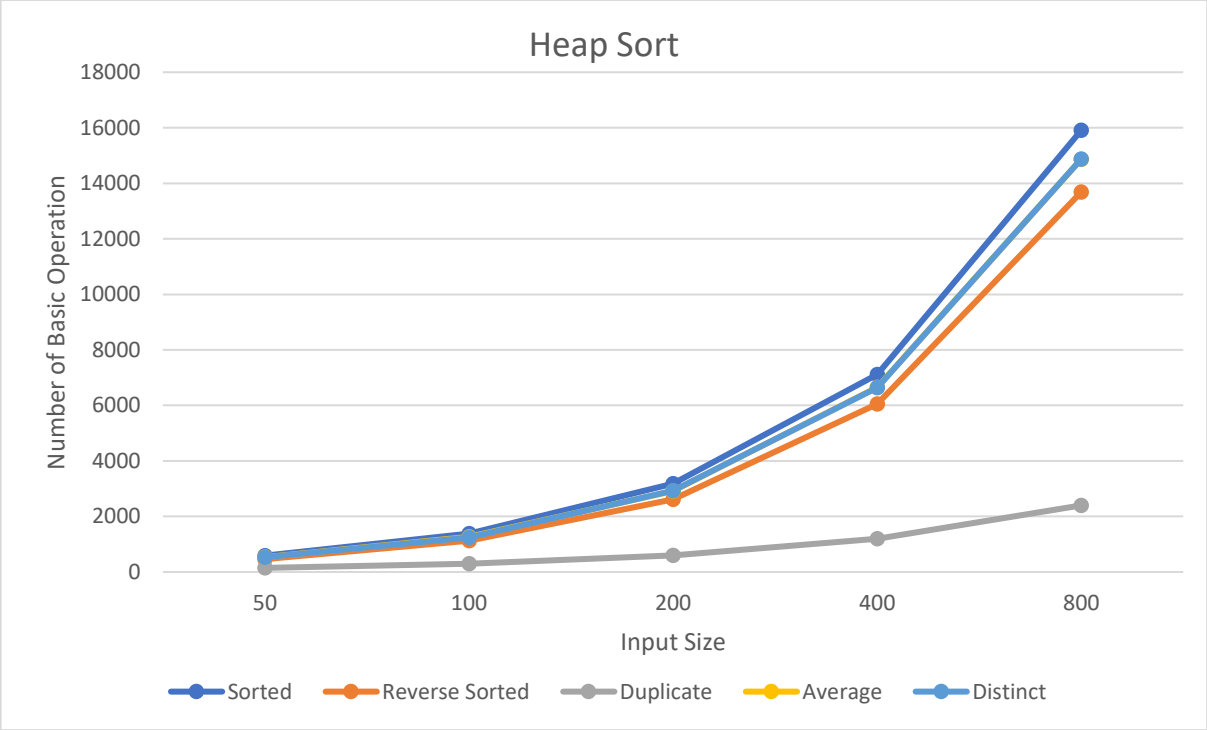
In the median of three quicksort, the pivot is selected as the best choice (median) for sorted array and reverse sorted array. Therefore the array is handled as two equal-sized subarrays. However, the pivot is selected minimum or maximum value in the common quicksort. That leads to dividing two parts which are one of the parts that have only 1 element and that the other part has all elements except that one element.



As seen in the chart above, If the array is a duplicated array, count of calling the basic operation is same for both algorithm. Because median of three only makes a pivot selection to run by better time

complexity. If the median element between the first, middle and last element is already the first element, then there is no distinction for quicksort and median of three quicksort. Both of them run in the same way. We can say that in average median of three quick sort is a little better than normal quick sort because it divide it a bit better than normal quick sort.

f-) Heap Sort



Heap Sort / Input Size	50	100	200	400	800
Sorted	590	1386	3178	7112	15904
Reverse Sorted	464	1132	2612	6050	13680
Duplicate	148	298	598	1198	2398
Average	535	1261	2919	6646	14864
Distinct	530	1248	2922	6638	14870

Heap sort is a comparison based sorting technique based on Binary Heap data structure. In this algorithm we build max heap and swap the root element with last element of heap. By repeating this we get a sorted array.

