Digital Nurture **4.0 – Deep Skilling (Java FSE** A close-up of a logo

AI-generated content may be incorrect.)

Name : Aswath Kannan M  
Superset ID : 6420716 (Week 1)

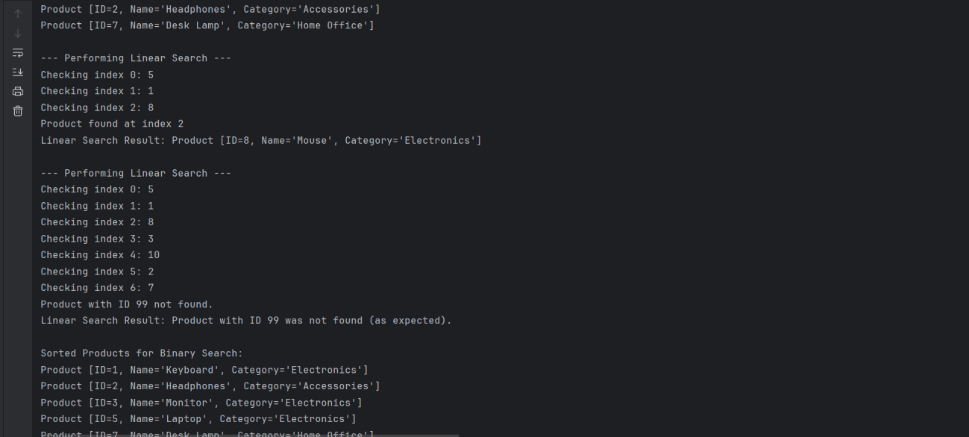
Algorithms\_Data structures  
**Qn 1:** E-commerce Platform Search Function  
Main.Java

import java.util.Arrays;  
import java.util.Comparator;  
  
public class Main {  
  
  
 static 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 int getProductId() {  
 return productId;  
 }  
  
 @Override  
 public String toString() {  
 return "Product [ID=" + productId + ", Name='" + productName + "', Category='" + category + "']";  
 }  
 }  
  
 public static Product linearSearch(Product[] products, int targetId) {  
 System.*out*.println("\n--- Performing Linear Search ---");  
 for (int i = 0; i < products.length; i++) {  
 System.*out*.println("Checking index " + i + ": " + products[i].getProductId());  
 if (products[i].getProductId() == targetId) {  
 System.*out*.println("Product found at index " + i);  
 return products[i];  
 }  
 }  
 System.*out*.println("Product with ID " + targetId + " not found.");  
 return null;  
 }  
  
 public static Product binarySearch(Product[] products, int targetId) {  
 System.*out*.println("\n--- Performing Binary Search ---");  
 int low = 0;  
 int high = products.length - 1;  
  
 while (low <= high) {  
 int mid = low + (high - low) / 2;  
 int midProductId = products[mid].getProductId();  
  
 System.*out*.println("Checking range [" + low + ", " + high + "], Mid: " + mid + " (ID: " + midProductId + ")");  
  
 if (midProductId == targetId) {  
 System.*out*.println("Product found at index " + mid);  
 return products[mid];  
 } else if (midProductId < targetId) {  
 low = mid + 1;  
 } else {  
 high = mid - 1;  
 }  
 }  
 System.*out*.println("Product with ID " + targetId + " not found.");  
 return null;  
 }  
  
 public static void main(String[] args) {  
 System.*out*.println("E-commerce Platform Search Function Analysis\n");  
  
 Product[] unsortedProducts = {  
 new Product(5, "Laptop", "Electronics"),  
 new Product(1, "Keyboard", "Electronics"),  
 new Product(8, "Mouse", "Electronics"),  
 new Product(3, "Monitor", "Electronics"),  
 new Product(10, "Webcam", "Accessories"),  
 new Product(2, "Headphones", "Accessories"),  
 new Product(7, "Desk Lamp", "Home Office")  
 };  
  
