**Exercise 1: Inventory Management System**

**Product.java -> calss**

public class Product {

int productId;

String productName;

int quantity;

double price;

String producttype;

public Product(int productId, String productName, int quantity, double price,String producttype)

{

this.productId = productId;

this.productName = productName;

this.quantity = quantity;

this.price = price;

this.producttype = producttype;

}

public String toString() {

return productId + " | " + productName + " | Qty: " + quantity + " | Price: ₹" + price + " | Product type: ₹" + producttype;

}

}

**Inventorysystem.java**

import java.util.HashMap;

public class InventorySystem {

HashMap<Integer, Product> inventory = new HashMap<>();

public void addProduct(Product p) {

inventory.put(p.productId, p);

}

public void updateProduct(int id, int newQty, double newPrice) {

if (inventory.containsKey(id)) {

Product p = inventory.get(id);

p.quantity = newQty;

p.price = newPrice;

} else {

System.***out***.println("Product ID not found.");

}

}

public void deleteProduct(int id) {

inventory.remove(id);

}

public void displayInventory() {

for (Product p : inventory.values()) {

System.***out***.println(p);

}

}

public static void main(String[] args) {

InventorySystem inv = new InventorySystem();

System.***out***.println("----adding products---- ");

inv.addProduct(new Product(101, "Keyboard", 10, 500,"electronic"));

inv.addProduct(new Product(102, "Mouse", 25, 300,"electronic"));

inv.addProduct(new Product(103, "speaker", 5, 700,"electronic"));

inv.addProduct(new Product(101, "Keyboard", 15, 499.99, "Electronic"));

inv.addProduct(new Product(102, "Mouse", 25, 300.00, "Electronic"));

inv.addProduct(new Product(103, "Monitor", 10, 7999.99, "Electronic"));

inv.addProduct(new Product(104, "Desk Chair", 5, 3999.00, "Furniture"));

inv.addProduct(new Product(105, "Laptop", 8, 55999.00, "Electronic"));

inv.addProduct(new Product(106, "USB Cable", 50, 150.00, "Accessory"));

inv.addProduct(new Product(107, "Pen Drive", 35, 499.00, "Electronic"));

System.***out***.println("----Inventory display---- ");

inv.displayInventory();

System.***out***.println("----updateing product details with id 101---- ");

inv.updateProduct(101, 15, 450.00);

System.***out***.println("----deleteing product with id 102---- ");

inv.deleteProduct(102);

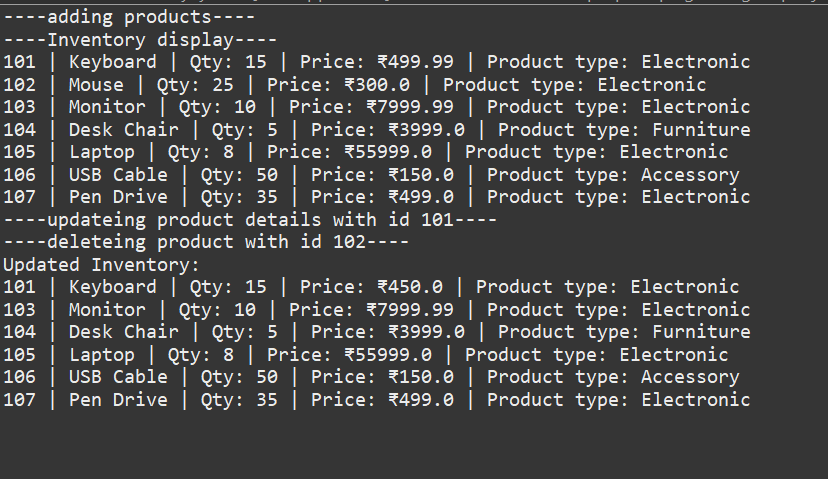
System.***out***.println("Updated Inventory:);

inv.displayInventory();

}

}

**Output :**



**Why Data Structures and Algorithms Are Essential:**

* **Efficient retrieval and storage**: Large inventories might include thousands of products. Using optimal data structures reduces time for operations like lookup, insertion, or deletion.
* **Searchability**: Allows you to quickly search by product ID, name, or other attributes.
* **Scalability**: As the inventory grows, performance should remain acceptable.
* **Maintainability**: Clean and efficient algorithms make it easier to extend and maintain the system.

