**Exercise 1: Inventory Management System**

**PROGRAM:**

**using System;**

**using System.Collections;**

**class Program**

**{**

**static void Main()**

**{**

**Console.WriteLine("Welcome to your simple Inventory App (using Hashtable)!");**

**var inv = new Hashtable();**

**while (true)**

**{**

**Console.WriteLine("\nWhat would you like to do?");**

**Console.WriteLine("1.Add Items");**

**Console.WriteLine("2.View all");**

**Console.WriteLine("3.Update");**

**Console.WriteLine("4.Delete");**

**Console.WriteLine("5.Exit");**

**Console.Write("> ");**

**var choice = Console.ReadLine();**

**switch (choice)**

**{**

**case "1": AddItem(inv); break;**

**case "2": ViewItems(inv); break;**

**case "3": UpdateItem(inv); break;**

**case "4": DeleteItem(inv); break;**

**case "5": Console.WriteLine("Bye!"); return;**

**default: Console.WriteLine("Please pick 1 to 5, thanks."); break;**

**}**

**}**

**}**

**static void AddItem(Hashtable inv)**

**{**

**Console.Write("Enter ID (number): ");**

**if (!int.TryParse(Console.ReadLine(), out int id))**

**{**

**Console.WriteLine("That’s not a valid ID.");**

**return;**

**}**

**if (inv.ContainsKey(id))**

**{**

**Console.WriteLine("Oops, that ID already exists!");**

**return;**

**}**

**Console.Write("Enter name: ");**

**var name = Console.ReadLine() ?? "";**

**Console.Write("Enter quantity: ");**

**if (!int.TryParse(Console.ReadLine(), out int qty))**

**{**

**Console.WriteLine("Invalid quantity.");**

**return;**

**}**

**Console.Write("Enter