Exercise 2:E-commerce Platform Search Function

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Linq;

// Product class with attributes for searching

public 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 productId, string productName, string category, decimal price)

{

ProductId = productId;

ProductName = productName;

Category = category;

Price = price;

}

// Implementation for sorting products by ProductId

public int CompareTo(Product other)

{

if (other == null) return 1;

return ProductId.CompareTo(other.ProductId);

}

public override string ToString()

{

return $"ID: {ProductId}, Name: {ProductName}, Category: {Category}, Price: ${Price:F2}";

}

}

// E-commerce search platform

public class ECommerceSearchPlatform

{

private Product[] products; // Unsorted array for linear search

private Product[] sortedProducts; // Sorted array for binary search

private int productCount;

public ECommerceSearchPlatform(int capacity)

{

products = new Product[capacity];

sortedProducts = new Product[capacity];

productCount = 0;

}

// Add product to the platform

public void AddProduct(Product product)

{

if (productCount < products.Length)

{

products[productCount] = product;

productCount++;

// Update sorted array

Array.Copy(products, sortedProducts, productCount);

Array.Sort(sortedProducts, 0, productCount);

}

else

{

Console.WriteLine("Platform capacity reached!");

}

}

// LINEAR SEARCH IMPLEMENTATION

// Time Complexity: O(n)

// Best Case: O(1) - element found at first position

// Average Case: O(n/2) = O(n) - element found in middle

// Worst Case: O(n) - element at end or not found

public Product LinearSearchById(int productId)

{

int comparisons = 0;

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

{

comparisons++;

if (products[i].ProductId == productId)

{

Console.WriteLine($"Linear Search - Comparisons made: {comparisons}");

return products[i];

}

}

Console.WriteLine($"Linear Search - Product not found. Comparisons made: {comparisons}");

return null;

}

// Linear search by product name (case-insensitive)

public List<Product> LinearSearchByName(string productName)

{

List<Product> results = new List<Product>();

int comparisons = 0;

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

{

comparisons++;

if (products[i].ProductName.ToLower().Contains(productName.ToLower()))

{

results.Add(products[i]);

}

}

Console.WriteLine($"Linear Search by Name - Comparisons made: {comparisons}");

return results;

}

// BINARY SEARCH IMPLEMENTATION

// Time Complexity: O(log n)

// Best Case: O(1) - element found at middle

// Average Case: O(log n)

// Worst Case: O(log n) - element at leaf or not found

public Product BinarySearchById(int productId)

{

int left = 0;

int right = productCount - 1;

int comparisons = 0;

while (left <= right)

{

comparisons++;

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

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

{

Console.WriteLine($"Binary Search - Comparisons made: {comparisons}");

return sortedProducts[mid];

}

if (sortedProducts[mid].ProductId < productId)

{

left = mid + 1;

}

else

{

right = mid - 1;

}

}

Console.WriteLine($"Binary Search - Product not found. Comparisons made: {comparisons}");

return null;

}

// Recursive Binary Search Implementation

public Product BinarySearchRecursive(int productId)

{

int comparisons = 0;

var result = BinarySearchRecursiveHelper(productId, 0, productCount - 1, ref comparisons);

Console.WriteLine($"Recursive Binary Search - Comparisons made: {comparisons}");

return result;

}

private Product BinarySearchRecursiveHelper(int productId, int left, int right, ref int comparisons)

{

if (left > right)

return null;

comparisons++;

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

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

return sortedProducts[mid];

if (sortedProducts[mid].ProductId > productId)

return BinarySearchRecursiveHelper(productId, left, mid - 1, ref comparisons);

else

return BinarySearchRecursiveHelper(productId, mid + 1, right, ref comparisons);

}

// Performance benchmark method

public void BenchmarkSearchAlgorithms(int productId, int iterations = 1000)

{

Stopwatch sw = new Stopwatch();

// Benchmark Linear Search

sw.Start();

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

{

LinearSearchById(productId);

}

sw.Stop();

long linearTime = sw.ElapsedTicks;

// Benchmark Binary Search

sw.Restart();

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

{

BinarySearchById(productId);

}

sw.Stop();

long binaryTime = sw.ElapsedTicks;

Console.WriteLine($"\nPerformance Benchmark ({iterations} iterations):");

Console.WriteLine($"Linear Search Time: {linearTime} ticks");

Console.WriteLine($"Binary Search Time: {binaryTime} ticks");

