|  |  |
| --- | --- |
| **WEEK 1 – Algorithms and Data Structures** | **Superset Id : 6429486**  **Name : Akshaya V** |

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

**Qn : Explain Big O notation and how it helps in analyzing algorithms.**

Big O Notation is a mathematical way to describe the efficiency of an algorithm in terms of time and space as the input size increases.

It tells you how the runtime or memory usage of an algorithm grows relative to the input size

It helps us to

* Compare algorithms regardless of hardware.
* Predict performance for large inputs.
* Optimize code by choosing more efficient algorithms.

**Qn : Describe the best, average, and worst-case scenarios for search operations.**

**1.Linear Search**

It goes through each item in the list one by one until it finds the target or reaches the end.

|  |  |  |
| --- | --- | --- |
| Best Case | The item is found at the first position | O(1) |
| Average Case | |  | | --- | |  |  |  | | --- | | The item is somewhere in the middle of the list | | O(n/2) |
| Worst Case | |  | | --- | |  |  |  | | --- | | The item is at the last position or not present at all | | O(n) |

**Example:**

Search for product Pencil in a array of 5 items

* Best Case: 1st item - 1 check
* Average Case: 3rd item - 3 checks
* Worst Case: Not in list - 5 checks

**2.Binary Search**

Requires a sorted array and it repeatedly divides the search range in half.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Best Case | |  | | --- | |  |  |  | | --- | | The item is exactly at the middle index. | | O(1) |
| Average Case | Item is somewhere in the list, requires log steps. | O(log n) |
| Worst Case | Item is not in the list, but all log n comparisons are needed. | O(log n) |

**Example:**

Search in a sorted array of 16 items

* Best Case: 8th item (middle) - 1 comparison
* Average Case: 3-4 comparisons
* Worst Case: Not in list – log2(16) = 4 comparisons

**Code:**

import java.util.\*;

public class EcommerceSearch{

    static class Product{

        int id;

        String name;

        String category;

        public Product(int id,String name,String category){

            this.id=id;

            this.name=name;

            this.category=category;

        }

        public String toString(){

            return "Product : ID="+id+",Name='"+name+"',Category='"+category;

        }

    }

    public static Product findProductLinear(Product[] products,int searchId){

        for(Product p:products){

            if(p.id==searchId)return p;

        }

        return null;

    }

    public static Product findProductBinary(Product[] products,int searchId){

        int start=0,end=products.length-1;

        while(start<=end){

            int mid=(start+end)/2;

            if(products[mid].id==searchId)return products[mid];

            else if(products[mid].id<searchId)start=mid+1;

            else end=mid-1;

        }

        return null;

    }

    public static void main(String[] args){

        Product[] inventory={

            new Product(55,"Kurtas","Clothing"),

            new Product(56,"Bracelet","Accessories"),

            new Product(57,"Keyboard","Electronics"),

            new Product(58,"Moisturizer","SkinCare"),

            new Product(59,"Slippers","Footwear")

        };

        System.out.println("6429486 - Akshaya V");

        System.out.println("All Products:");

        for(Product p:inventory){

            System.out.println(p);

        }

        Scanner sc=new Scanner(System.in);

        System.out.print("\nEnter Product ID to search:");

        int searchId=sc.nextInt();

        Product resultLinear=findProductLinear(inventory,searchId);

        System.out.println("\nLinear Search Result:"+(resultLinear!=null?resultLinear:"Product not found"));

        Arrays.sort(inventory,Comparator.comparingInt(p->p.id));

