Time and Space Complexity Analysis — Comprehensive Notes

1) Why compare algorithms?

When given two algorithms for the same task:

- **Goal:** Decide which is preferable (faster, more efficient, or less resource-heavy).
- **Key resources:**
    1. **Time complexity** → How many steps (or operations) are needed?
    2. **Space complexity** → How much memory is consumed?

  Often, algorithms that use more time also use more space, but they are studied separately.

---

2) Challenges in comparing algorithms

Naïve approach

- Run both algorithms on the same input, compare runtimes.
- Problems:
    - Which **input** should you choose (small vs. large, sorted vs. random)?
    - **Implementer differences**: Novice vs. expert programmer.
    - **Language/runtime differences**: C++ vs. Python vs. assembly.
    - **Hardware differences**: CPU type, cores, RAM, compiler optimizations.

This kind of comparison is called **performance analysis** (real implementation-focused).
Instead, theoretical CS uses **algorithm analysis**, which abstracts away from machines.

---

3) Abstract cost model

To avoid hardware/language dependence, we define a **unit cost model**:

- Addition = 1 operation
- Multiplication = 1 operation (even if slower in practice)
- Comparison (`if`) = 1 operation
- `for` loop = initialization (1 op) + increment (1 op) + condition check (1 op) per iteration

> This allows us to **count operations** instead of measuring actual seconds → called **op count**.

---

4) Role of input size

- Denote input size as `n`.
- Example: For sorting, `n` = length of the array.
- Plot:
  - **X-axis:** input size `n`
  - **Y-axis:** operation count
- Problem: For the same `n`, many possible inputs exist → runtimes differ.

---

5) Best case, Worst case, Average case

Best case

- Input is arranged so algorithm finishes fastest.
- Example: Insertion sort's best case is when array is already sorted → $\Theta(n)$.
- Rare in practice → considered **self-serving** (not reliable for evaluation).

Worst case

- Input triggers the longest possible runtime.
- Example: Insertion sort's worst case is reverse-sorted input → $\Theta(n^2)$.
- Guarantees an **upper bound**: "no input will take longer."
- Widely used in CS.

Average case

- Expected runtime over all possible inputs, assuming a probability distribution.
- Example: Random arrays for sorting.
- Requires probability/expectation analysis.
- Gives a realistic view of "typical" runtime but harder to analyze.

In practice:

- Best case is rarely considered.
- Worst case is most common.
- Average case is used for some algorithms where worst case is misleading.

---

6) Example: Insertion Sort vs Merge Sort

Suppose detailed op-count analysis yields:

Insertion sort worst case:

```
f_IS(n) = 0.05n² + 2.5n + 1.5
```

-

Merge sort worst case:

```
f_MS(n) = 2000n log n + 1500n + 20000
```

-

Observations

- For **small n**, insertion sort is faster (smaller constants, simpler operations).
- For **large n**, merge sort overtakes ($n \log n$ grows slower than n^2).
- **Crossover point**: The input size n where merge sort becomes better.

---

7) Constants and asymptotics

- Constants depend on:
  - Hardware (multiplication cost vs addition cost, etc.)
  - Clever coding (reducing comparisons/swaps)
- To **abstract away constants and small inputs**, we use **asymptotic notation**.

---

8) Asymptotic notation

Big-O (O)

- Upper bound ("≤")

- Example: Insertion sort is **$O(n^2)$** (never worse than quadratic time).

Big-Ω (Ω)

- Lower bound ("≥")
- Example: Insertion sort is **$\Omega(n)$** (at least linear time, even in best case).

Big-Θ (Θ)

- Tight bound ("=")
- Example: Insertion sort is **$\Theta(n^2)$** in the worst case (both upper and lower bound quadratic).

Asymptotic analysis focuses on **long-term trends** as $n \to \infty$, ignoring constants and small input effects.

---

9) Graphical comparison

For large input sizes:

- **Insertion Sort:** curve rises as n^2 → steeper slope.
- **Merge Sort:** curve rises as $n \log n$ → shallower slope.
- Intersection point = "crossover size."
- Beyond crossover: Merge sort dominates.

---

10) Summary (takeaways)

- **Metric:** Resource consumption (time, space).
- **Ignore:** Implementation details, programming language, machine.

- **Method:** Count primitive operations (cost model).
- **Inputs:** Measure complexity as function of input size n.
- **Cases:** Best (rare), Worst (guarantee), Average (typical but harder).
- **Goal:** Use asymptotic analysis (Big-O, Ω, Θ) to describe long-term growth rates.
- **Result:** In practice, worst-case asymptotics guide algorithm choice.

---

11) Quick Example (Python Op Count Approximation)

```
def insertion_sort_ops(n):
    # best case: already sorted
    best = n        # ~ 1 comparison per element
    # worst case: reverse sorted
    worst = n*(n-1)//2  # number of shifts
    # average ~ half of worst
    avg = worst // 2
    return best, avg, worst

print(insertion_sort_ops(10))
# (10, 45, 90)
```

---

12) Key Intuition

- **Best case:** Almost never happens → ignore.
- **Worst case:** Guarantees safety (upper bound).
- **Average case:** Most realistic but requires probability.
- **Asymptotic trend:** Only long-term growth matters → constants don't.

👉 Next step (as in your lecture): Learn formal **Big-O, Big-Ω, Big-Θ definitions** and apply them systematically.