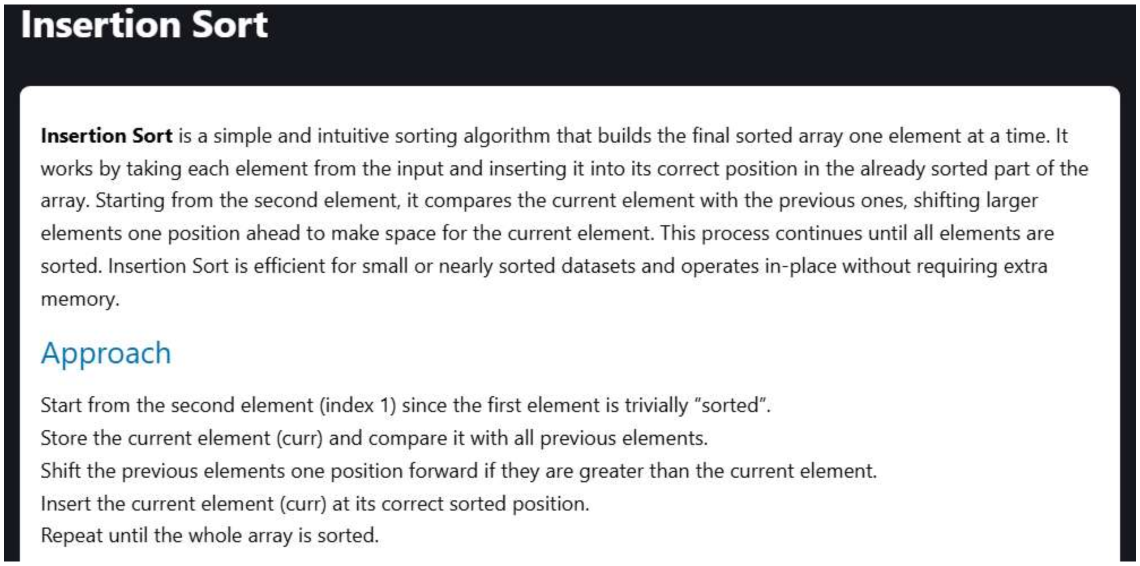


Insertion Sort

Insertion Sort is a simple and intuitive sorting algorithm that builds the final sorted array one element at a time. It works by taking each element from the input and inserting it into its correct position in the already sorted part of the array. Starting from the second element, it compares the current element with the previous ones, shifting larger elements one position ahead to make space for the current element. This process continues until all elements are sorted. Insertion Sort is efficient for small or nearly sorted datasets and operates in-place without requiring extra memory.

Approach

Start from the second element (index 1) since the first element is trivially “sorted”.
Store the current element (curr) and compare it with all previous elements.
Shift the previous elements one position forward if they are greater than the current element.
Insert the current element (curr) at its correct sorted position.
Repeat until the whole array is sorted.

Approach

Start from the second element (index 1) since the first element is trivially “sorted”.
Store the current element (curr) and compare it with all previous elements.
Shift the previous elements one position forward if they are greater than the current element.
Insert the current element (curr) at its correct sorted position.
Repeat until the whole array is sorted.

Dry Run (on [4, 5, 1, 3, 9])

Initial array:
[4, 5, 1, 3, 9]

Pass 1 (i = 1):
curr = 5, prev = 0

5 > 4, no shifting → array remains [4, 5, 1, 3, 9]

Pass 2 (i = 2):
curr = 1, prev = 1

1 < 5 → shift 5 → [4, 5, 5, 3, 9]
1 < 4 → shift 4 → [4, 4, 5, 3, 9]
Insert 1 → [1, 4, 5, 3, 9]

Pass 3 (i = 3):
curr = 3, prev = 2

3 < 5 → shift 5 → [1, 4, 5, 5, 9]
3 < 4 → shift 4 → [1, 4, 4, 5, 9]
Insert 3 → [1, 3, 4, 5, 9]

Pass 4 (i = 4):
curr = 9, prev = 3

9 > 5, no shifting → array remains [1, 3, 4, 5, 9]

Final Sorted Array:
[1, 3, 4, 5, 9]

