Insertion Sort

Friday, December 26, 2025 9:09 PM

What is it?

A Simple comparison-based sorting algorithm that builds the final sorted array one item at a time.

Analogy

Think of sorting playing cards in your hand. You pick up a card and place it into its current position among the cards you are already holding.

Concept

It splits the array into a sorted part (left) and an unsorted part (right). Elements are picked from the unsorted part & placed at the current position in the sorted part.

How it Works

1. **Start**: Assume the first element (index 0) is already "sorted".
2. **Pick**: Take the next element (key) from the unsorted portion.
3. **Compare & shift**: Compare the key with elements in the sorted portion (moving right-to-left)
   - If the sorted element is larger than the key, shift that element one position to the right to make Space.
4. **Insert**: Once you find a value Smaller than the Key (or) reach the start of the list), insert the Key into that empty gap.
5. **Repeat**: Continue until the unsorted portion is empty.

Algorithm Complexity (Big O)

- **Time Complexity**:
  - **Best case**: $O(n)$
    The array is already Sorted (only 1 Comparison per element, no shifts).
  - **Worst Case**: $O(n^2)$
    The array is reverse sorted (max shifts required).
  - **Average case**: $O(n^2)$
- **Space Complexity**:
  It is an in-place algorithm.

Key Characteristics

- **Adaptive**: It is efficient for datasets that are already substantially sorted.
- **Stable**: Yes. It preserves the relative order of equal elements.
- **Online**: It can sort a list as it receives it (element by element).
- **Efficiency**: Generally faster than Bubble Sort & Selection Sort for small datasets, but inefficient for large unsorted lists.

Pseudocode

```
FUNCTION InsertionSort(list)
      n = length(list)
      /* start from the second element (index 1) */
      FOR i FROM 1 TO n
          key = list[i]
          j = i-1
          /* Move elements of lists[0...i-1] that are
          greater than key to one position ahead of their
          current position */
          WHILE j >= 0 AND list[j] > key
              list[j+1] = list[j]
              j = j-1
          END WHILE
          /* Place the key in its correct position */
          list[j+1] = key
      END FOR
END FUNCTION
```

Initial Setup

- list = [5, 2, 4] (indices 0, 1, 2)
- n = 3 (length of the list)

Iteration 1: i=1

We start the FOR loop at index 1 because we consider index[0] i.e, '5' is already "sorted".

1. Set key and J
   - key = list[1] → 2 (we save 2 so it doesn't get lost)
   - j = i-1 → 0
2. The WHILE loop (Compare & Shift)
   - check: Is j >= 0 AND list[j] > key ?
     - Is 0 >= 0 ? Yes
     - Is list[0] (i.e, 5) > key (i.e, 2) ? Yes
   - Action (The overwrite):
     - list[j+1] = list[j] → Copy list[0] into list[1]
     - State: [5, 5, 4] (Notice 2 is gone from the array, but safe in key).
     - j = j-1 → -1.
3. WHILE Check Again
   - Check: Is j >= 0 ?
     - Is -1 >= 0 ? No
   - Action: Exit the loop
4. Insert Key
   - list[j+1] = key
     - list[-1+1] is list[0]
     - Set list[0] = 2.
   - State: [2, 5, 4]

[2, 5, 4] (indices 0, 1, 2)

Iteration 2: i=2

Now we process the next element

1. Set Key and J:
   - Key = list[2] → 4 (save 4 so its safe)
   - j = i-1 → 1
2. The WHILE loop (Compare & shift):
   - Check: Is j >= 0 AND list[j] > key ?
     - Is 1 >= 0 ? Yes
     - Is 5 > 4 ? Yes
   - Action
     - list[j+1] = list[j] → Copy list[1] into list[2]
     - State: [2, 5, 5] (Notice 4 is overwritten in the array, but safe in key.)
     - j = j-1 → 0
3. WHILE Check Again.
   - Check: Is j >= 0 AND list[j] > Key ?
     - Is 0 >= 0 ? Yes
     - Is 2 > 4 ? No
   - Action: Exit the loop
4. Insert key
   - list[j+1] = key
     - list[0+1] is list[1]
     - Set list[1] = 4
   - State: [2, 4, 5]

Final Result

The FOR loop finishes because i reaches n-1.

- Final list: [2, 4, 5]