**MANDATORY HANDS-ON SOLUTIONS**

**DATA STRUCTURES & ALGORITHMS**

**Week 1**

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

**Step 1: Understanding Asymptotic Notation**

**Big O Notation**Big O notation is a mathematical representation that describes the upper bound of an algorithm's running time as the input size grows.

It is crucial for analyzing and comparing the efficiency of different algorithms in terms of time and space complexity.

* **Best Case:** The minimum time required for an algorithm to complete.
* **Average Case:** The expected time taken across all possible inputs.
* **Worst Case:** The maximum time an algorithm might take.

|  |  |  |  |
| --- | --- | --- | --- |
| Case | Linear Search | Binary Search | Explanation |
| Best Case | O(1) | O(1) | Item found at the first (linear) or middle (binary). |
| Average Case | O(n) | O(log n) | Item found halfway through (linear) or log steps (binary). |
| Worst Case | O(n) | O(log n) | Item found at last (linear) or not found at all (binary). |

**Step 2: Setup**

Creating a class Product that contains attributes essential for search functionality:

**File: Product.java**

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;

}

}

**Step 3: Implementation**

**SearchDemo.java**

The main() function contains linear search, and binary search implementations.

import java.util.Arrays;

import java.util.Comparator;

public class SearchDemo {

public static void main(String[] args) {

// Creating an array of Product objects

Product[] products = {

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

new Product(102, "Shampoo", "Personal Care"),

new Product(103, "Notebook", "Stationery"),

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

};

// Linear Search Test

int index1 = linearSearch(products, "Shampoo");

if (index1 != -1)

System.out.println("Found using Linear Search at index: " + index1);

else

System.out.println("Not found using Linear Search");

// Binary Search Test

int index2 = binarySearch(products, "Notebook");

if (index2 != -1)

System.out.println("Found using Binary Search at index: " + index2);

else

System.out.println("Not found using Binary Search");

}

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

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

if (products[i].productName.equalsIgnoreCase(name)) {

return i;

}

}

return -1;

}

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

// Sort array by product name

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

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

while (left <= right) {

int mid = left + (right - left) / 2;

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

if (cmp == 0) return mid;

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

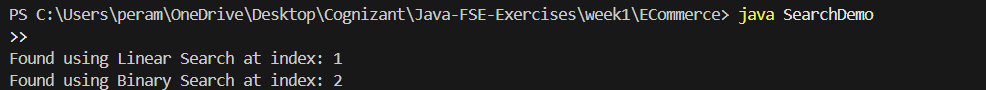
else right = mid - 1;

}

return -1;

}

}



**Step 4. Analysis**

**Time Complexity Comparison**

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

* **Linear Search** checks each element one by one until a match is found. This makes it slower for large datasets because, in the worst case, it may need to check every item.
* **Binary Search** is much faster because it divides the sorted array in half at each step, reducing the number of comparisons.

**Which Algorithm is More Suitable for an E-Commerce Platform?**

For an e-commerce platform where thousands of products may exist, **binary search is more efficient** due to its **logarithmic time complexity**. Since platforms often store sorted data, binary search can be used effectively to speed up the search process.

**Use Binary Search When:**

* The product list is sorted.
* Fast performance is required.
* The dataset is large.

**Use Linear Search When:**

* The dataset is small and unsorted.
* Simplicity is more important than speed.
* Sorting the data is not practical or necessary.

**Exercise 7: Financial Forecasting**

**Step 1: Understanding Recursive Algorithms**

Recursion is a method where a function calls itself to solve smaller subproblems.  
It is especially useful for problems like tree traversal, factorials, Fibonacci sequences, and prediction models like financial forecasting, where results build on past data.

**Step 2: Setup**

Creating a method to calculate the future value using a recursive approach.

In finance, a common compound growth model is:

FutureValue = PresentValue × (1 + growthRate)^years

We can recursively define:

FV(n) = FV(n-1) × (1 + r)

Base case: n = 0 → FV(0) = PresentValue

**Step 3: Implementation:**

FinancialForecast.java

public class FinancialForecast {

// Recursive method to calculate future value

public static double predictFutureValue(double presentValue, double growthRate, int years) {

if (years == 0) {

return presentValue; // base case

}

return predictFutureValue(presentValue, growthRate, years - 1) \* (1 + growthRate);

}

public static void main(String[] args) {

double presentValue = 10000; // Initial amount

double growthRate = 0.08; // 8% annual growth

int years = 5; // Forecast for 5 years

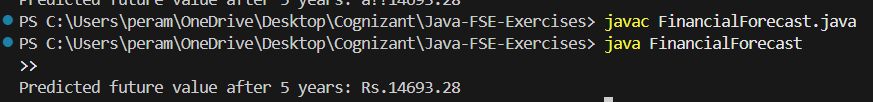
double futureValue = predictFutureValue(presentValue, growthRate, years);

System.out.printf("Predicted future value after %d years: Rs.%.2f\n", years, futureValue);

}

}

**OUTPUT:**

****

**Step 4: Analysis**

**Time Complexity Analysis:**

The recursive method predictFutureValue() calculates the compounded future value of an investment year by year:

return predictFutureValue(presentValue, growthRate, years - 1) \* (1 + growthRate);

Each recursive call reduces years by 1 until it reaches the base case (years == 0). Therefore, the time complexity is: O(n) — where n is the number of years.

Since the function calls itself once per year, it performs n recursive calls in total.

**Optimization:**

Even though the recursion in this is simple and non-redundant, it’s still safer and more efficient to convert it into an iterative version, especially for larger forecasts.

**Iterative Approach**

public static double predictFutureValueIterative(double presentValue, double growthRate, int years) {

for (int i = 0; i < years; i++) {

presentValue \*= (1 + growthRate);

}

return presentValue;

}

* Time Complexity: O(n)
* Space Complexity: O(1) — No stack buildup.

This avoids the recursion overhead and handles large values of years gracefully.