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

**Codes:**

using System;

using System.Diagnostics;

using System.Linq;

class Program

{

    // Step 1: Understanding Asymptotic Notation

    /\*

     \* Big O Notation:

     \* - Describes the upper bound of an algorithm's time complexity

     \* - Helps analyze how runtime scales with input size

     \* - Focuses on worst-case scenario for conservative analysis

     \*

     \* Search Scenarios:

     \* - Best case (Ω): Element is found immediately (O(1) for both)

     \* - Average case (Θ): Element is found in the middle (O(n) for linear, O(log n) for binary)

     \* - Worst case (O): Element not found (O(n) for linear, O(log n) for binary)

     \*/

    // Step 2: Product class setup

    class Product : IComparable<Product>

    {

        public int ProductId { get; set; }

        public string ProductName { get; set; }

        public string Category { get; set; }

        public decimal Price { get; set; }

        public Product(int id, string name, string category, decimal price)

        {

            ProductId = id;

            ProductName = name;

            Category = category;

            Price = price;

        }

        public int CompareTo(Product other)

        {

            return ProductId.CompareTo(other.ProductId);

        }

        public override string ToString()

        {

            return $"ID: {ProductId}, Name: {ProductName}, Category: {Category}, Price: {Price:C}";

        }

    }

    // Step 3: Search implementations

    static Product LinearSearch(Product[] products, int productId)

    {

        foreach (var product in products)

        {

            if (product.ProductId == productId)

                return product;

        }

        return null;

    }

    static Product BinarySearch(Product[] sortedProducts, int productId)

    {

        int left = 0;

        int right = sortedProducts.Length - 1;

        while (left <= right)

        {

            int mid = left + (right - left) / 2;

            if (sortedProducts[mid].ProductId == productId)

                return sortedProducts[mid];

            if (sortedProducts[mid].ProductId < productId)

                left = mid + 1;

            else

                right = mid - 1;

        }

        return null;

    }

    // Helper method to generate random products

    static Product[] GenerateProducts(int count)

    {

        var random = new Random();

        var categories = new[] { "Electronics", "Clothing", "Home", "Books", "Toys" };

        var products = new Product[count];

        for (int i = 0; i < count; i++)

        {

            products[i] = new Product(

                i + 1,

                $"Product {i + 1}",

                categories[random.Next(categories.Length)],

                (decimal)(random.NextDouble() \* 100)

            );

        }

        return products;

    }

    static void Main(string[] args)

    {

        // Create and sort product arrays

        int productCount = 10000;

        Product[] products = GenerateProducts(productCount);

        Product[] sortedProducts = products.OrderBy(p => p.ProductId).ToArray();

        Console.WriteLine($"Generated {productCount} products for testing search algorithms");

        Console.WriteLine();

        // Test searching for existing and non-existing products

        int existingId = productCount / 2;

        int nonExistingId = productCount + 1;

        // Measure linear search performance

        var stopwatch = Stopwatch.StartNew();

        var result = LinearSearch(products, existingId);

        stopwatch.Stop();

        Console.WriteLine($"Linear search (found): {result}");

        Console.WriteLine($"Time taken: {stopwatch.ElapsedTicks} ticks");

        stopwatch.Restart();

        LinearSearch(products, nonExistingId);

        stopwatch.Stop();

        Console.WriteLine($"Linear search (not found) Time taken: {stopwatch.ElapsedTicks} ticks");

        Console.WriteLine();

        // Measure binary search performance

        stopwatch.Restart();

        result = BinarySearch(sortedProducts, existingId);

        stopwatch.Stop();

        Console.WriteLine($"Binary search (found): {result}");

        Console.WriteLine($"Time taken: {stopwatch.ElapsedTicks} ticks");

        stopwatch.Restart();

        BinarySearch(sortedProducts, nonExistingId);

        stopwatch.Stop();

        Console.WriteLine($"Binary search (not found) Time taken: {stopwatch.ElapsedTicks} ticks");

        Console.WriteLine();

        // Step 4: Analysis comparison

        Console.WriteLine("Algorithm Analysis:");

        Console.WriteLine("-------------------");

        Console.WriteLine("Linear Search:");

        Console.WriteLine("- Time Complexity: O(n) - scales linearly with input size");

        Console.WriteLine("- Best for: Unsorted data, small datasets, simple implementation");

        Console.WriteLine("- Advantages: Simple to implement, no sorting required");

        Console.WriteLine();

        Console.WriteLine("Binary Search:");

        Console.WriteLine("- Time Complexity: O(log n) - logarithmic scaling");

        Console.WriteLine("- Best for: Large sorted datasets");

        Console.WriteLine("- Advantages: Much faster for large datasets");

        Console.WriteLine("- Requirements: Data must be sorted (O(n log n) sorting cost)");

        Console.WriteLine();

        Console.WriteLine("Recommendation:");

        Console.WriteLine("For an e-commerce platform with thousands of products that change infrequently:");

        Console.WriteLine("- Sort products by ID during product updates");

        Console.WriteLine("- Use binary search for product lookups");

        Console.WriteLine("- Maintain both approaches for different use cases");