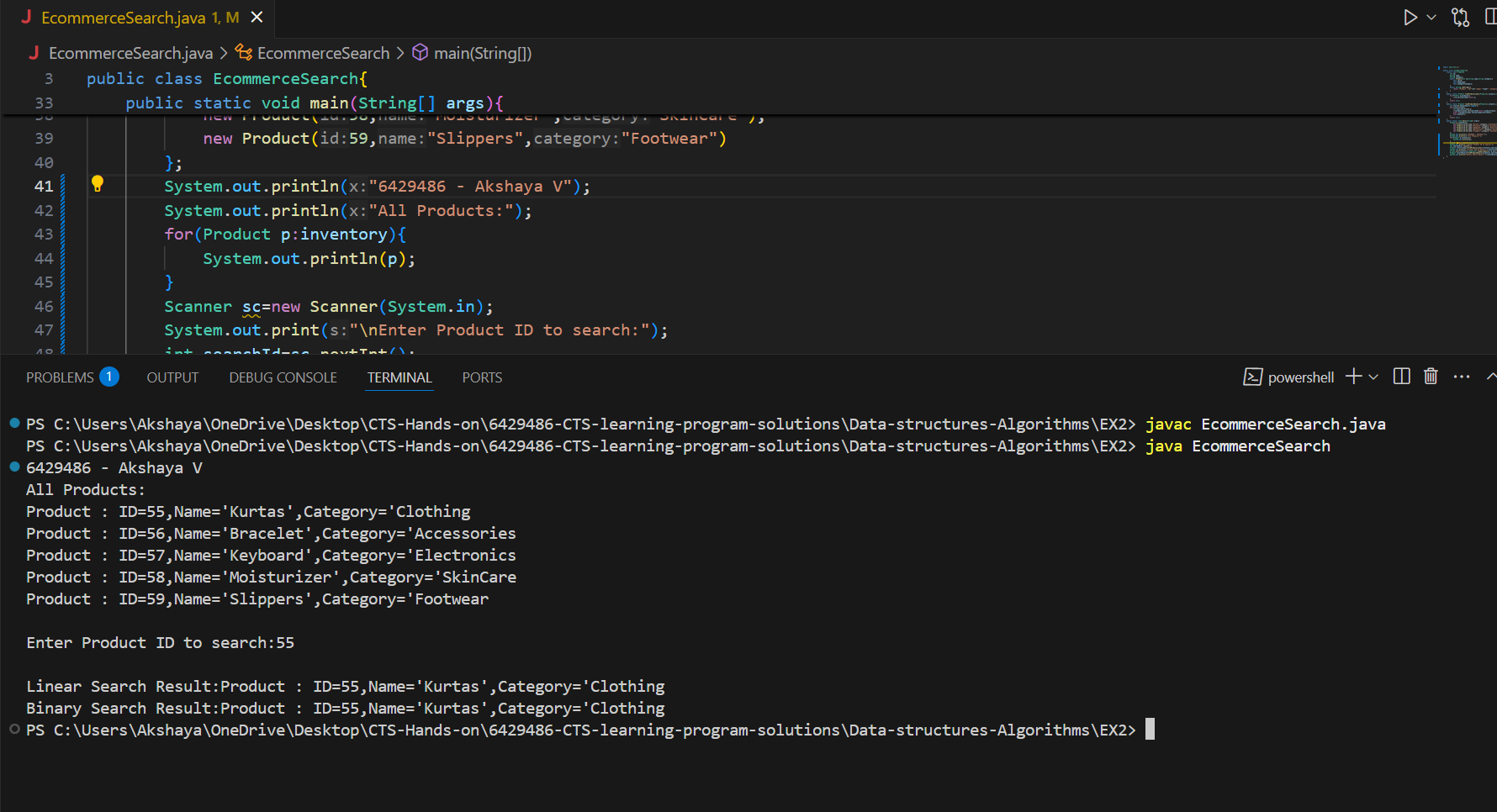
        Product resultBinary=findProductBinary(inventory,searchId);

        System.out.println("Binary Search Result:"+(resultBinary!=null?resultBinary:"Product not found"));

    }

}

**Output:**

****

**Qn : Compare the time complexity of linear and binary search algorithms.**

|  |  |  |
| --- | --- | --- |
| Criteria | Linear Search | Binary Search |
| Time Complexity | O(n) | O(log n) |
| Best Case | O(1) if target is 1st element | O(1) is target is middle element |
| Average Case | O(n/2)-O(n) | O(log n) |
| Worst Case | O(n) if target not present or last | O(log n) |

**Qn : Discuss which algorithm is more suitable for your platform and why.**

Binary Search is more suitable because:

* + E-commerce platforms usually deal with large product datasets.
  + Search operations are very frequent and must be fast and responsive.
  + Products can be pre-sorted by ID, name, or price, enabling efficient binary search.
  + Offers consistent O(log n) time for all cases (except best).

Linear Search is only preferable when:

* The product list is very small (e.g., under 10 items).
* Or the data is unsorted and sorting is not worth the overhead.

**Exercise 7: Financial Forecasting**

**Qn : Explain the concept of recursion and how it can simplify certain problems.**

Recursion is a programming concept where a function calls itself to solve a problem by breaking it into smaller subproblems.

Instead of writing repetitive code, recursion simplifies the logic and makes the solution cleaner and closer to mathematical formulas.

A recursive function typically has:

Base Case – the condition when recursion stops.

Recursive Case – where the function calls itself with a smaller input.  
Example:

Factorial program:

int factorial(int n){

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

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

}

Recursion helps us to ,

1. Simplifies complex problems

2. Reduces code size

3. Great for tasks like factorial, Fibonacci, tree/graph traversal

**Code:**

import java.util.Scanner;

public class FinancialForecast{

    public static double forecastRecursive(double presentValue,double rate,int periods){

        if(periods==0){

            return presentValue;

        }

        return forecastRecursive(presentValue,rate,periods-1)\*(1+rate);

    }

    public static double forecastIterative(double presentValue,double rate,int periods){

        double futureValue=presentValue;

        for(int i=1;i<=periods;i++){

            futureValue\*=(1+rate);

        }

        return futureValue;

