SHIVANI CHAVAN (NUID – 001582611)

Program Structures and Algorithms

Spring 2022

Assignment Number: 2

Task:

- Fix the Timer Class
- Execute the test cases in BenchmarkTest.java, TimerTest.java, InsertionSortTest.java
- Implement InsertionSort using the helper function
- Write a main method which generates arrays of different sizes through the method of doubling
- Derive conclusions based on observations and evidence

Section: Code Changes Snapshots

A) Timer Class

repeat() method

```
public <T, U> double repeat(int n, Supplier<T> supplier, Function<T, U> function, UnaryOperator<T> preFunction, Consumer<U> postFunction) {
    // FIXME: note that the timer is running when this method is called and should still be running when it returns.
    //  by replacing the following code
    logger.trace("repeat: with " + n + " runs");
    double avgMilSecs=0;
    pause();
    for(int repeat=1;repeat<=n;repeat++){
        if(preFunction != null)
            preFunction.apply(supplier.get());
        resume();
        U retVal=function.apply(supplier.get());
        if(repeat<n)
            pauseAndLap();
        else
            avgMilSecs=stop();
        if(postFunction!=null)
            postFunction.accept(retVal);
    }
    resume();
    return avgMilSecs;
}
```

getClock() method

```
private static long getClock() {
    // FIXME by replacing the following code
     return System.nanoTime();
    // END
}
```

toMillisecs()

```
private static double toMillisecs(long ticks) {
    // FIXME by replacing the following code
    return ticks/1000000L;
}
```

B) sort() method of Insertion Sort

```
public void sort(X[] xs, int from, int to) {
    final Helper<X> helperSort = getHelper();
    for(int iVar=from + 1; iVar<to;iVar++){
        for(int jVar = iVar - 1;jVar>=from;jVar--)
        {
            if(helperSort.compare(xs[jVar],xs[jVar+1]) == 1)
                helperSort.swap(xs,jVar, j: jVar+1);
            else
                break;
        }
    }
    // END
}
```

C) Driver class

```
public class BenchmarkAssignment {

    public static void main(String[] args){

        Benchmark_Timer benchmark_timer = new Benchmark_Timer<Integer[]>( description: "Benchmarking on insertion sort",
                (Integer[] arraySort)->{
                    new InsertionSort().sort(arraySort, makeCopy: true);
                });

        int[] arrayLength = {200,400,800,1600,3200};

        for (ArrayTypes Atype: ArrayTypes.values()) {
            System.out.println("\n");
            for (int inc = 0; inc < arrayLength.length; inc++) {
                Integer[] arraySort;
                arraySort = genArray(Atype, arrayLength[inc]);
                double avgMsTaken = benchmark_timer.run(arraySort,  m: 100);
                System.out.println("Average time taken to sort a " + Atype + " array of size n " + arrayLength[inc] +
                        " is " + avgMsTaken + "ms");
            }
        }

    }
```

```java
public static Integer[] genArray(ArrayTypes type, int size){
    Integer[] array = new Integer[size];
    Random random = new Random();
    switch (type){
        case RANDOM:
            for(int j=0;j<size;j++){
                array[j] =random.nextInt();
            }
            break;
        case ORDERED:
            for(int j=0;j<size;j++){
                array[j] =j;
            }
            break;
        case REVERSE_ORDERED:
            for(int j=0;j<size;j++){
                array[j] =size-j;
            }
            break;
        case PARTIALLY_ORDERED:
            int mid = size/2;
            for(int j=0;j<size;j++){
                if(j<mid)
                    array[j]=j;
                else
                    array[j]=mid-1 + random.nextInt( bound: Integer.MAX_VALUE-mid+1);
            }
            break;
    }
    return array;
}

}
```

