Ebenezer Ajay Williams Vincent Sankey Raj (002166250)

Program Structures & Algorithms

Fall 2021

Assignment No. 2(Benchmark)

- **Task (List of tasks performed in the Assignment)**
    1) **Code added to the Timer.java class .**
    2) **Code added to the InsertionSort.java class.**
    3) **New class called BenchmarkAssignment2.java is created.**
    4) **Experiment is performed with list sizes starting from 200 up to 6400 using the doubling method**
    5) **Graph is plotted to deduce that the worst case always takes more time than the average case which in turn takes more time than the best case.**
    6) **Proof of relationship conclusion is shown.**
    7) **Unit tests have been run and are successful.**

- **Relationship Conclusion:**

  Worst case O(n^2) > average case O(N^2) > best case O(n)

  Insertion sort is a quadratic function in time taken to sort (power of 2) (quadratic in nature)

- **Evidence to support the conclusion:**

1. **Output (Snapshot of Code output in the terminal)**

```
2021-09-26 15:02:24 INFO  Benchmark_Timer - Begin run: Insertion Sort with 100 runs
2021-09-26 15:02:24 INFO  Benchmark_Timer - Begin run: Insertion Sort with 100 runs
2021-09-26 15:02:24 INFO  Benchmark_Timer - Begin run: Insertion Sort with 100 runs
2021-09-26 15:02:24 INFO  Benchmark_Timer - Begin run: Insertion Sort with 100 runs
Average time to sort a sorted array  of size 200 is : 1.49
Average time to sort a reversed array  of size 200 is : 2.2
Average time to sort a partially sorted array  of size 200 is : 0.91
Average time to sort a random array of size 200 is : 0.7
2021-09-26 15:02:25 INFO  Benchmark_Timer - Begin run: Insertion Sort with 100 runs
2021-09-26 15:02:25 INFO  Benchmark_Timer - Begin run: Insertion Sort with 100 runs
2021-09-26 15:02:25 INFO  Benchmark_Timer - Begin run: Insertion Sort with 100 runs
2021-09-26 15:02:25 INFO  Benchmark_Timer - Begin run: Insertion Sort with 100 runs
Average time to sort a sorted array  of size 400 is : 0.62
Average time to sort a reversed array  of size 400 is : 2.14
Average time to sort a partially sorted array  of size 400 is : 1.11
Average time to sort a random array of size 400 is : 1.41
2021-09-26 15:02:25 INFO  Benchmark_Timer - Begin run: Insertion Sort with 100 runs
2021-09-26 15:02:25 INFO  Benchmark_Timer - Begin run: Insertion Sort with 100 runs
2021-09-26 15:02:26 INFO  Benchmark_Timer - Begin run: Insertion Sort with 100 runs
2021-09-26 15:02:26 INFO  Benchmark_Timer - Begin run: Insertion Sort with 100 runs
Average time to sort a sorted array  of size 800 is : 0.34
Average time to sort a reversed array  of size 800 is : 3.16
Average time to sort a partially sorted array  of size 800 is : 0.82
Average time to sort a random array of size 800 is : 2.09
2021-09-26 15:02:26 INFO  Benchmark_Timer - Begin run: Insertion Sort with 100 runs
2021-09-26 15:02:26 INFO  Benchmark_Timer - Begin run: Insertion Sort with 100 runs
2021-09-26 15:02:27 INFO  Benchmark_Timer - Begin run: Insertion Sort with 100 runs
2021-09-26 15:02:27 INFO  Benchmark_Timer - Begin run: Insertion Sort with 100 runs
Average time to sort a sorted array  of size 1600 is : 0.44
Average time to sort a reversed array  of size 1600 is : 9.63
Average time to sort a partially sorted array  of size 1600 is : 1.47
Average time to sort a random array of size 1600 is : 3.89
```

Code modified to get the required output on the terminal

- **BenchmarkAssignment2.java is the main method used to call perform the experiment. (code at the end of this document)**

- repeat() method

```
public <T, U> double repeat(int n, Supplier<T> supplier, Function<T, U> function, UnaryOperator<T> preFunction, Con
    logger.trace("repeat: with " + n + " runs");
    double returnVal = 0d;
    for(int i=0;i<n;i++){
        pause();
        if(Optional.ofNullable(preFunction).isPresent())
            preFunction.apply(supplier.get());
        resume();
        U retVal=function.apply(supplier.get());
        if(i<n-1)
            pauseAndLap();
        else
            returnVal =stop();
        if(Optional.ofNullable(postFunction).isPresent())
            postFunction.accept(retVal);
        resume();
    }
    return returnVal;
}
```