#### Suitable Data Structures:

* **ArrayList** (or List):
  + Best when order matters.
  + Easy to iterate but slow search (O(n)).
* **HashMap**:
  + Best for fast access using keys (like productId).
  + Constant time average for add, delete, update operations.

#### Time Complexity (Comparison of HashMap vs ArrayList):

* **Add Operation:**
  + HashMap: O(1)
  + ArrayList: O(1) if added at the end
* **Search Operation:**
  + HashMap: O(1)
  + ArrayList: O(n)
* **Update Operation:**
  + HashMap: O(1)
  + ArrayList: O(n)
* **Delete Operation:**
  + HashMap: O(1)
  + ArrayList: O(n)

**Exercise 2: E-commerce Platform Search Function**

**Product.java -> class**

import java.util.Arrays;

import java.util.Comparator;

public class Product {

int productId;

String productName;

String category;

public Product(int productId, String productName, String category) {

this.productId = productId;

this.productName = productName;

this.category = category;

}

*@Override*

public String toString() {

return String.*format*("%-5d | %-15s | %-10s", productId, productName, category);

}

// 🔍 Linear Search method (static)

public static Product linearSearch(Product[] products, String name) {

for (Product p : products) {

if (p.productName.equalsIgnoreCase(name)) {

return p;

}

}

return null;

}

// 🔍 Binary Search method (static)

public static Product binarySearch(Product[] products, String name) {

Arrays.*sort*(products, Comparator.*comparing*(p -> p.productName.toLowerCase()));

int left = 0, right = products.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

int cmp = name.compareToIgnoreCase(products[mid].productName);

if (cmp == 0)

return products[mid];

else if (cmp < 0)

right = mid - 1;

else

left = mid + 1;

}

return null;

}

}

**Ecomers search.java -> class**

public class ECommercePlatform {

public static void main(String[] args) {

Product[] products = {

new Product(101, "Keyboard", "Electronic"),

new Product(102, "Mouse", "Electronic"),

new Product(103, "Desk", "Furniture"),

new Product(104, "Notebook", "Stationery"),

new Product(105, "Monitor", "Electronic")

};

// 🔍 Perform Linear Search

System.***out***.println("----finding product through linear search through target---- ");

Product foundLinear = Product.*linearSearch*(products, "Monitor");

System.***out***.println("🔍 Linear Search Result: " + (foundLinear != null ? foundLinear : "Product Not Found"));

// 🔍 Perform Binary Search

#### What is Big O Notation?

* **Big O notation** describes the **upper bound** of an algorithm's running time or space requirement in terms of input size n.
* It provides a **high-level understanding** of performance regardless of the machine or environment.
* It helps developers:
  + Analyze scalability.
  + Predict performance with larger inputs.
  + Compare alternative algorithms efficiently.

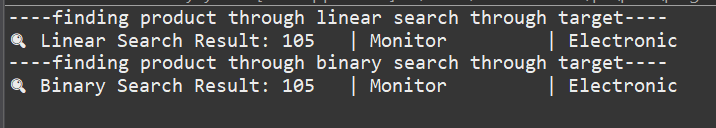
#### Time Complexity Comparison:

* **Linear Search:**
  + Time complexity: O(n)
  + Space complexity: O(1)
  + No need for sorting
* **Binary Search:**
  + Time complexity: O(log n)
  + Space complexity: O(1)
  + Requires sorted data

#### ⚖️ Which Algorithm Is More Suitable?

* **Use Linear Search** if:
  + Your dataset is small.
  + Products are not sorted.
  + Search is infrequent or for non-indexable attributes (e.g., complex filters).
* **Use Binary Search** if:
  + Dataset is large and can be **kept sorted**.
  + Search is frequent and **performance-critical**.
  + Searching by indexable attributes like productId.

