**DATA STRUCTURES AND ALGORITHMS HANDS-ON**

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

1. Understanding Asymptotic Notation

Big O Notation is a way to measure and describe how the runtime of an algorithm grows as the size of the input increases. It focuses on the *worst-case* scenario, giving an upper limit on time or space used, which helps in comparing algorithm efficiency.

For example:

* O(1) means the operation takes constant time regardless of input size.
* O(n) means time grows linearly with input size.
* O(log n) is much faster and typical for algorithms like binary search.

Best, Average, and Worst-Case Scenarios:

* In a linear search, the best case is when the target is at the start (O(1)), the worst case is when it’s at the end or not present at all (O(n)), and the average is somewhere in between.
* In a binary search, the best case also takes O(1) if the target is at the center, but on average or in the worst case, it takes O(log n), as it divides the search space in half each time.

2.Code:

import java.util.Arrays;

import java.util.Comparator;

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 + ")";

}

}

public class SearchDemo {

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

for (int i = 0; i < products.length; i++) {

if (products[i].productId == targetId) {

return i;

}

}

return -1;

}

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

int left = 0, right = products.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

if (products[mid].productId == targetId) return mid;

else if (products[mid].productId < targetId) left = mid + 1;

else right = mid - 1;

}

return -1;

public static void main(String[] args) {

Product[] products = {

new Product(103, "Shoes", "Footwear"),

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

new Product(105, "Watch", "Accessories"),

new Product(102, "T-Shirt", "Apparel"),

new Product(104, "Laptop", "Electronics")

};

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

int index1 = linearSearch(products, 104);

System.out.println(index1 != -1 ? "Found: " + products[index1] : "Not Found");

Arrays.sort(products, Comparator.comparingInt(p -> p.productId));

System.out.println("\nBinary Search (sorted by productId):");

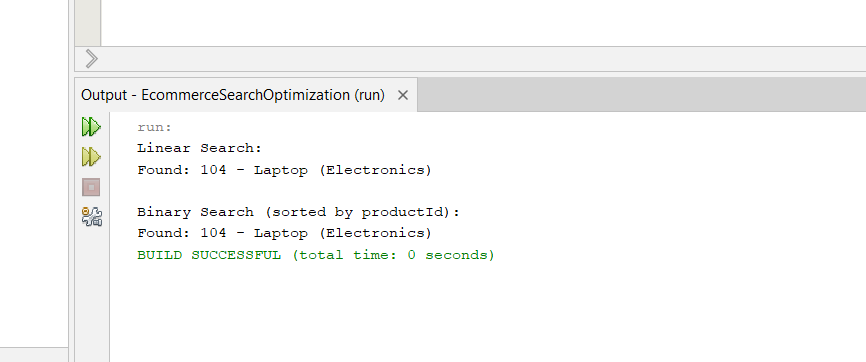
int index2 = binarySearch(products, 104);

System.out.println(index2 != -1 ? "Found: " + products[index2] : "Not Found");

}

}

**Output:**

****

3. Implementation Approach

Linear Search: It goes through each product in the array one by one until it finds a match. It works even if the data isn’t sorted.

Binary Search: It only works on a sorted array. It checks the middle product, and depending on whether the target is greater or lesser, it cuts the search space in half. This drastically reduces the number of comparisons needed.

Before using binary search, the product list must be sorted alphabetically (or in some consistent order) by the product name.

4. Time Complexity Analysis

Linear Search has a time complexity of O(n). It’s simple but slow for large datasets since it may have to check every item.

Binary Search has a time complexity of O(log n). It’s much faster for large datasets, but only works when the data is sorted.

**Exercise 7: Financial Forecasting**

1. Recursion Concept

Recursion is a technique where a method calls itself to solve smaller instances of the same problem.  
It’s useful for problems like factorials, tree traversal, or repetitive calculations like financial projections over years.

2.Code:

public class FinancialForecast {

public static double predictFutureValue(double currentValue, double annualGrowthRate, int years) {

if (years == 0) {

return currentValue; }

return predictFutureValue(currentValue \* (1 + annualGrowthRate), annualGrowthRate, years - 1);

}

public static double predictFutureValueMemo(double currentValue, double annualGrowthRate, int years, double[] memo) {

if (memo[years] != 0) return memo[years];

if (years == 0) return currentValue;

memo[years] = predictFutureValueMemo(currentValue \* (1 + annualGrowthRate), annualGrowthRate, years - 1, memo);

return memo[years];

}

public static void main(String[] args) {

double initialValue = 10000;

double growthRate = 0.08;

int years = 5;

double future = predictFutureValue(initialValue, growthRate, years);

System.out.println("Recursive Prediction: " + future);

double[] memo = new double[years + 1];

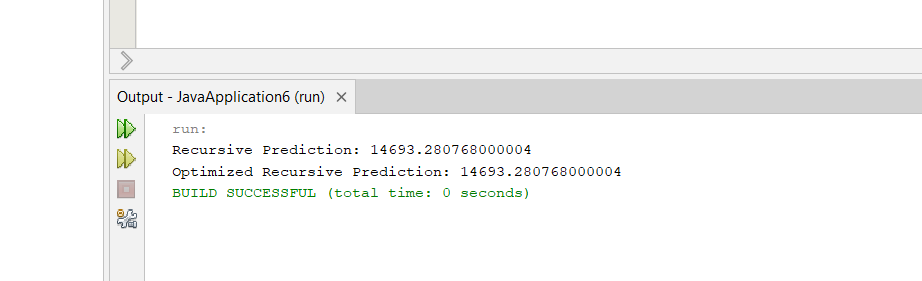
double optimizedFuture = predictFutureValueMemo(initialValue, growthRate, years, memo);

System.out.println("Optimized Recursive Prediction: " + optimizedFuture);

}

}

**Output:**



3.Analysis:

* The time complexity of the predictFutureValue recursive function is O(n), where n is the number of years being forecasted. This is because the function makes one recursive call for each year, reducing the problem by 1 each time until it reaches 0. Each call performs a constant amount of work, so the overall complexity is linear in the number of years.
* To avoid excessive computation in recursive forecasting, especially when the same subproblems are recalculated multiple times, we can use **memoization**. Memoization stores the result of each subproblem the first time it is computed, so that future calls can reuse it instead of recomputing. This significantly improves performance in recursive algorithms, particularly when extended to more complex scenarios like multiple rates or branches in forecasting.