**1️⃣ Hash Tables & Hashing**

Hash tables are fundamental data structures that allow efficient storage and retrieval of key-value pairs. They use a **hash function** to map keys to indices in an array, enabling near-instant access to data. Hash tables support operations like insertion, deletion, and lookup with an **average time complexity of O(1)**. However, collisions can occur when two keys map to the same index, requiring collision resolution techniques such as **chaining** or **open addressing**. Hashing is widely used in databases, caching, and indexing, making it a crucial concept in computer science.

**🔹 Basics of Hash Tables**

A **hash table** (or **hash map**) is a data structure that associates **keys** with **values** using a **hash function**. The function converts a key into an index where the value is stored, allowing for **fast lookups, insertions, and deletions**. Unlike arrays, which require **O(n)** time for searching, hash tables can retrieve values in **O(1) time** on average. However, hash tables are prone to **collisions**, where multiple keys hash to the same index, requiring resolution strategies such as **separate chaining** or **open addressing**.

**🔹 Types of Hash Functions**

A **hash function** takes an input (key) and produces a numeric value (hash code), which determines the index in the hash table. A good hash function should be **deterministic, uniform, and fast** while minimizing collisions. Common types include:

1. **Division Method**: Uses hash(key) = key % table\_size.
2. **Multiplication Method**: Uses a fraction of key \* constant.
3. **Folding Method**: Breaks key into parts, sums them, and hashes.
4. **Mid-Square Method**: Squares the key and extracts middle digits.  
   Choosing the right hash function impacts performance, affecting **collisions, speed, and memory usage**.

**🔹 Applications of Hash Tables**

Hash tables are used in various real-world applications due to their speed and efficiency. They are widely used in **databases** for indexing, **caching systems** for quick lookups, and **symbol tables** in compilers for variable storage. Other applications include **password hashing**, where cryptographic hash functions ensure security, and **routing tables** in networking. Search engines use hashing to store and retrieve web pages, while spell checkers and dictionary applications use hash tables to store words for **quick search and auto-complete features**.

**🔹 Hash Collision and Resolution Methods**

A **collision** occurs when two different keys produce the same hash value, mapping to the same index. To handle collisions, two main strategies are used:

1. **Chaining (Separate Chaining)**: Uses linked lists or dynamic arrays to store multiple values at the same index.
2. **Open Addressing**: Finds another open slot in the table for the new value using probing techniques like **linear probing, quadratic probing, and double hashing**.  
   Effective collision resolution is crucial to maintaining the **O(1) time complexity** of hash tables and preventing **performance degradation**.

**🔹 Chaining (Separate Chaining)**

**Separate chaining** stores multiple values at the same index using a **linked list, array, or balanced tree**. When a collision occurs, the new key-value pair is added to a list at the corresponding index. This method is simple to implement and does not require **rehashing** or **probing**, but it may increase memory usage and affect performance if chains become too long. To optimize, **self-balancing BSTs (like AVL trees)** can be used instead of linked lists, ensuring faster retrievals.

**🔹 Open Addressing**

Unlike chaining, **open addressing** does not use additional data structures. Instead, it finds an **empty slot within the table** using a probing sequence. When a collision occurs, a new index is determined by methods such as **linear probing, quadratic probing, or double hashing**. Open addressing minimizes memory overhead but may suffer from **primary clustering** (when many keys crowd the same region). It requires a **low load factor** to maintain efficient lookups and prevent excessive probing.

**🔹 Linear Probing vs. Quadratic Probing**

* **Linear Probing**: When a collision occurs, the next available slot is checked sequentially (index = (hash + i) % table\_size). While simple, it leads to **primary clustering**, slowing operations.
* **Quadratic Probing**: Instead of checking the next slot linearly, a quadratic function is applied (index = (hash + i²) % table\_size). This reduces clustering but requires a **prime-sized table** to avoid infinite loops. Both techniques require careful table management to ensure efficient **collision handling**.

**🔹 Double Hashing**

**Double hashing** is a collision resolution technique that uses a **secondary hash function** to determine the step size when a collision occurs. Instead of linear or quadratic probing, the new index is calculated as:

index=(hash1(key)+i×hash2(key))%table\_sizeindex = (hash1(key) + i \times hash2(key)) \% table\\_size

Double hashing significantly reduces clustering and distributes values more uniformly, improving search efficiency. However, it requires **two hash functions** and additional computation, making it slightly slower than **linear probing** for small tables.

**🔹 Load Factor**

The **load factor** represents how full a hash table is. It is calculated as:

LoadFactor=Number of ElementsTable SizeLoad Factor = \frac{Number\ of\ Elements}{Table\ Size}

A **high load factor (>0.7)** increases collisions, slowing down operations, while a **low load factor (<0.5)** wastes memory. When the load factor exceeds a threshold, **rehashing** is performed to resize the table and maintain efficiency.

**🔹 Rehashing**

**Rehashing** increases the hash table size (usually doubling it) when the load factor becomes too high. During rehashing, all existing elements are **recomputed and placed into a new table** using a new hash function. This prevents excessive collisions and maintains **fast access times**. However, rehashing is an **expensive operation (O(n))**, so it is only performed when necessary.