price: ");**

**if (!decimal.TryParse(Console.ReadLine(), out decimal price))**

**{**

**Console.WriteLine("Invalid price.");**

**return;**

**}**

**// storing a tuple of name, qty, price**

**inv[id] = Tuple.Create(name, qty, price);**

**Console.WriteLine("Item added!");**

**}**

**static void ViewItems(Hashtable inv)**

**{**

**Console.WriteLine("\n--- Your Inventory ---");**

**if (inv.Count == 0)**

**{**

**Console.WriteLine("(empty)");**

**return;**

**}**

**foreach (DictionaryEntry e in inv)**

**{**

**var t = (Tuple<string, int, decimal>)e.Value;**

**Console.WriteLine($"{e.Key}: {t.Item1}, Qty={t.Item2}, ₹{t.Item3:F2}");**

**}**

**}**

**static void UpdateItem(Hashtable inv)**

**{**

**Console.Write("Enter ID to update: ");**

**if (!int.TryParse(Console.ReadLine(), out int id) || !inv.ContainsKey(id))**

**{**

**Console.WriteLine("No such item.");**

**return;**

**}**

**var old = (Tuple<string, int, decimal>)inv[id];**

**Console.Write($"New name (or Enter to keep '{old.Item1}'): ");**

**var name = Console.ReadLine();**

**Console.Write($"New qty (or Enter for {old.Item2}): ");**

**var qtyStr = Console.ReadLine();**

**Console.Write($"New price (or Enter for ₹{old.Item3:F2}): ");**

**var priceStr = Console.ReadLine();**

**var finalName = string.IsNullOrWhiteSpace(name) ? old.Item1 : name;**

**var finalQty = int.TryParse(qtyStr, out var qv) ? qv : old.Item2;**

**var finalPrice = decimal.TryParse(priceStr, out var pv) ? pv : old.Item3;**

**inv[id] = Tuple.Create(finalName, finalQty, finalPrice);**

