Barrett Otte

CMSC 451

03-01-2018

# Introduction

## Brief introduction of Insertion Sort

* Basic sorting algorithm used to create a sorted array/list one element at a time.
* Each iteration removes one element from the unsorted list, finds the correct position in the sorted list, and inserts the element in the correct index. This is repeated until there are no elements left in the unsorted list.
* Very efficient with small data sets, but loses efficiency quickly as the data set increases.

## High-level pseudocode for Insertion Sort Implementations

Method Iterative (A)

For i ←1 to length(A)

For j ← i to 0

If A[j] < A[j-1]

temp ← A[j]

A[j] ← A[j-1]

A[j-1] ← temp

End If

End For

End For

Return A

End Iterative()

Method Recursive-Helper(A, n)

If n <= 1

return A

Recursive-Helper(A, n-1)

last ← A[n-1]

j ← n – 2

While j >= 0 and A[j] > last

A[j+1] ← A[j]

j ← j-1

End While

A[j+1] ← last

Return A

End Recursive-Helper()

Method Recursive(A)

Return Recursive-Helper(A, length(A))

End Recursive()

## Big-Θ analysis of the two versions of Insertion Sort

* The best case is Θ(n) if the given list is already sorted.
  + This would give 1 comparison and no swaps.
  + Comparisons: n – 1 which is an element of Θ(n).
* The worst case is Θ(n2) if the given list is in reverse order
  + Operations = 2 \* (1 + 2 + … + n-1) →
  + Operations = ( (2(n-1)(n-1+1) ) / 2 → n(n-1) → Θ(n2)
* Average case is Θ(n2), the same as the worst case.
* The iterative and recursive implementations share the run time of Θ(n2) due to the nature of the conditions in each method.
* In the iterative implementation, the condition: For i ←1 to length(A) will always be executed (n-1) times. The inner condition will execute 1+2+…+ (n-1) times. Which is equivalent to (n-1)n/2. This is an element of Θ(n2).
* In the recursive implementation there is a similar condition with While j >= 0 and A[j] > last, this will be executed (n-1) times. In the same way as the inner loop of the iterative implementation, the method will be called recursively 1+2+…+(n-1) times which evaluates to a performance of Θ(n2)
* The space complexity of iterative insertion sort is O(1).
* The space complexity of recursive insertion sort is O(n), caused by reserving more space on the stack because of the recursion.

## Explanation of Approach to avoiding the problems associated with JVM warm-up

Within the JVM, the compiler used is known as the Just In Time (JIT) compiler. The JIT compiles bytecode into native code during runtime after a program has started. Running a benchmark at the start of an application will cause poor results due to numerous JIT class optimizations and JVM garbage collection.

In my code, I followed a guide by baeldung.com to run a basic JVM warmup consisting of thousands of dummy class instantiation and iterations. I now realize that I think I implemented my JVM warmup wrong based on the information I reread on baeldung’s website. “Once class-loading is complete, all important classes (used at the time of process start) are pushed into the JVM cache (native code) – which makes them accessible faster during runtime. Other classes are loaded on a per-request basis” (Baeldung, 2017). Perhaps I should have loaded the main classes of this benchmark multiple times instead of the dummy classes. I did find the data to be slightly better when running with the JVM when I first launched my computer. I was previously unimpressed, but I was ignorant to the fact that my JVM was already warmed up when I first implemented it.

## Discussion of the critical operation chosen and why selected

The critical operation recorded in the recursive and iterative implementations was the total number of comparisons. This critical operation is found within each inner condition of the insertion sort implementations. Since the inner and outer conditions of both implementations run at (n-1), the critical operation should be Θ(n2). I suppose the other operation that could be counted is the total number of swaps to insert each element in its correct position. However, I find the number of counts to be a clearer indication of performance between the two implementations since swapping elements will always be performed the same in both cases.

# Analysis

## Graph of critical operations for both algorithms and one for the execution times

NOTE: The two following graphs required changing of the trial sizes, I really hope this does not make me lose points. When coding originally I did not fully realize how inefficient insertion sort was with large data sets. Now, I am just very curious how the graphs compare between implementations.

