Chapter 2 - Insertion-Sort

Problem Statement

Input: A sequence of n numbers $<a_1,a_2,\ldots,a_n>$

Output: A permutation (reordering) $< a'_1,a'_2,\ldots,a'_n >$ of the input sequence such that $a'_1 \le a'_2 \le \ldots \le a'_n$

This is a good algorithm for sorting small number of elements.

For example:

- Start with an empty left hand.
- Add one card at a time in the correct position.
- Find the correct position for a card; compare it with each of the cards already in the hand from left to right.
- The cards are sorted at all times, the initial cards are not sorted as they are added one at a time.

Pseudocode:

- The input parameter is an array $A[1..n]$ with a length of n
- The ".." denotes a range within an array
- When using an array $A[1..n]$ allocate the array to be one entry longer, i.e., $A[0..n]$
- The array A is sorted in place.

Insertion-Sort(A)	cost	times
for $j \leftarrow 2$ to length[A]	c_1	n
do key $\leftarrow A[j]$	c_2	n-1
//Insert $A[j]$ into the sorted sequence //$A[1..j$-1]	0	n-1
$i \leftarrow j - 1$	c_4	n-1
while $i > 0$ and $A[i] >$ key	c_5	$\sum^{n}_{j=2} t_j$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum^{n}_{j=2} (t_j\text{-}1)$
$i \leftarrow i - 1$	c_7	$\sum^{n}_{j=2} (t_j\text{-}1)$
$A[i+1] \leftarrow$ key	c_8	n-1

Correctness: Often a loop invariant is used to help understand why an algorithm gives the correct answer.

Loop Invariant: At the start of each iteration of the outer loop, the subarray $A[1\ldots j\text{-}1]$ consists of the original elements from position $1,2,\ldots,j$-1, but in sorted order.

To prove loop invariance is correct three things must be shown:

1. Initialization: true before the 1st time. $j = 2$, everything to the left is just element 5 so 1 item is trivially sorted.
2. Maintenance: if true before a particular iteration then it is true after. Note, that inner loop does $A[j+1]$, $A[j+1]$, … until key is in proper position.
3. Termination: true when loop terminates, e.g., when loops $j > n \Rightarrow j = n + 1$ so $j - 1 = n$

The loop invariant for Insertion-Sort starts each iteration of the "outer" for loop indexed by j. The sub array $A[1\ldots j\text{-}1]$ consists of the elements originally in $A[1\ldots j\text{-}1]$ but in sorted order.

Analysis of Insertion Sort:

- Assume that the ith line takes c_i which is a constant. Note: line 3 is a comment so no time is taken.
- For $j = 2,3,\ldots,n$, let t_j be the number of times the **while** loop test is executed for j.
- Note: that when a **for** or **while** loop exists in the usual way the test is executed one time more than the loop body.

The running time of the algorithm is

$\sum_{\text{over all statements}}$(cost of statement)·(number of times statement is executed)

Let $T(n)$ = running time of Insertion-Sort

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

The run time is dependent on the values of t_j.

Best Case: The array is already sorted.

$T(n) = c_1 n + c_2(n\text{-}1) + c_4(n\text{-}1) + c_5(n\text{-}1) + c_8(n\text{-}1)$

$\quad = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$

Worst Case: The array is sorted in reverse order.

$T(n) = c_1 n + c_2(n\text{-}1) + c_4(n\text{-}1) + c_5(n(n\text{-}1)/2 \text{ -}1) + c_6(n(n\text{-}1)/2) + c_7(n(n\text{-}1)/2) + c_8(n\text{-}1)$

$\quad = (c_5/2 + c_6/2 + c_7/2)n^2 + (c_1 + c_2 + c_4 + c_5/2 - c_6/2 - c_7/2 + c_8)n - (c_2 + c_4 + c_5 + c_8)$

Average Case: Typically the worse case is calculated instead of the average case due to the following three reasons:

1. The worst case run time provides an upper bound on the run time for an input.
2. The worst case often occurs, e.g., when searching the worst case often occurs when an item being searched for is not present.
3. The average case is not analyzed because it is often as bad as the worst case.