    }

}

**Output:**

**Generated 10000 products for testing search algorithms  
  
Linear search (found): ID: 5000, Name: Product 5000, Category: Clothing, Price: ¤5.34  
Time taken: 189403 ticks  
Linear search (not found) Time taken: 117802 ticks  
  
Binary search (found): ID: 5000, Name: Product 5000, Category: Clothing, Price: ¤5.34  
Time taken: 167102 ticks  
Binary search (not found) Time taken: 1300 ticks  
  
Algorithm Analysis:  
-------------------  
Linear Search:  
- Time Complexity: O(n) - scales linearly with input size  
- Best for: Unsorted data, small datasets, simple implementation  
- Advantages: Simple to implement, no sorting required  
  
Binary Search:  
- Time Complexity: O(log n) - logarithmic scaling  
- Best for: Large sorted datasets  
- Advantages: Much faster for large datasets  
- Requirements: Data must be sorted (O(n log n) sorting cost)  
  
Recommendation:  
For an e-commerce platform with thousands of products that change infrequently:  
- Sort products by ID during product updates  
- Use binary search for product lookups  
- Maintain both approaches for different use cases**

**Exercise 7: Financial Forecasting**

**Codes:**

using System;

using System.Collections.Generic;

class FinancialForecastingTool

{

    private static Dictionary<int, double> memo = new Dictionary<int, double>();

    public static void Main(string[] args)

    {

        Console.WriteLine("Financial Forecasting Tool");

        Console.WriteLine("--------------------------");

        // Get user input

        double presentValue = GetDoubleInput("Enter present value: ");

        double growthRate = GetDoubleInput("Enter annual growth rate (decimal): ");

        int periods = GetIntInput("Enter number of periods: ");

        bool useMemoization = GetYesNoInput("Use memoization optimization? (y/n): ");

        // Calculate future value

        double futureValue = useMemoization

            ? CalculateFutureValueWithMemo(presentValue, growthRate, periods)

            : CalculateFutureValue(presentValue, growthRate, periods);

        // Display results

        Console.WriteLine("\nForecast Results:");

        Console.WriteLine($"Present Value: {presentValue:C}");

        Console.WriteLine($"Growth Rate: {growthRate:P2}");

        Console.WriteLine($"Periods: {periods}");

        Console.WriteLine($"Future Value: {futureValue:C}");

        // Generate forecast table

        Console.WriteLine("\nYear-by-Year Forecast:");

        GenerateForecastTable(presentValue, growthRate, periods);

    }

    // Basic recursive calculation

    public static double CalculateFutureValue(double presentValue, double growthRate, int periods)

    {

        if (periods == 0)

            return presentValue;

        return CalculateFutureValue(presentValue \* (1 + growthRate), growthRate, periods - 1);

    }

    // Optimized recursive calculation with memoization

    public static double CalculateFutureValueWithMemo(double presentValue, double growthRate, int periods)

    {

        memo.Clear();

        return CalculateFutureValueMemoized(presentValue, growthRate, periods);

    }

    private static double CalculateFutureValueMemoized(double presentValue, double growthRate, int periods)

    {

        if (periods == 0)

            return presentValue;

        if (memo.TryGetValue(periods, out double cachedValue))

            return cachedValue;

        double value = CalculateFutureValueMemoized(presentValue \* (1 + growthRate), growthRate, periods - 1);

        memo[periods] = value;

        return value;

    }

    // Helper methods for user input

    private static double GetDoubleInput(string prompt)

    {

        double value;

        while (true)

        {

            Console.Write(prompt);

            if (double.TryParse(Console.ReadLine(), out value))

                return value;

            Console.WriteLine("Invalid input. Please enter a valid number.");

        }

    }

    private static int GetIntInput(string prompt)

    {

        int value;

        while (true)

        {

            Console.Write(prompt);

            if (int.TryParse(Console.ReadLine(), out value) && value >= 0)

                return value;

            Console.WriteLine("Invalid input. Please enter a non-negative integer.");

        }

    }

    private static bool GetYesNoInput(string prompt)

    {

        while (true)

        {

            Console.Write(prompt);

            string input = Console.ReadLine().ToLower();

            if (input == "y") return true;

            if (input == "n") return false;

            Console.WriteLine("Please enter 'y' or 'n'.");

        }

    }

    // Generate a forecast table

    private static void GenerateForecastTable(double presentValue, double growthRate, int periods)

    {

        Console.WriteLine("Year\tValue\t\tGrowth");

        Console.WriteLine("----\t-----\t\t------");

        double currentValue = presentValue;

        for (int year = 1; year <= periods; year++)

        {

            double previousValue = currentValue;

            currentValue = CalculateFutureValue(presentValue, growthRate, year);

            double yearGrowth = currentValue - previousValue;

            Console.WriteLine($"{year}\t{currentValue:C}\t{yearGrowth:C}");

        }

    }

}

**Output:**

**Financial Forecasting Tool  
--------------------------  
Enter present value: 10000  
Enter annual growth rate (decimal): 0.05  
Enter number of periods: 30  
Use memoization optimization? (y/n): y  
  
