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

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

It helps evaluate **how well an algorithm scales** as the input grows.

It allows comparison of different algorithms regardless of hardware or programming language.

**Best Case**: The scenario where the algorithm performs the fewest operations (e.g., item found at the first index).

**Average Case**: The expected performance of the algorithm over many runs with varying input positions.

**Worst Case**: The scenario where the algorithm does the most work (e.g., item not present or found at the end).

**Linear Search**:

* Best case: O(1) — item found at the beginning.
* Worst case: O(n) — item found at the end or not found.

**Binary Search**:

* Best case: O(1) — item found at the middle.
* Average and worst case: O(log n) — halves the search space on each step.

Binary search requires the input array to be **sorted**, whereas linear search does not.

Product.java

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;

}

public String toString() {

return "Product[ID=" + productId + ", Name=" + productName + ", Category=" + category + "]";

}

}

SearchUtils.java

public class SearchUtils {

// Linear Search by product name

public static Product linearSearch(Product[] products, String name) {

for (Product product : products) {

if (product.productName.equalsIgnoreCase(name)) {

return product;

}

}

return null;

}

// Binary Search by product name (array must be sorted by name)

public static Product binarySearch(Product[] products, String name) {

int left = 0;

int right = products.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

int comparison = products[mid].productName.compareToIgnoreCase(name);

if (comparison == 0) {

return products[mid];

} else if (comparison < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

}

Main.java

import java.util.Arrays;

import java.util.Comparator;

public class Main {

public static void main(String[] args) {

Product[] products = {

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

new Product(102, "Shirt", "Apparel"),

new Product(103, "Book", "Stationery"),

new Product(104, "Phone", "Electronics"),

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

};

// Sort for binary search

Arrays.*sort*(products, Comparator.*comparing*(p -> p.productName.toLowerCase()));

// Linear Search

Product linearResult = SearchUtils.linearSearch(products, "Phone");

System.***out***.println("Linear Search Result: " + (linearResult != null ? linearResult : "Not Found"));

// Binary Search

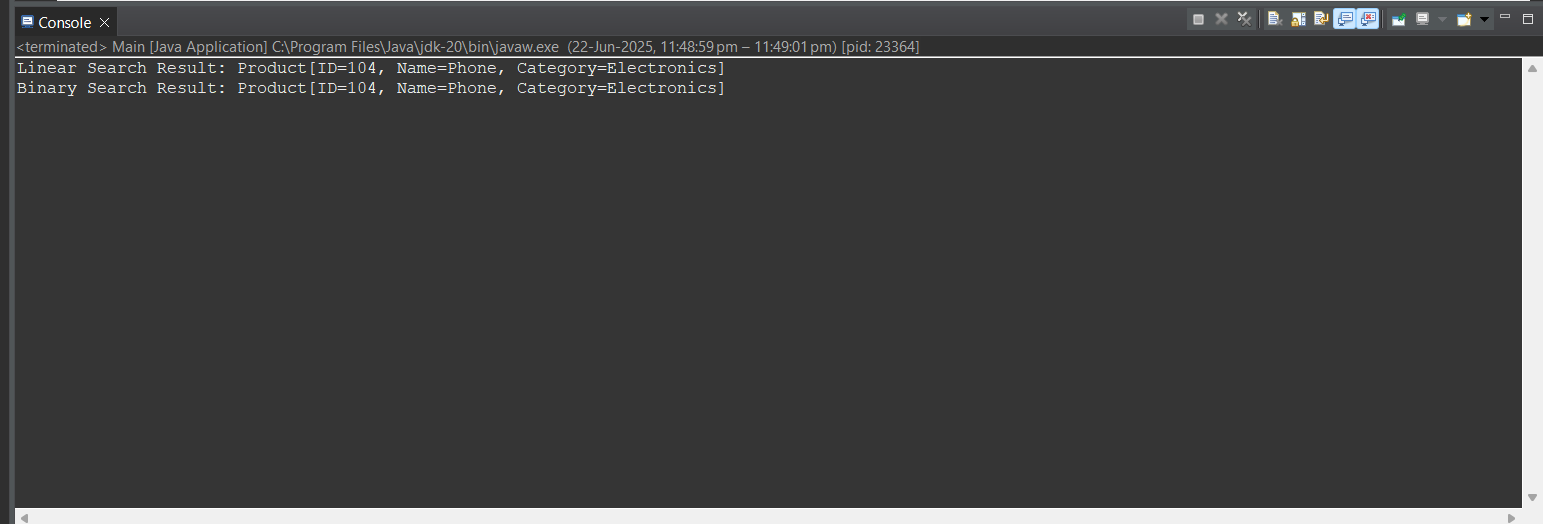
Product binaryResult = SearchUtils.binarySearch(products, "Phone");

System.***out***.println("Binary Search Result: " + (binaryResult != null ? binaryResult : "Not Found"));

}

}

**Output:**

****

**Analysis**

* **Linear Search:**
  + Time complexity: **O(n)** — goes through each element one by one.
  + Works on **unsorted data**, so it's more flexible.
  + Becomes inefficient with **large datasets**.
  + Useful for **small lists** or when sorting isn’t practical.
* **Binary Search:**
  + Time complexity: **O(log n)** — repeatedly divides the array in half.
  + Requires the array to be **sorted** in advance.
  + Performs much faster on **large datasets**.
  + Ideal when data changes infrequently or is already sorted.
* **Comparison:**
  + Binary search is significantly **faster** than linear search as the number of items increases.
  + Sorting (if needed) adds extra cost, but it’s worth it for repeated searches.
  + For dynamic datasets or one-time searches, linear search may be simpler to implement.
* **Conclusion:**
  + Use **binary search** for large, static datasets where fast search is essential.
  + Use **linear search** for small or frequently updated datasets.

# **Exercise 7: Financial Forecasting**

**Understanding Recursion**

* Recursion is a method that calls itself with a smaller subproblem until it hits a base case.
* It's useful when a problem naturally breaks down into smaller versions of itself.
* In financial forecasting, recursion can be used to simulate the compounding of value over a given number of periods.

FinancialForecast.java

public class FinancialForecast {

// Recursive method to calculate future value

public static double forecastValue(double currentValue, double growthRate, int periods) {

if (periods == 0) {

return currentValue; // base case

}

return *forecastValue*(currentValue, growthRate, periods - 1) \* (1 + growthRate);

}

public static void main(String[] args) {

double initialValue = 1000.0;

double growthRate = 0.05; // 5% annual growth

int years = 5;

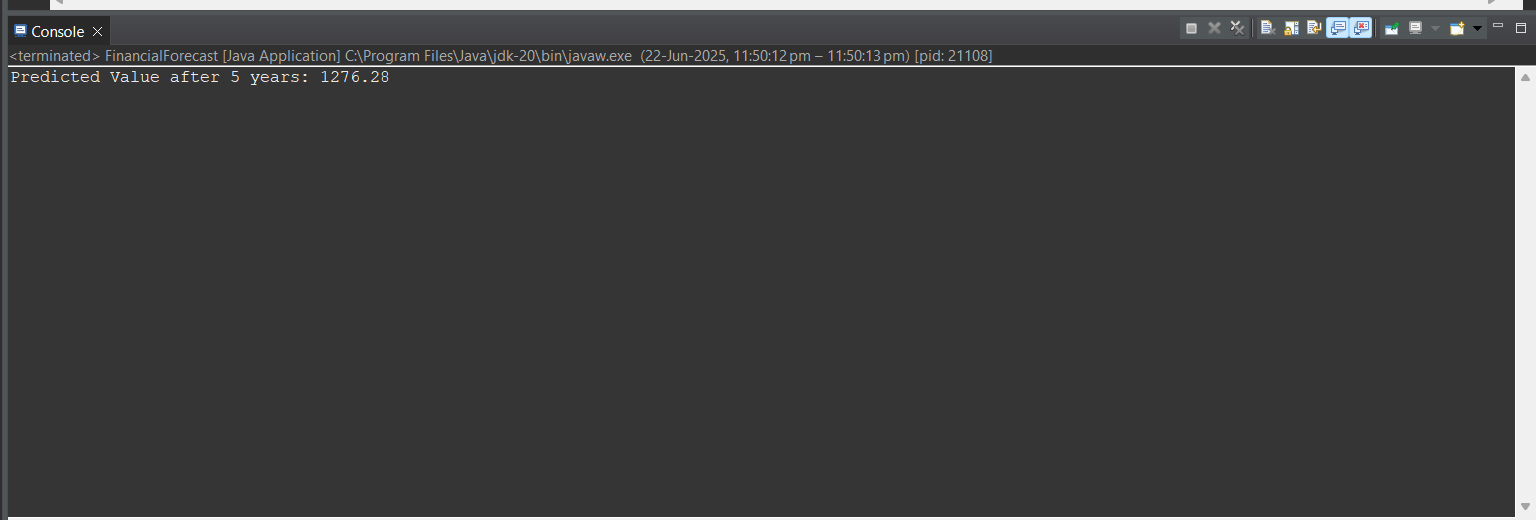
double predictedValue = *forecastValue*(initialValue, growthRate, years);

System.***out***.printf("Predicted Value after %d years: %.2f\n", years, predictedValue);

}

}

Output:



Analysis

**Time Complexity:**  
The recursive function runs once for each period, so the time complexity is **O(n)**, where n is the number of periods.

**Drawback:**  
Java has a limited stack size. Too many recursive calls (like thousands of years) may cause a **StackOverflowError**.

**Optimization:**  
We can avoid recursion completely and use a loop for better performance:

public static double forecastValueIterative(double currentValue, double growthRate, int periods) {

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

currentValue \*= (1 + growthRate);

}

return currentValue;

}