**Console.WriteLine("Item updated.");**

**}**

**static void DeleteItem(Hashtable inv)**

**{**

**Console.Write("Enter ID to delete: ");**

**if (!int.TryParse(Console.ReadLine(), out int id) || !inv.ContainsKey(id))**

**{**

**Console.WriteLine("No such item.");**

**return;**

**}**

**inv.Remove(id);**

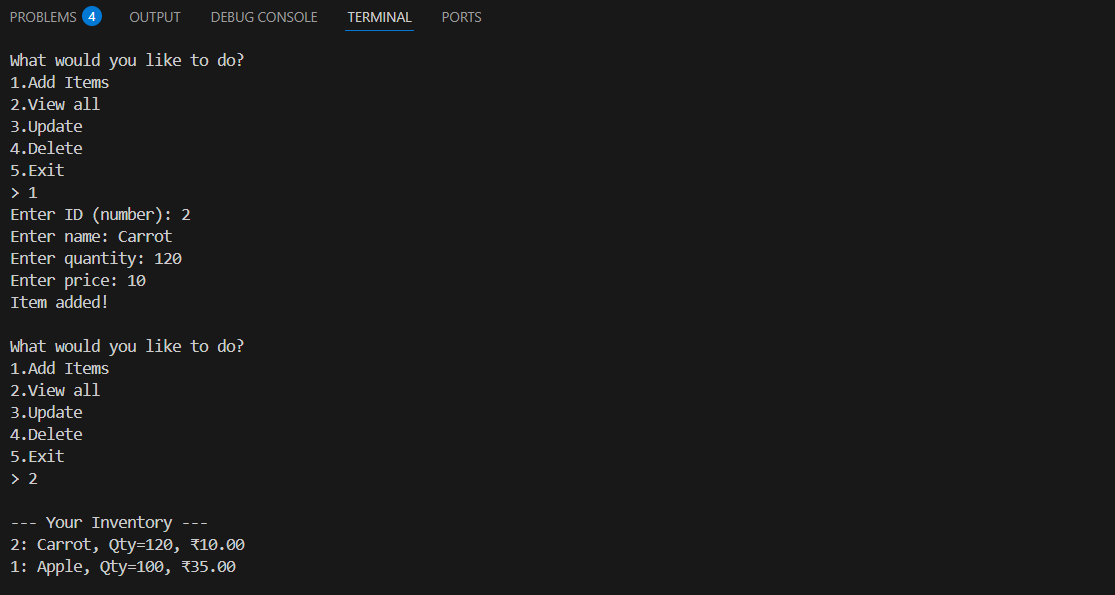
**Console.WriteLine("Item deleted.");**

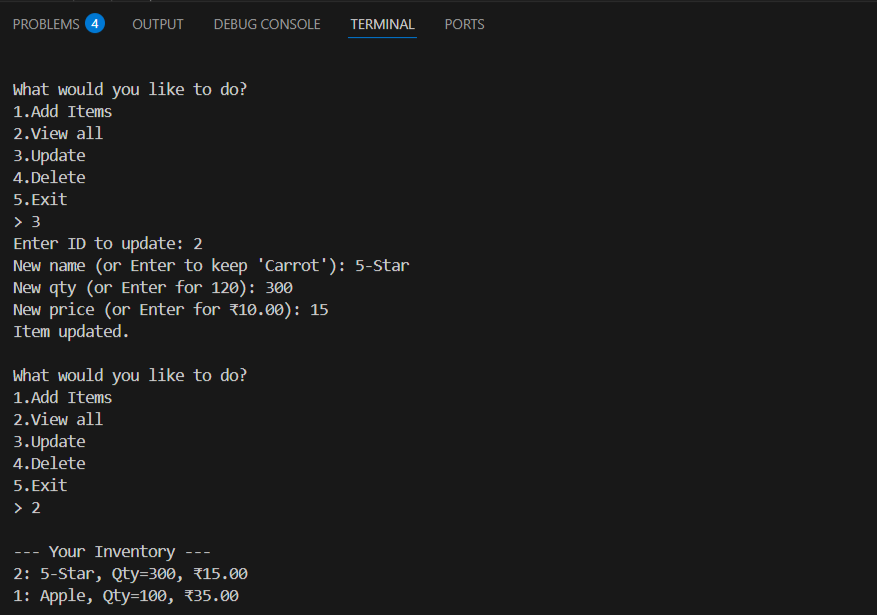
**}**

**}**

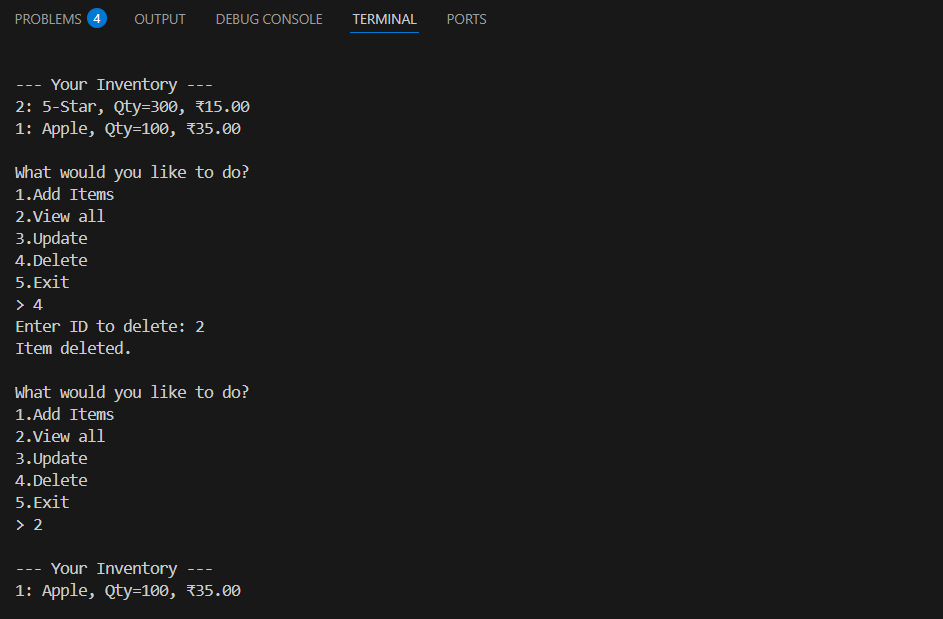
**OUTPUT:**

**Add Items:**



**Update:** 

**Delete:**



**ANALYSIS:**

**Add Operation:**

* **Best Case (O(1)): The ID is not already in the hashtable, and the item is added directly.**
* **Average Case (O(1)): On average, adding an item is quick unless there are many collisions.**
* **Worst Case (O(n)): If many items hash to the same location (rare), insertion takes longer.**

**View All:**

* **Best/Average/Worst Case (O(n)): All items are printed, so time depends on the number of items.**

**Update Operation (search + write):**

* **Best Case (O(1)): The item exists and is found immediately.**
* **Average Case (O(1)): Hash lookups are generally fast, even with many items.**
* **Worst Case (O(n)): If there's a hash collision issue or the item is not found.**

**Delete Operation:**

* **Best Case (O(1)): The ID exists and is deleted right away.**
* **Average Case (O(1)): Deletion is fast unless there's a collision.**
* **Worst Case (O(n)): Slow only if many items collide at the same hash index.**

