ANALYSIS OF ALGORITHMS PROJECT #1 REPORT
CSE 2046

Edanur Öztürk 150117007
Feyza Nur Bulgurcu 150117033
Emre Eren 150118020

This project is about sorting algorithms. There are 7 types of different sorting algorithms and we are asked to experiment with these sorting algorithms. The inputs are bounded 0 to 10000. This project aims to calculate the time complexities of these sorting algorithms and compare them to the theoretical complexity values.

For calculating complexity: we decided to use the physical unit of time and by using this information we calculated the time complexities of these sorting algorithms in different input sizes and styles.

We chose 10 different sizes, 3 different list types, and to apply 10 trials. We implemented this project in Java.

- We have 3 types of inputs. One is random, another is sorted and the other is reverse sorted. We have chosen these inputs because of a reason of course:
  All sorting algorithms become best or worst case with these 3 types of inputs and we wanted to show this by applying 3 different input styles.

- For example, we have chosen one as a sorted input because that quicksort 3way would become the worst case in this input style.

- We chose 10 different input sizes to compare and visualize better.

- Since we use high-level language like java, we wanted the results to be more consistent, so we did 5 trials on 2 different computers and got their average.

The explanation of the method we used: Firstly, we will take two arrays with sizes of n and k. They will be the same type(state) (if n is random k must be random too). Then both arrays will be sorted by an algorithm and we will calculate the duration of these processes and store them.

Later by using these stored values we will obtain the experimental growth rate of these algorithms. We will calculate the theoretical growth rate. Then experimental growth rate and theoretical growth rate will be compared.

List Size	1024			2048			3072		
List Type	Random	Sorted	Reverse Sorted	Random	Sorted	Reverse Sorted	Random	Sorted	Reverse Sorted
Hoare's QuickSort	0.1052	0.26057	0.21306	0.15677	1.01717	0.74672	0.22657	2.29477	1.76914
3way QuickSort	0.0871	0.99827	1.08765	0.18111	4.10534	4.46058	0.2549	8.84642	9.84936
Counting Sort	0.02733	0.02592	0.02443	0.03621	0.02897	0.02822	0.03473	0.03011	0.03303
BinaryInsertion Sort	0.16896	0.02547	0.17786	0.51803	0.04702	0.70581	0.94444	0.0708	1.47168
Insertion Sort	0.13032	0.00284	0.25378	0.5595	0.00579	1.0361	1.20309	0.0085	2.27769
HeapSort	0.08097	0.05748	0.06545	0.16811	0.13217	0.12593	0.28648	0.21278	0.21023
MergeSort	0.07798	0.03116	0.03645	0.15228	0.07221	0.05751	0.23026	0.10749	0.08627

This table is from our excel file for simplicity we just showed results of 3 different sizes.[1]

As we told before, we ran the program more than one time and took the average of them. After taking the average of our data we calculated our values and compared them with theoretical values.

For the best case of the Insertion Sort algorithm, the theoretical growth rate is $O(n)$[2]. So we have to obtain the experimental growth rate as $O(n)$. The case where we get the best growth rate is to apply insertion sort on a sorted list [3].

Two different sized array inputs are required for calculating the experimental growth rate. We chose these inputs' sizes as 1024 and 3072 and both are sorted lists:

$$n_1 = 1024,\ t_1 = 0.00284ms$$

$$n_2 = 3072,\ t_2 = 0.0085ms$$

$$n = n_2/n_1 = 3,\ t = t_2/t_1 \cong 3$$

$$n/t \cong 1$$

Since the rate between execution time rate and theoretical growth rate is near(roughly equal) to 1, our experimental growth rate is approximately $O(n)$.

The worst-case growth rate for the Insertion Sort algorithm is $O(n^2)$[4]. Hence our experimental growth rate must achieve this growth rate too. We achieve this case when we apply Insertion sort on the reverse sorted list [5].

We apply the calculations, we mentioned beforehand, again:

$$n_1 = 1024,\ t_1 = 0.25378ms$$

$$n_2 = 3072,\ t_2 = 2.27769ms$$

$$n = n_2^2 / n_1^2 = 9,\ t = t_2 / t_1 \cong 8.97$$

$$n/t \cong 1$$

The rate between execution time rate and theoretical growth rate is near 1, our experimental growth rate is approximately $O(n^2)$.

