**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 {

**private** **int** productId;

**private** String productName;

**private** String category;

**public** Product(**int** productId, String productName, String category) {

this.productId = productId;

this.productName = productName;

this.category = category;

}

**public** **int** getProductId() {

return productId;

}

**public** String getProductName() {

return productName;

}

**public** String getCategory() {

return category;

}

@**Override**

**public** String toString() {

return "Product{" +

"id=" + productId +

", name='" + productName + '\'' +

", category='" + category + '\'' +

'}';

}

}

**Step 3**: Implementation  
Code:

import java**.**util**.**Arrays;

import java**.**util**.**Comparator;

**public** **class** SearchAlgorithms {

*// Linear Search*

**public** **static** Product linearSearch(Product[] products, **int** targetId) {

**long** startTime = System.nanoTime();

for (Product product : products) {

if (product.getProductId() == targetId) {

**long** endTime = System.nanoTime();

System.out.printf("Linear Search completed in %d nanoseconds\n", (endTime - startTime));

return product;

}

}

**long** endTime = System.nanoTime();

System.out.printf("Linear Search completed in %d nanoseconds\n", (endTime - startTime));

return null;

}

*// Binary Search*

**public** **static** Product binarySearch(Product[] products, **int** targetId) {

**long** startTime = System.nanoTime();

**int** left = 0;

**int** right = products.length - 1;

while (left <= right) {

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

**int** midId = products[mid].getProductId();

if (midId == targetId) {

**long** endTime = System.nanoTime();

System.out.printf("Binary Search completed in %d nanoseconds\n", (endTime - startTime));

return products[mid];

} else if (midId < targetId) {

left = mid + 1;

} else {

right = mid - 1;

}

}

**long** endTime = System.nanoTime();

System.out.printf("Binary Search completed in %d nanoseconds\n", (endTime - startTime));

return null;

}

**public** **static** **void** sortById(Product[] products) {

Arrays.sort(products, Comparator.comparingInt(Product::getProductId));

}

}

**public** **class** Main {

**public** **static** **void** main(String[] args) {

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*

Product foundLinear = SearchAlgorithms.linearSearch(products, searchId);

System.out.println("Linear Search Result: " + foundLinear);

*// Sort and then Binary Search*

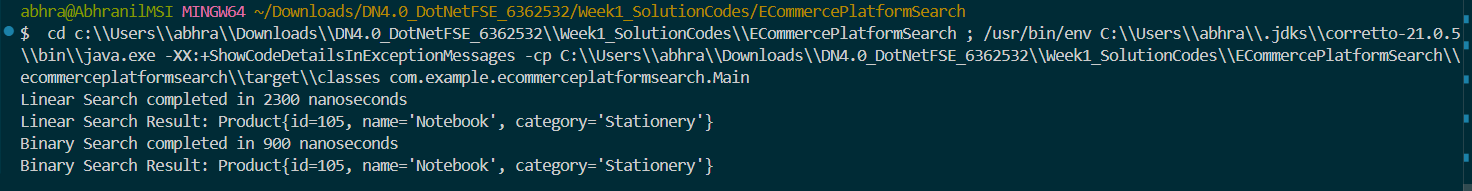
SearchAlgorithms.sortById(products);

Product foundBinary = SearchAlgorithms.binarySearch(products, searchId);

System.out.println("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 base recusrsion*

**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];

}

}

**Step 3**: Implementation

Code:

**public** **class** Main {

**public** **static** **void** main(String[] args) {

**double** presentValue = 1000.0;

**double** growthRate = 0.08;

**int** periods = 30;

*// Timing and output for base recursion*

**long** startRecursive = System.nanoTime();

**double** futureValueRecursive = FinancialForecast.calculateFutureValue(presentValue, growthRate, periods);

**long** endRecursive = System.nanoTime();

System.out.println("Future Value (Recursive): " + futureValueRecursive);

System.out.println("Time taken (Recursive): " + (endRecursive - startRecursive) + " ns");

*// Timing and output for iteration*

**long** startIterative = System.nanoTime();

**double** futureValueIterative = FinancialForecast.calculateFutureValueIterative(presentValue, growthRate,

periods);

**long** endIterative = System.nanoTime();

System.out.println("Future Value (Iterative): " + futureValueIterative);

System.out.println("Time taken (Iterative): " + (endIterative - startIterative) + " ns");

*// Timing and output for memoization*

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

**long** startMemo = System.nanoTime();

**double** futureValueMemoized = FinancialForecast.calculateFutureValueMemoized(presentValue, growthRate, periods,

cache);

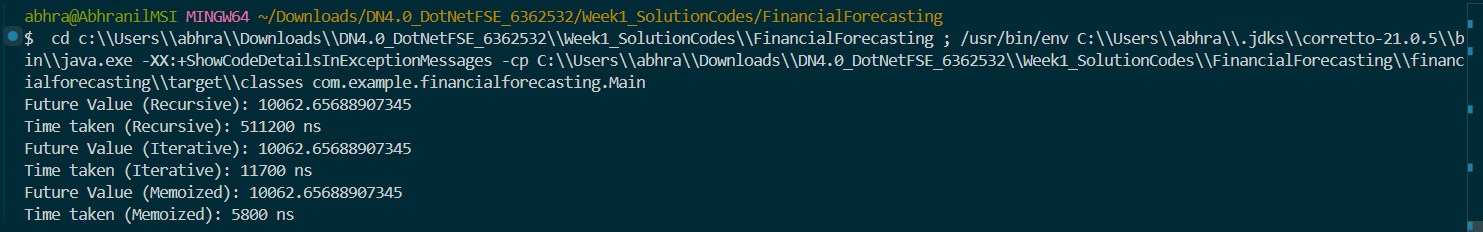
**long** endMemo = System.nanoTime();

System.out.println("Future Value (Memoized): " + futureValueMemoized);

System.out.println("Time taken (Memoized): " + (endMemo - startMemo) + " 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 |