- getClock() method

```
private static long getClock() {
    return System.nanoTime();
}
```

- toMillissecs() method (calls the randomMove() method m number of times)

```
private static double toMillisecs(long ticks) {
    return (double)TimeUnit.NANOSECONDS.toMillis(ticks);
}
```

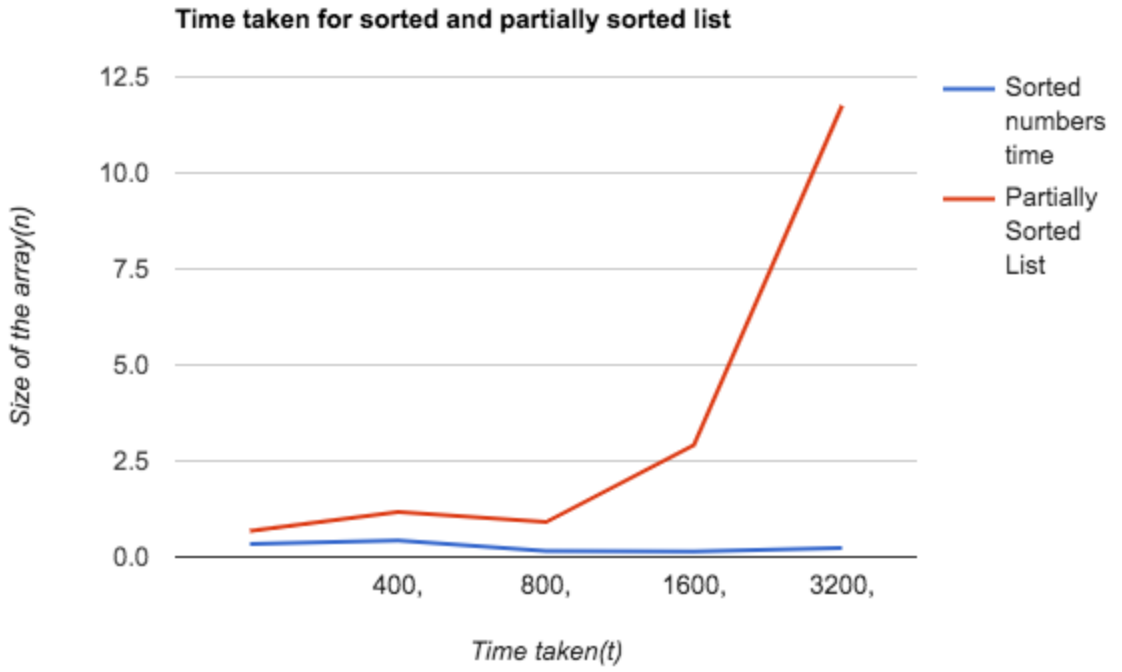
- sort() method in InsertionSort.java

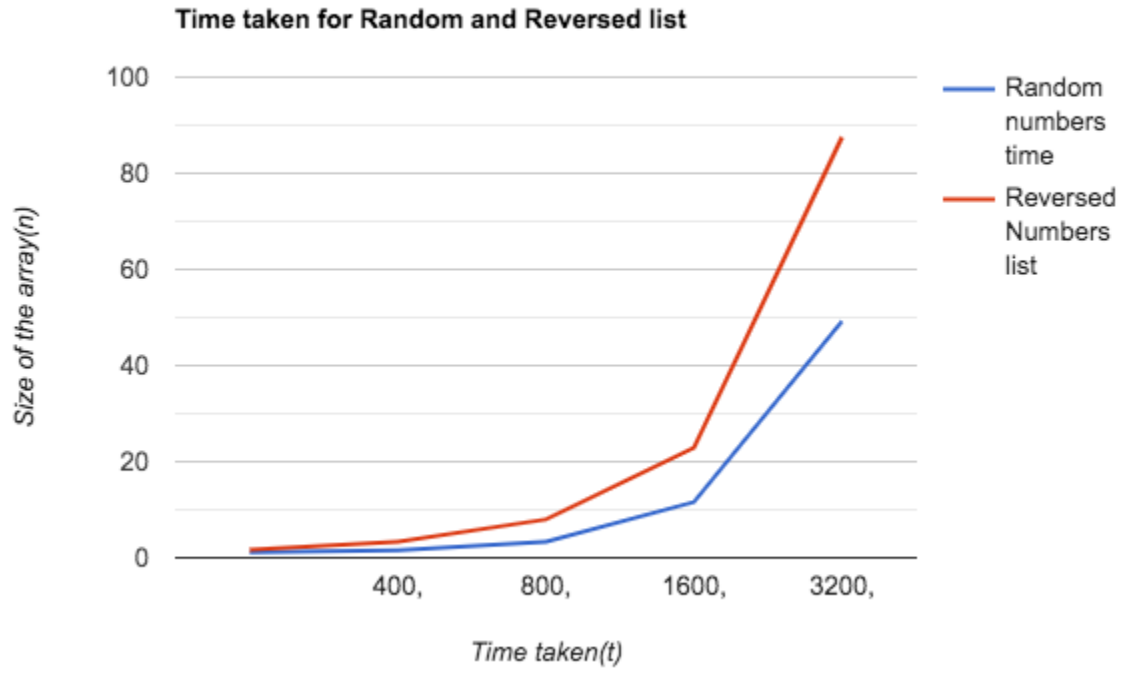
```
public void sort(X[] xs, int from, int to) {
    final Helper<X> helper = getHelper();
    int i = from +1;
    while(i<to){
        int j = i-1;
        while(j>=from){
            if(helper.compare(xs[j],xs[j+1])==1)
                helper.swap(xs,j, j: j+1);
            else
                break;
            j--;
        }
        i++;
    }
}
```

