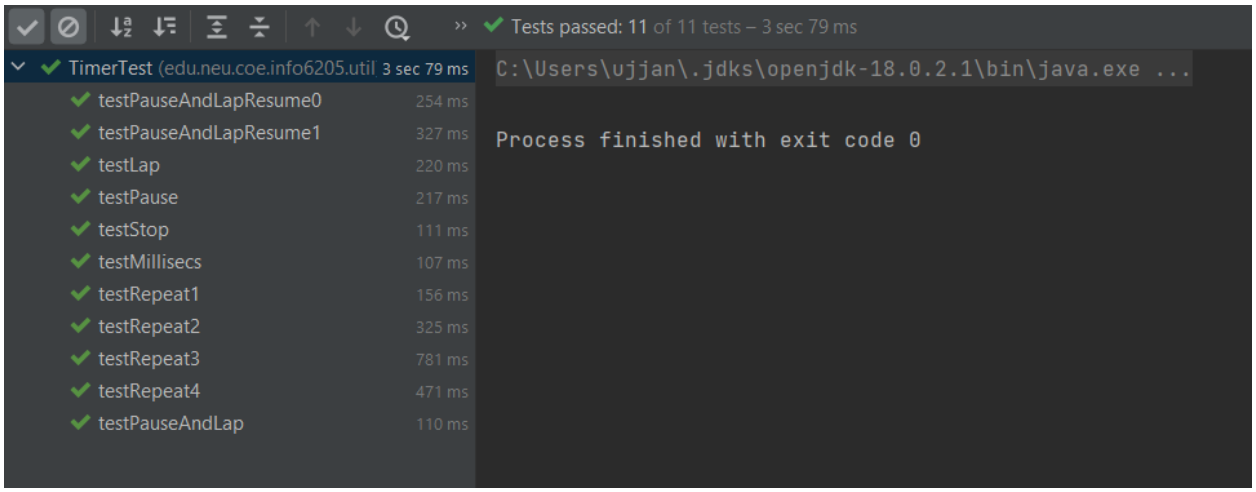
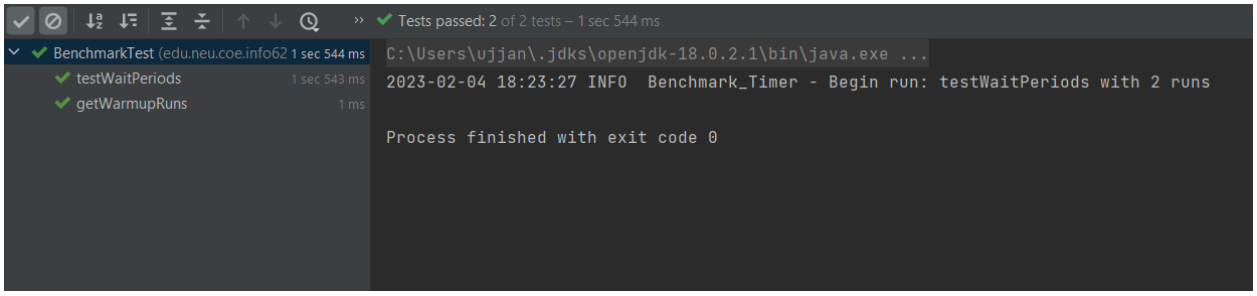
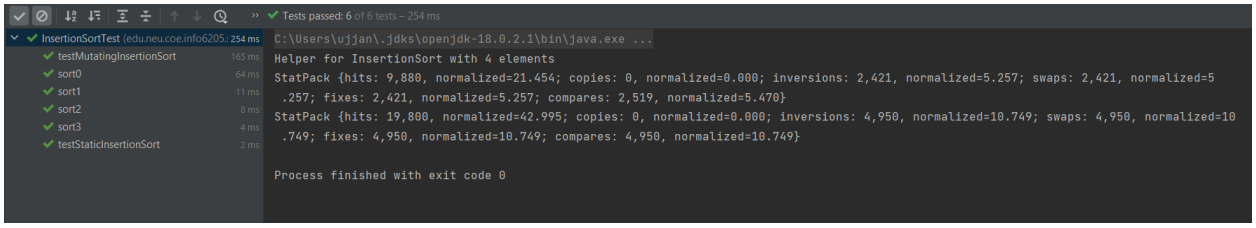
Ujjanth Arhan
002108348

