1. Real life example

👉 Imagine you **buy a metro smart card**.

- The first time, you **recharge ₹100** and swipe. That one recharge took time (say 5 minutes at the counter).
- Next 9 rides you just **swipe** (instant, 1 second).
- On the 11th ride, you again recharge (another 5 minutes).

If you calculate **per ride time**:

- Recharge time seems expensive.
- But if you **spread (amortize) the recharge time across multiple rides**, each ride effectively costs only a few seconds extra.

⚡ **Key Idea:**
Instead of analyzing the worst case of one operation (recharge), we spread the cost across many operations → that's **Amortized Complexity**.

---

2. Formal Explanation

- Worst case: Sometimes a single operation looks very costly (like resizing an array, rehashing in a hash map, or recharging your metro card).
- Amortized case: When spread across multiple operations, the *average per operation* is still cheap.

⚡ Classic use cases:

- Dynamic Array resizing (ArrayList in Java, vector in C++)
- Union-Find with path compression
- Hashing with rehash

---

3. Problem Example – Dynamic Array Doubling

📌 Problem: You are inserting n elements into a dynamic array that **doubles its size whenever it's full**. What's the amortized cost per insertion?

Naive Analysis:

- Normal insert = O(1) (if space is available).
- But when resizing: copying takes O(current size).
- Looks costly if we only look at that step.

Amortized Analysis:

- Insert 1st element → no copy.

When array grows from size 1 → 2 → 4 → 8 → … → n, copies happen like:

```
1 + 2 + 4 + 8 + … + n/2 ≈ 2n
```

-
- Total work for all n insertions = O(n).
- Hence **amortized cost per insertion = O(1)**.