We find basic operation by calculating comparison of element with its child.

```
1.  void heapify(int arr[], int n, int i)
2.  {
3.      int largest = i;
4.      int l = 2 * i + 1;
5.      int r = 2 * i + 2;
6.      heapCounter +=2;
7.      if (l < n && arr[l] > arr[largest]) largest = l;
8.      if (r < n && arr[r] > arr[largest]) largest = r;
9.
10.     if (largest != i) {
11.         swap(arr[i], arr[largest]);
```

```
12.         heapify(arr, n, largest);
13.     }
14. }
```

The best case for heap sort is every elements same in the array. Because while constructing heap there is minimum comparison for duplicate array. Theoretically it must be linear time complexity O(n). If we compare with our empirical results:

Duplicate array:

$$\frac{n}{2n} = \frac{1}{2} \quad \Rightarrow \quad \frac{1}{2} \cong \frac{148(50\ input\ size)}{298(100\ input\ size)}$$

$$\frac{n}{4n} = \frac{1}{4} \quad \Rightarrow \quad \frac{1}{4} \cong \frac{148(50\ input\ size)}{598(200\ input\ size)}$$

Best Case ➔ $O(n)$

For heap sort, since we created max heap at first, sorted array has the more comparison than average case and reverse sorted array has the less comparison than average case. But they are both same time complexity O(nlogn). If we compare with our empirical results:

Sorted array: $\frac{nlogn}{2nlog2n} \Rightarrow \frac{200 * \log 200}{400 * \log 400} \cong \frac{3178}{7112}$

Reverse sorted array: $\frac{nlogn}{2nlog2n} \Rightarrow \frac{200 * \log 200}{400 * \log 400} \cong \frac{2612}{6050}$

Worst Case ➔ $O(nlogn)$

In average case it has time complexity O(nlogn). Average case and distinct case are same. If we compare with our empirical results:

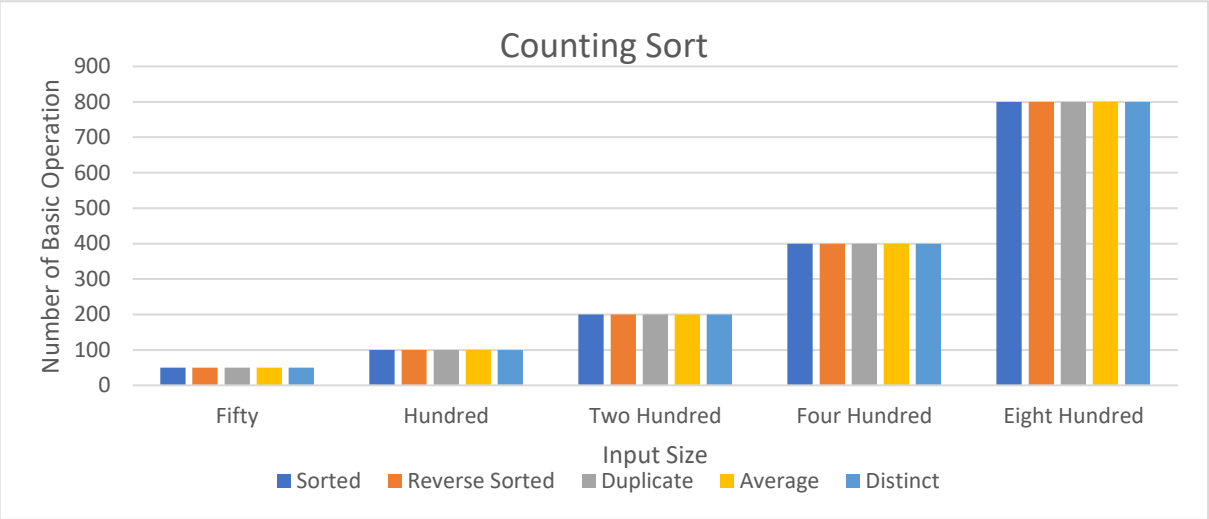
$$\frac{nlogn}{2nlog2n} \quad \frac{400 * \log 400}{800 * \log 800} \cong \Rightarrow \quad \frac{6646}{14864}$$

Average Case ➔ $O(nlogn)$

As we can see, our experimental and theoretical results are almost equal.