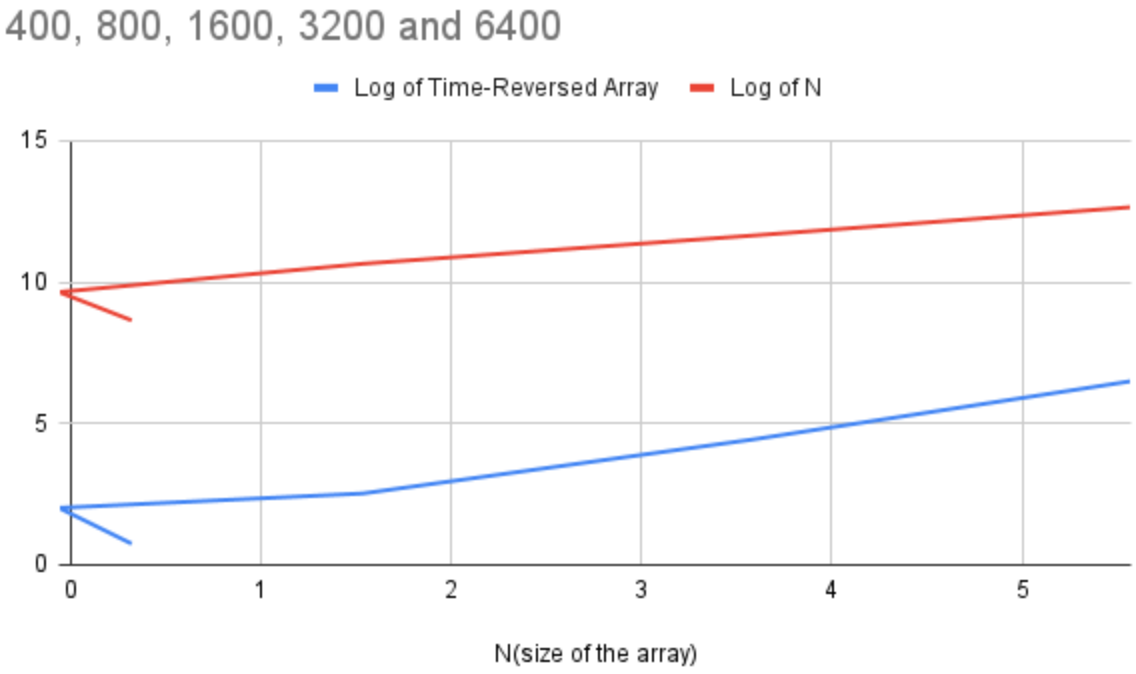
2. Graphical Representation

Line graph showing the relationship between the time taken to sort array of length n when the array is sorted vs partially sorted and Random vs reversed list.(the graph has been split into 2 inorder to adjust to the scale).

Line graph between the Log of N(size of the array) and Log of T(time taken to sort the reverse array) is drawn.


Time taken for sorted and partially sorted list
12.5
10.0
7.5
5.0
2.5
0.0
Size of the array(n)
400,
800,
1600,
3200,
Time taken(t)
Sorted numbers time
Partially Sorted List


Time taken for Random and Reversed list
100
80
60
40
20
0
Size of the array(n)
400,
800,
1600,
3200,
Time taken(t)
Random numbers time
Reversed Numbers list



We can see that O(n^2) > average case O(N^2) > best case O(n) and that the slope of the graph between two points in approximately 2.

