# Design Patterns and Principles

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

=========================================================================

SOLUTION:

**Step 1: Understand Asymptotic Notation**

**Big O Notation**

Big O notation is a mathematical representation used to describe the time or space complexity of an algorithm in terms of input size n. It helps developers understand the scalability and performance of algorithms, especially as the input size grows.

* It ignores constant factors and lower-order terms.
* Example: O(n) means time grows linearly with input size.

**Best, Average, and Worst-Case Scenarios**

1. **Best Case**: The condition where the algorithm performs the minimum number of steps (e.g., first item match in a search).
   * Linear Search: O(1) (item at the beginning)
   * Binary Search: O(1) (middle element is the match)
2. **Average Case**: The expected complexity over all possible inputs.
   * Linear Search: O(n/2) ⇒ simplifies to O(n)
   * Binary Search: O(log n)
3. **Worst Case**: The condition with maximum operations needed (e.g., item not found or at the last position).
   * Linear Search: O(n)
   * Binary Search: O(log n)

**Step 2: SETUP**

// 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**

import java.util.Arrays;

import java.util.Comparator;

public class SearchDemo {

// Linear Search

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

for (Product p : products) {

if (p.productName.equalsIgnoreCase(productName)) {

return p;

}

}

return null;

}

// Binary Search (on sorted array)

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

int low = 0, high = products.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

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

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

else if (compare < 0) low = mid + 1;

else high = mid - 1;

}

return null;

}

public static void main(String[] args) {

Product[] products = {

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

new Product(2, "Shoes", "Fashion"),

new Product(3, "Phone", "Electronics"),

new Product(4, "Book", "Education")

};

// Sort for binary search

Product[] sortedProducts = Arrays.copyOf(products, products.length);

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

// Perform searches

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

Product result2 = binarySearch(sortedProducts, "Phone");

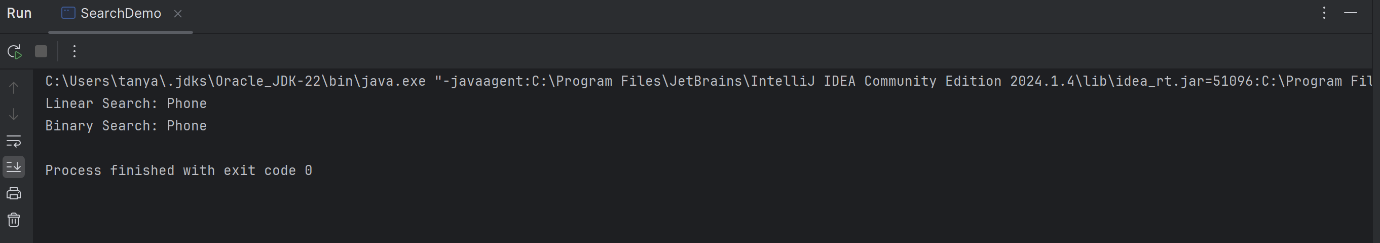
System.out.println("Linear Search: " + (result1 != null ? result1.productName : "Not Found"));

System.out.println("Binary Search: " + (result2 != null ? result2.productName : "Not Found"));

}

}

**OUTPUT**

****

**Step 4: Analysis**

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** | **Sorted Array Required?** |
| --- | --- | --- | --- | --- |
| Linear Search | O(1) | O(n) | O(n) | No |
| Binary Search | O(1) | O(log n) | O(log n) | Yes |

**Which is more suitable?**

* **Binary Search** is more efficient in time complexity (O(log n)) but **requires data to be sorted**, which may have overhead for dynamic datasets.
* **Linear Search** is simple and works on unsorted data but is inefficient for large data (O(n)).

**Conclusion:**

* Use **Binary Search** for large, static, or infrequently updated product lists (e.g., product catalog).
* Use **Linear Search** for small datasets or when real-time updates are frequent and sorting is costly.

===================

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

=================================================================================

SOLUTION:

**1. Understand Recursive Algorithms**

**Recursion** is a programming technique where a function calls itself to solve a problem. It breaks down a problem into smaller sub-problems of the same type until reaching a base case, which is directly solvable. Recursion is particularly useful in problems involving repetitive patterns or hierarchical data, such as tree traversals, Fibonacci sequences, or financial forecasts with compounding growth.

In forecasting, if the value grows by a certain percentage each year, recursion can simplify the logic to apply growth year after year.

**2. Setup: Recursive Method for Future Value**

We want to calculate the **future value (FV)** of an investment recursively, using:

* **Initial value (P)**: Principal or starting value
* **Growth rate (r)**: Growth rate per period (e.g., per year)
* **Time (n)**: Number of periods (e.g., years)

**Formula**:  
FV = P × (1 + r)^n

**Code:**

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

// Base case

if (years == 0) {

return principal;

}

// Recursive case

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

}

**4.IMPLEMENTATION**

public class FinancialForecast {

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

if (years == 0) {

return principal;

}

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

}

public static void main(String[] args) {

double initialAmount = 10000; // Example: ₹10,000

double annualGrowthRate = 0.08; // 8% annual growth

int period = 5; // 5 years

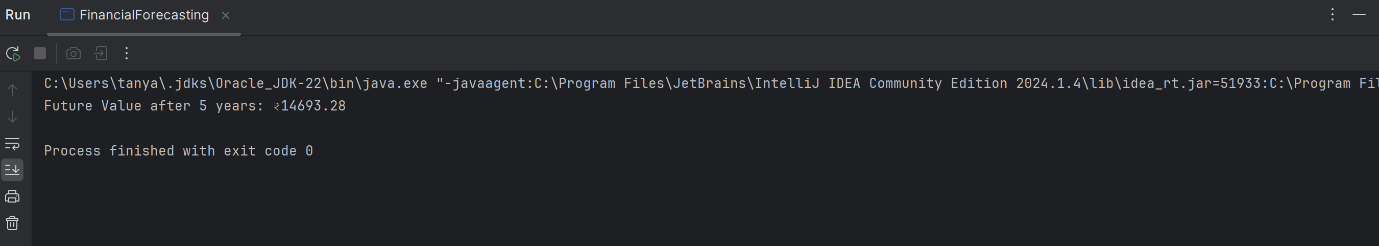
double result = futureValue(initialAmount, annualGrowthRate, period);

System.out.printf("Future Value after %d years: ₹%.2f%n", period, result);

}

}

**OUTPUT:**



**4. Analysis**

**Time Complexity**

The time complexity of this recursive method is **O(n)** because it performs one recursive call per year (until years == 0).

**Optimization Tips**

While this version is efficient enough for small n, consider the following to optimize:

* **Memoization**: Store already computed values to avoid recalculating if needed for multiple queries.
* **Iterative approach**: Use a loop instead of recursion to avoid stack overflow and improve performance.

**=======================**