Insertion Sort

- It is a Simple Sorting algorithm that works the way we Sort playing Cards in our hands.
- It is an efficient algorithm for Sorting a Small number of elements.
- Every repetition of insertion Sort removes an element from the input date, inserting it into the Correct position in the already Sorted list.
- It is inplace: it rearranges the numbers within the array A.

example

(a) 5 2 4 6 1 3 (b) 2 5 4 6 1 3

(c) 2 4 5 6 1 3 (d) 2 4 5 6 1 3

(e) 1 2 4 5 6 3 (f) 1 2 3 4 5 6

Algorithm Insertion-Sort(A)

```
for j = 2 to A.length
    Key = A[j]
    // insert A[j] into the Sorted Seq A[1 ... j-1]
    i = j-1
    while i>0 and A[i] > Key
        A[i+1] = A[i]
        i = i-1
    A[i+1] = Key
```

Loop invariant - It helps to understand why an algorithm is Correct

Initialization :- It is true prior to the first iteration of the loop.

<u>Maintenance</u> :- If it is true before an iteration of the loop, it remains true before the next iteration.

<u>Termination</u> :- When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Analysis

- The time taken by algorithm is depends on input size.
- Algo takes diff amount of time to sort input seq of same size, depending on how nearly sorted they already are.
- Running time of an algo is the no of primitive operations or steps executed.

		Cost	times
1.	for $j = 2$ to A.length	c_1	n
2.	key = A[j]	c_2	$n-1$
	// remarks	c_3	$n-1$
3.	$i = j-1$	c_4	$n-1$
4.	while $i>0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
5.	$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
6.	$i = i-1$	c_7	-do-
7.	$A[i+1] = key$	c_8	$n-1$

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

$$c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

[t_j denote the no of times the while loop test]

<u>Best Case</u>: if array is already sorted ($t_j = 1$)

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$

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

express as $an + b$ for Const 'a' and 'b' (linear function of n)

Worst Case

if array is in reverse sorted order [compare $A[j]$ element with each $A[1 \ldots j-1]$ elements]

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

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

$$= n^2\left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) + n\left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) - (c_2 + c_4 + c_5 + c_8)$$

express as $an^2 + bn + c$ for Constant a, b (Quadratic func of n)

Average Case

- we check half of the subarray $A[1 \ldots j-1]$ and so t_j is about $j/2$ $\quad T(n) = \sum_{j=2}^{n} \Theta(j/2) = \Theta(n^2)$ [All permutations equally likely]
- Resulting are average case running time out to be a quadratic function of input size like worst case.

Point to remember

1. Time Complexity = $O(n^2)$
2. Auxiliary Space = $O(1)$
3. Algo Paradigm = Incremental approach
4. Sorting in place = Yes
5. Stable = Yes
6. Uses :- 1) When no. of elements is small.
   2) Useful when input array is almost sorted, only few elements are misplaced in complete big array.

- A Stable Sort is one which preserve the original order of input set, where Comparision algo donot distinguish between two or more items.