Program Structures and Algorithms
Spring 2023(SEC –1)

NAME: Prem Kumar Raghava Manoharan
NUID: 002726784

Task:

- Implement InsertionSort (in the InsertionSort class) by simply looking up the insertion code used by Arrays.sort. If you have the instrument = true setting in test/resources/config.ini, then you will need to use the helper methods for comparing and swapping (so that they properly count the number of swaps/compares). The easiest is to use the helper.swapStableConditional method, continuing if it returns true, otherwise breaking the loop. Alternatively, if you are not using instrumenting, then you can write (or copy) your own compare/swap code. Either way, you must run the unit tests in InsertionSortTest.

- Implement a main program (or you could do it via your own unit tests) to actually run the following benchmarks: measure the running times of this sort, using four different initial array ordering situations: random, ordered, partially-ordered and reverse-ordered. I suggest that your arrays to be sorted are of type Integer. Use the doubling method for choosing n and test for at least five values of n. Draw any conclusions from your observations regarding the order of growth.

Relationship Conclusion:

- **Random:** The time complexity of insertion sort for a randomly ordered array is O(n^2) on average. This is because the algorithm must traverse the entire sorted portion of the array for every insertion, which can take up to n-1 comparisons.

- **Ordered:** The time complexity of insertion sort for an ordered array is O(n), as every element is already in its proper place and no swaps are required. This is the best-case scenario for insertion sort.

- **Partially-ordered:** The time complexity of insertion sort for a partially-ordered array will depend on how ordered the array is. The more ordered the array, the closer the running time will be to O(n), while the more randomly ordered the array is, the closer the running time will be to O(n^2).

- **Reverse-ordered:** The time complexity of insertion sort for a reverse-ordered array is O(n^2), as the algorithm must traverse the entire sorted portion of the array for every insertion and each insertion requires a maximum of n-1 comparisons.

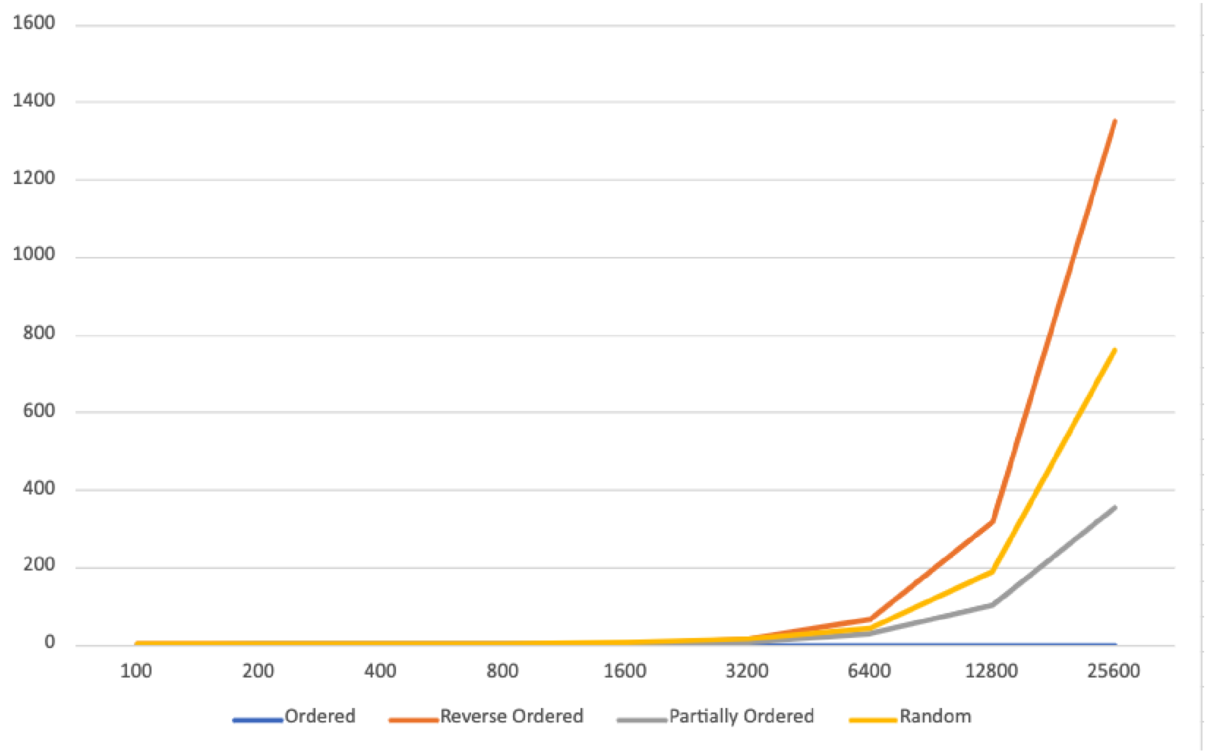
Evidence to support that conclusion:

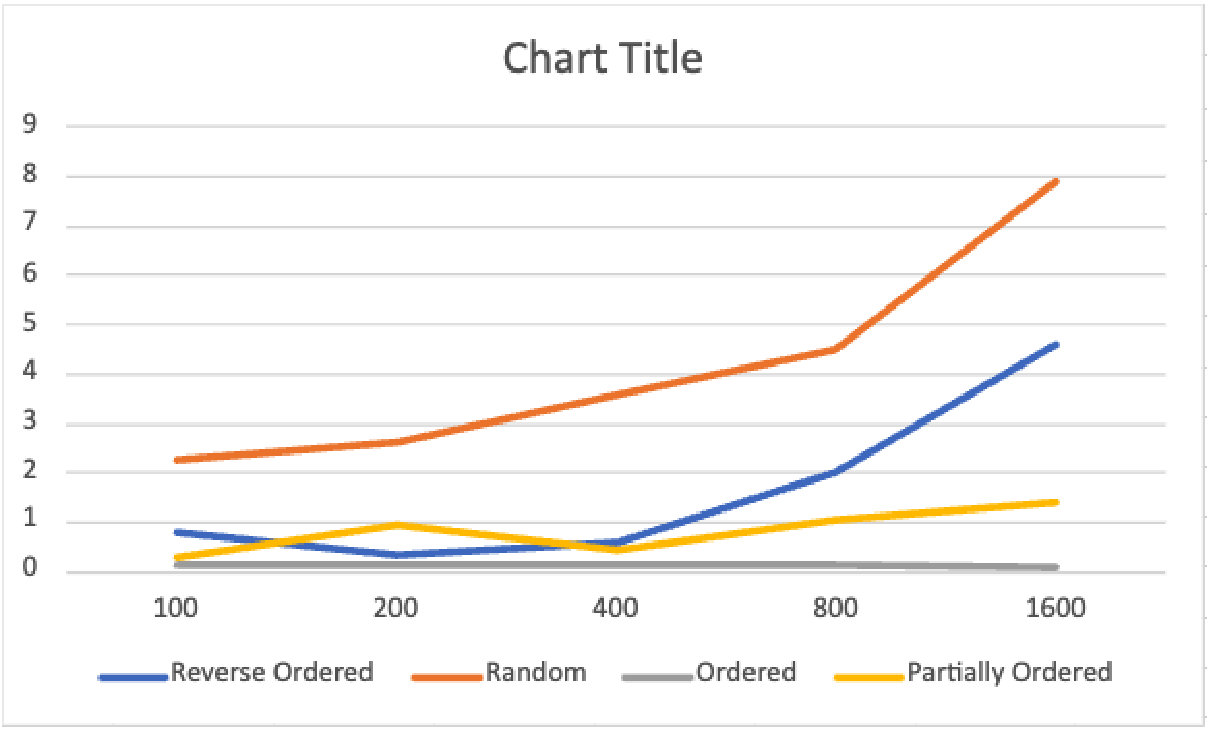
- **Random:** By considering the below graphical representation of the data observed using doubling technique Random array takes more time evening from the beginning of the smaller N values and it increases quadratically. But after some time, N above 1600 it took lesser time than Reverse ordered array.
- **Ordered:** In the below graph we can see that ordered array is taking linear time to sort because all the elements are sorted, and no swaps required.
- **Partially-ordered**: In the below graph partially ordered array line graph is not in a proper trend and it has both ups and downs this is because the more ordered the array, the closer the running time will be to O(n),while the more randomly ordered the array is, the closer the running time will be to O(n^2).
- **Reverse-ordered**: Initially I observed reverse order was taking lesser time than the random ordered array because of the side of N , but after N get higher and higher the reverse order array take more

