DESIGN ANALYSIS AND ALGORITHMS

A Comprehensive Notes

AMORTIZED ANALYSIS

Amortized Analysis:

Amortized Analysis is a technique used in **algorithm analysis** to determine the **average time complexity** of an algorithm over a **sequence of operations**, rather than focusing on the time complexity of a single operation.

It is used when some operations may be **expensive** (take a long time), but most are **cheap** (take less time), and we want to determine the **average performance** across multiple operations.

Methods of amortized analysis

1. Aggregate Method.

The **Aggregate Method** is the simplest form of amortized analysis. It's used to compute the **amortized (average) cost per operation** by analyzing the **total cost of a sequence of operations** and **dividing it by the number of operations**.

🔑 **Key Idea:**

Instead of analyzing each operation individually, we:

1. Compute the **total actual cost** for **n operations**.
2. Divide the total cost by **n** to get the **amortized cost per operation**.

$$\textbf{Amortized cost per operation} = \frac{\textbf{Total cost of } n \textbf{ operations}}{n}$$

📌 **Example**: Dynamic Array Push – Amortized Analysis (Aggregate Method)

Suppose we have a dynamic array (like vector in C++ or ArrayList in Java) that:

- Doubles in size when it becomes full.
- Push/Insert normally takes O(1) time.

- But when full, it takes O(n) to resize and copy elements.

🔍 Let's Analyse **n** Operations

- Start with initial capacity = **1**.
- Resizes happen at sizes: 1 → 2 → 4 → 8 → ... up to n.

Each resize copies:

$$1 + 2 + 4 + 8 + \cdots + \frac{n}{2}$$

This is a geometric series, and it sums to: $\leq 2n$

👉 Total Cost of n Insertions

- n for n insertions
- $\leq$ 2n for resizes

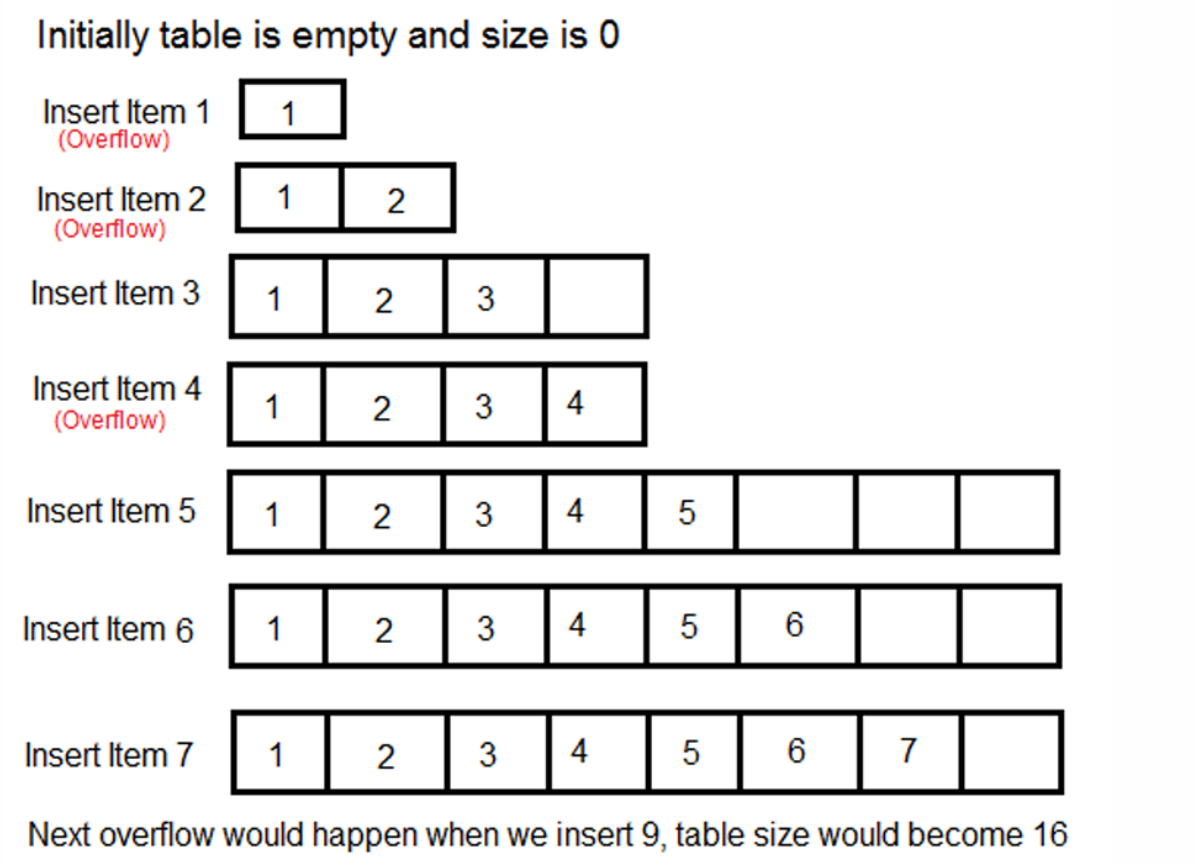
So, $Total\ Cost \leq 3n$

✅ Amortized Cost per Operation

$$\frac{3n}{n} = 3 = O(1)$$

Even with expensive resizing, the average (amortized) cost remains constant: **O(1)**.

📊 Step-by-Step Example (n = 8 insertions)



Insertion # 1 2 3 4 5 6 7 8

Table Size 1 2 4 4 8 8 8 8

Cost 1 2 3 1 5 1 1 1

➡️ Total Cost = 15
➡️ Amortized Cost = 15 / 8 = 1.875 ≈ **O(1)**

🟢 Conclusion

- Using the Aggregate Method, we show:
  - Amortized time complexity of dynamic table insertion = **O(1).**

- High-cost resizes are rare and spread across multiple operations.

2. Accounting Method

The **Accounting Method** is a way to analyze the average performance (amortized cost) of operations by **assigning a fixed "imaginary" cost (called amortized cost)** to each operation, possibly more than its actual cost.

The **extra cost (credit)** is **stored** and later used to **pay for more expensive operations**, like resizing.

This ensures that even though some operations are costly, the average cost per operation over a sequence remains **low.**

Example:

🧱 **Stack Supports:**

1. PUSH(x) → Insert element x → Cost: O(1)
2. POP() → Remove top element → Cost: O(1)
3. MULTIPOP(k) → Pop up to k elements → Cost: O(k)

Cost Assignment (Accounting Method):

Operation	Actual Cost	Amortized Cost	Explanation
PUSH(x)	1	2	1 for actual push, 1 saved as credit

Operation	Actual Cost	Amortized Cost	Explanation
POP()	1	0	Paid from saved credit at push
MULTIPOP(k)	k	0	All k elements were prepaid at push

Dynamic Table for Stack Operations

Operation	Action	Stack After Operation	Charge (Cost)	Credits on Elements	Table Size
PUSH(A)	Insert element A into the stack.	[A]	2 (1 for push, 1 saved)	A: [1]	1
PUSH(B)	Insert element B into the stack.	[A, B]	2 (1 for push, 1 saved)	A: [1], B: [1]	2
PUSH(C)	Insert element C into the stack.	[A, B, C]	2 (1 for push, 1 saved)	A: [1], B: [1], C: [1]	4 (Resize)
POP()	Remove the top element C.	[A, B]	0 (Covered by C's credit)	A: [1], B: [1]	4
MULTIPOP(1)	Remove up to 1 element. Since A has a pre-paid credit, no extra charge.	[B]	0 (Covered by A's credit)	B: [1]	4

Operation	Action	Stack After Operation	Charge (Cost)	Credits on Elements	Table Size
MULTIPOP(1)	Remove up to 1 element. Since B has a pre-paid credit, no extra charge.	[]	0 (Covered by B's credit)	[]	4

Conclusion:

Using the Accounting Method, the total cost for all operations is 6, and the amortized cost per operation is O(1).

Even with dynamic resizing and multiple operations like PUSH, POP, and MULTIPOP, the average cost per operation remains constant and efficient. The cost of resizing is effectively distributed across operations, ensuring that the amortized time complexity remains O(1).

Binary Heap:

A Binary Heap is a Binary Tree with following properties.

1. It's a **complete binary tree** i.e., all levels are completely filled except possibly the last level and the last level has all keys as left as possible. This property of Binary Heap makes them suitable to be stored in an array.
2. A Binary Heap is either **Min Heap** or **Max Heap**. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to Min Heap.

