**Superset ID: 6394725**

**Data Structures and Algorithms**

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

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**Steps:**

1. **Understand Asymptotic Notation:**
   1. Explain Big O notation and how it helps in analyzing algorithms.
   2. Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
   1. Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   1. Implement linear search and binary search algorithms.
   2. Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   1. Compare the time complexity of linear and binary search algorithms.
   2. Discuss which algorithm is more suitable for your platform and why.

## **Step 1: Understanding Asymptotic Notation**

### **Big O Notation**

* Big O notation describes the **upper bound** of an algorithm’s runtime as the input size grows.
* It helps us understand **how efficiently** an algorithm performs.

|  |  |
| --- | --- |
| **Notation** | **Meaning** |
| O(1) | Constant time |
| O(log n) | Logarithmic time |
| O(n) | Linear time |
| O(n log n) | Log-linear time |
| O(n²) | Quadratic time |

### **Search Operation Cases**

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

* **Linear Search** doesn’t require sorting but is slower.
* **Binary Search** is faster but works **only on sorted arrays**.

**Step 2: Setup the Product Class**

**Product.java**

package com;

class Product {

int productId;

String productName;

String category;

Product(int productId, String productName, String category) {

this.productId = productId;

this.productName = productName;

this.category = category;

}

*@Override*

public String toString() {

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

}

}

**Step 3: Implementation**

Linear Search and Binary Search Implementation

**SearchAlgorithms.java**

package com;

public class SearchAlgorithms {

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

for (Product product : products) {

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

return product;

}

}

return null;

}

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

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

while (left <= right) {

int mid = (left + right) / 2;

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

if (compare == 0) {

return products[mid];

} else if (compare < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

}

Main Class to Test the Searches

**EcommerceSearch.java**

package com;

import java.util.Arrays;

import java.util.Comparator;

public class ECommerceSearch {

public static void main(String[] args) {

Product[] products = {

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

new Product(102, "Shoes", "Fashion"),

new Product(103, "Mobile", "Electronics"),

new Product(104, "Watch", "Accessories"),

new Product(105, "Bag", "Fashion")

};

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

Product result1 = SearchAlgorithms.linearSearch(products, "Mobile");

System.***out***.println(result1 != null ? result1 : "Product not found");

// Sort array for binary search

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

System.***out***.println("\n Binary Search:");

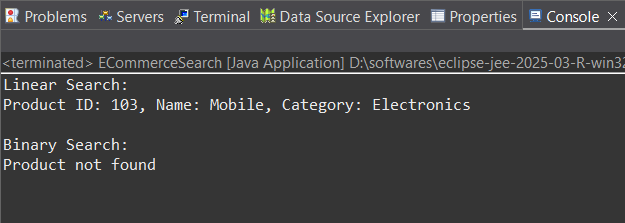
Product result2 = SearchAlgorithms.binarySearch(products, "Mobile");

System.***out***.println(result2 != null ? result2 : "Product not found");

}

}

**Output:**

**Step 4: Analysis**

### **Time Complexity Comparison**

|  |  |  |
| --- | --- | --- |
| **Search Type** | **Time Complexity** | **Suitable for** |
| Linear Search | O(n) | Small, unsorted datasets |
| Binary Search | O(log n) | Large, sorted datasets |

For **E-commerce platforms**, where:

* Thousands or millions of products exist,
* Fast and relevant search is critical,

**Binary search** is faster than linear search but **not practical** alone because:

* It works only on sorted arrays,
* Real e-commerce searches involve **multiple fields** (name, category, brand),
* Products are added/removed constantly.

### **Real-world Alternative:**

* Use **search engines** like **Elasticsearch** or **Solr** for full-text, multi-field, scalable search.
* Use **database indexing** (e.g., SQL LIKE, MongoDB text search) for efficient keyword filtering.

## **Conclusion**

* **Linear Search** is simple but inefficient for large datasets.
* **Binary Search** is faster but only usable on sorted data.
* For actual e-commerce platforms, it's best to use **indexed or full-text search systems** that can scale and support real-time updates.

**Exercise 7: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

**Steps:**

1. **Understand Recursive Algorithms:**
   1. Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
   1. Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   1. Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   1. Discuss the time complexity of your recursive algorithm.
   2. Explain how to optimize the recursive solution to avoid excessive computation.