Time Complexity

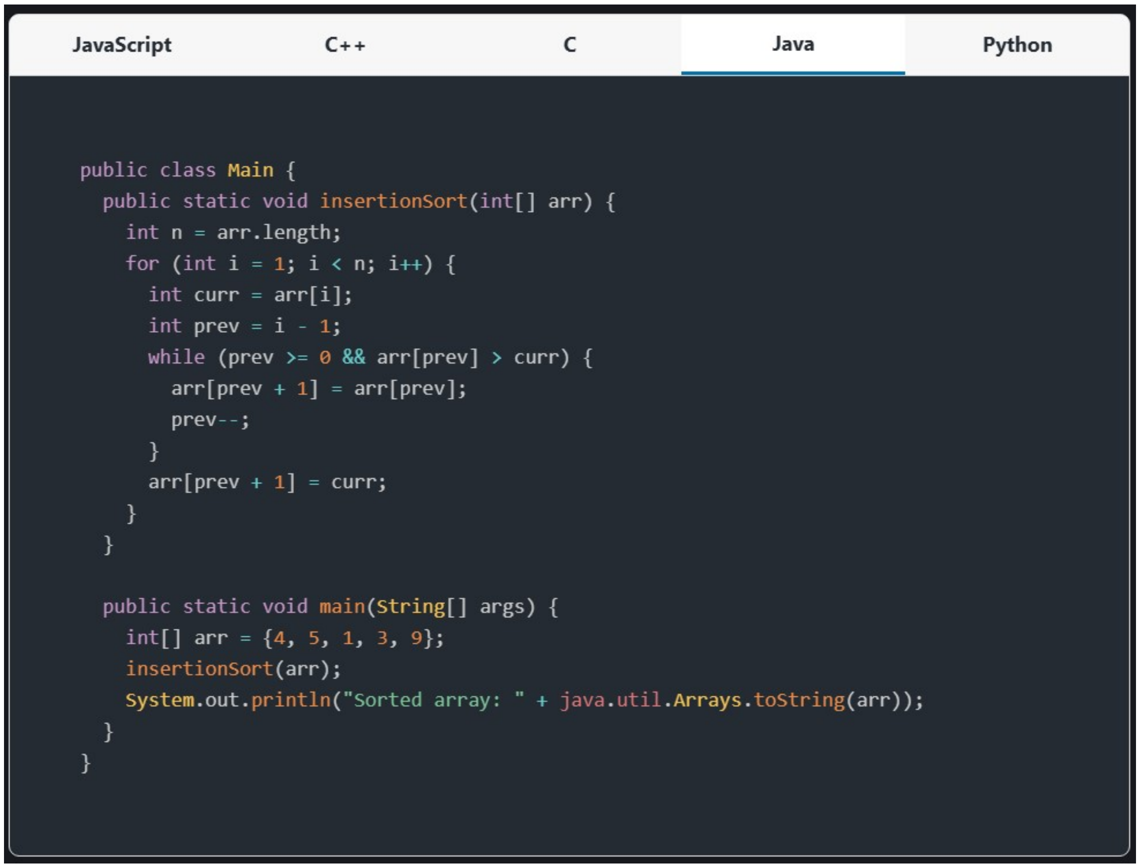
Best Case (already sorted): O(n)
Average Case: O(n^2)
Worst Case (reverse sorted): O(n^2)

Why?

Best case: No shifting happens.
Worst case: Every element has to be compared and shifted back to the start.