 System.*out*.println("Unsorted Products for Linear Search:");  
 for (Product p : unsortedProducts) {  
 System.*out*.println(p);  
 }  
  
 // Demonstrate Linear Search  
 int targetIdLinear = 8;  
 Product foundProductLinear = *linearSearch*(unsortedProducts, targetIdLinear);  
 if (foundProductLinear != null) {  
 System.*out*.println("Linear Search Result: " + foundProductLinear);  
 }  
 targetIdLinear = 99; // Non-existent ID  
 foundProductLinear = *linearSearch*(unsortedProducts, targetIdLinear);  
 if (foundProductLinear == null) {  
 System.*out*.println("Linear Search Result: Product with ID " + targetIdLinear + " was not found (as expected).");  
 }  
  
 Product[] sortedProducts = Arrays.*copyOf*(unsortedProducts, unsortedProducts.length);  
 Arrays.*sort*(sortedProducts, Comparator.*comparingInt*(Product::getProductId));  
  
 System.*out*.println("\nSorted Products for Binary Search:");  
 for (Product p : sortedProducts) {  
 System.*out*.println(p);  
 }  
  
 // Demonstrate Binary Search  
 int targetIdBinary = 7;  
 Product foundProductBinary = *binarySearch*(sortedProducts, targetIdBinary);  
 if (foundProductBinary != null) {  
 System.*out*.println("Binary Search Result: " + foundProductBinary);  
 }  
 targetIdBinary = 1;  
 foundProductBinary = *binarySearch*(sortedProducts, targetIdBinary);  
 if (foundProductBinary != null) {  
 System.*out*.println("Binary Search Result: " + foundProductBinary);  
 }  
 targetIdBinary = 99; // Non-existent ID  
 foundProductBinary = *binarySearch*(sortedProducts, targetIdBinary);  
 if (foundProductBinary == null) {  
 System.*out*.println("Binary Search Result: Product with ID " + targetIdBinary + " was not found (as expected).");  
 }  
  
 // Analysis and Discussion  
 System.*out*.println("\n--- Analysis of Linear vs. Binary Search ---");  
 System.*out*.println("\nTime Complexity Comparison:");  
 System.*out*.println("Linear Search: O(n)");  
 System.*out*.println("Binary Search: O(log n)");  
  
 System.*out*.println("\nDiscussion: Which algorithm is more suitable for an e-commerce platform search?");  
 System.*out*.println("----------------------------------------------------------------------------------");  
 System.*out*.println("For an e-commerce platform with a large number of products, Binary Search is generally");  
 System.*out*.println("far more suitable than Linear Search, provided the data can be kept sorted.");  
 System.*out*.println("\nReasons:");  
 System.*out*.println("1. Performance for Large Datasets:");  
 System.*out*.println(" - Binary Search (O(log n)) is significantly faster for large 'n'. If you have 1 million");  
 System.*out*.println(" products, Linear Search might require up to 1 million comparisons, while Binary Search");  
 System.*out*.println(" would only need about log2(1,000,000) ≈ 20 comparisons. This difference is critical for");  
 System.*out*.println(" fast user experience.");  
 System.*out*.println("2. Scalability:");  
 System.*out*.println(" - As the number of products grows, Binary Search's performance degrades much more slowly,");  
 System.*out*.println(" making it highly scalable.");  
 System.*out*.println("\nConsiderations for Binary Search:");  
 System.*out*.println("1. Data Must Be Sorted:");  
 System.*out*.println(" - Requires data to be sorted by the search key. Maintaining this order has an overhead,");  
 System.*out*.println(" but it's often managed by underlying data structures (like database indexes or balanced trees).");  
 System.*out*.println("\nConclusion:");  
 System.*out*.println("For optimal search performance on an e-commerce platform, \*\*Binary Search (or index-based lookups)\*\*");  
 System.*out*.println("is the preferred algorithm due to its superior time complexity and scalability for large datasets.");  
 }  
}

**Output**

A screenshot of a computer program

AI-generated content may be incorrect.



A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

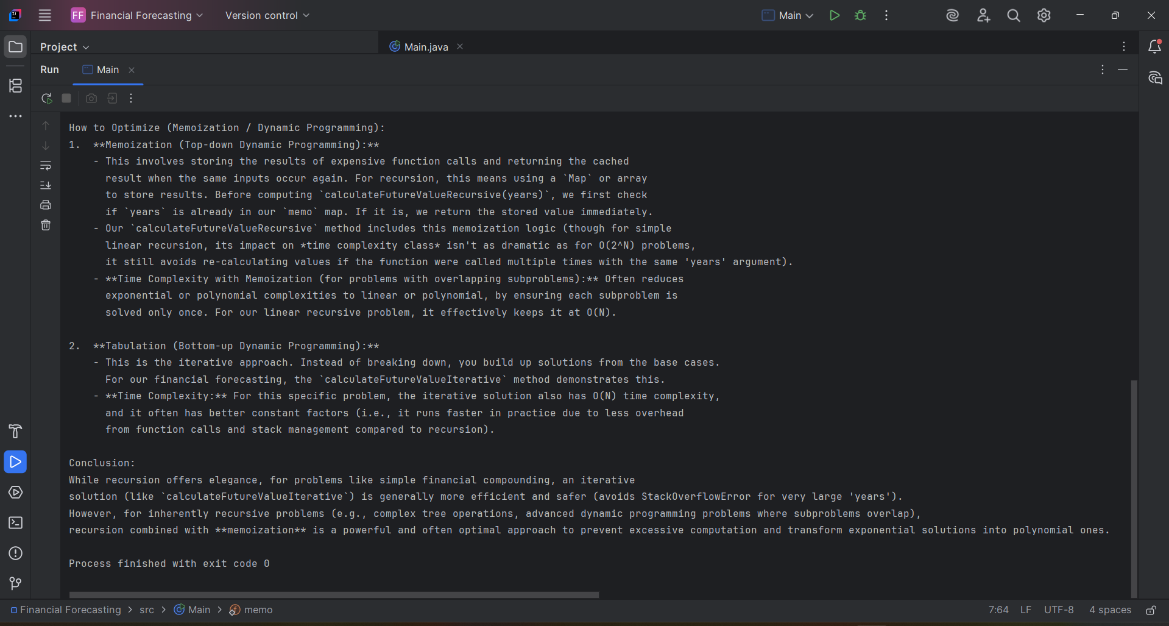
AI-generated content may be incorrect.

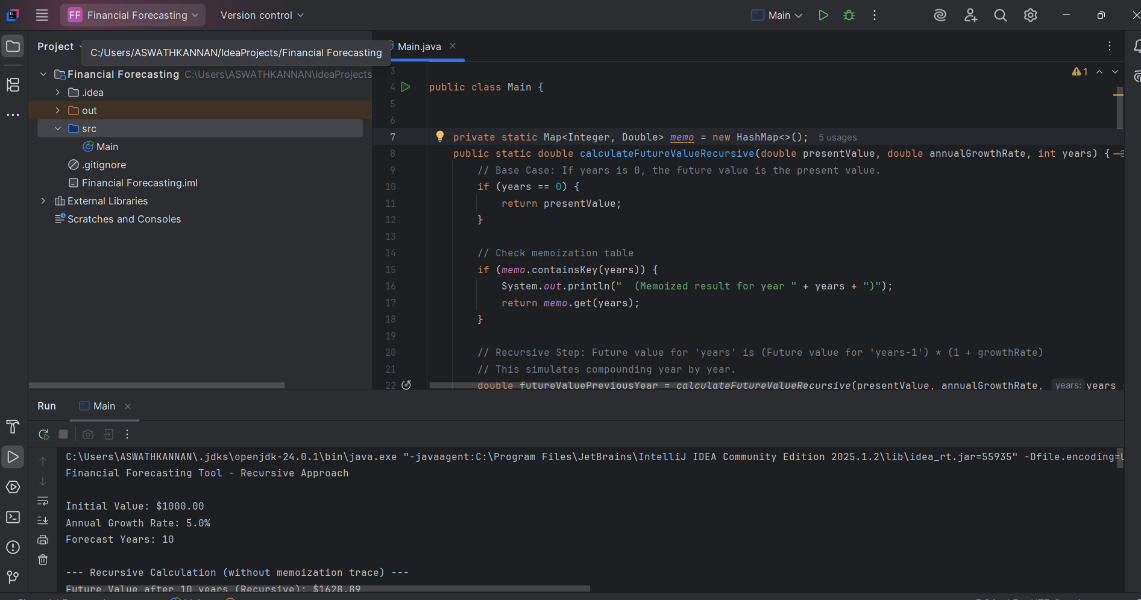
**Qn 2:** Financial Forecasting  
Main.Java