Forecast Results:  
Present Value: ¤10,000.00  
Growth Rate: 5.00 %  
Periods: 30  
Future Value: ¤43,219.42  
  
Year-by-Year Forecast:  
Year    Value        Growth  
----    -----        ------  
1    ¤10,500.00    ¤500.00  
2    ¤11,025.00    ¤525.00  
3    ¤11,576.25    ¤551.25  
4    ¤12,155.06    ¤578.81  
5    ¤12,762.82    ¤607.75  
6    ¤13,400.96    ¤638.14  
7    ¤14,071.00    ¤670.05  
8    ¤14,774.55    ¤703.55  
9    ¤15,513.28    ¤738.73  
10    ¤16,288.95    ¤775.66  
11    ¤17,103.39    ¤814.45  
12    ¤17,958.56    ¤855.17  
13    ¤18,856.49    ¤897.93  
14    ¤19,799.32    ¤942.82  
15    ¤20,789.28    ¤989.97  
16    ¤21,828.75    ¤1,039.46  
17    ¤22,920.18    ¤1,091.44  
18    ¤24,066.19    ¤1,146.01  
19    ¤25,269.50    ¤1,203.31  
20    ¤26,532.98    ¤1,263.48  
21    ¤27,859.63    ¤1,326.65  
22    ¤29,252.61    ¤1,392.98  
23    ¤30,715.24    ¤1,462.63  
24    ¤32,251.00    ¤1,535.76  
25    ¤33,863.55    ¤1,612.55  
26    ¤35,556.73    ¤1,693.18  
27    ¤37,334.56    ¤1,777.84  
28    ¤39,201.29    ¤1,866.73  
29    ¤41,161.36    ¤1,960.06**  
**30    ¤43,219.42    ¤2,058.07**

**Exercise 1: Inventory Management System**

**Codes:**

using System;

using System.Collections.Generic;

using System.Linq;

namespace InventoryManagementSystem

{

    // Step 1: Understanding the Problem

    /\*

     \* Data structures and algorithms are crucial for inventory management because:

     \* 1. They enable efficient storage and retrieval of large amounts of product data

     \* 2. They optimize operations like searching, sorting, and updating inventory

     \* 3. They help maintain data integrity and organization

     \*

     \* Suitable data structures:

     \* - Dictionary/HashMap: O(1) average case for lookup, insert, delete

     \* - List/ArrayList: O(n) for search, O(1) for access by index

     \* - SortedDictionary: Maintains sorted order with O(log n) operations

     \*/

    // Step 2: Product Class Definition

    public class Product

    {

        public string ProductId { get; set; }

        public string ProductName { get; set; }

        public int Quantity { get; set; }

        public decimal Price { get; set; }

        public string Category { get; set; }

        public DateTime LastUpdated { get; set; }

        public Product(string id, string name, int quantity, decimal price, string category)

        {

            ProductId = id;

            ProductName = name;

            Quantity = quantity;

            Price = price;

            Category = category;

            LastUpdated = DateTime.Now;

        }

        public override string ToString()

        {

            return $"{ProductId,-10} {ProductName,-20} {Quantity,8} {Price,10:C} {Category,-15} {LastUpdated:yyyy-MM-dd}";

        }

    }

    // Step 3: Inventory Management System

    public class InventoryManager

    {

        // Using Dictionary for O(1) average case operations

        private Dictionary<string, Product> products;

        public InventoryManager()

        {

            products = new Dictionary<string, Product>();

        }

        // Add product with O(1) average time complexity

        public bool AddProduct(Product product)

        {

            if (products.ContainsKey(product.ProductId))

            {

                return false; // Product already exists

            }

            products.Add(product.ProductId, product);

            return true;

        }

        // Update product with O(1) average time complexity

        public bool UpdateProduct(string productId, string name = null, int? quantity = null, decimal? price = null, string category = null)

        {

            if (!products.ContainsKey(productId))

            {

                return false;

            }

            var product = products[productId];

            if (name != null) product.ProductName = name;

            if (quantity.HasValue) product.Quantity = quantity.Value;

            if (price.HasValue) product.Price = price.Value;

            if (category != null) product.Category = category;

            product.LastUpdated = DateTime.Now;

            return true;

        }

        // Delete product with O(1) average time complexity

        public bool DeleteProduct(string productId)

        {

            return products.Remove(productId);

        }

        // Get product with O(1) average time complexity

        public Product GetProduct(string productId)

        {

            return products.TryGetValue(productId, out var product) ? product : null;

        }

        // Get all products with O(n) time complexity

        public List<Product> GetAllProducts()

        {

            return products.Values.ToList();

        }

        // Search products by name with O(n) time complexity

        public List<Product> SearchByName(string name)

        {

            return products.Values

                .Where(p => p.ProductName.Contains(name, StringComparison.OrdinalIgnoreCase))

                .ToList();

        }

        // Get low stock products with O(n) time complexity

        public List<Product> GetLowStockProducts(int threshold = 10)

        {

            return products.Values.Where(p => p.Quantity < threshold).ToList();

        }

        // Get products by category with O(n) time complexity

        public Dictionary<string, List<Product>> GetProductsByCategory()