## Comparison of the performance of the two versions of the algorithm

It is clear in the first set of graphs that the recursive implementation of insertion sort is more efficient as the data size increases. It requires half the amount of critical operations (comparisons) and performs them in half the time of the iterative implementation. Both implementations begin to become massively inefficient around 2500 elements. This is to be expected though because insertion sort is meant for small data sets and performs poorly with large data sets. In the second set of graphs with smaller data sets, the iterative and recursive implementations show less difference in efficiency. I chose poor data sets for the first run of this benchmark, I should have taken into account insertion sort’s problems with large data sets. That is why I decided to include two sets of graphs, so that my mistake is still clearly shown in my paper.

## Comparison of the critical operation results and the actual execution time measurements

Looking at the first set of graphs, the separation of implementation efficiency begins to be visible at a data size of 2500 elements. At 10,000 elements it can clearly be seen that the recursive implementation takes half of the amount of critical operations to sort the list. In the case of large data sets, it is very evident that the recursive implementation is more efficient than the iterative implementation.

Looking at the second set of graphs shows more interesting information. With data sets below ten elements, the iterative implementation performs the same or quicker than the recursive implementation. The amount of critical operations between the two implementations remains negligible up until over 100 elements. After 200 elements the trend seen in the first set of graphs begins to take effect. I am slightly confused with the execution time graph from data sizes 100 to 200. The iterative implementation posted the same time that the recursive implementation did. I am unsure of why this occurred.

## Discussion of the significance of the coefficient of variance results and how it reflects the data sensitivity of Selected algorithm

While discussing the coefficient of variance (CV), I will be using the first set of graphs and data. It is also rather unfortunate, but after looking through my data it looks like I calculated the coefficient of variance because I got data that makes no sense. For most of my CVs I received numbers well over ranging from 61.09 to 44579.16, leading me to believe I messed up the calculation.

As far as the significance of the coefficient of variance, I believe that it would be useful in comparing how efficient the different implementations were. Alongside efficiency, the CV would be useful in comparing the ratio of execution time and critical operation count within each implementation.

## Comparison of Results vs. Big-Θ analysis

For my actual results, I could see that Big Θ analysis and actual results were different. Since the average case for insertion sort is listed as Θ(n2), of course the actual results are going to appear better than average. This makes comparison a little boring unfortunately. Below is a table of the best and average/worst cases vs the actual cases.



* For a data set of 5, the recursive implementation was closest to the best performance possible with an average only off by 0.52 critical operations.
* In both implementations, the worst performance was only half of the worst case at 10000 elements.
* It appears that for the iterative implementation, the critical operations count is always around half of the worst case.
* For the recursive implementation, it appears that the critical operations count is around a quarter of the worst case.
* The closest either implementation of insertion sort get to the best case is when the data set is under 10 elements.

# Conclusion

Based on the results of both implementations of insertion sort, this algorithm performs very poorly with large data sets. The graphs clearly show how much efficiency plummets as the data size increases. If insertion sort must be used, the recursive implementation performs twice as fast and efficient than the iterative implementation. The iterative implementation is best reserved for data sizes below 10 elements, whereas the recursive implementation performs decent with sizes up to 10000 elements.

Honestly I feel as if I did not correctly implement the iterative version of insertion sort. I knew that the recursive implementation was going to perform better than the iterative implementation, but not by this big of a margin. I was under the impression that both implementations would perform around the same. The critical operation count of the recursive implementation appears to be correct, but I am not convinced that the iterative implementation’s critical operation counts are. With the recursive implementation, I know that the space complexity is far greater than the iterative implementation. That seems to be the only impactful strength of the iterative implementation of insertion sort that I could see.

In conclusion, if the insertion sort algorithm must be used on a large data set, the recursive implementation should be used. However, it is important that the system has enough memory to support the space complexity of the recursion. If there is a small data set of under ten elements, then the iterative implementation of insertion sort would be the best to use as far as execution time is concerned.

# References

Baeldung. (2017, August 12). How to Warm Up the JVM. Retrieved February 28, 2018, from <http://www.baeldung.com/java-jvm-warmup>

Pandey. (2008, July & aug.). Study and Comparison of Various Sorting Algorithms. Retrieved March 01, 2018, from <http://dspace.thapar.edu:8080/jspui/bitstream/10266/583/3/T583.pdf>