As you can see above there are tiny differences between theoretical and experimental growth rates. These differences may happen for several reasons:

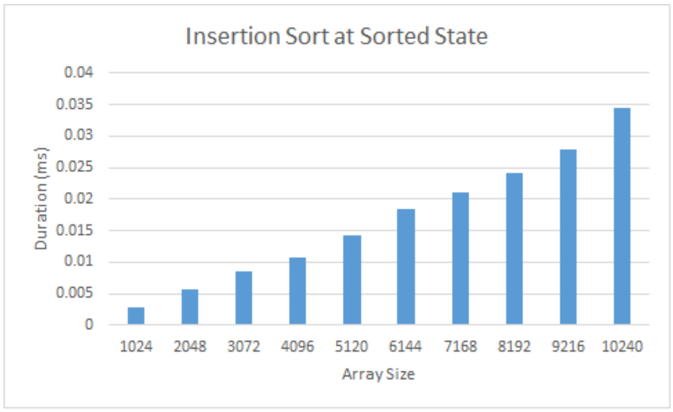
- Java is not a low-level language. Hence there might be some excess duration because of background processing (Garbage collection vs.).
- Computers have operating systems. In an operating system, several processes can run at the same time. We can have some unwanted processes or I/O operations while our program is running.

Although to eliminate these problems we run this program more than one time in different environments problems may still show because of the small number of these trials.

1) Insertion Sort

Theoretically, Insertion Sort's best-case complexity is $\Omega(n)$, the average-case time complexity is $\Theta(n^2)$ and worst-case time complexity is $O(n^2)$.[6]

Best-case Scenario:



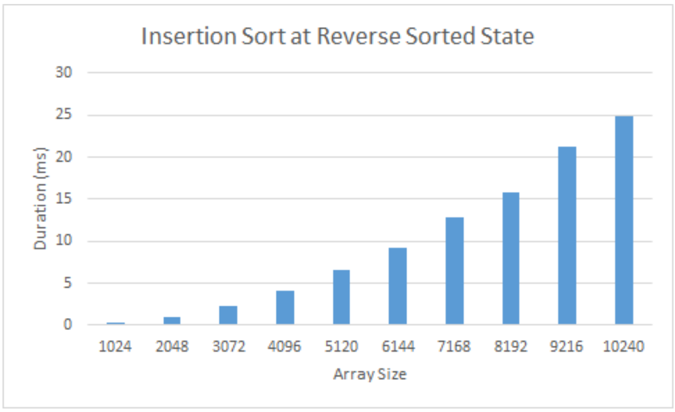
$n_2 = 3072,\ t_2 = 0.0085ms$

$n_1 = 1024,\ t_1 = 0.00284ms$

$n = n_2/n_1 = 3,\ t = t_2/t_1 \cong 3$

$n/t \cong 1 \ \ => \ \Omega(n)$

Worst-case Scenario:



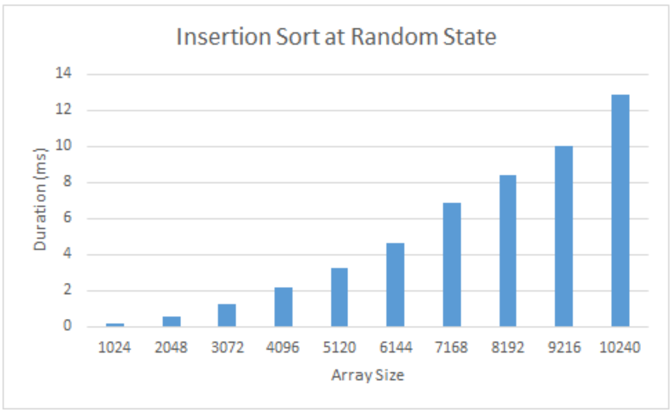
$n_1 = 1024,\ t_1 = 0.25378ms$

$n_2 = 3072,\ t_2 = 2.27769ms$

$n = n_2^2/n_1^2 = 9,\ t = t_2/t_1 \cong 8.97$

$n/t \cong 1 \ \ => \ O(n^2)$

Average case Scenario:



$n_1 = 1024,\ t_1 = 0.13032ms$

$n_2 = 3072,\ t_2 = 1.15133ms$

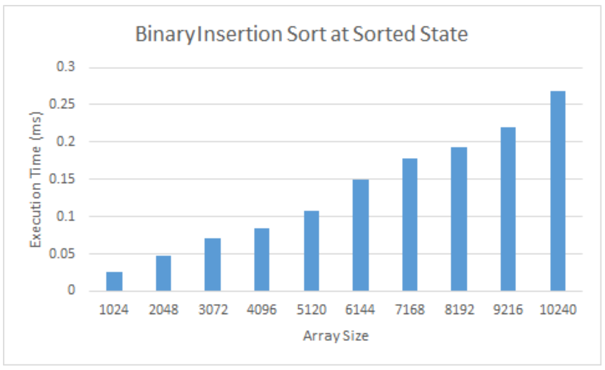
$n = n_2^2/n_1^2 = 9,\ t = t_2/t_1 \cong 8.83$

$n/t \cong 1 \ \ => \ \Theta(n^2)$

2) Binary Insertion Sort

Theoretically, Binary Insertion Sort's best case time complexity is $\Omega(n)$, the worst-case time complexity is $O(n^2)$.[7]

Best-case Scenario:



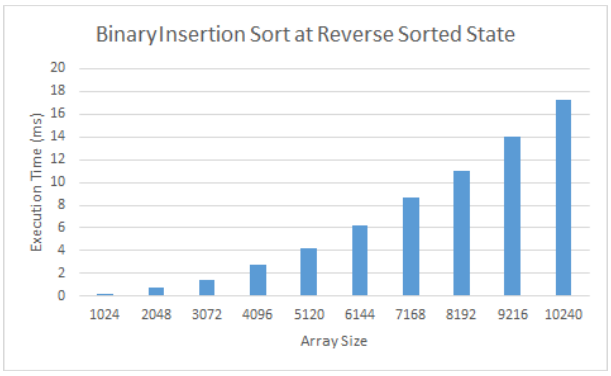
$n_2 = 3072,\ t_2 = 0.0708ms$

$n_1 = 1024,\ t_1 = 0.02547ms$

$n = n_2/n_1 = 3,\ t = t_2/t_1 \cong 2.77$

$n/t \cong 1 \ => \ \Omega(n)$

Worst-case Scenario:



$n_2 = 5120,\ t_2 = 4.207ms$

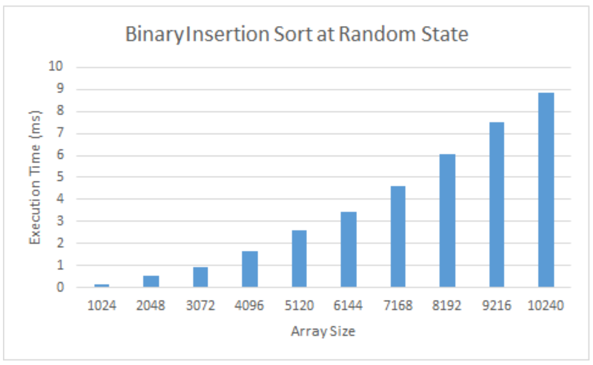
$n_1 = 2048,\ t_1 = 0.70581ms$

$n = n_2^2/n_1^2 = 6.25$

$t = t_2/t_1 \cong 5.96$

$n/t \cong 1 \ => \ O(n^2)$

Average case Scenario:



$n_1 = 2048,\ t_1 = 0.51803ms$

$n_2 = 4096,\ t_2 = 1.63501ms$

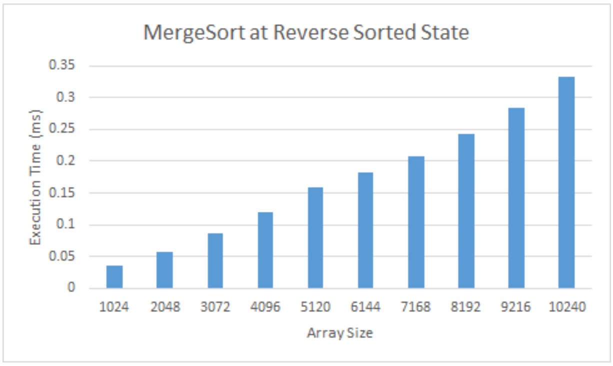
$n_b = n_2/n_1 = 2,\ n_w = n_2^2/n_1^2 = 4$

$t = t_2/t_1 \cong 3.16$

$t => \Omega(n) \subseteq \Theta(random) \subseteq O(n^2)$

3) Merge Sort

Theoretically, Merge Sort's time complexity is always $O(nlogn)$. [8]



