# UNIT 5

# Amortized Analysis

# Amortized Analysis

- Amortized analysis evaluates the overall efficiency of an algorithm across multiple operations.
- Instead of studying one operation in isolation, it considers the entire sequence of operations.
- It helps show that even if some steps are costly, the system remains efficient over time.

## Purpose

- Many data structures occasionally perform expensive work.
- These costly operations do not occur frequently, so their impact becomes negligible.
- Amortized analysis proves that the long-run cost per operation stays low.

## Core Idea

- Total cost of all operations is calculated first.
- Then this total is divided by the number of operations.

- This gives a stable and predictable cost per operation, even when individual steps vary.

## Example Scenario

- Consider a dynamic array that expands when it becomes full.
- The resizing step requires copying all elements, which is expensive.
- But since resizing happens rarely, the overall insertion cost remains O(1).
- This demonstrates why amortized analysis is needed.

## Methods of amortized analysis

## 1. Aggregate Method / Analysis

**What is Aggregate Analysis?**

- Aggregate analysis examines the total cost of a sequence of n operations.
- After finding the overall worst-case time, it divides this cost by n.
- The result gives a uniform amortized cost per operation.

**Basic Idea**

- If all n operations together take T(n) time,
- → then amortized cost = T(n) / n per operation.
- Useful when every operation seems constant time individually, but sequences can grow more expensive

