**Exercise 1: Inventory Management System**

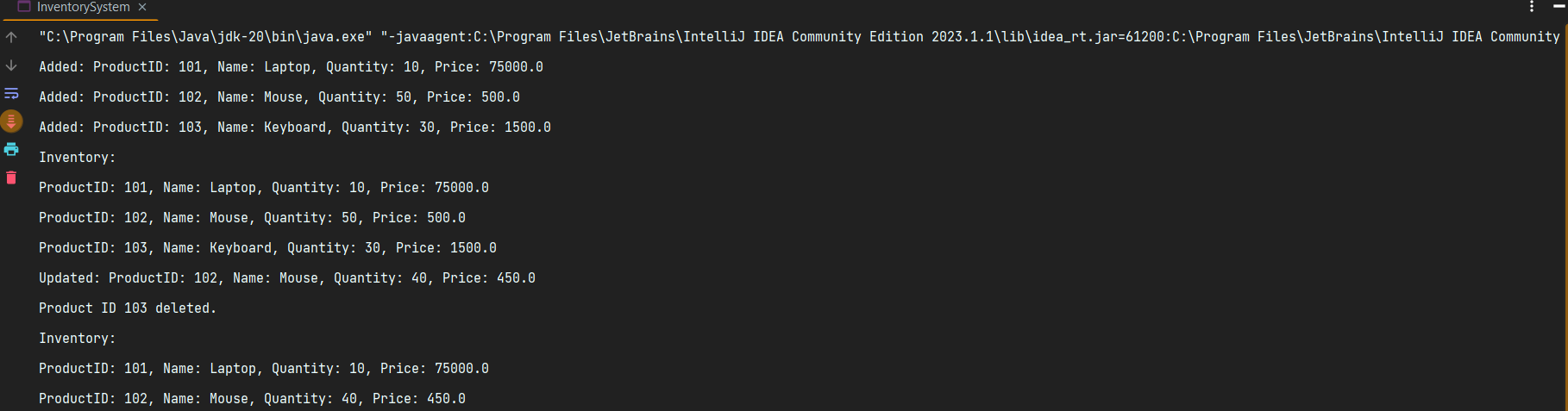
**Why Data Structures and Algorithms are Essential:**  
Efficient data handling is critical when managing large inventories. Data structures allow organized storage, while algorithms enable fast searching, updating, and deleting. Without them, operations become slow and inefficient, especially as the dataset grows.

**Suitable Data Structures:**

* **ArrayList** – For ordered storage, but slower search/update.
* **HashMap** – Ideal for fast access using productId as a key.

**Class Design and Implementation:**

*import* java.util.\*;  
*class* Product {  
 *int* productId;  
 String productName;  
 *int* quantity;  
 *double* price;  
 *public* Product(*int* productId, String productName, *int* quantity, *double* price) {  
 *this*.productId = productId;  
 *this*.productName = productName;  
 *this*.quantity = quantity;  
 *this*.price = price;  
 }  
 @Override  
 *public* String toString() {  
 *return* "ProductID: " + productId + ", Name: " + productName +  
 ", Quantity: " + quantity + ", Price: " + price;  
 }  
}  
*public class* InventorySystem {  
 *static* HashMap<Integer, Product> *inventory* = *new* HashMap<>();  
 *public static void* addProduct(Product product) {  
 *inventory*.put(product.productId, product);  
 System.***out***.println("Added: " + product);  
 }  
 *public static void* updateProduct(*int* id, *int* quantity, *double* price) {  
 *if* (*inventory*.containsKey(id)) {  
 Product p = *inventory*.get(id);  
 p.quantity = quantity;  
 p.price = price;  
 System.***out***.println("Updated: " + p);  
 } *else* {  
 System.***out***.println("Product ID " + id + " not found.");  
 }  
 }  
 *public static void* deleteProduct(*int* id) {  
 *if* (*inventory*.remove(id) != *null*) {  
 System.***out***.println("Product ID " + id + " deleted.");  
 } *else* {  
 System.***out***.println("Product ID " + id + " not found.");  
 }  
 }  
 *public static void* displayInventory() {  
 System.***out***.println("Inventory:");  
 *for* (Product p : *inventory*.values()) {  
 System.***out***.println(p);  
 }  
 }  
 *public static void* main(String[] args) {  
 *addProduct*(*new* Product(101, "Laptop", 10, 75000.0));  
 *addProduct*(*new* Product(102, "Mouse", 50, 500.0));  
 *addProduct*(*new* Product(103, "Keyboard", 30, 1500.0));  
  