**🔹 Hash Table vs. HashSet**

* **Hash Table** stores **key-value pairs**, supporting fast lookups, insertions, and deletions.
* **HashSet** stores **unique keys only**, making it useful for membership checks and duplicate prevention.  
  Both structures use hashing internally but differ in functionality and use cases.

**🔹 Hash Table Time Complexity**

| **Operation** | **Average Case** | **Worst Case** |
| --- | --- | --- |
| Insert | O(1) | O(n) |
| Delete | O(1) | O(n) |
| Search | O(1) | O(n) |

In the worst case (high collisions or poor hash function), operations degrade to **O(n)**, similar to linked lists.

**🔹 Hashing vs. Encryption**

* **Hashing** is a one-way process used for **fast lookup** and **data integrity** (e.g., SHA-256).
* **Encryption** is a **two-way** process that requires a key for decryption (e.g., AES).  
  Hashing is **irreversible**, while encryption can be decrypted with a key.

**🔹 Popular Hashing Algorithms**

* **SHA-1 (Secure Hash Algorithm 1)**: Produces a **160-bit hash** but is now considered weak.
* **MD5 (Message Digest Algorithm 5)**: Produces a **128-bit hash** but suffers from **collision attacks**.
* **CRC32 (Cyclic Redundancy Check)**: A simple **error-detection algorithm** used in **file integrity verification**.

**2️⃣ Stacks**

A **stack** is a linear data structure that follows the **Last-In-First-Out (LIFO)** principle, meaning the last element added is the first one removed. Stacks are widely used in **recursion, expression evaluation, and memory management**. The primary operations in a stack include:

* **Push()** – Adds an element to the top.
* **Pop()** – Removes and returns the top element.
* **Peek()** – Retrieves the top element without removing it.
* **isEmpty()** – Checks if the stack is empty.

Stacks can be implemented using **arrays** (fixed size) or **linked lists** (dynamic size). They are commonly used in **function calls, undo-redo operations, and balancing parentheses** in expressions.

**🔹 Basics of Stacks**

A stack is a **restricted** data structure where elements are inserted and removed from only **one end (the top)**. It follows the **LIFO (Last-In-First-Out)** order, making it useful for **backtracking and recursion**. The main stack operations are:

* **Push(x)**: Inserts an element at the top.
* **Pop()**: Removes and returns the top element.
* **Peek()**: Returns the top element without removing it.
* **Size()**: Returns the number of elements in the stack.

Stacks can be implemented using **arrays (static size)** or **linked lists (dynamic size)**, with linked lists being more flexible but requiring extra memory.

**🔹 Stack Using Linked List**

A **linked list-based stack** dynamically grows and shrinks as needed, unlike an array-based stack with a fixed size. It consists of **nodes**, where each node contains:

1. **Data** (the value stored in the stack).
2. **Pointer** to the next node.

The **top** of the stack points to the most recently added node. Operations are performed as follows:

* **Push()**: Create a new node and set its next pointer to the current top.
* **Pop()**: Remove the top node and update the top pointer.
* **Peek()**: Return top->data.

This implementation avoids **stack overflow** but requires **extra memory for pointers**.

**🔹 Applications of Stack**

Stacks have multiple real-world applications, including:  
✔ **Function calls and recursion** (maintaining execution order).  
✔ **Expression evaluation** (postfix and infix conversion).  
✔ **Undo-redo operations** in text editors.  
✔ **Backtracking algorithms** (solving mazes, DFS traversal).  
✔ **Parentheses matching** in code compilation.  
✔ **Memory management** (stack-based allocation).  
✔ **Reversing a string** (pushing characters and popping in reverse).

Understanding stacks is crucial for solving problems involving **order-dependent operations**.

**🔹 Stack Underflow & Overflow**

* **Stack Overflow** occurs when attempting to push an element into a full stack (in fixed-size implementations like arrays).
* **Stack Underflow** occurs when trying to pop an element from an empty stack.

For **linked-list-based stacks**, overflow is rare but depends on system memory. Proper **error handling** prevents these conditions.

**🔹 Parenthesis Checking using a Stack**

Balanced parentheses are essential in programming. A stack helps check expressions like {[()()]}:

1. **Push** each opening bracket ((, {, [).
2. **Pop** when encountering a closing bracket (), }, ]) and check for a match.
3. If the stack is empty at the end, the expression is **balanced**. Otherwise, it's **unbalanced**.

This is useful in **parsing expressions, compilers, and syntax checkers**.

**🔹 Stack Used in Undo-Redo Operations**

Text editors use two stacks for **undo-redo functionality**:

* **Undo stack** stores previous states (pushing each change).
* **Redo stack** stores undone states (when an undo is performed).

When a redo operation is triggered, elements are **pushed back to the undo stack**, allowing seamless **state restoration**.

**🔹 Convert Stack into a Queue**

A stack can be converted into a **FIFO queue** using two stacks:

1. **Push elements into Stack1** (acting as an enqueue operation).
2. **Transfer elements to Stack2** when dequeuing (reversing order).

Using this approach, we can efficiently implement **queue operations** (enqueue and dequeue) using **stacks**.

**🔹 Implement a Stack with Push, Pop, and getMax() in O(1)**

To retrieve the maximum element in **O(1)**, maintain a **secondary stack** that stores the maximum value at each step:

* **Push()**: Insert an element into the main stack. If it's greater than the current max, push it onto the max stack.
* **Pop()**: Remove the element from the main stack, and if it matches the max stack, pop from both.
* **getMax()**: Return the top of the max stack in O(1).

