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

**Step 1**: Understand Asymptotic Notation

**Big O notation** describes the *upper bound* of an algorithm’s running time or space requirements in terms of input size n.

It helps to:

* Evaluate efficiency
* Compare algorithms
* Predict scalability

Search Scenarios:

* Best Case: Element found immediately
* Average Case: Element found in middle
* Worst Case: Element not found or at end

**Step 2**: Setup  
Code:

public class Product

{

public int ProductId { get; }

public string ProductName { get; }

public string Category { get; }

public Product(int productId, string productName, string category)

{

ProductId = productId;

ProductName = productName;

Category = category;

}

public override string ToString()

{

return $"Product{{id={ProductId}, name='{ProductName}', category='{Category}'}}";

}

}

**Step 3**: Implementation  
Code:

public class SearchAlgorithms

{

// Linear Search

public static Product LinearSearch(Product[] products, int targetId)

{

var stopwatch = Stopwatch.StartNew();

foreach (var product in products)

{

if (product.ProductId == targetId)

{

stopwatch.Stop();

Console.WriteLine($"Linear Search completed in {stopwatch.Elapsed.TotalNanoseconds()} nanoseconds");

return product;

}

}

stopwatch.Stop();

Console.WriteLine($"Linear Search completed in {stopwatch.Elapsed.TotalNanoseconds()} nanoseconds");

return null;

}

// Binary Search

public static Product BinarySearch(Product[] products, int targetId)

{

var stopwatch = Stopwatch.StartNew();

int left = 0, right = products.Length - 1;

while (left <= right)

{

int mid = (left + right) / 2;

int midId = products[mid].ProductId;

if (midId == targetId)

{

stopwatch.Stop();

Console.WriteLine($"Binary Search completed in {stopwatch.Elapsed.TotalNanoseconds()} nanoseconds");

return products[mid];

}

else if (midId < targetId)

{

left = mid + 1;

}

else

{

right = mid - 1;

}

}

stopwatch.Stop();

Console.WriteLine($"Binary Search completed in {stopwatch.Elapsed.TotalNanoseconds()} nanoseconds");

return null;

}

public static void SortById(Product[] products)

{

Array.Sort(products, (a, b) => a.ProductId.CompareTo(b.ProductId));

}

}

// Extension method for nanoseconds

public static class StopwatchExtensions

{

public static double TotalNanoseconds(this TimeSpan ts)

{

return ts.Ticks \* (1000000000.0 / TimeSpan.TicksPerSecond);

}

}

class Program

{

static void Main()

{

Product[] products =

{

new Product(103, "Mouse", "Electronics"),

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

new Product(105, "Notebook", "Stationery"),

new Product(102, "Mobile", "Electronics"),

new Product(104, "Pen", "Stationery")

};

int searchId = 105;

// Linear Search

var foundLinear = SearchAlgorithms.LinearSearch(products, searchId);

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

// Sort and then Binary Search

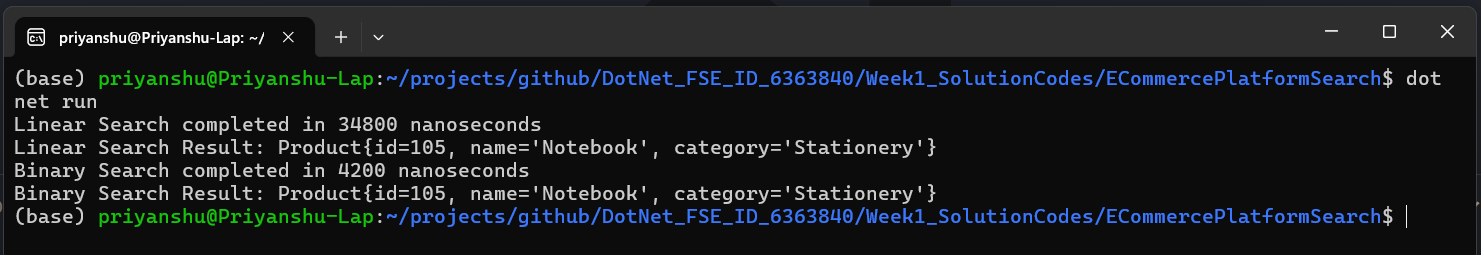
SearchAlgorithms.SortById(products);

var foundBinary = SearchAlgorithms.BinarySearch(products, searchId);

Console.WriteLine("Binary Search Result: " + foundBinary);

}

}

**Step 4**: Analysis  
Output:  


Linear Search Implementation

* Time Complexity: O(n)
* Space Complexity: O(1)

Binary Search Implementation

* Time Complexity: O(log n)
* Space Complexity: O(1)

**Conclusion:** Therefore, for an **e-commerce platform** with **large-scale data**, **binary search** is the right approach due to its O(log n) performance, as data can be sorted corresponding to its ID.