Output:



**Exercise 3: Sorting Customer Orders**

**Main.java -> main () calss**

public class Main {

public static void main(String[] args) {

Order[] orders = {

new Order(1, "Alice", 500.0),

new Order(2, "Bob", 1500.0),

new Order(3, "Charlie", 750.0),

new Order(4, "Diana", 300.0)

};

System.***out***.println("Original Orders:");

for (Order o : orders) System.***out***.println(o);

// Bubble Sort

Order[] bubbleSorted = orders.clone();

OrderSorter.*bubbleSort*(bubbleSorted);

System.***out***.println("\nSorted by Bubble Sort:");

for (Order o : bubbleSorted) System.***out***.println(o);

// Quick Sort

Order[] quickSorted = orders.clone();

OrderSorter.*quickSort*(quickSorted, 0, quickSorted.length - 1);

System.***out***.println("\nSorted by Quick Sort:");

for (Order o : quickSorted) System.***out***.println(o);

}

}

Order.java

public class Order {

int orderId;

String customerName;

double totalPrice;

public Order(int orderId, String customerName, double totalPrice) {

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

*@Override*

public String toString() {

return "[" + orderId + ", " + customerName + ", ₹" + totalPrice + "]";

}

}

Ordersoted.java

public class OrderSorter {

// 🔹 Bubble Sort

public static void bubbleSort(Order[] orders) {

int n = orders.length;

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (orders[j].totalPrice > orders[j + 1].totalPrice) {

Order temp = orders[j];

orders[j] = orders[j + 1];

orders[j + 1] = temp;

}

}

}

}

// 🔹 Quick Sort

public static void quickSort(Order[] orders, int low, int high) {

if (low < high) {

int pivotIndex = *partition*(orders, low, high);

*quickSort*(orders, low, pivotIndex - 1);

*quickSort*(orders, pivotIndex + 1, high);

}

}

private static int partition(Order[] orders, int low, int high) {

double pivot = orders[high].totalPrice;

int i = low - 1;

for (int j = low; j < high; j++) {

if (orders[j].totalPrice < pivot) {

i++;

Order temp = orders[i];

orders[i] = orders[j];

orders[j] = temp;

}

}

Order temp = orders[i + 1];

orders[i + 1] = orders[high];

orders[high] = temp;

return i + 1;

}

}

#### ****Bubble Sort****:

* **How it works**: Repeatedly compares adjacent elements and swaps them if they’re in the wrong order.
* **Key trait**: Simple to implement but inefficient on large datasets.
* **Time Complexity**:
  + Best Case: O(n) (already sorted, with optimized version)
  + Average/Worst Case: O(n²)

#### ****Insertion Sort****:

* **How it works**: Builds the final sorted array one element at a time by inserting each item into its proper place.
* **Good for**: Small datasets or nearly sorted data.
* **Time Complexity**:
  + Best Case: O(n)
  + Worst Case: O(n²)

#### ****Quick Sort****:

* **How it works**: Selects a 'pivot', partitions the array into elements less than and greater than the pivot, and recursively sorts them.
* **Efficient for**: Large datasets.
* **Time Complexity**:
  + Best/Average Case: O(n log n)
  + Worst Case: O(n²) (rare; when pivot is poorly chosen)

#### ****Merge Sort****:

* **How it works**: Divides the array into halves, recursively sorts them, then merges them.
* **Stable and consistent** in performance.
* **Time Complexity**: Always O(n log n)

**Exercise 4: Employee Management System**

**Order.java - > calss**

class Order {

int orderId;

String customerName;

double totalPrice;

public Order(int orderId, String customerName, double totalPrice) {

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

*@Override*

public String toString() {

return "[" + orderId + ", " + customerName + ", " + totalPrice + "]";

}

}

**OrderSorter.java -> calss**

public class OrderSorter {

// Bubble Sort

public static void bubbleSort(Order[] orders) {

int n = orders.length;

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (orders[j].totalPrice > orders[j + 1].totalPrice) {

// Swap

Order temp = orders[j];

orders[j] = orders[j + 1];

orders[j + 1] = temp;

}

}

}

}

// Quick Sort

public static void quickSort(Order[] orders, int low, int high) {

if (low < high) {

int pi = *partition*(orders, low, high);

*quickSort*(orders, low, pi - 1);

*quickSort*(orders, pi + 1, high);

}

}

private static int partition(Order[] orders, int low, int high) {

double pivot = orders[high].totalPrice;

int i = low - 1;

for (int j = low; j < high; j++) {

if (orders[j].totalPrice < pivot) {

i++;

// Swap

Order temp = orders[i];

orders[i] = orders[j];

orders[j] = temp;

}

}

// Swap pivot

Order temp = orders[i + 1];

orders[i + 1] = orders[high];

orders[high] = temp;

return i + 1;

}

}

**Main.java -> main () calss**

public class Main {

public static void main(String[] args) {

Order[] orders = {

new Order(101, "Vamsi", 850.0),

new Order(102, "Durga", 1025.5),

new Order(103, "Durga vamsi", 6000.75),

new Order(104, "Rolex", 850.25),

new Order(101, "Alice", 150.0)

