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

**What is Big O Notation?**

Big O notation describes **how the time or space complexity of an algorithm grows** as the input size increases. It ignores constant factors and focuses on the dominant term affecting performance.  
It helps developers:

* Analyze algorithms independently of hardware.
* Compare different approaches for efficiency.
* Predict scalability as data size increases.

**Best, Average, Worst Cases for Search**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Best Case** | **Average Case** | **Worst Case** |
| **Linear Search** | O(1) (first element) | O(n) | O(n) |
| **Binary Search** | O(1) (middle element) | O(log n) | O(log n) |

* **Linear Search:** Simple but slower for large datasets.
* **Binary Search:** Faster but requires the data to be **sorted**.

**Product Class**

public class Product {

int productId;

String productName;

String category;

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

}

**LinearSearch.java**

public class LinearSearch {

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

for (int i = 0; i < products.length; i++) {

if (products[i].productId == targetId) {

return i;

}

}

return -1; // Not found

}

}

**BinarySearch.java**

import java.util.Arrays;

import java.util.Comparator;

public class BinarySearch {

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

int left = 0;

int right = products.length - 1;

while (left <= right) {

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

if (products[mid].productId == targetId) {

return mid;

} else if (products[mid].productId < targetId) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return -1; // Not found

}

public static void sortProducts(Product[] products) {

Arrays.sort(products, Comparator.comparingInt(p -> p.productId));

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

Product[] products = {

new Product(3, "Keyboard", "Electronics"),

new Product(1, "Laptop", "Electronics"),

new Product(2, "Mouse", "Electronics"),

new Product(5, "T-Shirt", "Apparel"),

new Product(4, "Shoes", "Footwear")

};

int targetId = 5;

int linearIndex = LinearSearch.linearSearch(products, targetId);

if (linearIndex != -1) {

System.out.println("Linear Search: Found " + products[linearIndex].productName);

} else {

System.out.println("Linear Search: Product not found.");

}

BinarySearch.sortProducts(products);

int binaryIndex = BinarySearch.binarySearch(products, targetId);

if (binaryIndex != -1) {

System.out.println("Binary Search: Found " + products[binaryIndex].productName);

} else {

System.out.println("Binary Search: Product not found.");

}

}

}

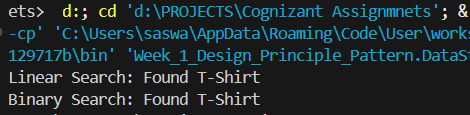
**Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| **Search Method** | **Time Complexity** | **Pros** | **Cons** |
| **Linear Search** | O(n) | Easy to implement, works on **unsorted data** | Slow for large lists |
| **Binary Search** | O(log n) | Much faster for large data | Needs **sorted data** |

**Which is more suitable?**

For **large product lists**, **binary search** is more efficient because of its logarithmic time complexity — but only works if your product list is sorted by the key you’re searching for (e.g., productId).

**Expected Output**



**Exercise 7: Financial Forecasting**

**What is Recursion?**  
Recursion is a programming technique where a method **calls itself** to solve a problem by breaking it down into **smaller subproblems**.

**How does it help?**

* Makes code **clean and elegant** for problems that naturally have repetitive patterns (like computing factorials, Fibonacci, or future values over time).
* Instead of using loops, recursion reduces problems step by step until reaching a **base case**.

**Example:**  
Calculating future value recursively means you:

* Take the present value
* Multiply by (1 + growthRate)
* Repeat for timePeriod - 1 years

**Implementation**

**ForecastingTool.java**

package com.FinancialForecastingTool;

public class ForecastingTool {

public double futureForecast(double presentValue, double growthRate, int timePeriod) {

// Base case: if no more years, return present value

if (timePeriod <= 0) {

return presentValue;

}

// Recursive case: grow once, then forecast for remaining years

return futureForecast(presentValue, growthRate, timePeriod - 1) \* (1 + growthRate);

}

}

**FinancialTool.java**

package com.FinancialForecastingTool;

public class FinancialTool {

private double futureValue;

public void financialForecast(double presentValue, double growthRate, int timePeriod) {

ForecastingTool forecastTool = new ForecastingTool();

futureValue = forecastTool.futureForecast(presentValue, growthRate, timePeriod);

System.out.println("Principal Amount = " + String.format("%.2f", presentValue)

+ ", Time Period = " + timePeriod

+ ", Annual Growth Rate = " + growthRate

+ ", Future Value = " + String.format("%.2f", futureValue) + "\n");

}

}

**Main.java**

package com.FinancialForecastingTool;

public class Main {

public static void main(String[] args) {

FinancialTool ft = new FinancialTool();

ft.financialForecast(15000.00, 0.055, 10);

ft.financialForecast(85000.50, 0.045, 7);

ft.financialForecast(37500.75, 0.062, 5);

ft.financialForecast(60000.00, 0.038, 15);

}

}

**Analysis**

**Time Complexity**

* For timePeriod = n, the method makes **n recursive calls**.
* So, **Time Complexity = O(n)**

**Space Complexity**

* Each recursive call adds a **frame to the call stack**.
* So, **Space Complexity = O(n)**

**Optimization**

**Problem:**  
For very large timePeriod values, deep recursion can cause **stack overflow**.

**Solution:**  
Use an **iterative approach** instead of recursion:

public double futureForecastIterative(double presentValue, double growthRate, int timePeriod) {

double result = presentValue;

for (int i = 0; i < timePeriod; i++) {

result \*= (1 + growthRate);

}

return result;

}

Same **O(n)** time complexity, but only **O(1)** space — safe for large time periods.

**Expected Output**

