**Exercise 1: E-commerce Platform Search Function**

1. **Understand Asymptotic Notation :**

**What is Big O Notation?**

Big O notation describes the **upper bound** of an algorithm's time or space complexity in terms of input size n.

|  |  |  |
| --- | --- | --- |
| **Notation** | **Meaning** | **Example** |
| O(1) | Constant Time | Accessing an element in an array |
| O(n) | Linear Time | Linear search |
| O(log n) | Logarithmic Time | Binary search |
| O(n log n) | Log-linear Time | Merge Sort |
| O(n²) | Quadratic Time | Nested loops |

**Best, Average, and Worst-Case for Search:**

|  |  |  |  |
| --- | --- | --- | --- |
| **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 |

1. **Setup: Product Class:**

**Product.java:**

package Ecommerce;

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;

}

@Override

public String toString() {

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

}

}

1. **Implementation:**

package Ecommerce;

import java.util.Arrays;

import java.util.Comparator;

public class SearchUtil {

**//Linear Search:**

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

for (Product p : products) {

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

return p;

}

}

return null;

}

**//Binary Search:**

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

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

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

while (left <= right) {

int mid = (left + right) / 2;

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

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

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

else right = mid - 1;

}

return null;

}

}

1. **Analysis:**

|  |  |  |
| --- | --- | --- |
| **Algorithm** | **Time Complexity** | **Space Complexity** |
| **Linear Search** | O(n) | O(1) |
| **Binary Search** | O(log n) | O(1), but needs sorted array |

**Suitable Algorithm:**

|  |  |  |
| --- | --- | --- |
| **Criteria** | **Linear Search** | **Binary Search** |
| **No sorting needed?** | Yes | Required Sorting |
| **For large data sets?** | Not Suitable | Suitable |
| **Fast repeated lookups?** | Not Suitable | Suitable |
| **Dynamic product list (frequent insert/delete)?** | Can be done easily | sorting must be maintained |

**Recommendation:**

* For small or rarely searched lists: use Linear Search.
* For large, sorted product lists with frequent searches: use Binary Search.

**Main.java:**

package Ecommerce;

public class Main {

public static void main(String[] args) {

Product[] products = {

new Product(101, "Shoes", "Fashion"),

new Product(102, "Laptop", "Electronics"),

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

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

};

String searchItem = "Watch";

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

Product foundLinear = SearchUtil.linearSearch(products, searchItem);

System.out.println(foundLinear != null ? foundLinear : "Not Found");

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

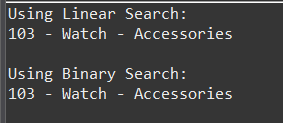
Product foundBinary = SearchUtil.binarySearch(products, searchItem);

System.out.println(foundBinary != null ? foundBinary : "Not Found");

}

}

**Output:**

****

**Exercise 2: 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.

**Exercise:**

**Recursion** is a programming technique where a function **calls itself** in order to solve a problem.

Every recursive function must have:

1. **Base Case**: The condition under which the function stops calling itself.
2. **Recursive Case**: The part where the function calls itself to break the problem into smaller subproblems.

**How Recursion Works:**

When a recursive function is called, it keeps calling itself with smaller inputs until the base case is reached. Once the base case is reached, the function stops calling itself and starts returning values back up the call stack.

**Simplifying Problems with Recursion**

Recursion simplifies complex problems by **breaking them down into simpler subproblems** of the same type.

Example:

int factorial(int n) {

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

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

}

Recursion helps simplify problems like tree traversal, factorials, and here — forecasting based on past growth.

**Step 2: Setup**

**Problem:**

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)

**Step 3: Implementation :**

public class FinancialForecast {

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

if (years == 0) {

return amount;

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

}

public static void main(String[] args) {

double initialAmount = 1000.0;

double growthRate = 0.10;

int years = 5;

double result = futureValue(initialAmount, growthRate, years);

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

}

}

**OUTPUT:**

****

**Step 4: Analysis**

**Time Complexity of Recursive Algorithm:**

* The function calls itself **once** per year, so:
* **Time Complexity = O(n)** where n = number of years.

**Drawbacks:**

* Recursive calls consume **stack memory**.
* For large values (e.g., 10,000 years), it may cause **StackOverflowError**.