Min Heap:

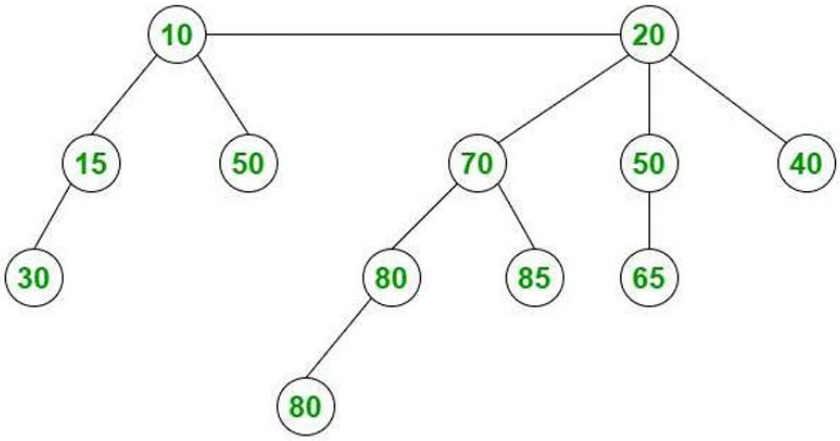
```
    2
   / \
  3   5
 / \
4   6
```

Max Heap:

```
    10
   /  \
  9    5
 / \
7   6
```

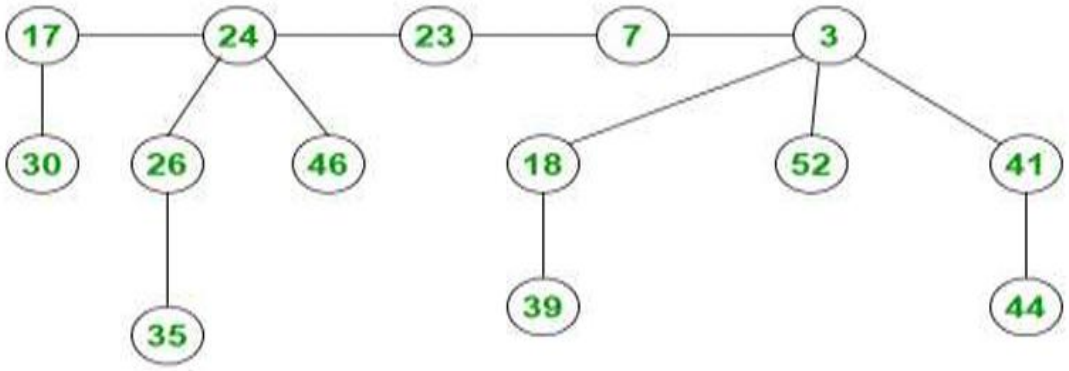
Binomial Heap:

A Binomial Heap is a collection of **Binomial Tree** where each Binomial Tree follows the **Min-Heap property.**



Fibonacci Heap:

Fibonacci Heap is a collection of trees with **Min-Heap** or **Max-Heap** property. In Fibonacci Heap, trees can have any shape even all trees can be single nodes (This is unlike Binomial Heap where every tree has to be a Binomial Tree). Fibonacci Heap maintains a pointer to a minimum value (which is the root of a tree).



Time-Space Trade-Off in Algorithms:

A tradeoff is a situation where one thing increases and another thing decreases. It is a way to solve a problem in:

- Either in less time and by using more space, or
- In very little space by spending a long amount of time.

The best Algorithm is that which helps to solve a problem that requires less space in memory and also takes less time to generate the output. But in general, it is not always possible to achieve both of these conditions at the same time.

Types of Space-Time trade-off:

1. Compressed or Uncompressed Data:

- **Compressed Data**: Storing data in a compressed format uses less memory, but it requires **extra time** for compression and decompression operations. For example, when reading a

compressed file, the program must first decompress it before using the data, leading to slower performance in terms of access time.

- **Uncompressed Data**: Storing data uncompressed uses more memory but allows for **faster access** since no compression or decompression is required. This is beneficial in cases where data access speed is more critical than memory usage, such as in real-time systems.

Trade-off:

- If you want to minimize memory usage, choose compressed data but at the cost of processing time for compression/decompression.
- If you prioritize speed, opt for uncompressed data, which uses more memory but is faster to access.

2. Re-rendering or Stored Images:

- **Re-rendering**: Instead of storing the image in memory, it can be **re-rendered** (or regenerated) when needed. Re-rendering typically involves recalculating the visual representation, which requires time, but **doesn't consume memory** for storing the image.
- **Stored Images**: Storing pre-generated images consumes memory but allows for **instant retrieval** and display. However, this increases memory usage since the image must be stored in memory or disk.

