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 a comparable data with a natural order.
Merge Sort Post-Condition: A' is a permutation of A (containing all the same elements) in strictly non-decreasing order.
Merge Pre-Condition: Left and right are both non-empty arrays of a comparable data type with a natural order in strictly non-decreasing order.
Merge Post-Condition: Combined has all the elements of both left and right in strictly non-decreasing order.
A heap sort can be implemented in a variety of languages using the pseudocode below in Algorithm 3.

IV. Testing

A. Testing Plan and Results

All arrays used in testing are Java ArrayList<Integer> unless otherwise spcified. All times are recorded in milliseconds using a stopwatch class borrowed from Robert Sedgewock and Kevin Wayne It is important to note that the soptwatch class used takes the elapsed real-time between the start of the sort algorithm and the end of the sort algorithim as opposed to taking the elapsed processor-time becasue 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
Insertion Sort Test Results

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
```

Algorithm 3 HEAP-SORT(A)

```
1: procedure HEAPIFY(A, i, total)
2:     var left = i * 2
3:     var right = left + 1
4:     var iPrime = i
5:     if left <= total and A[left] > A[i] then
6:         i = left
7:     end if
8:     if right <= total and A[right] > A[i] then
9:         i = right
10:    end if
11:    if i! = iPrime then
12:        var temp = A[iPrime]
13:        A[iPrime] = A[i]
14:        A[i] = tmp
15:        A = heapify(A, i, total)
16:    end if
17:    return A
18: end procedure
19: procedure HEAP-SORT(A)
20:    var size = A.length
21:    for var i = size/2 downto 1 do
22:        A = heapify(A, i, size
23:    end for
24:    for var i = size downto 1 do
25:        var tmp = A[1]
26:        A[0] = A[i]
27:        A[i] = tmp
28:        size = size − 1
29:        A = heapify(A, 1, size)
30:    end for
31:    return A
32: end procedure
```

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

Table II

Merge Sort Test Results

Tested Input	Expected Results	Actual Results	Time
Empty A	Empty A	Empty A	0.0002
A of 1000 Strings	Sorted A 1000 Strings	Sorted A 1000 Strings	0.0297
A 1 Element	Original A	Original A	0.0001
A 10 Elements	Sorted A 10 Elements	Sorted A 10 Elements	0.0004
A 100 Elements	Sorted A 100 Elements	Sorted A 100 Elements	0.0028
A 1000 Elements	Sorted A 1000 Elements	Sorted A 1000 Elements	0.0294
A 10000 Elements	Sorted A 10000 Elements	Sorted A 10000 Elements	0.2098
A 100000 Elements	Sorted A 100000 Elements	Sorted A 100000 Elements	2.1904
A 1000000 Elements	Sorted A 1000000 Elements	Sorted A 1000000 Elements	22.9314
A 10000000 Elements	Sorted A 10000000 Elements	Sorted A 10000000 Elements	241.0322
A 1000 Identical Elements	Original A	Original A	0.0298

Table III
Heap Sort Test Results

Tested Input	Expected Results	Actual Results	Time
Empty A	Empty A	Empty A	0.0002
A of 1000 Strings	Sorted A 1000 Strings	Sorted A 1000 Strings	.0188
A 1 Element	Original A	Original A	0.0003
A 10 Elements	Sorted A 10 Elements	Sorted A 10 Elements	0.0006
A 100 Elements	Sorted A 100 Elements	Sorted A 100 Elements	0.002
A 1000 Elements	Sorted A 1000 Elements	Sorted A 1000 Elements	0.0162
A 10000 Elements	Sorted A 10000 Elements	Sorted A 10000 Elements	0.0468
A 100000 Elements	Sorted A 100000 Elements	Sorted A 100000 Elements	0.889
A 1000000 Elements	Sorted A 1000000 Elements	Sorted A 1000000 Elements	23.0635
A 10000000 Elements	Sorted A 10000000 Elements	Sorted A 10000000 Elements	244.1952
A 1000 Identical Elements	Original A	Original A	0.0156

B. Problems Encountered

V. Experimental Analysis

VI. Conclusions

References

Appendix