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

|  |
| --- |
| package DSA;  import java.util.Arrays;  import java.util.Comparator;  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;      }      public String toString() {          return "[" + productId + ", " + productName + ", " + category + "]";      }  }  class searchForProduct {      public static Product linearSearchMethod(Product[] products, String name) {          for (Product prods : products) {              if (prods.productName.equalsIgnoreCase(name)) {                  return prods;              }          }          return null;      }      public static Product binarySearchMethod(Product[] products, String name) {          int start = 0, end = products.length - 1;          while (start <= end) {              int mid = start + (end - start) / 2;              int cmp\_val = products[mid].productName.compareToIgnoreCase(name);              if (cmp\_val == 0) {                  return products[mid];              } else if (cmp\_val < 0) {                  start = mid + 1;              } else {                  end = mid - 1;              }          }          return null;      }      public static void main(String[] args) {          Product[] products = {                  new Product(101, "Laptop", "Electronics"),                  new Product(102, "Shoes", "Fashion"),                  new Product(103, "Phone", "Electronics"),                  new Product(104, "Watch", "Accessories"),                  new Product(105, "Bag", "Fashion")          };          System.out.println("Search 'Phone':");          Product result1 = linearSearchMethod(products, "Phone");          System.out.println(result1 != null ? result1 : "Product not found");          Arrays.sort(products, Comparator.comparing(p -> p.productName.toLowerCase()));          System.out.println("\n Search for 'Phone':");          Product result2 = binarySearchMethod(products, "Phone");          System.out.println(result2 != null ? result2 : "Product not found");      }  } |

OUTPUT - :



Big O notation helps to analyse how the runtime or memory usage of an algorithm increases with increasing input size. It generally describes the worst-case scenario, providing an upper limit on the algorithm's performance.

1. Best case-: The target element is found with the fewest comparisons possible.

Eg-: Linear search : O(1) When the target element is found at the first position.

Binary search : O(1) When the target/key element is found at the first index during the search operation.

1. Worst case-: The target element is found with the maximum comparisons possible.

Eg-: Linear search : O(N) When the target element is found at the last position performing

Binary search : O(log N) element found after repeatedly halving the search space or not present.

1. Average case-: This is the average number of operations an algorithm performs over all possible inputs. Eg-: Linear search : O(N) When the target element is found somewhere at the middle position. Binary search : O(log N)  element found after repeatedly halving the search space.

ANALYSIS-:

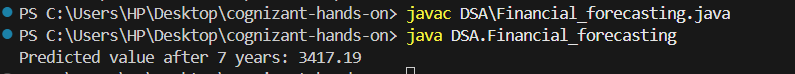
For small product arrays, linear search is fine.

For large product categories, binary search can be used with sorted data or better: use HashMaps or search for production-level systems.

**Exercise 7: Financial Forecasting**

|  |
| --- |
| package DSA;  class Financial\_forecasting {      public static double predictFutureVal(double currentValue, double growthRate, int years) {          if (years == 0) {              return currentValue;          }          return predictFutureVal(currentValue \* (1 + growthRate), growthRate, years - 1);      }      public static void main(String[] args) {          double currentValue = 200;          double growthRate = 0.5;          int yrs = 7;          double fv = predictFutureVal(currentValue, growthRate, yrs);          System.out.printf("Predicted value after %d years: %.2f\n", yrs, fv);      }  } |

**OUTPUT-:**

****

**Recursion-:** It is a programming concept in which a function is defined and this function is called again whenever needed without declaring it multiple times. Here a bigger problem can be solved by breaking it down to multiple smaller problems .

**Time Complexity-:**

This recursion runs once per year: T(n) = T(n - 1) + O(1)**.** Therefore , **Time Complexity = O(n)**

In order to optimise the excessive consumption we can use Tabulation (bottom-up approach) used in dynamic programming . It makes use of a loop to compute and store results from the smallest subproblem up to the desired one, when we need to avoid recursion.