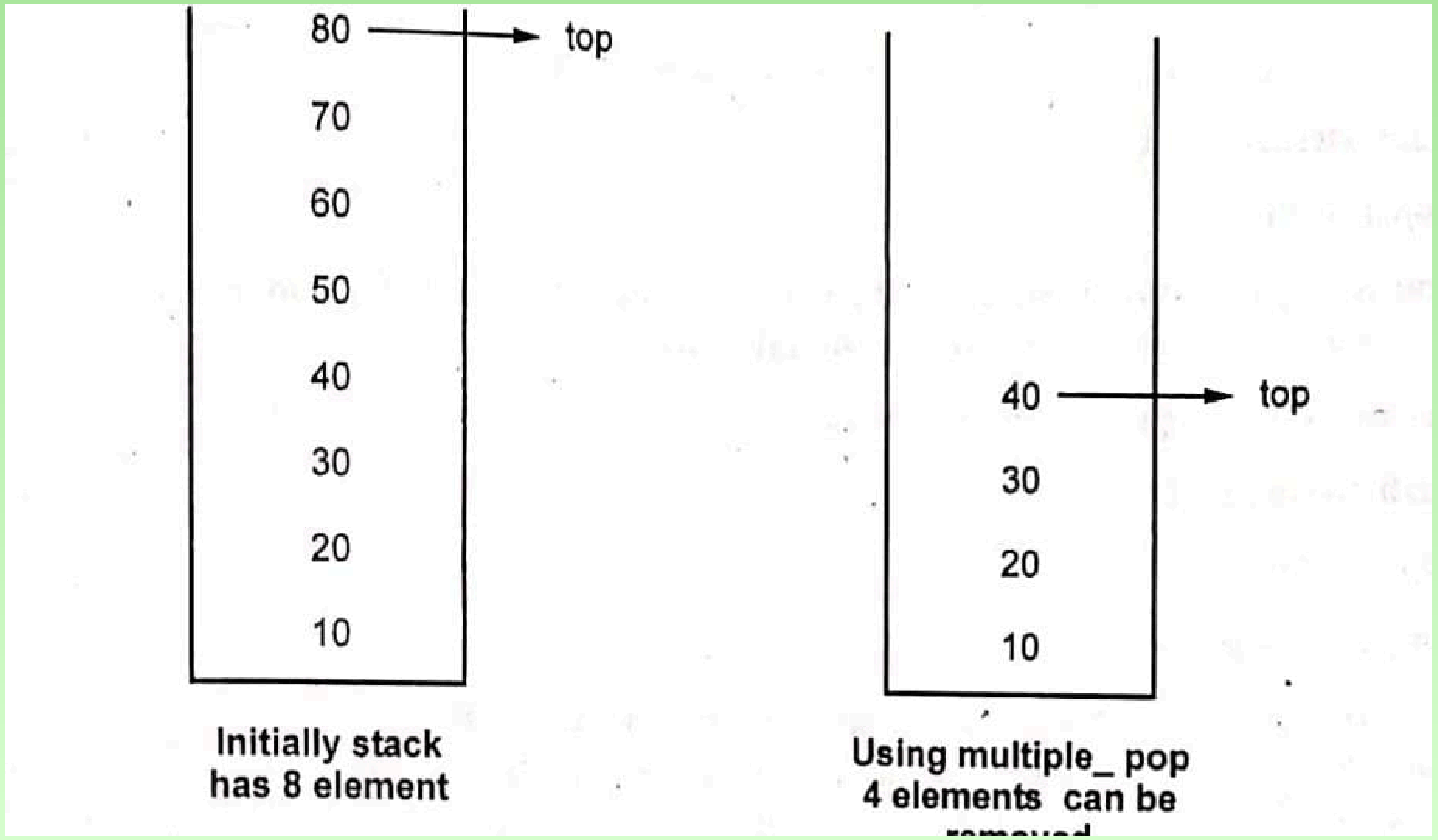
**Stack Example (Push & Pop)**

- push(item) stores the element and increases the top pointer.

- pop() decreases the top pointer and returns the element.
- Both operations independently run in O(1) time.
- Therefore, performing n push/pop operations takes O(n) total time.

## Multiple Pop Scenario

- A special operation multiple_pop(stack, k) removes up to k elements.
- However, it can pop only as many elements as the stack currently holds.
- So if the stack has n elements, at most n pops can occur.


80
top
70
60
50
40
30
20
10
Initially stack has 8 element
40
top
30
20
10
Using multiple_ pop 4 elements can be

## Observation

- Even if we call multiple_pop many times,
- → each element in the stack can be removed only once.
- So the total number of pops = total number of pushes.

## Cost Interpretation

- Although a single multiple_pop can take O(n),
- → the overall sequence of operations across the entire program never exceeds O(n) pops in total.
- This means the aggregate total is O(n), not O($n^2$).

## 2. Accounting Method

### Introduction

- The accounting method assigns a fixed charge (amortized cost) to each operation.
- This charge may be more or less than the operation's real cost.
- Extra charge gets saved as credit, which is used later to pay for expensive operations.

### Key Formula

Amortized Cost = Actual Cost + Credit (saved or spent)

- Credit is like stored money that helps pay for future costly steps.
- Ensures credit never becomes negative.
-

### Assigning Actual Costs (Stack Example)

- push(item) → actual cost = 1
- pop() → actual cost = 1
- multiple_pop(k) → actual cost = min(stack size, k)

## Assigning Amortized Costs

- We choose amortized costs that make analysis easier:
    - push(item) → 2
    - pop() → 0
    - multiple_pop() → 0
- Extra charge on push helps pay for future pops.

## How It Works

- When an item is pushed:
- → pay amortized cost 2
- → 1 pays for actual push, 1 is saved as credit on that item
- When popping:
- → actual cost is paid using the saved credit
- → No amortized cost needed

## Why It Works

- Every pushed item stores exactly enough credit to pay for its own pop later.

- Since each pop uses credit from a earlier push:
- → pops never need extra cost
- Same logic applies to multiple_pop() (it only pops existing items, each having credit).

- Total amortized cost of n operations = O(n)
- Average amortized cost per operation = O(n)/n = O(1)
- Even though some operations are heavy, the amortized performance stays constant.

## 3. Potential Method

### Introduction

- The potential method is another way to do amortized analysis.
- It works like “saving energy” inside the data structure for future expensive operations.
- This saved energy is called potential.

### Key Idea

- Every operation changes the “potential energy” of the structure.
- If potential increases → we are saving credit for later.
- If potential decreases → we are using saved credit to pay actual cost.
- This helps balance costly and cheap operations.

### How It Works

- Let the data structure change after each operation:
- → $D_0, D_1, D_2, \ldots, D_n$

- Let actual cost of ith operation = $c_i$
- Let $\Phi(D_i)$ = potential of structure after ith operation
- Amortized cost is defined as:

$$\text{Amortized cost } c_i' = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

**Conditions for Potential Function**

- $\Phi(D_0) = 0$ (initial potential starts at zero)
- $\Phi(D_i) \geq 0$ for all i (potential must never be negative)
- If $\Phi$ increases → we are overcharging (saving credit)
- If $\Phi$ decreases → we are undercharging (spending saved credit)

Stack Example – Choosing Potential

- For a stack, the natural potential is the number of elements in the stack.
- So potential = size of stack
- This keeps $\Phi$ always $\geq 0$

