Insertion Sort — Comprehensive Notes

1) What is sorting?

- **Task:** Reorder a collection according to a specified **total order**.
- **Inputs in CS:** Typically an **array** (or list) of items.
- **Order examples:**
  - Numbers: ascending using `≤`
  - Strings: lexicographic (dictionary) order
  - Custom objects: any order defined by a comparator or `key` function

Two invariants (must-haves)

1. **Same multiset of elements.** Output contains exactly the input elements (no additions/deletions).
2. **Sorted according to the given order.** For ascending, `Aˆ[1] ≤ Aˆ[2] ≤ … ≤ Aˆ[n]`.

Descending sort is identical except you use `≥` as the order.

---

2) Where sorting shows up

- Spreadsheets (sort by any column)
- Program languages' generic `sort` (numbers, strings, user-defined objects via comparator or key)
- Organizing "big data" pipelines before joins/merges, etc.

Common algorithms (you'll meet later): **Insertion sort**, **Merge sort**, **Heapsort**, **Quicksort**, and hybrids like **Timsort** (Python's default). A practical fast combo is **Quicksort + Insertion sort (for small runs)**.

---

3) Big-picture idea of Insertion Sort

We maintain two regions inside the array:

- **Sorted prefix (left/pink)**
- **Unsorted suffix (right/blue)**

We scan elements from left to right. For each new element `A[j]` in the unsorted part, **insert** it into the correct position inside the sorted prefix, shifting larger elements one spot to the right.

Illustrative example

Start: `[2, 7, 4, 1, 3, 6, 5, 0]`
Sorted grows step by step:

- `[2] | 7 4 1 3 6 5 0`
- insert 7 → `[2, 7] | 4 1 3 6 5 0` (7 already in place)
- insert 4 → `[2, 4, 7] | 1 3 6 5 0`
- insert 1 → `[1, 2, 4, 7] | 3 6 5 0`
- …
- final: `[0, 1, 2, 3, 4, 5, 6, 7]`

---

4) The Insert subroutine (local step)

Two equivalent ways to implement the "insert `A[j]` into sorted `A[1..j-1]`":

(A) Right-to-left swapping (matches your lecture narrative)

You compare adjacent pairs and swap while the new item is "too small," bubbling it left into place.

Pseudocode (1-indexed, inclusive bounds)

```
procedure INSERT(A, j):
    for i = j-1 down to 1:
        if A[i] > A[i+1]:
            swap(A[i], A[i+1])
        else:
            break
```

- The loop both **finds the position** and **does the shifts** (via swaps).
- Early `break` if the element is already in the proper place (best case).

(B) Shifting version (classic insertion sort style; fewer swaps)

Store `key = A[j]`, shift larger elements right by one, and place the key once.

Pseudocode (1-indexed)

```
procedure INSERT(A, j):
    key ← A[j]
    i ← j - 1
    while i ≥ 1 and A[i] > key:
        A[i+1] ← A[i]       # shift right
        i ← i - 1
    A[i+1] ← key            # drop key into hole
```

- This does **O(shifts)** assignments and typically fewer writes than swap-based.

---

5) Full Insertion Sort algorithm

Pseudocode (1-indexed arrays, both loop ends inclusive)

```
procedure INSERTION-SORT(A, n):
```

```
for j = 1 to n:
    INSERT(A, j)
```

Notes:

- In practice we often start at `j = 2` since a single element is trivially sorted.
- Pseudocode conventions here are **1-indexed** and `for` bounds are **inclusive**.

---

6) Correctness sketch (loop invariant)

Invariant: At the start of each outer iteration `j`, the prefix `A[1..j-1]` is sorted and contains exactly the original elements from positions `1..j-1`.

- **Initialization:** For `j=2`, `A[1]` is trivially sorted.
- **Maintenance:** `INSERT` places `A[j]` into the correct place in `A[1..j-1]`, preserving sortedness and membership.
- **Termination:** When `j = n+1`, the invariant implies `A[1..n]` is sorted and a permutation of the input.

---

7) Time/space complexity

Cost model (as in the lecture)

Let:

- C_1 = cost of a comparison,
- C_2 = cost of a swap (or constant number of assignments in shifting),
- C_3 = loop bookkeeping per iteration,

- (optionally) C_4 = constant for a quick `break/return`.