 *displayInventory*();  
  
 *updateProduct*(102, 40, 450.0);  
 *deleteProduct*(103);  
  
 *displayInventory*();  
 }  
}

Output:  


**Time Complexity:**

* Add: O(1) average
* Update: O(1) average
* Delete: O(1) average
* Search: O(1) average

**Optimization:** Use HashMap for constant-time operations, especially when dealing with large inventories.

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

**Big O Notation:**  
Measures the growth of algorithm time or space with input size. Helps compare efficiencies.

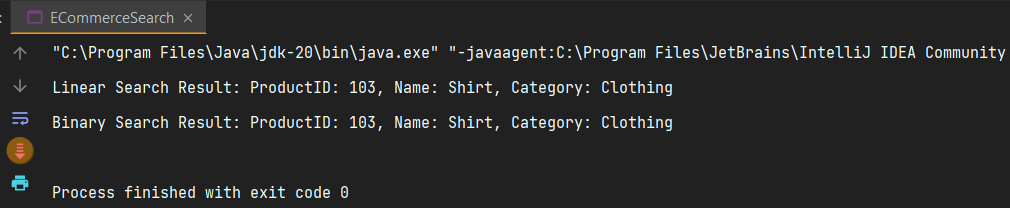
**Search Time Cases:**

* **Best**: First item (O(1))
* **Average**: Middle (O(n/2) or O(log n))
* **Worst**: Last or not found (O(n) or O(log n))

**Java Code**:

*import* java.util.\*;  
*class* Product {  
 *int* productId;  
 String productName;  
 String category;  
 Product(*int* id, String name, String category) {  
 *this*.productId = id;  
 *this*.productName = name;  
 *this*.category = category;  
 }  
 *public* String toString() {  
 *return* "ProductID: " + productId + ", Name: " + productName + ", Category: " + category;  
 }  
}  
*public class* ECommerceSearch {  
 *static* Product[] *products* = {  
 *new* Product(101, "Laptop", "Electronics"),  
 *new* Product(102, "Mouse", "Electronics"),  
 *new* Product(103, "Shirt", "Clothing"),  
 *new* Product(104, "Book", "Education")  
 };  
 *public static int* linearSearch(String name) {  
 *for* (*int* i = 0; i < *products*.length; i++) {  
 *if* (*products*[i].productName.equalsIgnoreCase(name)) *return* i;  
 }  
 *return* -1;  
 }  
 *public static int* binarySearch(Product[] sortedProducts, String name) {  
 *int* low = 0, high = sortedProducts.length - 1;  
 *while* (low <= high) {  
 *int* mid = (low + high) / 2;  
 *int* cmp = sortedProducts[mid].productName.compareToIgnoreCase(name);  
 *if* (cmp == 0) *return* mid;  
 *else if* (cmp < 0) low = mid + 1;  
 *else* high = mid - 1;  
 }  
 *return* -1;  
 }  
 *public static void* main(String[] args) {  
 *int* index = *linearSearch*("Shirt");  
 System.***out***.println("Linear Search Result: " + (index != -1 ? *products*[index] : "Not Found"));  
  
 Arrays.*sort*(*products*, *Comparator*.*comparing*(p -> p.productName.toLowerCase()));  
 *int* binIndex = *binarySearch*(*products*, "Shirt");  
 System.***out***.println("Binary Search Result: " + (binIndex != -1 ? *products*[binIndex] : "Not Found"));  
 }  
}

Output:



Time Complexity:

Algorithm Best Average Worst

Linear Search O(1) O(n) O(n)

Binary Search O(1) O(log n) O(log n)

**Comparison:**  
Binary search is faster but requires sorted data. Use it when products are ordered.

**Exercise 3: Sorting Customer Orders**

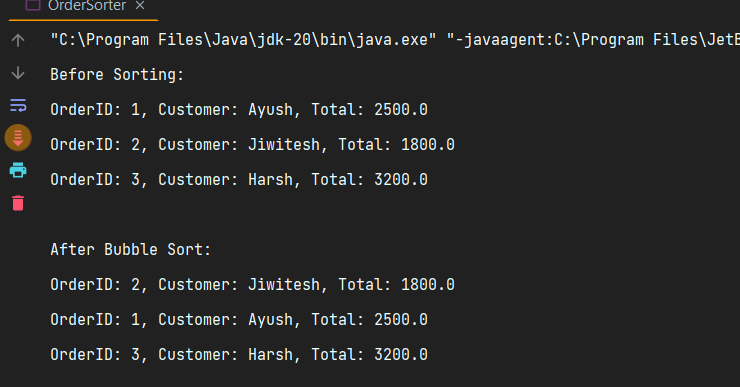
**Sorting Algorithms:**

* **Bubble Sort** – O(n²), Repeatedly compares adjacent elements and swaps them if they're in the wrong order.
* **Insertion Sort** – O(n²), Builds the sorted array one element at a time by inserting each item into its correct position.
* **Quick Sort** – Average: O(n log n), Worst: O(n²), A divide-and-conquer algorithm. Picks a pivot, partitions the array around it, and recursively sorts the subarrays.
* **Merge Sort** – O(n log n), Another divide-and-conquer approach. Recursively splits the array in halves, sorts each half, and merges them.

**Java Code**:

*class* Order {  
 *int* orderId;  
 String customerName;  
 *double* totalPrice;  
 Order(*int* id, String name, *double* price) {  
 *this*.orderId = id;  
 *this*.customerName = name;  
 *this*.totalPrice = price;  
 }  
 *public* String toString() {  
 *return* "OrderID: " + orderId + ", Customer: " + customerName + ", Total: " + totalPrice;  
 }  
}  
*public class* OrderSorter {  
 *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;  
 }  
 }  
 *public static void* quickSort(Order[] arr, *int* low, *int* high) {  
 *if* (low < high) {  
 *int* pi = *partition*(arr, low, high);  
 *quickSort*(arr, low, pi-1);  
 *quickSort*(arr, pi+1, high);  
 }  
 }  
 *private static int* partition(Order[] arr, *int* low, *int* high) {  
 *double* pivot = arr[high].totalPrice;  
 *int* i = low - 1;  
 *for* (*int* j = low; j < high; j++) {  
 *if* (arr[j].totalPrice < pivot) {  
 i++;  
 Order temp = arr[i];  
 arr[i] = arr[j];  
 arr[j] = temp;  
 }  
 }  
 Order temp = arr[i+1];  
 arr[i+1] = arr[high];  
 arr[high] = temp;  
 *return* i + 1;  
 }  
 *public static void* main(String[] args) {  
 Order[] orders = {  
 *new* Order(1, "Ayush", 2500),  
 *new* Order(2, "Jiwitesh", 1800),  
 *new* Order(3, "Harsh", 3200)  
 };  
 System.***out***.println("Before Sorting:");  
 *for* (Order o : orders) System.***out***.println(o);  
 *bubbleSort*(orders);  
 System.***out***.println("\nAfter Bubble Sort:");  
 *for* (Order o : orders) System.***out***.println(o);  
 }  
}

Output:



**Analysis:**

**Time Complexity:**

| **Algorithm** | **Best** | **Average** | **Worst** |
| --- | --- | --- | --- |
| Bubble Sort | O(n) | O(n²) | O(n²) |
| Quick Sort | O(n log n) | O(n log n) | O(n²) (rare) |

**Quick Sort is preferred due to better average-case performance.**

**Exercise 4: Employee Management System**

**Array Representation in Memory:**

Arrays are stored in **contiguous memory blocks**. The first element starts at a base address, and each element is located using: address = base\_address + index × element\_size

This enables **fast O(1) access** via indexing.

