* .
* **1. Bubble Sort**
* def bubble\_sort(arr):
* n = len(arr)
* for i in range(n):
* for j in range(0, n - i - 1):
* if arr[j] > arr[j + 1]:
* arr[j], arr[j + 1] = arr[j + 1], arr[j]
* # Example
* arr = [64, 34, 25, 12, 22, 11, 90]
* print("Original:", arr)
* bubble\_sort(arr)
* print("Sorted:", arr)
* **2. Insertion Sort**
* def insertion\_sort(arr):
* for i in range(1, len(arr)):
* key = arr[i]
* j = i - 1
* while j >= 0 and key < arr[j]:
* arr[j + 1] = arr[j]
* j -= 1
* arr[j + 1] = key
* # Example
* arr = [12, 11, 13, 5, 6]
* print("Original:", arr)
* insertion\_sort(arr)
* print("Sorted:", arr)
* **3. Selection Sort**
* def selection\_sort(arr):
* n = len(arr)
* for i in range(n):
* min\_idx = i
* for j in range(i+1, n):
* if arr[j] < arr[min\_idx]:
* min\_idx = j
* arr[i], arr[min\_idx] = arr[min\_idx], arr[i]
* # Example
* arr = [64, 25, 12, 22, 11]
* print("Original:", arr)
* selection\_sort(arr)
* print("Sorted:", arr)
* **4. Merge Sort**
* def merge\_sort(arr):
* if len(arr) > 1:
* mid = len(arr) // 2
* L = arr[:mid]
* R = arr[mid:]
* merge\_sort(L)
* merge\_sort(R)
* i = j = k = 0
* while i < len(L) and j < len(R):
* if L[i] < R[j]:
* arr[k] = L[i]
* i += 1
* else:
* arr[k] = R[j]
* j += 1
* k += 1
* while i < len(L):
* arr[k] = L[i]
* i += 1
* k += 1
* while j < len(R):
* arr[k] = R[j]
* j += 1
* k += 1
* # Example
* arr = [38, 27, 43, 3, 9, 82, 10]
* print("Original:", arr)
* merge\_sort(arr)
* print("Sorted:", arr)
* **5. Quick Sort**
* def quick\_sort(arr):
* if len(arr) <= 1:
* return arr
* pivot = arr[len(arr)//2]
* left = [x for x in arr if x < pivot]
* middle = [x for x in arr if x == pivot]
* right = [x for x in arr if x > pivot]
* return quick\_sort(left) + middle + quick\_sort(right)
* # Example
* arr = [10, 7, 8, 9, 1, 5]
* print("Original:", arr)
* arr = quick\_sort(arr)
* print("Sorted:", arr)
* **6. Heap Sort**
* def heapify(arr, n, i):
* largest = i
* l = 2\*i + 1
* r = 2\*i + 2
* if l < n and arr[l] > arr[largest]:
* largest = l
* if r < n and arr[r] > arr[largest]:
* largest = r
* if largest != i:
* arr[i], arr[largest] = arr[largest], arr[i]
* heapify(arr, n, largest)
* def heap\_sort(arr):
* n = len(arr)
* for i in range(n//2 - 1, -1, -1):
* heapify(arr, n, i)
* for i in range(n-1, 0, -1):
* arr[0], arr[i] = arr[i], arr[0]
* heapify(arr, i, 0)
* # Example
* arr = [12, 11, 13, 5, 6, 7]
* print("Original:", arr)
* heap\_sort(arr)
* print("Sorted:", arr)
* **7. Counting Sort (works for non-negative integers)**
* def counting\_sort(arr):
* max\_val = max(arr)
* count = [0] \* (max\_val + 1)
* for num in arr:
* count[num] += 1
* i = 0
* for num in range(len(count)):
* while count[num] > 0:
* arr[i] = num
* i += 1
* count[num] -= 1
* # Example
* arr = [4, 2, 2, 8, 3, 3, 1]
* print("Original:", arr)
* counting\_sort(arr)
* print("Sorted:", arr)
* **8. Radix Sort (uses Counting Sort internally)**
* def counting\_sort\_exp(arr, exp):
* n = len(arr)
* output = [0] \* n
* count = [0] \* 10
* for i in range(n):
* index = arr[i] // exp
* count[index % 10] += 1
* for i in range(1, 10):
* count[i] += count[i - 1]
* i = n - 1
* while i >= 0:
* index = arr[i] // exp
* output[count[index % 10] - 1] = arr[i]
* count[index % 10] -= 1
* i -= 1
* for i in range(n):
* arr[i] = output[i]
* def radix\_sort(arr):
* max\_val = max(arr)
* exp = 1
* while max\_val // exp > 0:
* counting\_sort\_exp(arr, exp)
* exp \*= 10
* # Example
* arr = [170, 45, 75, 90, 802, 24, 2, 66]
* print("Original:", arr)
* radix\_sort(arr)
* print("Sorted:", arr)
* ✅ With these, you have **all the major sorting algorithms** in Python (simple + efficient).