Console.WriteLine($"Binary Search is {(double)linearTime / binaryTime:F2}x faster");

}

// Display all products

public void DisplayProducts()

{

Console.WriteLine($"\nAll Products ({productCount} total):");

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

{

Console.WriteLine($"{i + 1}. {products[i]}");

}

}

// Display sorted products

public void DisplaySortedProducts()

{

Console.WriteLine($"\nSorted Products by ID ({productCount} total):");

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

{

Console.WriteLine($"{i + 1}. {sortedProducts[i]}");

}

}

public int GetProductCount() => productCount;

}

// Program class with comprehensive testing and analysis

public class Program

{

public static void Main()

{

Console.WriteLine("=== E-COMMERCE PLATFORM SEARCH FUNCTION ANALYSIS ===\n");

// 1. Asymptotic Notation Explanation

ExplainAsymptoticNotation();

// 2. Setup and populate the platform

ECommerceSearchPlatform platform = new ECommerceSearchPlatform(20);

PopulatePlatform(platform);

// 3. Display products

platform.DisplayProducts();

platform.DisplaySortedProducts();

// 4. Demonstrate search algorithms

Console.WriteLine("\n=== SEARCH ALGORITHM DEMONSTRATIONS ===");

DemonstrateSearchAlgorithms(platform);

// 5. Performance analysis

Console.WriteLine("\n=== PERFORMANCE ANALYSIS ===");

AnalyzePerformance(platform);

// 6. Recommendations

Console.WriteLine("\n=== PLATFORM RECOMMENDATIONS ===");

ProvideRecommendations();

}

private static void ExplainAsymptoticNotation()

{

Console.WriteLine("=== BIG O NOTATION EXPLANATION ===");

Console.WriteLine("Big O notation describes the upper bound of algorithm performance as input size grows.\n");

Console.WriteLine("Common Time Complexities (from best to worst):");

Console.WriteLine("• O(1) - Constant: Same time regardless of input size");

Console.WriteLine("• O(log n) - Logarithmic: Time increases slowly as input grows");

Console.WriteLine("• O(n) - Linear: Time increases proportionally with input");

Console.WriteLine("• O(n log n) - Linearithmic: Common in efficient sorting algorithms");

Console.WriteLine("• O(n²) - Quadratic: Time increases quadratically with input\n");

Console.WriteLine("SEARCH ALGORITHM SCENARIOS:");

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

Console.WriteLine("• Best Case: O(1) - Element is first in array");

Console.WriteLine("• Average Case: O(n/2) = O(n) - Element is in middle");

Console.WriteLine("• Worst Case: O(n) - Element is last or not found\n");

Console.WriteLine("Binary Search O(log n) - Requires sorted array:");

Console.WriteLine("• Best Case: O(1) - Element is at middle position");

Console.WriteLine("• Average Case: O(log n) - Standard binary search");

Console.WriteLine("• Worst Case: O(log n) - Element at leaf level or not found\n");

}

private static void PopulatePlatform(ECommerceSearchPlatform platform)

{

// Add sample products with various IDs for testing

platform.AddProduct(new Product(101, "Wireless Headphones", "Electronics", 79.99m));

platform.AddProduct(new Product(205, "Running Shoes", "Sports", 129.50m));

platform.AddProduct(new Product(156, "Coffee Maker", "Appliances", 89.99m));

platform.AddProduct(new Product(078, "Smartphone", "Electronics", 699.00m));

platform.AddProduct(new Product(342, "Yoga Mat", "Sports", 29.99m));

platform.AddProduct(new Product(198, "Laptop Bag", "Accessories", 45.00m));

platform.AddProduct(new Product(267, "Bluetooth Speaker", "Electronics", 59.99m));

platform.AddProduct(new Product(134, "Kitchen Knife Set", "Kitchen", 119.99m));

platform.AddProduct(new Product(089, "Fitness Tracker", "Electronics", 149.99m));

platform.AddProduct(new Product(423, "Camping Tent", "Outdoors", 199.99m));

platform.AddProduct(new Product(012, "Water Bottle", "Sports", 19.99m));

platform.AddProduct(new Product(356, "Gaming Mouse", "Electronics", 69.99m));

platform.AddProduct(new Product(245, "Desk Chair", "Furniture", 179.99m));

platform.AddProduct(new Product(167, "Cookbook", "Books", 24.99m));

platform.AddProduct(new Product(398, "Sunglasses", "Accessories", 89.50m));

