**Module -2(Algorithm\_ Data Structures)**

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

Big O notation is a mathematical way to describe the efficiency of an algorithm based on how input size increases.

Common Big O notations :

O(1) : Constant time

O(log n): Logarithmic time

O(n): Linear time

|  |  |
| --- | --- |
| **Best Case** | The ideal situation like element found immediately. |

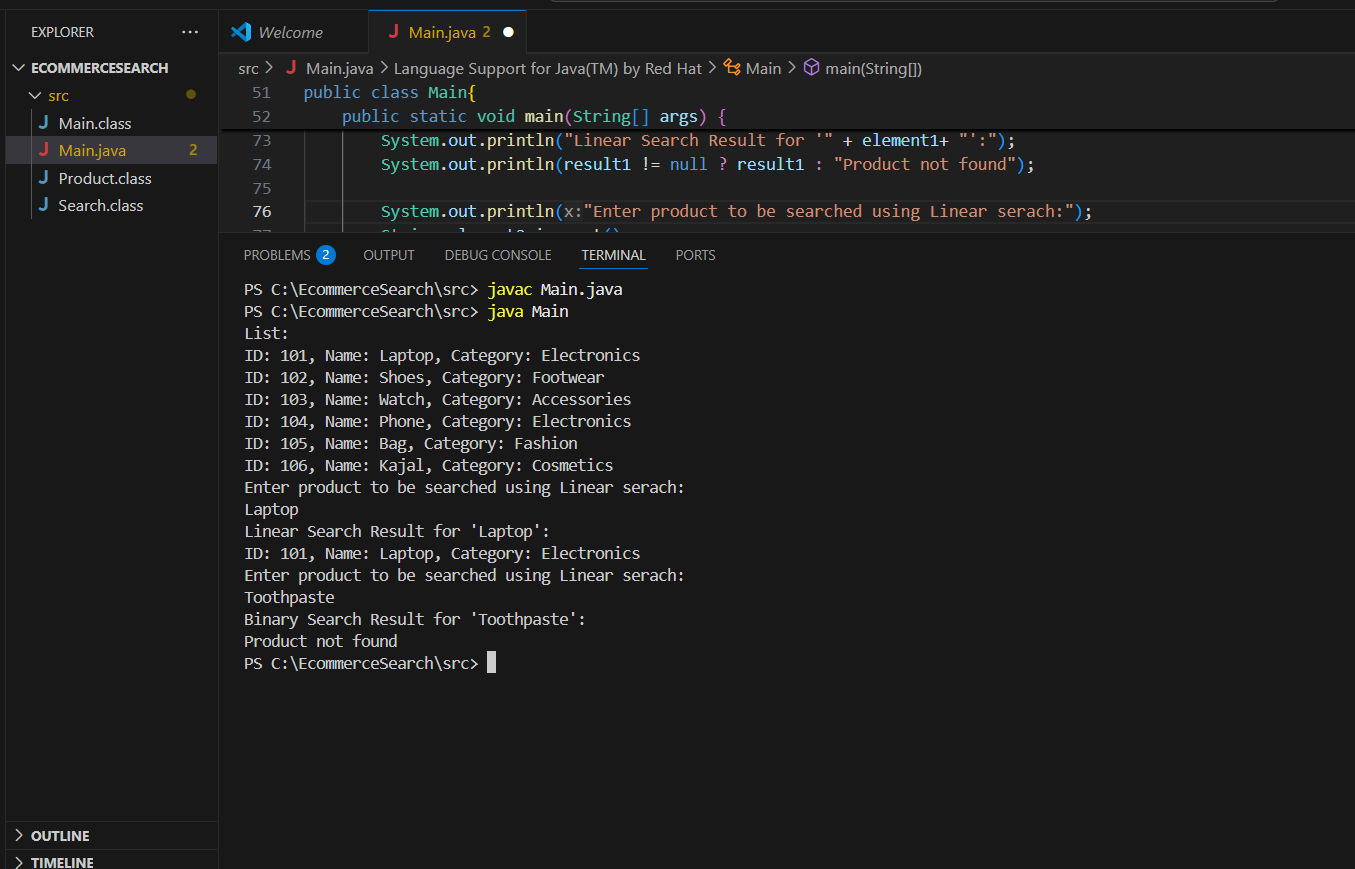
|  |  |
| --- | --- |
| **Average Case** | Expected time for any random inputs. |

|  |  |
| --- | --- |
| **Worst Case** | The slowest possible time like element not found or found at last position. |

Solution Code:

|  |
| --- |
| import java.util.\*;  class Product{     int productId;     String productName;     String category;     public Product(int id,String name,String category)     {      this.productId=id;      this.productName=name;      this.category=category;     }     public String toString() {          return "ID: " + productId + ", Name: " + productName + ", Category: " + category;      }  }   class Search{      public static Product linearSearch(Product[] products,String element)      {          for(Product product: products)          {              if(product.productName.equalsIgnoreCase(element))              {                  return product;              }          }          return null;      }      public static void sort(Product[] products)      {          Arrays.sort(products, Comparator.comparing(p -> p.productName.toLowerCase()));      }      public static Product binarySearch(Product[] products,String element)      {          int low=0;          int high=products.length-1;          element=element.toLowerCase();          while(low<=high)          {              int mid=(low+high)/2;              String midname=products[mid].productName.toLowerCase();              int val=element.compareTo(midname);              if(val == 0) return products[mid];              else if(val < 0) high=mid-1;              else low=mid+1;          }          return null;      }  }  public class Main{      public static void main(String[] args) {          Scanner in=new Scanner(System.in);          Product[] products={              new Product(101, "Laptop", "Electronics"),              new Product(102, "Shoes", "Footwear"),              new Product(103, "Watch", "Accessories"),              new Product(104, "Phone", "Electronics"),              new Product(105, "Bag", "Fashion"),              new Product(106, "Kajal", "Cosmetics")          };          System.out.println("List:");          for(Product i: products)          {                System.out.println(i);          }          Search.sort(products);          System.out.println("Enter product to be searched using Linear serach:");          String element1=in.next();          Product result1 = Search.linearSearch(products, element1);          System.out.println("Linear Search Result for '" + element1+ "':");          System.out.println(result1 != null ? result1 : "Product not found");          System.out.println("Enter product to be searched using Linear serach:");          String element2=in.next();          Product result2 = Search.binarySearch(products, element2);          System.out.println("Binary Search Result for '" + element2 + "':");          System.out.println(result2 != null ? result2 : "Product not found");      }  } |

Output:



Linear search has a time complexity of O(n),that means it checks for each element one by one, which becomes slow as the list grows. Binary search, on the other hand, has a time complexity of O(log n), as it repeatedly divides the search space into half, making it faster for large and sorted datasets.

For an e-commerce platform, binary search is generally more suitable because it is faster and more efficient for large datasets. However, this efficiency comes with the requirement that the product list must be sorted. Whereas linear search does not require sorting and works well on small or unsorted datasets, but it is not practical for high-performance systems.

**Exercise 7: Financial Forecasting**

Recursion is a programming technique where a method calls itself.

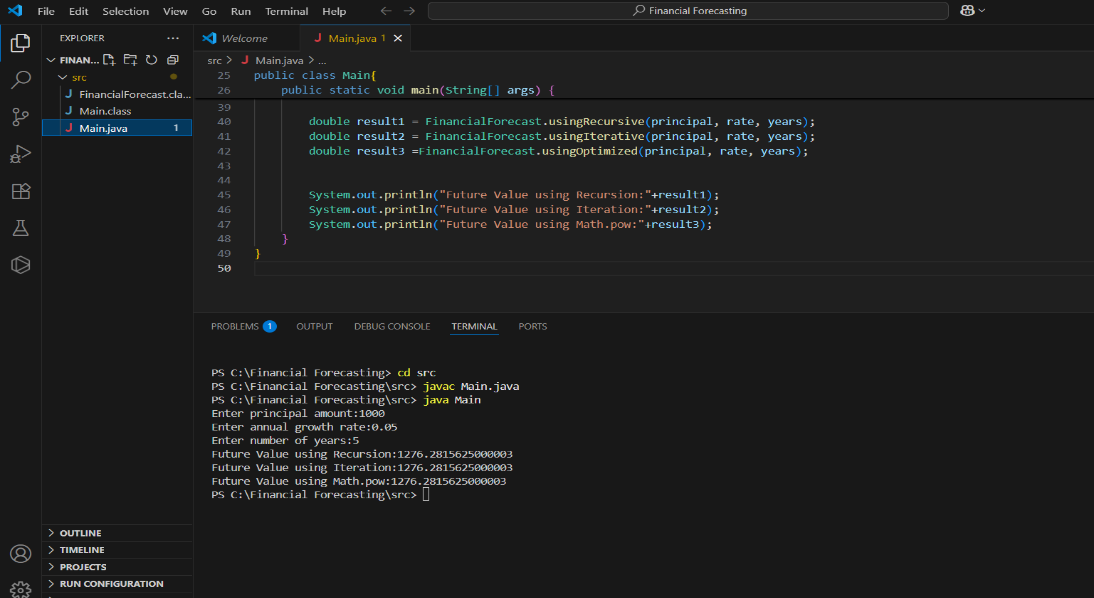
Recursion is useful when:

A problem can be broken down into similar subproblems.

**Solution Code:**

|  |
| --- |
| import java.util.Scanner;  class FinancialForecast {        public static double usingRecursive(double principal, double rate, int years) {          if (years == 0)              return principal;          return (1 + rate) \* usingRecursive(principal, rate, years - 1);      }        public static double usingIterative(double principal, double rate, int years) {          for (int i = 0; i < years; i++) {              principal \*= (1 + rate);          }          return principal;      }        public static double usingOptimized(double principal, double rate, int years) {          return principal \* Math.pow(1 + rate, years);      }  }  public class Main{      public static void main(String[] args) {          Scanner in = new Scanner(System.in);            System.out.print("Enter principal amount:");          double principal = in.nextDouble();          System.out.print("Enter annual growth rate:");          double rate = in.nextDouble();          System.out.print("Enter number of years:");          int years = in.nextInt();            double result1 = FinancialForecast.usingRecursive(principal, rate, years);          double result2 = FinancialForecast.usingIterative(principal, rate, years);          double result3 =FinancialForecast.usingOptimized(principal, rate, years);      System.out.println("Future Value using Recursion:"+result1);          System.out.println("Future Value using Iteration:"+result2);          System.out.println("Future Value using Math.pow:"+result3);      }  } |

**Output:**



The recursive algorithm has a time complexity of **O(n)** because it makes one call for each year. The most optimized solution is using Math.pow() because it calculates the future value in constant time, without using loops or recursion.