};

System.***out***.println("Original Orders:");

for (Order o : orders) System.***out***.println(o);

System.***out***.println("----bubble sort o(n)----");

Order[] bubbleSorted = orders.clone();

OrderSorter.*bubbleSort*(bubbleSorted);

System.***out***.println("\nBubble Sorted Orders:");

for (Order o : bubbleSorted) System.***out***.println(o);

System.***out***.println("----quick sort o(nlogn)----");

Order[] quickSorted = orders.clone();

OrderSorter.*quickSort*(quickSorted, 0, quickSorted.length - 1);

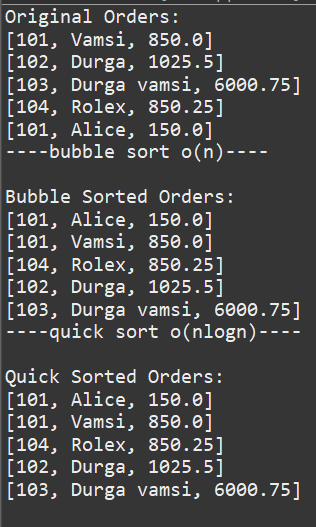
System.***out***.println("\nQuick Sorted Orders:");

for (Order o : quickSorted) System.***out***.println(o);

}

}

**OUTPUT:**



#### Array Memory Representation:

* Arrays are **contiguous blocks of memory**, meaning all elements are stored in adjacent locations.
* Each element can be accessed in **constant time** using its index due to the direct memory addressing.

#### Advantages of Using Arrays:

* **Fast access** to elements using indices (O(1) time).
* **Efficient iteration** since elements are stored sequentially.
* **Predictable memory usage** as the size is fixed at declaration.
* **Low overhead** compared to some dynamic structures.

#### Time Complexity:

* **Add Operation**:
  + Time: O(1), if space is available.
  + Worst case: O(n), if resizing is required (in case of dynamic array implementations).
* **Search Operation**:
  + Time: O(n), as a linear search is needed unless sorted or indexed.
* **Traverse Operation**:
  + Time: O(n), as each element is visited once.
* **Delete Operation**:
  + Time: O(n), due to shifting elements after deletion.

#### Limitations of Arrays:

* **Fixed size**: Once declared, the size cannot change unless a new array is created and copied.
* **Inefficient deletion/insertion**: Requires shifting elements.
* **Wasted memory**: May lead to unused space if the array is under-utilized.
* **No dynamic resizing**: Must anticipate maximum size in advance.

#### When to Use Arrays:

* When the number of elements is known and fixed.
* When fast indexed access is required.
* When memory layout and performance for access are more critical than flexibility in size or structure.

**Exercise 5: Task Management Syste**

**Task.java -> calss**

class Task {

int taskId;

String taskName;

String status;

Task next; // pointer for singly linked list

public Task(int taskId, String taskName, String status) {

this.taskId = taskId;

this.taskName = taskName;

this.status = status;

this.next = null;

}

*@Override*

public String toString() {

return "[" + taskId + ", " + taskName + ", " + status + "]";

}

}

**TaskLinkedlist.java -> calss**

class TaskLinkedList {

private Task head;

// Add task at end

public void addTask(int id, String name, String status) {

Task newTask = new Task(id, name, status);

if (head == null) {

head = newTask;

} else {

Task temp = head;

while (temp.next != null) {

temp = temp.next;

}

temp.next = newTask;

}

}

// Search task by ID

public Task searchTask(int id) {

Task temp = head;

while (temp != null) {

if (temp.taskId == id) return temp;

temp = temp.next;

}

return null;

}

// Delete task by ID

public boolean deleteTask(int id) {

if (head == null) return false;

if (head.taskId == id) {

head = head.next;

return true;

}

Task prev = head;

Task current = head.next;

while (current != null) {

if (current.taskId == id) {

prev.next = current.next;

return true;

}

prev = current;

current = current.next;

}

return false;

}

// Traverse and print all tasks

public void traverseTasks() {

Task temp = head;

if (temp == null) {

System.***out***.println("Task list is empty.");

return;

}

while (temp != null) {

System.***out***.println(temp);

temp = temp.next;