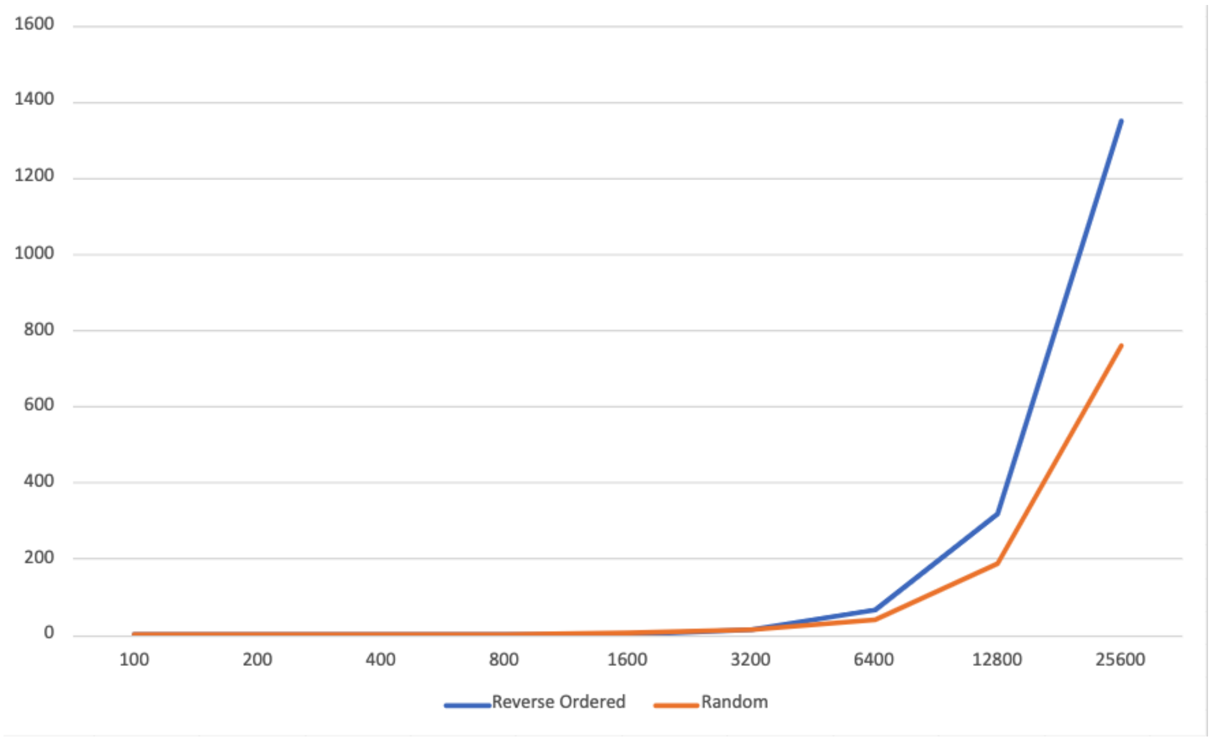
time than any other case , because this is exactly the order of 2 which is O(n^2) since all the elements will be swapped. But in the random case some of the elements can be in the correct positions.

- **By Observing the trends of the below graphical representation we can agree the above observations.**