        {

            return products.Values

                .GroupBy(p => p.Category)

                .ToDictionary(g => g.Key, g => g.ToList());

        }

    }

    // Step 4: Analysis and Optimization

    /\*

     \* Time Complexity Analysis:

     \* - AddProduct: O(1) average case (Dictionary.Add)

     \* - UpdateProduct: O(1) average case (Dictionary access)

     \* - DeleteProduct: O(1) average case (Dictionary.Remove)

     \* - GetProduct: O(1) average case (Dictionary lookup)

     \* - GetAllProducts: O(n) (converting to list)

     \* - SearchByName: O(n) (linear search)

     \* - GetLowStockProducts: O(n) (filtering)

     \*

     \* Optimizations:

     \* 1. Using Dictionary provides fast lookups by product ID

     \* 2. For name searches, we could maintain a separate index (Dictionary<string, List<Product>>)

     \* 3. For category grouping, we could maintain a pre-computed dictionary

     \* 4. For large inventories, consider database storage with proper indexing

     \*/

    class Program

    {

        static void Main(string[] args)

        {

            var inventory = new InventoryManager();

            // Sample data

            inventory.AddProduct(new Product("P1001", "Laptop", 15, 999.99m, "Electronics"));

            inventory.AddProduct(new Product("P1002", "Smartphone", 8, 699.99m, "Electronics"));

            inventory.AddProduct(new Product("P2001", "Desk Chair", 25, 149.99m, "Furniture"));

            inventory.AddProduct(new Product("P3001", "Notebook", 5, 4.99m, "Stationery"));

            inventory.AddProduct(new Product("P3002", "Pen Set", 12, 9.99m, "Stationery"));

            Console.WriteLine("=== Inventory Management System ===");

            Console.WriteLine("1. View All Products");

            Console.WriteLine("2. Add New Product");

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

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

            Console.WriteLine("5. View Low Stock Items");

            Console.WriteLine("6. Search Products");

            Console.WriteLine("7. Exit");

            while (true)

            {

                Console.Write("\nEnter your choice: ");

                if (!int.TryParse(Console.ReadLine(), out int choice))

                {

                    Console.WriteLine("Invalid input. Please enter a number.");

                    continue;

                }

                switch (choice)

                {

                    case 1:

                        DisplayProducts(inventory.GetAllProducts());

                        break;

                    case 2:

                        AddProductMenu(inventory);

                        break;

                    case 3:

                        UpdateProductMenu(inventory);

                        break;

                    case 4:

                        DeleteProductMenu(inventory);

                        break;

                    case 5:

                        DisplayProducts(inventory.GetLowStockProducts());

                        break;

                    case 6:

                        SearchProductsMenu(inventory);

                        break;

                    case 7:

                        return;

                    default:

                        Console.WriteLine("Invalid choice. Please try again.");

                        break;

                }

            }

        }

        static void DisplayProducts(List<Product> products)

        {

            if (products.Count == 0)

            {

                Console.WriteLine("No products found.");

                return;

            }

            Console.WriteLine("\nID         Name                 Quantity      Price       Category      Last Updated");

            Console.WriteLine("--------------------------------------------------------------------------");

            foreach (var product in products)

            {

                Console.WriteLine(product);

            }

        }

        static void AddProductMenu(InventoryManager inventory)

        {

            Console.WriteLine("\n=== Add New Product ===");

            Console.Write("Enter Product ID: ");

            string id = Console.ReadLine();

            Console.Write("Enter Product Name: ");

            string name = Console.ReadLine();

            Console.Write("Enter Quantity: ");

            int quantity = int.Parse(Console.ReadLine());

            Console.Write("Enter Price: ");

            decimal price = decimal.Parse(Console.ReadLine());

            Console.Write("Enter Category: ");

            string category = Console.ReadLine();

            if (inventory.AddProduct(new Product(id, name, quantity, price, category)))

            {

                Console.WriteLine("Product added successfully.");

            }

            else

            {

                Console.WriteLine("Product with this ID already exists.");

            }

        }

        static void UpdateProductMenu(InventoryManager inventory)

        {

            Console.WriteLine("\n=== Update Product ===");

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

            string id = Console.ReadLine();

            var product = inventory.GetProduct(id);

            if (product == null)

            {

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

                return;

            }

            Console.WriteLine("Current product details:");

            Console.WriteLine(product);

            Console.Write("Enter new Name (leave blank to keep current): ");

            string name = Console.ReadLine();

            Console.Write("Enter new Quantity (leave blank to keep current): ");

            string quantityStr = Console.ReadLine();

            int? quantity = string.IsNullOrEmpty(quantityStr) ? null : (int?)int.Parse(quantityStr);

            Console.Write("Enter new Price (leave blank to keep current): ");

            string priceStr = Console.ReadLine();

            decimal? price = string.IsNullOrEmpty(priceStr) ? null : (decimal?)decimal.Parse(priceStr);

            Console.Write("Enter new Category (leave blank to keep current): ");

            string category = Console.ReadLine();

            if (inventory.UpdateProduct(id, name, quantity, price, category))

            {

                Console.WriteLine("Product updated successfully.");