Section – Code Changes File Upload

A) Timer class

Timer.java

B) Driver Class

BenchmarkAssignment.java

C) Insertion Sort

InsertionSort.java

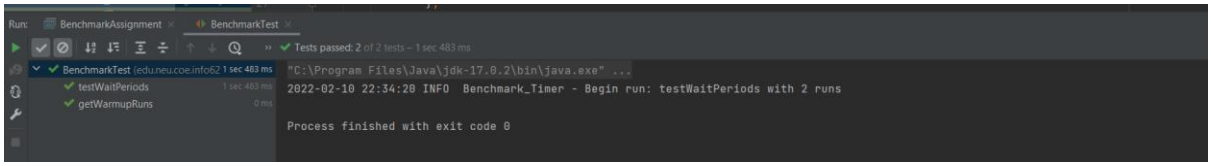
Section – Output Snapshots:

```
2022-02-10 22:37:42 INFO  Benchmark_Timer - Begin run: Benchmarking on insertion sort with 100 runs
Avg. time taken to sort a RANDOM array of size 200 is 1.14ms
2022-02-10 22:37:42 INFO  Benchmark_Timer - Begin run: Benchmarking on insertion sort with 100 runs
Avg. time taken to sort a RANDOM array of size 400 is 1.24ms
2022-02-10 22:37:42 INFO  Benchmark_Timer - Begin run: Benchmarking on insertion sort with 100 runs
Avg. time taken to sort a RANDOM array of size 800 is 1.53ms
2022-02-10 22:37:42 INFO  Benchmark_Timer - Begin run: Benchmarking on insertion sort with 100 runs
Avg. time taken to sort a RANDOM array of size 1600 is 2.9ms
2022-02-10 22:37:43 INFO  Benchmark_Timer - Begin run: Benchmarking on insertion sort with 100 runs
Avg. time taken to sort a RANDOM array of size 3200 is 9.24ms


2022-02-10 22:37:44 INFO  Benchmark_Timer - Begin run: Benchmarking on insertion sort with 100 runs
Avg. time taken to sort a PARTIALLY_ORDERED array of size 200 is 0.3ms
2022-02-10 22:37:44 INFO  Benchmark_Timer - Begin run: Benchmarking on insertion sort with 100 runs
Avg. time taken to sort a PARTIALLY_ORDERED array of size 400 is 0.25ms
2022-02-10 22:37:44 INFO  Benchmark_Timer - Begin run: Benchmarking on insertion sort with 100 runs
Avg. time taken to sort a PARTIALLY_ORDERED array of size 800 is 0.44ms
2022-02-10 22:37:44 INFO  Benchmark_Timer - Begin run: Benchmarking on insertion sort with 100 runs
Avg. time taken to sort a PARTIALLY_ORDERED array of size 1600 is 0.84ms
2022-02-10 22:37:44 INFO  Benchmark_Timer - Begin run: Benchmarking on insertion sort with 100 runs
Avg. time taken to sort a PARTIALLY_ORDERED array of size 3200 is 2.41ms
```

