Insertion sort is a simple sorting algorithm that builds the final sorted array one item at a time. It works by repeatedly taking the next unsorted element and inserting it into its correct position within the sorted portion of the list. Insertion sort is efficient for small data sets or nearly sorted data but performs less efficiently on large, randomly ordered data sets.

Here's how the algorithm works in each step:

1. Start with the first element in the list, treating it as a sorted list of one element.

2. Take the next element and compare it to the elements in the sorted list.

3. Shift elements in the sorted list to the right until the correct position for the new element is found.

4. Insert the new element into its correct position.

5. Repeat until all elements are sorted.

Here’s a Python function to implement insertion sort:

def insertion\_sort(arr):

# Traverse through 1 to len(arr)

for i in range(1, len(arr)):

key = arr[i] # Element to be inserted in sorted part of array

j = i - 1

# Move elements of arr[0..i-1] that are greater than key

# to one position ahead of their current position

while j >= 0 and arr[j] > key:

arr[j + 1] = arr[j]

j -= 1

# Insert key at the correct position

arr[j + 1] = key

return arr

### Example Usage

```python

arr = [12, 11, 13, 5, 6]

sorted\_arr = insertion\_sort(arr)

print("Sorted array:", sorted\_arr)

### Explanation

- `key` is the current element that needs to be positioned correctly in the sorted portion.

- The `while` loop shifts elements that are greater than `key` to the right, making space for `key` to be placed in the correct position.

Here's a detailed explanation of the insertion sort algorithm with a visual aid, a step-by-step example, and code comments for clarity. We’ll illustrate how insertion sort works with a small dataset, along with a conceptual graph.

### Visual Explanation of Insertion Sort

Suppose we have an array: `[12, 11, 13, 5, 6]`. The algorithm works as follows:

1. \*\*Step 1\*\*: Consider the first element `[12]` as sorted.

2. \*\*Step 2\*\*: Insert `11` into the sorted portion `[12]`. Shift `12` to the right to make room for `11`.

- \*\*Array after insertion\*\*: `[11, 12, 13, 5, 6]`

3. \*\*Step 3\*\*: Insert `13`. Since `13` is greater than `12`, it remains in its place.

- \*\*Array after insertion\*\*: `[11, 12, 13, 5, 6]`

4. \*\*Step 4\*\*: Insert `5`. Shift `13`, `12`, and `11` to the right to make room for `5`.

- \*\*Array after insertion\*\*: `[5, 11, 12, 13, 6]`

5. \*\*Step 5\*\*: Insert `6`. Shift `13`, `12`, and `11` to the right to make room for `6`.

- \*\*Array after insertion\*\*: `[5, 6, 11, 12, 13]`

The array is now sorted: `[5, 6, 11, 12, 13]`.

---

### Graphical Representation of Steps

Imagine a bar chart where each bar represents an element in the array. At each step:

- Elements in the sorted portion (left side) remain fixed.

- The "key" element (currently being inserted) moves through the sorted section to its correct position, and the bars shift accordingly.

In a conceptual graph, the insertion sort can be visualized with each step showing fewer and fewer shifts as elements near their final positions.

---

### Python Code with Comments

Here's the Python function for insertion sort, with comments explaining each part of the process.

```python

def insertion\_sort(arr):

# Traverse from the second element to the last element

for i in range(1, len(arr)):

key = arr[i] # Current element to be inserted in sorted part of array

j = i - 1

# Move elements that are greater than key to one position ahead

# of their current position, creating a space for the key

while j >= 0 and arr[j] > key:

arr[j + 1] = arr[j] # Shift element to the right

j -= 1

# Insert the key at its correct position

arr[j + 1] = key

return arr

```

### Example Usage and Output

```python

# Sample array to be sorted

arr = [12, 11, 13, 5, 6]

sorted\_arr = insertion\_sort(arr)

print("Sorted array:", sorted\_arr)

```

### Explanation of Each Part of the Code

1. \*\*Outer Loop (`for i in range(1, len(arr))`)\*\*:

- Starts with the second element (`i = 1`) since a single-element array (first element) is already "sorted."

2. \*\*Key Assignment (`key = arr[i]`)\*\*:

- `key` stores the current element that needs to be placed in the sorted portion.

3. \*\*Inner While Loop\*\*:

- This loop shifts elements in the sorted portion to the right until it finds the correct position for `key`.

- The loop continues as long as `j >= 0` (not out of bounds) and `arr[j] > key` (elements larger than `key` need shifting).

4. \*\*Insertion of Key (`arr[j + 1] = key`)\*\*:

- After shifting elements, the correct position for `key` is found at `j + 1`.

### Time Complexity and Graph of Performance

Insertion sort has an average and worst-case time complexity of \(O(n^2)\), where \(n\) is the number of elements. This quadratic time complexity occurs due to nested loops, especially for an array sorted in reverse order. However, it performs well on nearly sorted or small arrays, often close to \(O(n)\).

For performance analysis, if we plot a graph of \*\*input size (n)\*\* vs. \*\*time taken\*\*, insertion sort would show a parabolic curve due to its \(O(n^2)\) behavior on large, unsorted arrays but near-linear on small or nearly sorted arrays.

Let me know if you’d like a graph illustration of insertion sort’s performance in Python!