This ensures efficient **maximum retrieval** while maintaining standard stack operations.

**🔹 Remove Middle Element from a Stack**

Using recursion, the middle element can be removed **without using extra space**:

1. **Recursively pop** elements until the middle is reached.
2. **Skip the middle element** and push remaining elements back.

This method maintains **stack order** while eliminating the middle element.

**🔹 Sort a Stack**

A stack can be sorted using recursion:

1. **Pop the top element** and recursively sort the rest.
2. **Insert the popped element** back in the correct position.

This method sorts the stack in **O(n²) time** but preserves **stack properties**.

**🔹 Palindrome Checking using Stack**

A stack can verify palindromes (e.g., "madam"):

1. **Push the first half** of the string into a stack.
2. **Compare the second half** with popped elements.
3. If all match, the string is a palindrome.

This method ensures **efficient palindrome checking** in **O(n) time**.

**🔹 Reverse a String using Stack**

To reverse a string using a stack:

1. **Push characters** onto a stack.
2. **Pop characters** one by one to rebuild the reversed string.

This approach leverages the **LIFO property** for efficient **string reversal**.

**🔹 Valid Parentheses Problem**

A common **Leetcode** problem requiring stack-based validation of expressions like ({[]}):

1. **Push opening brackets** onto a stack.
2. **Pop and check matching pairs** for closing brackets.
3. If valid, return **true**, otherwise return **false**.

This method ensures **efficient syntax validation** in **O(n) time**.

**🔹 Stack Implementation using Queue**

A stack can be implemented using **two queues**:

1. **Push()**: Enqueue an element into **Queue1**.
2. **Pop()**: Transfer all but the last element to **Queue2**, dequeue the last, then swap the queues.

This maintains stack behavior while using **queue operations**.

**🔹 Stack that Rejects Duplicate Values**

A **unique stack** can be implemented using a **set**:

1. **Before pushing, check if the value exists** in the set.
2. **If not present, insert into both stack and set**.
3. **When popping, remove from the set as well**.

This ensures only **distinct elements** are stored.

**🔹 Purpose of Stack Pointer**

The **stack pointer** keeps track of the **top of the stack** in memory. It:  
✔ Points to the latest added element.  
✔ Adjusts during push/pop operations.  
✔ Prevents memory corruption.

It plays a critical role in **function calls, recursion, and CPU register management**.

**3️⃣ Queues**

A **queue** is a linear data structure that follows the **First-In-First-Out (FIFO)** principle, meaning the first element added is the first one removed. Queues are widely used in **task scheduling, process management, and breadth-first search (BFS) algorithms**.

The primary queue operations include:

* **Enqueue(x)** – Inserts an element at the rear.
* **Dequeue()** – Removes and returns the front element.
* **Front()** – Retrieves the front element without removing it.
* **Rear()** – Retrieves the last element.
* **isEmpty()** – Checks if the queue is empty.

Queues can be implemented using **arrays** (fixed size) or **linked lists** (dynamic size). They are commonly used in **CPU scheduling, printer job queues, and network traffic management**.

**🔹 Basics of Queues**

A queue is a **restricted** data structure where elements are added at one end (rear) and removed from the other end (front), ensuring **FIFO (First-In-First-Out)** order.

Queue operations:  
✔ **Enqueue(x)** – Add an element at the rear.  
✔ **Dequeue()** – Remove an element from the front.  
✔ **Front()** – Retrieve the front element.  
✔ **Rear()** – Retrieve the last element.  
✔ **Size()** – Return the number of elements in the queue.

Queues can be implemented using **arrays** (fixed size) or **linked lists** (dynamic size), with linked lists offering more flexibility but requiring extra memory for pointers.

**🔹 Applications of Queue**

Queues have various real-world applications, including:  
✔ **CPU Scheduling** – Managing processes in operating systems.  
✔ **BFS (Breadth-First Search)** – Exploring graph nodes layer by layer.  
✔ **Print Queue** – Managing multiple print jobs.  
✔ **Network Data Packets** – Handling incoming and outgoing data packets.  
✔ **Task Scheduling** – Managing background tasks.  
✔ **Customer Service Systems** – Processing requests in order.

Queues are essential in scenarios where tasks must be **processed in order**, making them integral to **operating systems and real-time computing**.

**🔹 Types of Queues**

There are different types of queues based on variations in insertion and deletion rules:

1. **Simple Queue** – The basic FIFO queue.
2. **Circular Queue** – A queue where the last position connects to the first to reuse memory.
3. **Double-Ended Queue (Deque)** – Insertions and deletions allowed from both ends.
4. **Priority Queue** – Elements are dequeued based on priority rather than order.
5. **Bounded Queue** – A queue with a fixed maximum size limit.

Understanding these variations is crucial for optimizing **memory usage, priority-based scheduling, and multi-tasking**.

**🔹 Circular Queue Implementation**

A **circular queue** is an advanced queue where the last position is connected to the first, making it a **closed loop**. This eliminates unused space in array-based queues.

Key points:  
✔ **Enqueue at the rear** and **dequeue from the front**.  
✔ **If the rear reaches the end**, wrap around to the front.  
✔ **Check for full queue** when front == (rear + 1) % size.