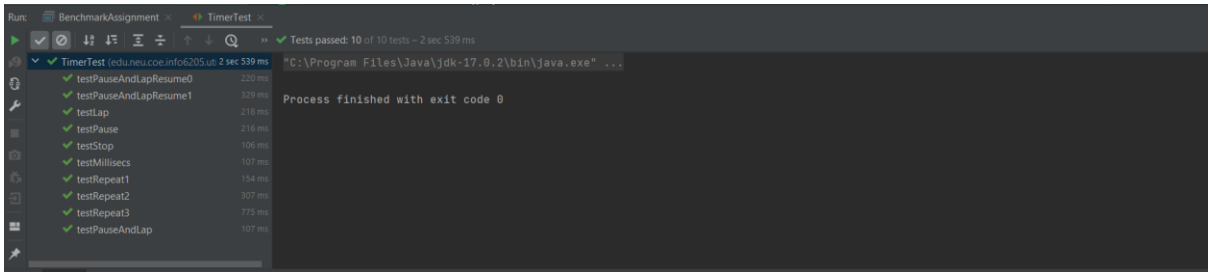
```
2022-02-10 22:37:44 INFO  Benchmark_Timer - Begin run: Benchmarking on insertion sort with 100 runs
Avg. time taken to sort a ORDERED array of size 200 is 0.22ms
2022-02-10 22:37:44 INFO  Benchmark_Timer - Begin run: Benchmarking on insertion sort with 100 runs
Avg. time taken to sort a ORDERED array of size 400 is 0.22ms
2022-02-10 22:37:44 INFO  Benchmark_Timer - Begin run: Benchmarking on insertion sort with 100 runs
Avg. time taken to sort a ORDERED array of size 800 is 0.23ms
2022-02-10 22:37:44 INFO  Benchmark_Timer - Begin run: Benchmarking on insertion sort with 100 runs
Avg. time taken to sort a ORDERED array of size 1600 is 0.27ms
2022-02-10 22:37:44 INFO  Benchmark_Timer - Begin run: Benchmarking on insertion sort with 100 runs
Avg. time taken to sort a ORDERED array of size 3200 is 0.22ms


2022-02-10 22:37:44 INFO  Benchmark_Timer - Begin run: Benchmarking on insertion sort with 100 runs
Avg. time taken to sort a REVERSE_ORDERED array of size 200 is 0.27ms
2022-02-10 22:37:44 INFO  Benchmark_Timer - Begin run: Benchmarking on insertion sort with 100 runs
Avg. time taken to sort a REVERSE_ORDERED array of size 400 is 0.51ms
2022-02-10 22:37:44 INFO  Benchmark_Timer - Begin run: Benchmarking on insertion sort with 100 runs
Avg. time taken to sort a REVERSE_ORDERED array of size 800 is 1.35ms
2022-02-10 22:37:45 INFO  Benchmark_Timer - Begin run: Benchmarking on insertion sort with 100 runs
Avg. time taken to sort a REVERSE_ORDERED array of size 1600 is 4.84ms
2022-02-10 22:37:45 INFO  Benchmark_Timer - Begin run: Benchmarking on insertion sort with 100 runs
Avg. time taken to sort a REVERSE_ORDERED array of size 3200 is 18.97ms
```

Section – Test Cases

A) BenchmarkTest



B) TimerTest



C) InsertionSortTest

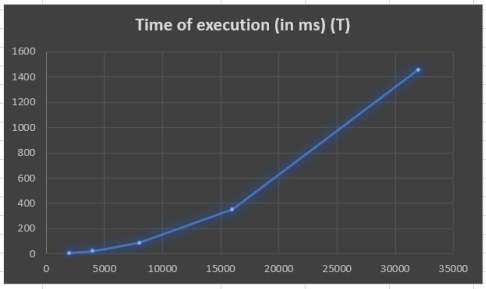
Section – Observations

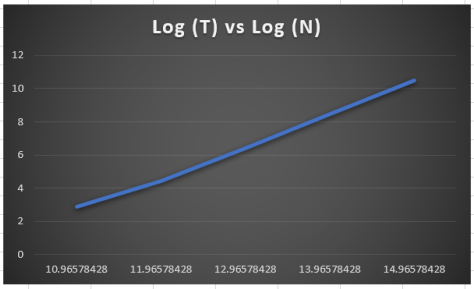
A) Random Array

Conclusions:

- When we double the size of the array in observation, the time taken to sort the array increases approximately by a factor of four as seen highlighted in yellow in the ratio column.
- Also, the slope of the plot Log(T) vs Log(N) is approximately 2.
- Therefore, we can say that the order of growth for insertion sort of a randomly-ordered array is quadratic in nature.

