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

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**SOLUTION:**

1. Understand Asymptotic Notation:

**Big O:**

* Big O Notation describes the upper bound of the **time (or space) complexity** of an algorithm.
* It helps to measure the efficiency of an algorithm.

**Best , Average and Worst Case Scenarios for search operations:**

|  |  |  |
| --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |
|  |  |  |

|  |  |  |
| --- | --- | --- |
| Case | Linear Search | Binary Search |
| Best | O(1) (first element) | O(1) (middle element) |
| Average | O(n/2) = O(n) | O(log n) |
| Worst | O(n) | O(log n) |

Product.java  
  
package searchFunctionality;

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 [productId=" + productId + ", productName=" + productName + ", category=" + category + "]";

}

}

LinearSearch.java  
  
package searchFunctionality;

public class LinearSearch {

public static Product search(Product[] products,String target) {

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

if(products[i].productName.equalsIgnoreCase(target)) {

return products[i];

}}

return null;

}  
}

BinarySearch.java

package searchFunctionality;

import java.util.Arrays;

import java.util.Comparator;

public class BinarySearch {

//sorting

public static void sortByName(Product[] products) {

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

}

//binary search

public static Product search(Product[] products, String target) {

int left=0;

int right=products.length-1;

while(left<=right) {

int mid=(left+right)/2;

int cmp=products[mid].productName.compareToIgnoreCase(target);

if(cmp==0)

return products[mid];

else if (cmp<0)

left=mid+1;

else

right=mid-1;

}

return null;

}

}

Main.java

package searchFunctionality;

public class Main {

public static void main(String[] args) {

Product[] products = {

new Product(1, "Jeans", "Outfits"),

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

new Product(3, "Shoes", "Fashion"),

new Product(4, "Watch", "Accessories"),

new Product(5, "Sneakers", "Footware")

};

//Linear Search

Product result1 = LinearSearch.*search*(products, "Camera");

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

//Binary Search

BinarySearch.*sortByName*(products);

Product result2 = BinarySearch.*search*(products, "dress");

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

}

}

**4. Analysis based on Time Complexity:**

| Case | Linear Search | Binary Search |
| --- | --- | --- |
| Best | O(1) | O(1) |
| Average | O(n) | O(log n) |
| Worst | O(n) | O(log n) |

Which one is better?

