**2 E-commerce Platform Search Function**

**Step 1 : Understanding Asymptotic Notation:**

Big O notation is used to describe the time or space complexity of an algorithm as the input size (n) grows. It helps developers understand the efficiency of an algorithm, especially with large datasets.

**Best Case**

* Item found immediately. first element or middle.
* **Time:**
  + Linear Search: O(1)
  + Binary Search: O(1)

**Average Case**

* Item found somewhere in the middle.
* **Time:**
  + Linear Search: O(n)
  + Binary Search: O(log n)

**Worst Case**

* Item is last or not found at all.
* **Time:**
  + Linear Search: O(n)
  + Binary Search: O(log n)

**Step 2 : SetUp**

**Product.java**

package com.ecommerce.search;

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;

}

@Override

public String toString() {

return "Product ID: " + productId + ", Name: " + productName + ", Category: " + category;

}

}

**SearchDemo.java**

package com.ecommerce.search;

import java.util.Arrays;

import java.util.Comparator;

public class SearchDemo {

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

**for** (Product p : products) {

**if** (p.productName.equalsIgnoreCase(name)) {

**return** p;

} }

return null; }

**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** cmp = products[mid].productName.compareToIgnoreCase(name);

**if** (cmp == 0) return products[mid];

**else** **if** (cmp < 0) left = mid + 1;

**else** right = mid - 1;

}

return null;

}

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

Product[] products = {

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

new Product(102, "Shoes", "Footwear"),

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

new Product(104, "T-Shirt", "Clothing"),

new Product(105, "Headphones", "Electronics")

};

Product result1 = *linearSearch*(products, "Phone");

System.*out*.println("Linear Search Result: " + (result1 != null ? result1 : "Not Found"));

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

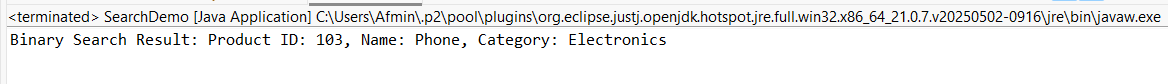
Product result2 = *binarySearch*(products, "Phone");

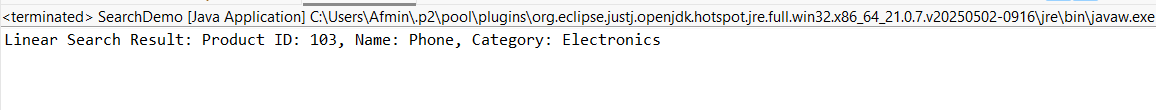
System.*out*.println("Binary Search Result: " + (result2 != null ? result2 : "Not Found"));

}

}

**Step 3 : Implementation**





**Step 4: Comparison of Time Complexity**

Linear Search:

* + Best Case: O(1)
  + Average/Worst Case: O(n)
  + No need to sort data

Binary Search:

* + Best Case: O(1)
  + Average/Worst Case: O(log n)
  + Requires sorted data
* Binary Search is better for large e-commerce platforms.
* Faster performance on large datasets.
* Ideal when product data is sorted.
* Ensures quicker search results for users.

So, Binary Search is suitable for e-commerce platforms  
  
**7 : Financial Forecasting  
  
Recursive Algorithms**

Recursion is when a function calls itself to solve a smaller version of the same problem.

It simplifies problems that involve repeated calculations over time (like financial growth).

Example: calculating compound interest year by year.

**SetUp**

Formula used:

Future Value = Present Value \* (1+rate)^n

Create a method that applies this formula recursively over n years.

**Implementation**

public class FinancialForecast {

public static double futureValue(double presentValue, double rate, int years) {

if (years == 0) {

return presentValue;

}

return futureValue(presentValue \* (1 + rate), rate, years - 1);

}

public static void main(String[] args) {

double presentValue = 10000;

double growthRate = 0.08;

int years = 5;

double result = futureValue(presentValue, growthRate, years);

System.out.println(“Future Value after “+years: “+result);

}

}

**Analysis**

**Time Complexity**

* O(n) = One recursive call per year.
* Space Complexity: O(n) due to recursive call stack.

**Optimization**

* Use iteration instead of recursion to:
  + Improve performance.
  + Avoid stack overflow for large n.

**Optimized Version**

public static double futureValueIterative(double presentValue, double rate, int years) {

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

presentValue \*= (1 + rate);

}

return presentValue;}

**INPUT**

double presentValue = 10000;

double rate = 0.08;

int years = 5

**OUTPUT**  
  
Future Value after 5 years: ₹14693.28