**DATA STRUCTURES AND ALGORITHMS(HANDS ON)**

Exercise 2: E-commerce Platform Search Function

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**Steps:**

1. **Understand Asymptotic Notation:**
   * Explain Big O notation and how it helps in analyzing algorithms.
   * Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.
   * Discuss which algorithm is more suitable for your platform and why.

SOLUTION:

Big O notation is a mathematical concept used in computer science to describe the efficiency of an algorithm. It provides an upper bound on the growth rate of an algorithm's time or space complexity as the input size increases.

|  |  |  |  |
| --- | --- | --- | --- |
| **Big O** | **Name** | **Description** | **Example Algorithm** |
| O (1) | Constant time | Executes in the same time regardless of input size. | Accessing array element |
| O (log n) | Logarithmic time | Reduces input size each step | Binary search |
| O(n) | Linear time | Time grows directly with input size | Linear search, traversal |
| O (n logn) | Log-linear time | Efficient sorting | Merge sort, quick sort |
| O(n^2) | Quadratic time | Time grows with square of input | Bubble sort, nested loops |
| O(2^n) | Exponential time | Doubles with each input increase | Recursive Fibonacci |
| O(n!) | Factorial time | Extremely slow for large inputs | Brute-force permutations |

**b. Describe the best, average, and worst-case scenarios for search operations.**

For search operations, we are typically looking for an item in a collection of n items.

* **Best Case:**
  + **Definition:** The scenario where the algorithm performs the minimum number of operations.
  + **Example (Linear Search):** The target item is the very first element checked. Complexity: O (1).
  + **Example (Binary Search):** The target item is the middle element of the sorted collection, found on the first check. Complexity: O (1).
* **Average Case:**
  + **Definition:** The expected performance of the algorithm over all possible inputs, assuming a uniform distribution of inputs or a typical input.
  + **Example (Linear Search):** The target item is found somewhere in the middle of the collection, or it's not present and we have to check about half the elements on average (though formally still O(n) as we're interested in the upper bound of growth). For a successful search, on average, n/2 comparisons.
  + **Example (Binary Search):** The target item is found after a few divisions of the search space. Complexity: O (log n).
* **Worst Case:**
  + **Definition:** The scenario where the algorithm performs the maximum number of operations. Big O notation typically focuses on this.
  + **Example (Linear Search):** The target item is the last element checked, or the item is not present in the collection at all, requiring a full scan. Complexity: O(n).
  + **Example (Binary Search):** The target item is found at the very end of the search (after maximum divisions), or the item is not present, requiring the search space to be narrowed down completely. Complexity: O (log n).

**CODE**:

using System;

using System.Collections.Generic;

using System.Linq; // Needed for OrderBy

// --- Product Class Definition ---

// Represents a single product in the e-commerce system.

public class Product

{

    public int ProductId { get; private set; } // Set once, then read-only

    public string ProductName { get; private set; } // Set once, then read-only

    public string Category { get; private set; } // Set once, then read-only

    public decimal Price { get; private set; } // Added a price attribute

    public int StockQuantity { get; private set; } // Added stock quantity

    // Constructor to initialize a new Product instance with all essential details.

    public Product(int productId, string productName, string category, decimal price, int stockQuantity)

    {

        // Basic validation in constructor to ensure valid product creation

        if (productId <= 0)

            throw new ArgumentOutOfRangeException(nameof(productId), "Product ID must be positive.");

        if (string.IsNullOrWhiteSpace(productName))

            throw new ArgumentException("Product name cannot be null or empty.", nameof(productName));

        if (string.IsNullOrWhiteSpace(category))

            throw new ArgumentException("Category cannot be null or empty.", nameof(category));

        if (price < 0)

            throw new ArgumentOutOfRangeException(nameof(price), "Price cannot be negative.");

        if (stockQuantity < 0)

            throw new ArgumentOutOfRangeException(nameof(stockQuantity), "Stock quantity cannot be negative.");

        ProductId = productId;

        ProductName = productName;

        Category = category;

        Price = price;

        StockQuantity = stockQuantity;

    }

    // Override ToString for easy and comprehensive printing of product details.

    public override string ToString()

    {

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

    }

}

// --- ProductSearchService Class Definition ---

// Manages the search operations for products.

// Encapsulates different search algorithms.

public class ProductSearchService

{

    // Using a List<Product> instead of Product[] for more flexibility if the product catalog grows

    // or changes during runtime (though for this exercise, it's treated as static).

    private readonly List<Product> \_products;

    // Constructor to initialize the search service with a product collection.

    public ProductSearchService(IEnumerable<Product> products)

    {

        // Ensure products are not null, and convert to List for internal management.