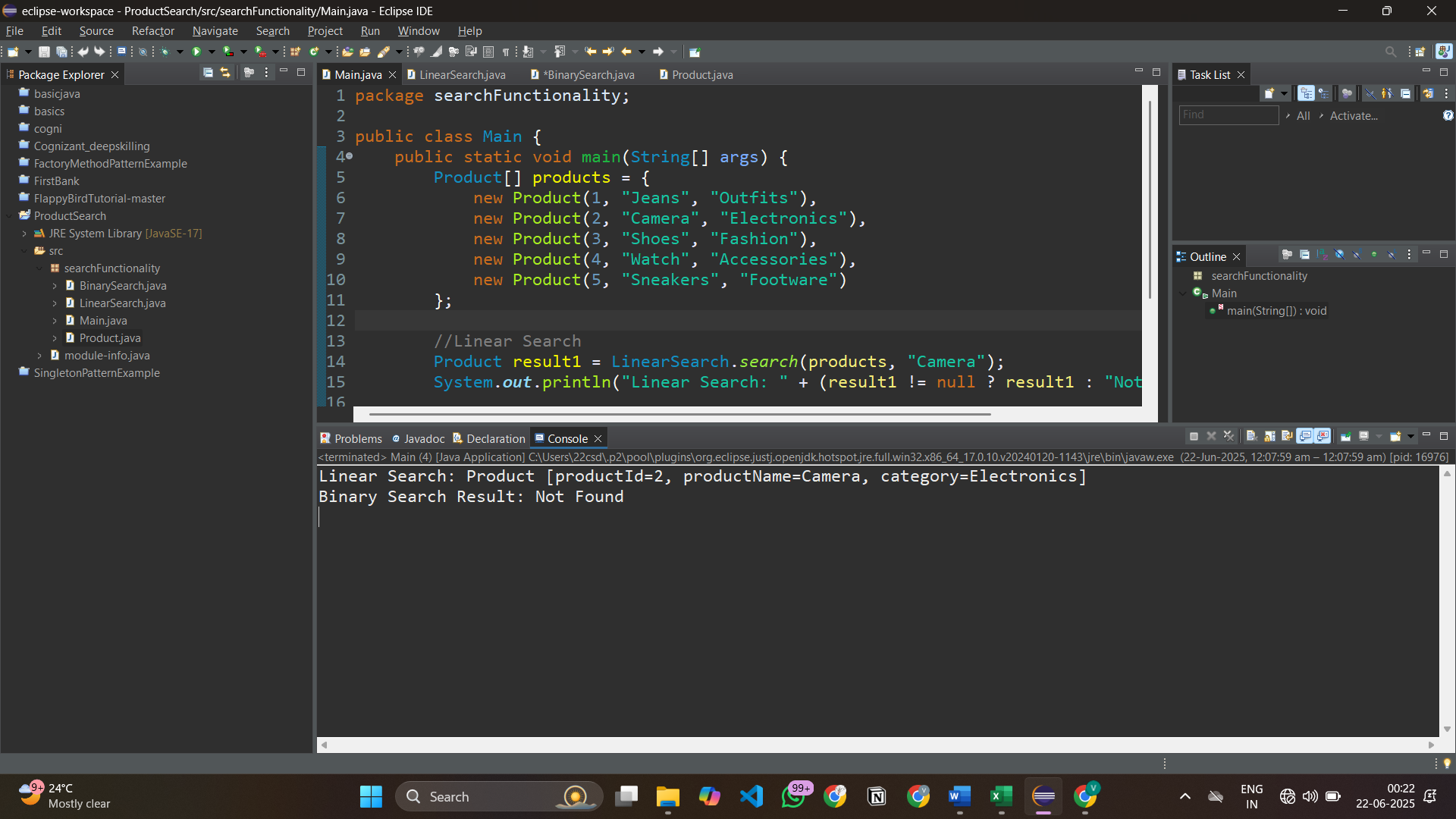
**Use Linear Search:**

* When the list is unsorted.
* When the data is small.

**Use Binary Search:**

* When the list is sorted.
* For larger data sets.

**OUTPUT:**



**Exercise 7: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

**SOLUTION:**

1. Understand Recursive Algorithms:

**Recursion:**

* Recursion is when a method (or function) calls itself to solve a problem in smaller parts.
* It simplifies complex problems (like tree traversal, factorial, Fibonacci).
* Writing cleaner, shorter, and more understandable code is made possible by recursion, which can make some problems easier, particularly those that require repeating a process on smaller components.

**Structure of a Recursive Function:**

* Base Case – When to stop (avoids infinite loop)
* Recursive Case – The function calls itself with a smaller input

**Example:**

int factorial(int n) {

if (n == 0) return 1; // base case

return n \* factorial(n-1); // recursive case

}

RecursiveFunction.java

package predictFuture;

public class RecursiveFunction {

//Recursive function to calculate future value

public static double Value(double initialAmt, double growthRate, int years) {

//Base case

if (years == 0) {

return initialAmt;

}

//Recursive case

return *Value*(initialAmt, growthRate, years - 1) \* (1 + growthRate);

}

public static void main(String[] args) {

double initialAmount = 5000.0;

double growthRate = 0.80;

int years = 7;

double future = *Value*(initialAmount, growthRate, years);

System.***out***.printf("Future value after %d years: ₹%.2f", years, future);

}

}

4. Analysis:

**Time Complexity:**

* O(n) where n=number of years

**Optimization:**

* Using loop instead of recursion.

public static double Value(double initialAmt, double growthRate, int years){

double val=initalAmt;

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

val=val\*(1+growthRate);

}

return val;

}

OUTPUT:

