**Week-1**

**Algorithms And Data Structures**

**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.

**What is Big O Notation?**

* Big O Notation describes how the runtime of an algorithm grows relative to input size.
* It gives an upper bound on time complexity and helps us analyse efficiency.

|  |  |  |
| --- | --- | --- |
| **Time Complexity** | **Notation** | **Description** |
| Constant Time | O(1) | The time taken is independent of the input size. |
| Logarithmic Time | O(logn) | Time increases logarithmically as input grows. Common in binary search. |
| Linear Time | O(n) | Time grows proportionally with input size. |
| Linearithmic Time | O(nlogn) | Grows faster than linear; used in efficient sorting (e.g., merge sort). |
| Quadratic Time | O(n2) | Time grows with the square of the input size (e.g., nested loops). |
| Exponential Time | O(2n) | Time doubles with each increase in input size. Very inefficient. |
| Factorial Time | O(n!) | Time grows extremely fast. Used in brute-force permutations, etc. |

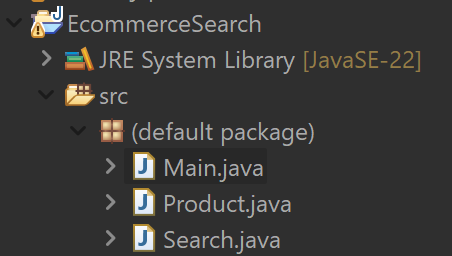
**Why it Matters in Searching:**

* Helps determine scalability of search algorithms.
* Important for platforms with thousands or millions of products.

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |
| Linear Search | O(1) item is at start | O(n) | O(n) item is at end or not present |
| Binary Search | O(1) item at middle | O(log n) | O(log n) divide search space in half each time |

**Implementation:**

**Folder Structure:**

****

**Product.java**

// Product.java

public class Product {

private int productId;

private String productName;

private String category;

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

public int getProductId() {

return productId;

}

public String getProductName() {

return productName;

}

public String getCategory() {

return category;

}

*@Override*

public String display() {

return "[" + productId + "] " + productName + " - " + category;

}

}

**Search.java**

// Search.java

import java.util.Arrays;

import java.util.Comparator;

public class Search {

// Linear Search by product name

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

for (Product product : products) {

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

return product;

}

}

return null;

}

// Binary Search by product name (array must be sorted)

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

Arrays.*sort*(products, Comparator.*comparing*(Product::getProductName));

int left = 0;

int right = products.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

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

if (cmp == 0) {

return products[mid];

} else if (cmp < 0) {

right = mid - 1;

} else {

left = mid + 1;

}

}

return null;

}

}

**Main.java**

//Main.java

public class Main {

public static void main(String[] args) {

Product[] products = {

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

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

new Product(103, "Watch", "Accessories"),

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

new Product(105, "T-Shirt", "Fashion")

};

String target = "Phone";

// Linear Search

Product foundLinear = Search.*linearSearch*(products, target);

System.*out*.println("Linear Search: " + (foundLinear != null ? foundLinear : "Product not found"));

// Binary Search

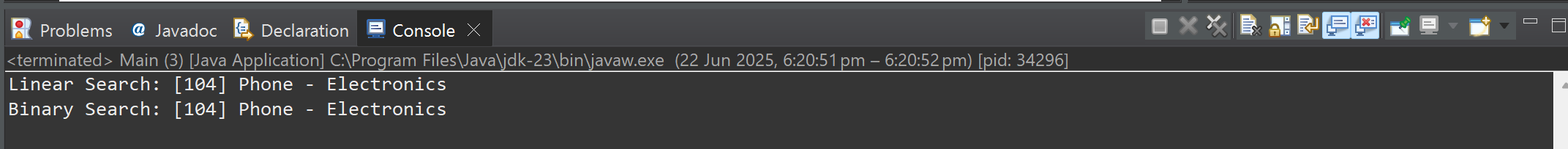
Product foundBinary = Search.*binarySearch*(products, target);

System.*out*.println("Binary Search: " + (foundBinary != null ? foundBinary : "Product not found"));

}

}

**Output:**

****

**Conclusion:**

When running a big and ever-growing online store with lots of products, using search methods that are fast and efficient is key. Binary Search, or similar algorithms that work really well with sorted or indexed data and can find what you're looking for quickly, is often the best choice. Even though sorting all the data takes some time upfront, it's a smart trade-off because it makes every search afterward much faster. This way, customers get quicker results, and the whole system runs smoother.

**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.

**Recursion**

Recursion refers to a programming technique in which functions solve problems by calling themselves. It is essentially the act of breaking a big problem into smaller problems that are identical in nature until it reaches a point where it is a "base case" that the function can solve simply.  
  
**Fundamental Components:**  
  
**1.Base Case:** The stopping point. It is the simplest one of the problem where the function can directly solve it, avoiding eternal calls.  
**2.Recursive Step:** This is the situation in which the function calls itself but this time working on a simpler or smaller version from the original problem, marching towards the base case.  
  
**Examples:**

int factorial(int n) {

if (n == 0) return 1; // Base case

return n \* factorial(n - 1); // Recursive call

}  
Recursion makes sense in examples that naturally "nest" or repeat a pattern. Some cases are:  
  
1. Traversing nested structures: Like searching through folders (a folder within a folder) or tree-like data.  
2. Divide and Conquer algorithms: Breaking a problem into smaller, similar pieces so as to solve them (e.g., sorting).  
3. Mathematical sequences: Calculating something like a Fibonacci series.

Recursion gives a neat alternative for some complicated problems, but it needs to be designed thoughtfully to avoid infinite loops and crashes in the program.

**Implementation:**

We want to predict the **future value** of an investment given:

* Initial amount
* Growth rate per year (e.g., 10%)
* Number of years

**Formula:**

Future Value = Present Value × (1 + rate)^years

We can solve this **recursively**:

FutureValue(amount, rate, years) = FutureValue(amount \* (1 + rate), rate, years - 1)

**Forecast.java**

//Forecast.java

public class Forecast {

// Recursive method to calculate future value

public static double FutureValue(double amount, double rate, int years) {

if (years == 0) {

return amount;

} else {

return (1 + rate) \* FutureValue(amount, rate, years - 1);

}

}

public static void main(String[] args) {

double amount= 10000;

double rate = 0.05;

int years = 5;

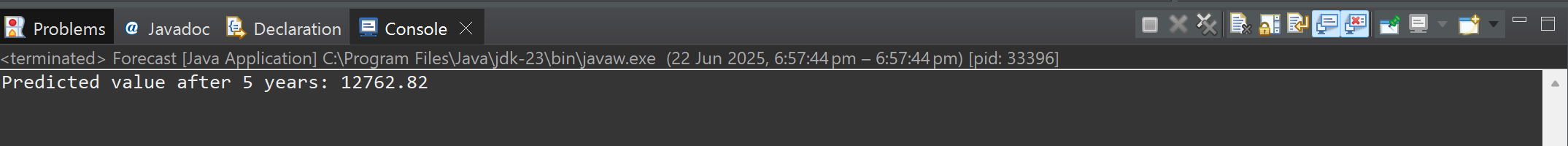
double predictedValue = FutureValue(amount, rate, years);

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

}

}

**Output:**

****

**Complexities:**

**Time Complexity:**  
The recursive method makes one call for each year.  
➤So, Time = O(n) (where n is number of years)

**Space Complexity:**  
Since recursion adds a call to the stack each time,  
➤ Space = O(n)

**Optimization Techniques:**

**1. Use Memoization**

* Store results of previous calculations in a cache (not needed here as it's linear, but useful in more complex problems like Fibonacci).

**2. Use Iterative Approach (Better for performance):**

public static double FutureValueIterative(double amount, double rate, int years) {

double result = amount;

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

result \*= (1 + rate);

}

return result;

}

This reduces space complexity to O(1).

**Conclusion:**

In this financial forecasting exercise, we used recursion to compute future values based on compound growth. While recursion works, it has higher space usage due to the call stack. An iterative approach is more efficient, offering the same result with reduced memory usage and better performance.

**-- THE END --**