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

**PROGRAM:**

**using System;**

**using System.Collections.Generic;**

**namespace ECommerceSearch**

**{**

**//1.Product Class**

**class Product**

**{**

**public int ProductId;**

**public string ProductName;**

**public string Category;**

**public Product(int id, string name, string cat)**

**{**

**ProductId = id;**

**ProductName = name;**

**Category = cat;**

**}**

**public void Display()**

**{**

**Console.WriteLine($"ID: {ProductId}, Name: {ProductName}, Category: {Category}");**

**}**

**}**

**//Program class**

**class Program**

**{**

**static void Main(string[] args)**

**{**

**// 1. Create a list of products**

**List<Product> products = new List<Product>()**

**{**

**new Product(101, "Laptop", "Electronics"),**

**new Product(102, "Shirt", "Clothing"),**

**new Product(103, "Phone", "Electronics"),**

**new Product(104, "Shoes", "Footwear")**

**};**

**Console.WriteLine("=== E-Commerce Product Search ===");**

**// 2. Linear search**

**Console.Write("Enter product name (linear search): ");**

**string? searchName = Console.ReadLine();**

**if (!string.IsNullOrEmpty(searchName))**

**{**

**Product? found1 = LinearSearch(products, searchName);**

**if (found1 != null) found1.Display();**

**else Console.WriteLine("Product not found.");**

**}**

**// 3. Dictionary lookup (fast!)**

**var dict = new Dictionary<string, Product>(StringComparer.OrdinalIgnoreCase);**

**foreach (var p in products)**

**dict[p.ProductName] = p;**

**Console.Write("\nEnter product name (dictionary lookup): ");**

**searchName = Console.ReadLine();**

**if (!string.IsNullOrEmpty(searchName))**

**{**

**if (dict.TryGetValue(searchName, out var found2))**

**found2.Display();**

**else**

**Console.WriteLine("Product not found.");**

**}**

**// 4. Binary search**

**products.Sort((a, b) => a.ProductName**

**.CompareTo(b.ProductName));**

**Console.Write("\nEnter product name (binary search): ");**

**searchName = Console.ReadLine();**

**if (!string.IsNullOrEmpty(searchName))**

**{**

**Product? found3 = BinarySearch(products, searchName);**

**if (found3 != null) found3.Display();**

**else Console.WriteLine("Product not found.");**

**}**

**// 5. Explain time complexity**

**Console.WriteLine("\n--- Time Complexity ---");**

**Console.WriteLine("Linear Search: O(n)");**

**Console.WriteLine("Dictionary Lookup: O(1) average (hash table) :contentReference[oaicite:1]{index=1}");**

**Console.WriteLine("Binary Search: O(log n) on sorted data (uses built‑in BinarySearch) :contentReference[oaicite:2]{index=2}");**