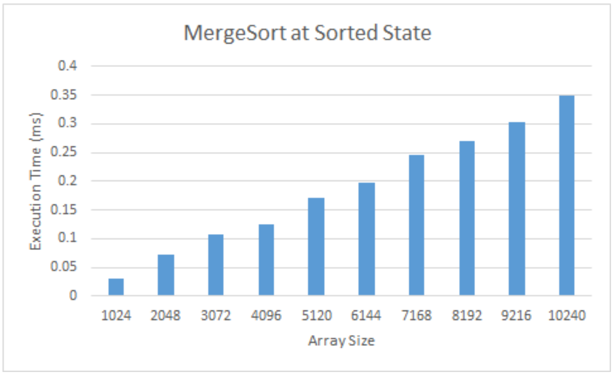
$n_1 = 2048,\ t_1 = 0.05751ms$

$n_2 = 4096,\ t_2 = 0.11901ms$

$n = n_2 logn_2 / n_1 logn_1 \cong 2.18$

$t = t_2 / t_1 \cong 2.07$

$n/t \cong 1 \ => O(nlogn)$



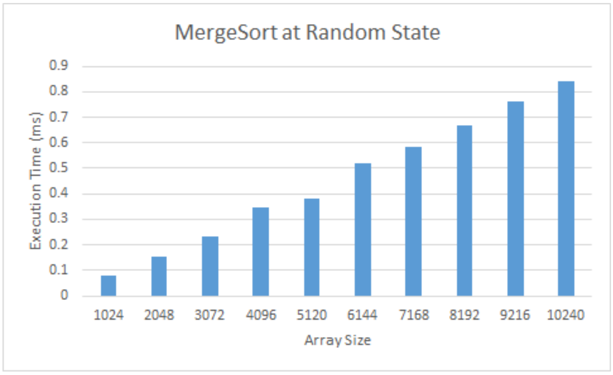
$n_1 = 3072,\ t_1 = 0.10749ms$

$n_2 = 7168,\ t_2 = 0.24599ms$

$n = n_2 logn_2 / n_1 logn_1 \cong 2.58$

$t = t_2 / t_1 \cong 2.28$

$n/t \cong 1 \ => O(nlogn)$



$n_1 = 3072,\ t_1 = 0.23026ms$

$n_2 = 7168,\ t_2 = 0.58208ms$

$n = n_2 logn_2 / n_1 logn_1 \cong 2.58$

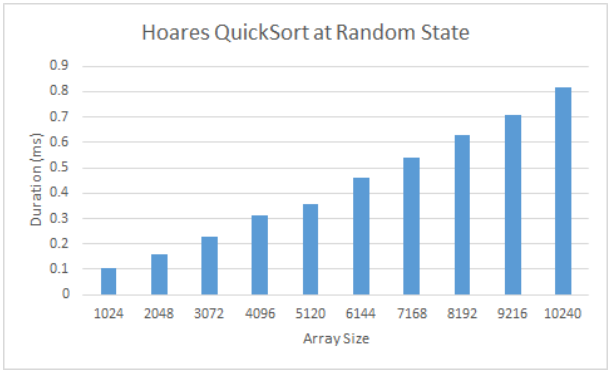
$t = t_2 / t_1 \cong 2.53$

$n/t \cong 1 \ => O(nlogn)$

4) Quick Sort (Hoare's sort)

Theoretically, Hoare's Sort's best-case time complexity is $\Omega(nlogn)$, the worst-case time complexity is $O(n^2)$.[9]

Best case scenario:



$n_1 = 1024,\ t_1 = 0.1052ms$

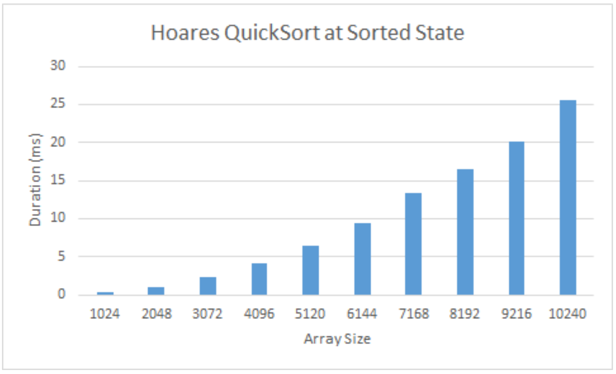
$n_2 = 3072,\ t_2 = 0.22657ms$

$n = n_2 logn_2 / n_1 logn_1 \cong 3.48$

$t = t_2 / t_1 \cong 2.15$

$\Omega(nlogn)$

Worst-case Scenarios:



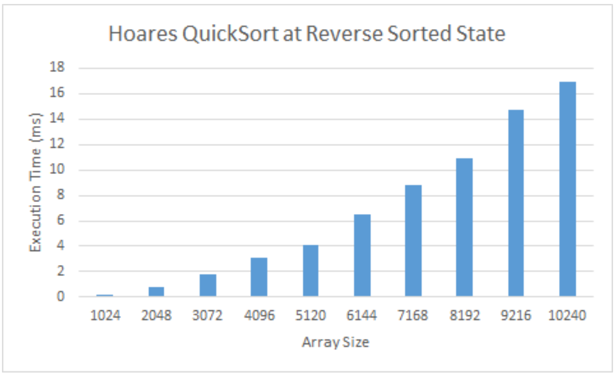
$n_2 = 6144,\ t_2 = 9.38176ms$

$n_1 = 2048,\ t_1 = 1.01717ms$

$n = n_2^2 / n_1^2 = 9$

$t = t_2 / t_1 \cong 9.22$

$n/t \cong 1.04 \cong 1 \ => O(n^2)$



$n_1 = 2048,\ t_1 = 0.74672ms$

$n_2 = 6144,\ t_2 = 6.45566ms$

$n = n_2^2 / n_1^2 = 9$

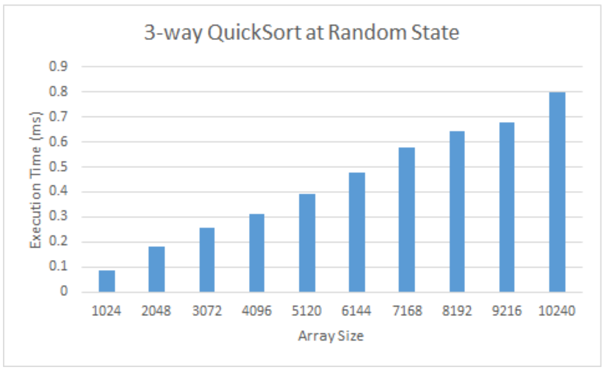
$t = t_2 / t_1 \cong 8.64$

$n/t \cong 1 \ => O(n^2)$

5) Quick Sort (median of 3 pivot selection)

Theoretically, Quicksort's best case time complexity is $\Omega(nlogn)$, the worst-case time complexity is $O(n^2)$. [10]

Best-case Scenario:



$n_1 = 1024,\ t_1 = 0.0871ms$

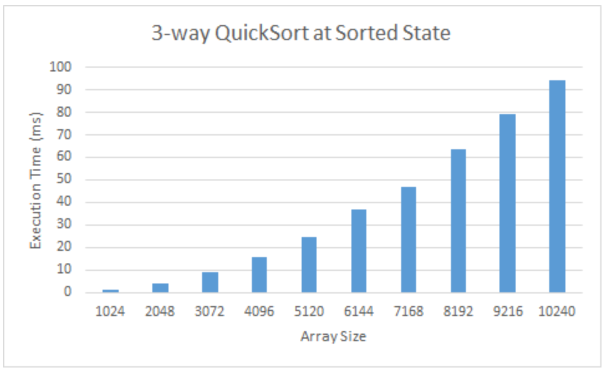
$n_2 = 3072,\ t_2 = 0.2549ms$

$n = n_2 logn_2 / n_1 logn_1 = 3.48$

$t = t_2 / t_1 \cong 2.92$

$n/t \cong 1.19 \cong 1 \ \ => \ \Omega(nlogn)$

Worst-case Scenarios:



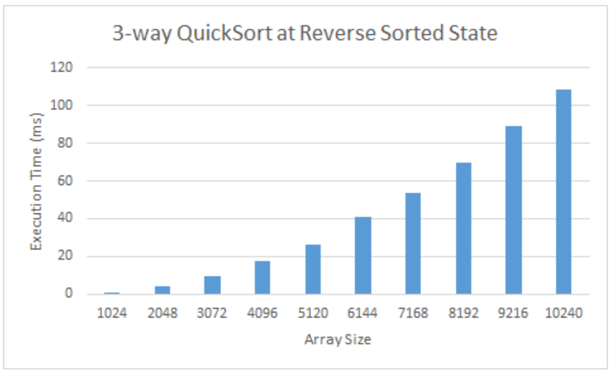
$n_1 = 1024,\ t_1 = 0.99827ms$

$n_2 = 3072,\ t_2 = 8.8464ms$

$n = n_2^2 / n_1^2 = 9$

$t = t_2 / t_1 \cong 8.86$

$n/t \cong 1.02 \cong 1 \ \ => \ O(n^2)$



$n_1 = 2048,\ t_1 = 4.46058ms$

$n_2 = 6144,\ t_2 = 40.9169ms$

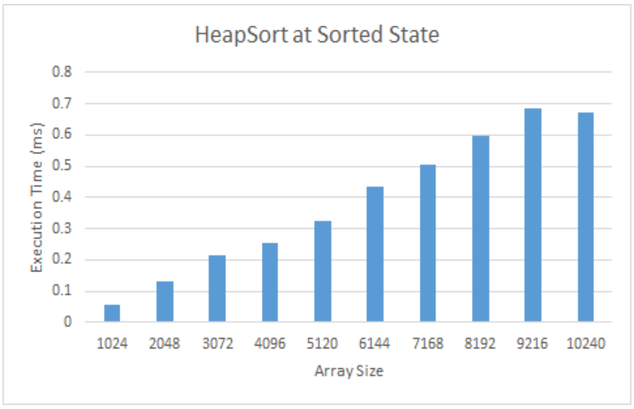
$n = n_2^2 / n_1^2 = 9$

$t = t_2 / t_1 \cong 9.17$

$n/t \cong 0.98 \cong 1 \ \ => \ O(n^2)$

6) Heap Sort

Theoretically, Heap Sort's time complexity is always $O(nlogn)$. [11]