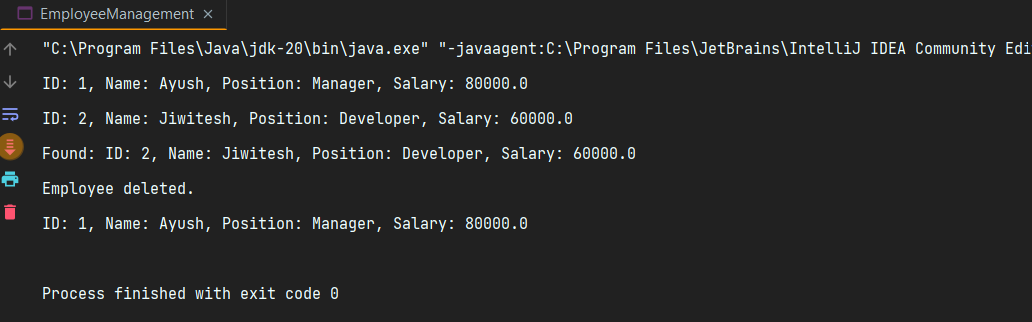
**Advantages:**

1. **Fast access** to any element.
2. **Cache-friendly** due to contiguous memory.
3. **Simple structure**, easy to implement.
4. **Low memory overhead** (no pointers like in linked lists).

Code:

*class* Employee {  
 *int* employeeId;  
 String name;  
 String position;  
 *double* salary;  
 *public* Employee(*int* id, String name, String position, *double* salary) {  
 *this*.employeeId = id;  
 *this*.name = name;  
 *this*.position = position;  
 *this*.salary = salary;  
 }  
 *public* String toString() {  
 *return* "ID: " + employeeId + ", Name: " + name + ", Position: " + position + ", Salary: " + salary;  
 }  
}  
*public class* EmployeeManagement {  
 *static* Employee[] *employees* = *new* Employee[100];  
 *static int count* = 0;  
 *public static void* addEmployee(Employee e) {  
 *employees*[*count*++] = e;  
 }  
 *public static void* searchEmployee(*int* id) {  
 *for* (*int* i = 0; i < *count*; i++) {  
 *if* (*employees*[i].employeeId == id) {  
 System.***out***.println("Found: " + *employees*[i]);  
 *return*;  
 }  
 }  
 System.***out***.println("Employee not found.");  
 }  
 *public static void* displayAll() {  
 *for* (*int* i = 0; i < *count*; i++) System.***out***.println(*employees*[i]);  
 }  
 *public static void* deleteEmployee(*int* id) {  
 *for* (*int* i = 0; i < *count*; i++) {  
 *if* (*employees*[i].employeeId == id) {  
 *for* (*int* j = i; j < *count* - 1; j++) {  
 *employees*[j] = *employees*[j+1];  
 }  
 *count*--;  
 System.***out***.println("Employee deleted.");  
 *return*;  
 }  
 }  
 System.***out***.println("Employee not found.");  
 }  
 *public static void* main(String[] args) {  
 *addEmployee*(*new* Employee(1, "Ayush", "Manager", 80000));  
 *addEmployee*(*new* Employee(2, "Jiwitesh", "Developer", 60000));  
 *displayAll*();  
 *searchEmployee*(2);  
 *deleteEmployee*(2);  
 *displayAll*();  
 }  
}

Output:



**Time complexity of each Operation in Array:**

* **Add:** O(1) (if space available)
* **Search:** O(n)
* **Traverse:** O(n)
* **Delete:** O(n) (requires shifting)