3. Smaller Code or Loop Unrolling:

- **Smaller Code**: Writing more compact code or using abstract constructs can reduce the **size of the program** in terms of memory usage. However, it may require more **computational steps** and hence take longer to execute.
- **Loop Unrolling**: Loop unrolling is a technique where loops are expanded to reduce the overhead of loop control. This can **speed up execution** by reducing the number of iterations, but it leads to

larger code and can increase memory usage, especially in cases of large loops.

4. **Lookup Tables or Recalculation:**

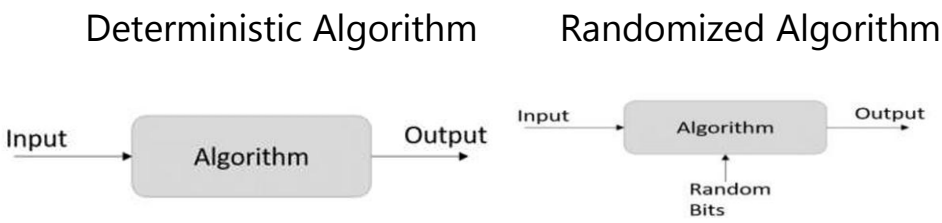
- **Lookup Tables**: A **lookup table** stores precomputed values so that repeated calculations can be avoided by simply looking up the result. This speeds up execution time but uses additional **memory** to store the table.
- **Recalculation**: Instead of storing precomputed values, the program **calculates the result** each time it's needed. This avoids the memory overhead but may result in **slower execution** due to repeated calculations.

Randomized algorithms:

Randomized algorithms are such algorithms that make random choices (coin-tossing) during their executions. As a consequence, their outcomes do not depend only on their inputs.

advantages of **randomized algorithms**:

1. **Faster on Average**: They often provide better **average-case performance** compared to deterministic algorithms.
2. **Simpler to Implement**: Some problems can be solved more easily with randomized algorithms, requiring less complex code.
3. **Probabilistic Guarantees**: They offer **probabilistic correctness** or performance, with small chances of error.



Examples: Randomized Quick Sort Algorithm, The Subset Sum Problem

Game Theory and Randomized Algorithms

Game Theory is a mathematical framework that studies the interactions between different agents (players) in strategic situations where the outcome depends not only on their decisions but also on the decisions of others.

Zero-Sum Game in Game Theory

A **Zero-Sum Game** is a special type of game where the total amount of utility (or "gain" and "loss") is fixed, and each player's gain or loss is perfectly balanced by the losses or gains of the other players. In other words, the sum of the winnings and losses across all players equals zero at all times.

Approximation Algorithms

Approximation algorithms are algorithms designed to provide approximate solutions to optimization problems that are **NP-hard** or otherwise intractable to solve exactly in polynomial time. These problems typically do not have efficient algorithms that guarantee an optimal solution within a reasonable time frame, so approximation algorithms aim to produce a solution that is close to optimal within a **guaranteed bound**.

Greedy Algorithms

Greedy algorithms make **locally optimal choices** at each step, with the hope of arriving at a **globally optimal solution**. They work by iteratively selecting the best option at each stage without considering the broader problem context, aiming to improve the current solution with each step.

Examples:

• Knapsack Problem: Select items with the highest value-to-weight ratio until the knapsack is full.

• Minimum Spanning Tree (MST): Add edges to a graph in increasing order of weight, avoiding cycles.

Travelling Salesperson Approximation Algorithm:

The travelling salesperson approximation algorithm requires some prerequisite algorithms to be performed so we can achieve a near optimal solution.

- **Minimum Spanning Tree** − The minimum spanning tree is a tree data structure that contains all the vertices of main graph with minimum number of edges connecting them. We apply prim's algorithm for minimum spanning tree in this case.
- **Pre-order Traversal** − The pre-order traversal is done on tree data structures where a pointer is walked through all the nodes of the tree in a [root – left child – right child] order.

Algorithm:

1. Choose Starting Point:
   Select any vertex from the graph as the starting and ending point (root node for the MST).
2. Construct Minimum Spanning Tree:
   Use Prim's Algorithm to construct the Minimum Spanning Tree (MST) with the selected root node.
3. Pre-order Traversal:
   Perform a Pre-order Traversal on the MST to obtain a sequence of vertices representing a potential TSP tour.
4. Construct Hamiltonian Path:
   The pre-order traversal will give a sequence of vertices. Remove any repeated nodes and form a valid Hamiltonian path.