Tabulated values:

N(size of the array)	Sorted Array Time	Reversed Time	Partially Sorted Time	Random List Time
400	0.47	1.68	0.97	1.25
800	0.42	4.04	0.69	0.96
1600	0.24	5.75	0.93	2.9
3200	0.15	21.63	3.01	12.02
6400	0.28	90.03	11.86	47.53

Log of Time-Random Array	Log of Time-Reversed Array	Log of N
0.3219280949	0.748461233	8.64385619
-0.05889368905	2.014355293	9.64385619
1.5360529	2.523561956	10.64385619
3.587364991	4.43496176	11.64385619
5.570766497	6.492333915	12.64385619

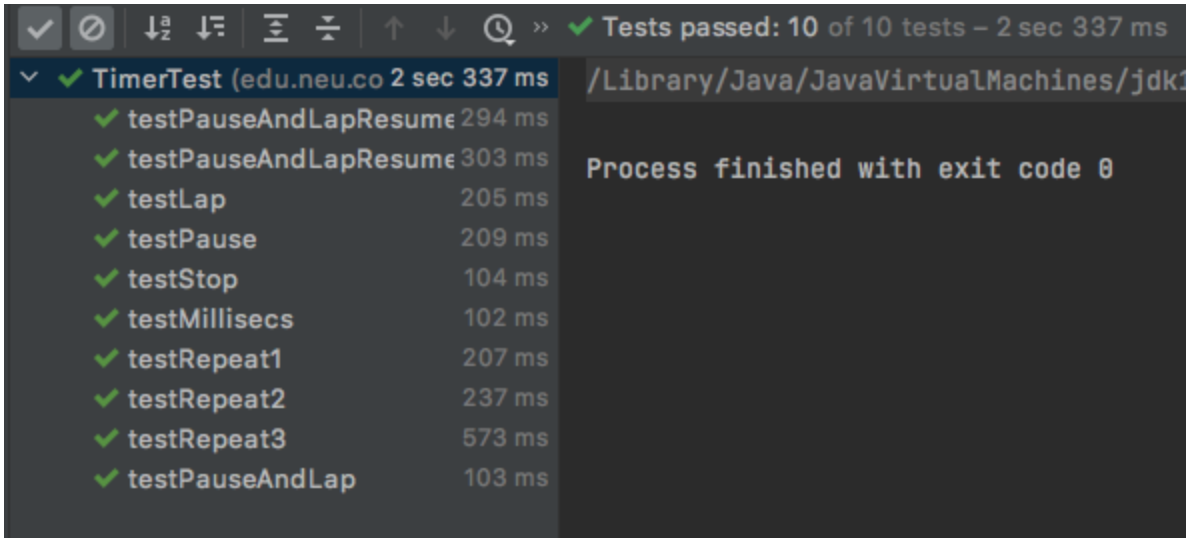
Proof for the relation.

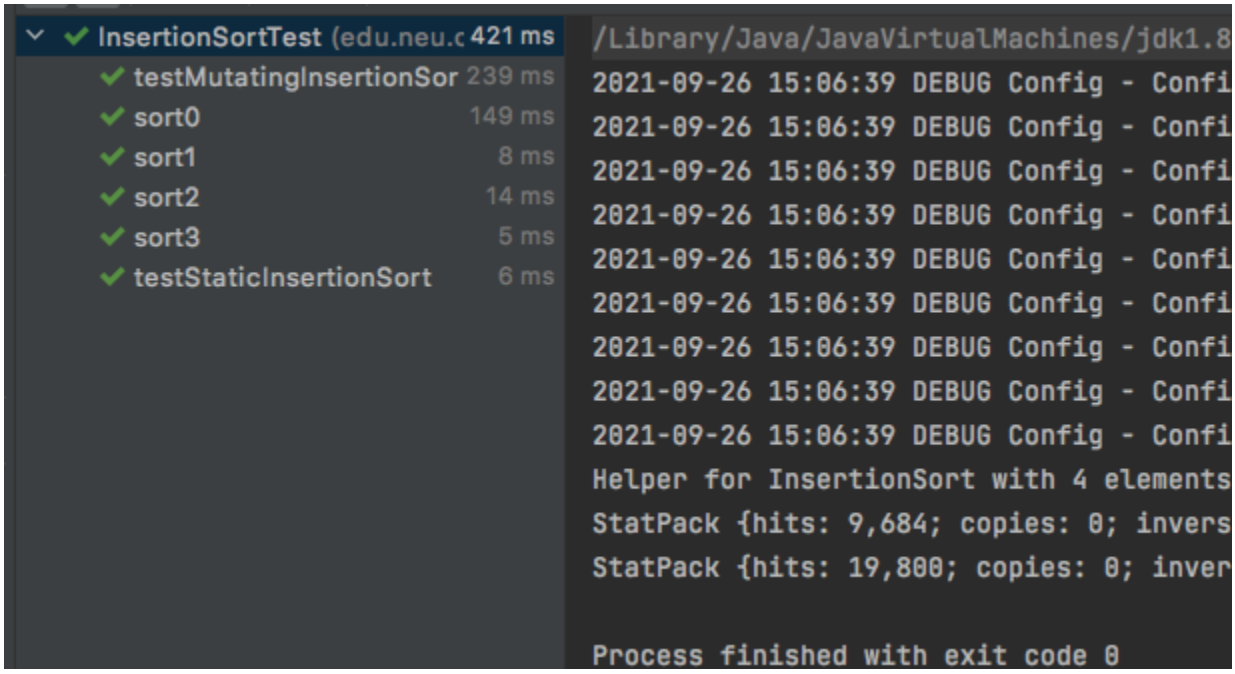
As seen in the values of the above table and from the graph above, it can be observed that as the size of the array is doubled, the time taken is quadrupled (multiplied by 4). (Some of the above values are distorted because of my cpu speed, but it changes every time i run it and i was able to observe it).