import java.util.HashMap;  
import java.util.Map;  
  
public class Main {  
  
 private static Map<Integer, Double> *memo* = new HashMap<>();public static double calculateFutureValueRecursive(double presentValue, double annualGrowthRate, int years) {  
 // Base Case: If years is 0, the future value is the present value.  
 if (years == 0) {  
 return presentValue;  
 }  
 if (*memo*.containsKey(years)) {  
 System.*out*.println(" (Memoized result for year " + years + ")");  
 return *memo*.get(years);  
 }  
  
 double futureValuePreviousYear = *calculateFutureValueRecursive*(presentValue, annualGrowthRate, years - 1);  
 double currentYearFutureValue = futureValuePreviousYear \* (1 + annualGrowthRate);  
  
 *memo*.put(years, currentYearFutureValue);  
 return currentYearFutureValue;  
 }  
  
 public static double calculateFutureValueIterative(double presentValue, double annualGrowthRate, int years) {  
 double futureValue = presentValue;  
 for (int i = 0; i < years; i++) {  
 futureValue \*= (1 + annualGrowthRate);  
 }  
 return futureValue;  
 }

public static void main(String[] args) {  
 System.*out*.println("Financial Forecasting Tool - Recursive Approach\n");  
  
 double initialValue = 1000.0; // $1000  
 double growthRate = 0.05; // 5% annual growth  
 int forecastYears = 10; // Forecast for 10 years  
  
 System.*out*.println("Initial Value: $" + String.*format*("%.2f", initialValue));  
 System.*out*.println("Annual Growth Rate: " + (growthRate \* 100) + "%");  
 System.*out*.println("Forecast Years: " + forecastYears);  
 System.*out*.println("\n--- Recursive Calculation (without memoization trace) ---");  
 *memo*.clear();  
 double futureValueRecursiveUnoptimized = *calculateFutureValueRecursive*(initialValue, growthRate, forecastYears);  
 System.*out*.println("Future Value after " + forecastYears + " years (Recursive): $" + String.*format*("%.2f", futureValueRecursiveUnoptimized));  
  
  
   
 System.*out*.println("\n--- Recursive Calculation (with memoization trace) ---");  
 *memo*.clear(); // Clear memoization table for a new calculation  
 double futureValueRecursiveOptimized = *calculateFutureValueRecursive*(initialValue, growthRate, forecastYears);  
 System.*out*.println("Future Value after " + forecastYears + " years (Recursive with Memoization): $" + String.*format*("%.2f", futureValueRecursiveOptimized));  
  
  
 System.*out*.println("\n--- Iterative Calculation (for comparison) ---");  
 double futureValueIterative = *calculateFutureValueIterative*(initialValue, growthRate, forecastYears);  
 System.*out*.println("Future Value after " + forecastYears + " years (Iterative): $" + String.*format*("%.2f", futureValueIterative));  
 System.*out*.println("\n--- Analysis of Recursive Algorithm ---");  
  