**Limitations:**  
Fixed size, inefficient insert/delete. Use arrays for small/mostly static data.

**Exercise 5: Task Management System**

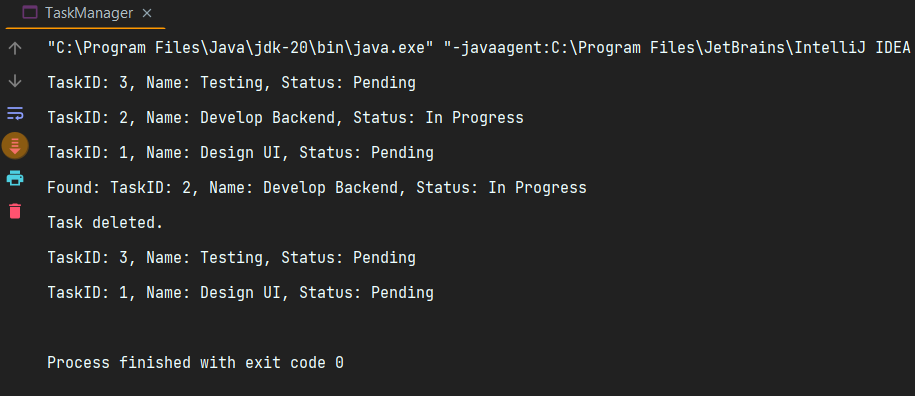
**Linked Lists:**

* **Singly:** One-directional
* **Doubly:** Bi-directional, more overhead

Code:

*class* Task {  
 *int* taskId;  
 String taskName;  
 String status;  
 Task next;  
 *public* Task(*int* id, String name, String status) {  
 *this*.taskId = id;  
 *this*.taskName = name;  
 *this*.status = status;  
 *this*.next = *null*;  
 }  
 *public* String toString() {  
 *return* "TaskID: " + taskId + ", Name: " + taskName + ", Status: " + status;  
 }  
}  
*public class* TaskManager {  
 *static* Task *head* = *null*;  
 *public static void* addTask(*int* id, String name, String status) {  
 Task newTask = *new* Task(id, name, status);  
 newTask.next = *head*;  
 *head* = newTask;  
 }  
 *public static void* deleteTask(*int* id) {  
 Task temp = *head*, prev = *null*;  
 *while* (temp != *null* && temp.taskId != id) {  
 prev = temp;  
 temp = temp.next;  
 }  
 *if* (temp == *null*) {  
 System.***out***.println("Task not found.");  
 *return*;  
 }  
 *if* (prev == *null*) *head* = temp.next;  
 *else* prev.next = temp.next;  
 System.***out***.println("Task deleted.");  
 }  
 *public static void* searchTask(*int* id) {  
 Task temp = *head*;  
 *while* (temp != *null*) {  
 *if* (temp.taskId == id) {  
 System.***out***.println("Found: " + temp);  
 *return*;  
 }  
 temp = temp.next;  
 }  
 System.***out***.println("Task not found.");  
 }  
 *public static void* displayTasks() {  
 Task temp = *head*;  
 *while* (temp != *null*) {  
 System.***out***.println(temp);  
 temp = temp.next;  
 }  
 }  
 *public static void* main(String[] args) {  
 *addTask*(1, "Design UI", "Pending");  
 *addTask*(2, "Develop Backend", "In Progress");  
 *addTask*(3, "Testing", "Pending");  
  
 *displayTasks*();  
 *searchTask*(2);  
 *deleteTask*(2);  
 *displayTasks*();  
 }  
}

Output:



**Time** complexity of each **Operation:**

* **Add:** O(1) at head
* **Search:** O(n)
* **Traverse:** O(n)
* **Delete:** O(n)

**Linked List Advantages:**

* **Dynamic memory usage (no pre-sizing needed)**
* **Efficient insert/delete (O(1) at head)**

**Exercise 6: Library Management System**

**Linear Search**

* **Checks each element one by one.**
* **Works on unsorted arrays.**
* **Time Complexity: O(n)**

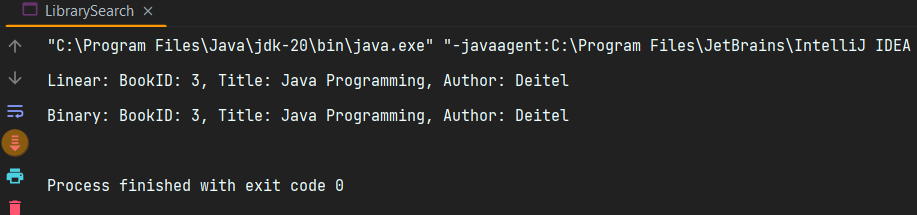
**Binary Search**

* **Repeatedly halves a sorted array to find the target.**
* **Only works on sorted arrays.**
* **Time Complexity: O(log n)**

