Data structures and Algorithms

Exercise 2: E-commerce Platform Search Function

**Big O notation:**

**Big O notation** is a mathematical way to describe **how an algorithm's performance scales** as the input size grows.

It focuses on:

* **Time Complexity** – How long the algorithm takes.
* **Space Complexity** – How much memory it uses.

Scenarios for Search Algorithms

* **Best Case**: The item is found at the first attempt.
* **Average Case**: The item is somewhere in the middle.
* **Worst Case**: The item is not found, or it's at the end of the structure.

|  |  |  |  |
| --- | --- | --- | --- |
| **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) |

Import java.util.\*;

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+")";

    }

}

public class searchMethod {

    public static Product linearSearch(Product[] products,int id){

        for(Product p:products){

            if(p.productId==id)

            return p;

        }

        return null;

    }

    public static Product binarySearch(Product[] products,int id){

        int left=0;

        int n=products.length;

        int right=n-1;

        while(left<=right){

            int mid=(left+right)/2;

            if(products[mid].productId==id)

                return products[mid];

            else if(products[mid].productId<id)

                left=mid+1;

            else

                right=mid-1;

        }

        return null;

    }

    public static void main(String[] args) {

        Scanner sc=new Scanner(System.in);

        Product[] products={

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

            new Product(267, "Sneakers", "Fashion"),

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

            new Product(157, "Toaster", "Appliances"),

            new Product(123, "Bracelets", "Fashion")

        };

        Arrays.sort(products, new Comparator<Product>() {

            public int compare(Product a, Product b) {

                return a.productId - b.productId;

            }

        });

System.out.println("Enter ProductId to search: ");

        int searchId=sc.nextInt();

Product linearsearch=linearSearch(products, searchId);

        System.out.println("Linear Search Result:");

        System.out.println(linearsearch!=null ? linearsearch : "Product not found");

        Product binarysearch=binarySearch(products, searchId);

      System.out.println("Binary Search Result:");

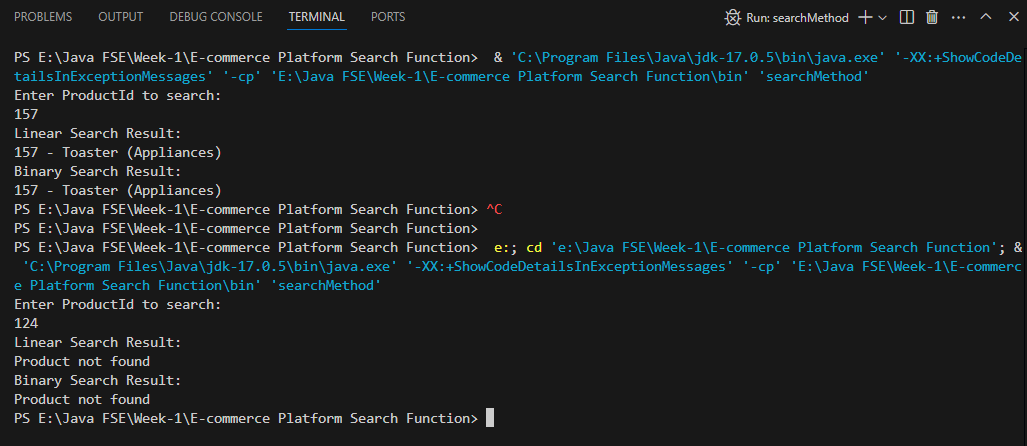
        System.out.println(binarysearch!=null ? binarysearch : "Product not found");

        sc.close();

    }

}

OUTPUT:



|  |
| --- |
|  |

|  |  |  |
| --- | --- | --- |
|  | **Linear search** | **Binary search** |
| Time Complexity | O(n) | O(log n) |

|  |
| --- |
|  |

**Small datasets or unsorted data** - **Linear Search** is acceptable.

**Large, sorted datasets** - **Binary Search** is preferred for faster performance.

For a real-world **e-commerce platform**, **binary search** is more suitable when data is **pre-sorted**, ensuring **fast search performance** with millions of products.

Exercise 7: Financial Forecasting

Recursion:

**Recursion** is a programming technique where a **function calls itself** to solve a problem.

It works by **breaking a complex problem into smaller sub problems** of the same type, solving each recursively until it reaches a simple base case that stops the recursion.

To Calculate Future value:

Using Simple Interest,

|  |
| --- |
| Future Value= P+ (P×R×T) / 100 |

Using Compound Interest,

|  |
| --- |
| Future Value=P×(1+100R​)T |

import java.util.Scanner;

public class FinancialForecast {

    public static double calculateSI(double principal, double rate, int years)

{

        if (years == 0) {

            return principal;

        }

        return calculateSI(principal, rate, years - 1)+(principal\*rate/100);

    }

public static double calculateCI(double principal, double rate, int years)

{

        if (years == 0) {

            return principal;

        }

        return calculateCI(principal, rate, years - 1) \* (1 + rate / 100);

    }

    public static void main(String[] args) {

        Scanner sc =new Scanner(System.in);

        System.out.println("Enter Starting amount:");

        double principal =sc.nextDouble();

        System.out.println("Enter Intrest rate:");

double rate = sc.nextDouble();

        System.out.println("Enter Number of years:");

        int years = sc.nextInt();

        double si = calculateSI(principal, rate, years);

        double ci = calculateCI(principal, rate, years);

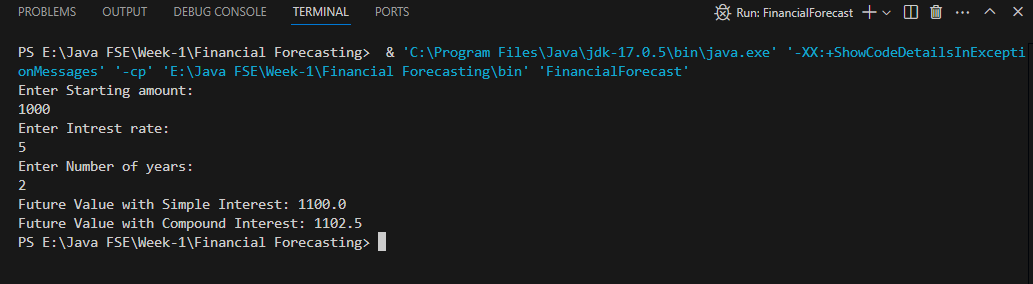
        System.out.println("Future Value with Simple Interest: " + si);

        System.out.println("Future Value with Compound Interest: " + ci);

    }

}

OUTPUT:



**Time Complexity** for recursive algorithm – **O (n)**

**Iteration** can be used to optimize the recursive solution to avoid excessive computation.