**🔹 1. Bubble Sort**

**How it works:**

* Repeatedly traverse the array.
* Compare each adjacent pair and swap if out of order.
* After the 1st pass, the largest element is at the end.
* After the 2nd pass, the 2nd largest is in place, and so on.

👉 Imagine bubbles rising to the surface — the largest element “bubbles up.”

**Complexities:**

* Best: **O(n)** (if array already sorted → just one pass, no swaps)
* Average: **O(n²)**
* Worst: **O(n²)**
* Space: **O(1)** (in-place)
* Stability: ✅ Stable (adjacent equal elements don’t swap unnecessarily)

**When to use:** Rarely in practice; good for teaching sorting basics.

**🔹 2. Insertion Sort**

**How it works:**

* Start from the second element.
* Compare it with elements before it and “insert” it into the correct position.
* Repeat for all elements.

👉 Think of sorting playing cards in your hand.

**Complexities:**

* Best: **O(n)** (if array already sorted, just compare once per element)
* Average: **O(n²)**
* Worst: **O(n²)**
* Space: **O(1)** (in-place)
* Stability: ✅ Stable

**When to use:**

* Very efficient for **small arrays** or **nearly sorted arrays**.
* Often used in hybrid algorithms (e.g., Python’s **Timsort** uses it for small chunks).

**🔹 3. Selection Sort**

**How it works:**

* Repeatedly find the **minimum element** in the unsorted part.
* Swap it with the first element of the unsorted part.
* After each pass, one more element is placed in its correct spot.

👉 Like picking the smallest card from a pile and placing it in front.

**Complexities:**

* Best: **O(n²)**
* Average: **O(n²)**
* Worst: **O(n²)**
* Space: **O(1)**
* Stability: ❌ Not stable (because of swapping min with first element).

**When to use:**

* Rare in practice (worse than Insertion Sort).
* Useful if **swaps are expensive but comparisons are cheap**.

**🔹 4. Merge Sort**

**How it works (Divide & Conquer):**

* Divide the array into halves.
* Recursively sort each half.
* Merge the two sorted halves into one sorted array.

👉 Like breaking a book in half, sorting each half, then merging page by page.

**Complexities:**

* Best: **O(n log n)**
* Average: **O(n log n)**
* Worst: **O(n log n)**
* Space: **O(n)** (needs extra memory for merging)
* Stability: ✅ Stable

**When to use:**

* Good for **linked lists** (doesn’t need random access).
* Preferred when **stability is required**.
* Used in **external sorting** (sorting huge data on disk).

**🔹 5. Quick Sort**

**How it works (Divide & Conquer):**

* Pick a **pivot** (e.g., last element).
* Partition the array → elements smaller go left, larger go right.
* Recursively sort left and right partitions.

👉 Like organizing books around a shelf divider (pivot).

**Complexities:**

* Best: **O(n log n)**
* Average: **O(n log n)**
* Worst: **O(n²)** (bad pivot choices, e.g., already sorted input with naive pivot)
* Space: **O(log n)** (recursive stack, in-place)
* Stability: ❌ Not stable (unless modified).

