# **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:**
   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.

**What is Big O Notation?**

**Big O Notation** describes how an algorithm's **runtime or space requirements grow with input size** (n). It gives an **upper bound** (worst-case scenario) and focuses on the **dominant term** for large inputs, allowing for performance prediction independent of hardware.

**Common Notations:**

* **O(1):** Constant time (e.g., array access)
* **O(**logn**):** Logarithmic time (e.g., binary search)
* **O(**n**):** Linear time (e.g., linear search)
* **O(**nlogn**):** Linearithmic time (e.g., efficient sorting)
* **O(**n2**):** Quadratic time (e.g., nested loops)

**Why Big O Notation Matters in Searching**

Big O is crucial for searching because it helps:

1. **Predict Scalability:** Understand how search performance changes with growing product catalogs (e.g., millions of items).
2. **Compare Efficiency:** Standardized way to compare different search algorithms (e.g., linear vs. binary search) theoretically.
3. **Optimize User Experience:** Faster algorithms (lower Big O) mean quicker results and better user satisfaction.
4. **Inform Design:** Guides selection of the best search algorithm based on data size and search frequency.

It's a key tool for designing performant and scalable search functionalities.

**Linear Search Time Complexity**

Linear search sequentially checks each element in the collection until the target is found or the end of the collection is reached.

* **Best Case:** O(1) - The target element is found at the very beginning of the collection (first element).
* **Average Case:** O(n) - On average, the target element is found somewhere in the middle, requiring approximately n/2 comparisons.
* **Worst Case:** O(n) - The target element is at the very end of the collection, or it's not present at all. This requires checking every single element.

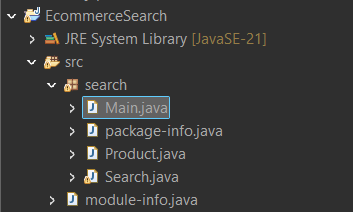
**Binary Search Time Complexity**

Binary search repeatedly divides the search interval in half. This algorithm requires the collection to be **sorted**.

* **Best Case:** O(1) - The target element is found at the middle of the collection on the first comparison.
* **Average Case:** O(logn) - The search space is halved in each step.
* **Worst Case:** O(logn) - The target element is found after the maximum number of divisions (logarithmic steps).

### Implementation:

**Folder Structure:**



**Code:**

**Product.java**

package search;

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 toString() {

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

}

}

**Search.java**

package search;

import java.util.Arrays;

import java.util.Comparator;

public class Search {

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

for (Product product : products) {

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

return product;

}

}return null; }

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

int left = 0;

int right = products.length - 1;

while (left <= right) {

int mid = left + (right - left) / 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**

import java.util.Arrays;

import java.util.Comparator;

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

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

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

System.out.println("Result: " + (foundLinear != null ? foundLinear.toString() : "Product not found"));

String notFoundTarget = "Tablet";

Product notFoundLinear = Search.linearSearch(products, notFoundTarget);

System.out.println("Result for '" + notFoundTarget + "': " + (notFoundLinear != null ? notFoundLinear.toString() : "Product not found"));

// Binary Search (Array must be sorted)

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

System.out.println("\n--- Binary Search (Array Sorted) ---");

System.out.println("Sorted Products for Binary Search:");

for (Product p : products) {

System.out.println(" " + p.getProductName());

}

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

System.out.println("Result: " + (foundBinary != null ? foundBinary.toString() : "Product not found"));

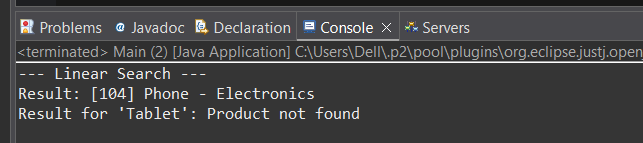
Product notFoundBinary = Search.binarySearch(products, notFoundTarget);