**}**

**static Product? LinearSearch(List<Product> list, string name)**

**{**

**foreach (var p in list)**

**{**

**if (p.ProductName.Equals(name, StringComparison.OrdinalIgnoreCase))**

**return p;**

**}**

**return null;**

**}**

**static Product? BinarySearch(List<Product> sortedList, string name)**

**{**

**int idx = sortedList.BinarySearch(**

**new Product(0, name, ""),**

**Comparer<Product>.Create((a, b) =>**

**a.ProductName**

**.CompareTo(b.ProductName))**

**);**

**if (idx >= 0) return sortedList[idx];**

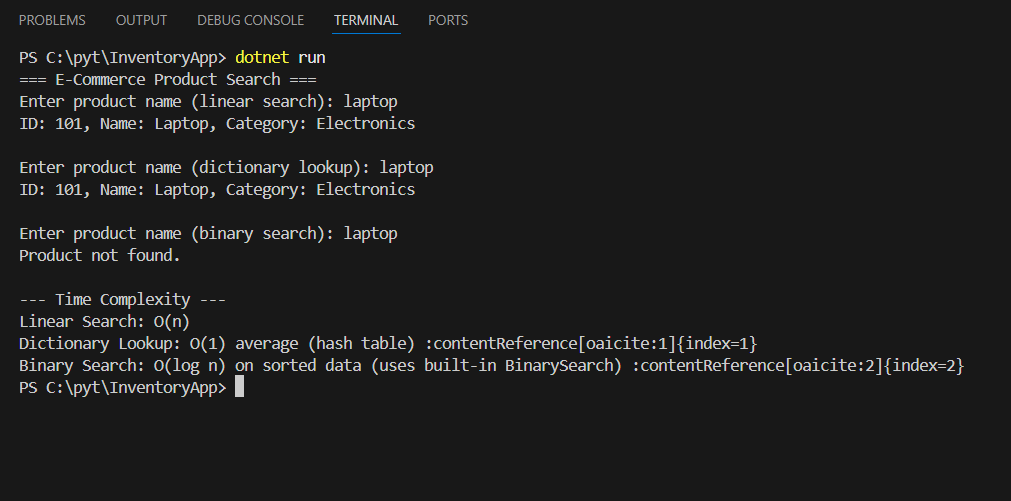
**return null;**

**}**

**}**

**}**

**OUTPUT:**

****

**ANALYSIS:**

**Linear Search:**

* **Best Case (O(1)): The item is found at the beginning of the list.**
* **Average Case (O(n)): The item is located somewhere in the middle.**
* **Worst Case (O(n)): The item is at the end or not present in the list at all.**

**Binary Search (works only on sorted data):**

* **Best Case (O(1)): The item is found at the middle of the list in the first check.**
* **Average Case (O(log n)): The item is found after dividing the list a few times.**
* **Worst Case (O(log n)): The item is found at the end of the search or not found at all.**

**Exercise 7: Financial Forecasting**

**PROGRAM:**

**using System;**

**class FinancialForecast**

**{**

**// === 1. Recursive method ================================================**

**// This method predicts future money after a number of years.**

**// - Base case: if years == 0, we've done all growth—just return current value.**

**// - Recursive case: grow for one year, then call this method again with years-1.**

**static double PredictFutureValue(double currentValue, double growthRate, int years)**

**{**

**// Base case: stop when no years are left**

**if (years == 0)**

**return currentValue;**

**// Apply growth for one year**

**double nextValue = currentValue \* (1 + growthRate);**

**// Recursive case: call the method again for the rest of the years**

**return PredictFutureValue(nextValue, growthRate, years - 1);**

**}**

**// === 2. Beginner-friendly main method ================================**

**static void Main()**

**{**

**Console.WriteLine("=== Financial Forecast Tool ===\n");**

**// Get current amount from user**

**Console.Write("Enter current value: ");**

**double currentValue = Convert.ToDouble(Console.ReadLine());**

**// Get growth rate**

**Console.Write("Enter annual growth rate (e.g. 0.05 for 5%): ");**

**double growthRate = Convert.ToDouble(Console.ReadLine());**

**// Get number of years**

**Console.Write("Enter number of years to forecast: ");**

**int years = Convert.ToInt32(Console.ReadLine());**

**// === 3. Call the recursive method and show result ===**

**double futureValue = PredictFutureValue(currentValue, growthRate, years);**

**Console.WriteLine($"\n✅ After {years} years, you'll have: ₹{futureValue:F2}");**

**// === 4. Explain recursion and time complexity =====================**

**Console.WriteLine("\n--- About the recursion behind the scenes ---");**

**Console.WriteLine("1. Base case: 'if (years == 0)' → this stops the recursion.");**

**Console.WriteLine("2. Recursive case: method calls itself with years-1.");**

**Console.WriteLine($"3. Total calls done ≈ {years + 1} → time complexity is O(n).");**

**Console.WriteLine("   (Because each call is one year, and work per call is constant)");**

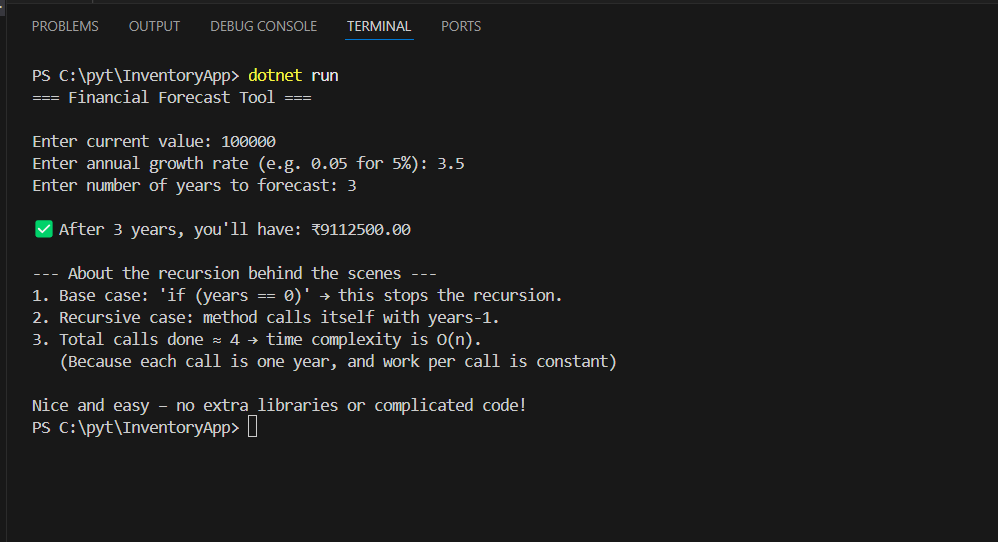
**Console.WriteLine("\nNice and easy – no extra libraries or complicated code!");**