}

private static void DemonstrateSearchAlgorithms(ECommerceSearchPlatform platform)

{

int[] testIds = { 78, 267, 999, 12 }; // Mix of existing and non-existing IDs

foreach (int id in testIds)

{

Console.WriteLine($"\n--- Searching for Product ID: {id} ---");

// Linear Search

Console.WriteLine("Linear Search Result:");

var linearResult = platform.LinearSearchById(id);

if (linearResult != null)

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

// Binary Search

Console.WriteLine("\nBinary Search Result:");

var binaryResult = platform.BinarySearchById(id);

if (binaryResult != null)

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

// Recursive Binary Search

Console.WriteLine("\nRecursive Binary Search Result:");

var recursiveResult = platform.BinarySearchRecursive(id);

if (recursiveResult != null)

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

Console.WriteLine(new string('-', 50));

}

// Demonstrate name search

Console.WriteLine("\n--- Searching by Product Name: 'head' ---");

var nameResults = platform.LinearSearchByName("head");

if (nameResults.Count > 0)

{

Console.WriteLine("Products found:");

nameResults.ForEach(p => Console.WriteLine($" {p}"));

}

else

{

Console.WriteLine("No products found with that name.");

}

}

private static void AnalyzePerformance(ECommerceSearchPlatform platform)

{

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

Console.WriteLine($"Dataset size: {platform.GetProductCount()} products\n");

Console.WriteLine("Linear Search:");

Console.WriteLine("• Time Complexity: O(n)");

Console.WriteLine("• Space Complexity: O(1)");

Console.WriteLine("• Works on unsorted data");

Console.WriteLine($"• Max comparisons needed: {platform.GetProductCount()}");

Console.WriteLine($"• Average comparisons: {platform.GetProductCount() / 2.0:F1}\n");

Console.WriteLine("Binary Search:");

Console.WriteLine("• Time Complexity: O(log n)");

Console.WriteLine("• Space Complexity: O(1) for iterative, O(log n) for recursive");

Console.WriteLine("• Requires sorted data");

Console.WriteLine($"• Max comparisons needed: {Math.Ceiling(Math.Log(platform.GetProductCount()) / Math.Log(2))}");

Console.WriteLine($"• Average comparisons: {Math.Log(platform.GetProductCount()) / Math.Log(2):F1}\n");

// Performance benchmark

platform.BenchmarkSearchAlgorithms(156, 10000);

// Scalability analysis

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

int[] sizes = { 100, 1000, 10000, 100000, 1000000 };

Console.WriteLine("Dataset Size | Linear Search | Binary Search");

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

foreach (int size in sizes)

{

int linearComparisons = size / 2; // average case

int binaryComparisons = (int)Math.Ceiling(Math.Log(size) / Math.Log(2));

Console.WriteLine($"{size,12} | {linearComparisons,13} | {binaryComparisons,13}");

}

}

private static void ProvideRecommendations()

{

Console.WriteLine("ALGORITHM SELECTION RECOMMENDATIONS FOR E-COMMERCE:\n");

Console.WriteLine("Use LINEAR SEARCH when:");

Console.WriteLine("• Dataset is small (< 100 products)");

Console.WriteLine("• Data changes frequently (frequent additions/deletions)");

Console.WriteLine("• Searching by non-indexed fields (name, description)");

Console.WriteLine("• Memory is extremely limited");

Console.WriteLine("• Implementation simplicity is priority\n");

Console.WriteLine("Use BINARY SEARCH when:");

Console.WriteLine("• Dataset is large (> 1000 products)");

Console.WriteLine("• Searching by indexed/sorted fields (ID, price)");

Console.WriteLine("• Fast search response is critical");

Console.WriteLine("• Data is relatively stable");

Console.WriteLine("• Can maintain sorted order\n");

Console.WriteLine("OPTIMAL E-COMMERCE STRATEGY:");

Console.WriteLine("• Use HASH TABLES/DICTIONARIES for O(1) ID lookups");

Console.WriteLine("• Use BINARY SEARCH for range queries (price ranges)");

Console.WriteLine("• Use LINEAR SEARCH with INDEXES for text search");

Console.WriteLine("• Consider DATABASE INDEXES for large-scale platforms");

Console.WriteLine("• Implement CACHING for frequently searched items");

Console.WriteLine("• Use ELASTICSEARCH/SOLR for complex search requirements\n");

Console.WriteLine("FINAL RECOMMENDATION:");

Console.WriteLine("For a production e-commerce platform, implement a hybrid approach:");

Console.WriteLine("• Hash table for exact ID matches: O(1)");

Console.WriteLine("• Binary search for sorted numerical ranges: O(log n)");

Console.WriteLine("• Full-text search engines for product names/descriptions");

Console.WriteLine("• Database with proper indexing for scalability");

}

}

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer screen

AI-generated content may be incorrect.

