# Insertion Sort: Analysis and Correctness

Insertion sort is a comparison-based sorting algorithm that we will use as an example to understand some aspects of algorithmic analysis and to demonstrate how an iterative algorithm can be shown to be correct.

The principle behind insertion sort is to remove an element from an unsorted input list and insert in the correct position in an already-sorted, but partial list that contains elements from the input list. It can be implemented using an additional list or with the same list. We will use the latter scenario in our example. The pseudocode for Insertion Sort is as follows, where $A$ is a list with indices from 1 to $N$.

**Algorithm 1** InsertionSort(A)

```
1: for j ← 2 to N do
2:     key ← A[j]
3:     i ← j − 1
4:     while i > 0 AND A[i] > key do
5:         A[i + 1] ← A[i]
6:         i ← i − 1
7:     end while
8:     A[i + 1] ← key
9: end for
```

## Runtime complexity

We will first analyze `Insertion Sort` and determine its runtime complexity. We will use $t_i$ to represent the time needed to execute statement $i$ in the pseudocode above. Each statement can be executed in constant time because all operations involve a fixed number of bits. The running time of the algorithm is, therefore, determined by the number of times each statement is executed.

Statement 1 is the header for the `for` loop. This statement is executed $N$ times (once more than the number of loop iterations). Statements 2 and

3 are each executed $N-1$ times. Statement 4 is the header for the `while` loop. The number of times this statement is executed is variable; the number of executions depends on the contents of the array $A$ and value of $j$. Let $x$ be the number of times statement 4 is executed. Statements 5 and 6 will be executed as long as the condition for the `while` statement remains true, which means they will be executed $x-1$ times. (Statements 7 and 9 are for convenience and do not involve any execution.) Statement 8 is executed as many times as statements 2 and 3 because they are part of the same `for` loop body, which is $N-1$ times.

This type of analysis is based on the Random Access Machine model of a computer. In this model:

- **Each simple operation takes constant time.** What are simple operations? Arithmetic operations on fixed-size data elements, data movement operations (load, store, copy) and control operations (conditional and unconditional branches, subroutine calls and returns).
- Loops and subroutines **do not necessarily take constant time**.
- **All memory accesses take a constant amount of time.**

The running time of `Insertion Sort` can be described by a function $t(N)$ where

$$t(N) = N \cdot t_1 + (N-1) \cdot (t_2 + t_3) + x \cdot t_4 + (x-1) \cdot (t_5 + t_6) + (N-1) t_8.$$

We do not know $x$ exactly but we can derive an upper bound for $x$. For a fixed $j$, i.e., for one iteration of the outer loop, let $x_j$ be the maximum number of times Statement 4 can be executed. $i$ starts at $j-1$ therefore statement 4 is executed no more than $j-1+1=j$ times because $i$ is decremented by 1 within the `while` loop. Thus $x_j = j$ and an upper bound on $x$ is $\sum_{j=2}^{N} x_j = \sum_{j=2}^{N} j = (\sum_{j=1}^{N} j) - 1 = N(N+1)/2 - 1 = (N^2 + N - 2)/2$.

Using the upper bound on $x$ that we have established (above), we can express $t(N)$ in the worst case as

$$t(N) = k_1 n^2 + k_2 n + k_3,$$

where $k_1, k_2, k_3$ are some constants that we can obtain by algebraic manipulation.

Now $(k_1 + k_2 + k_3)N^2 \geq t(N)$ for $N \geq 1$ thus establishing that, in the worst case, $t(N) \in O(N^2)$. Also, using $k = \min\{k_1, k_2, k_3\}$, $kN^2 \leq t(N)$ for $N \geq 1$. Thus $t(N) \in \Omega(N^2)$.

$t(N) \in \Omega(N^2)$ and $t(N) \in O(N^2)$ in the worst case therefore $t(N) \in \Theta(N^2)$ in the worst case.

We have established the asymptotic complexity of `Insertion Sort` in the worst case. We will now prove the correctness of the algorithm.

## Proof of Correctness

We will establish the correctness of `Insertion Sort` using loop invariants. A loop invariant is a statement that is true across multiple iterations of a loop. In the remainder of this discussion, we will use $A[i..j]$ to denote the sublist of $A$ starting from the $i^{th}$ element and extending to the $j^{th}$ element (both end points inclusive).

**Theorem 1** `Insertion Sort` *(Algorithm 1) correctly sorts input list A.*

*Proof.*

The first invariant, `Inv1`, that we will use is that at the start of each `for` loop iteration (Statement 4) $A[1..j-1]$ is a sorted permutation of the original $A[1..j-1]$.

`Inv1` holds at the start because $A[1..1]$ is sorted obviously. To prove that `Inv1` holds for each iteration, we must reason about the execution within the loop. In particular we must show that after Statement 8 is executed $A[1..j]$ is a sorted permutation of the original $A[1..j]$. To this end, we will use another invariant concerning the `while` loop, `Inv2`, which states that at the start of the `while` loop body $A[i..j]$ are each greater than or equal to $key$.

`Inv2` is true upon initialization (at the first iteration of the inner loop) because $i = j-1$ and $A[i] > key$ by explicit testing and $A[j] = key$. The inner loop maintains the invariant because the statement $A[i+1] \leftarrow A[i]$ moves a value in $A$ that is known to be greater than $key$ into $A[i+1]$ which also held a value that was at least equal to $key$.

The inner loop (the `while` loop) does not destroy data in $A$ because the first iteration copies $A[j]$ into $key$. As long as $key$ is restored to $A$ we maintain the invariant that $A[1..j]$ contains the first $j$ elements of the original list. When the inner loop terminates, we know that

- $A[1..i]$ is sorted and is at most equal to $key$ (which is true by default if $i = 0$ and true because $A[1..i]$ is sorted and $A[i] \leq key$ if $i > 0$);
- $A[i+1..j]$ is sorted and at least equal to $key$ because the loop invariant held before $i$ was decremented and that invariant was $A[i..j] \geq key$;
- $A[i+1] = A[i+2]$ if the loop is executed at least once and $A[i+1] = key$ if the loop did not execute at all.

With these observations, we know that $A[i+1] \leftarrow key$ does not destroy any data and gives us $A[1..j]$, which is a sorted permutation of the original $j$ elements of $A$.

Because `Inv1` is maintained after a loop iteration, it is also maintained even when the outer loop terminates. When the outer loop terminates, $j = N+1$ and so $A[1..N]$ is sorted.

□