## Push Operation

- Actual cost = 1
- Stack size increases by 1 → potential increases by 1
- Amortized cost = 1 (actual) + 1 (potential change) = 2

## Pop Operation

- Actual cost = 1
- Stack size decreases by 1 → potential drops by 1
- Amortized cost = 1 + (−1) = 0

## Multiple Pop

- If k elements (or as many as available) are removed:
- → potential drops by k
- Actual cost = k
- Amortized cost = k + (−k) = 0

## Result

- Push = constant amortized cost
- Pop = constant amortized cost
- Multiple_pop = constant amortized cost
- Total cost for n operations = O(n)
- Therefore, amortized cost per operation = O(1)

## Tractable and Non-Tractable Problems

### Introduction

- Problems in computer science are classified based on how efficiently they can be solved.
- Two major categories: Tractable and Non-tractable problems.

### Tractable Problems

- A problem is tractable if it can be solved in polynomial time (e.g., $O(n)$, $O(n^2)$, $O(n^3)$).
- These problems are considered practically solvable, even for large inputs.
- Their running time grows at a manageable rate.

### Examples of Tractable Problems

- Searching in an unsorted list
- Sorting data
- Binary search in a sorted list

- Multiplying two integers
- Finding a minimum spanning tree (Kruskal / Prim)

**Non-tractable Problems**

- A problem is non-tractable (or intractable) if no known polynomial-time algorithm exists.
- These problems typically require exponential or super-polynomial time, making them very slow for large inputs.
- They are considered computationally hard.

**Examples of Non-Tractable Problems**

- Traveling Salesman Problem (TSP)
- 0/1 Knapsack Problem
- Many NP-complete and NP-hard problems

# Randomized Algorithms

## Introduction

- A randomized algorithm uses random numbers to make decisions during execution.
- Because of randomness, the behavior of the algorithm can change every time it runs.
- Even with the same input, it may follow different execution paths.

## Important Characteristics

- Execution time may vary each time the program runs.
- Output may differ depending on the randomness used.
- With the same input, the internal steps or results might not always be the same.

## Types of Randomized Algorithms

1. Las Vegas Algorithms
   - Always produce the correct output.
   - Randomness only affects the running time.
   - Example: Randomized Quick Sort (choosing pivot randomly).

## Monte Carlo Algorithms

- Running time is predictable, but result may be incorrect with a small probability.
- Randomness affects the quality or accuracy of the output.

## Advantages

- Easy to write and understand.
- Often perform faster or more efficiently than traditional deterministic algorithms.
- Help avoid worst-case behavior (e.g., Quick Sort becomes more stable).

## Disadvantages

- Small chance of error in some algorithms can be unsafe for critical systems.
- Randomness does not always guarantee better performance.
- Harder to predict exact running time for every execution.

# Approximation Algorithms

## Introduction

- Approximation algorithms are used for optimization problems where finding the exact best answer is very difficult.
- They provide a near-optimal solution instead of an exact one.

## Why Do We Need Approximation Algorithms?

- Many optimization problems (especially NP-hard problems) cannot be solved exactly in polynomial time.
- Exact algorithms exist, but are often too slow for large inputs.
- Approximation algorithms give fast and “good enough” solutions.

## Core Idea

- The goal is to find a solution that is close to the optimal, but much faster to compute.
- They rely on heuristics—simple rules or strategies based on experience.