 System.*out*.println("\nTime Complexity of the Pure Recursive Algorithm (without Memoization):");  
 System.*out*.println("The pure recursive algorithm (FutureValue(year) FutureValue(year-1) \* (1 + rate))");  
 System.*out*.println("has a time complexity of O(N), where N is the number of 'years'.");  
 System.*out*.println("This is because each call to calculateFutureValueRecursive(years) makes one recursive");  
 System.*out*.println("call to calculateFutureValueRecursive(years - 1), leading to a linear chain of calls.");  
 System.*out*.println("It's similar to how factorial calculation iterates down to the base case.");  
  
 System.*out*.println("\nOptimization: Avoiding Excessive Computation (Memoization/Dynamic Programming)");  
 System.*out*.println("----------------------------------------------------------------------------------");  
 System.*out*.println("The current recursive algorithm is relatively simple and doesn't suffer from");  
 System.*out*.println("redundant computations if it just calculates a single value directly down to 0.");  
 System.*out*.println("However, if the recursive definition were more complex (e.g., depending on F(n-1) AND F(n-2) like Fibonacci),");  
 System.*out*.println("a naive recursive solution would suffer from \*\*overlapping subproblems\*\* and \*\*redundant calculations\*\*.");  
  
 System.*out*.println("\nExample of Redundancy (if this were Fibonacci-like):");  
 System.*out*.println("To calculate F(5), you need F(4) and F(3). To calculate F(4), you need F(3) and F(2).");  
 System.*out*.println("Notice F(3) is calculated multiple times. This leads to an exponential time complexity (e.g., O(2^N) for Fibonacci).");  
  
 System.*out*.println("\nHow to Optimize (Memoization / Dynamic Programming):");  
 System.*out*.println("1. \*\*Memoization (Top-down Dynamic Programming):\*\*");  
 System.*out*.println(" - This involves storing the results of expensive function calls and returning the cached");  
 System.*out*.println(" result when the same inputs occur again. For recursion, this means using a `Map` or array");  
 System.*out*.println(" to store results. Before computing `calculateFutureValueRecursive(years)`, we first check");  
 System.*out*.println(" if `years` is already in our `memo` map. If it is, we return the stored value immediately.");  
 System.*out*.println(" - Our `calculateFutureValueRecursive` method includes this memoization logic (though for simple");  
 System.*out*.println(" linear recursion, its impact on \*time complexity class\* isn't as dramatic as for O(2^N) problems,");  
 System.*out*.println(" it still avoids re-calculating values if the function were called multiple times with the same 'years' argument).");  
 System.*out*.println(" - \*\*Time Complexity with Memoization (for problems with overlapping subproblems):\*\* Often reduces");  
 System.*out*.println(" exponential or polynomial complexities to linear or polynomial, by ensuring each subproblem is");  
 System.*out*.println(" solved only once. For our linear recursive problem, it effectively keeps it at O(N).");  
  
 System.*out*.println("\n2. \*\*Tabulation (Bottom-up Dynamic Programming):\*\*");  
 System.*out*.println(" - This is the iterative approach. Instead of breaking down, you build up solutions from the base cases.");  
 System.*out*.println(" For our financial forecasting, the `calculateFutureValueIterative` method demonstrates this.");  
 System.*out*.println(" - \*\*Time Complexity:\*\* For this specific problem, the iterative solution also has O(N) time complexity,");  
 System.*out*.println(" and it often has better constant factors (i.e., it runs faster in practice due to less overhead");  
 System.*out*.println(" from function calls and stack management compared to recursion).");  
  
 System.*out*.println("\nConclusion:");  
 System.*out*.println("While recursion offers elegance, for problems like simple financial compounding, an iterative");  
 System.*out*.println("solution (like `calculateFutureValueIterative`) is generally more efficient and safer (avoids StackOverflowError for very large 'years').");  
 System.*out*.println("However, for inherently recursive problems (e.g., complex tree operations, advanced dynamic programming problems where subproblems overlap),");  
 System.*out*.println("recursion combined with \*\*memoization\*\* is a powerful and often optimal approach to prevent excessive computation and transform exponential solutions into polynomial ones."); } }

**Output**

**A screenshot of a computer program

AI-generated content may be incorrect.**