Insert step

- **Best case:** New element already ≥ last of sorted prefix → one check then break.
  Cost ≈ $C_3 + C_1$ $(+ C_4)$ = **Θ(1)**

- **Worst case:** Must travel to the beginning, doing $j-1$ iterations, each with compare + swap + loop cost.
  Cost ≈ $(j-1)\cdot(C_1 + C_2 + C_3)$ = **Θ(j)**

Insertion Sort overall

- **Best case (already sorted array):** Each insert is Θ(1) → **Θ(n)**

- **Worst case (reverse-sorted):** Inserts cost $1 + 2 + \dots + (n-1)$ →
  $\Sigma\ (j-1) = n(n-1)/2$ times $(C_1 + C_2 + C_3)$ → **Θ(n^2)**

- **Average case: Θ(n^2)** (not derived in this lecture, but standard)

Space

- **In-place: O(1)** extra space.

- **Stable:** Yes (equal keys preserve relative order) — both versions are stable if you use `>` and not `≥`.

---

8) Practical notes

- Great for **small arrays** and **nearly-sorted data** (adaptive behavior).

- Often used inside faster sorts (e.g., switch to insertion sort when subarray size ≤ threshold).

- For custom objects, pass a **key extractor** or **comparator**.

9) Python implementations

A) Swap-based “insert” (close to the lecture demo)

```
def insertion_sort_swap(a):
    a = list(a)  # copy if you want to keep input
    n = len(a)
    for j in range(n):            # 0-indexed; corresponds to 1..n in pseudocode
        i = j - 1
        while i >= 0 and a[i] > a[i+1]:
            a[i], a[i+1] = a[i+1], a[i]   # swap neighbors
            i -= 1
    return a
```

B) Shifting version (classic; fewer writes)

```
def insertion_sort(a):
    a = list(a)
    n = len(a)
    for j in range(1, n):         # start at second item
        key = a[j]
        i = j - 1
        # move elements greater than key one step to the right
        while i >= 0 and a[i] > key:
            a[i+1] = a[i]
            i -= 1
        a[i+1] = key
    return a
```

C) With a **key** function (custom ordering)

```
def insertion_sort_key(a, key=lambda x: x):
    a = list(a)
    n = len(a)
    for j in range(1, n):
        item = a[j]
```

```
        k = key(item)
        i = j - 1
        while i >= 0 and key(a[i]) > k:
            a[i+1] = a[i]
            i -= 1
        a[i+1] = item
    return a
```

D) Descending order

- Replace `>` with `<` in while-condition (or sort keys negated, or pass `reverse=True` in an adapted API).

---

10) Worked trace (shifting version) on your example

Array `a = [2, 7, 4, 1, 3, 6, 5, 0]`

- `j=1, key=7`: compare with 2 → 7 ≥ 2 → no shifts → `[2, 7, 4, 1, 3, 6, 5, 0]`
- `j=2, key=4`: compare with 7 → shift 7 → `[2, 7, 7, 1, 3, 6, 5, 0]`
  then 4 ≥ 2 → stop → place key → `[2, 4, 7, 1, 3, 6, 5, 0]`
- `j=3, key=1`: shift 7, 4, 2 → `[2, 2, 4, 7, 3, 6, 5, 0]` → place key at 0 → `[1, 2, 4, 7, 3, 6, 5, 0]`
- …
- final → `[0, 1, 2, 3, 4, 5, 6, 7]`

---

11) When is each case triggered?

- **Best case Θ(n):** Input already sorted ascending.

- **Worst case Θ(n²):** Input sorted **descending** (reverse order), each new element must travel to the front.
- **Nearly sorted:** Very fast in practice (few shifts).

---

12) Quick reference (copy/paste)

Pseudocode (1-indexed)

```
procedure INSERTION-SORT(A, n):
    for j = 2 to n:
        key ← A[j]
        i ← j - 1
        while i ≥ 1 and A[i] > key:
            A[i+1] ← A[i]
            i ← i - 1
        A[i+1] ← key
```

Python (0-indexed)

```
def insertion_sort(a):
    a = list(a)
    for j in range(1, len(a)):
        key = a[j]
        i = j - 1
        while i >= 0 and a[i] > key:
            a[i+1] = a[i]
            i -= 1
        a[i+1] = key
    return a
```

---

13) Practice prompts

1. Prove stability of the shifting version (equal keys keep relative order).

2. Modify insertion sort to sort **descending**.

3. Adapt `insertion_sort_key` to sort a list of `(last_name, first_name)` by `last_name`, then `first_name`.

4. Show that if the array has at most `k` inversions per element, insertion sort runs in `Θ(n·k)` time.