As a result, heap sort has same time complexity for worst and average case which is O(nlogn) and has best case time complexity O(n). Its an efficient sorting algorithm.

g-) Counting Sort



Counting Sort / Input Size	Fifty	Hundred	Two Hundred	Four Hundred	Eight Hundred
Sorted	50	100	200	400	800
Reverse Sorted	50	100	200	400	800
Duplicate	50	100	200	400	800
Average	50	100	200	400	800
Distinct	50	100	200	400	800

Counting sort is a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (kind of hashing). Then doing some arithmetic to calculate the position of each object in the output sequence.

We find the basic operation by size of array but actually there is no basic operation as swapping or comparison. So counting sort has not basic operation. It just depends on array size and range of values.

```
1.  for(int i=0; i<size; i++){
2.          countSortMax++;
3.          counts[arr[i]]++;
4.      }
5.      int j=0;
6.      for(int i=0; i<size; i++){
7.        if(j<10001){
8.        while(counts[j] == 0 ){
9.              if(j<10000) j++;
10.             else break;
11.         }
12.         if(j<10001){
13.             arr[i] = j;
14.             j++;
```

In counting sort, actually there is no basic operation. Time complexity is O(n+k) for every case which n is input size and k is the range of values. However in our experiment our range is constant which is 10000. Therefore time complexity is O(n) for this experiment.

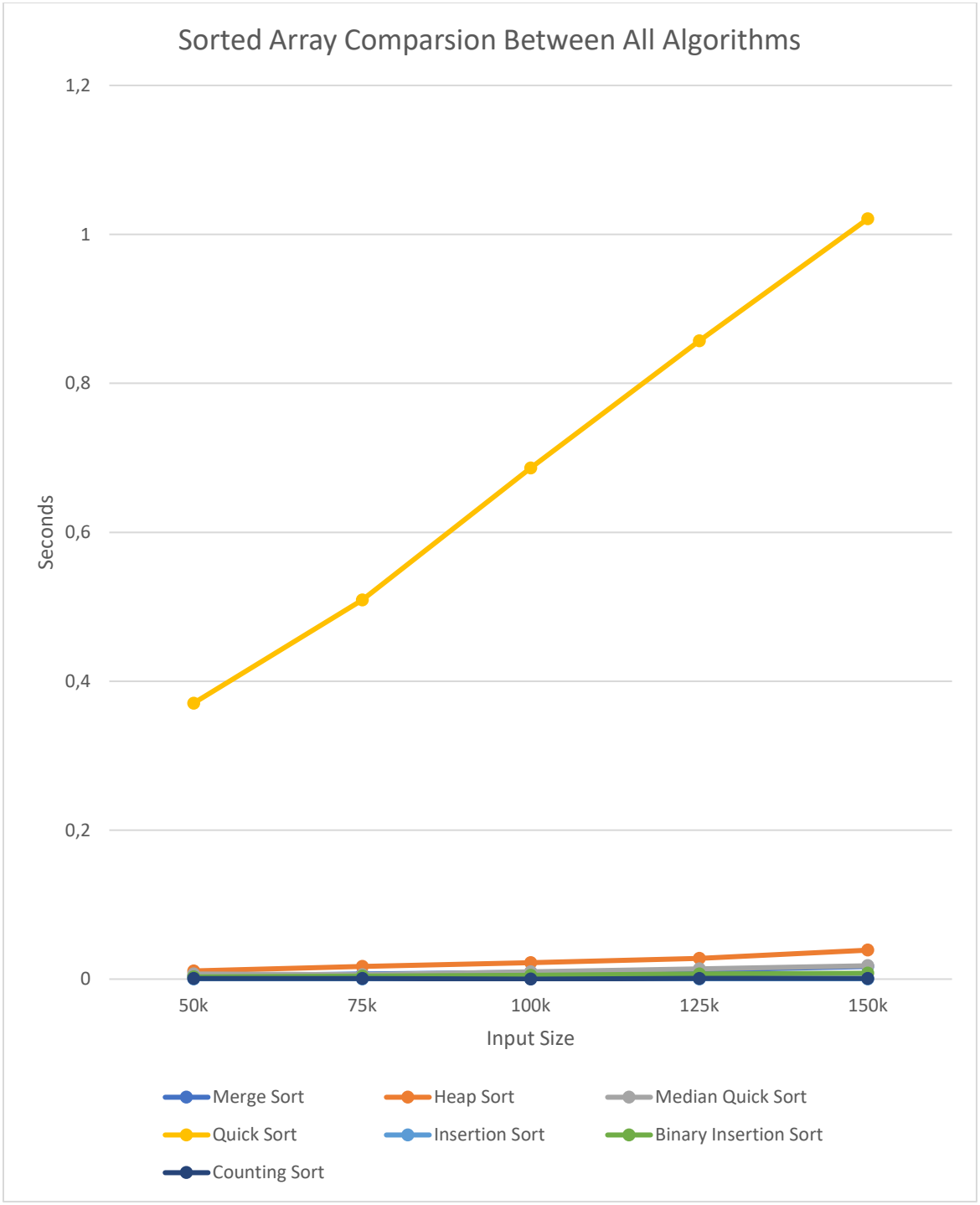
There is not best case or worst case for this sorting algorithm. They are all same. Counting sort depends on input size and range of values.