        \_products = products?.ToList() ?? throw new ArgumentNullException(nameof(products));

    }

    /// <summary>

    /// Performs a linear search for products whose name contains the specified part.

    /// Case-insensitive partial match.

    /// </summary>

    /// <param name="namePart">The partial name to search for.</param>

    /// <returns>A list of matching products.</returns>

    public List<Product> SearchByNamePartial(string namePart)

    {

        Console.WriteLine($"\n--- Performing Linear Search for products with name containing '{namePart}' ---");

        // Using LINQ's Where clause makes the linear search more concise and readable.

        // It still performs a linear scan behind the scenes.

        var results = \_products

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

            .ToList();

        return results;

    }

    /// <summary>

    /// Performs a linear search for products whose category contains the specified part.

    /// Case-insensitive partial match.

    /// </summary>

    /// <param name="categoryPart">The partial category name to search for.</param>

    /// <returns>A list of matching products.</returns>

    public List<Product> SearchByCategoryPartial(string categoryPart)

    {

        Console.WriteLine($"\n--- Performing Linear Search for products in category containing '{categoryPart}' ---");

        var results = \_products

            .Where(p => p.Category.Contains(categoryPart, StringComparison.OrdinalIgnoreCase))

            .ToList();

        return results;

    }

    /// <summary>

    /// Performs a binary search for a product by its exact ProductId.

    /// Assumes the internal product array will be sorted by ProductId before searching.

    /// </summary>

    /// <param name="targetId">The ProductId to search for.</param>

    /// <returns>The found Product, or null if not found.</returns>

    public Product BinarySearchById(int targetId)

    {

        Console.WriteLine($"\n--- Performing Binary Search for Product ID: {targetId} ---");

        // Create a sorted copy of the product list for binary search.

        // This ensures the original list isn't modified and is sorted for the ID search.

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

        int low = 0;

        int high = sortedProducts.Length - 1;

        while (low <= high)

        {

            int mid = low + (high - low) / 2; // Robust mid-point calculation

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

            {

                Console.WriteLine($"Product ID {targetId} found.");

                return sortedProducts[mid]; // Found the product

            }

            else if (sortedProducts[mid].ProductId < targetId)

            {

                low = mid + 1; // Target is in the right half

            }

            else

            {

                high = mid - 1; // Target is in the left half

            }

        }

        Console.WriteLine($"Product ID {targetId} not found.");

        return null; // Product not found

    }

}

// --- Program Class Definition ---

// The main application entry point for user interaction.

public class Program

{

    public static void Main(string[] args)

    {

        // Initialize the product catalog with more attributes.

        // Using a List<Product> for initial data makes it flexible.

        List<Product> productCatalog = new List<Product>

        {

            new Product(1, "Redmi Note 7 Pro", "Electronics", 12999.00m, 500),

            new Product(2, "ClassMate 160 pages Notebook", "Stationary", 85.50m, 1200),

            new Product(3, "One Plus Ear Buds", "Electronics", 3999.00m, 300),

            new Product(4, "Red Silk Curtain", "Home Decorations", 750.00m, 150),

            new Product(5, "Boat Rockerz 255 Pro+", "Electronics", 1299.00m, 400),

            new Product(6, "Faber-Castell Color Pencils", "Stationary", 250.00m, 800),

            new Product(7, "Samsung Galaxy S22", "Electronics", 65000.00m, 200),

            new Product(8, "Ikea Wooden Study Table", "Furniture", 4500.00m, 80),

            new Product(9, "Usha Mist Air Fan", "Home Appliances", 2500.00m, 100),

            new Product(10, "Raymond Men’s Formal Shirt", "Clothing", 1800.00m, 250),

            new Product(11, "Dell Inspiron 15 Laptop", "Electronics", 48000.00m, 180),

            new Product(12, "Havells LED Bulb 9W", "Home Appliances", 120.00m, 1000),

            new Product(13, "Parker Ink Pen", "Stationary", 550.00m, 300),

            new Product(14, "Bombay Dyeing Bedsheet - King Size", "Home Decorations", 1500.00m, 90),

            new Product(15, "Nike Revolution 6 Shoes", "Footwear", 3500.00m, 120),

            new Product(16, "Wildcraft Backpack 35L", "Accessories", 2200.00m, 70),

            new Product(17, "Canon Pixma Inkjet Printer", "Electronics", 8000.00m, 60),

            new Product(18, "Prestige Pressure Cooker 5L", "Kitchen Appliances", 1600.00m, 110),