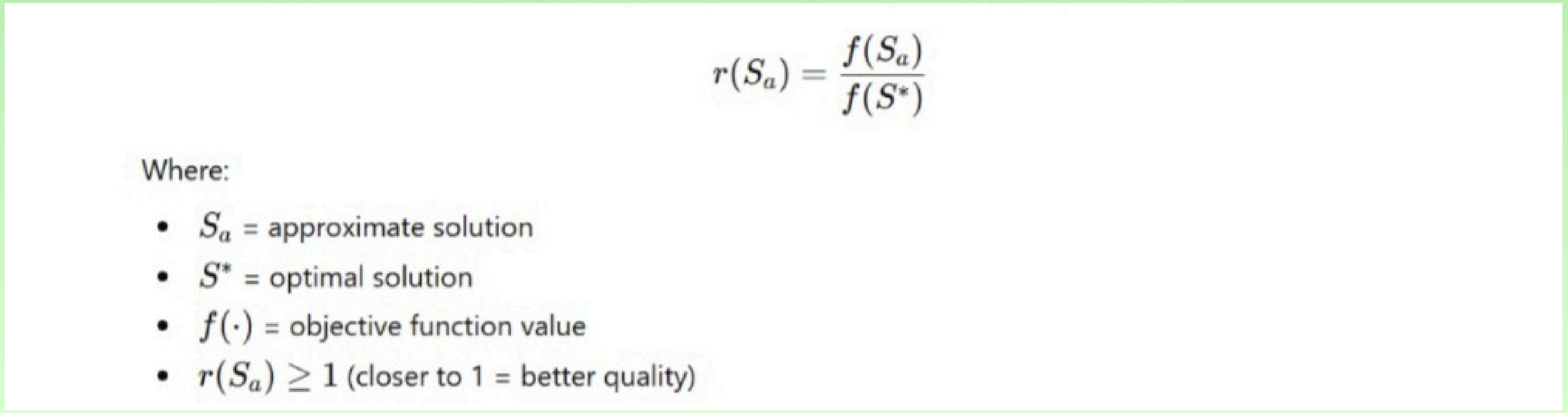
- Example heuristics:
    - TSP → visit the nearest unvisited city
    - Knapsack → pick items with highest value/least weight first

## How They Work

- Start with imperfect or partial data.
- Apply a heuristic strategy.
- Produce a solution that may not be perfect but is fast and acceptable.
- Often used when exact accuracy is less important than speed.

## Accuracy Ratio

- Used to measure how close the approximate solution is to the optimal solution.



$$r(S_a) = \frac{f(S_a)}{f(S^*)}$$

Where:

- $S_a$ = approximate solution
- $S^*$ = optimal solution
- $f(\cdot)$ = objective function value
- $r(S_a) \geq 1$ (closer to 1 = better quality)

## Performance Ratio

- Performance Ratio ($R_a$) is the worst-case upper bound of the accuracy ratio for the algorithm.
- It describes the overall quality of the approximation algorithm.
- If Ra$R_a$Ra is close to 1, the algorithm is considered very good.

# Embedded Algorithms

Embedded algorithms are specialized computational methods created for systems that run on dedicated hardware. These systems—such as smart home devices, wearables, medical sensors, automotive controllers, and industrial machines—must operate under strict constraints like low memory, limited processing speed, and minimal power usage.
Since embedded devices perform fixed, highly-specific operations, the algorithms must be optimized for:

- Fast response time
- Low energy consumption
- High reliability
- Consistent performance even under resource limits

**Power-Optimized Scheduling in Embedded Systems**

Scheduling in embedded systems determines when and how different tasks execute, especially when multiple operations must share the same limited hardware resources.

A power-optimized scheduling algorithm focuses on reducing overall energy usage without compromising deadlines or system performance.

**Why Is Power Optimization Important?**

- Many embedded devices run on batteries (e.g., IoT devices, sensors).
- Lower power consumption increases device lifespan.
- Reduced heat generation improves reliability and safety.

**Key Principles of Power-Optimized Scheduling**

- **Idle-Time Exploitation**

The processor is placed into low-power or sleep modes whenever tasks are not actively running. Efficient scheduling ensures idle gaps are long enough to save energy.

- **Adaptive Task Execution**

Tasks with flexible deadlines may be postponed or compressed to match the system's power availability, especially in solar-powered or energy-harvesting devices.

- **Energy-Aware Task Mapping**

Each task is assigned to the hardware component that consumes the least power (e.g., choosing between CPU cores, accelerators, or microcontrollers).

**Insertion Sort Algorithm in Embedded Systems**

Insertion Sort is frequently used in embedded systems because it meets the system’s strict design limitations.

**Why Insertion Sort Fits Embedded Systems**

- **Predictable Execution Time**

The algorithm behaves consistently for small datasets, making it easier to guarantee timing constraints in real-time embedded applications.

- **Minimal Instruction Overhead**

Insertion Sort uses simple comparisons and shifts, which align well with low-power processors that lack advanced arithmetic units.

## High Stability for Ordered Input

- Many embedded datasets arrive in nearly sorted form (e.g., sensor readings), making Insertion Sort extremely fast under such conditions.

## Easy Error Detection and Correction

- Because the algorithm handles data step-by-step, errors can be detected early and corrected with minimal processing—important for safety-critical devices like medical or automotive systems.

SHARE

subscribe

THANK YOU