As a result in our experiment counting sort is very efficient algorithm. Because our range is maximum 10000. But if range was a large number, counting sort could be inefficient algorithm.

COMPARING SORTING ALGORITHMS FOR SAME ARRAY

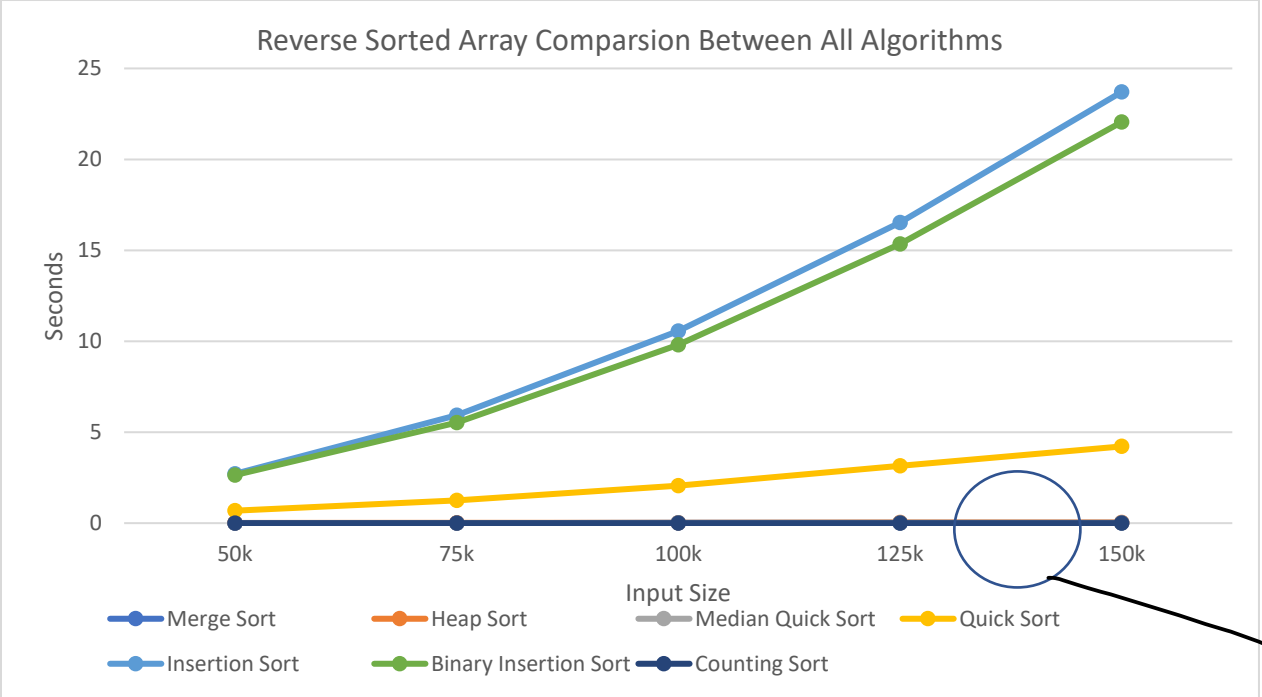
Our input sizes are 50000, 75000, …… ,150000.

