**Data Structure and Algorithms**

**Exercise 2: E-commerce Platform Search Function**

**Big – O Notation:**

**Big O notation**is a powerful tool used in computer science to describe the time complexity or space complexity of algorithms. **Big-O** is a way to express the **upper bound**of an algorithm’s time or space complexity.

* Describes the asymptotic behavior (order of growth of time or space in terms of input size) of a function, not its exact value.
* Can be used to compare the efficiency of different algorithms or data structures.
* ****It provides an**upper limit** on the time taken by an algorithm in terms of the size of the input. We mainly consider the worst case scenario of the algorithm to find its time complexity in terms of Big O

**Product.java:**

**package** ecommerceSearch;

**import** java.util.\*;

**public** **class** Product {

**int** productId;

String productName;

String category;

**public** Product() {}

**public** Product(**int** productId, String productName, String Category) {

**this**.productId= productId;

**this**.productName= productName;

**this**.category = Category;

}

**public** String linearSearch(Product[] products, **int** id) {

**for**(**int** i=0;i<products.length;i++) {

**if**(products[i].productId == id) {

**return** "Product is found: "+ products[i].productName + "(" + products[i].category + ")";

}

}

**return** "Product Not Found";

}

**public** String binarySearch(Product[] products, **int** id) {

**int** low=0;

**int** high=products.length -1;

**while**(low<= high) {

**int** mid = low + (high - low)/2;

**if**(products[mid].productId == id) {

**return** "Product is found: " + products[mid].productName + "(" + products[mid].category + ")";

}

**else** **if**(products[mid].productId<id) {

low = mid+1;

}

**else** {

high= mid-1;

}

}

**return** "Product Not Found";

}

**public** **static** **void** main(String[] args) {

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

// Predefined list of products

Product[] products = {

**new** Product(1, "iPhone", "Smartphones"),

**new** Product(2, "Samsung", "Smartphones"),

**new** Product(3, "Notes", "Stationery"),

**new** Product(4, "Laptop", "Electronics"),

**new** Product(5, "Pen", "Stationery"),

**new** Product(6, "Redmi", "Smartphones")

};

Product[] sortedProducts = products.clone();

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

System.***out***.print("Enter a product ID to search: ");

**int** id = scn.nextInt();

scn.nextLine();

System.***out***.print("Enter the type of search (Linear/Binary): ");

String searchType = scn.nextLine();

// Create object

Product obj = **new** Product();

String result;

**if** (searchType.equalsIgnoreCase("linear")) {

result = obj.linearSearch(products, id);

} **else** **if** (searchType.equalsIgnoreCase("binary")) {

result = obj.binarySearch(sortedProducts, id);

} **else** {

result = "⚠️ Invalid search type. Please enter 'Linear' or 'Binary'.";

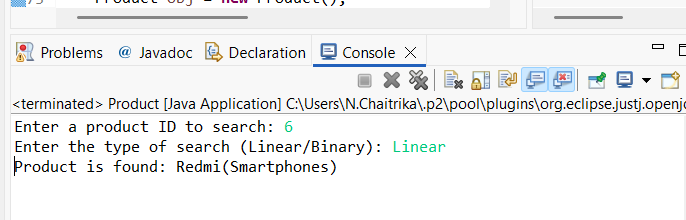
}

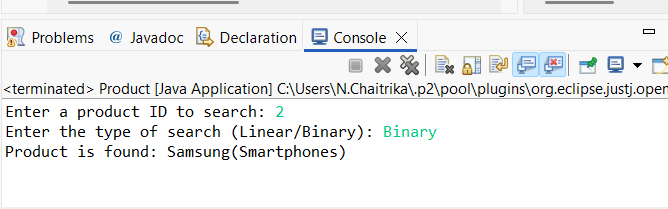
System.***out***.println(result);

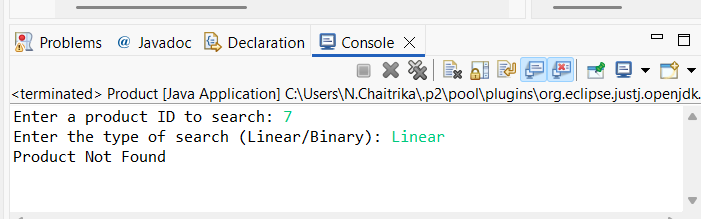
}

}

**Output:**

****

****

****

**Analysis:**

**Linear Search:**

The time complexity of linear search varies based on the position of the target element.

Best Case: O(1)

If the element is at the first Index

Worst Case: O(n)

The element is either at the very end or not present at all

**Binary Search:**

Binary search is much more efficient but requires the data to be **sorted**.

Best Case: O(1)

If the target matches the middle element on the first comparison

Worst Case: O(log n)

**Which Search is More Suitable?**

**Linear search** is more suitable when the dataset is **small** or **unsorted**, as it doesn’t require any preprocessing or sorting of the data. It’s also easier to implement and works well for short lists.

**Binary search**, on the other hand, is ideal when dealing with **large datasets** that are already **sorted**. It significantly reduces the number of comparisons and provides faster performance in such scenarios.

**Exercise 7: Financial Forecast**

**Recursion:**

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function.

It works by breaking down a complex problem into smaller, similar subproblems until a simple base case is reached, which can be solved directly.

The results of these smaller problems are then combined to solve the original problem.

**FinancialForecast.java:**

**package** FutureValuePrediction;

**import** java.util.Scanner;

**public** **class** FinancialForecast {

**public** **static** **double** futureValue(**double** pv, **double** rate, **int** n) {

**if** (n == 0) {

**return** pv;

}

**return** (1 + rate) \* *futureValue*(pv, rate, n - 1);

}

**public** **static** **void** main(String[] args) {

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

System.***out***.print("Enter Present Value (PV): ");

**double** pv = sc.nextDouble();

System.***out***.print("Enter annual growth rate (in %): ");

**double** rate = sc.nextDouble() / 100;

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

**int** n = sc.nextInt();

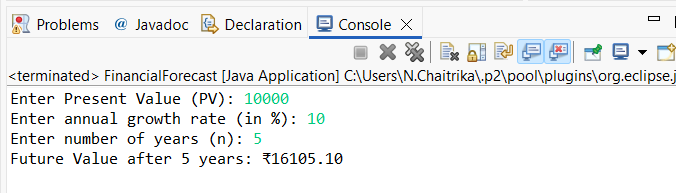
**double** fv = *futureValue*(pv, rate, n);

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

}

}

**Output:**

****

**Time Complexity:**

The function calls itself once per year, so:

Time Complexity: O(n) where n is the number of years.

**Space Complexity:**

Due to the recursive call stack: O(n) space is used.

**Optimization:**

If it had overlapping subproblems, you could use Memoization (Storing previous values)

Convert to iterative

public static double futureValueIter(double pv, double rate, int n) {

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

pv \*= (1 + rate);

}

return pv;

}

Time Complexity: O(n)

Space Complexity: O(1)