Array Size (N)	Time of execution (in ms) (T)	Log (N)	Log(T)	Slope	Ratio
2000	7.34	10.96578428	2.875780063	1.575101252	
4000	21.87	11.96578428	4.450881315	1.966633688	2.979564
8000	85.48	12.96578428	6.417515003	2.04727577	3.908551
16000	353.31	13.96578428	8.464790773	2.042389402	4.133248
32000	1455.38	14.96578428	10.50718017	0.702080157	4.119272

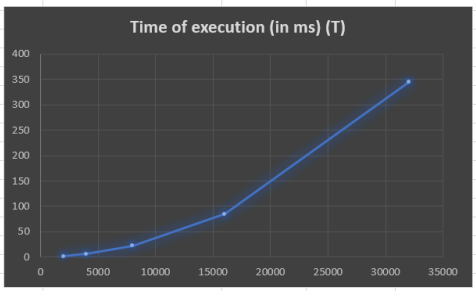


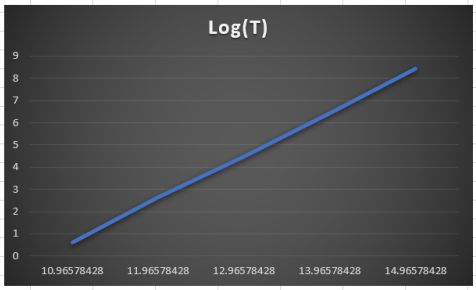


B) PARTIALLY ORDERED

- We can observe from the execution time T, that sorting of a partially ordered array is faster than randomly sorted arrays and reverse sorted arrays.
- The observations are similar to that of a randomly sorted array wherein the slope is approximately coming as 2.
- Also, the execution time increases as a factor of 4.

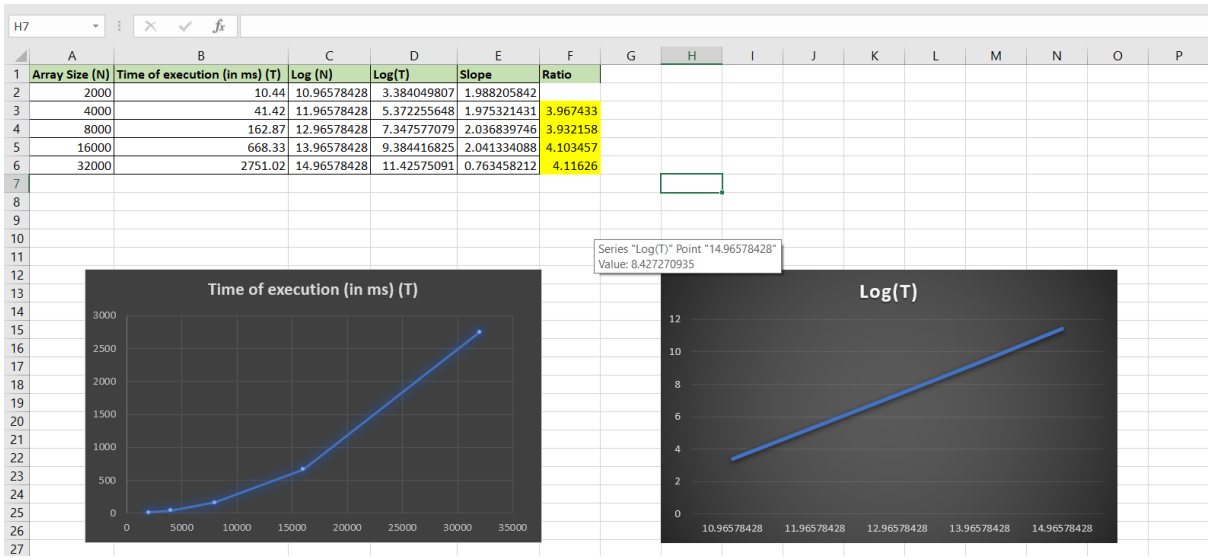
Array Size (N)	Time of execution (in ms) (T)	Log (N)	Log(T)	Slope	Ratio
2000	1.53	10.96578428	0.613531653	2.050951187	
4000	6.34	11.96578428	2.66448284	1.833768027	4.143791
8000	22.6	12.96578428	4.498250868	1.910291175	3.564669
16000	84.95	13.96578428	6.408542042	2.018728893	3.75885
32000	344.24	14.96578428	8.427270935	0.563102526	4.052266