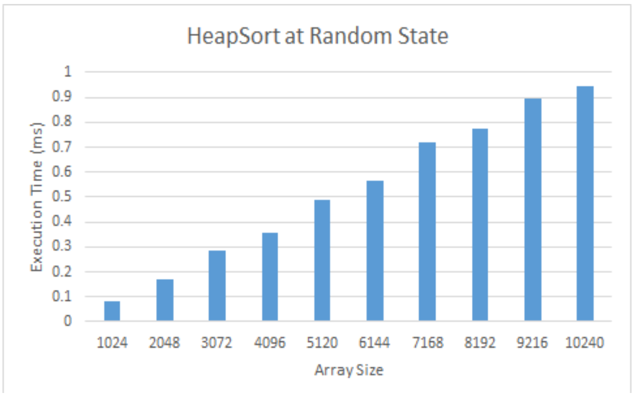
$n_1 = 2048,\ t_1 = 0.13127ms$

$n_2 = 4096,\ t_2 = 0.25328ms$

$n = n_2 logn_2 / n_1 logn_1 = 2.18$

$t = t_2 / t_1 \cong 1.93$

$n/t \cong 1.13 \cong 1 \ => \ O(nlogn)$



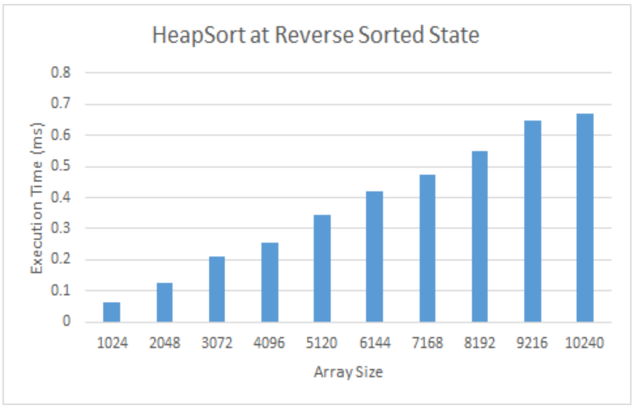
$n_1 = 1024,\ t_1 = 0.0871ms$

$n_2 = 3072,\ t_2 = 0.2549ms$

$n = n_2 logn_2 / n_1 logn_1 = 3.48$

$t = t_2 / t_1 \cong 2.93$

$n/t \cong 1.19 \cong 1 \ => \ O(nlogn)$



$n_2 = 6144,\ t_2 = 0.42196ms$

$n_1 = 2048,\ t_1 = 0.12593ms$

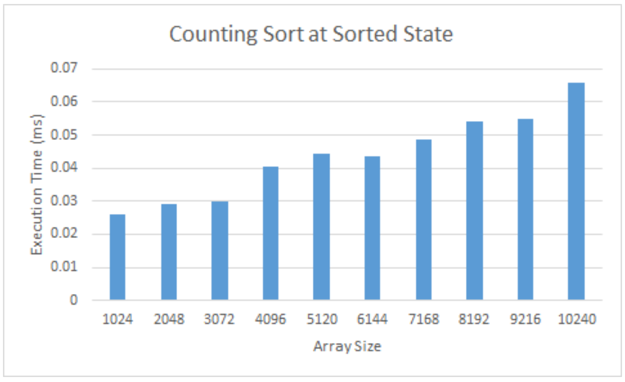
$n = n_2 logn_2 / n_1 logn_1 = 3.43$

$t = t_2 / t_1 \cong 3.35$

$n/t \cong 1.02 \cong 1 \ => \ O(nlogn)$

7) Counting Sort

Theoretically, Counting Sort's time complexity is $O(n + k)$.[12]