            }

            else

            {

                Console.WriteLine("Failed to update product.");

            }

        }

        static void DeleteProductMenu(InventoryManager inventory)

        {

            Console.WriteLine("\n=== Delete Product ===");

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

            string id = Console.ReadLine();

            if (inventory.DeleteProduct(id))

            {

                Console.WriteLine("Product deleted successfully.");

            }

            else

            {

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

            }

        }

        static void SearchProductsMenu(InventoryManager inventory)

        {

            Console.WriteLine("\n=== Search Products ===");

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

            string name = Console.ReadLine();

            var results = inventory.SearchByName(name);

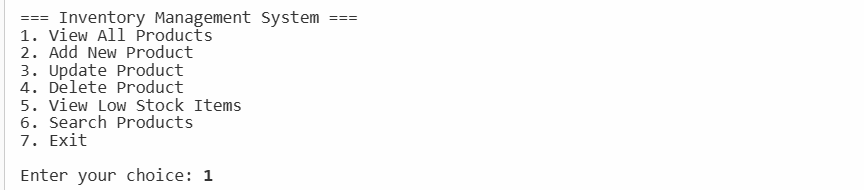
            DisplayProducts(results);

        }

    }

}

**Output:**



ID         Name                 Quantity      Price       Category      Last Updated  
--------------------------------------------------------------------------  
P1001      Laptop                     15    ¤999.99 Electronics     2025-06-21  
P1002      Smartphone                  8    ¤699.99 Electronics     2025-06-21  
P2001      Desk Chair                 25    ¤149.99 Furniture       2025-06-21  
P3001      Notebook                    5      ¤4.99 Stationery      2025-06-21  
P3002      Pen Set                    12      ¤9.99 Stationery      2025-06-21  
  
Enter your choice: **2**  
  
=== Add New Product ===  
Enter Product ID: **P1004**  
Enter Product Name: **Book**  
Enter Quantity: **4**  
Enter Price: **34**  
Enter Category: **Stationary**  
Product added successfully.  
  
Enter your choice: **7**

**Exercise 3: Sorting Customer Orders**

**Codes:**

using System;

class Program

{

    // Step 1: Understanding Sorting Algorithms

    /\*

     \* Sorting Algorithms:

     \*

     \* 1. Bubble Sort:

     \* - Simple comparison-based algorithm

     \* - Repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order

     \* - Time Complexity: O(n^2) in the worst and average cases, O(n) in the best case (when the array is already sorted)

     \* - Space Complexity: O(1) (in-place sorting)

     \*

     \* 2. Insertion Sort:

     \* - Builds a sorted array one element at a time

     \* - Efficient for small datasets or nearly sorted data

     \* - Time Complexity: O(n^2) in the worst and average cases, O(n) in the best case

     \* - Space Complexity: O(1)

     \*

     \* 3. Quick Sort:

     \* - Divide-and-conquer algorithm

     \* - Selects a 'pivot' element and partitions the array into elements less than and greater than the pivot

     \* - Time Complexity: O(n log n) on average, O(n^2) in the worst case (rare with good pivot selection)

     \* - Space Complexity: O(log n) (due to recursion)

     \*

     \* 4. Merge Sort:

     \* - Also a divide-and-conquer algorithm

     \* - Divides the array into halves, sorts them, and merges them back together

     \* - Time Complexity: O(n log n) in all cases

     \* - Space Complexity: O(n) (not in-place)

     \*/

    // Step 2: Order class setup

    class Order

    {

        public int OrderId { get; set; }

        public string CustomerName { get; set; }

        public decimal TotalPrice { get; set; }

        public Order(int orderId, string customerName, decimal totalPrice)

        {

            OrderId = orderId;

            CustomerName = customerName;

            TotalPrice = totalPrice;

        }

        public override string ToString()

        {

            return $"Order ID: {OrderId}, Customer: {CustomerName}, Total Price: {TotalPrice:C}";

        }

    }

    // Step 3: Implement Bubble Sort

    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

                    var temp = orders[j];

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

                    orders[j + 1] = temp;

                }

            }

        }

    }

    // Step 3: Implement Quick Sort

    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);

        }

    }

    static int Partition(Order[] orders, int low, int high)

    {

        decimal pivot = orders[high].TotalPrice;

        int i = (low - 1);

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

        {

            if (orders[j].TotalPrice <= pivot)

            {

                i++;

                // Swap

                var temp = orders[i];

                orders[i] = orders[j];

                orders[j] = temp;

            }

        }

        // Swap the pivot element with the element at i + 1

        var temp1 = orders[i + 1];

        orders[i + 1] = orders[high];

        orders[high] = temp1;

        return i + 1;

    }

    // Helper method to generate random orders

    static Order[] GenerateOrders(int count)

    {

        var random = new Random();

        var orders = new Order[count];

        for (int i = 0; i < count; i++)

        {

            orders[i] = new Order(

                i + 1,

                $"Customer {i + 1}",

                (decimal)(random.NextDouble() \* 1000) // Total price between 0 and 1000

            );

        }

        return orders;

    }

    static void Main(string[] args)