Circular queues improve **memory efficiency** and are widely used in **buffer management, streaming services, and memory scheduling**.

**🔹 Double-Ended Queue (Deque)**

A **deque** (double-ended queue) allows insertions and deletions from **both ends**. There are two types:

1. **Input-restricted deque** – Insertions allowed at one end, deletions at both.
2. **Output-restricted deque** – Deletions allowed at one end, insertions at both.

Operations:  
✔ **insertFront(x)** – Insert at the front.  
✔ **insertRear(x)** – Insert at the rear.  
✔ **deleteFront()** – Remove from the front.  
✔ **deleteRear()** – Remove from the rear.

Deques are useful in **sliding window algorithms, palindromes, and LRU (Least Recently Used) cache implementation**.

**🔹 Bounded Queue**

A **bounded queue** is a queue with a **fixed maximum size limit**, meaning it cannot exceed a predefined number of elements.

Characteristics:  
✔ **Enqueue operations fail** when the queue is full.  
✔ **Dequeue operations remove** elements as usual.  
✔ Used in **memory-constrained environments** where resources must be managed efficiently.

Bounded queues are commonly used in **rate-limiting APIs, thread pools, and controlled request processing**.

**🔹 Priority Queue**

A **priority queue** is an advanced data structure where elements are dequeued based on **priority** rather than order.

Key Features:  
✔ **Higher-priority elements are dequeued first**, regardless of insertion order.  
✔ Can be implemented using **heaps** for efficient O(log n) operations.  
✔ Used in **Dijkstra’s algorithm, CPU scheduling, and event-driven simulations**.

Priority queues optimize **time-sensitive** operations by ensuring important tasks are processed first.

**🔹 Implement Double-Ended Queue using Linked List**

A deque can be implemented using a **doubly linked list**, where each node has:

1. **Data (value stored)**.
2. **Pointer to next node**.
3. **Pointer to previous node**.

Operations:  
✔ **insertFront(x)** – Add node at the front.  
✔ **insertRear(x)** – Add node at the rear.  
✔ **deleteFront()** – Remove front node.  
✔ **deleteRear()** – Remove rear node.

Using linked lists ensures **dynamic memory allocation**, preventing overflow issues.

**🔹 Enqueue, Dequeue, Display Operations**

✔ **Enqueue(x)** – Add an element to the rear.  
✔ **Dequeue()** – Remove an element from the front.  
✔ **Display()** – Print all elements from front to rear.

These operations define the basic functionality of **queues**, whether implemented using **arrays, linked lists, or circular structures**.

**4️⃣ Sorting Algorithms**

Sorting is a fundamental operation in computer science used for **organizing data**, improving **search efficiency**, and optimizing **data retrieval**. Sorting algorithms can be broadly classified as **comparison-based** (Quick Sort, Merge Sort) and **non-comparison-based** (Counting Sort, Radix Sort). Sorting plays a crucial role in **database indexing, numerical computations, and algorithm optimizations**.

Each algorithm has different **time complexities** and is suited for specific scenarios. The choice of sorting algorithm depends on **data size, memory constraints, and required stability**.

**🔹 Time Complexity of Sorting Algorithms**

Sorting algorithms have different **best, average, and worst-case** time complexities:  
✔ **Bubble Sort** – O(n²) worst-case, O(n) best-case (already sorted).  
✔ **Insertion Sort** – O(n²) worst-case, O(n) best-case.  
✔ **Selection Sort** – O(n²) worst-case, O(n²) best-case.  
✔ **Merge Sort** – O(n log n) in all cases.  
✔ **Quick Sort** – O(n log n) average, O(n²) worst-case.

Understanding complexity helps in choosing an **efficient algorithm** based on **data size and constraints**.

**🔹 Insertion Sort**

Insertion Sort works by building a **sorted subarray** one element at a time, inserting each new element in the correct position.

✔ Best for **small datasets or nearly sorted data**.  
✔ **Time Complexity** – O(n²) worst-case, O(n) best-case.  
✔ **Stable** – Maintains the order of equal elements.

Used in **online algorithms**, where data arrives in real-time and needs **immediate sorting**.

**🔹 Selection Sort**

Selection Sort repeatedly **selects the smallest element** and swaps it to its correct position.

✔ **Always O(n²) time complexity**.  
✔ **Not stable** – Relative order of equal elements may change.  
✔ **In-place sorting** – Requires constant extra space (O(1)).

Simple but inefficient for large datasets due to **quadratic complexity**.

**🔹 Merge Sort**

Merge Sort follows the **Divide and Conquer** approach by:

1. **Dividing** the array into two halves recursively.
2. **Sorting** each half.
3. **Merging** them back in sorted order.

✔ **Time Complexity** – O(n log n) in all cases.  
✔ **Stable Sorting Algorithm**.  
✔ Requires **O(n) extra space**, making it inefficient for memory-constrained systems.

Used in **external sorting (large files), linked lists, and parallel processing**.

**🔹 Quick Sort**

Quick Sort is an **efficient** sorting algorithm using **Divide and Conquer**:

1. Select a **pivot**.
2. **Partition** the array into elements smaller and larger than the pivot.
3. **Recursively sort** the partitions.

✔ **Time Complexity** – O(n log n) average, O(n²) worst-case.  
✔ **In-place sorting**, requiring O(log n) extra space.  
✔ **Highly efficient for large datasets** but suffers in worst-case scenarios.

