# CSE 41321 Homework #1


**Taylor Allen**


**1-20-2020**

1. Insert method:

```java
/**
* Algorithm to insert a number into a new array by copying a new array, allocating a new slot, and
inserting
* @param array Original array of integers
* @param index Location where value will be inserted
* @param value Value to be inserted
* @return Old array with value inserted into it
*/

// O(n)
static int[] insert(int array[], int index, int value) {
  // Create new array one larger than original array
  int newArray[] = new int[array.length + 1]; // O(1)

  // Copy elements up to insert point from original array to new array
  for(int i = 0; i < index; i++) { // O(n)
     newArray[i] = array[i]; // O(1)
  }

  // Place insert value into new array
  newArray[index] = value; // O(n)

  // Copy elements after insert point from original array to new array
  for(int i = index; i < array.length; i++) { // O(n)
     newArray[i+1] = array[i]; // O(1)
  }

  return newArray; // O(1)
}
```

2. Main method:

```java
// O(n^3)
public static void main(String[] args) {
```

```java
    Random r = new Random(); // O(1)

    // Setting to allow fine-tuning the granularity of the readings
    final int NUM_READINGS = 60; // O(1)
    final int INSERTS_PER_READING = 10000; // O(1)
    final int NANO_SECONDS_PER_SECOND = 1000000000; // O(1)

    // Start with an array containing 1 element
    int[] array = {0}; // O(1)

    // Take NUM_READINGS readings
    for(int i = 0; i < NUM_READINGS; i++) { // O(n^3)
      // Each reading will be taken after INSERTS_PER_READING inserts
      double startTime = System.nanoTime(); // O(1)
      for(int x = 0; x < INSERTS_PER_READING; x++) { // O(n^2)
        int index = r.nextInt(array.length); // O(n)
        int value = r.nextInt(); // O(1)
        array = Homework1.insert(array, index, value); // O(n)
      }
      double stopTime = System.nanoTime(); // O(1)
      double timePerInsert = (stopTime - startTime) / INSERTS_PER_READING; // O(1)

      // Output reading in tabular format
      // array.length was -1 so added +1 for formatting purposes
      System.out.println("Array Length: " + (array.length - 1) + "\t\tSeconds Per Insert: " +
timePerInsert / NANO_SECONDS_PER_SECOND); // O(1)
    }

}
```

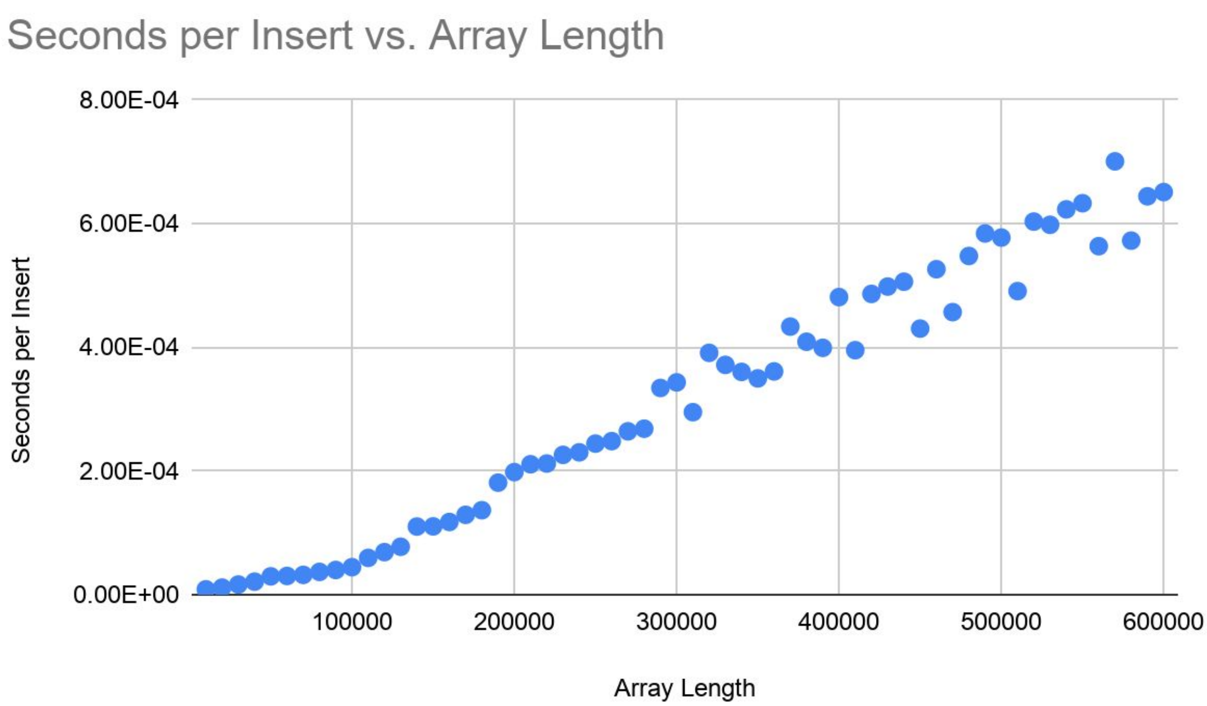
Output:

```
Array Length: 10000		Seconds Per Insert: 8.70958E-6
Array Length: 20000		Seconds Per Insert: 1.1479229999999999E-5
Array Length: 30000		Seconds Per Insert: 1.6470799999999998E-5
Array Length: 40000		Seconds Per Insert: 2.114018E-5
Array Length: 50000		Seconds Per Insert: 2.9702669999999997E-5
Array Length: 60000		Seconds Per Insert: 3.034581E-5
Array Length: 70000		Seconds Per Insert: 3.207017E-5
Array Length: 80000		Seconds Per Insert: 3.691808E-5
Array Length: 90000		Seconds Per Insert: 3.999754E-5
Array Length: 100000		Seconds Per Insert: 4.454989E-5
Array Length: 110000		Seconds Per Insert: 5.9540379999999995E-5
Array Length: 120000		Seconds Per Insert: 6.884544E-5
```