a-) Sorted Array



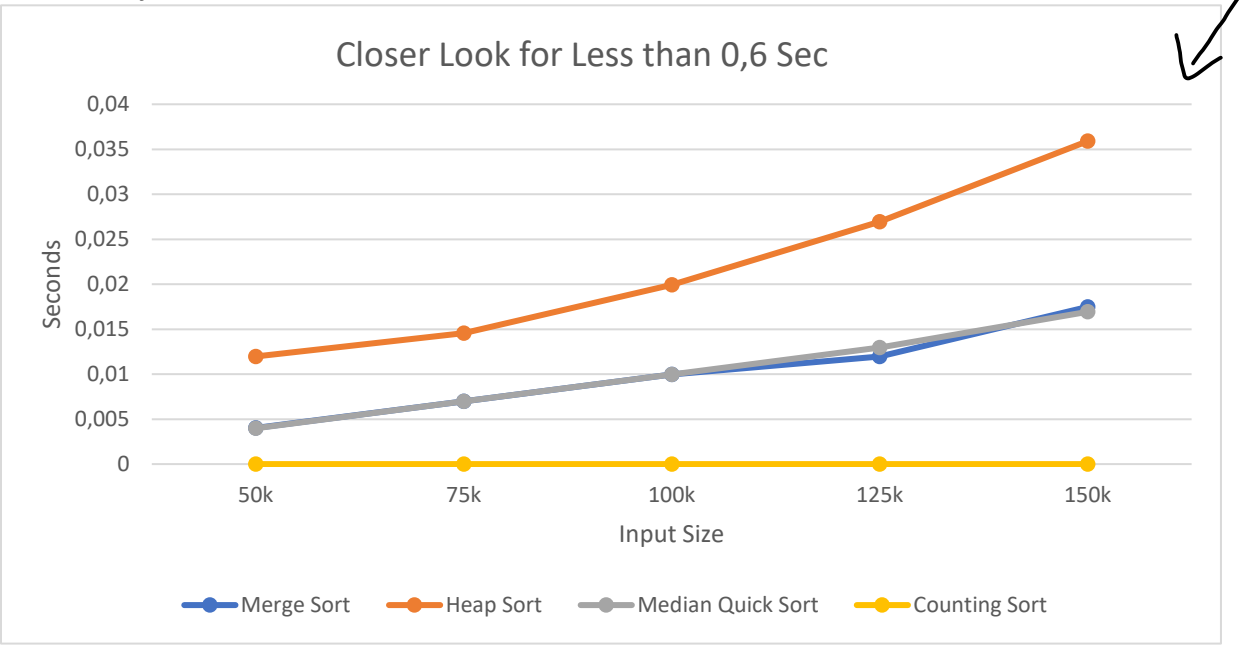
It is obvious in the chart above, quick sort is slowest algorithm for sorted array, insertion sort and quick sort are fastest algorithm for sorted array. Main reason for this, quick sort's time complexity is $O(n^2)$, insertion and counting sort's time comlexities are O(n) for sorted array as we calculated.