    }

    public static void main(String[] args){

        Scanner sc=new Scanner(System.in);

        System.out.println("6429486 - Akshaya V");

        System.out.print("Enter present value(Rs): ");

        double presentValue=sc.nextDouble();

        System.out.print("Enter annual growth rate(in %): ");

        double ratePercent=sc.nextDouble();

        double rate=ratePercent/100.0;

        System.out.print("Enter number of years to forecast: ");

        int years=sc.nextInt();

        double resultRecursive=forecastRecursive(presentValue,rate,years);

        double resultIterative=forecastIterative(presentValue,rate,years);

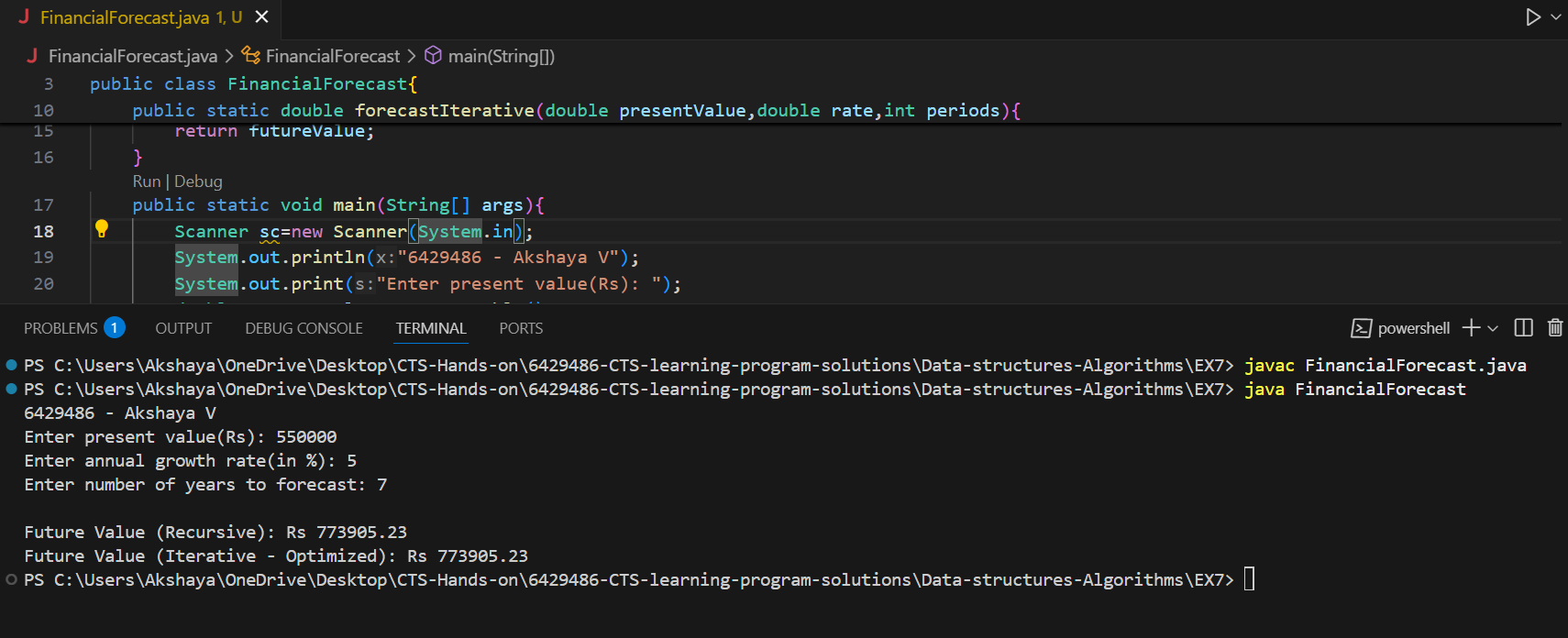
        System.out.printf("\nFuture Value (Recursive): Rs %.2f\n",resultRecursive);

        System.out.printf("Future Value (Iterative - Optimized): Rs %.2f\n",resultIterative);

    }

}

**Output:**



**Qn : Discuss the time complexity of your recursive algorithm.**

Time Complexity: O(n)

Let n = periods, The recursive method makes one call per year until

periods equals 0

Therefore, the number of recursive calls is exactly n

Each call does a single multiplication, so the work per call is constant (O(1))

Hence, the total time complexity is:

T(n) = T(n - 1) + O(1) 🡪 T(n) = O(n**)**

**Qn : Explain how to optimize the recursive solution to avoid excessive computation.**

**Optimization Technique-Use Mathematical Formula (Direct Computation)**

Formula for calculating compound growth,

FV=PV×(1+r)^n

**Optimised method:**

public static double forecastDirect(double presentValue, double rate, int periods){

return presentValue \* Math.pow(1 + rate, periods);

}

* Time Complexity: O(1) - most efficient
* Space Complexity: O(1)