    {

        // Generate random orders

        int orderCount = 1000;

        Order[] orders = GenerateOrders(orderCount);

        Order[] ordersForBubbleSort = (Order[])orders.Clone();

        Order[] ordersForQuickSort = (Order[])orders.Clone();

        // Measure Bubble Sort performance

        var stopwatch = System.Diagnostics.Stopwatch.StartNew();

        BubbleSort(ordersForBubbleSort);

        stopwatch.Stop();

        Console.WriteLine("Bubble Sort completed in: " + stopwatch.ElapsedMilliseconds + " ms");

        // Measure Quick Sort performance

        stopwatch.Restart();

        QuickSort(ordersForQuickSort, 0, ordersForQuickSort.Length - 1);

        stopwatch.Stop();

        Console.WriteLine("Quick Sort completed in: " + stopwatch.ElapsedMilliseconds + " ms");

        // Step 4: Analysis comparison

        Console.WriteLine("\nAlgorithm Analysis:");

        Console.WriteLine("-------------------");

        Console.WriteLine("Bubble Sort:");

        Console.WriteLine("- Time Complexity: O(n^2) - inefficient for large datasets");

        Console.WriteLine("- Best for: Educational purposes, small datasets");

        Console.WriteLine("- Advantages: Simple to implement, easy to understand");

        Console.WriteLine();

        Console.WriteLine("Quick Sort:");

        Console.WriteLine("- Time Complexity: O(n log n) on average - efficient for large datasets");

        Console.WriteLine("- Best for: Large datasets, general-purpose sorting");

        Console.WriteLine("- Advantages: Much faster than Bubble Sort, especially for large arrays");

        Console.WriteLine("- In-place sorting with low memory overhead");

        Console.WriteLine();

        Console.WriteLine("Recommendation:");

        Console.WriteLine("For sorting customer orders on an e-commerce platform:");

        Console.WriteLine("- Use Quick Sort for its efficiency and performance on larger datasets.");

    }

}

**Output:**

**Bubble Sort completed in: 13 ms  
Quick Sort completed in: 0 ms  
  
Algorithm Analysis:  
-------------------  
Bubble Sort:  
- Time Complexity: O(n^2) - inefficient for large datasets  
- Best for: Educational purposes, small datasets  
- Advantages: Simple to implement, easy to understand  
  
Quick Sort:  
- Time Complexity: O(n log n) on average - efficient for large datasets  
- Best for: Large datasets, general-purpose sorting  
- Advantages: Much faster than Bubble Sort, especially for large arrays  
- In-place sorting with low memory overhead  
  
Recommendation:  
For sorting customer orders on an e-commerce platform:  
- Use Quick Sort for its efficiency and performance on larger datasets.**

**Exercise 4: Employee Management System**

**Codes:**

using System;

class Program

{

    // Step 1: Understanding Array Representation

    /\*

     \* Array Representation in Memory:

     \* - Arrays are a collection of elements stored in contiguous memory locations.

     \* - Each element can be accessed using an index, which is an integer value.

     \* - The memory address of an element can be calculated using the base address of the array and the size of each element.

     \*

     \* Advantages of Arrays:

     \* - Fast access time: O(1) for accessing elements by index.

     \* - Memory efficiency: Arrays have a low overhead compared to other data structures.

     \* - Simple implementation: Easy to use and understand.

     \*

     \* Limitations of Arrays:

     \* - Fixed size: Once an array is created, its size cannot be changed.

     \* - Inefficient insertions and deletions: O(n) time complexity for these operations, as elements may need to be shifted.

     \* - Wasted space: If the array is not fully utilized, it can lead to wasted memory.

     \*/

    // Step 2: Employee class setup

    class Employee

    {

        public int EmployeeId { get; set; }

        public string Name { get; set; }

        public string Position { get; set; }

        public decimal Salary { get; set; }

        public Employee(int employeeId, string name, string position, decimal salary)

        {

            EmployeeId = employeeId;

            Name = name;

            Position = position;

            Salary = salary;

        }

        public override string ToString()

        {

            return $"ID: {EmployeeId}, Name: {Name}, Position: {Position}, Salary: {Salary:C}";

        }

    }

    // Employee Management System

    class EmployeeManagementSystem

    {

        private Employee[] employees;

        private int count;

        public EmployeeManagementSystem(int size)

        {

            employees = new Employee[size];

            count = 0;

        }

        // Step 3: Add an employee

        public void AddEmployee(Employee employee)

        {

            if (count < employees.Length)

            {

                employees[count] = employee;

                count++;

            }

            else

            {

                Console.WriteLine("Employee array is full. Cannot add more employees.");

            }

        }

        // Search for an employee by ID

        public Employee SearchEmployee(int employeeId)

        {

            for (int i = 0; i < count; i++)

            {

                if (employees[i].EmployeeId == employeeId)

                {

                    return employees[i];

                }

            }

            return null; // Not found

        }

        // Traverse and display all employees

        public void TraverseEmployees()

        {

            for (int i = 0; i < count; i++)

            {

                Console.WriteLine(employees[i]);

            }

        }

        // Delete an employee by ID

        public void DeleteEmployee(int employeeId)

