Comparing InsertionSort Run Times for Varying Input Sizes
Lab Assignment #2

by
Justin Anthony Timberlake

CS 303 Algorithms and Data Structures
September 6, 2017

1. **Problem Specification**

The goal of this lab was to implement and test the given algorithm for insertionSort as provided in our textbook. The next goal was to compare run times of said sorting algorithm for varying input sizes with lists provided in the assignment and display these results.

2. **Program Design**

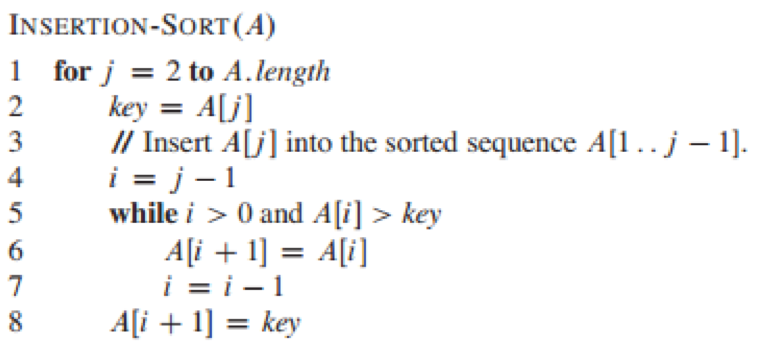
As with the previous lab, the design for this lab was also fairly simple. Apart from the main() method, there were only 3 other methods which needed to be created.

The steps taken to develop the program were:

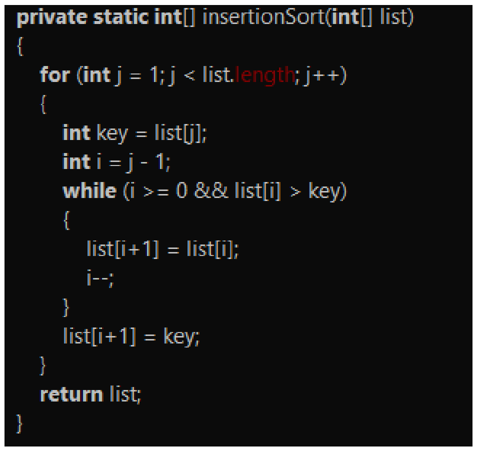
a) Copy over the readInNums method I wrote in the previous assignment to be reused for reading in the inputs once more.

b) Develop the printArray method, which was used in the lab to show the TA the outputs of my original input array as well as the difference in the sorted array. However, this method would only serve a purpose in the in-class portion of the assignment as larger values of array sizes would not display properly in the standard Java output terminal within Eclipse.

c) Develop the insertionSort method itself. This was done by translating directly from the pseudo-code provided and making only minor adjustments to match for 0 base counting as opposed to the book's standard 1-base counting. In other words, the referenced indices sometimes had to be adjusted to fit the actual programming language (Java). Specifically, in lines 1 and 5, "j = 2" was implemented as "j = 1" and "i > 0" was implemented as "i >= 0" as seen in the provided algorithm and my implementation of the algorithm using Java.



```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j - 1].
4      i = j - 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i - 1
8      A[i + 1] = key
```



```
private static int[] insertionSort(int[] list)
{
   for (int j = 1; j < list.length; j++)
   {
      int key = list[j];
      int i = j - 1;
      while (i >= 0 && list[i] > key)
      {
         list[i+1] = list[i];
         i--;
      }
      list[i+1] = key;
   }
   return list;
}
```

d) Fill in the appropriate portions of text to display the unsorted and sorted versions of the list of 100 integers provided in class.

e) Remove the text prompts and add the calculations for the run times to main() around the calling of the insertionSort method. The run time is calculated in milliseconds exactly as it was in the previous assignment. The new printout for the homework section of this assignment only displays how many values are being sorted and the elapsed time for each size of the array.

The following methods were used in this lab:

a) readInNums (Scanner derp, int size) : receives a Scanner object, locally referred to as "derp" as well as a value for the size of the intended read in array to accommodate each input text file. In the in-class portion, the size argument was not included because we were only testing for 100 values in the given array.

b) printArray(int [] list) : prints any given array of integers sent to it.

c) insertionSort(int [] list): takes an unsorted list of integers of any size and sorts the values in Θ(n^2) time. This is done by following the algorithm previously referenced. In an outer loop using the variable "j" for indexing, from the value of the second position to the length of the given list, a "key" value is assigned the value of the value at the index "j" followed by assigning another value "i" to "j-1". Then, in an inner loop, for every value of j, while "i" >= 0 and the value at the index "i" of the list are greater than 0, the value at the position "i+1" in the list is set equal to the value before it (index "i") and "i" is then decremented by 1. Finally, the value at "i+1" of the list is set to the value of "key" which was defined at the beginning of this iteration of the outer "j" loop. This then repeats for the next index of "j".