Optimized by **choosing the right pivot**, making it one of the **fastest sorting algorithms**.

**🔹 Bubble Sort**

Bubble Sort repeatedly swaps adjacent elements if they are in the wrong order.

✔ **Time Complexity** – O(n²) worst-case, O(n) best-case (already sorted).  
✔ **Stable Sorting Algorithm**.  
✔ **Inefficient for large datasets** but useful for teaching sorting concepts.

Rarely used in real-world applications due to its **slowness**.

**🔹 Stable Sorting Algorithms**

A sorting algorithm is **stable** if **equal elements remain in their original order**.

✔ **Stable algorithms** – Merge Sort, Insertion Sort, Bubble Sort.  
✔ **Unstable algorithms** – Quick Sort, Selection Sort.

Stable sorting is essential for **sorting records with multiple criteria**, such as **sorting by name while maintaining order by age**.

**🔹 In-Place Sorting**

A sorting algorithm is **in-place** if it uses **constant extra space (O(1))**.

✔ **In-Place Sorting** – Quick Sort, Selection Sort, Insertion Sort.  
✔ **Not In-Place** – Merge Sort (O(n) extra space).

In-place sorting is crucial when **memory is limited**.

**🔹 Quick Sort vs. Merge Sort**

✔ **Quick Sort** – Faster in practice, works well with in-place sorting.  
✔ **Merge Sort** – Consistent O(n log n) complexity but requires extra space.

Quick Sort is **better for general-purpose sorting**, while Merge Sort is preferred for **linked lists and large external data**.

**🔹 Worst-Case Complexity of Quick Sort**

Quick Sort’s worst-case O(n²) occurs when:  
✔ The pivot is **always the smallest or largest element**.  
✔ The array is **already sorted or reverse sorted**.

To avoid this, **randomized pivot selection** or **median-of-three method** is used.

**🔹 Disadvantage of Quick Sort over Merge Sort**

✔ **Quick Sort** has a **worst-case O(n²)**, while Merge Sort always runs in O(n log n).  
✔ **Quick Sort is not stable**, while Merge Sort is.  
✔ Merge Sort is **better for linked lists**, as it doesn’t require random access.

**🔹 Why Merge Sort is Preferred for Linked Lists**

Merge Sort works better for **linked lists** because:  
✔ **Linked lists don’t support random access**, making Quick Sort inefficient.  
✔ **Merge Sort doesn’t require extra swaps**, reducing time complexity.  
✔ **Efficient merging of sorted lists** is straightforward.

Used in **database queries, external sorting, and linked list sorting**.

**🔹 Merge Sort on an Array of Strings**

Sorting strings with Merge Sort ensures **lexicographical order**, making it ideal for **dictionary sorting, search engines, and natural language processing**.

✔ **Stable Sorting**, maintaining order of equal strings.  
✔ Efficient for **large text datasets**.

**🔹 Importance of Pivot in Quick Sort**

The **choice of pivot** determines Quick Sort’s efficiency:  
✔ **Bad pivot** → O(n²) worst-case time.  
✔ **Good pivot** → Balanced partitions with O(n log n) time.

Common pivot strategies:  
✔ **First or Last Element** – Risky for sorted inputs.  
✔ **Random Pivot** – Reduces worst-case occurrence.  
✔ **Median-of-Three** – Chooses the median of three values.

**🔹 How Pivot Selection Affects Quick Sort Performance**

✔ **Poor pivot selection** creates **unbalanced partitions**, leading to O(n²) complexity.  
✔ **Good pivot selection** ensures balanced subarrays, maintaining O(n log n) time.  
✔ **Random pivot selection** helps in avoiding **worst-case scenarios**.

Choosing an efficient pivot is **key to optimizing Quick Sort**.

**🔹 Choosing the Appropriate Sorting Algorithm**

✔ **Small arrays** – Insertion Sort.  
✔ **Unsorted large data** – Quick Sort.  
✔ **Sorted/Nearly sorted data** – Insertion Sort (O(n) best-case).  
✔ **Stable sorting required** – Merge Sort.  
✔ **Limited memory** – Quick Sort (in-place).

Choosing the right algorithm **optimizes performance and memory usage**.

**🔹 Sorting an Array of Objects based on a Property**

Sorting objects involves defining a **custom comparator**:  
✔ **Example** – Sorting an array of Employee objects by salary.  
✔ Used in **database sorting, ranking systems, and structured data organization**.

Custom sorting helps in **structuring complex data** efficiently.

**🔹 Check if an Array is Sorted in O(n) Time**

Checking if an array is sorted requires a **single pass (O(n))**:  
✔ Compare each element with the next one.  
✔ If any element is greater than the next, return False.

Used in **pre-check optimizations before applying sorting algorithms**.

### 5️⃣ ****Linked Lists****

Linked lists are **dynamic data structures** that store elements in **nodes**, with each node pointing to the next. They are useful for **efficient insertions and deletions** compared to arrays.

📌 **Key Advantages:**  
✔ **Efficient Insertions & Deletions** – O(1) at the head or tail.  
✔ **Dynamic Memory Allocation** – No need for a fixed size.  
✔ **No Memory Wastage** – Unlike arrays, which may have unused space.