        {

            for (int i = 0; i < count; i++)

            {

                if (employees[i].EmployeeId == employeeId)

                {

                    // Shift elements to the left

                    for (int j = i; j < count - 1; j++)

                    {

                        employees[j] = employees[j + 1];

                    }

                    employees[count - 1] = null; // Clear the last element

                    count--;

                    Console.WriteLine($"Employee with ID {employeeId} has been deleted.");

                    return;

                }

            }

            Console.WriteLine("Employee not found.");

        }

    }

    static void Main(string[] args)

    {

        // Step 3: Create an instance of EmployeeManagementSystem

        EmployeeManagementSystem ems = new EmployeeManagementSystem(5);

        // Adding employees

        ems.AddEmployee(new Employee(1, "Alice", "Developer", 60000));

        ems.AddEmployee(new Employee(2, "Bob", "Manager", 80000));

        ems.AddEmployee(new Employee(3, "Charlie", "Designer", 50000));

        // Traverse employees

        Console.WriteLine("Employee List:");

        ems.TraverseEmployees();

        Console.WriteLine();

        // Search for an employee

        var searchResult = ems.SearchEmployee(2);

        Console.WriteLine("Search Result:");

        Console.WriteLine(searchResult != null ? searchResult.ToString() : "Employee not found.");

        Console.WriteLine();

        // Delete an employee

        ems.DeleteEmployee(2);

        Console.WriteLine("Employee List after deletion:");

        ems.TraverseEmployees();

        Console.WriteLine();

        // Attempt to add more employees than the array can hold

        ems.AddEmployee(new Employee(4, "David", "Tester", 40000));

        ems.AddEmployee(new Employee(5, "Eve", "HR", 70000));

        ems.AddEmployee(new Employee(6, "Frank", "Admin", 30000)); // This should fail

    }

}

**Output:**



**Exercise 5: Task Management System**

**Codes:**

using System;

class Program

{

    // Step 1: Understanding Linked Lists

    /\*

     \* Types of Linked Lists:

     \*

     \* 1. Singly Linked List:

     \* - Consists of nodes where each node contains data and a reference (or pointer) to the next node.

     \* - Allows traversal in one direction (from head to tail).

     \* - Efficient for insertions and deletions as they do not require shifting elements.

     \*

     \* 2. Doubly Linked List:

     \* - Each node contains data, a reference to the next node, and a reference to the previous node.

     \* - Allows traversal in both directions (forward and backward).

     \* - More memory overhead due to the extra pointer, but allows for more flexible operations.

     \*/

    // Step 2: Task class setup

    class Task

    {

        public int TaskId { get; set; }

        public string TaskName { get; set; }

        public string Status { get; set; }

        public Task(int taskId, string taskName, string status)

        {

            TaskId = taskId;

            TaskName = taskName;

            Status = status;

        }

        public override string ToString()

        {

            return $"Task ID: {TaskId}, Name: {TaskName}, Status: {Status}";

        }

    }

    // Node class for the singly linked list

    class Node

    {

        public Task Data { get; set; }

        public Node Next { get; set; }

        public Node(Task data)

        {

            Data = data;

            Next = null;

        }

    }

    // Singly Linked List for managing tasks

    class TaskLinkedList

    {

        private Node head;

        // Step 3: Add a task

        public void AddTask(Task task)

        {

            Node newNode = new Node(task);

            if (head == null)

            {

                head = newNode;

            }

            else

            {

                Node current = head;

                while (current.Next != null)

                {

                    current = current.Next;

                }

                current.Next = newNode;

            }

        }

        // Search for a task by ID

        public Task SearchTask(int taskId)

        {

            Node current = head;

            while (current != null)

            {

                if (current.Data.TaskId == taskId)

                {

                    return current.Data;

                }

                current = current.Next;

            }

            return null; // Not found

        }

        // Traverse and display all tasks

        public void TraverseTasks()

        {

            Node current = head;

            while (current != null)

            {

                Console.WriteLine(current.Data);

                current = current.Next;

            }

        }

        // Delete a task by ID

        public void DeleteTask(int taskId)

        {

            if (head == null) return;

            // If the head needs to be removed

            if (head.Data.TaskId == taskId)

            {

                head = head.Next;

                Console.WriteLine($"Task with ID {taskId} has been deleted.");

                return;

            }

            Node current = head;

            while (current.Next != null)

            {

                if (current.Next.Data.TaskId == taskId)

                {

                    current.Next = current.Next.Next; // Bypass the node to delete it

                    Console.WriteLine($"Task with ID {taskId} has been deleted.");

                    return;

                }

                current = current.Next;

            }

            Console.WriteLine("Task not found.");

        }

    }

    static void Main(string[] args)

    {

        // Step 3: Create an instance of TaskLinkedList

        TaskLinkedList taskList = new TaskLinkedList();

        // Adding tasks

        taskList.AddTask(new Task(1, "Design UI", "In Progress"));

        taskList.AddTask(new Task(2, "Implement Backend", "Pending"));

        taskList.AddTask(new Task(3, "Write Documentation", "Completed"));