}

}

}

**Main.java -> main () calss**

public class Main {

public static void main(String[] args) {

TaskLinkedList taskList = new TaskLinkedList();

// Adding tasks

taskList.addTask(1, "Design UI", "Pending");

taskList.addTask(2, "Develop Backend", "In Progress");

taskList.addTask(3, "Testing", "Not Started");

System.***out***.println("All Tasks:");

taskList.traverseTasks();

// Search for a task

System.***out***.println("\nSearching Task ID 2:");

Task t = taskList.searchTask(2);

System.***out***.println(t != null ? t : "Task not found.");

// Delete a task

System.***out***.println("\nDeleting Task ID 2:");

boolean deleted = taskList.deleteTask(2);

System.***out***.println(deleted ? "Task deleted." : "Task not found.");

// Display after deletion

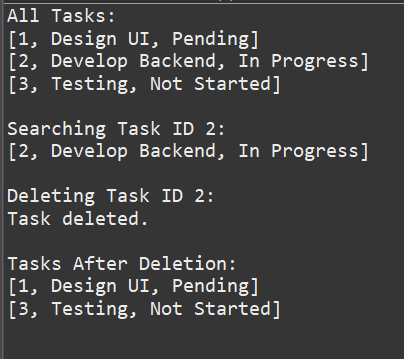
System.***out***.println("\nTasks After Deletion:");

taskList.traverseTasks();

}

}

**OUTPUT:**

****

#### Singly Linked List:

* A **singly linked list** is a linear data structure where each node contains:
  + **Data** (such as a task)
  + A **reference (pointer)** to the next node in the list
* The last node’s pointer is null, indicating the end of the list.
* It allows **unidirectional traversal** (from head to tail).

#### Doubly Linked List:

* A **doubly linked list** is a more flexible variant where each node contains:
  + **Data**
  + A **pointer to the next node**
  + A **pointer to the previous node**
* Allows **bidirectional traversal**, which makes insertion and deletion at both ends more efficient.

#### Time Complexity of Operations:

* **Add Operation**:
  + Time: O(1) if inserting at the head
  + Time: O(n) if inserting at the end without a tail pointer
* **Search Operation**:
  + Time: O(n), requires linear traversal
* **Traverse Operation**:
  + Time: O(n), all nodes must be visited sequentially
* **Delete Operation**:
  + Time: O(n), as it requires locating the target node before deletion

**Exercise 6: Library Management System**

**Book.java -> class**

import java.util.Arrays;

import java.util.Comparator;

public class Book {

int bookId;

String title;

String author;

public Book(int bookId, String title, String author) {

this.bookId = bookId;

this.title = title;

this.author = author;

}

*@Override*

public String toString() {

return "[" + bookId + ", " + title + ", " + author + "]";

}

// 🔹 Linear Search by Title

public static Book linearSearch(Book[] books, String targetTitle) {

for (Book book : books) {

if (book.title.equalsIgnoreCase(targetTitle)) {

return book;

}

}

return null;

}

// 🔹 Binary Search by Title (array must be sorted)

public static Book binarySearch(Book[] books, String targetTitle) {

int left = 0;

int right = books.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

int cmp = books[mid].title.compareToIgnoreCase(targetTitle);

if (cmp == 0) {

return books[mid];

} else if (cmp < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

// 🔹 Utility: Sort array of books by title

public static void sortBooksByTitle(Book[] books) {

Arrays.*sort*(books, Comparator.*comparing*(b -> b.title.toLowerCase()));

}

}

**Main.java -> main() calss**

public class Main {

public static void main(String[] args) {

Book[] books = {

new Book(101, "Data Structures", "Mark Allen"),

new Book(102, "Algorithms", "Robert Lafore"),

new Book(103, "Clean Code", "Robert Martin"),

new Book(104, "Java Basics", "Herbert Schildt")

};

// Linear Search

System.***out***.println("----finding Book through linear search through target book tittle---- ");

System.***out***.println("Linear Search for 'Clean Code':");

Book result1 = Book.*linearSearch*(books, "Clean Code");

System.***out***.println(result1 != null ? result1 : "Book not found.");

// Sort before Binary Search

Book.*sortBooksByTitle*(books);

// Binary Search

System.***out***.println("----finding Book through binary search through target book tittle---- ");

System.***out***.println("Binary Search for 'Java Basics':");

Book result2 = Book.*binarySearch*(books, "Java Basics");