## **Step-1 : Understanding Recursive Algorithms**

### **What is Recursion?**

Recursion is a technique in programming where a function calls itself to solve smaller instances of a problem until a base condition is met. It's often used for problems that can be divided into subproblems of the same kind.

### **Why Use Recursion?**

* **Simplicity**: Recursive solutions are often more readable and concise for problems like tree traversal, Fibonacci numbers, etc.
* **Natural Fit**: It models problems that inherently follow a repetitive or self-similar structure.

## **Step-2 : Setup: Recursive Financial Forecast**

We want to predict **future value** based on:

* **Initial Value (P)**: The present investment or value.
* **Growth Rate (r)**: The fixed percentage rate (e.g., 5% = 0.05).
* **Years (n)**: Number of years in the future.

### **Recursive Formula:**

FutureValue(P, r, n) = FutureValue(P, r, n-1) \* (1 + r)

Base case: FutureValue(P, r, 0) = P

## **Step-3 : Implementation**

Recursive Future Value Calculation in Java

**FinancialForecast.java**

package com;

public class FinancialForecast {

public static double futureValue(double principal, double rate, int years) {

// Base case

if (years == 0) {

return principal;

}

// Recursive case

return *futureValue*(principal, rate, years - 1) \* (1 + rate);

}

public static void main(String[] args) {

double initial = 1000.0; // Initial amount

double rate = 0.05; // 5% interest

int years = 5;

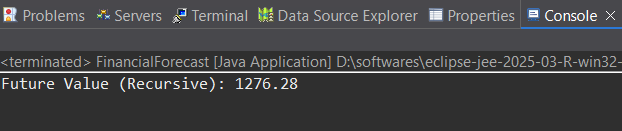
double result = *futureValue*(initial, rate, years);

System.***out***.printf("Future Value (Recursive): %.2f%n", result);

}

}

**Output**

**Step-4 : Analysis**

### **Time Complexity**

* **Time**: O(n) — Each year requires one recursive call.
* **Space**: O(n) — Due to the call stack in recursion.

### **Drawbacks of Pure Recursion**

* Inefficient for large n due to deep recursion.
* Risk of **stack overflow** if recursion depth exceeds limits.

## **Optimization**

### **1. Memorization**

**FinancialForecastMemo.java**

package com;

import java.util.HashMap;

public class FinancialForecastMemo {

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

public static double futureValue(double principal, double rate, int years) {

if (years == 0) {

return principal;

}

if (*memo*.containsKey(years)) {

return *memo*.get(years);

}

double value = *futureValue*(principal, rate, years - 1) \* (1 + rate);

*memo*.put(years, value);

return value;

}

public static void main(String[] args) {

double initial = 1000.0;

double rate = 0.05;

int years = 5;

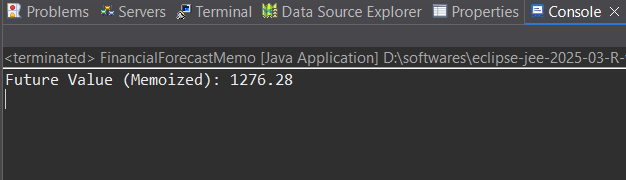
double result = *futureValue*(initial, rate, years);

System.***out***.printf("Future Value (Memoized): %.2f%n", result);

}

}

**Output :**



### **2. Iterative Solution (Best for Performance)**

**FinancialForecastIter.java**

package com;

public class FinancialForecastIter {

public static double futureValue(double principal, double rate, int years) {

double value = principal;

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

value \*= (1 + rate);

}

return value;

}

public static void main(String[] args) {

double initial = 1000.0;

double rate = 0.05;

int years = 5;

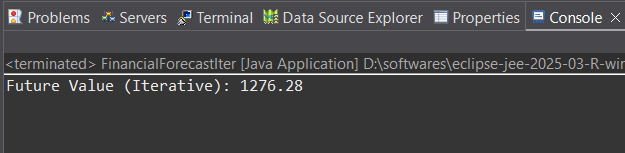
double result = *futureValue*(initial, rate, years);

System.***out***.printf("Future Value (Iterative): %.2f%n", result);

}

}

**Output :**

**Conclusion**

* **Recursion** is elegant and intuitive for problems like forecasting, but not always efficient.
* **Optimizations** like **memorization** or using **iteration** can drastically improve performance.
* For practical applications, **iterative solutions** are preferred for scalability and reliability.