📌 **Key Disadvantages:**  
❌ **Extra Memory Overhead** – Requires additional space for pointers.  
❌ **Slower Access** – O(n) time for searching (no direct indexing).

## 🔹 ****Basics of Linked Lists****

A **linked list** consists of **nodes**, where each node has:  
✔ **Data** – The value stored.  
✔ **Pointer** – A reference to the next node.

### ****Types of Linked Lists****

1️⃣ **Singly Linked List** – Each node points to the next node only.  
2️⃣ **Doubly Linked List** – Each node points to both next and previous nodes.  
3️⃣ **Circular Linked List** – The last node connects back to the first node.

📌 **Common Operations:**  
✔ **Insertion** – At the beginning, end, or middle.  
✔ **Deletion** – Removing nodes efficiently.  
✔ **Traversal** – Iterating through nodes to access elements.

## 🔹 ****Merge Two Sorted Linked Lists****

Merging two sorted linked lists **efficiently** ensures the result remains sorted.

**Approach:**  
1️⃣ Compare the heads of both lists.  
2️⃣ Append the smaller node to the result list.  
3️⃣ Move the pointer forward in the list where a node was taken.  
4️⃣ Repeat until all nodes are merged.

📌 **Time Complexity:** O(n + m) – where n and m are the sizes of the two lists.  
📌 **Space Complexity:** O(1) – In-place merging without extra memory.

🚀 **Used in:**  
✔ Merge Sort for Linked Lists.  
✔ Combining sorted datasets.

## 🔹 ****Remove Middle Element from Linked List****

Removing the middle element in a linked list is done using **two-pointer technique**:  
✔ **Slow pointer** moves **one step** at a time.  
✔ **Fast pointer** moves **two steps** at a time.  
✔ When the fast pointer reaches the end, the slow pointer is at the middle.

📌 **Time Complexity:** O(n) – Single traversal.  
📌 **Space Complexity:** O(1) – No extra memory used.

🚀 **Used in:**  
✔ Implementing **stack middle element removal**.  
✔ **Optimizing deletion** without needing extra traversal.

## 🔹 ****Circular Queue using Linked List****

A **Circular Queue** eliminates the need for shifting elements like in a normal queue.

✔ **Rear connects to Front** – Making queue circular.  
✔ **Efficient Enqueue/Dequeue** – O(1) time complexity.

📌 **Key Operations:**  
✔ **Enqueue** – Add elements at the rear.  
✔ **Dequeue** – Remove elements from the front.  
✔ **Check Full/Empty** – Detect queue status.

🚀 **Used in:**  
✔ CPU Scheduling (Round Robin).  
✔ Memory Buffers (Ring Buffers).

## 🔹 ****Implement Double-Ended Queue (Deque) using Linked List****

A **Deque (Double-Ended Queue)** allows:  
✔ **Insertion & Deletion at both ends**.  
✔ **Efficient memory usage** (compared to arrays).

📌 **Operations:**  
✔ **Insert at Front** – O(1).  
✔ **Insert at Rear** – O(1).  
✔ **Delete from Front** – O(1).  
✔ **Delete from Rear** – O(1).

🚀 **Used in:**  
✔ **Sliding Window Problems**.  
✔ **Deque-based BFS/DFS algorithms**.

### ****6️⃣ Arrays****

Arrays are one of the **most fundamental** data structures, used in nearly every programming problem. They provide **contiguous memory storage**, allowing **fast access** and **efficient iteration**.

📌 **Advantages of Arrays:**  
✔ **Direct Indexing (O(1) Access)** – Elements are stored in **continuous memory locations**.  
✔ **Cache-Friendly** – Due to sequential memory allocation.  
✔ **Easy to Implement** – Simpler than linked lists or other dynamic structures.

📌 **Disadvantages:**  
❌ **Fixed Size** – Static arrays require predefined sizes.  
❌ **Insertion/Deletion is Costly** – O(n) time required for shifting elements.  
❌ **Wasted Memory** – If unused elements are allocated.

## 🔹 ****Reverse an Array****

Reversing an array means **flipping the order** of its elements.

**Approach:**  
1️⃣ **Two-Pointer Swap:** Swap the first and last elements, moving towards the center.  
2️⃣ **Recursive Approach:** Swap first and last, then call the function on the smaller subarray.  
3️⃣ **Using Stack:** Push all elements onto a stack and pop them back in reverse order.

📌 **Time Complexity:** O(n) – Iterates through half the array.  
📌 **Space Complexity:** O(1) – Uses only a few extra variables.

🚀 **Used in:**  
✔ String and sentence reversal.  
✔ Data processing and transformations.

## 🔹 ****Remove Duplicates from an Array in O(n)****

Removing duplicates efficiently helps **optimize space** and **avoid unnecessary computations**.

**Approach:**  
✔ **Using a HashSet:** Store unique values and reconstruct the array.  
✔ **Two-Pointer Technique (For Sorted Arrays):** Move only when encountering a new element.  
✔ **Bit Manipulation (For Small Ranges):** Use bitwise flags to track seen numbers.

📌 **Time Complexity:** O(n) – Single pass through the array.  
📌 **Space Complexity:** O(1) (if done in-place) or O(n) (if using extra storage).

🚀 **Used in:**  
✔ **Data Cleaning** before processing datasets.  
✔ **Optimizing Search Queries** by avoiding duplicates.