            new Product(19, "Adidas Sports T-shirt", "Clothing", 999.00m, 300),

            new Product(20, "Philips Beard Trimmer", "Personal Care", 1499.00m, 200),

            new Product(21, "Redmi Note 8 Pro", "Electronics", 14999.00m, 450),

            new Product(22, "Redmi Note 9 Pro", "Electronics", 16999.00m, 400),

            new Product(23, "Apple iPhone 15", "Electronics", 85000.00m, 100),

            new Product(24, "HP Pavilion Laptop", "Electronics", 55000.00m, 150),

            new Product(25, "Levi's 511 Jeans", "Clothing", 2500.00m, 200)

        };

        // Create an instance of our search service with the product catalog.

        ProductSearchService searchService = new ProductSearchService(productCatalog);

        Console.Title = "E-Commerce Product Search System"; // Set console window title

        Console.WriteLine("=== Welcome to the E-Commerce Product Search System ===");

        bool running = true;

        while (running)

        {

            DisplayMainMenu();

            string choiceInput = Console.ReadLine();

            if (int.TryParse(choiceInput, out int choice))

            {

                switch (choice)

                {

                    case 1: // Search by Product Name

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

                        string namePart = Console.ReadLine();

                        var nameResults = searchService.SearchByNamePartial(namePart);

                        DisplaySearchResults(nameResults, $"product name containing '{namePart}'");

                        break;

                    case 2: // Search by Product ID

                        Console.Write("Enter the exact Product ID to search: ");

                        string idInput = Console.ReadLine();

                        if (int.TryParse(idInput, out int productId))

                        {

                            Product foundProduct = searchService.BinarySearchById(productId);

                            if (foundProduct != null)

                            {

                                Console.WriteLine($"Found: {foundProduct}");

                            }

                            else

                            {

                                Console.WriteLine("Product with the specified ID was not found.");

                            }

                        }

                        else

                        {

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

                        }

                        break;

                    case 3: // Search by Category

                        Console.Write("Enter a partial category name to search: ");

                        string categoryPart = Console.ReadLine();

                        var categoryResults = searchService.SearchByCategoryPartial(categoryPart);

                        DisplaySearchResults(categoryResults, $"category containing '{categoryPart}'");

                        break;

                    case 4: // Exit

                        Console.WriteLine("Exiting the application. Goodbye!");

                        running = false;

                        break;

                    default:

                        Console.WriteLine("Invalid choice. Please enter a number between 1 and 4.");

                        break;

                }

            }

            else

            {

                Console.WriteLine("Invalid input. Please enter a number corresponding to your choice.");

            }

            if (running) // Don't pause if we're about to exit

            {

                Console.WriteLine("\n--- Press Enter to continue... ---");

                Console.ReadLine(); // Pause for user to read output

                Console.Clear(); // Clear console for cleaner look

            }

        }

    }

    private static void DisplayMainMenu()

    {

        Console.WriteLine("\n--- Main Menu ---");

        Console.WriteLine("1. Search by Product Name (partial match)");

        Console.WriteLine("2. Search by Product ID (exact match)");

        Console.WriteLine("3. Search by Category (partial match)");

        Console.WriteLine("4. Exit");

        Console.Write("Please enter your choice: ");

    }

    /// <summary>

    /// Helper method to display search results.

    /// </summary>

    /// <param name="results">List of products found.</param>

    /// <param name="searchDescription">Description of the search performed.</param>

    private static void DisplaySearchResults(List<Product> results, string searchDescription)

    {

        if (results.Any())

        {

            Console.WriteLine($"\nFound {results.Count} product(s) for {searchDescription}:");

            foreach (var product in results)

            {

                Console.WriteLine($"- {product}");