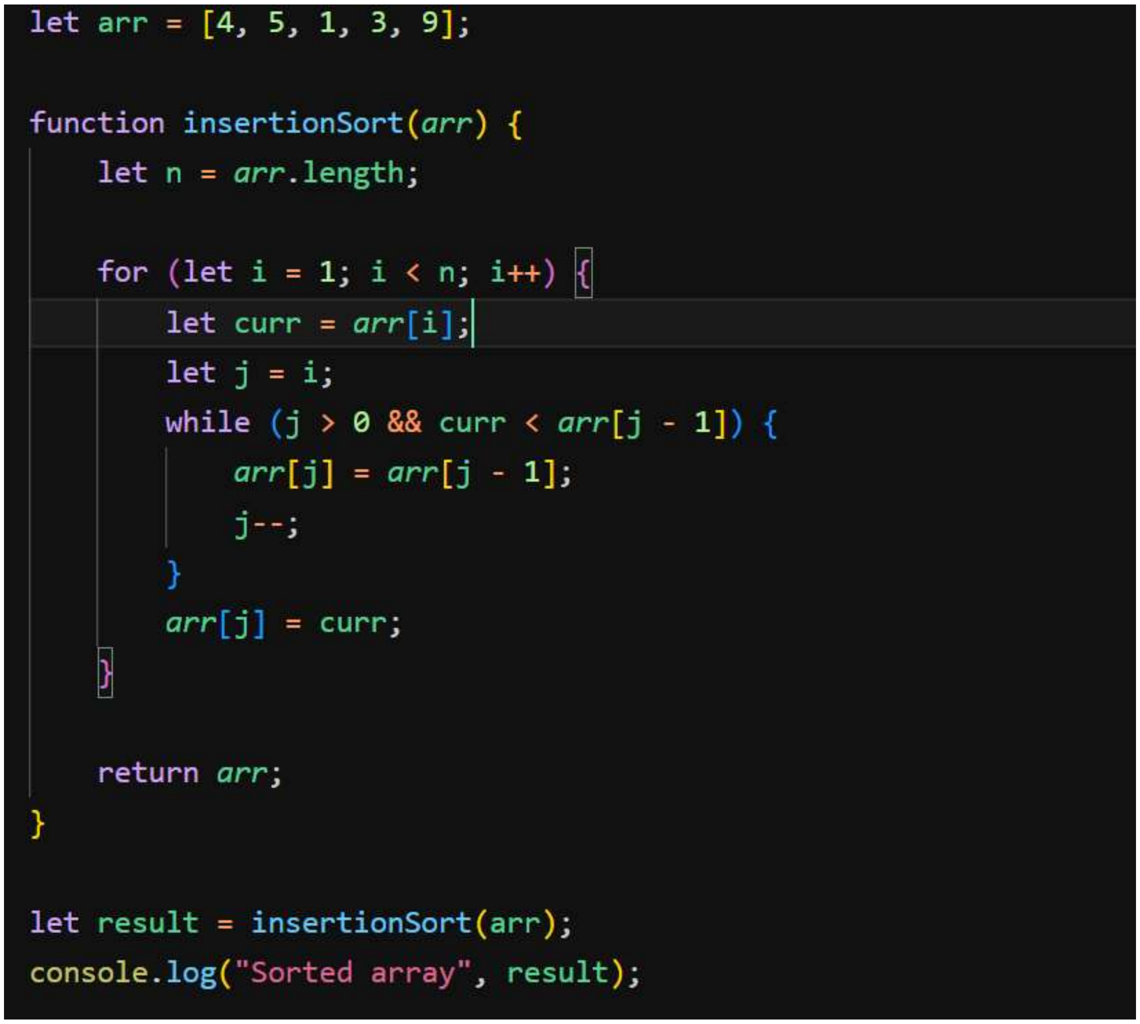
Space Complexity

O(1) → No extra array is used; sorting is done in-place.



```java
public class Main {
  public static void insertionSort(int[] arr) {
    int n = arr.length;
    for (int i = 1; i < n; i++) {
      int curr = arr[i];
      int prev = i - 1;
      while (prev >= 0 && arr[prev] > curr) {
        arr[prev + 1] = arr[prev];
        prev--;
      }
      arr[prev + 1] = curr;
    }
  }

  public static void main(String[] args) {
    int[] arr = {4, 5, 1, 3, 9};
    insertionSort(arr);
    System.out.println("Sorted array: " + java.util.Arrays.toString(arr));
  }
}
```



```javascript
let arr = [4, 5, 1, 3, 9];

function insertionSort(arr) {
    let n = arr.length;

    for (let i = 1; i < n; i++) {
        let curr = arr[i];
        let j = i;
        while (j > 0 && curr < arr[j - 1]) {
            arr[j] = arr[j - 1];
            j--;
        }
        arr[j] = curr;
    }

    return arr;
}

let result = insertionSort(arr);
console.log("Sorted array", result);
```