**Exercise 7: Financial Forecasting**

**Step 1**: Understand Recursive Algorithms:

Recursion is a function that calls itself with a smaller input until it reaches a base case. It simplifies problems by dividing them into smaller, identical subproblems. For financial forecasting with a constant growth rate, recursion models compound growth by applying the growth rate to the result of the previous period.

**Step 2**: Setup:

Code:

public class FinancialForecast

{

// Using recursion

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

{

if (periods == 0)

return presentValue;

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

}

// Using iteration

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

{

double futureValue = presentValue;

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

{

futureValue \*= (1 + growthRate);

}

return futureValue;

}

// Using memoization

public static double CalculateFutureValueMemoized(double presentValue, double growthRate, int periods, double[] cache)

{

if (periods == 0)

return presentValue;

if (cache[periods] != 0)

return cache[periods];

cache[periods] = CalculateFutureValueMemoized(presentValue \* (1 + growthRate), growthRate, periods - 1, cache);

return cache[periods];

}

}

// Extension method for nanoseconds

public static class StopwatchExtensions

{

public static double TotalNanoseconds(this TimeSpan ts)

{

return ts.Ticks \* (1000000000.0 / TimeSpan.TicksPerSecond);

}

}

**Step 3**: Implementation:

Code:

class Program

{

static void Main()

{

double presentValue = 1000.0;

double growthRate = 0.08;

int periods = 30;

// Recursive

var sw1 = Stopwatch.StartNew();

double recursive = FinancialForecast.CalculateFutureValue(presentValue, growthRate, periods);

sw1.Stop();

Console.WriteLine($"Future Value (Recursive): {recursive}");

Console.WriteLine($"Time taken (Recursive): {sw1.Elapsed.TotalNanoseconds()} ns");

// Iterative

var sw2 = Stopwatch.StartNew();

double iterative = FinancialForecast.CalculateFutureValueIterative(presentValue, growthRate, periods);

sw2.Stop();

Console.WriteLine($"Future Value (Iterative): {iterative}");

Console.WriteLine($"Time taken (Iterative): {sw2.Elapsed.TotalNanoseconds()} ns");

// Memoized

double[] cache = new double[periods + 1];

var sw3 = Stopwatch.StartNew();

double memoized = FinancialForecast.CalculateFutureValueMemoized(presentValue, growthRate, periods, cache);

sw3.Stop();

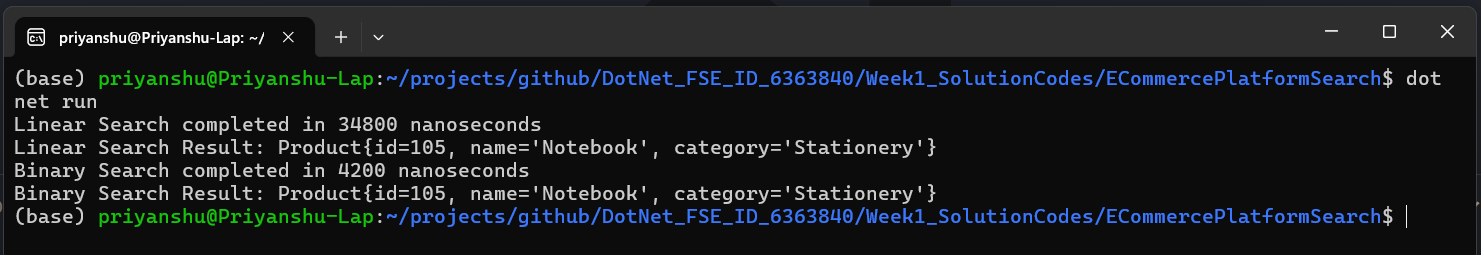
Console.WriteLine($"Future Value (Memoized): {memoized}");

Console.WriteLine($"Time taken (Memoized): {sw3.Elapsed.TotalNanoseconds()} ns");

}

}

**Step 4**: Analysis:  
Output:

  
**Time Complexity**: O(n)

Optimization to Avoid Excessive Computation:

* Iterative Solution
* Instead of calling functions recursively, we build the result iteratively.
* No call stack = no stack overflow
* Fast and memory-efficient
* Excellent for simple recurrence like compound growth
* Memoization
* Memoization stores results of already computed subproblems to avoid repeating them.
* Avoids redundant calculations
* Converts exponential or linear recursion into linear or constant time
* Especially helpful when overlapping subproblems exist.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Approach** | **Time Complexity** | **Space Complexity** | **Risk** | **Speed** |
| Plain Recursion | O(n) | O(n) | Stack overflow | Slow |
| Memoization | O(n) | O(n) | Safer | Faster |
| Iterative | O(n) | O(1) | None | Fastest |