Pseudocode for APPROX_TSP:

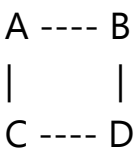
```
APPROX_TSP(G, c):
    r ← root node of the minimum spanning tree

    # Step 2: Construct (MST) using Prim's algorithm
    T ← MST_Prim(G, c, r)

    # Step 3: Initialize an empty set to track visited nodes
    visited ← {}

    # Step 4: Traverse all nodes (V is the number of vertices)
    for i in range(0, V):
        # Step 4.1: Perform a Pre-order traversal of the MST
        H ← Preorder_Traversal(T, r)

        # Step 4.2: Mark nodes as visited during traversal
        for node in H:
            if node not in visited:
                visited.add(node)
                # Append to the path/tour
                tour.append(node)

    # Step 5: Complete the cycle by adding the start node at the end
    tour.append(tour[0])

    return tour
```

Approximation Algorithm for Vertex Cover (Greedy Approach):

Steps:

1. **Select an uncovered edge (u, v)** from the graph.
2. **Add both vertices (u and v)** to the vertex cover set.
3. **Remove all edges incident on u or v** (since they are now covered by the newly added vertices).
4. **Repeat** the process until all edges are covered.

Example:



Step-by-step Process:

1. **Select an uncovered edge (A, B)**. Add vertices **A** and **B** to the vertex cover set.
    - Vertex cover = {A, B}
2. **Remove edges incident on A and B**:
    - Remaining edges: (C, D)
3. **Select an uncovered edge (C, D)**. Add vertices **C** and **D** to the vertex cover set.
    - Vertex cover = {A, B, C, D}
4. **All edges are now covered**. The algorithm stops.

Final Approximate Vertex Cover: {A, B, C, D}

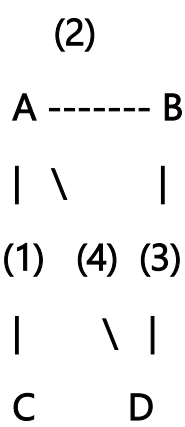
Monte Carlo VS Las Vegas

Property	Las Vegas Algorithm	Monte Carlo Algorithm
Correctness	Always correct	May be incorrect (with small error probability)
Running Time	Randomized (depends on input and randomness)	Fixed (deterministic time complexity)
Example	Randomized QuickSort	Miller-Rabin Primality Test

Dijkstra’s Algorithm

Dijkstra's algorithm finds the **shortest path** from a **source vertex** to **all other vertices** in a **weighted, non-negative graph**.

Example



Vertex	Distance from A	Previous
A	0	-
B	Inf	-

Vertex	Distance from A	Previous
C	Inf	-
D	Inf	-

- Start at A → distance = 0
- Update neighbors:
  - B = min(°, 0 + 2) = 2
  - C = min(°, 0 + 1) = 1
  - D = min(°, 0 + 4) = 4

- Pick vertex with smallest dist = C
  - D = min(4, 1 + 2) = 3
- Pick B (distance = 2)
  - D = min(3, 2 + 3) = 3 (already 3, no change)
  - Pick D (done)

📌 **Final Shortest Distances from A:**

- A → A = 0
- A → B = 2
- A → C = 1
- A → D = 3

✅ **Time Complexity:** O(V^2)

Embedded Algorithms

Embedded Algorithms are specialized algorithms designed to run on **embedded systems**, which are devices with limited processing power, memory, and storage. These algorithms are optimized to perform efficiently in such resource-constrained environments while meeting specific tasks like control, processing sensor data, or running real-time operations. Examples of embedded algorithms include control systems (like PID), signal processing, data compression, and encryption algorithms. They are commonly used in devices like smartphones, robots, cars, and IoT (Internet of Things) devices.

Example Applications:

1. **Robotics**: Embedded algorithms like PID controllers are used for real-time control of robotic arms or autonomous vehicles.
2. **Automotive**: Embedded algorithms are used in control systems, safety features, and navigation systems in vehicles.
3. **IoT Devices**: Low-level embedded algorithms are used to process sensor data and make real-time decisions, such as temperature regulation or motion detection.
4. **Consumer Electronics**: Algorithms for signal processing, like audio compression or image processing, are embedded in devices like cameras or smart speakers.