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

Big O notation describes how the runtime of an algorithm grows with the size of input n. It helps in comparing algorithms regardless of hardware.

It tells us,

* How fast your search will run
* How it scales when product listings grow

**Search Scenarios:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Search Type** | **Best Case** | **Average Case** | **Worst Case** |
| **Linear** | O(1) | O(n) | O(n) |
| **Binary** | O(1) | O(log n) | O(log n) |

**Program:**

import java.util.Arrays;

import java.util.Comparator;

class Product {

int productId;

String productName;

String category;

public Product(int id, String name, String cat) {

this.productId = id;

this.productName = name;

this.category = cat;

}

@Override

public String toString() {

return "ID: " + productId + ", Name: " + productName + ", Category: " + category;

}

}

public class ECommerceSearch {

// Linear Search - No need to sort

public static Product linearSearch(Product[] products, String targetName) {

for (Product product : products) {

if (product.productName.equalsIgnoreCase(targetName)) {

return product;

}

}

return null;

}

// Binary Search

public static Product binarySearch(Product[] products, String targetName) {

int left = 0;

int right = products.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

int cmp = products[mid].productName.compareToIgnoreCase(targetName);

if (cmp == 0) return products[mid];

else if (cmp < 0) left = mid + 1;

else right = mid - 1;

}

return null;

}

public static void sortByProductName(Product[] products) {

Arrays.sort(products, Comparator.comparing(p -> p.productName.toLowerCase()));

}

public static void main(String[] args) {

Product[] products = {

new Product(1, "Laptop", "Electronics"),

new Product(2, "Mouse", "Accessories"),

new Product(3, "Keyboard", "Accessories"),

new Product(4, "Charger", "Electronics")

};

//Linear Search

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

Product result1 = linearSearch(products, "Mouse");

if (result1 != null)

System.out.println("Found: " + result1);

else

System.out.println("Product not found");

sortByProductName(products);

// Binary Search

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

Product result2 = binarySearch(products, "Mouse");

if (result2 != null)

System.out.println("Found: " + result2);

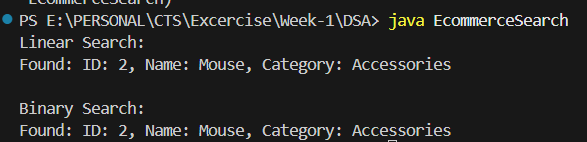
else

System.out.println("Product not found");

}

}

**Output:**

****

**Analysis:**

|  |  |  |
| --- | --- | --- |
| **Criteria** | **Linear Search** | **Binary Search** |
| Time Complexity | O(n) | O(log n) |
| Space Complexity | O(1) | O(1) |
| Sorted Required? | No | Yes |
| Use Case | Small or unsorted list | Large, sorted product list |

**Conclusion:**

1. linear search for small or frequently changing product lists.
2. binary search for large, sorted datasets, where performance matters.

**Exercise 7: Financial Forecasting**

**Recursion:**

Recursion is when a function calls itself with a smaller input until a base case is reached.

It simplifies problems that:

* Can be broken into similar sub-problems
* Follow a repeated pattern (e.g., growth year by year)

For financial forecasting, recursion is ideal when predicting value over n time periods by repeatedly applying a growth rate.

**Approach:**

If you have 0 years left: return the current value.

Otherwise:

Multiply the current value by (1 + current year's growth rate),

then do the same for the next year.

**Program:**

import java.util.HashMap;

import java.util.Map;

public class FinancialForecast {

// 1. Simple Recursive Method

public static double calculateFutureValueRecursive(double initialValue, double[] growthRates, int year) {

if (year == growthRates.length) {

return initialValue;

}

double newValue = initialValue \* (1 + growthRates[year]);

return calculateFutureValueRecursive(newValue, growthRates, year + 1);

}

// 2. Recursive with Memoization

static Map<Integer, Double> memo = new HashMap<>();

public static double calculateFutureValueMemo(double initialValue, double[] growthRates, int year) {

if (year == growthRates.length) {

return initialValue;

}

if (memo.containsKey(year)) {

return memo.get(year);

}

double newValue = initialValue \* (1 + growthRates[year]);

double result = calculateFutureValueMemo(newValue, growthRates, year + 1);

memo.put(year, result);

return result;

}

// 3. Iterative Version

public static double calculateFutureValueIterative(double initialValue, double[] growthRates) {

double futureValue = initialValue;

for (double rate : growthRates) {

futureValue \*= (1 + rate);

}

return futureValue;

}

// Main Method to Compare All

public static void main(String[] args) {

double initialValue = 1000.0;

double[] growthRates = {0.05, 0.07, 0.03}; // 3 years growth

// Clear memoization map before use

memo.clear();

// Method 1: Simple Recursion

double valueRecursive = calculateFutureValueRecursive(initialValue, growthRates, 0);

System.out.println("Future Value (Recursive): " + valueRecursive);

// Method 2: Recursive with Memoization

double valueMemo = calculateFutureValueMemo(initialValue, growthRates, 0);

System.out.println("Future Value (Memoization): " + valueMemo);

// Method 3: Iterative

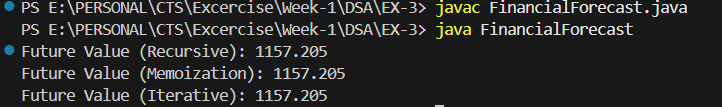
double valueIterative = calculateFutureValueIterative(initialValue, growthRates);

System.out.println("Future Value (Iterative): " + valueIterative);

}

}

**Output:**

****

**Analysis:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Method** | **Time Complexity** | **Space Complexity** | **Pros** | **Cons** |
| **1. Simple Recursion** | O(n) | O(n) (call stack) | Easy to write and understand | Risk of StackOverflowError for large n |
| **2. Memoization** | O(n) | O(n) (stack + map) | Caches results for reuse | Overkill here (no overlapping subproblems) |
| **3. Iterative** | O(n) | O(1) | Fastest, no stack overhead | Slightly more code than recursion |

**Optimization:**

Use memoization or convert recursion to an iterative loop to reduce stack overhead and improve performance in larger datasets.