b-) Reverse Sorted Array



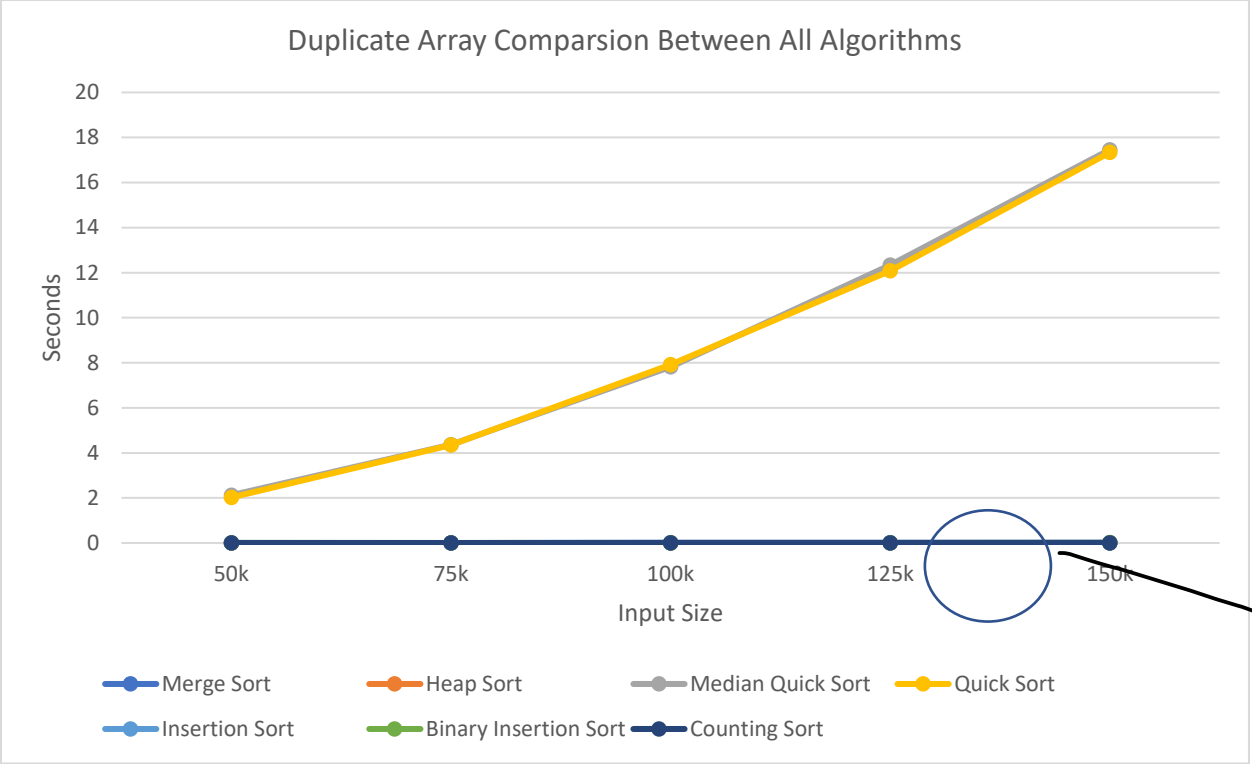
We see a very huge difference insertion sort and others except binary insertion sort. We can say that the main reason for this difference is number of basic operations which affects their time complexity. Binary Insertion and Insertion sort's time complexities are $O(n^2)$ for reverse sorted arrays as we calculated. Quicksort is not also an efficient sorting algorithm to sort reverse sorted arrays. Because it compares elements by decreasing two by two.

Closer Look for less than 0,6 sec



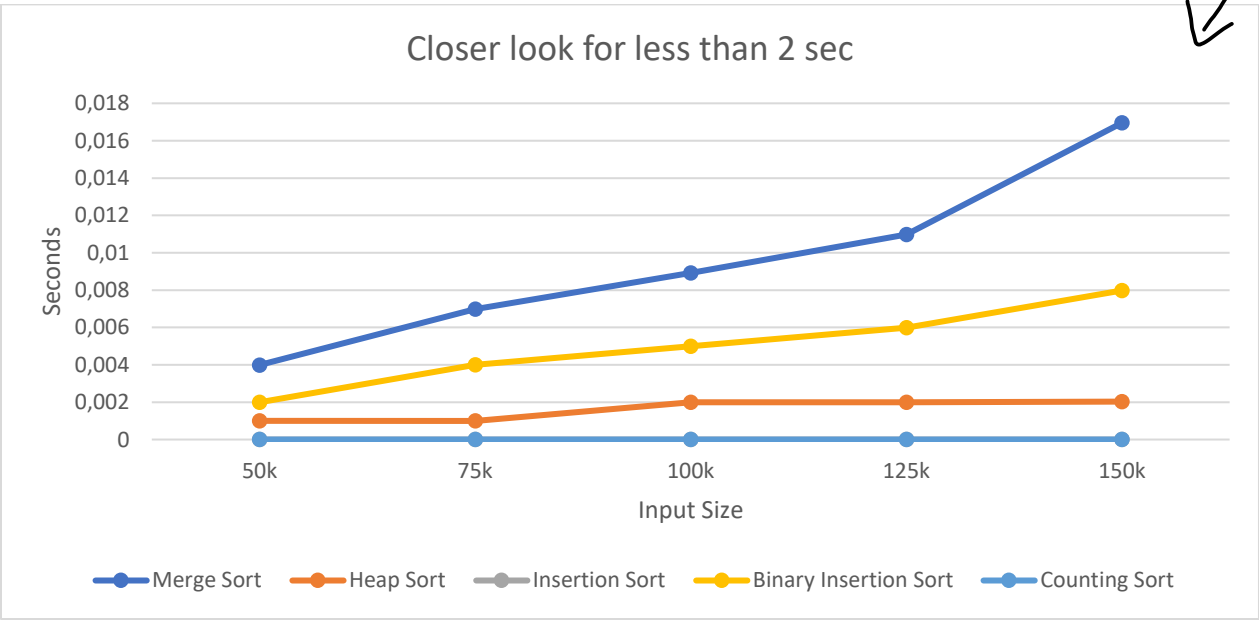
These 4 sorting algorithm work in really small amount of seconds. This is why, if we use these 4 for a reverse sorted array, it would be better for our program. Counting Sort is best algorithm for reverse sorted array. Its time complexity O(n) as we calculated.