            }

        }

        else

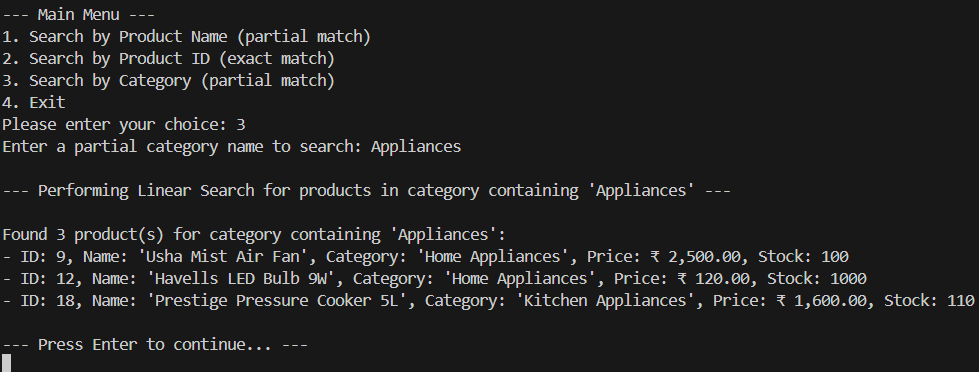
        {

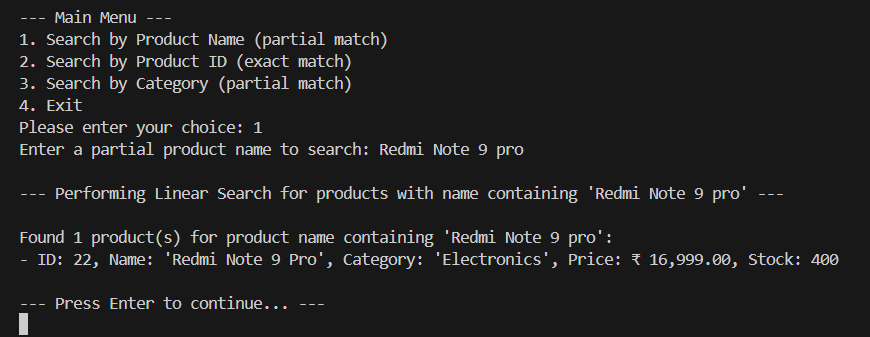
            Console.WriteLine($"\nNo products found for {searchDescription}.");

        }

    }

}

**OUTPUT:** ****



A screenshot of a computer program

AI-generated content may be incorrect.

**EX.7: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

Steps:

1. Understand Recursive Algorithms:
   * Explain the concept of recursion and how it can simplify certain problems.
2. Setup:
   * Create a method to calculate the future value using a recursive approach.
3. Implementation:
   * Implement a recursive algorithm to predict future values based on past growth rates.
4. Analysis:
   * Discuss the time complexity of your recursive algorithm.
   * Explain how to optimize the recursive solution to avoid excessive computation.

Solution:

1.Recursion is a programming concept where a function calls itself within its own definition to solve a problem. Here we use the STACK data structure.

It simplifies problems that:

* Have a repetitive, self-similar structure (e.g., calculating future values based on previous years).
* Can be broken down into base case + recursive case.

2. We'll define a recursive method to calculate the future value of an investment over time, based on:

* Initial value
* Annual growth rate
* Number of years to forecast.

3.**CODE:**

using System;

class FinancialForecast

{

    // Recursive method to calculate future value

    public static double CalculateFutureValue(double initialValue, double growthRate, int years)

    {

        // Base case: no more years to grow

        if (years == 0)

            return initialValue;

        // Recursive case: grow this year's value and call for remaining years

        return CalculateFutureValue(initialValue \* (1 + growthRate), growthRate, years - 1);

    }

    static void Main()

    {

        Console.Write("Enter initial value: ");

        double initial = double.Parse(Console.ReadLine());

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

        double rate = double.Parse(Console.ReadLine());

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

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

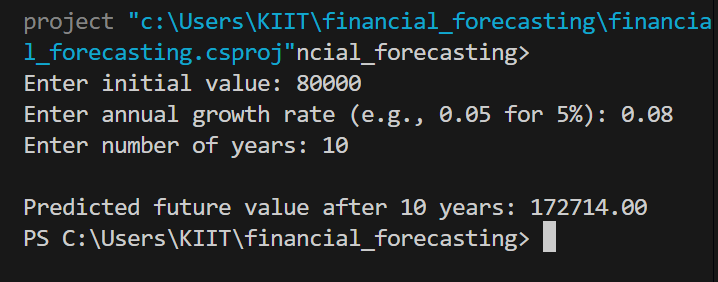
        double futureValue = CalculateFutureValue(initial, rate, years);

        Console.WriteLine($"\nPredicted future value after {years} years: {futureValue:F2}");

    }

}

**OUTPUT**:



**ANALYSIS:**

>**Time complexity**:

i)The algorithm runs O(n) times where n denotes the number of years.so it’s linear

ii)It makes one recursive call per year.

**Note:** For very large values of n, recursion can lead to **stack overflow.**

**Optimization:**

Use tail recursion or convert to iteration to avoid deep call stacks.

public static double CalculateFutureValueIterative(double initialValue, double growthRate, int years)

{

double result = initialValue;

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

{

result \*= (1 + growthRate);

}

return result;

}