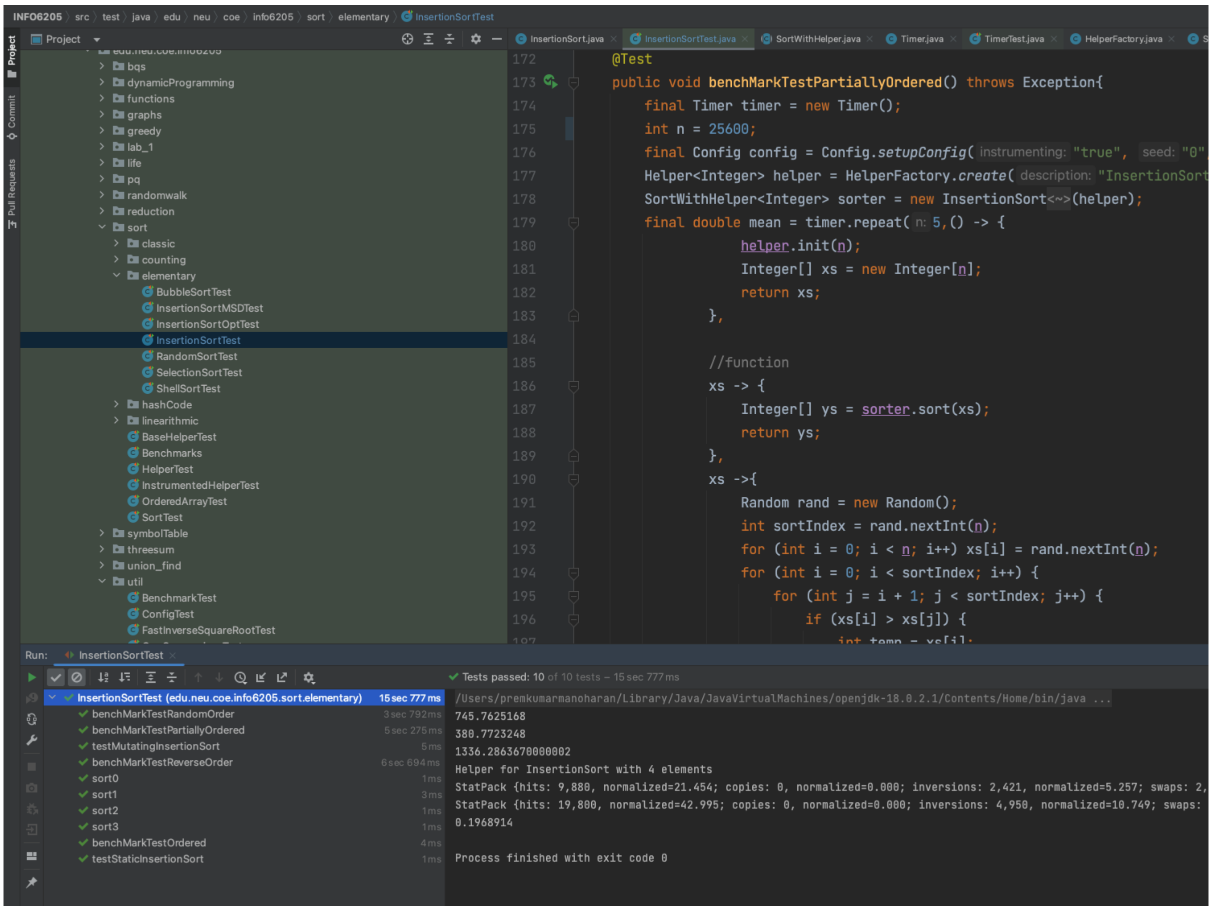
Graphical Representation:







Unit Test Screenshots:



```
@Test
public void benchMarkTestPartiallyOrdered() throws Exception{
    final Timer timer = new Timer();
    int n = 25600;
    final Config config = Config.setupConfig( instrumenting: "true",  seed: "0"
    Helper<Integer> helper = HelperFactory.create( description: "InsertionSort
    SortWithHelper<Integer> sorter = new InsertionSort<~>(helper);
    final double mean = timer.repeat( n: 5,() -> {
                helper.init(n);
                Integer[] xs = new Integer[n];
                return xs;
            },

            //function
            xs -> {
                Integer[] ys = sorter.sort(xs);
                return ys;
            },
            xs ->{
                Random rand = new Random();
                int sortIndex = rand.nextInt(n);
                for (int i = 0; i < n; i++) xs[i] = rand.nextInt(n);
                for (int i = 0; i < sortIndex; i++) {
                    for (int j = i + 1; j < sortIndex; j++) {
                        if (xs[i] > xs[j]) {
```

```
✓ Tests passed: 10 of 10 tests – 15 sec 777 ms

✓ InsertionSortTest (edu.neu.coe.info6205.sort.elementary)   15 sec 777 ms
    ✓ benchMarkTestRandomOrder                                3 sec 792 ms
    ✓ benchMarkTestPartiallyOrdered                           5 sec 275 ms
    ✓ testMutatingInsertionSort                               5 ms
    ✓ benchMarkTestReverseOrder                               6 sec 694 ms
    ✓ sort0                                                   1 ms
    ✓ sort1                                                   3 ms
    ✓ sort2                                                   1 ms
    ✓ sort3                                                   1 ms
    ✓ benchMarkTestOrdered                                    4 ms
    ✓ testStaticInsertionSort                                 1 ms
```

```
/Users/premkumarmanoharan/Library/Java/JavaVirtualMachines/openjdk-18.0.2.1/Contents/Home/bin/java ...
745.7625168
380.7723248
1336.2863670000002
Helper for InsertionSort with 4 elements
StatPack {hits: 9,880, normalized=21.454; copies: 0, normalized=0.000; inversions: 2,421, normalized=5.257; swaps: 2,
StatPack {hits: 19,800, normalized=42.995; copies: 0, normalized=0.000; inversions: 4,950, normalized=10.749; swaps:
0.1968914

Process finished with exit code 0
```