**}**

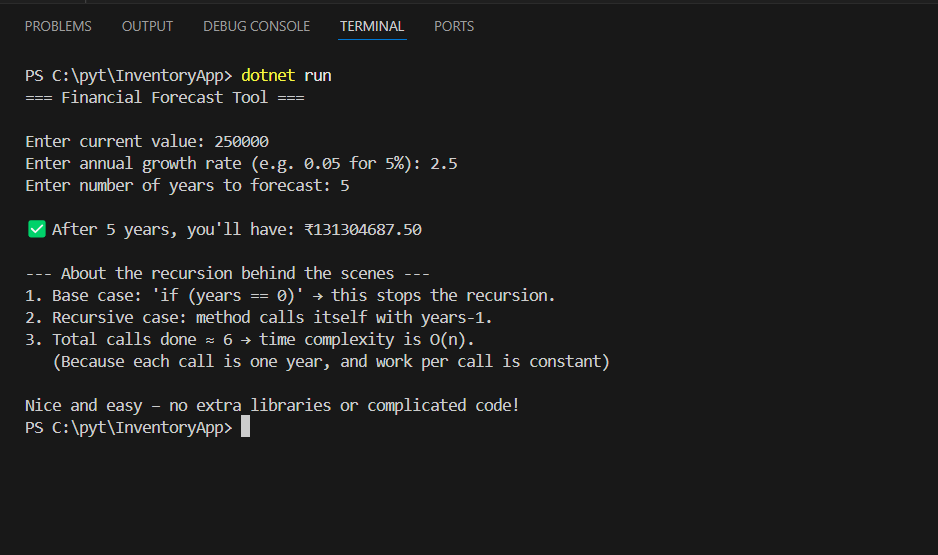
**}**

**OUTPUT:**

**1:**

****

**2:**



**RECURSION:**

**Recursion is a programming technique where a function calls itself in order to solve a problem. Each time the function calls itself, it works on a smaller part of the problem until it reaches a condition where it stops calling itself. This stopping point is called the base case.**

**Recursion is useful for problems that can be broken down into similar smaller problems. It helps simplify code and makes it more readable, especially in tasks like mathematical calculations (e.g., factorial, Fibonacci), tree or graph traversal, and searching or sorting.**

**Instead of writing long loops or complex logic, recursion allows the same logic to be repeated in a cleaner and more organized way. However, if not written carefully, it can lead to performance issues or stack overflow errors.**

**ANALYSIS:**

**Time Complexity :**

* **Best Case (O(1)): When years = 0, the base case is hit immediately.**
* **Average/Worst Case (O(n)): When years > 0, the function calls itself n times where n is the number of years.**

**Optimization :**

**In this program, recursion works fine because it only calls itself once per year, so it's not too heavy. But if we increase the number of years to a very large value, it could lead to stack overflow or make the program slower.**

**One way to optimize this is to convert the recursive method into an iterative loop using a for loop. This avoids repeated function calls and saves memory. It also runs slightly faster because it doesn’t depend on the call stack.**

**So, while recursion makes the code clean and easy to understand, using iteration can be more efficient for bigger inputs.**