# CMSC 451
# Final Project
# Mark Tasker
# 05/10/2020

# 1) Introduction

## 1.1) Description and pseudo code

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, insertion sort provides several advantages:

- Simple implementation
- Efficient for (quite) small data sets, much like other quadratic sorting algorithms
- More efficient in practice than most other simple quadratic (i.e., O($n^2$)) algorithms such as selective sort or bubble sort
- Adaptive, i.e., efficient for data sets that are already substantially sorted-the time complexity is O($kn$) when each element in the input is no more than $k$ places away from its sorted position
- Stable: i.e., does not change the relative order of elements with equal keys
- In-place: i.e., only requires a constant amount O(1) of additional memory space
- Online: i.e., can sort a list as it receives it

Insertion sort iterates, consuming one input element each repetition and growing a sorted output list. At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain. Sorting is typically done in-place--by iterating up the array and growing the sorted list behind it. At each array position, it checks the value there against the largest value in the sorted list (which happens to be next to it, in the previous array-position checked). If larger, it leaves the element in place and moves to the next. If smaller,

it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts into that correct position.

**Pseudo code:**

```
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
```

## 1.2) Big - Θ analysis of algorithm:

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value. Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average case complexity of an algorithm.

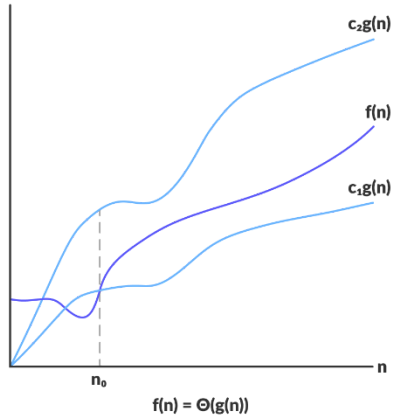


*Figure 1 Big - Θ Example*

**Pseudo code with time cost:**

| INSERTION-SORT($A$) | *cost* | *times* |
|---|---|---|
| 1 **for** $j = 2$ **to** $A.length$ | $c_1$ | $n$ |
| 2 $key = A[j]$ | $c_2$ | $n-1$ |
| 3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$. | 0 | $n-1$ |
| 4 $i = j-1$ | $c_4$ | $n-1$ |
| 5 **while** $i > 0$ and $A[i] > key$ | $c_5$ | $\sum_{j=2}^{n} t_j$ |
| 6 $A[i+1] = A[i]$ | $c_6$ | $\sum_{j=2}^{n}(t_j - 1)$ |
| 7 $i = i-1$ | $c_7$ | $\sum_{j=2}^{n}(t_j - 1)$ |
| 8 $A[i+1] = key$ | $c_8$ | $n-1$ |

Total cost for insertion sort would be:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1) .$$

Based on pseudo code and time costs, best, average and worst cases are:

**BEST CASE:**

The array is already sorted. $t_j$ will be 1 for each element while condition will be checked once and fail because A[i] is not greater than key. Hence cost for steps 1, 2, 4 and 8 will remain the same. Cost for step 5 will be n-1 and cost for step 6 and 7 will be 0 as $t_j$-1 = 1-1 = 0. So cost for best case is:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

We can express this running time as an+b where a and b are constants that depend on costs $c_i$. Hence, running time is a linear function of size n, that is, the number of elements in the array.

## WORST CASE:

The array is reverse sorted. $t_j$ will be j for each element as key will be compared with each element in the sorted array and hence, while condition will be checked j-1 times for comparing key with all elements in the sorted array plus one more time when i becomes 0 (after which i > 0 will fail, control goes to step 8).

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

and

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

The explanation for the first summation is simple - the sum of numbers from 1 to n is n(n+1)/2, since the summation starts from 2 and not 1, we subtract 1 from the result. We can simplify the second summation similarly by replacing n by n-1 in the first summation.

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

We can express this worst-case running time as $an^2+bn+c$ where a, b and c are constants that depend on costs $c_i$. Hence, running time is a quadratic function of size n, that is, the number of elements in the array.

## AVERAGE CASE:

The average case running time is the same as the worst-case (a quadratic function of n). On average, half the elements in A[1..j-1] are less than A[j], and half the elements are greater. On average, therefore, we check half of the subarray A[1..j-1], and so $t_j$ is about j/2. The resulting average-case running time turns out to be a quadratic function of the input size, just like the worst-case running time.

**SUM-UP:**

- **Best case:** $\Theta(n)$
- **Worst case:** $\Theta(n^2)$
- **Average case:** $\Theta(n^2)$

## 1.3) Avoiding JVM warm-up problems

When a JVM based app is launched, the first requests it receives are generally significantly slower than the average response time. This warm-up effect is usually due to class loading and bytecode interpretation at startup. After a certain number of iterations, the main code path is compiled and "hot". We use caches to avoid unnecessary calls requesting for the same data. But while our application starts and until these caches are all filled up, we experience high latencies. To make the most out of our application we need to exercise latency sensitive code-paths and fill its caches before it enters the pool of production instances.

Before running benchmark and measuring execution times the sorting algorithm is first run without any measurements to 'warm-up' and only after that we start benchmark and run sorting algorithm once again. Using this approach, we avoid problems associated with JVM warm-up.