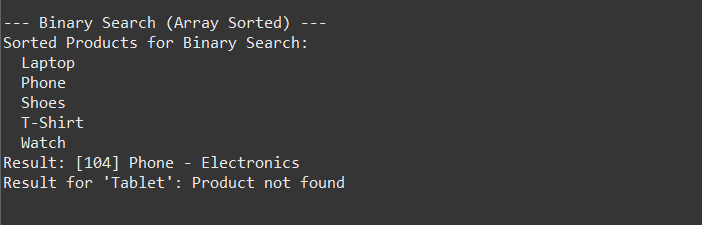
System.out.println("Result for '" + notFoundTarget + "': " + (notFoundBinary != null ? notFoundBinary.toString() : "Product not found"));

}

}

Output:





Conclusion:

For large and growing e-commerce platforms, **Binary Search is vastly superior to Linear Search**. While Linear Search is simple (O(n) worst-case), Binary Search's O(logn) efficiency (after an initial sort) ensures significantly faster search results, leading to a much smoother user experience and better system scalability. The upfront cost of sorting for Binary Search is a worthwhile trade-off for its consistent, high-speed performance on large product catalogs.

### 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.

**Recursion**

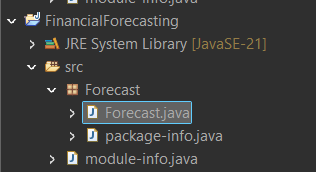
**Recursion** refers to a programming technique in which a function solves a problem by calling **itself**. It's essentially the act of breaking a large problem down into smaller, identical sub-problems until it reaches a point (a "base case") where the problem is simple enough to be solved directly without further recursion.

**Fundamental Components of a Recursive Function:**

1. **Base Case:**
   1. This is the **stopping point** for the recursion. It's the simplest instance of the problem that the function can solve directly, without making any further recursive calls.
   2. The base case is crucial to prevent infinite recursion, which would lead to a stack overflow error.
2. **Recursive Step (or Recursive Call):**
   1. In this step, the function calls itself, but this time with a **modified (usually smaller or simpler) version of the original problem**.
   2. The problem is progressively reduced in size or complexity with each recursive call, moving closer to the base case.

### Implementation:

**Folder Structure:**



**Code:**

**Forecast.java**

package Forecast;

import java.util.Scanner;

public class Forecast {

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

// Base Case: Stop recursion when years are 0.

if (years == 0) {

return amount;

} else {

// Recursive Step: Apply growth and call for remaining years.

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

}

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.*in*);

// Get user input for financial parameters

System.*out*.print("Enter the initial investment amount: ");

double amount = scanner.nextDouble();

System.*out*.print("Enter the annual growth rate (e.g., 0.05 for 5%): ");

double rate = scanner.nextDouble();

System.*out*.print("Enter the number of years: ");

int years = scanner.nextInt();

scanner.close(); // Close the scanner to prevent resource leaks

// Calculate and print the predicted future value

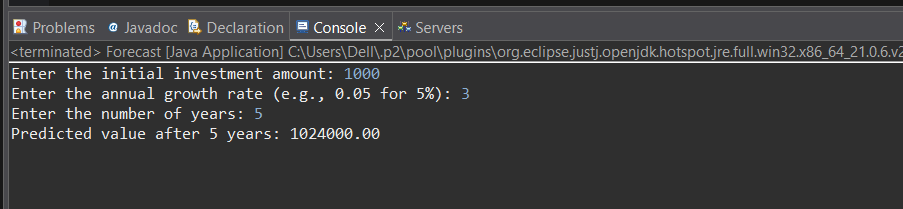
double predictedValue = *FutureValue*(amount, rate, years);

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

}

}

Output:



Conclusion:

This financial forecasting exercise effectively demonstrates the application of **recursion** to calculate future values based on compound growth. We've seen how recursion breaks down a problem into smaller, self-similar sub-problems, with a clear base case to stop the process.

While recursion offers an elegant solution and can simplify code for certain problems, its performance characteristics are important to note:

* **Time Complexity:** The recursive FutureValue method exhibits O(n) time complexity, as it makes a call for each year.
* **Space Complexity:** Due to the call stack built up during recursive calls, it also has O(n) space complexity.

For problems like future value calculation, an **iterative approach** is generally more efficient, offering the same result with a constant space complexity of O(1), making it preferable for performance-critical applications or very large years values to avoid potential stack overflow issues. This highlights that while recursion is powerful, understanding its implications for resource usage is crucial for choosing the optimal algorithmic approach.