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

1. Explain Big O notation and how it helps in analyzing algorithms

**Big O Notation** is a mathematical concept used to describe the **time complexity** and **space complexity** of an algorithm as the input size grows.

**Example:**

A **Linear Search** takes up to **O(n)** time.

A **Binary Search** takes up to **O(log n)** time.

1. Describe the best, average, and worst-case scenarios for search operations

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

* + Best Case: When the element is found immediately.
  + **Average Case:** Element is somewhere in the middle.
  + **Worst Case:** Element is at the end or not present at all.

**CODE:**

Product.java

package com.example.search;

public class Product {

int prodID;

String prodName;

String Category;

public Product(int prodId, String prodName,String Category) {

this.prodID=prodID;

this.prodName=prodName;

this.Category=Category;

}

}

SearchUtil.java

package com.example.search;

import java.util.Arrays;

import java.util.Comparator;

public class SearchUtil {

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

for(Product p : products) {

if(p.prodName.equalsIgnoreCase(targetName)) {

return p;

}

}

return null;

}

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

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

int low=0,high=products.length-1;

while(low<=high)

{

int mid=(low+high)/2;

int compare=products[mid].prodName.compareToIgnoreCase(targetName);

if(compare==0)return products[mid];

else if(compare<0) low=mid+1;

else high=mid-1;

}

return null;

}

}

Main.java

package com.example.search;

public class Main {

public static void main(String[] args) {

Product[] products= {

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

new Product(202, "Shirt", "Clothing"),

new Product(303, "Phone", "Electronics"),

new Product(404, "Book", "Stationery")

};

Product res1=SearchUtil.*linearSearch*(products,"phone");

if(res1!=null) {

System.*out*.println("Linear Search Found: "+res1.prodName );

}

else

{

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

}

Product res2=SearchUtil.*binarySearch*(products, "Book");

if (res2 != null)

System.*out*.println("Binary Search Found: " + res2.prodName);

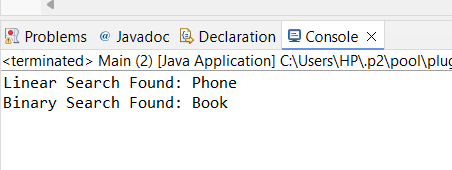
else

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

}

}

OUTPUT:



**Exercise 7: Financial Forecasting**

1. Explain the concept of recursion and how it can simplify certain problems

**Recursion** is a programming technique where a function calls itself to solve smaller instances of a problem until it reaches a base case. It is useful when a problem can be broken down into subproblems that are similar in nature.

**Example:**

In financial forecasting, if we want to predict the future value of an investment over multiple years, recursion allows us to calculate each year's growth based on the previous year’s value, making the solution more intuitive and cleaner.

**CODE:**

**FinancialForecaster.java**

package com.example.finance;

public class FinancialForecaster {

public static double predictFutureValue(double currVal, double growthRate,int years) {

if(years==0) {

return currVal;

}

return *predictFutureValue*(currVal\*(1+growthRate),growthRate,years-1);

}

public static void main(String[] args) {

double currVal=1000.0;

double growthRate=0.10;

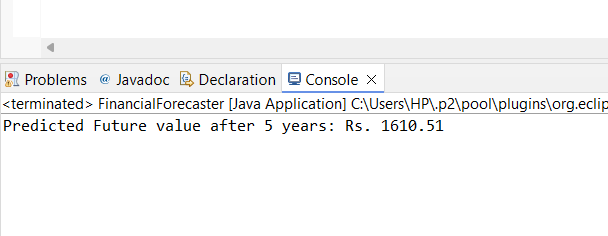
int years=5;

double futureVal=*predictFutureValue*(currVal,growthRate,years);

System.*out*.printf("Predicted Future value after %d years: Rs. %.2f\n",years,futureVal);

}

}

**OUTPUT:**  


1. Discuss the time complexity of your recursive algorithm

The **time complexity** of the recursive forecasting algorithm is:

* **O(n)** where n is the number of years.
* One recursive call and one multiplication is performed per year.
* So, for 5 years → 5 recursive calls.

1. Explain how to optimize the recursive solution to avoid excessive computation

Recursive solutions can lead to **stack overflow** or **redundant calculations** in some problems. To optimize

* 1. Replace recursion with iteration

public static double predictFutureValueIterative(double currentValue, double growthRate, int years) {

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

currentValue \*= (1 + growthRate);

}

return currentValue;

}

* 1. Use **memorization**
     + Store results of previously computed values in a map or array to avoid recomputation.
     + Not needed in this simple forecasting problem, but useful in problems like Fibonacci series.