## 🔹 ****Find the First Missing Number in an Array****

Finding the **smallest missing number** in an unsorted array is a common problem in competitive programming.

**Approach:**  
✔ **Sorting + Linear Scan:** Sort the array and check the missing number (O(n log n)).  
✔ **Using HashSet:** Store elements and check for the smallest missing integer (O(n) time, O(n) space).  
✔ **Cyclic Sort (Optimal for 1 to N):** Place each element at its correct index and detect the missing one in O(n) time and O(1) space.

📌 **Time Complexity:** O(n) – Using HashSet or Cyclic Sort.  
📌 **Space Complexity:** O(1) – If done in-place.

🚀 **Used in:**  
✔ **Database ID Allocation** (finding the next available ID).  
✔ **Competitive Programming Challenges**.

## 🔹 ****Find the Subarray with the Maximum Sum (Kadane’s Algorithm)****

Finding the **contiguous subarray** with the **largest sum** is useful in **financial modeling, AI, and data processing**.

**Kadane’s Algorithm (Efficient Approach):**  
1️⃣ Initialize maxSum and currentSum.  
2️⃣ Iterate through the array, adding elements to currentSum.  
3️⃣ If currentSum becomes negative, reset it to zero.  
4️⃣ Track the maximum value of currentSum.

📌 **Time Complexity:** O(n) – Single pass.  
📌 **Space Complexity:** O(1) – Uses only a few variables.

🚀 **Used in:**  
✔ **Stock Market Analysis** – Finding the best profit period.  
✔ **Image Processing** – Finding the brightest region in an image.

## 🔹 ****Merge Two Sorted Arrays in O(n) Time****

Merging two sorted arrays **efficiently** helps in **database merging, file processing, and sorting algorithms**.

**Approach (Two-Pointer Technique):**  
1️⃣ Start at the beginning of both arrays.  
2️⃣ Compare elements and pick the smallest.  
3️⃣ Move the pointer in the corresponding array.  
4️⃣ Continue until all elements are merged.

📌 **Time Complexity:** O(n) – Iterates through both arrays once.  
📌 **Space Complexity:** O(1) – If merging in-place, O(n) otherwise.

🚀 **Used in:**  
✔ **Merge Sort Algorithm**.  
✔ **Merging two sorted datasets efficiently**.

### ****7️⃣ Recursion****

Recursion is a **powerful problem-solving technique** where a function calls itself to **break a problem into smaller subproblems**. It is heavily used in **divide-and-conquer algorithms, tree traversal, dynamic programming, and backtracking**.

📌 **Key Concepts:**  
✔ **Base Case:** Defines when the recursion stops.  
✔ **Recursive Case:** Calls the function with a smaller input.  
✔ **Stack Usage:** Each function call is stored in the **call stack**.  
✔ **Time Complexity Consideration:** Recursive solutions may lead to **stack overflow** or inefficiencies if not optimized.

## 🔹 ****Basics of Recursion****

Recursion helps solve complex problems by **reducing them into simpler subproblems**.

**Example: Factorial Calculation (n!)**

n!=n×(n−1)!n! = n \times (n - 1)!

✔ **Base Case:** If n=0n = 0 or n=1n = 1, return 1.  
✔ **Recursive Case:** Multiply nn with the factorial of n−1n - 1.

def factorial(n):

if n == 0 or n == 1: # Base case

return 1

return n \* factorial(n - 1) # Recursive case

📌 **Time Complexity:** O(n) – Recursive depth is proportional to n.  
📌 **Space Complexity:** O(n) – Due to recursive function calls stored in the stack.

🚀 **Used in:**  
✔ **Mathematical problems (Factorial, Fibonacci, GCD)**.  
✔ **Algorithmic problem-solving (Sorting, Tree traversal, Dynamic programming)**.

## 🔹 ****Tail vs. Head Recursion****

Recursion is classified into **Tail Recursion** and **Head Recursion** based on when the recursive call is made.

📌 **Tail Recursion:** The **recursive call is the last operation** in the function. The function **does not need** to store intermediate results.  
✔ **Optimized by compilers** (Tail Call Optimization – TCO).

def tail\_recursive\_factorial(n, result=1):

if n == 0:

return result

return tail\_recursive\_factorial(n - 1, result \* n) # Tail Call

📌 **Time Complexity:** O(n)  
📌 **Space Complexity:** O(1) (if optimized by compiler).

📌 **Head Recursion:** The **recursive call is made first**, before performing any computation.  
✔ **Consumes more stack space**, as all calls remain active until the base case is reached.

def head\_recursive\_factorial(n):

if n == 0:

return 1

result = head\_recursive\_factorial(n - 1) # Head Call

return n \* result

🚀 **Used in:**  
✔ **Tail Recursion** – Used when iterative conversion is needed (e.g., Factorial, Sum of numbers).  
✔ **Head Recursion** – Used when computation depends on previous results (e.g., Tree Traversals).

## 🔹 ****Binary Recursion****

Binary recursion occurs when **a function makes two recursive calls** in each step. This is often used in **divide-and-conquer** strategies.

📌 **Example: Fibonacci Sequence**

F(n)=F(n−1)+F(n−2)F(n) = F(n-1) + F(n-2)

def fibonacci(n):

if n <= 1:

return n

return fibonacci(n - 1) + fibonacci(n - 2) # Two recursive calls

📌 **Time Complexity:** O(2^n) – Exponential growth due to duplicate calls.  
📌 **Space Complexity:** O(n) – Due to recursion stack.