c-) Duplicate Array



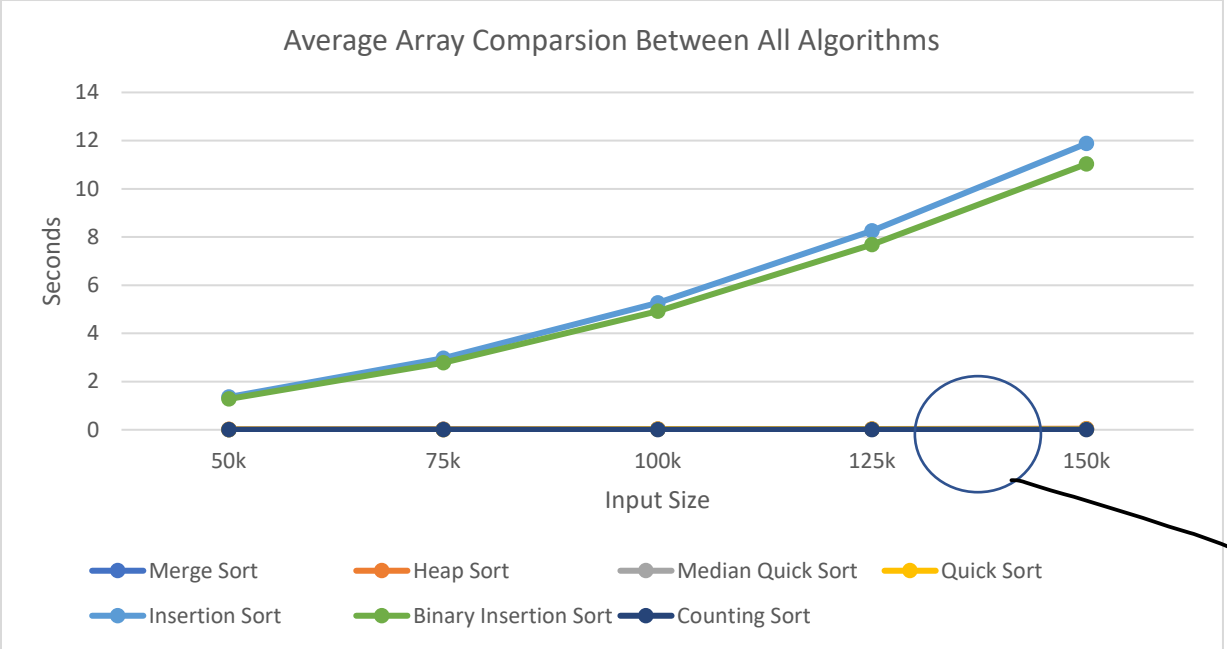
Duplicate array like sorted array, we can see the almost same chart of sorted array. We really have huge amount of number of basic operation in quick sort and median of three quick sort for this array type. Their time complexities are $O(n^2)$ as we calculated.

Closer look for less than 2 second



As we can see in the chart above, the best algorithm to sort is counting and insertion sort. Their time complexities are O(n) as we calculated. There is very small difference between these five sorting algorithms. Therefore we can use these algorithms instead of quicksort algorithm for duplicate arrays.

d-) Average Array



Insertion and binary insertion sort are worst case for average case. Their time complexities are $O(n^2)$ as we calculated.

Closer look for less than 1 sec

Counting sort is best case for average case. It has O(n) time complexity. Other sorting algorithms time comlexities are $O(n^2)$ or O(nlogn). That's why counting sort is the best algorithm in average for our experiment.

Summary:

For best case, insertion sort and counting sort have best time complexity which is $O(n)$.

For worst case, insertion, binary insertion, quick, median of three quick sort have worst $O(n^2)$ time complexity.

For average case, counting sort has the best time complexity which is O(n) time complexity.

In our experiment counting sort is the best algorithm for every case. Its time complexity is O(n) for every case because we have restriction for our input's range. If our range was bigger counting sort might not be the best algorithm.