Part 1: Screenshots of Timer test and Benchmark test





Part 2: Screenshot of Insertion Sort test



Part 3: Benchmarking insertion sort

Observations:

The following table shows a benchmark for insertion sort of various sizes of Integer array where the size is represented as N. (The values are present in milliseconds) The input array provided is of 4 categories:

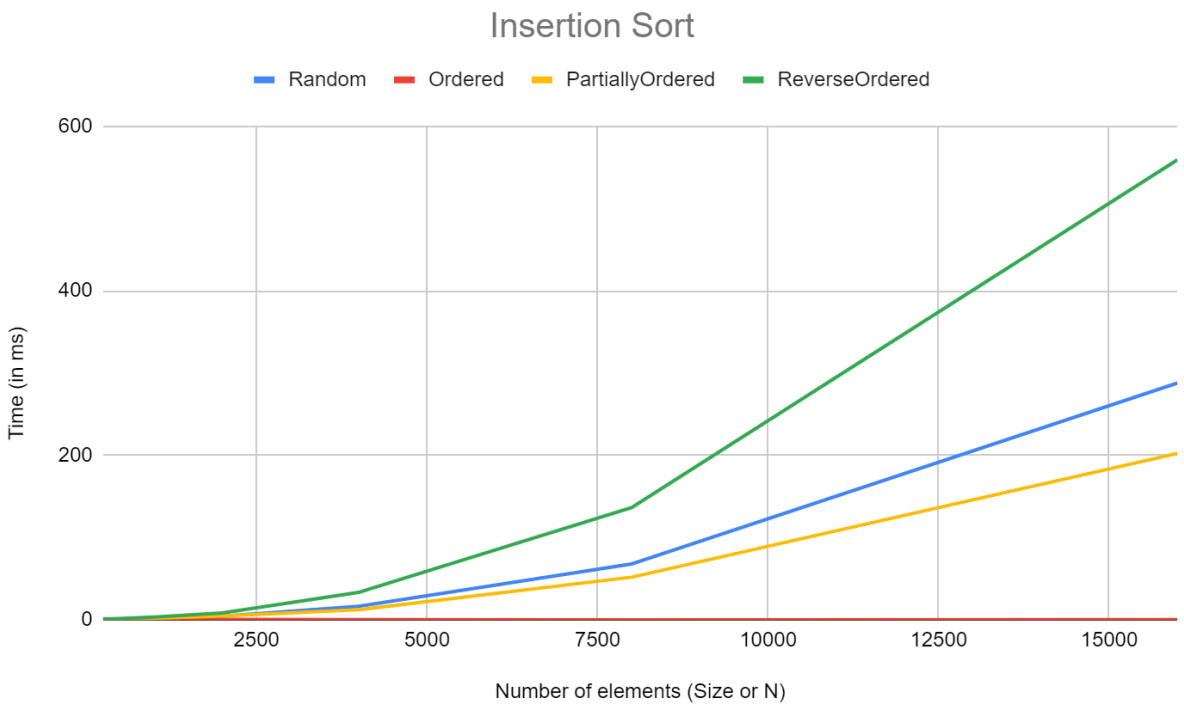
a. Random - The values of the input array are obtained randomly by using java's Random package.
b. Ordered - The values of the input array are present in sorted order.
c. Partially Ordered - The values of the input array are partially sorted. In this case, the first half of the array is sorted while the rest of the array is filled by random values as obtained for the random array.
d. Reverse Ordered - The values of the input array are present in descending order.

	Sort Order			
N	Random	Ordered	PartiallyOrdered	ReverseOrdered
250	1.63	0.81	0.81	0.88
500	1.13	0.67	1.12	1.69
1000	2.39	0.57	1.95	3.6
2000	4.69	0.7	4.85	8.59
4000	16.6	0.46	12.49	33.58
8000	68.12	0.31	52.01	136.58
16000	287.86	0.36	202.33	559