C) REVERSE ORDERED

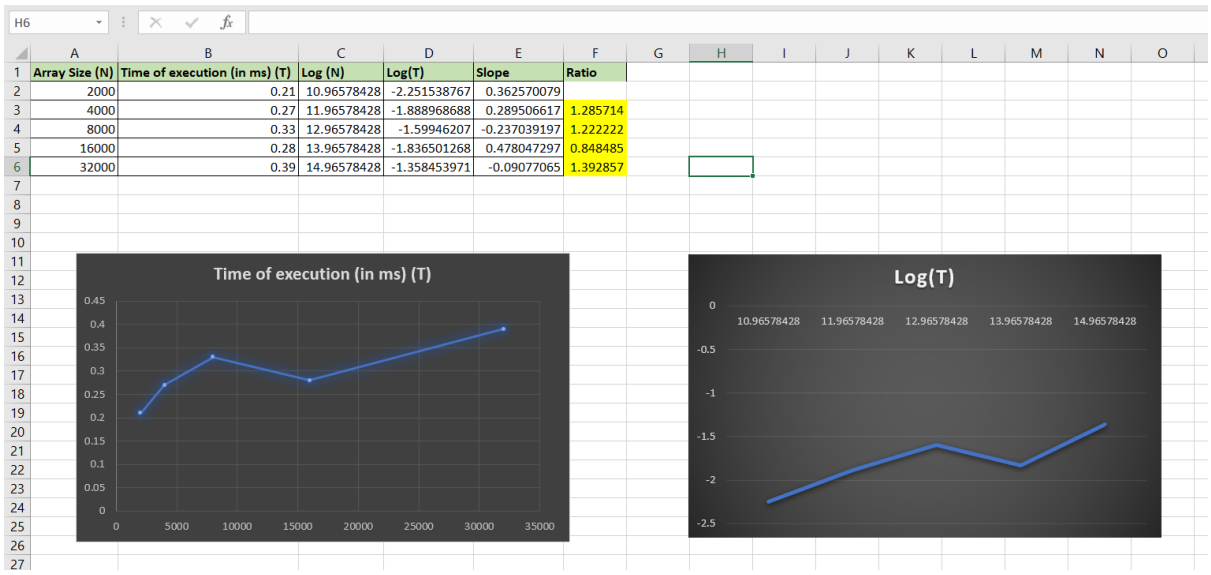
- The execution time increases by a factor of four
- The slope of the plot is coming down to 2.
- It takes more time to sort a reverse ordered array than a random array.



Array Size (N)	Time of execution (in ms) (T)	Log (N)	Log(T)	Slope	Ratio
2000	10.44	10.96578428	3.384049807	1.988205842	
4000	41.42	11.96578428	5.372255648	1.975321431	3.967433
8000	162.87	12.96578428	7.347577079	2.036839746	3.932158
16000	668.33	13.96578428	9.384416825	2.041334088	4.103457
32000	2751.02	14.96578428	11.42575091	0.763458212	4.11626

D) ORDERED

- When the array size doubles, the execution time is not increasing as a factor of 4. There is a slight increase in the running time as we increase the size.
- The slope of T vs N appears linear as opposed to quadratic
- Insertion sorting of a sorted array is the fastest



Array Size (N)	Time of execution (in ms) (T)	Log (N)	Log(T)	Slope	Ratio
2000	0.21	10.96578428	-2.251538767	0.362570079	
4000	0.27	11.96578428	-1.888968688	0.289506617	1.285714
8000	0.33	12.96578428	-1.59946207	-0.237039197	1.222222
16000	0.28	13.96578428	-1.836501268	0.478047297	0.848485
32000	0.39	14.96578428	-1.358453971	-0.09077065	1.392857

CONCLUSION –

For insertion sort of an array of the same size, the speed from fastest to slowest would be – sorted arrays, partially ordered, random arrays and reverse ordered arrays

The order of growth of execution time is quadratic for reverse ordered, random and partially ordered arrays.

We can say that insertion sort is suitable for partially ordered arrays. The worst case is if the array is reverse ordered.