        // Traverse tasks

        Console.WriteLine("Task List:");

        taskList.TraverseTasks();

        Console.WriteLine();

        // Search for a task

        var searchResult = taskList.SearchTask(2);

        Console.WriteLine("Search Result:");

        Console.WriteLine(searchResult != null ? searchResult.ToString() : "Task not found.");

        Console.WriteLine();

        // Delete a task

        taskList.DeleteTask(2);

        Console.WriteLine("Task List after deletion:");

        taskList.TraverseTasks();

        Console.WriteLine();

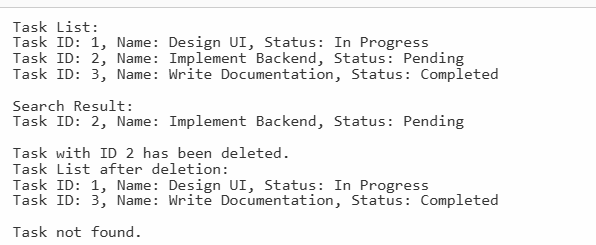
        // Attempt to delete a non-existing task

        taskList.DeleteTask(4);

    }

}

**Output:**



**Exercise 6: Library Management System**

**Codes:**

using System;

using System.Linq;

class Program

{

    // Step 1: Understanding Search Algorithms

    /\*

     \* Search Algorithms:

     \*

     \* 1. Linear Search:

     \* - A simple search algorithm that checks each element in the list sequentially until the desired element is found or the list ends.

     \* - Time Complexity: O(n) in the worst and average cases, O(1) in the best case (when the element is found at the first position).

     \* - Space Complexity: O(1) as it uses a constant amount of space.

     \*

     \* 2. Binary Search:

     \* - A more efficient search algorithm that works on sorted arrays. It divides the search interval in half repeatedly.

     \* - Time Complexity: O(log n) in the worst case, O(1) in the best case (when the middle element is the target).

     \* - Space Complexity: O(1) for the iterative version, O(log n) for the recursive version due to call stack.

     \*

     \* When to Use:

     \* - Use Linear Search for small or unsorted datasets where simplicity is preferred.

     \* - Use Binary Search for large, sorted datasets where efficiency is crucial.

     \*/

    // Step 2: Book class setup

    class Book

    {

        public int BookId { get; set; }

        public string Title { get; set; }

        public string Author { get; set; }

        public Book(int bookId, string title, string author)

        {

            BookId = bookId;

            Title = title;

            Author = author;

        }

        public override string ToString()

        {

            return $"ID: {BookId}, Title: {Title}, Author: {Author}";

        }

    }

    // Step 3: Implement Linear Search

    static Book LinearSearch(Book[] books, string title)

    {

        foreach (var book in books)

        {

            if (book.Title.Equals(title, StringComparison.OrdinalIgnoreCase))

            {

                return book;

            }

        }

        return null; // Not found

    }

    // Step 3: Implement Binary Search

    static Book BinarySearch(Book[] sortedBooks, string title)

    {

        int left = 0;

        int right = sortedBooks.Length - 1;

        while (left <= right)

        {

            int mid = left + (right - left) / 2;

            int comparison = string.Compare(sortedBooks[mid].Title, title, StringComparison.OrdinalIgnoreCase);

            if (comparison == 0)

            {

                return sortedBooks[mid]; // Found

            }

            else if (comparison < 0)

            {

                left = mid + 1; // Search in the right half

            }

            else

            {

                right = mid - 1; // Search in the left half

            }

        }

        return null; // Not found

    }

    static void Main(string[] args)

    {

        // Sample books

        Book[] books = new Book[]

        {

            new Book(1, "The Great Gatsby", "F. Scott Fitzgerald"),

            new Book(2, "To Kill a Mockingbird", "Harper Lee"),

            new Book(3, "1984", "George Orwell"),

            new Book(4, "Pride and Prejudice", "Jane Austen"),

            new Book(5, "The Catcher in the Rye", "J.D. Salinger")

        };

        // Linear Search

        Console.WriteLine("Linear Search:");

        string searchTitle = "1984";

        var linearResult = LinearSearch(books, searchTitle);

        Console.WriteLine(linearResult != null ? linearResult.ToString() : "Book not found.");

        Console.WriteLine();

        // Binary Search (requires sorted array)

        Console.WriteLine("Binary Search:");

        var sortedBooks = books.OrderBy(b => b.Title).ToArray(); // Sort books by title

        searchTitle = "Pride and Prejudice";

        var binaryResult = BinarySearch(sortedBooks, searchTitle);

        Console.WriteLine(binaryResult != null ? binaryResult.ToString() : "Book not found.");

        Console.WriteLine();

        // Time Complexity Analysis

        Console.WriteLine("Time Complexity Analysis:");

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

        Console.WriteLine("Binary Search: O(log n)");

        Console.WriteLine();

        // Discussion on when to use each algorithm

        Console.WriteLine("When to Use Each Algorithm:");

        Console.WriteLine("- Use Linear Search for small or unsorted datasets.");

        Console.WriteLine("- Use Binary Search for large, sorted datasets for better performance.");

    }

}

**Output:**