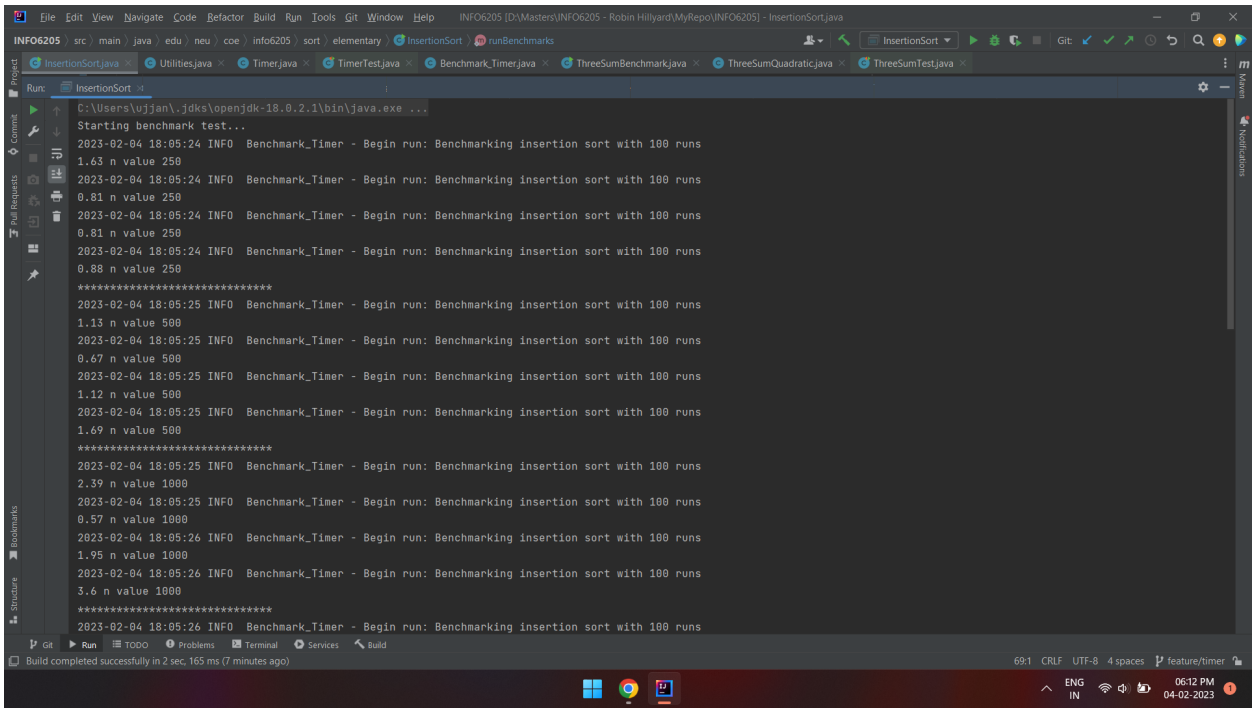
The graph depicted below shows the Size vs Time chart. The order of increase of slopes are: Ordered, Partially Ordered, Random and Reverse Ordered respectively. As per the insertion sort algorithm, the time is supposed to be least for an already sorted array and highest for an array that is in descending order. This is confirmed from the actual values obtained. Explanation:

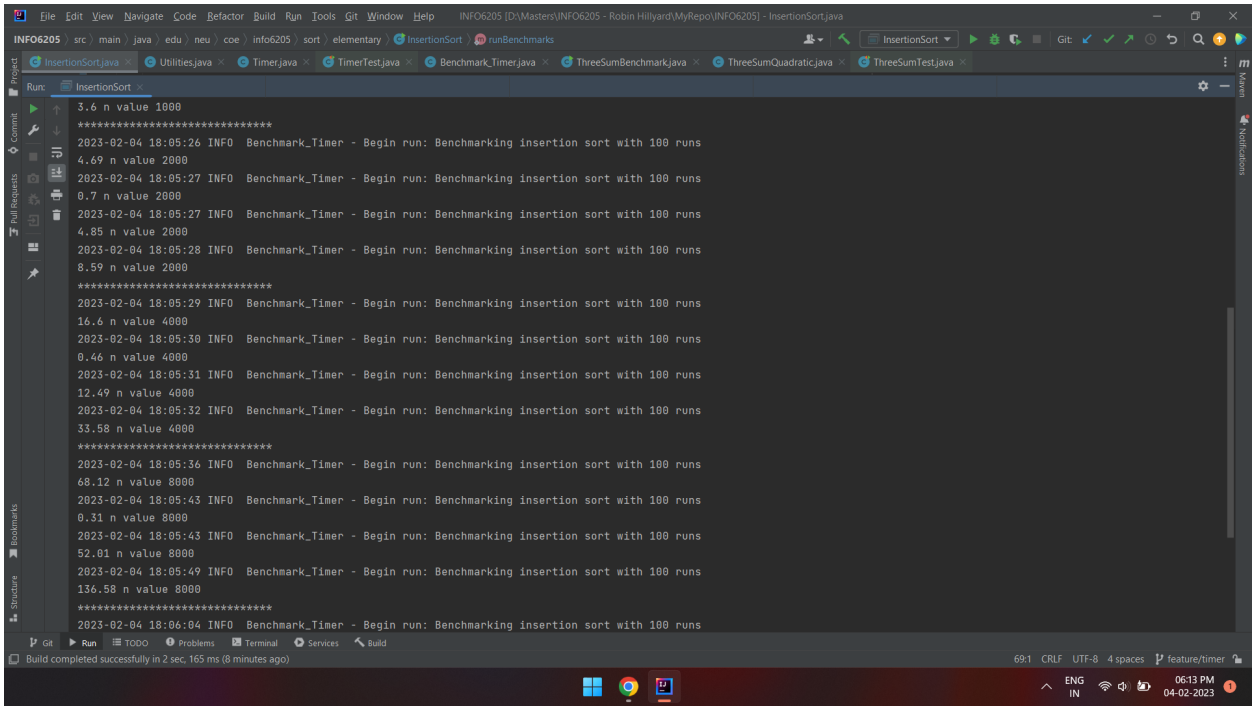
1. Random: In this case, the elements are randomly present. So, we can assume that the elements are sorted and some are not. For the sorted elements the time complexity would be O(n) and for the elements that are not sorted the complexity would be O(n^2) as the values need to be swapped. This translates to a time complexity of O(n^2).
2. Ordered: In this case, all the elements are already sorted so no swaps are needed meaning it takes the least amount of time. The time complexity in this scenario is O(n) as it only needs to traverse the array once for each element. This is the best case scenario.
3. Partially ordered: In this case, only half the elements are ordered and the other half are not sorted. This can be approximated as having O(n) for the first half and O(n^2) for the next half which translates to a complexity of O(n^2).
4. Reverse ordered: in this case, the elements are ordered in reverse order so the values are not sorted. This means that the values need to be swapped for each element. This is the worst case scenario which translates to a complexity of O(n^2).

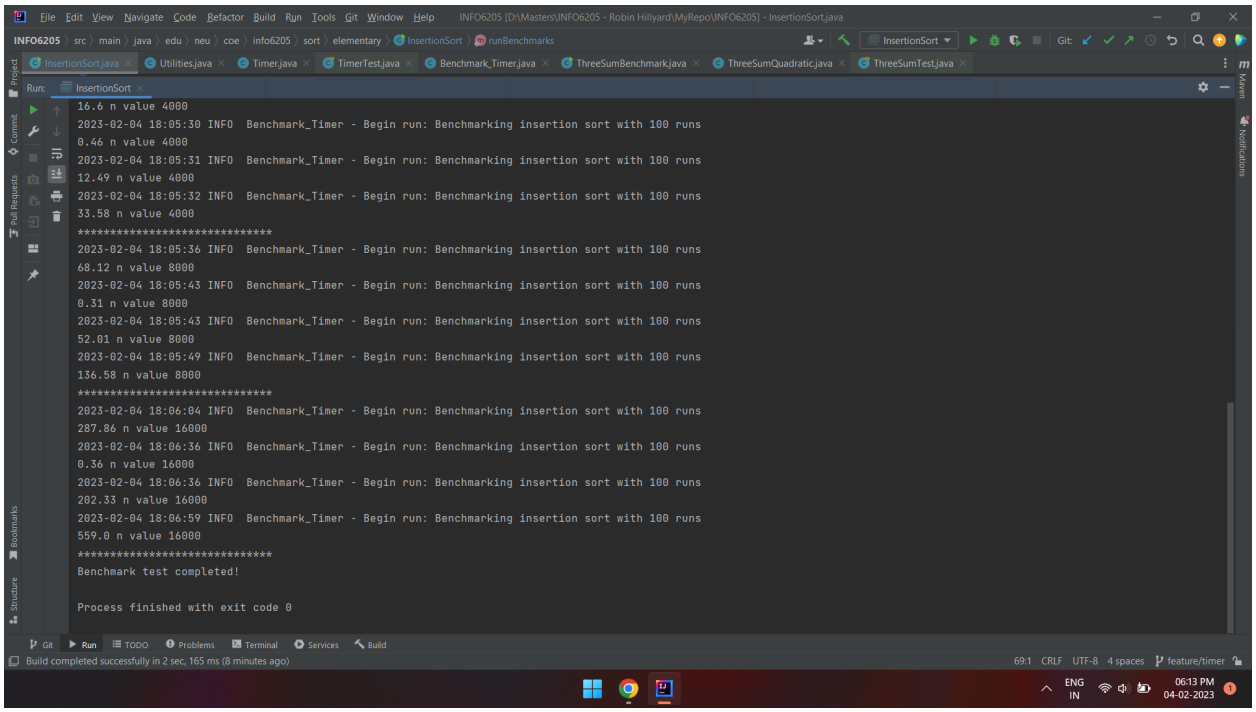
From the graph, we can see that the pattern observed matches the predictions made above with the worst case being reverse ordered elements, average cases being random elements and partially sorted elements and the best case being the one where all the elements are already sorted.