Exercise 7:Financial Forecasting

using System;

using System.Collections.Generic;

using System.Linq;

namespace FinancialForecasting

{

public class FinancialForecaster

{

// Dictionary to store memoized results for optimization

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

/// <summary>

/// Calculates future value using simple recursive compound growth

/// Time Complexity: O(n) where n is the number of periods

/// Space Complexity: O(n) due to recursion stack

/// </summary>

public double CalculateFutureValueRecursive(double initialValue, double growthRate, int periods)

{

// Base case: no more periods to calculate

if (periods == 0)

return initialValue;

// Recursive case: calculate current period value and recurse for remaining periods

return CalculateFutureValueRecursive(initialValue \* (1 + growthRate), growthRate, periods - 1);

}

/// <summary>

/// Optimized recursive calculation using memoization

/// Time Complexity: O(n) for first calculation, O(1) for subsequent calls with same parameters

/// Space Complexity: O(n) for memoization cache + recursion stack

/// </summary>

public double CalculateFutureValueMemoized(double initialValue, double growthRate, int periods)

{

// Create a unique key for memoization

string key = $"{initialValue}\_{growthRate}\_{periods}";

int hashKey = key.GetHashCode();

if (memoCache.ContainsKey(hashKey))

return memoCache[hashKey];

double result;

if (periods == 0)

{

result = initialValue;

}

else

{

result = CalculateFutureValueMemoized(initialValue \* (1 + growthRate), growthRate, periods - 1);

}

memoCache[hashKey] = result;

return result;

}

/// <summary>

/// Calculates future value with variable growth rates using recursion

/// This method takes historical data and predicts future values

/// </summary>

public double PredictFutureValueWithVariableGrowth(double[] historicalValues, int periodsToPredict)

{

if (historicalValues.Length < 2)

throw new ArgumentException("Need at least 2 historical values to calculate growth rate");

// Calculate average growth rate from historical data

double averageGrowthRate = CalculateAverageGrowthRateRecursive(historicalValues, 0, 0.0, 0);

// Use the most recent value as starting point

double lastValue = historicalValues[historicalValues.Length - 1];

return CalculateFutureValueRecursive(lastValue, averageGrowthRate, periodsToPredict);

}

/// <summary>

/// Recursively calculates average growth rate from historical data

/// </summary>

private double CalculateAverageGrowthRateRecursive(double[] values, int index, double sumGrowthRates, int count)

{

// Base case: reached the end of array

if (index >= values.Length - 1)

{

return count > 0 ? sumGrowthRates / count : 0.0;

}

// Calculate growth rate for current period

double growthRate = (values[index + 1] - values[index]) / values[index];

// Recursive case: add current growth rate and continue

return CalculateAverageGrowthRateRecursive(values, index + 1, sumGrowthRates + growthRate, count + 1);

}

/// <summary>

/// Tail-recursive optimized version (though C# doesn't optimize tail recursion)

/// This demonstrates the concept of tail recursion

/// </summary>

public double CalculateFutureValueTailRecursive(double currentValue, double growthRate, int periods)

{

return CalculateFutureValueTailRecursiveHelper(currentValue, growthRate, periods, currentValue);

}

private double CalculateFutureValueTailRecursiveHelper(double initialValue, double growthRate,

int periodsRemaining, double accumulator)

{

if (periodsRemaining == 0)

return accumulator;

return CalculateFutureValueTailRecursiveHelper(initialValue, growthRate, periodsRemaining - 1,

accumulator \* (1 + growthRate));

}

/// <summary>

/// Iterative version for comparison - more efficient than recursion

/// Time Complexity: O(n), Space Complexity: O(1)

/// </summary>

public double CalculateFutureValueIterative(double initialValue, double growthRate, int periods)

{

double result = initialValue;

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

{

result \*= (1 + growthRate);

}

return result;

}

/// <summary>

/// Mathematical formula approach - most efficient

/// Time Complexity: O(1), Space Complexity: O(1)