**Code:**

*import* java.util.Arrays;  
*import* java.util.*Comparator*;  
*class* Book {  
 *int* bookId;  
 String title;  
 String author;  
 Book(*int* id, String title, String author) {  
 *this*.bookId = id;  
 *this*.title = title;  
 *this*.author = author;  
 }  
 *public* String toString() {  
 *return* "BookID: " + bookId + ", Title: " + title + ", Author: " + author;  
 }  
}  
*public class* LibrarySearch {  
 *static* Book[] *books* = {  
 *new* Book(1, "Data Structures", "Narasimha"),  
 *new* Book(2, "Operating Systems", "Galvin"),  
 *new* Book(3, "Java Programming", "Deitel")  
 };  
 *public static int* linearSearch(String title) {  
 *for* (*int* i = 0; i < *books*.length; i++) {  
 *if* (*books*[i].title.equalsIgnoreCase(title)) *return* i;  
 }  
 *return* -1;  
 }  
 *public static int* binarySearch(Book[] sortedBooks, String title) {  
 *int* low = 0, high = sortedBooks.length - 1;  
 *while* (low <= high) {  
 *int* mid = (low + high) / 2;  
 *int* cmp = sortedBooks[mid].title.compareToIgnoreCase(title);  
 *if* (cmp == 0) *return* mid;  
 *else if* (cmp < 0) low = mid + 1;  
 *else* high = mid - 1;  
 }  
 *return* -1;  
 }  
 *public static void* main(String[] args) {  
 *int* idx1 = *linearSearch*("Java Programming");  
 System.***out***.println("Linear: " + (idx1 != -1 ? *books*[idx1] : "Not found"));  
  
 Arrays.*sort*(*books*, *Comparator*.*comparing*(b -> b.title.toLowerCase()));  
 *int* idx2 = *binarySearch*(*books*, "Java Programming");  
 System.***out***.println("Binary: " + (idx2 != -1 ? *books*[idx2] : "Not found"));  
 }  
}

**Output:**

****

Time Complexity:

Algorithm Time

Linear Search O(n)

Binary Search O(log n)

**When to Use Linear vs Binary Search:**

* **Linear Search**: For unsorted or small datasets.
* **Binary Search**: Fast but needs sorted input.

**Exercise 7: Financial Forecasting**

**Recursion:**

A technique where a function calls itself to solve smaller subproblems. It stops when it hits a **base case**.

**Why It's Useful:**

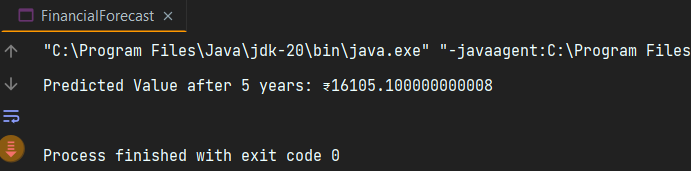
Simplifies complex problems like:

* Tree/graph traversal
* Factorial/Fibonacci
* Backtracking (e.g., puzzles)

**Code**:

*public class* FinancialForecast {  
 *public static double* forecast(*double* value, *double* growthRate, *int* years) {  
 *if* (years == 0) *return* value;  
 *return forecast*(value \* (1 + growthRate), growthRate, years - 1);  
 }  
 *public static void* main(String[] args) {  
 *double* initial = 10000;  
 *double* growthRate = 0.10;  
 *int* years = 5;  
 *double* future = *forecast*(initial, growthRate, years);  
 System.***out***.println("Predicted Value after " + years + " years: ₹" + future);  
 }  
}

Output:



**Time Complexity:**

* Recursive Time: O(n)

**How to Optimize Recursive Solutions:**

1. **Memoization** – Cache results to avoid recomputing.
2. **Tabulation** – Use loops instead of recursion (bottom-up).
3. **Tail Recursion** – Put the recursive call last for stack efficiency.
4. **Limit Depth** – Switch to iteration if calls get too deep.