Screenshot of the console output of the values obtained.




3.6 n value 1000
****
2023-02-04 18:05:26 INFO Benchmark_Timer - Begin run: Benchmarking insertion sort with 100 runs
4.69 n value 2000
2023-02-04 18:05:27 INFO Benchmark_Timer - Begin run: Benchmarking insertion sort with 100 runs
0.7 n value 2000
2023-02-04 18:05:27 INFO Benchmark_Timer - Begin run: Benchmarking insertion sort with 100 runs
4.85 n value 2000
2023-02-04 18:05:28 INFO Benchmark_Timer - Begin run: Benchmarking insertion sort with 100 runs
8.59 n value 2000
****
2023-02-04 18:05:29 INFO Benchmark_Timer - Begin run: Benchmarking insertion sort with 100 runs
16.6 n value 4000
2023-02-04 18:05:30 INFO Benchmark_Timer - Begin run: Benchmarking insertion sort with 100 runs
0.46 n value 4000
2023-02-04 18:05:31 INFO Benchmark_Timer - Begin run: Benchmarking insertion sort with 100 runs
12.49 n value 4000
2023-02-04 18:05:32 INFO Benchmark_Timer - Begin run: Benchmarking insertion sort with 100 runs
33.58 n value 4000
****
2023-02-04 18:05:36 INFO Benchmark_Timer - Begin run: Benchmarking insertion sort with 100 runs
68.12 n value 8000
2023-02-04 18:05:43 INFO Benchmark_Timer - Begin run: Benchmarking insertion sort with 100 runs
0.31 n value 8000
2023-02-04 18:05:43 INFO Benchmark_Timer - Begin run: Benchmarking insertion sort with 100 runs
52.01 n value 8000
2023-02-04 18:05:49 INFO Benchmark_Timer - Begin run: Benchmarking insertion sort with 100 runs
136.58 n value 8000
****
2023-02-04 18:06:04 INFO Benchmark_Timer - Begin run: Benchmarking insertion sort with 100 runs


16.6 n value 4000
2023-02-04 18:05:30 INFO Benchmark_Timer - Begin run: Benchmarking insertion sort with 100 runs
0.46 n value 4000
2023-02-04 18:05:31 INFO Benchmark_Timer - Begin run: Benchmarking insertion sort with 100 runs
12.49 n value 4000
2023-02-04 18:05:32 INFO Benchmark_Timer - Begin run: Benchmarking insertion sort with 100 runs
33.58 n value 4000
****
2023-02-04 18:05:36 INFO Benchmark_Timer - Begin run: Benchmarking insertion sort with 100 runs
68.12 n value 8000
2023-02-04 18:05:43 INFO Benchmark_Timer - Begin run: Benchmarking insertion sort with 100 runs
0.31 n value 8000
2023-02-04 18:05:43 INFO Benchmark_Timer - Begin run: Benchmarking insertion sort with 100 runs
52.01 n value 8000
2023-02-04 18:05:49 INFO Benchmark_Timer - Begin run: Benchmarking insertion sort with 100 runs
136.58 n value 8000
****
2023-02-04 18:06:04 INFO Benchmark_Timer - Begin run: Benchmarking insertion sort with 100 runs
287.86 n value 16000
2023-02-04 18:06:36 INFO Benchmark_Timer - Begin run: Benchmarking insertion sort with 100 runs
0.36 n value 16000
2023-02-04 18:06:36 INFO Benchmark_Timer - Begin run: Benchmarking insertion sort with 100 runs
202.33 n value 16000
2023-02-04 18:06:59 INFO Benchmark_Timer - Begin run: Benchmarking insertion sort with 100 runs
559.0 n value 16000
****
Benchmark test completed!
Process finished with exit code 0