/// </summary>

public double CalculateFutureValueFormula(double initialValue, double growthRate, int periods)

{

return initialValue \* Math.Pow(1 + growthRate, periods);

}

/// <summary>

/// Clears the memoization cache

/// </summary>

public void ClearCache()

{

memoCache.Clear();

}

/// <summary>

/// Gets the size of the memoization cache

/// </summary>

public int GetCacheSize()

{

return memoCache.Count;

}

}

// Example usage and demonstration

class Program

{

static void Main(string[] args)

{

var forecaster = new FinancialForecaster();

Console.WriteLine("=== Financial Forecasting Tool Demo ===\n");

// Example 1: Simple recursive calculation

Console.WriteLine("1. Simple Recursive Calculation:");

double initialInvestment = 1000.0;

double annualGrowthRate = 0.08; // 8% annual growth

int years = 10;

double futureValue = forecaster.CalculateFutureValueRecursive(initialInvestment, annualGrowthRate, years);

Console.WriteLine($"Initial Investment: ${initialInvestment:F2}");

Console.WriteLine($"Annual Growth Rate: {annualGrowthRate:P}");

Console.WriteLine($"Years: {years}");

Console.WriteLine($"Future Value (Recursive): ${futureValue:F2}\n");

// Example 2: Memoized recursive calculation

Console.WriteLine("2. Memoized Recursive Calculation:");

double memoizedResult = forecaster.CalculateFutureValueMemoized(initialInvestment, annualGrowthRate, years);

Console.WriteLine($"Future Value (Memoized): ${memoizedResult:F2}");

Console.WriteLine($"Cache size after calculation: {forecaster.GetCacheSize()}\n");

// Example 3: Prediction based on historical data

Console.WriteLine("3. Prediction Based on Historical Data:");

double[] historicalValues = { 1000, 1050, 1100, 1180, 1250, 1320, 1400, 1480, 1580, 1650 };

int periodsToPredict = 5;

Console.WriteLine("Historical Values: " + string.Join(", ", historicalValues.Select(v => $"${v:F0}")));

double predictedValue = forecaster.PredictFutureValueWithVariableGrowth(historicalValues, periodsToPredict);

Console.WriteLine($"Predicted value after {periodsToPredict} periods: ${predictedValue:F2}\n");

// Example 4: Performance comparison

Console.WriteLine("4. Performance Comparison:");

PerformanceComparison(forecaster, initialInvestment, annualGrowthRate, years);

// Example 5: Time complexity demonstration

Console.WriteLine("\n5. Time Complexity Demonstration:");

TimeComplexityDemo(forecaster);

}

static void PerformanceComparison(FinancialForecaster forecaster, double initial, double rate, int periods)

{

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

// Recursive

sw.Restart();

double recursive = forecaster.CalculateFutureValueRecursive(initial, rate, periods);

sw.Stop();

long recursiveTime = sw.ElapsedTicks;

// Memoized

forecaster.ClearCache();

sw.Restart();

double memoized = forecaster.CalculateFutureValueMemoized(initial, rate, periods);

sw.Stop();

long memoizedTime = sw.ElapsedTicks;

// Iterative

sw.Restart();

double iterative = forecaster.CalculateFutureValueIterative(initial, rate, periods);

sw.Stop();

long iterativeTime = sw.ElapsedTicks;

// Formula

sw.Restart();

double formula = forecaster.CalculateFutureValueFormula(initial, rate, periods);

sw.Stop();

long formulaTime = sw.ElapsedTicks;

Console.WriteLine($"Recursive: ${recursive:F2} - Time: {recursiveTime} ticks");

Console.WriteLine($"Memoized: ${memoized:F2} - Time: {memoizedTime} ticks");

Console.WriteLine($"Iterative: ${iterative:F2} - Time: {iterativeTime} ticks");

Console.WriteLine($"Formula: ${formula:F2} - Time: {formulaTime} ticks");

}

static void TimeComplexityDemo(FinancialForecaster forecaster)

{

Console.WriteLine("Measuring execution time for different period counts:");

int[] periodCounts = { 10, 20, 30, 40, 50 };

foreach (int periods in periodCounts)

{

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

forecaster.CalculateFutureValueRecursive(1000, 0.05, periods);

sw.Stop();

Console.WriteLine($"Periods: {periods,2} - Time: {sw.ElapsedTicks,8} ticks");

}

}

}

}

A screenshot of a computer

AI-generated content may be incorrect.