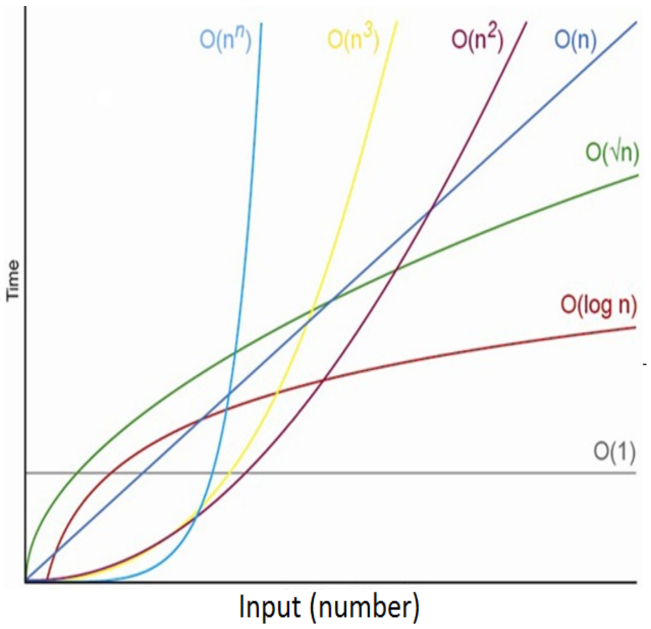
3. **Testing Plan**

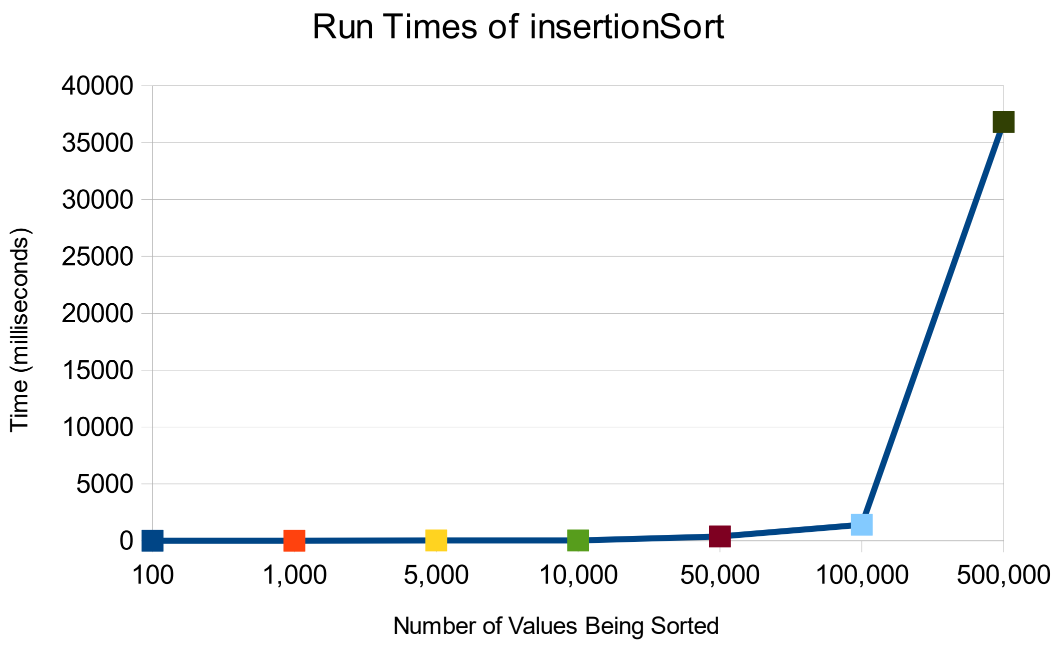
Testing this code is fairly straightforward. Upon running the program, each input file is read in and stored into an array which is then sorted. Each time this happens, the output displays what the number of values being sorted is, followed by the associated elapsed time for sorting through each size of arrays, which include sizes of 100, 1000, 5000, 10000, 50000, 100000, and finally 500000 for each "input_#.txt" file. No other interaction with the program is required for retrieval of all referenced data in this report.

4. **Test Cases**

Here is the direct printout from a sample iteration through the program, which provides us with all sizes of input arrays and their associated run time in milliseconds.

Run Times of insertionSort	
Input Size (number of values)	**Time (milliseconds)**
100	0
1,000	4
5,000	33
10,000	24
50,000	371
100,000	1,412
500,000	36,819





5. **Analysis and Conclusions**

We already know that insertionSort typically executes in n^2 time. Thus, by plotting the physical run times in ascending order of input sizes, one can compare with the general graph of quadratic time expansion also provided and see that the two graph lines exhibit very similar behavior. With these visuals we can see and visualize how insertionSort does in fact exhibit n^2 time complexity.

6. **References**

All of the code drafted in this assignment was created by myself, some of which was taken from assignment 1, along with a reference to the textbook for the insertionSort algorithm itself. The graph of time complexity run times was taken from this webpage :
https://www.codeproject.com/Articles/1012294/Algorithm-Time-Complexity-What-Is-It