The insertion-sort algorithm: Time problem

An array *A* of *n* elements can be sorted in ascending order by the so-called insertion-sort algorithm. This algorithm works by scanning through the array and performing two types of operations *O*:

O_1: It compares a yet unsorted element with a sorted element *(outer while loop).*

O_2: It exchanges elements if they are not in the right order *(inner while loop).*

The time to run the code is affected by the times both operations are performed.

Best-case scenario (Array is pre-sorted in ascending order)

In this case, the algorithm has to perform only operations of the first kind. However, as both operations account to the run time $O_{Total} = n$. The last iteration of the operation ends the outer while loop as *j = len(input_list),* hence O_{Total} *(n)* and not *(n-1)* (j being the counter for the outer while loop).

Worst-case scenario (Array is pre-sorted in descending order)

In the worst-case scenario, where the list is sorted the other way around in decreasing order, type-2 operations must be performed in addition to type-1 operations.

To place only the last (lowest) unsorted number at the correct place (at the beginning), the algorithm has to perform *n-1* times the first operation and *n-1* times the second operation, leading to $O_{Total} = 2*(n-1)$. The process, operation 1 and 2, must be repeated for n-1 times. The total amount of operations to bring alle elements of the array in order is therefore $O_{Total} = 2*(1 + \dots + n-1) \Rightarrow O_{Total} = n*(n-1)$.

Example: Looking at the array A1 of length 6, the best case needs 6 operations in total: $O_{Best}(A1) = 6$ and the worst case however $O_{Worst}\ (A1) = 6*(5) = 30$. The array *A2 = 12* will already need *O(A2) = 12*11 = 132* operations to finish the worst-case scenario. Accordingly, $O_{Worst}\ (24) = 552$ and $O_{Worst}\ (48) = 2256$. This growth is nearly quadratic, see Figure 1.

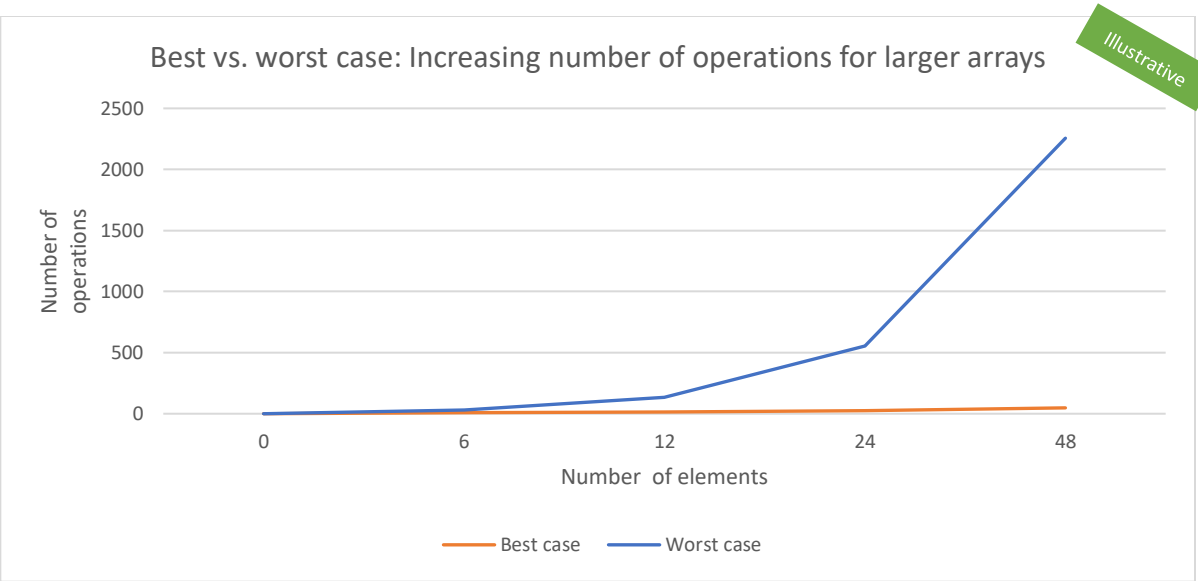

Figure 1: Example figure

Conclusion

If the arrays are perfectly in order, the algorithm would be very efficient, as the number of needed operations would increase linear. However, the average case where the array holds elements in randomized order is closer to the worst case. The number of operations needed equals $O_{Total} = n*(n-1)$, which approximates to $O = n^2$ with an increasing number of elements, and therefore grows quadratically. The insertion-sort algorithm is hence rather inefficient for large unsorted arrays.