System.***out***.println(result2 != null ? result2 : "Book not found.");

}

}

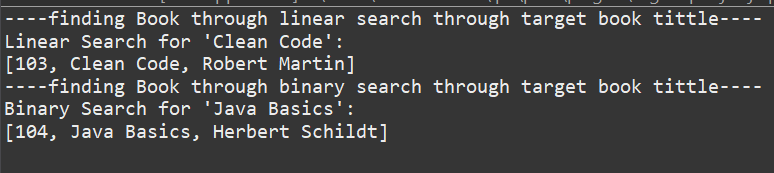
System.***out***.println("----finding product through binary search through target---- ");

Product foundBinary = Product.*binarySearch*(products, "Monitor");

System.***out***.println("🔍 Binary Search Result: " + (foundBinary != null ? foundBinary : "Product Not Found"));

}

}

**OUTPUT: **

#### Linear Search:

* **Definition**: A sequential search algorithm that examines each element in the list one by one until the desired item is found or the end is reached.
* **Characteristics**:
  + Does **not require the list to be sorted**.
  + Can be applied to **any data structure**, including arrays and lists.
  + **Inefficient for large datasets** as it checks every item.

#### Binary Search:

* **Definition**: A divide-and-conquer algorithm that repeatedly splits a sorted list in half to search for an item.
* **Characteristics**:
  + **Requires the list to be sorted** (e.g., by title).
  + Compares the target with the middle element, then discards one half of the list based on the result.
  + Much **faster for large, sorted datasets**.

#### Time Complexity Comparison:

* **Linear Search**:
  + Best Case: O(1) (first element is a match)
  + Average Case: O(n)
  + Worst Case: O(n)
* **Binary Search**:
  + Best Case: O(1) (middle element is a match)
  + Average Case: O(log n)
  + Worst Case: O(log n)

#### When to Use Each Algorithm:

* **Linear Search** is appropriate when:
  + The dataset is **unsorted**.
  + The list is **small**.
  + Search frequency is **low** or performance is not critical.
* **Binary Search** is appropriate when:
  + The dataset is **large**.
  + The list is **pre-sorted** or can be maintained in sorted order.
  + Frequent searching is expected and performance is a priority.

**Exercise 7: Financial Forecasting**

**FinancialForecast.java -> calss**

public class FinancialForecast {

// Recursive method to compute future value

public static double futureValueRecursive(double presentValue, double growthRate, int years) {

if (years == 0) {

return presentValue;

}

return (1 + growthRate) \* *futureValueRecursive*(presentValue, growthRate, years - 1);

}

// Optimized version using memoization

public static double futureValueMemo(double presentValue, double growthRate, int years, Double[] memo) {

if (years == 0) return presentValue;

if (memo[years] != null) return memo[years];

memo[years] = (1 + growthRate) \* *futureValueMemo*(presentValue, growthRate, years - 1, memo);

return memo[years];

}

}

**Main.java -> main() calss**

public class Main {

public static void main(String[] args) {

double presentValue = 10000; // Initial amount

double growthRate = 0.08; // 8% annual growth

int years = 5;

// Regular recursion

double future = FinancialForecast.*futureValueRecursive*(presentValue, growthRate, years);

System.***out***.println("Recursive Future Value after " + years + " years: ₹" + future);

// Memoized recursion

Double[] memo = new Double[years + 1];

double optimizedFuture = FinancialForecast.*futureValueMemo*(presentValue, growthRate, years, memo);

System.***out***.println("Memoized Future Value after " + years + " years: ₹" + optimizedFuture);

}

}

**OUTPUT:**

****

**What Is Recursion?**

* **Recursion** is a problem-solving technique in which a function calls itself to solve smaller instances of the same problem.
* It involves:
  + **Base Case**: The condition under which the recursion stops.
  + **Recursive Case**: The part where the function calls itself with a simpler or smaller input.

**Why Use Recursion?**

* **Simplifies complex problems** by breaking them into smaller, manageable sub-problems.
* Particularly useful in problems involving:
  + Mathematical computations (e.g., factorial, Fibonacci series).
  + Tree/graph traversal.
  + Divide-and-conquer strategies (e.g., merge sort, quicksort).

#### Time Complexity:

* The recursive function computes a value once per period.
* Therefore, the **time complexity is O(n)**, where n is the number of periods.
* The **space complexity** is also O(n) due to the recursive call stack.