✔ **Optimized Approach:** Use **memoization (caching)** to reduce redundant calculations.

def fibonacci\_memoized(n, memo={}):

if n in memo:

return memo[n]

if n <= 1:

return n

memo[n] = fibonacci\_memoized(n - 1, memo) + fibonacci\_memoized(n - 2, memo)

return memo[n]

🚀 **Used in:**  
✔ **Tree Traversal (Preorder, Inorder, Postorder).**  
✔ **Divide and Conquer Algorithms (Merge Sort, Quick Sort).**  
✔ **Dynamic Programming Problems (Fibonacci, Knapsack, Longest Common Subsequence).**

### ****8️⃣ General DSA Topics****

Before tackling complex algorithms, it is crucial to understand the **fundamental principles** of **data structures and algorithms (DSA)**. These core concepts help in optimizing **performance, memory usage, and problem-solving strategies**.

## 🔹 ****Time and Space Complexity Analysis****

Analyzing an algorithm’s **efficiency** involves studying its **time complexity** (execution speed) and **space complexity** (memory usage).

📌 **Time Complexity (Big-O Notation):**  
✔ **O(1)** – Constant time (e.g., accessing an array element).  
✔ **O(log n)** – Logarithmic time (e.g., Binary Search).  
✔ **O(n)** – Linear time (e.g., Traversing an array).  
✔ **O(n log n)** – Log-linear time (e.g., Merge Sort, Quick Sort).  
✔ **O(n²), O(2^n), O(n!)** – Quadratic, exponential, and factorial complexities (e.g., Nested loops, Recursive problems).

📌 **Space Complexity:**  
✔ Measures **additional memory** an algorithm needs.  
✔ Example: **Recursive algorithms** use **O(n) space** due to the call stack.

🚀 **Used in:**  
✔ Evaluating **algorithm performance**.  
✔ Selecting **optimal** data structures.

## 🔹 ****Linear vs. Non-Linear Data Structures****

Data structures are classified based on how **data elements** are organized.

📌 **Linear Data Structures:**  
✔ Elements are arranged **sequentially**.  
✔ Examples: **Arrays, Linked Lists, Stacks, Queues**.  
✔ Suitable for **simple operations like searching and sorting**.

📌 **Non-Linear Data Structures:**  
✔ Elements are connected **non-sequentially**.  
✔ Examples: **Trees, Graphs, Hash Tables**.  
✔ Used in **complex applications like databases, networks, and AI**.

🚀 **Used in:**  
✔ **Linear structures** – Best for simple and ordered data operations.  
✔ **Non-linear structures** – Ideal for **hierarchical** and **network-based** data representations.

## 🔹 ****Divide and Conquer Approach****

A **problem-solving paradigm** where a problem is broken into **smaller subproblems**, solved **recursively**, and then combined.

📌 **Steps:**  
✔ **Divide** – Split the problem into subproblems.  
✔ **Conquer** – Solve subproblems recursively.  
✔ **Combine** – Merge the solutions.

📌 **Examples:**  
✔ **Merge Sort (O(n log n))** – Splits an array, sorts subarrays, then merges them.  
✔ **Quick Sort (O(n log n))** – Partitions array around a pivot.  
✔ **Binary Search (O(log n))** – Divides search space into halves.

🚀 **Used in:**  
✔ **Sorting, Searching, Dynamic Programming, Computational Geometry**.

## 🔹 ****Memory Pool****

A **preallocated block of memory** used for managing memory dynamically, reducing **fragmentation and allocation overhead**.

📌 **How it works:**  
✔ A large block of memory is **reserved** in advance.  
✔ When an object needs memory, it is **allocated from the pool** instead of calling malloc/new.  
✔ When an object is destroyed, memory is **returned to the pool** for reuse.

📌 **Benefits:**  
✔ **Faster memory allocation/deallocation**.  
✔ **Avoids memory fragmentation**.  
✔ **Efficient in performance-critical applications**.

🚀 **Used in:**  
✔ **Game engines, real-time systems, high-performance applications**.

## 🔹 ****Virtual Memory****

A **memory management technique** that allows a system to use **more memory than physically available** by using disk storage as **extended RAM**.

📌 **How it works:**  
✔ The OS divides memory into **pages**.  
✔ **Inactive pages** are moved to disk (swap space).  
✔ When needed, pages are **loaded back into RAM**.

📌 **Benefits:**  
✔ **Runs large applications efficiently**.  
✔ **Prevents memory exhaustion**.  
✔ **Enhances multitasking**.

🚀 **Used in:**  
✔ **Operating systems, multitasking, memory-intensive applications**.

## 🔹 ****Purpose of Stack Pointer****

A **special CPU register** that keeps track of the **top of the stack** in memory.

📌 **Functions:**  
✔ **Manages function calls and local variables**.  
✔ **Helps in recursion by storing return addresses**.  
✔ **Facilitates memory allocation in stack memory**.

📌 **Example:**  
When calling a function, the **stack pointer**:  
1️⃣ Stores return addresses.  
2️⃣ Saves local variables.  
3️⃣ Moves up/down as functions execute/return.

🚀 **Used in:**  
✔ **Function calls, Recursion, System Programming**.