Insertion Sort Analysis in Java

Preston Stosur-Bassett

Abstract

I. Motivation

In order to show how an algorithm might run on a given set of hardware, and how the algorithm will perform when given large amounts of data, algorithms are analysed. More specifically, insertion sort is analysed to determined what the run time might be for sorting large amounts of data on any modern computer.

II. Background

A sorting algorithm is used to sort data with a natural order. One such sorting algorithm is insertion sort, which sorts by iterating through a list of data, taking the current position and repositioning it into a more appropriate place in the list. How will this procedure perform when handling high volumes of data? How will it perform when executed on different machines? Because insertion sort is a more basic sorting algorithm, numerous papers and articles have been written answering these two questions.

III. Procedure

An insertion sort can be implemented in a multitude of languages using the pseudocode provided below.
Insertion Sort Pre-Condition: A is a non-empty array of data with a natural order.
Insertion Sort Post-Condition: A' is a permutation of A (containing all the same elements) in strictly non-decreasing order.

Algorithm 1 Insertion-Sort(A)

```
1:  procedure INSERTION-SORT(A)
2:      i = 2
3:      while i upto A.length do
4:          key = A[i]
5:          j = i − 1
6:          while j downto 1 and key < A[j] do
7:              A[j + 1] = A[j]
8:              j = j − 1
9:          end while
10:         A[j + 1] = key
11:         i = i + 1
12:     end while
13: end procedure
```

Outer-Loop Invariant: The subarray A'[1 ... i - 1] contains all the same elements as the subarray A[1 .. i - 1].
Outer-Loop Initialization: The outer-loop invariant holds because A'[1 ... i - 1] and A[1 ... i - 1]

both contain the same one element.
Outer-Loop Maintenance: The outer-loop invariant holds because A'[1 ... i - 1] and A[1 ... i - 1] both contain the same elements, although they maybe in different orders.
Outer-Loop Termination: When the outer-loop terminates, i = A.length, which implies that the entire array has been traversed and the guard has been negated. The negation of the guard implies that A'[1 ... i - 1] contains all the elements in A[1 ... i - 1].

Inner-Loop Invariant: A'[1 ... j] is sorted in strictly non-decreasing order.
Inner-Loop Initialization: Before the first iteration of the loop, j = 1, meaning the subarray A'[1 ... j] contains exactly one element, which is already sorted.
Inner-Loop Maintenance: At the beginning of each iteration of the loop the inner-loop invariants holds because j counts down from i and A'[j+1] is swapped with A'[j] only if A'[j+1] is less than A[j].
Inner-Loop Termination: The negation of the implies that j = A.length and A'[1 ... j] has been entirely traversed and sorted in strictly non-decreasing order which maintains the inner-loop invariant.

Conclusion: The termination of both the inner and outer loops implies that the entire array has been traversed, A' is a permutation of A containing all the same elements in strictly non-decreasing order. This satisfies the post condition.

IV. Testing

A. Testing Plan and Results

All arrays used in testing are Java ArrayList<Integer> unless otherwise specified. All times are recorded in milliseconds using a stopwatch class borrowed from [NEED CITATION]. It is important to note that the stopwatch class used takes the elapsed real-time between the start of the insertion sort algorithm and the end of the insertion sort algorithm as opposed to taking the elapsed processor-time because these tests were run on a multi-core computer. In the table below A denotes Array. Times in the table below are given as averages out of 10 trials.

Table I
Test Results

Tested Input	Expected Results	Actual Results	Time
Empty A	Empty A	Empty A	0.0003
A of 1000 Strings	Sorted A 1000 Strings	Sorted A 1000 Strings	0.021
A 1 Element	Original A	Original A	0.0003
A 10 Elements	Sorted A 10 Elements	Sorted A 10 Elements	0.0005
A 100 Elements	Sorted A 100 Elements	Sorted A 100 Elements	0.0021
A 1000 Elements	Sorted A 1000 Elements	Sorted A 1000 Elements	0.019
A 10000 Elements	Sorted A 10000 Elements	Sorted A 10000 Elements	0.129
A 100000 Elements	Sorted A 100000 Elements	Sorted A 100000 Elements	6.4923
A 1000000 Elements	Sorted A 1000000 Elements	Sorted A 1000000 Elements	2135.5007
A 10000000 Elements	Sorted A 10000000 Elements	OS Crash	N/A
A 1000 Identical Elements	Original Array	Original Array	0.0052

B. Problems Encountered

One major issue encountered during the development of this insertion sort was that after completing the sort, A' was completely sorted properly except for the first element in the array. No matter what value the first element of A had, it did not change position in A'. For example, if A[5, 6, 3, 4, 7] was

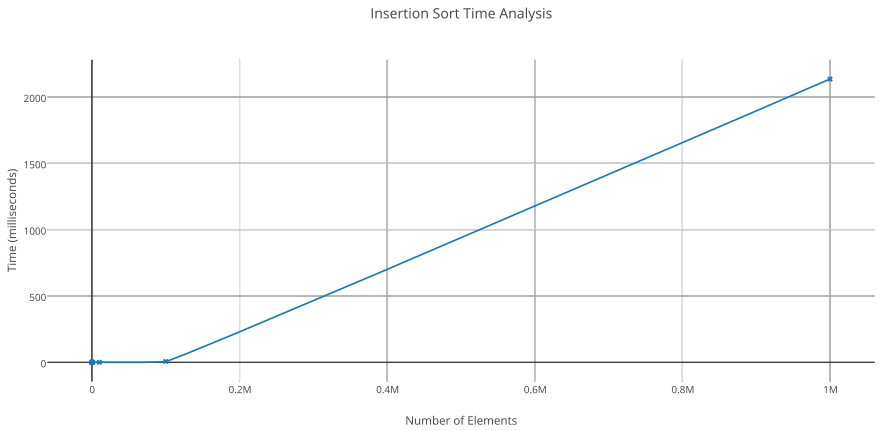

Figure 1. Insertion Sort Time Analysis

passed to the insertion sort algorithm, the returned array would look like A'[5, 3, 4, 6, 7]. Changing the guard for the inner for loop (see Algorithm 1 line 6) from *key < A[j] and j downto 1* to *j downto 1 and key < A[j]* corrected this issue.

V. Experimental Analysis

The insertion sort demonstrated below (see Algorithm 1) was implemented in Java and executed on an HP SprectreXT TouchSmart with 4 Core Intel i7 processor clocked at 1.9GHz running Ubuntu Gnome 14.10 64-bit.

VI. Conclusions

References

Appendix