| | |
|---|---|
| Array Length: 130000 | Seconds Per Insert: 7.757959E-5 |
| Array Length: 140000 | Seconds Per Insert: 1.1022546E-4 |
| Array Length: 150000 | Seconds Per Insert: 1.1049584E-4 |
| Array Length: 160000 | Seconds Per Insert: 1.1763497999999999E-4 |
| Array Length: 170000 | Seconds Per Insert: 1.2918436E-4 |
| Array Length: 180000 | Seconds Per Insert: 1.3667217E-4 |
| Array Length: 190000 | Seconds Per Insert: 1.8122926E-4 |
| Array Length: 200000 | Seconds Per Insert: 1.9825574E-4 |
| Array Length: 210000 | Seconds Per Insert: 2.1095612E-4 |
| Array Length: 220000 | Seconds Per Insert: 2.1213794E-4 |
| Array Length: 230000 | Seconds Per Insert: 2.261389E-4 |
| Array Length: 240000 | Seconds Per Insert: 2.3026535999999998E-4 |
| Array Length: 250000 | Seconds Per Insert: 2.4434462E-4 |
| Array Length: 260000 | Seconds Per Insert: 2.4835452E-4 |
| Array Length: 270000 | Seconds Per Insert: 2.6421778000000005E-4 |
| Array Length: 280000 | Seconds Per Insert: 2.6847182000000003E-4 |
| Array Length: 290000 | Seconds Per Insert: 3.3432138E-4 |
| Array Length: 300000 | Seconds Per Insert: 3.433738E-4 |
| Array Length: 310000 | Seconds Per Insert: 2.9517687E-4 |
| Array Length: 320000 | Seconds Per Insert: 3.9112259E-4 |
| Array Length: 330000 | Seconds Per Insert: 3.7164008E-4 |
| Array Length: 340000 | Seconds Per Insert: 3.6029528E-4 |
| Array Length: 350000 | Seconds Per Insert: 3.4982302E-4 |
| Array Length: 360000 | Seconds Per Insert: 3.6109947999999996E-4 |
| Array Length: 370000 | Seconds Per Insert: 4.3356129E-4 |
| Array Length: 380000 | Seconds Per Insert: 4.0908647E-4 |
| Array Length: 390000 | Seconds Per Insert: 3.9923777E-4 |
| Array Length: 400000 | Seconds Per Insert: 4.8127917999999997E-4 |
| Array Length: 410000 | Seconds Per Insert: 3.9541692999999997E-4 |
| Array Length: 420000 | Seconds Per Insert: 4.8642253E-4 |
| Array Length: 430000 | Seconds Per Insert: 4.9830762E-4 |
| Array Length: 440000 | Seconds Per Insert: 5.0616744E-4 |
| Array Length: 450000 | Seconds Per Insert: 4.3030890999999996E-4 |
| Array Length: 460000 | Seconds Per Insert: 5.2636209E-4 |
| Array Length: 470000 | Seconds Per Insert: 4.5696527E-4 |
| Array Length: 480000 | Seconds Per Insert: 5.4775541E-4 |
| Array Length: 490000 | Seconds Per Insert: 5.841426700000001E-4 |
| Array Length: 500000 | Seconds Per Insert: 5.775200600000001E-4 |
| Array Length: 510000 | Seconds Per Insert: 4.9094159E-4 |
| Array Length: 520000 | Seconds Per Insert: 6.0336583E-4 |
| Array Length: 530000 | Seconds Per Insert: 5.9817282E-4 |
| Array Length: 540000 | Seconds Per Insert: 6.2314002E-4 |
| Array Length: 550000 | Seconds Per Insert: 6.330716E-4 |

| | |
|---|---|
| Array Length: 560000 | Seconds Per Insert: 5.6358866E-4 |
| Array Length: 570000 | Seconds Per Insert: 7.0079973E-4 |
| Array Length: 580000 | Seconds Per Insert: 5.725832299999999E-4 |
| Array Length: 590000 | Seconds Per Insert: 6.4425056E-4 |
| Array Length: 600000 | Seconds Per Insert: 6.5110074E-4 |

3. Profiling data:




4. The Big-O complexity of my implementation of the insert method is O(n) (linear). The Big-O complexity of my main method was O(n^2). In my insert method, the for loop contributes to the linear time complexity. This is because the for loop runs for every n times we go from the zeroth index to the “index” index, and it runs every n times when the code goes from the “index” index to the end of the array index. The main method is exponential because of the nested for loop. This method is O(n^3), however, the method is technically O(n^2) because we drop the extra exponent.
5. The performance of the algorithm degrades as the array length grows. This is because of the exponential complexity of the implementation. This means that as the array length grows, the time to run the algorithm exponentially increases. This is not good in terms of time complexity because in the case we have only ~600,000 items in our array at a time, however, when the array may be significantly larger, the time to run the algorithm will take a hit.