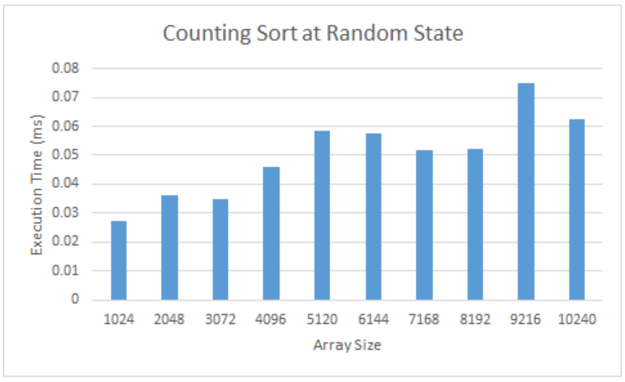
$n_1 = 2048,\ t_1 = 0.029ms$

$n_2 = 7168,\ t_2 = 0.0487ms$

$n = n_2 + k\ /n_1 + k = 1.43$

$t = t_2 / t_1 \cong 1.68$

$n/t \cong 0.85 \cong 1\ \ => O(n + k)$



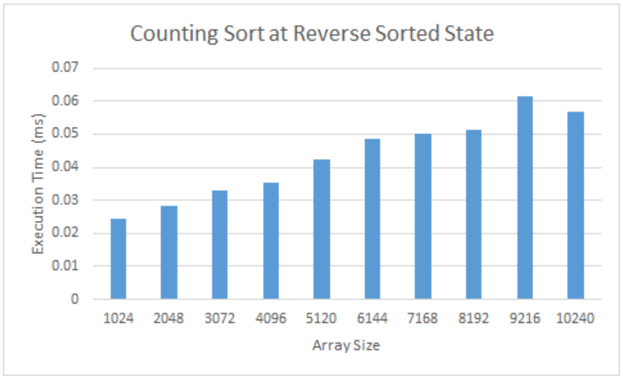
$n_1 = 2048,\ t_1 = 0.03621ms$

$n_2 = 6144,\ t_2 = 0.0575ms$

$n = n_2 + k/n_1 + k = 1.34$

$t = t_2 / t_1 \cong 1.59$

$n/t \cong 0.87 \cong 1\ \ => O(nlogn)$



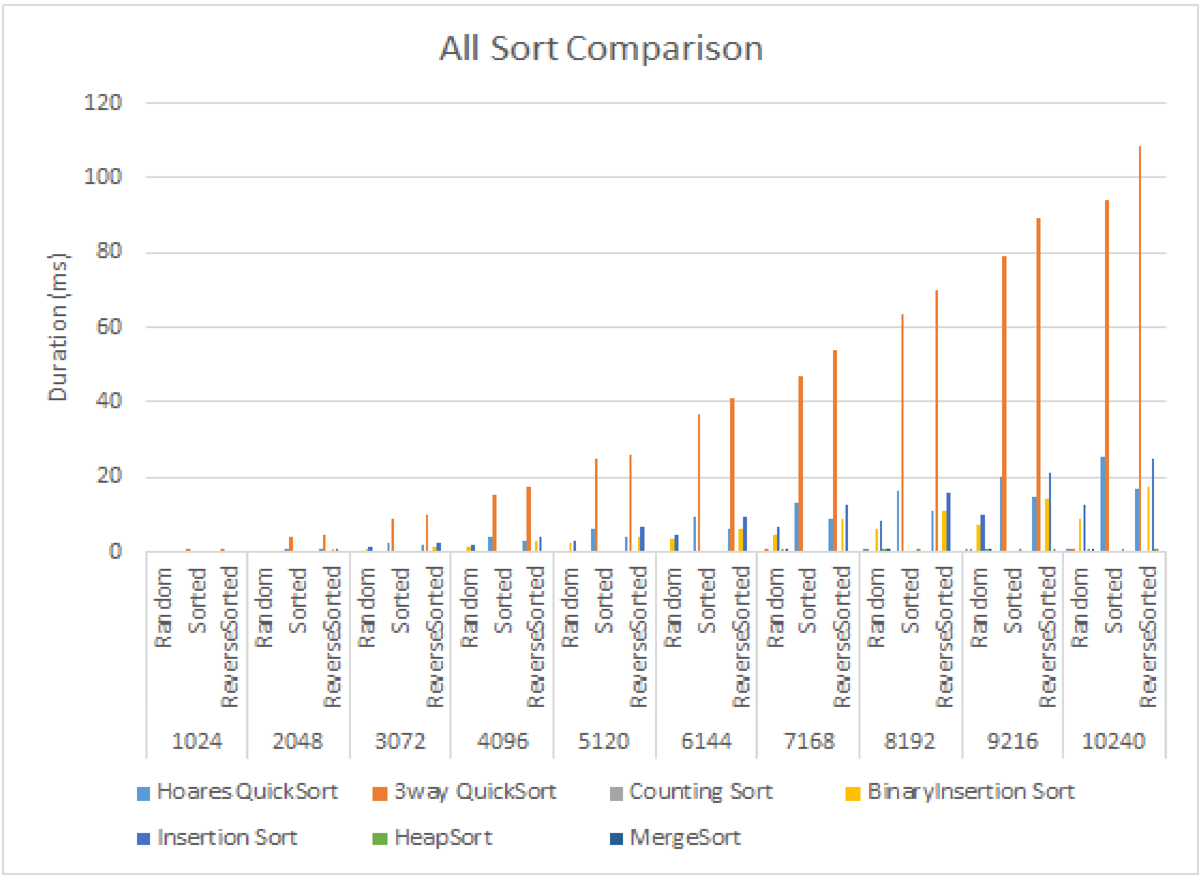
$n_1 = 2048,\ t_1 = 0.0282ms$

$n_2 = 6144,\ t_2 = 0.0485ms$

$n = n_2 + k/n_1 + k = 1.34$

$t = t_2 / t_1 \cong 1.72$

$n/t \cong 0.78 \cong 1\ \ => O(n + k)$



Takeaways from Our Experiment

1. The best-case and the average for Quick Sort are generally obtained when input is in a random state. The best-case and average-case time complexity of QuickSort are $O(nlogn)$.

2. The worst-case for QuickSort is obtained when the input list is in a sorted or reverse sorted state. The worst-case time complexity of QuickSort is $O(n^2)$.

3. 3-way QuickSort works with the same time complexity as regular QuickSort.

4. The state does not differ in the time complexity of Counting Sort. The time complexity of the Counting sort always works in the time complexity of $O(n + k)$. (k is a range of numbers)

5. The best case for Insertion Sort is obtained when the input list is in a sorted state. The best-case time complexity of Insertion Sort is $O(n)$.

6. The worst-case for Insertion Sort is obtained when the input list is in a reverse sorted state. The worst-case time complexity of Insertion sort is $O(n^2)$.

7. The average case for Insertion Sort is generally obtained when input is in a random state. The average case time complexity of Insertion Sort is $O(n^2)$.

8. Binary Insertion Sort works with the same complexity as regular Insertion Sort. It is just a variant of Insertion Sort.

9. Cases do not differ for HeapSort. HeapSort always works in $O(nlogn)$ time complexity.

10. Cases do not differ for MergeSort. It always works in $O(nlogn)$ time complexity.

REFERENCES

1: Our performance metrics will be added to the zip file.
https://docs.google.com/spreadsheets/d/1LnoEQmNpPGcj0k8HGDycrFwwBh7ltJ1aS38wqIdfUWI/edit#gid=316978090

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12: Big-O Algorithm Complexity Cheat Sheet (Know Thy Complexities!) @ericdrowell (bigocheatsheet.com)