**When to use:**

* Very popular in practice.
* Usually fastest due to **good cache performance** and **in-place nature**.
* Most languages’ default sorting (C++, Java) are QuickSort/Hybrid.

**🔹 6. Heap Sort**

**How it works:**

* Build a **Max Heap** (binary tree where parent ≥ children).
* Repeatedly extract the max (root), place it at the end, and rebuild heap.

👉 Like repeatedly taking the tallest person out of a group and placing them in order.

**Complexities:**

* Best: **O(n log n)**
* Average: **O(n log n)**
* Worst: **O(n log n)**
* Space: **O(1)** (in-place)
* Stability: ❌ Not stable

**When to use:**

* Good when **memory is limited**.
* Worse than QuickSort in practice due to poor cache performance.

**🔹 7. Counting Sort**

**How it works (for integers in a small range):**

* Count occurrences of each number.
* Compute cumulative counts.
* Place numbers in correct positions in output array.

👉 Like counting how many students scored 0, 1, 2 … and placing them in sorted order.

**Complexities:**

* Best: **O(n + k)**
* Average: **O(n + k)**
* Worst: **O(n + k)**  
  (where k = range of input values)
* Space: **O(n + k)**
* Stability: ✅ Stable

**When to use:**

* Works only on **integers / discrete values** with **small range**.
* Used in **digit-based sorting** (like Radix Sort).

**🔹 8. Radix Sort**

**How it works:**

* Sort elements digit by digit (least significant → most significant).
* Uses **Counting Sort** as a subroutine for each digit.

👉 Like sorting exam papers by units place, then tens place, then hundreds place.

**Complexities:**

* Best: **O(nk)**
* Average: **O(nk)**
* Worst: **O(nk)**  
  (where k = number of digits in max number)
* Space: **O(n + k)**
* Stability: ✅ Stable

**When to use:**

* Good for **large integers / fixed-length strings**.
* Outperforms comparison-based sorts when k is small.

**🔹 9. Bucket Sort**

**How it works:**

* Divide range into **buckets**.
* Distribute elements into buckets.
* Sort each bucket (usually with Insertion Sort).
* Concatenate buckets.

👉 Like distributing exam scores into grade buckets, then sorting within each grade.

**Complexities:**

* Best: **O(n + k)**
* Average: **O(n + k)**
* Worst: **O(n²)** (if all go into one bucket)
* Space: **O(n + k)**
* Stability: ✅ Stable (depends on bucket sorting method).

**When to use:**

* Works best for **uniformly distributed real numbers** in a range.

**📊 Master Comparison Table (Interview-Ready)**

| **Algorithm** | **Best** | **Average** | **Worst** | **Space** | **Stable** | **Notes** |
| --- | --- | --- | --- | --- | --- | --- |
| Bubble Sort | O(n) | O(n²) | O(n²) | O(1) | ✅ | Simple, teaching only |
| Insertion Sort | O(n) | O(n²) | O(n²) | O(1) | ✅ | Best for small / nearly sorted |
| Selection Sort | O(n²) | O(n²) | O(n²) | O(1) | ❌ | Few swaps, bad in general |
| Merge Sort | O(n log n) | O(n log n) | O(n log n) | O(n) | ✅ | Great for linked lists, stable |
| Quick Sort | O(n log n) | O(n log n) | O(n²) | O(log n) | ❌ | Very fast in practice |
| Heap Sort | O(n log n) | O(n log n) | O(n log n) | O(1) | ❌ | In-place, slower than QuickSort |
| Counting Sort | O(n + k) | O(n + k) | O(n + k) | O(n + k) | ✅ | Only for integers with small range |
| Radix Sort | O(nk) | O(nk) | O(nk) | O(n + k) | ✅ | Works on numbers & strings |
| Bucket Sort | O(n + k) | O(n + k) | O(n²) | O(n + k) | ✅ | Great for uniform distributions |

✅ With this, you’ll have a **crystal-clear story for interviews**.  
A common question they ask:  
👉 *“If you had to sort 10 million integers, which would you use?”*

* If range is **small** → Counting/Radix.
* If range is **large/random** → QuickSort or MergeSort.
* If **memory is tight** → HeapSort.