Insertion Sort

Varun Kumar

July 4, 2025

1. Logic

Insertion Sort builds the sorted array one element at a time by repeatedly picking the next element and inserting it into the correct position among the already sorted elements.

Key Idea

At the i^{th} iteration, the first i elements are sorted, and the $(i+1)^{th}$ element is inserted into its correct position.

2. Number of Comparisons and Shifts

Let n be the number of elements.

Worst Case (Reverse Sorted)

- Comparisons: $\frac{n(n-1)}{2}$
- Shifts: $\frac{n(n-1)}{2}$

Best Case (Already Sorted)

- Comparisons: $n-1$
- Shifts: 0

3. Optimal Behavior

Insertion sort is adaptive by default. It reduces unnecessary comparisons and shifts when the array is already or partially sorted.

4. Pseudocode

```
function insertionSort(arr):
    n = length(arr)
    for i = 1 to n-1:
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j+1] = arr[j]
            j = j - 1
        arr[j+1] = key
```

Stability Note

Insertion Sort is stable because it inserts elements in a way that preserves the relative order of equal elements.

5. Example Walkthrough

Given: `[5, 1, 4, 2, 8]`

Pass 1

`[5, 1, 4, 2, 8]` ⇒ `[1, 5, 4, 2, 8]`

Pass 2

`[1, 5, 4, 2, 8]` ⇒ `[1, 4, 5, 2, 8]`

Pass 3

`[1, 4, 5, 2, 8]` ⇒ `[1, 2, 4, 5, 8]`

Pass 4

`[1, 2, 4, 5, 8]` → Already in position → Done

6. Time & Space Complexity and Its Properties

Case	**Complexity**	**Property**	**Value**
Best Case	$O(n)$	Stable	Yes
Average Case	$O(n^2)$	In-place	Yes
Worst Case	$O(n^2)$	Adaptive	Yes
Space Complexity	$O(1)$ (in-place)	Recursive	No

7. Python Code with Explanation

```
def insertion_sort(arr):
    # Traverse from 1 to len(arr)
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1

        # Move elements greater than key to one position ahead
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1

        # Insert the key at the correct position
        arr[j + 1] = key

    return arr
```

8. Important Notes on Conditions

A. Difference Between `j >= 0` and `j > 0`

- `j >= 0` ensures that even the element at index `0` is compared and shifted if needed. It is the correct condition.
- `j > 0` skips the element at index `0`, which may result in incorrect sorting.

Example

Given `[3, 2]`:

- With `j >= 0`: Result → `[2, 3]` (Correct)
- With `j > 0`: Result → `[3, 2]` (Incorrect)

B. How to Make Insertion Sort Unstable

- Insertion Sort is stable because it does not move equal elements.
- If you change the condition in the while loop from `arr[j] > key` to `arr[j] >= key`, equal elements can be shifted, reversing their original order.

Listing 1: Unstable Version

```
while j >= 0 and arr[j] >= key: # Unstable: may move equal elements
    arr[j + 1] = arr[j]
    j -= 1
```

Stability Impact

`arr[j] > key` → Stable
`arr[j] >= key` → Unstable

C. Summary of Conditions

Condition	Effect
`j >= 0`	Correct — includes index 0 in comparisons
`j > 0`	Incorrect — skips index 0, may miss minimum value
`arr[j] > key`	Stable — preserves order of equal elements
`arr[j] >= key`	Unstable — may shift equal elements, reversing order

9. Making `arr[j] >= key` Stable

Why it Becomes Unstable

The condition `arr[j] >= key` causes instability because:

- It shifts equal elements to the right.
- This breaks their original relative order.

Stability Violation

Using `arr[j] >= key` moves earlier equal elements after the current one, reversing their order.

How to Make it Stable Even with >=

We can track the original positions of the elements by pairing each value with its index. This allows us to break ties using original position, preserving stability.

Listing 2: Stable Insertion Sort Using Index Tracking

```
def stable_insertion_sort(arr):
    # Attach original indices
    arr_with_index = [(val, idx) for idx, val in enumerate(arr)]

    for i in range(1, len(arr_with_index)):
        key = arr_with_index[i]
        j = i - 1

        while j >= 0 and (
            arr_with_index[j][0] > key[0] or
            (arr_with_index[j][0] == key[0] and arr_with_index[j][1] > key
                [1])
        ):
            arr_with_index[j + 1] = arr_with_index[j]
            j -= 1

        arr_with_index[j + 1] = key

    # Extract values without indices
    return [val for val, idx in arr_with_index]
```

Explanation

- Elements are stored as tuples: `(value, original_index)`.
- If values are equal, we preserve their original order using the index.
- This approach works even when using `>=`.

Result

The sort remains stable even when `arr[j] >= key` is used.

10. Step-by-Step Example with Index Tracking

We apply `stable_insertion_sort` on the input:

Input

`[5a, 3, 5b, 2]` *Here, 5a and 5b are equal values but come from different positions.*

Initial Array with Index

$$[(5,0),(3,1),(5,2),(2,3)]$$

—

Pass 1: i = 1 (key = (3,1))

Compare with (5,0):

$$(5 > 3) \Rightarrow \text{Shift (5,0) to right}$$

$$[(5,0),(5,0),(5,2),(2,3)] \Rightarrow \text{Insert (3,1) at index 0}$$

$$[(3,1),(5,0),(5,2),(2,3)]$$

—

Pass 2: i = 2 (key = (5,2))

Compare with (5,0):

$$(5 == 5 \text{ and } 0 < 2) \Rightarrow \text{Don't shift (5,0)} \Rightarrow \text{Insert (5,2) after (5,0)}$$

$$[(3,1),(5,0),(5,2),(2,3)]$$

—

Pass 3: i = 3 (key = (2,3))

Compare with (5,2):

$$(5 > 2) \Rightarrow \text{Shift}$$

Compare with (5,0):

$$(5 > 2) \Rightarrow \text{Shift}$$

Compare with (3,1):

$$(3 > 2) \Rightarrow \text{Shift}$$

Insert (2,3) at index 0:

$$[(2,3),(3,1),(5,0),(5,2)]$$

—

Final Sorted Output (Without Index)

$$[2,3,5a,5b]$$

Stability Verified

Even with `arr[j] >= key`, the relative order of equal elements (5a before 5b) is preserved.