Calculating the slope between two points we can see that the slope lies between 1.6 - 2.3 which is approximately 2.

With this we can conclude that Insertion sort is quadratic in Nature.

- **Unit tests result:(Snapshot of successful unit test run)**





Final code of BenchmarkAssignment2.java

```
package edu.neu.coe.info6205.assignment;

import edu.neu.coe.info6205.sort.elementary.InsertionSort;
import edu.neu.coe.info6205.util.Benchmark_Timer;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class BenchmarkAssignment2 {
```

```
    private static Benchmark_Timer<Integer[]> btm;

    public static Double log2(Double x)
    {
        return  Math.log(x) ;
    }


    public static void main(String[] args) {
        btm = new Benchmark_Timer<Integer[]>("Insertion Sort",(x)-> new
InsertionSort().sort(x,true));
        List<Double> sortedList = new ArrayList<Double>();
        List<Double> reversedList = new ArrayList<Double>();
        List<Double> partiallySortedList = new ArrayList<Double>();
        List<Double> randomList = new ArrayList<Double>();

        List<Integer> nList = new ArrayList<Integer>();

        for( int i = 100; i <= 6400 ; i*=2){
            Integer[] baseArr = new Integer[i];
            nList.add(i);

            for(int j = 0 ; j < i ; j++){
                baseArr[j] = j;
            }
            Integer[] sortedArr = new Integer[i];
            Integer[] reversedArr = new Integer[i];
            Integer[] partiallysortedArr = new Integer[i];
            Integer[] randomArr = new Integer[i];

            Random random = new Random();

            //sorted array
            sortedArr = baseArr;

            //reversed array
            for(int j = 0 ; j < i; j++){
                reversedArr[i -1-j] = j;
            }

            // partially sorted
            for(int j=0;j<i/2;j++){
                partiallysortedArr[j]=0;
            }
            for( int j = i/2 ; j < i ; j++){
                partiallysortedArr[j] = random.nextInt(i)+1;
            }
```

```
            //random array
            for(int j = 0 ; j < i; j++){
                randomArr[j] = random.nextInt(i);
            }

            double avgTimeTakenSorted = btm.run(sortedArr,100);
            double avgTimeTakenReversed = btm.run(reversedArr,100);
            double avgTimeTakenPartial = btm.run(partiallysortedArr,100);
            double avgTimeTakenRandom = btm.run(randomArr,100);

            sortedList.add(avgTimeTakenSorted);
            randomList.add(avgTimeTakenRandom);
            partiallySortedList.add(avgTimeTakenPartial);
            reversedList.add(avgTimeTakenReversed);
            System.out.println("Average time to sort a sorted array  of size " +
i + " is : " + avgTimeTakenSorted);
            System.out.println("Average time to sort a reversed array  of size "
+ i + " is : " + avgTimeTakenReversed);
            System.out.println("Average time to sort a partially sorted array
of size " + i + " is : " + avgTimeTakenPartial);
            System.out.println("Average time to sort a random array of size " +
i + " is : " + avgTimeTakenRandom);


        }
        System.out.println("List of size of N : " + nList);
        System.out.println("Sorted Time List :" + sortedList);
        System.out.println("Reversed Time List :" + reversedList);
        System.out.println("Partially Sorted Time List :" +
partiallySortedList);
        System.out.println("Random Time List :" + randomList);

    }
}
```