Java Sorting Algorithms Comparison

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. Sorting algorithms sort data into a natural order. By analysing sorting algorithms, the fastest algorithm for a given problem can be determined.

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. Heap sort is another sorting algorithm that greatly differs than insertion sort in that it uses a divide, conquer, and combine method; meaning that it breaks the set it is sorting into subsets until the subsets can no longer be broken up and then merge sort combines the subsets together rendering the correct answer.

III. Procedure

An insertion sort can be implemented in a multitude of languages using the pseudocode provided in Algorithm 1.

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:     if A.length < 2 then
 3:         return A
 4:     end if
 5:     i = 2
 6:     while i upto A.length do
 7:         key = A[i]
 8:         j = i − 1
 9:         while j downto 1 and key < A[j] do
10:             A[j + 1] = A[j]
11:             j = j − 1
12:         end while
13:         A[j + 1] = key
14:         i = i + 1
15:     end while
16:     return A
17: end procedure
```

Insertion Sort Outer-Loop Invariant: The subarray A'[1 ... i - 1] contains all the same elements as the subarray A[1 .. i - 1].
Insertion Sort Outer-Loop Initialization: The outer-loop invariant holds because A'[1 ... i - 1] and A[1 ... i - 1] both contain the same one element.
Insertion Sort 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 may be in different orders.
Insertion Sort 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].

Insertion Sort Inner-Loop Invariant: A'[1 ... j] is sorted in strictly non-decreasing order.
Insertion Sort 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.
Insertion Sort Inner-Loop Maintenance: At the beginning of each iteration of the loop the inner-loop invariant 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].
Insertion Sort Inner-Loop Termination: The negation of the guard implies that j = A.length and that A'[1 ... j] has been entirely traversed and sorted in strictly non-decreasing order, which maintains the inner-loop invariant.

Insertion Sort 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.

A merge sort can be implemented in a variety of languages using the pseudocode provided in Algorithm 2.
Merge Sort Pre-Condition: A is a non-empty array of comparable data with a natural order.
Merge Sort Post-Condition: A' is a permutation of A (containing all the same elements) in stricly non-decreasing order.

IV. Testing

A. Testing Plan and Results
B. Problems Encountered

V. Experimental Analysis
VI. Conclusions

Algorithm 2 MERGE-SORT(A)

1: **procedure** MERGE-SORT(A)
2:     **if** $A.length < 2$ **then**
3:         **return** A
4:     **end if**
5:     $mid = A.length/2$
6:     **for** $i = 1$ upto mid **do**
7:         $left[left.length] = A[i]$
8:     **end for**
9:     **for** $i = mid$ upto $A.length$ **do**
10:         $right[right.length] = A[i]$
11:     **end for**
12:     $left = $ Merge-Sort($left$)
13:     $right = $ Merge-Sort($right$)
14:     $A = $ Merge($left$, $right$)
15:     **return** A
16: **end procedure**

17: **procedure** MERGE(left, right)
18:     $var\ i = 0$
19:     $var\ y = 0$
20:     $var\ x = 0$
21:     **while** $left.length\ != i$ and $right.length\ != y$ **do**
22:         **if** $left[i] < right[y]$ **then**
23:             $combined[x] = left[i]$
24:             $i = i + 1$
25:             $x = x + 1$
26:         **end if**
27:         **if** $right[y] < left[i]$ **then**
28:             $combned[x] = right[y]$
29:             $y = y + 1$
30:             $x = x + 1$
31:         **end if**
32:     **end while**
33:     **while** $left.length\ != i$ **do**
34:         $combined[x] = left[i]$
35:         $i = i + 1$
36:         $x = x + 1$
37:     **end while**
38:     **while** $right.length\ != y$ **do**
39:         $combined[x] = right[y]$
40:         $y = y + 1$
41:         $x = x + 1$
42:     **end while**
43:     **return** $combined$
44: **end procedure**

REFERENCES

APPENDIX