**Algorithms\_Data Structures**

**Exercise 1:** E-commerce Platform Search Function

Efficient search functionality is critical for delivering a fast and user-friendly experience in an e-commerce platform. This report explores the implementation and analysis of two common search algorithms—**Linear Search** and **Binary Search**—and evaluates which is better suited for an optimized product lookup.

**Scenario:**

The task is to develop and compare search techniques in an e-commerce platform where users look up products based on attributes like name or category. The solution should be performance-efficient, particularly for large datasets.

 **Linear Search**: Checks each product one by one.

 **Binary Search**: Requires sorted array and repeatedly divides the search space in half.

**Program:**

Product.java

package searchFunction;

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 productId + " " + productName + " (" + category + ")";

}

}

ProductSearch.java

package searchFunction;

import java.util.Arrays;

import java.util.Comparator;

import java.util.Scanner;

public class productSearch {

//Linear Search

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

for(Product prod:products) {

if(prod.productName.equalsIgnoreCase(targetProduct)) {

return prod;

}

}

return null;

}

//Binary Search

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

int low = 0;

int high = products.length - 1;

while(low<=high) {

int mid=(low+high)/2;

int compare=products[mid].productName.compareToIgnoreCase(targetProduct);

if(compare==0) {

return products[mid];

}

else if(compare<0) {

low=mid+1;

}

else {

high=mid-1;

}

}

return null;

}

public static void main(String[] args) {

Product[] products = {

new Product(101, "Kurta set", "Clothing"),

new Product(102, "Fan", "Electronics"),

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

new Product(104, "Sunscream", "Personal Care"),

new Product(105, "Jeans", "Clothing")

};

Scanner s=new Scanner(System.***in***);

System.***out***.println("Search by product name");

String item=s.nextLine();

Product result1=*linearSearch*(products,item);

System.***out***.println("Linear Search result");

System.***out***.println(result1!=null?result1:"Not found");

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

System.***out***.println("Binary Search result");

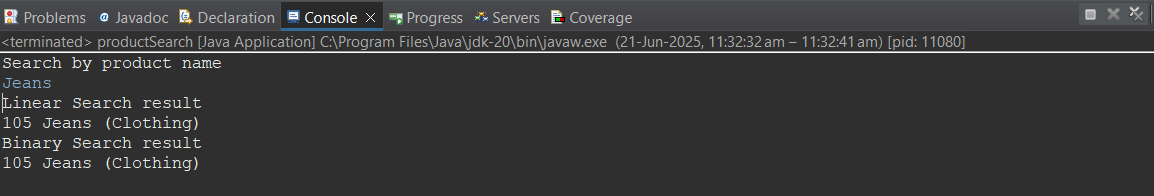
Product result2=*binarySearch*(products,item);

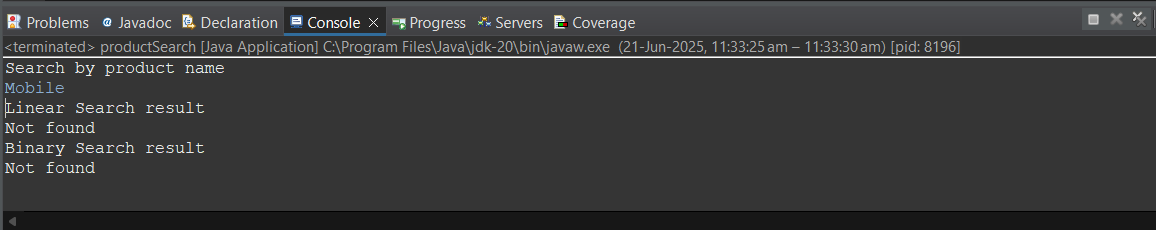
System.***out***.println(result2!=null?result2:"Not found");

}

}

**Output:**





**Comparision of Linear and Binary Search Algorithms:**

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

For small, unsorted datasets, **linear search** is acceptable. However, for real-world e-commerce platforms handling **thousands of products**, **binary search** or even more advanced structures (like hash maps or search trees are preferred) for performance optimization.

**Exercise 2:** Financial Forecasting

Forecasting future values using past data is a common problem in financial analytics. Recursive algorithms can offer elegant solutions to problems where the result at any time step depends on results from previous steps. This report outlines the implementation of a recursive approach to financial forecasting and discusses its performance characteristics.

**Scenario:**

We are building a tool that forecasts future financial values based on historical growth rates using **recursion**. The goal is to compute the future value of an investment over a number of years, assuming a constant growth rate.

**Recursive Algorithms**

**Recursion** is a method where a function calls itself to solve smaller instances of the same problem. It is particularly useful for problems that can be broken down into simpler subproblems.

**Example Concept:**

Future Value in Year *n* = Future Value in Year *(n−1)* × (1 + Growth Rate)

Recursion simplifies the problem, making the code more readable and logically expressive.

The recursive method calculates the future value based on:

* Initial investment (principal)
* Growth rate
* Number of years

**Program:**

FinanceForecast.java

package finance;

public class FinancialForecast {

public static double calculateFutureValue(double principal, double rate, int years) {

if (years == 0) {

return principal;

}

return *calculateFutureValue*(principal, rate, years - 1) \* (1 + rate);

}

public static void main(String[] args) {

double principal = 15000.0;

double annualRate = 0.10;

int years = 3;

double futureValue = *calculateFutureValue*(principal, annualRate, years);

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

}

}

**Output:**

