**Chirag H M-6373750(superset id)**

**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:**

**What is Big O Notation?**

* **Big O Notation** describes the **upper bound of an algorithm's running time** as the input size grows.
* It helps estimate how scalable and efficient an algorithm is.

| **Notation** | **Meaning** | **Example** |
| --- | --- | --- |
| O(1) | Constant time | Accessing an array element |
| O(n) | Linear time | Linear search |
| O(log n) | Logarithmic time | Binary search |
| O(n²) | Quadratic time | Bubble sort |

**Best, Average, and Worst Cases**

| **Scenario** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| **Best Case** | O(1) – First element match | O(1) – Middle element match |
| **Average Case** | O(n/2) → O(n) | O(log n) |
| **Worst Case** | O(n) – Last or not found | O(log n) |

Binary search is **much faster**, but only works on **sorted arrays**.

Product.java

package searchFunction;

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 + ")";

}

}

ProductSearch.java

package searchFunction;

import java.util.Arrays;

import java.util.Comparator;

public class ProductSearch {

// Linear Search

public static Product linearSearch(Product[] products, int targetId) {

for (Product product : products) {

if (product.productId == targetId) {

return product;

}

}

return null;

}

// Binary Search

public static Product binarySearch(Product[] products, int targetId) {

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

while (left <= right) {

int mid = left + (right - left) / 2;

if (products[mid].productId == targetId) {

return products[mid];

} else if (products[mid].productId < targetId) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

// Helper: Sort products by productId for binary search

public static void sortByProductId(Product[] products) {

Arrays.*sort*(products, Comparator.*comparingInt*(p -> p.productId));

}

}

Main.java

package searchFunction;

public class Main {

public static void main(String[] args) {

Product[] products = {

new Product(105, "Laptop", "Electronics"),

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

new Product(110, "Mobile", "Electronics"),

new Product(103, "Book", "Education"),

};

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

Product result1 = ProductSearch.*linearSearch*(products, 110);

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

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

ProductSearch.*sortByProductId*(products); // Sort before binary search

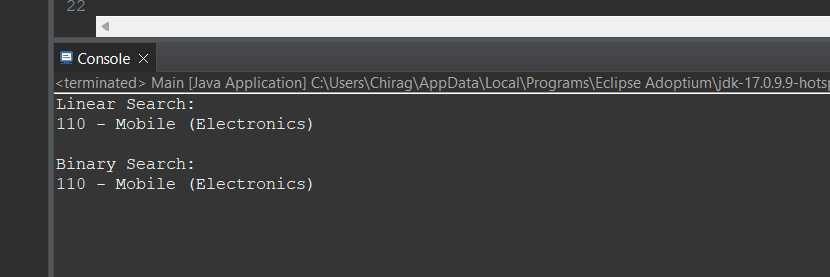
Product result2 = ProductSearch.*binarySearch*(products, 110);

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

}

}

Output:



**Time Complexity**

| **Algorithm** | **Time Complexity** | **Sorted Needed** |
| --- | --- | --- |
| Linear Search | O(n) | No |
| Binary Search | O(log n) | Yes |

**Which One to Use?**

* Use **Linear Search**:
  + When the list is small
  + When data is **unsorted** and one-time or rare search is enough
* Use **Binary Search**:
  + When the list is **large** and frequently searched
  + Sorting once is acceptable for improved speed

For an **e-commerce platform**, where users search frequently, **binary search on sorted data or use of indexing structures** (like hash maps or trees) is **more efficient**.

**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:

**Recursion** is a method where a function **calls itself** to solve smaller instances of a problem.

**Example:**

int factorial(int n) {

if (n == 0) return 1;

return n \* factorial(n - 1);

}

**When is recursion useful?**

* When a problem can be broken into **smaller identical subproblems**
* Examples: factorials, Fibonacci, tree traversals, forecasting

To calculate future value:

FV=PV×(1+r)nFV = PV \times (1 + r)^nFV=PV×(1+r)n

Where:

* **FV** = Future Value
* **PV** = Present Value
* **r** = Growth rate (e.g., 0.05 for 5%)
* **n** = Number of periods (years, months, etc.)

FinancialForecast.java

package financialForecast;

public class FinancialForecast {

// Recursive method to calculate future value

public static double futureValue(double presentValue, double rate, int periods) {

if (periods == 0) {

return presentValue;

}

return (1 + rate) \* *futureValue*(presentValue, rate, periods - 1);

}

}

Main.java

package financialForecast;

public class Main {

public static void main(String[] args) {

double presentValue = 10000.0; // Starting amount

double growthRate = 0.05; // 5% growth

int years = 5;

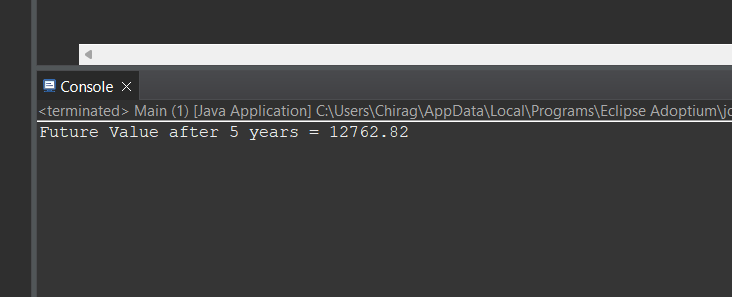
double result = FinancialForecast.*futureValue*(presentValue, growthRate, years);

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

}

}

Output:



Time Complexity:

futureValue(pv, r, n) → O(n)

For larger n, recursion may be inefficient due to:

* **Stack overflow**
* **Repeated computations**

### Iterative Alternative (more efficient):

public static double futureValueIterative(double presentValue, double rate, int periods) {

double result = presentValue;

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

result \*= (1 + rate);

}

return result;

}

### Time complexity is still **O(n)**, but **memory-efficient** (no recursion stack)