## 1.4) Counting critical operations

Critical operations performed as a part of the algorithm are operations performed on data, in this case, an array of numbers. During execution, we counted assignments of values and comparisons between array elements. Operations on indexes are not critical since the execution time is much smaller than regular operations on array elements (usually operations on indexes are also hardware supported).

# 2) Analysis

## 2.1) Graph of critical operations and of execution times

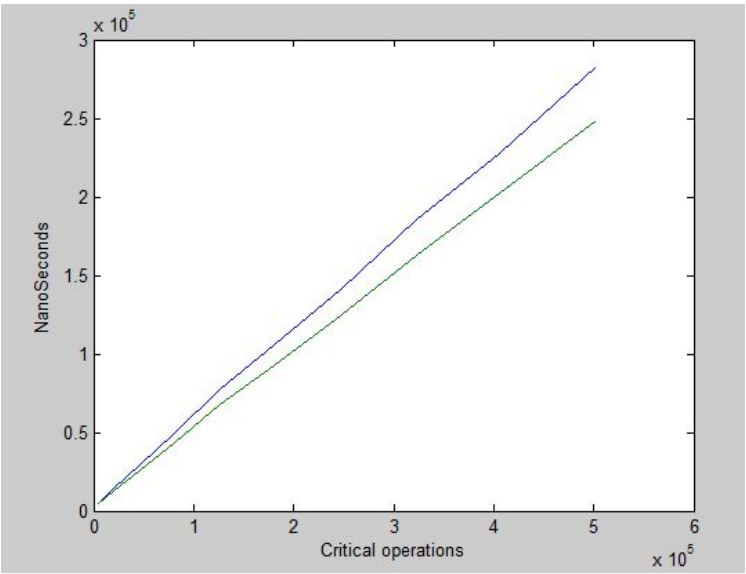


*Figure 2 BLUE – Recursive; GREEN – Iterative*

This graph plots average number of critical operations and average execution time. The blue line depicts results of the recursive method and the green line depicts results of the iterative method. Number of critical operations is the same for both methods but execution time differs. It is easy to notice how execution time of recursive method grows faster than execution time of iterative method.

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

In files iterative_report and recursive_report we can see results from several measurements and average execution time. By comparing execution times for the same number of elements we can see that even though in certain cases the recursive method beats the iterative one-on average the iterative method proves to be the more efficient one.

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

By comparing the number of critical operations we can conclude that for the same data the number of critical operations is the same both for iterative and recursive methods. On the other hand, when it comes to

execution time it is noticeable that the execution time of the recursive method is bigger and it also grows faster than iterative method execution time with rise of number of elements.

## 2.4) Significance of the coefficient of variation results and how it reflects the data sensitivity of algorithm

Although there is a difference between execution times of the same algorithm and that recursive method sometimes finishes faster, on average the data we get from measurements is correct and the algorithm behaves as it should.

## 2.5) Comparison of results to Big - Θ analysis

Comparing results from recursive_report and iterative_report we can notice that execution time really behaves as predicted – execution time is a function of a square number of elements. In the next picture the first column depicts number of element, and every other column contains execution time (in nano seconds) for given number of elements.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 100 | 4765 | 5867 | 5497 | 5378 | 5221 | 5378 | 5539 | 5378 | 4959 | 4889 | 5061 |
| 200 | 19927 | 17600 | 20395 | 14666 | 20223 | 14667 | 19541 | 14178 | 20199 | 14666 | 20095 |
| 300 | 43085 | 28844 | 45795 | 29822 | 47051 | 29822 | 44477 | 28356 | 45571 | 28845 | 45753 |
| 400 | 82033 | 50355 | 81159 | 49378 | 84131 | 51333 | 81905 | 49867 | 76473 | 47422 | 75293 |
| 500 | 129725 | 77245 | 129825 | 76755 | 123557 | 72844 | 128423 | 75289 | 126843 | 97289 | 123875 |
| 600 | 184475 | 108045 | 179723 | 114400 | 191471 | 110489 | 179603 | 104133 | 189125 | 109511 | 179385 |
| 700 | 236635 | 136400 | 252351 | 146667 | 244903 | 140311 | 243269 | 142756 | 245317 | 140311 | 235215 |
| 800 | 325653 | 184800 | 328581 | 185777 | 317323 | 179911 | 328457 | 185289 | 335371 | 189200 | 307735 |
| 900 | 397941 | 224400 | 406169 | 228311 | 422385 | 239556 | 424829 | 237600 | 405405 | 227823 | 403263 |
| 1000 | 497515 | 279156 | 514693 | 286489 | 505845 | 282089 | 501873 | 279645 | 515591 | 287466 | 500621 |

For example, in the second column for 100 elements execution time is 4.7K while for 1000 elements execution time is 497K. The number of elements is 10 times higher while execution time is 100 times higher ($f\{n^2\}$). Summing up all times to get average execution time proves that the execution time follows Big – Θ notation results.

# 3) Conclusion

Although the recursive functions are easier to write and understand, they do not perform as well as iteration implementation of the same functions. The main reason for the recursive algorithm performing slower than iterative is function calling overhead. Recursive function calls itself in each iteration which includes throwing variables on stack which adds-up execution time. Whether you use recursive or iterative functions depends upon the needs of your program. The stacking of recursive functions utilizes more memory, but if that is not an issue than I would personally prefer to go that way. As with all projects, your choice is dependent upon the needs of that particular application.